Decentralized On-line Task Reallocation on Parallel Computing Architectures with Safety-Critical Applications

Thanakorn Khamvilai  
School of Aerospace Engineering  
Georgia Institute of Technology  
Atlanta, GA, USA  
tkhamvilai3@gatech.edu

Philippe Baufreton  
Safran Electronics & Defense  
Massy, France  
philippe.baufreton@safrangroup.com

Louis Sutter  
School of Aerospace Engineering  
Georgia Institute of Technology  
Atlanta, GA, USA  
lutter6@gatech.edu

François Neumann  
Safran Electronics & Defense  
Massy, France  
francois.neumann@safrangroup.com

Eric Feron  
School of Aerospace Engineering  
Georgia Institute of Technology  
Atlanta, GA, USA  
eric.feron@aerospace.gatech.edu

Abstract—This work presents a decentralized allocation algorithm of safety-critical application on parallel computing architectures, where individual Computational Units can be affected by faults.

The described method consists in representing the architecture by an abstract graph where each node represents a Computational Unit. Applications are also represented by the graph of Computational Units they require for execution. The problem is then to decide how to allocate Computational Units to applications to guarantee execution of the safety-critical application. The problem is formulated as an optimization problem, with the form of an Integer Linear Program. A state-of-the-art solver is then used to solve the problem.

Decentralizing the allocation process is achieved through redundancy of the allocator executed on the architecture. No centralized element decides on the allocation of the entire architecture, thus improving the reliability of the system.

Experimental reproduction of a multi-core architecture is also presented. It is used to demonstrate the capabilities of the proposed allocation process to maintain the operation of a physical system in a decentralized way while individual component fails.

Index Terms—parallel computing, multi-core, reconfigurable, safety-critical, fault tolerance, decentralized, integer linear programming

I. INTRODUCTION AND PRIOR ART

The onset of multi-core processors appeared as a golden opportunity for the embedded systems industry to improve efficiency of embedded computers. Multicore processors carry several benefits over single core ones, bringing more computational power through parallelization without increasing chip’s internal frequency, and without increased energy consumption or increased heating. They now pervade cellular communication devices and embedded electronics for mass-market, for example, and many other industries are now taking advantage of such processors, such as the automotive industry [1], the biotechnology industry [2] and the circuit industry [3]. However, as far as critical systems are concerned, these benefits come with great certification challenges [4] [5], since parallel applications on a multi-core processor may interfere. The aerospace industry is yet undertaking to take up this challenge [6].

A reconfigurable multi-core architecture that could host safety-critical applications, e.g. [7], [8], [9], can become an example of a safe multi-core processor by taking advantage of the inherent redundancy of such processors that enables graceful degradation [10]: when some core fails, we can use the multiple remaining ones by reallocating affected applications to a healthy area of the chip.

The inherent redundancy in such parallel architecture can also be seen as an opportunity to increase the reliability of computing systems, be it in safety critical embedded systems or for computing centers requiring guaranties of continuity of service.

For example, several attempts have been made to increase the reliability of safety-critical systems using multi-core processors. In [11], an “hypervisor” is used to organize access to shared resources for applications, including safety-critical ones. However, a failure of this hypervisor is not taken into account in this patent. Therefore, such technique just moves the problem since the whole reliability is carried by the reallocation decision organ, which constitutes a single point of failure: the most complex and efficient reallocator is pointless if the system it executes on fails. In [12], backup allocations are pre-calculated for each failure case and they are stored by individual Computational Units (CUs). For

This effort has been funded in part by SAFRAN and by the National Science Foundation, Grants CNS 1544332 and 1446758.
small architecture with only a few CUs, this solution is satisfactory and ensures a continuous fault tolerance of the system without requiring a centralized allocator. However, storing backup configuration can require a lot of memory when the architecture becomes bigger. Also, the proposed approach does not consider application that can themselves be parallelized and executed on several CUs at the same time.

Our approach differs from these two solutions by providing an on-line and decentralized reallocation algorithm for a general architecture that can be represented by a graph and for parallelized applications requiring several CUs to execute.

Even though this work is motivated by a multi-core architecture, it presents a decentralized task allocation algorithm for an abstract parallel computing architecture made of a set of CUs connected together and forming a network. Such an architecture can represent for example a multi-core processor, with each CU standing for one core, a cluster of high-performance computers, or a team of mobile robots. The aim of the algorithm is to find the optimal allocation of an a priori defined set of tasks on the architecture while taking into account the faults affecting the CUs. The faults are assumed to be detected by the algorithm when they occur either via a timeout mechanism or a voter, but this work does not provide details of those fault detection mechanisms. As described later, two types of fault will be considered, the first one completely stopping the operation of the CU, and a second one considered to modify the computed output of the CU.

The second main feature of this work is the decentralized aspect of the allocation process. Decentralized means here that there is no central element deciding alone of the allocation for the rest of the architecture. Instead, we use redundant copies of the allocation algorithm executed on the architecture itself, meaning that the copies must reallocate themselves. This is achieved by using majority voting systems.

This work also presents an experimental setup reproducing several aspects of a parallel computing architecture and used to implement the proposed decentralized allocation algorithm. The setup uses a network of Raspberry Pi single board computers [18] to represent the CUs of the architecture.

