RESILIENT, MULTI-CORE AND SAFETY-CRITICAL COMPUTING ARCHITECTURES

A Thesis
Presented to
The Academic Faculty

By

Tom Guillaumet

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Aerospace Engineering

Georgia Institute of Technology

August 2017

RESILIENT, MULTI-CORE AND SAFETY-CRITICAL COMPUTING ARCHITECTURES

Approved by:

Pr. Eric Feron
School of Aerospace Engineering
Georgia Institute of Technology

Pr. John-Paul Clarke
School of Aerospace Engineering
Georgia Institute of Technology

Pr. Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: July 21, 2017

To my family,

ACKNOWLEDGMENTS

I would like to thank Pr. Eric Feron for giving me the opportunity to work on this research project. I have learned a lot through this experience, on both professional and personal levels. His enthusiasm and keen insight have been a valued source of motivation.

I thank Pr. John-Paul Clarke and Pr. Douglas Blough for being on my thesis committee and for helping me make this thesis better.

I would also like to thank Alysia Watson and Vivian Robinson-O'Neal for helping me patiently through my first days at Tech as well as the rest of these past two years.

I gratefully acknowledge Safran for sponsoring my research at GeorgiaTech, in particular Philippe Baufreton, Cédric Moreau and François Neumann.

I am forever grateful to my parents, who have always encouraged me. To them, to my sister, my grandparents, my cousins, my aunts, my uncles and Bernard, whose passion for aerospace led me to write this thesis, thank you for your unconditional love and support.

To my friends and colleagues at the Decision and Control Laboratory, Hélène, Aayush, Raphaël, Gabriel, Elaud and Harold, thank you for making work so enjoyable. To my friends in the US, in France or elsewhere, in particular Daniel, Steven, Kyle, Greg, Andrea, Ehsan, Tom, Rashad, Cédric, Marine and Maxence, thank you for listening and supporting me through my time in Atlanta.

Finally, to Emma, thank you for being such a beautiful person and for always being so caring.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this thesis, a reconfigurable multi-core architecture is described. Its suitability for executing safety-critical embedded applications is discussed. It is argued that its dynamic features allow for graceful degradation of the system, and that interference channels can be mitigated if spatial partitioning is enforced on its Network on Chip (NoC).

Furthermore, the problem of the allocation of applications on the architecture is formulated as an Integer Linear Programming optimization problem. An algorithm is developed to reallocate the applications running on the fabric when hardware faults occur. The proposed algorithm enforces spatial partitioning on the Network on Chip throughout the reconfigurations. It supports multiple types of NoC topologies, constraints and hardware faults.

Finally, the behavior of the presented algorithm is demonstrated in several configurations and for different scenarios of degradation of the architecture. Its performance in terms of computation time is studied, and the results indicate that its use in a real-time environment is possible.

CHAPTER 1
INTRODUCTION

1.1 Multi-Core Architectures

With the onset of multi- and many-core chips, the single-core market is closing down [1, 2]. Those chips constitute a new challenge for aerospace and safety-critical industries in general [3, 4]. Little is known about the certification of software running on these Systems on Chip (SoC). There is therefore a strong need for developing software architectures based on multi-core architectures, yet compliant with safety-criticality constraints.

To do so, it is possible to use the inherent advantages of multi-core systems [5]. The first one is the potential for graceful degradation: some cores and some parts of the Network on Chip (NoC) – the communication system of the SoC – may become faulty. When such a situation arises, the rest of the computing and communicating resources may remain available to run the application. Then, these systems allow on-chip redundancy: by enabling graceful degradation, an application should be able to maintain its service despite the fault of a part of its cores and network. Each core can be used for multiple functions, and allocated dynamically, so that the system is resilient to losses.

1.2 Certification of Safety-Critical Applications

1.2.1 Need for Constraints

The capabilities of multi-core processors (MCP) can only be utilized if the tasks are properly parallelized and the applications properly segregated [6]. The parallelization is done by resource partitioning: each component of a function that has been parallelized is run on a separate computational core until completion, where each result is then analyzed to give the final result. These partitions must remain independent of each other until completion and failure of one core should not affect other running cores. In addition to the proper building and execution requirements needed for parallelization, there is also a risk of interference, where one application may affect how another independent application runs. This leads to issues with determinism. Determinism is the ability to produce a predictable outcome based on preceding operations and data. This outcome occurs in a specific period of time with repeatability. The specific period of time is a key for safety-critical applications, as each computational core must complete its task in a specified amount of time. If the computation takes longer than allowed, the system and control software may not receive the required data on time, putting the system at risk.

When dealing with safety-critical airborne systems, different components reside at different levels of safety-criticality, with some being more critical than others. Here, we look at three levels of criticality, also known as the Design Assurance Level, or DAL [7]. There is DAL A criticality, where failure can cause a catastrophic effect that may cause a crash. DAL B criticality is where a failure has a large negative impact on the safety or performance of the vehicle, and DAL C criticality is where a failure is seen as significant but has a lesser impact than hazardous failures.

To implement a safety-critical system on a multi-core processor, several constraints must be satisfied. When running the system, all tasks must be completed within the required time constraints. Interference comes about when applications affect the runtime operations of other running applications. Interference will come about when resources are shared, or when certain configurations on the execution fabric do not hold their intended independence.

In addition to interference coming about from overlapping communication channels, a multi-core processor utilizes shared memory and shared communication buses. The shared memory is a source of interference as all computations utilize the memory available. Similar issues are seen when the MCP communicates with the host. All of the data and commands are passed through a series of routers that may handle multiple sources of data and results. Though the shared memory and shared communication bus are possible sources of interference, they can be properly utilized to reduce possible interference and ensure determinism. Determinism is a critical attribute for the processor. When interference comes about, determinism is lost, and the accuracy of the results cannot be guaranteed.

Next, it must be shown that all computations are executed within any required time constraint. Finally, any dynamic features must be properly understood, validated and documented.

1.2.2 Integrated Modular Avionics

Since the beginning of the 1990s, federated avionics architecture – where one computing resource executes only one application – tend to be replaced by Integrated Modular Avionics (IMA) architectures [8, 9]. IMA aims at simplifying the development of avionics software. In particular, it allows integrating functions of different criticality levels onto shared hardware [10, 11].

As for federated systems, IMA systems must enforce critical functional separation: an application cannot affect the behavior of another application. This key requirement of IMA is called robust partitioning. Its objective is to provide IMA systems with the same level of isolation and independence between applications as federated systems [12]. The partitioning must be enforced in both time and space.

To enforce space partitioning, storage locations must only be writable by one application. To enforce time partitioning, one application must be guaranteed access to the hardware resources it requires for a period of time [13].

The standard RTCA DO-297 [14] contains guidance for the certification of integrated avionics systems and architectures [15].

1.2.3 Certification and Implementation of Multi-Core Architectures

With the introduction of multi-core processors in avionics systems, the challenge is to keep enforcing robust partitioning in IMA architectures [16].

The FAA and EASA worked with industry to quantify a set of requirements that must be met to certify and use multi-core processors in civil aviation, described in the FAA CAST-32A Position Paper [17] and the EASA MULCORS [18] research report. When certain issues arise from failure to meet the parameters as stated previously, depending on the DAL criticality, systems can fail. To ensure determinism and operations within the required time constraints, multi-core parallelization capabilities must be properly utilized and interference channels must be removed. This robust partitioning of the multi-core chip ensures that every computational core or compute resource used remains independent and isolated from other concurrently running compute resources. With this implementation, the operations will be fully parallelized and each operation should complete within the allotted time.

However, the issue of shared memory and shared communication paths remains. To mitigate any possible interference, detailed resource definitions must be available within each computational core to track the relevant data. Each compute resource must track each operation being done inside and being sent to write only memory.

So far, the certification requirements given by the FAA and EASA do not cover the multi-core platforms that enable the dynamic allocation of software applications during operation, which is the case of REDEFINE. Nevertheless, the current guidance is used as a basis for this thesis.

1.3 Reconfigurable Multi-Core Architectures

There exist several different types of reconfigurable multi-core computing architectures [19, 20, 21]. The factors of reconfiguration can be the data path, the memory, the power, the interconnect, etc.

The reconfigurable architecture studied in the next chapter is the REDEFINE architecture [22]. Its factors of reconfiguration are:

- the data path: the compute resources can be re-aggregated at runtime on the fabric,
- the mode: the allocation of the different applications to the compute elements can be modified at runtime.

1.4 Objectives of the Thesis

The objective of the thesis is to analyze an actual reconfigurable multi-core architecture and to study its suitability for safety-critical embedded applications.

This will be fulfilled by answering the following questions:

- What is the structure of the architecture and is why is it suitable for safety-critical applications?
- How can we take advantage of its features to use it in a safety-critical embedded context?
- Can we design a task allocation algorithm to manage the architecture in real-time?