II. THEORETICAL ASPECT

A. Mathematical description of the allocation problem

This section describes the mathematical formulation of the general allocation problem that is considered in this work. The idea is to use this mathematical formulation in an Integer Linear Program (ILP), whose solution is the best allocation of the tasks on the parallel computing platform (multi-core processors, network of computers in a computing center, etc), according to criteria described in Section III-C2, taking into account the number of applications running, their priority, the number of reallocated applications and the length of communication paths between allocators and other applications.

The considered parallel computing platform is represented by a directed simple graph $G = (V, E)$, where $V$ is the set of vertices and $E \subseteq \{(x, y) \in V^2 \mid x \neq y\}$ is the set of edges [13]. Each vertex of $G$ represents a CU, for example one core in a multi-core processor or at a different scale, one computer in a massively parallel supercomputer, and each edge of $G$ represents a physical communication link between two CUs. The communication links are considered bidirectional, and therefore the orientation of edges can be chosen arbitrarily: we choose them to be oriented only to write more conveniently further constraints on the communication flow.

The graph $G$ therefore represents the topology of the platform. For example, the platform can have a simple square mesh topology, as represented in Fig. 1.

![Figure 1: Example of square mesh topology. Orientation of edges are arbitrary.](image)

From $G$, we define parameters that will be used later in this work.

**Definition 1.** $N_CUs$ is defined as the number of CUs in the computing platform, that is the number of vertices of $G$.

**Definition 2.** $N_{paths}$ is defined as the number of Physical Communication Links, or physical paths, in the platform, that is the number of edges of $G$.

Let $N_{app} \in \mathbb{N}$ and $\mathcal{A} = \{\text{app}_k, \ k \in [1, N_{app}]\}$ be a set of applications to be executed on the parallel computing platform. The applications in $\mathcal{A}$ are ranked by priority, $\text{app}_1$ having the highest priority and $\text{app}_{N_{app}}$ having the lowest one. The ranking is established $a priori$ and represents the tolerated order in which we stop applications in case of computing resource failures. In the context of a commercial aircraft, an example of such applications with different priority would be the engine controller, with the highest priority, and a health monitoring application, with a lower priority, which is in charge of analyzing data from the engine in order to estimate its wear and to predict when maintenance operations are required. In case of computing resource failures, it would
be tolerated in this context to stop the health monitoring application in order to maintain the execution of the engine controller.

For $k \in [1, N_{app}]$, we assume that the compiler for the considered architecture decomposes the application $app_k$ into a undirected simple graph $G_k = (V_k, E_k)$, where each vertex, that we will call Application Node, represents a sub-task of $app_k$ that must be executed by a CU, and each edge represents a required communication link between two Application Nodes, that we will call an Application Link. Fig. 2 gives an example of such application graphs.

![Figure 2: Example of application graphs. Each application node is identified with a unique index. app1 has highest priority, app3 has the lowest.](image)

From each graph $G_k$ for $k \in [1, N_{app}]$, we define the following parameters.

**Definition 3.** $N_{nodes}^k$ is defined as the number of Application Nodes in application $k$ and $N_{links}^k$ is defined as the number of Application Links in application $k$.

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

Each application node is given a global index $j \in [1, N_{nodes}]$ with the following procedure: the nodes of $app_1$ keep the same indices as in the local numbering of vertices in $G_1$; then the global indices for nodes of $app_2$ are obtained by increasing their local indices by $N_{nodes}^1$; and so on for the nodes of $app_k$, by increasing the local numbering by $\sum_{i=1}^{k-1} N_{nodes}^i$. The result of the global numbering of the nodes can be seen on Fig. 2. An identical process is applied to obtain a global numbering of the edges of the application graphs.

The problem that we tackle here is to assign applications to CUs of the architecture while faults affect some CUs, taking into account the priority of the applications and specific constraints of the architecture. A solution will look like Fig. 3. The approach that we take here to solve the problem is to formulate the allocation problem as Integer Linear Program (ILP) and use a state-of-the-art IP solver such as “GNU Linear Programming Kit” (GLPK) [14].

![Figure 3: Example of a solution with a fault on CU 11.](image)

An additional aspect of the problem that we propose to solve is to make the allocation process decentralized, in the sense detailed in the introduction and in Section IV with no central computing element allocating the tasks according to the solution of the ILP problem. The way this decentralized allocation is achieved is specifically described in Section IV, it involves several copies of the task computing the allocation and being executed on the platform itself. The number of such copies is the last parameter of our problem.

**Definition 5.** $N_{realloc}$ is defined as the number of copies of the Allocator Application.

The next section details how the allocation problem is formulated as an ILP problem.