1.5 Thesis Outline

This thesis is organized in five chapters. The first chapter consisted in an introduction to the work, a review of the previous work that led to this thesis and a description of its objectives.

The following chapters answer the research questions asked above. Chapter 2 describes the structure and the behavior of the REDEFINE architecture, developed at the Indian Institute of Science (IISc).

Chapter 3 addresses the mathematical formulation of the real-time allocation of applications problem on REDEFINE, and the elaboration of a centralized algorithm able to manage the architecture under certain assumptions.

Chapter 4 is an analysis of the results given by this algorithm in different configurations, and also includes an analysis of its performance.

Chapter 5 draws the conclusions of the thesis and proposes possibilities for future work.

CHAPTER 2

REDEFINE ARCHITECTURE

The basis for this thesis is the REDEFINE architecture, developed at the Indian Institute of Science (IISc). This architecture is described in this chapter, and its suitability for safety-critical applications is studied [23].

2.1 Description of the Reconfigurable Architecture

In this section, a high-level description of the reconfigurable multi-core architecture REDEFINE is given. This architecture allows for processes to be dynamically allocated to the computing resources at runtime.

2.1.1 Execution Model

REDEFINE is composed of an Executable Fabric, an external memory and a Resource Manager. The REDEFINE Resource Manager (RRM) is the front-end of the fabric, and connects it to a host, as seen on Figure 2.1.

The host creates and launches the different kernels that will be executed on the fabric, through the Resource Manager. Each kernel is defined as an interaction among a collection of HyperOps. HyperOps are the principal scheduling entities of REDEFINE, and are executed in a non-preemptive manner. The compiler is responsible for the creation of the HyperOps.

Once a kernel has been executed and its results are available, they are communicated to the host via the Resource Manager.

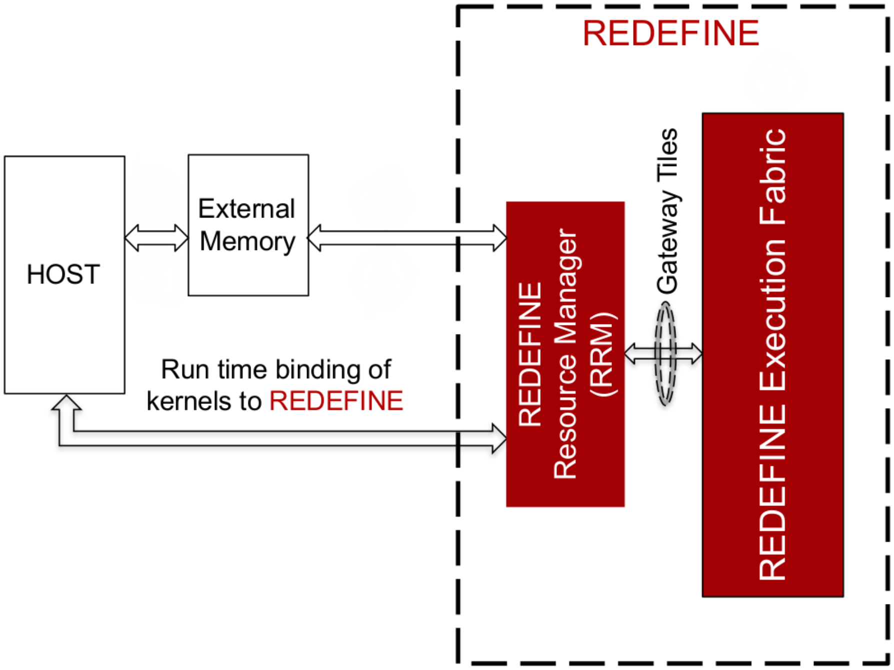

REDEFINE
HOST
External Memory
Run time binding of kernels to REDEFINE
REDEFINE Resource Manager (RRM)
Gateway Tiles
REDEFINE Execution Fabric

Figure 2.1: The different components of the REDEFINE architecture.

2.1.2 Fabric Description

The fabric is made of a certain number of tiles, connected in a toroidal mesh topology (as seen on Figure 2.3) through the Network on Chip (NoC). Each tile contains a Compute Resource (CR) and a router, as seen on Figure 2.2. The CR (a variation of the one presented in [24]) is composed of four Compute Elements (CEs), an orchestrator, a transporter, a Context Memory (CM) and a piece of the distributed memory of the fabric. The aggregation of all the individual tiles' memories forms the Distributed Shared Memory (DSM) of the architecture. Each Compute Element is an instruction set processor that can execute a subset of the RISC-V instruction set architecture as well as some REDEFINE-specific instructions.