### III. ILP FORMULATION OF THE TASK ALLOCATION PROBLEM

#### A. Matrix representation of graphs

**Definition 6.** From the graph representation $G = (V, E)$ of the parallel computing platform, the $N_{CUs} \times N_{paths}$ incidence matrix $G$ associated with $G$ is defined as:

$$[G]_{ij} := \begin{cases} -1 & \text{if } e_j \in E \text{ leaves } v_i \in V \\ 1 & \text{if } e_j \in E \text{ enters } v_i \in V \\ 0 & \text{otherwise} \end{cases}$$

And the $N_{CUs} \times N_{paths}$ NoC unoriented incidence matrix $\hat{G}$ associated with $G$ is defined as:

$$[\hat{G}]_{ij} := ||[G]_{ij}||$$

**Definition 7.** From the graph $G_k = (V_k, E_k)$ representing the $k$-th application, the $N_{nodes}^k \times N_{links}^k$ application unoriented incidence matrix $H^k$ associated with $G_k$ is defined as:

$$[H^k]_{ij} := \begin{cases} 1 & \text{if } v_i^k \in V_k \text{ and } e_j^k \in E_k \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

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

$$H := \begin{bmatrix} H^1 & \cdots & \vdots \\ \vdots & \ddots & \vdots \\ \vdots & \vdots & H^k \end{bmatrix}$$
B. Definition of the Decision Variables

**Definition 8.** The $N_{\text{CUs}} \times N_{\text{nodes}}$ decision matrix $X_{\text{CUs} \rightarrow \text{nodes}}$, mapping Application Nodes to CUs, is defined as:

$$X_{ij}^{\text{CUs} \rightarrow \text{nodes}} = \begin{cases} 1 & \text{if the CU } i \text{ is allocated to the Application Node } j \\ 0 & \text{otherwise} \end{cases}$$

**Definition 9.** The $N_{\text{paths}} \times N_{\text{links}}$ decision matrix $X_{\text{paths} \rightarrow \text{links}}$, mapping Application Links to Physical Links, is defined as:

$$X_{ij}^{\text{paths} \rightarrow \text{links}} = \begin{cases} 1 & \text{if the Physical Link } i \text{ is allocated to the Application Link } j \\ 0 & \text{otherwise} \end{cases}$$

**Definition 10.** The $N_{\text{apps}} \times 1$ 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 $N_{\text{nodes}} \times 1$ decision vector $M$, representing which application nodes are reallocated, is defined as:

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

**Definition 12.** For $k \in [1, N_{\text{apps}}]$, the $N_{\text{paths}} \times N_{\text{CUs}}$ decision matrix $X^{\text{Comm}, k}$, representing communication paths between the $k$-th allocator application and every CU of the platform, is defined as:

$$X_{ij}^{\text{Comm}, k} = \begin{cases} -1 & \text{if the Physical Link } i \text{ is used} \\ 1 & \text{if the Physical Link } i \text{ is used} \\ 0 & \text{otherwise} \end{cases} \text{ to communicate between the allocator } k \text{ and the CU } j \text{ in the negative direction}$$

Positive (respectively negative) direction means that the communication takes place in the same (respectively opposite) direction as the edge of the directed graph $\mathcal{G}$, as in Fig. [1].

C. Formulation of the optimization model

This section gives the detail of the formulation of the optimization problem that is solved each time a new fault is detected. This formulation includes the detail of the chosen objective function and constraints.

1) General form of the optimization model: The allocation problem is formulated as an Integer Linear Program (ILP) of the form [15]:

$$\begin{align*}
\text{maximize} & \quad f(x) = c^T x \\
\text{subject to} & \quad M_1 x \leq b_1 \\
& \quad M_2 x = b_2 \\
& \quad x \in \mathbb{Z}^n \\
\end{align*}$$

where $x$ is the global vector of decision variables derived from the vectorization and the aggregation of the decision matrices defined in Section III-B. We define it formally as:

$$x = \begin{pmatrix} \text{vec}(X_{\text{CUs} \rightarrow \text{nodes}}) \\
\text{vec}(X_{\text{paths} \rightarrow \text{links}}) \\
\text{vec}(X_{\text{Comm}, 1}) \\
\vdots \\
\text{vec}(X_{\text{Comm}, N_{\text{realloc}}}) \end{pmatrix}$$

$c$ is the coefficients of the objective function and $M_1$, $M_2$, $b_1$ and $b_2$ are parameters derived from the aggregation of the constraints of the problem that are described in the following sections. For example, for each scalar inequality constraint, after arranging the inequality with all decision variables on the left-hand side in the same order as in $x$ and constant terms on the right-hand side, a row containing the coefficients of the decision variables is added to $M_1$ and the constant term is added in the vector $b_1$. The same is done for equality constraints to build $M_2$ and $b_2$.

2) Objective function: Given the priority of the applications in an ascending order i.e. the first application has the highest priority and the $N_{\text{apps}}$-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 and the length of communication paths. The chosen objective function, in terms of $x$ as defined above in equation [2] is:

$$\begin{align*}
\max \left\{ f(x) = \sum_{k=1}^{N_{\text{apps}}} \alpha_k \cdot r_k - (\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} M_j \\
- \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CUs}}} \sum_{i=1}^{N_{\text{paths}}} \left| X_{ij}^{\text{Comm}, k} \right| \right\}.
\end{align*}$$

(3)
where
\[
\beta = N_{\text{realoc}} \times N_{\text{CUs}} \times N_{\text{paths}}
\]
\[
\alpha_{\text{apps}} = (\beta + 1) \times N_{\text{nodes}} + \beta + 1
\]
and \( \forall k < N_{\text{apps}} : \)
\[
\alpha_k = \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l + (\beta + 1) \times N_{\text{nodes}} + \beta + 1.
\]
(4)

The coefficients of the objective function are chosen to prioritize the different aspects that are optimized in this function.

1) The first priority is to execute each application, even if it means more reallocations and longer communication paths.

2) Then, minimizing the number of reallocations is more important than having shorter communication paths, since a reallocation temporarily interrupts the execution of the allocation.

3) When running all applications is not feasible, the priorities of the applications are enforced and executing any given application is more important than running any number of applications with a lower priority. However, if because of its geometry, a given application cannot be executed anyway, nothing prevents lower-priority applications from being executed.

These requirements motivated the choice for the coefficients in the objective function. The proof that these coefficients allow the objective function to meet these requirements is given in Appendix.

Note that the problem of minimizing or maximizing the absolute value of the \( X_{ij}^{\text{Comm},k} \) variables, which is a nonlinear program, can be reformulated as a linear program by introducing additional variables and constraints [15], that were not presented in the previous section for conciseness. For each entry \( X_{ij}^{\text{Comm},k} \) of \( X_{ij}^{\text{Comm}} \), an auxiliary variable \( X_{ij}^{\text{Comm},k} \) is introduced to represent its absolute value, and two extra constraints are added:
\[
+X_{ij}^{\text{Comm},k} \leq X_{ij}^{\text{Comm},k},
\]
\[
-X_{ij}^{\text{Comm},k} \leq X_{ij}^{\text{Comm},k}.
\]
\( X_{ij}^{\text{Comm},k} \) is then used instead of \( |X_{ij}^{\text{Comm},k}| \) in the objective function. Because the objective function tends to maximize \(-|X_{ij}^{\text{Comm},k}|\), so to minimize \( X_{ij}^{\text{Comm},k} \), one of the two previous constraints will be binding, the stricter one, where the left-hand side is the greatest and equal to \( \max(+X_{ij}^{\text{Comm},k}, -X_{ij}^{\text{Comm},k}) \), which is exactly \( |X_{ij}^{\text{Comm},k}| \). The other constraint will be non-binding and therefore does not affect the optimal point. It thus ensures that \( X_{ij}^{\text{Comm},k} \) is equal to \( |X_{ij}^{\text{Comm},k}| \).

3) Constraints:

a) Domain of decision variables: The decision variables \( X_{ij}^{\text{CUs} \rightarrow \text{nodes}}, X_{ij}^{\text{paths} \rightarrow \text{links}} \) and \( r \) and \( M \) are binary, i.e. the value of their entries must be either 0 or 1.

The entries of \( X_{ij}^{\text{Comm},k} \) for \( k \in [1, N_{\text{realoc}}] \) must belong to \( \{-1, 0, 1\} \).

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

- Each CU can be allocated to at most one application, as a way to enforce spatial partitioning of applications on the platform, i.e.
\[
\forall i \in [1, N_{\text{CUs}}], \sum_{j=1}^{N_{\text{nodes}}} X_{ij}^{\text{CUs} \rightarrow \text{nodes}} \leq 1. \quad (5)
\]

- Each running Application Node must be assigned to exactly one CU, i.e.
\[
\forall i \in [1, N_{\text{nodes}}], \sum_{j=1}^{N_{\text{CUs}}} X_{ij}^{\text{CUs} \rightarrow \text{nodes}} = rN(i). \quad (6)
\]

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

- A physical communication link of the platform can be allocated to at most one Application Link \( r \) i.e.
\[
\forall i \in [1, N_{\text{paths}}], \sum_{j=1}^{N_{\text{links}}} X_{ij}^{\text{paths} \rightarrow \text{links}} \leq 1. \quad (7)
\]

- Each running Application Link must be assigned to exactly one physical communication link of the platform, i.e.
\[
\forall i \in [1, N_{\text{paths}}], \sum_{j=1}^{N_{\text{links}}} X_{ij}^{\text{paths} \rightarrow \text{links}} = rL(i). \quad (8)
\]

\( L(i) \) is the application number corresponding to Application Link \( i \).

c) Compliance with the platform: An Application link, adjacent to an Application Node that has been mapped to a given CU, must be allocated to a Physical Link that is adjacent to that CU, i.e.
\[
X_{ij}^{\text{CUs} \rightarrow \text{nodes}} H = \tilde{G} X_{ij}^{\text{paths} \rightarrow \text{links}}. \quad (9)
\]

1 This does not mean that this communication link cannot be used for other communication purposes on the architecture, but only one of the Application Link computed by the compiler for the applications can be allocated to that physical communication link.
This equation [9] is equivalent to the scalar equations (III-C3):
\[
\forall i \in [1, N_{\text{CUs}}], \forall j \in [1, N_{\text{nodes}}], \quad \sum_{k=1}^{N_{\text{nodes}}} X_{ik}^{\text{CUs} \rightarrow \text{nodes}} H_{kj} = \sum_{l=1}^{N_{\text{links}}} G_{il} X_{lj}^{\text{paths} \rightarrow \text{links}}. \tag{10}
\]

The left-hand side is equal to one if and only if the CU \(i\) has been allocated to Application Node \(k\) and Application Node \(k\) is adjacent to Application Link \(j\). The right-hand side is equal to one if and only if the CU \(i\) is adjacent to the Physical Link \(l\) and the Physical Link \(l\) is allocated to Application Link \(j\), which proves the correctness of the constraint.