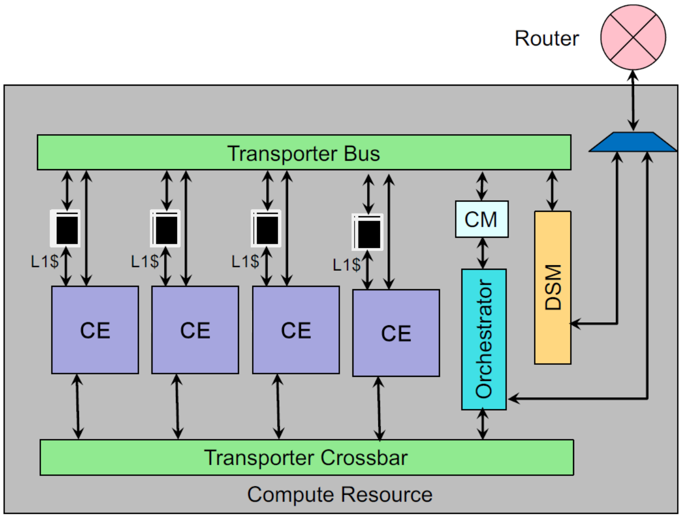


Figure 2.2: Representation of a REDEFINE tile. It contains a router and a Compute Resource.

A certain number of tiles, on one edge of the fabric, are dedicated to the communication between the Resource Manager and the fabric. These tiles, called the gateway tiles, are only composed of routers (as seen on Figure 2.3).

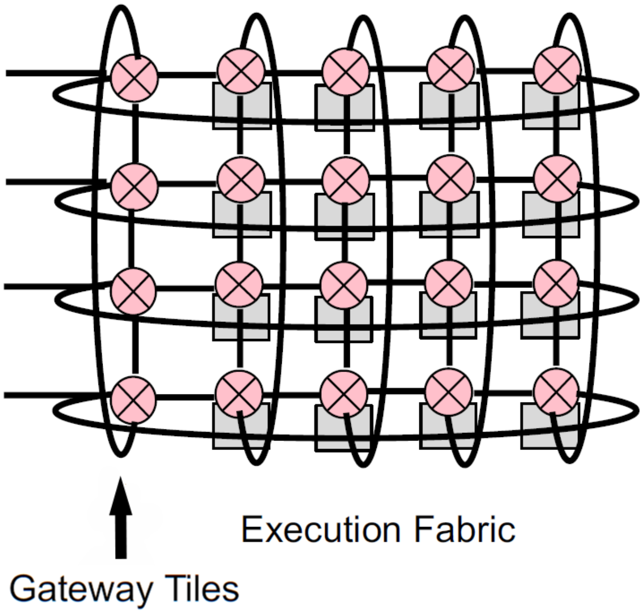


Figure 2.3: The toroidal mesh topology of the NoC for a 4 × 4 fabric. The pink circles represent routers, and the gray squares represent Compute Resources.

The routing algorithm implemented on REDEFINE is a XY deterministic algorithm [25]. To reach its destination, a packet first moves horizontally along the X-axis to reach the destination's column and then vertically along the Y-axis to reach its row, as shown on Figure 2.4.

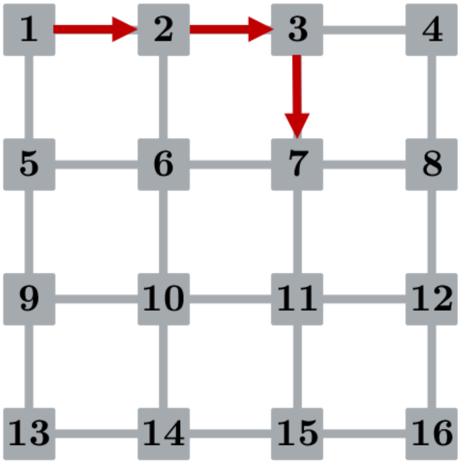


Figure 2.4: Illustration of the XY routing algorithm implemented on REDEFINE. The red arrows indicate the path followed by a packet sent from tile 1 to tile 7.

2.1.3 Compiler

The input to the compiler is a program described in a High-Level Language such as C. During the data-flow analysis, the compiler constructs the Data Flow Graph. The basic blocks are then aggregated into HyperOps to form an acyclic graph called the HyperOp Interaction Graph, describing the producer-consumer relationships between the HyperOps (as seen on Figure 2.5). Each HyperOp is a subset of the program's instructions.