d) Reallocating several applications: A given Application Node can either remain affected to the same CU, either be moved or be dropped:
\[
\forall i \in [1, N_{\text{CUs}}], \forall j \in [1, N_{\text{nodes}}], \text{ s.t. } X_{\text{old}_{ij}}^{\text{CUs} \rightarrow \text{nodes}} = 1, \\
(1 - r_N(j)) + M_j + X_{\text{old}_{ij}}^{\text{CUs} \rightarrow \text{nodes}} = X_{ij}^{\text{CUs} \rightarrow \text{nodes}}, \tag{11}
\]

with \(X_{\text{old}}^{\text{CUs} \rightarrow \text{apps}}\) be the parameter containing the mapping between CUs and Application Nodes computed during the previous allocation.

This constraint (11) is ignored for the initial allocation.

e) Faults: We assume the parallel computing platform is equipped a fault detection system that can detect and inform the allocators when a CU fails. From this information, we can add constraints to take into accounts fault in the platform. Within a CU \(i\):

- If the CU is healthy, any Application Node can be mapped on the CU.
- If the CU is faulty, then no Application Nodes can be mapped on the CU:
\[
\sum_{k=1}^{N_{\text{nodes}}} X_{ik}^{\text{CUs} \rightarrow \text{apps}} = 0. \tag{12}
\]

The detection of this fault is either assumed for the model or detected by the voter using the majority rule described in [V-A2].

f) Communication constraints: Since the allocators are executed on the platform, we must ensure that they will be able to send the allocation they computed to the other CUs of the platform, given the communication links that allows each CU to send a message only through its neighbors. Therefore, we must make sure that there exists a path from each allocator to the other CUs.
\[
\forall k \in [1, N_{\text{reallo}}], G X^{\text{Comm}, k} = S^k \tag{13}
\]

where \(S^k\) is the \(N_{\text{CUs}} \times N_{\text{CUs}}\) source-sink matrix \(S^k\), depending on \(X^{\text{CUs} \rightarrow \text{nodes}}\) and defined by:

\[
[S^k]_{ij} := \begin{cases} 
0 & \text{if } \deg(v_i) = 0 \text{ in } G \\
0 & \text{if } \text{CU } i \text{ is faulty} \\
-1 & \text{otherwise}
\end{cases} \]

where the degree of a vertex \(\deg(v_i)\) is the number of edges connected to it, and node_of_alloc(k) is the Application Node corresponding to allocator \(k\). When the CU is neither faulty nor without any neighbor, in each path between an allocator and a given CU \(i\), the allocator is the source (-1) and the CU \(i\) is the sink (+1).

g) Constraints specific to the architecture: Additional constraints can be added to respect specific aspects of the considered architecture.

For example, some multi-core architectures [7], where intra-application communication between CU can happen only in a specific way as illustrated in Fig. [4] orientation of the applications on the architecture matters because nodes that can communicate in a given orientation will not be able to do so if they are rotated on the architecture. Therefore the orientation as computed by the compiler must be enforced.

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

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

Let \(j^k\) be the index of the top-left node of the \(k\)-th application:
\[
\forall i \in [1, N_{\text{CUs}}], \forall k \in [1, N_{\text{apps}}], \\
X_{ij}^{\text{CUs} \rightarrow \text{nodes}} = X_{ij}^{\text{CUs} \rightarrow \text{nodes} (i+1)(j^k+1)} \tag{14}
\]

where \(N_{\text{row}}^k\) is the number of nodes per row of the \(k\)-th application.

IV. DECENTRALIZATION OF THE ALLOCATION SYSTEM

In this paper, we use the word decentralized to qualify a system where no single CU has control over all the other ones in the parallel architecture: there is no central CU whose failure jeopardizes the operation of the whole parallel architecture. In safety words, this means that no CU constitutes a single point of failure.
We focus here on CUs, but there are other elements that may be a single point of failure and that we do not take into account in this work. For example, electrical power may be provided by one unique and central power supply unit, which is an obvious single point of failure if not designed carefully. To mitigate the effect of other single point failures, methods for safety assessment process may be conducted [16].

A. N-modular redundancy and majority voting system

To develop a decentralized allocation system for the considered parallel computing platform, we chose to use the concept of N-modular redundancy with a majority voting system [17].

In this approach, \( N_{\text{realloc}} \) is an odd number greater or equal to 3, and \( N_{\text{realloc}} \) copies of the same sub-tasks are executed in parallel. \( N_{\text{realloc}} \) is taken odd to avoid the case where equal number of copies agree on two different results. The copies are fed with the same inputs and their outputs are then sent to a majority voting system. As illustrated in Fig. 5 the voting system compares the outputs of the redundant copies and filters them: only the result that has been computed by the majority of the redundant copies will be transmitted, i.e. the result computed by at least \( \frac{N_{\text{realloc}}+1}{2} \) redundant copies. The voting system is also used to report the failure of the redundant copies that do not match the majority result.

B. Decentralized implementation

The proposed idea to decentralize the allocation system is to execute \( N_{\text{realloc}} \) modular redundant copies of the allocator application on the architecture itself, with a voting system implemented on each CU. For further examples, \( N_{\text{realloc}} \) will be taken equal to 3.

In normal conditions, the \( N_{\text{realloc}} \) copies of the allocator compute the same allocation, since they solve an identical ILP problem, with same inputs and constraints, and because GLPK is a deterministic solver. This allocation is then broadcast to every CU, including the ones executing the allocators.

If a CU not running an allocator fails, all 3 allocators compute the same new allocation, in which the affected application is assigned to a new CU, according the algorithm described previously in Section III. This new allocation is then broadcast and received by all CUs. Since the 3 signals that the CUs receive are coherent, they all comply with it and therefore, the affected application is reallocated.

On the other hand, as illustrated in Fig. 6 if a CU that was running a copy of the allocator is affected by a fault, the 2 other ones will compute the same new allocation where the affected copy is assigned to a new healthy CU. Regardless of what the faulty allocator computes, only the two coherent allocation sent by the two healthy allocators will be taken into account by the CUs, and the faulty allocator will be reallocated.

V. PRACTICAL EXAMPLE

This section describes the experimental setup that is used as a representation of the parallel computing platform as well as the result of reallocating safety-critical applications using the previously-described optimization problem.

A. Representation of a parallel computing platform

1) 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 cluster of single-board computer, Raspberry Pi [18], in order to control and maintain operation of a physical system despite the presence of faults.

a) platform description: In this setup, a cluster of parallel CUs of \( 4 \times 4 \) units is replicated with a network of 16 Raspberry Pi computers. All of them are connected to a common routing switch in a local area network (LAN). Although the use of this common routing switch is a single point failure, it serves a purpose of visualizing that these Raspberry Pi computers are grouped as a single parallel CU. Also, for simplicity of visualization, the network is
considered to be a square mesh, instead of a toroidal mesh.

One alternative of using a wired LAN network is to use a routing protocol for multi-hop mobile ad hoc network such as [19]. This ad-hoc network is implemented on a data link layer, which allows the data transportation protocol operate in a wireless and decentralized fashion as if there is a common routing switch.

The goal of this parallel computing platform is to show the possibility to decentralize the allocation process; therefore, there is no central computing unit outside the network and three copies of the allocator are executed on the network, as described in Section III.

b) Faults: Two types of faults are considered in this experiment. The first type is computational fault, which randomly affect 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-A2. The second type of fault is assumed to stop the operation of the computing unit it affects. We also assume that this fault can be detected by the network. In practice, each time one of these faults affects a Raspberry Pi, the status signal sent by this Raspberry Pi to the allocators 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. 7 connected to each Raspberry Pi.

c) Controlled system: The physical system we chose to control with this parallel computing platform is a propulsion system, made of an electric fan mounted on a thrust stand. The fan is commanded by using Pulse width modulation (PWM). The measure of the thrust is used by a simple proportional controller executed as a safety-critical application on the platform in order to compute the value of the PWM command required to maintain the thrust at a constant value.

A breadboard as seen in Fig. 7 connected to each Raspberry Pi.
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 also use a standard Triple Modular Redundancy (TMR) architecture [17]. Three copies of the controller are executed on the parallel computing platform. 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 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 allocators 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 of actual computations.

Three copies of the allocator execute the allocation algorithm itself. They have second rank priority immediately below the controllers, which represent the safety-critical application in this case. Giving the allocators only the second rank in the priority list can be justified when considering the case where only a controller or an allocator can be executed on the platform: the resource must be allocated to the safety-critical application, in this case the controller, that maintains the operation of the system, whereas the allocator is only a protection against further faults, but cannot alone ensure operation of the controlled system.

In addition to these six applications, one dummy application is considered in this experiment: it occupies 2 Tiles of the Fabric, but does not perform actual computation except changing the voltage in the RGB LED to display its corresponding color. It has the lowest priority.

Figure 9 sums up the list of considered applications for the experiment, their priority and the number of Tiles they require. The initial allocation of the applications on the model is given in Fig. 10.

B. Results

Starting from the initial allocation given in Fig. 11, faults are triggered on the model. After each fault, the allocators detect the faulty Raspberry Pi and compute a new allocation that is then broadcast on the network. They maintain the execution of the safety-critical application as long as enough resources are available for it.

CUs surrounded by faulty neighbors are isolated from the rest of the platform and cannot communicate. As enforced by the communication constraints described in paragraph III-C3f, such a CU is not given any task to execute and is as good as faulty, as seen in Fig. 11a.

When a CU recovers from a fault, an application can be allocated back to it as seen in Fig. 11b. Applications are dropped according to their priority when more computing units become faulty. However, as illustrated in Fig. 11d when no space is available for all 1st priority applications, lower priority ones are still allowed to be executed.

The voting system implemented on the fan needs at least two functioning and coherent controllers to run the fan (Fig. 11d), as previously explained in section V-A2. In case the signals received from the controllers are incoherent, it decides not to trust any of them and the engine stops, as in Fig. 11e.
Since the controllers have the highest priority, they are the last remaining applications to be executed in Fig. 11g. After this step, further faults will affect the controllers but no reallocation can happen because no more allocator is executed.