The HyperOps are then partitioned into p-HyperOps. Each p-HyperOp is a subset of the HyperOp's instructions that will be executed by one Compute Element of the Compute Resource. Therefore, several instructions of a same HyperOp can be executed simultaneously within a Compute Resource. The constraint for the formation of the HyperOps is that one HyperOp execution should not span more than one Compute Resource.

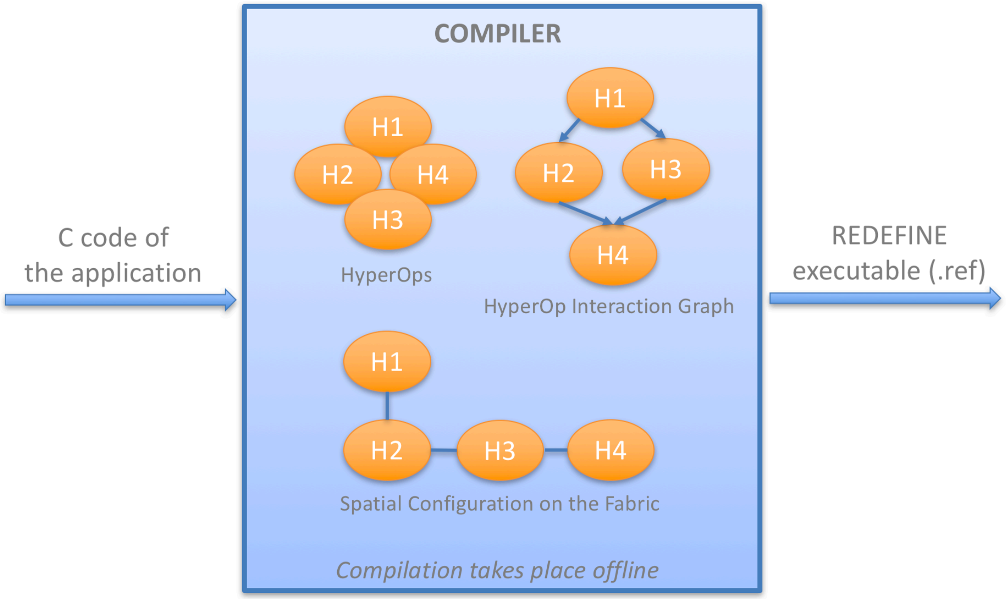

C code of
the application
COMPILER
H1
H2
H4
H3
HyperOps
H1
H2
H3
H4
HyperOp Interaction Graph
H1
H2
H3
H4
Spatial Configuration on the Fabric
Compilation takes place offline
REDEFINE
executable (.ref)

Figure 2.5: Illustration of the compilation process.

The compiler computes the number of HyperOps needed for each application, as well as the spatial configuration of these HyperOps on the fabric. Since intra-application communication is realized using the relative (x, y) positions of the Compute Resources, each application must be implemented on the fabric in the exact spatial configuration and orientation computed by the compiler.

The output of the compiler is the application written in a format that is executable on the REDEFINE architecture. The compiling process takes place offline, before execution.

2.1.4 Host

The user is responsible for creating the different kernels that will hold the different applications to be executed on the architecture. The user provides the binary of the kernel (output of the compiler) that will be run on the fabric. Once this is done, the different kernels can be launched on the fabric via the Resource Manager and the gateway tiles.

The resource requirements for a kernel are specified in terms of numbers of Compute Resources. The memory requirements of a kernel should fit into the Distributed Shared Memory banks it is allocated to. During the kernel execution phase, the external memory is not accessed by the kernel.

2.1.5 Runtime

At runtime, the kernels are launched on the fabric by the Resource Manager, provided that sufficient Compute Resources are available. Inside a kernel, the HyperOps are launched in a data-driven manner, following the HyperOp Interaction Graph. The orchestrator of each CR is responsible for selecting and launching the HyperOps on the Compute Elements. A HyperOp is considered ready to be launched when all its input operands are available. If several HyperOps are ready to be launched at the same time, the selection logic of the orchestrator selects the HyperOp that should be launched first, following a depth-first scheduling algorithm.

A kernel execution starts with a single HyperOp annotated as the Start-HyperOp and ends with a single HyperOp annotated as the End-HyperOp. After execution of the End-HyperOp of a kernel, the REDEFINE Resource Manager transfers the results of the kernel to the host, through the NoC and the gateway tiles. It also informs the host that the kernel execution has ended, and resets the resources used by the kernel.