VI. CONCLUSION

This work presented a decentralized allocation algorithm for parallel computing architectures, where individual Computational Units can be affected by faults. The described method consisted in representing the architecture by an abstract graph and formulating the allocation problem as an optimization problem, with the form of a Integer Linear Program.

Decentralizing the allocation process has been achieved through redundancy of the allocator executed on the architecture. That way, no centralized element decides of the allocation of the entire architecture.

An experimental reproduction of a parallel computing architecture has also been built. It has been used to demonstrate the capabilities of the proposed allocation process to maintain operation of a physical system in a decentralized way while individual component fail.

The proposed work assumed that faults affecting the Computational Units of the architecture were automatically detected by the allocation algorithm, so that it is able to compute a new allocation every time a fault affects a Computational Unit. This work can be improved by defining a more precise model of the considered faults and a method to detect them. One first approach to identify dead Computational Unit would be a simple heartbeat that each would send to the allocators. A CU not sending its heartbeat would be considered faulty. One challenge to tackle in this approach is the fact that the allocators do not have a fixed position in the architecture, and therefore, the heartbeat of each CU would have to be broadcast through the entire architecture to be sure to reach all allocators. Another solution would be to include the position of the allocator in the allocation message that they broadcast, so that the CU know where to send back their heartbeat. In both cases, the amount communication packets transmitted through the architecture drastically increases.

The second lead for improvement is the way communication isolation are taken into account in the allocation problem. For now, only individual nodes with all of their neighbors being faulty were considered isolated. However, an entire area of the architecture, can be isolated from the allocator. The problem becomes quite tricky when the architecture is split in two halves that are isolated one from the other: a decision must be made to decide which area is isolated from the other. It seems that the area with the highest number of allocators should be privileged, since they are the ones that will send the new allocation to other CU, and therefore the ones in the other isolated area will not be able to receive this new allocation. They should therefore be considered as lost CUs.

Also, it should be considered that not only the Computational Units can fail, but also the communication links between them. The effect of such faults would be the same as isolating the Computational Units from their neighbors and would make more of them unavailable. It would also change the communication paths usable to connect the allocators to other applications and would affect their position since minimizing these paths is a part of the optimization problem.

APPENDIX

Coefficients of the objective function

This appendix provides the proof that the coefficients in the objective function from equation 3 allow to meet the requirements stated in Section III-C. For convenience, this objective function is rewritten here:

\[
\max \left\{ f(x) = \sum_{k=1}^{N_{apps}} \alpha_k \cdot r_k - (\beta + 1) \sum_{j=1}^{N_{nodes}} M_j - \sum_{k=1}^{N_{realoc}} \sum_{j=1}^{N_{CUs}} \sum_{i=1}^{N_{paths}} \beta_{ij} \right\}
\]

where

\[
\beta = N_{realoc} \times N_{CUs} \times N_{paths}
\]

\[
\alpha_{N_{apps}} = (\beta + 1) \times N_{nodes} + \beta + 1
\]

and \(\forall k < N_{apps} :\)

\[
\alpha_k = \sum_{l=k+1}^{N_{apps}} \alpha_l + (\beta + 1) \times N_{nodes} + \beta + 1.
\]

The requirements of Section III-C are also rewritten below.

1) When solving the optimization problem, the objective function [3] privileges executing any given application, even if it implies more reallocations and longer communication paths.

2) When solving the optimization problem, the objective function [3] privileges minimizing the number of reallocations, even if it implies longer communication paths.

3) When solving the optimization problem, the objective function [3] privileges executing a given application compared to running any number of applications with a lower priority.

The following theorems prove that these requirements are met.
Theorem 1. \( \forall k \in [1, N_{\text{apps}}] : \)
\[
\alpha_k > (\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} 1 + \sum_{k=1}^{N_{\text{realloc}}} \sum_{i=1}^{N_{\text{paths}}} 1,
\]
that is, the contribution to the value of the objective function for executing application \( \text{app}_k \) is greater than the maximum contribution for reducing the number of reallocations and the length of the communication paths.

Proof. \( \forall k \in [1, N_{\text{apps}}] : \)
\[
\alpha_k \geq (\beta + 1) \times N_{\text{nodes}} + \beta,
\]
by definition of \( \alpha_k \).

Now,
\[
(\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} 1 + \sum_{k=1}^{N_{\text{realloc}}} \sum_{i=1}^{N_{\text{paths}}} 1 = (\beta + 1) \times N_{\text{nodes}} + \beta.
\]

So
\[
\alpha_k > \beta (\beta + 1) \sum_{j=1}^{N_{\text{nodes}}} 1 + \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CU}}} \sum_{i=1}^{N_{\text{paths}}} 1.
\]

Theorem 1 proves that requirement 1 is met.

Theorem 2.
\[
(\beta + 1) \times 1 > \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CU}}} \sum_{i=1}^{N_{\text{paths}}} 1,
\]
that is, the contribution to the value of the objective function for not reallocating one Application node is greater than the maximum contribution for reducing the number of reallocations and the length of communication paths.

Proof.
\[
\beta + 1 > \beta = N_{\text{realloc}} \times N_{\text{CU}} \times N_{\text{paths}} = \sum_{k=1}^{N_{\text{realloc}}} \sum_{j=1}^{N_{\text{CU}}} \sum_{i=1}^{N_{\text{paths}}} 1.
\]

Theorem 2 proves that requirement 2 is met.

Theorem 3. \( \forall k \in [1, N_{\text{apps}} - 1] : \)
\[
\alpha_k > \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l
\]
that is, the contribution to the value of the objective function for executing application \( \text{app}_{k+1} \) is greater than the contribution for executing every applications with lower priority than \( \text{app}_k \), which are \( \text{app}_{k+1} \) to \( \text{app}_{N_{\text{apps}}} \).

Proof. \( \forall k \in [1, N_{\text{apps}}] : \)
\[
\alpha_k = \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l + (\beta + 1) \times N_{\text{nodes}} + \beta + 1 > \sum_{l=k+1}^{N_{\text{apps}}} \alpha_l
\]
since \((\beta + 1) \times N_{\text{nodes}} + \beta + 1 > 0.\)

Theorem 3 proves that requirement 3 is met.

References

[1] 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.

[2] 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.

[3] 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.

[4] J. Nowotsch and M. Paulitsch, “Leveraging multi-core computing architectures in avionics,” in Dependable Computing Conference (EDCC), 2012 Ninth European. IEEE, 2012, pp. 132–143.

[5] F. Reichenbach and A. Wold, “Multi-core technology—next evolution step in safety critical systems for industrial applications?” in Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on. IEEE, 2010, pp. 339–346.

[6] U. Durak and F. Bapp, “Introduction to special issue on multi-core architectures in avionics systems,” 2019.

[7] 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.

[8] M. A. Watkins and D. H. Albonesi, “Remap: A reconfigurable heterogeneous multi-core architecture,” in 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Dec 2010, pp. 497–508.

[9] R. Gerard, “Network on chip (noc) for many-core system on chip in space applications,” Dec 2017.

[10] L. M. Kinnun, “Use of multicores processors in avionics systems and its potential impact on implementation and certification,” in Digital Avionics Systems Conference, 2009. DASC’09. IEEE/AAIA 28th. IEEE, 2009, pp. 1–E.

[11] “Symmetric multi-processor arrangement, safety critical system, and method thereof,” Patent US 2015/0 254 123 A1, Sep. 10, 2015.

[12] M. Oriol, T. Gamer, T. de Gooijer, M. Wahler, and E. Ferranti, “Fault-tolerant fault tolerance for component-based automation systems,” in Digital Avionics Systems Conference, 2009. DASC’09. IEEE/AAIA 28th. IEEE, 2009, pp. 1–E.

[13] F. Reichenbach and A. Wold, “Multi-core technology—next evolution step in safety critical systems for industrial applications?” in Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on. IEEE, 2010, pp. 339–346.

[14] M. Mesbah and M. Egerstedt, Graph theoretic methods in multiagent networks. Princeton University Press, 2010, vol. 33.

[15] “GLPK reference manual,” GNU Linear Programming Kit, 2012, https://www.gnu.org/software/glpk/TOCdocumentation.

[16] F. S. Hillier and G. J. Lieberman, Introduction to Operations Research, 10th ed. New York, NY, USA: McGraw-Hill, 2015.

[17] A. SAE, “Guidelines and methods for conducting the safety assessment of 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.

[18] “Symmetric multi-processor arrangement, safety critical system, and method thereof,” Patent US 2015/0 254 123 A1, Sep. 10, 2015.

[19] M. Oriol, T. Gamer, T. de Gooijer, M. Wahler, and E. Ferranti, “Fault-tolerant fault tolerance for component-based automation systems,” in Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems, June 2013, pp. 49–58.

[20] M. Mesbah and M. Egerstedt, Graph theoretic methods in multiagent networks. Princeton University Press, 2010, vol. 33.

[21] “GLPK reference manual,” GNU Linear Programming Kit, 2012, https://www.gnu.org/software/glpk/TOCdocumentation.

[22] F. S. Hillier and G. J. Lieberman, Introduction to Operations Research, 10th ed. New York, NY, USA: McGraw-Hill, 2015.

[23] A. SAE, “Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment,” London-UK: SAE, 1996.

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

[25] R. P. Foundation, “Raspberry Pi 3 Model B+,” https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ Accessed June 2018.

[26] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich, “Better approach to mobile ad-hoc networking (batman),” IETF draft, pp. 1–24, 2008.
(a) After 4 faults, Application 3 has to be dropped. The isolated Tile cannot be used.
(b) When a Tile recovers from a fault, an allocation can be reallocated to it.
(c) After more faults, Application 3 and a controller have to be dropped by lack of resources.
(d) One controller is dropped. The 3 controllers can still be executed, even if all 1st priority applications are not.
(e) One of the two controllers is affected by a computational fault: the fan stops since the fan voter does not trust any of the two incoherent controllers.
(f) The computational faults disappear: the fan starts again.
(g) All allocators have been dropped: no further reallocation is possible.

Figure 11: Result of the task allocation algorithm. A full video of the demo is available at the link below.
https://gtvault-my.sharepoint.com/:v:/g/personal/lutter6_gatech_edu/EQbz60nu1KkzFo0lftrycIBDyboI9KU0SHSls4ntq8PwAw?e=glyKij