The resources are freed for allocation to other kernels.

2.1.6 Parallelism

There are multiple levels of parallelism within the architecture, provided that sufficient resources are available:

- different kernels can be executed simultaneously on the fabric,
- within a kernel, different HyperOp instances can be executed simultaneously on different Compute Resources,
- within a HyperOp instance, different p-HyperOps can be executed simultaneously on different Compute Elements.

2.2 Suitability of REDEFINE for Safety-Critical Applications

REDEFINE is developed primarily for high-performance computing purposes, such as cryptographic algorithms [26]. Nevertheless, its dynamic features could also be used to build a fault-tolerant safety-critical embedded system.

In this section, the suitability of the REDEFINE architecture for safety-critical applications is studied through its memory structure and its Network on Chip. This constitutes a first high-level interference analysis of the architecture.

2.2.1 Memory Management

As memory is a shared resource, it can be a source for interference. The memory management and memory accesses on REDEFINE are studied first.

Distributed Shared Memory

On REDEFINE, the shared memory is distributed among all the Compute Resources. The compiler specifies the resource requirements of each kernels in terms of Compute Resources and memory space. These requirements are specified to the REDEFINE Resource Manager (RRM). At the kernel launch time, the RRM binds the logical CRs to the physical CRs and loads the Distributed Shared Memory banks with the kernel image available in the external memory.

Each Compute Element executing instructions from one kernel can read and write in the range of addresses assigned to this kernel. As a HyperOp execution is contained within a CR, all intra-HyperOp communications (communication among p-HyperOps) will take place using the CR's transporter. This way, the NoC is not used for intra-HyperOp communication. However, inter-HyperOp communication happens through context memory. A producer HyperOp provides data to a consumer HyperOp by performing a write to its context frame. Depending on the location of the context frame, the data transfer takes place via the NoC.

The kernel address space is distributed among the Compute Resources allocated to the kernel, and shared by all the Compute Elements of the CRs for data loads and stores. The memory management ensures spatial isolation by making CRs private to the allocated kernel.

Context Memory

Each Compute Resource contains a Context Memory (CM), as shown on Figure 2.2. The CR's orchestrator is responsible for the Context Memory management. The CEs can only write in the Context Memory, while the orchestrator can only read in this memory.

The Context Memory is divided in several context frames, each one being dedicated to one HyperOp instance that will be executed on the CR. The input operands to a HyperOp instance (up to 16) are stored in this context frame, as well as the HyperOp metadata, containing information such as the HyperOp ID, the number of its operands that have been received, the number of p-HyperOps it contains and the number of input operands yet to be received. When a HyperOp is launched on one Compute Resource, the orchestrator initializes the Compute Elements' register files with the input operands of all the p-HyperOps they will execute.

To exploit locality and reduce the NoC bandwidth required for the execution of a kernel, a data producer HyperOp and its consumer HyperOp can be allocated to the same CR at compile time. This implies giving up on the maximum available parallelism but reduces the NoC traffic. The less a kernel requires network bandwidth, the less its execution is sensitive to NoC traffic.

Register File and Cache

Each CE has a private instruction cache, a data cache and a register file, where the inputs operands for each p-HyperOp are loaded during the HyperOp launch. A HyperOp communicates its results to other HyperOps through the Context Memory or the Distributed Shared Memory (DSM).

In communication through the DSM, the producer HyperOp stores its results in the DSM address space and the data cache and then communicates the respective DSM address to the consumer HyperOp via inter-HyperOp communication channels. If the producer and the consumer HyperOp are associated with different CRs, at the end of the producer HyperOp's execution its data caches are reconciled with the DSM.

The consumer HyperOp's Compute Resource caches are flushed before its execution and the required data is read from the DSM.

The cache coherence is ensured through a dag-consistency Distributed Shared Memory model [27]. L1 cache fills and write-backs happen through the NoC. The results will then be sent back to the host via the NoC and the gateway tiles (see Figure 2.3).

2.2.2 Resource Partitioning

The interference analysis of REDEFINE must be done at different levels. At the highest level, one kernel running on REDEFINE should not affect the behavior of another kernel that is or will be running on the fabric. On REDEFINE, the segmented memory management ensures that the logical address translation of one kernel does not lead to accessing a physical memory space allocated to another kernel. As memory and Compute Resources are not shared by multiple kernels, a kernel cannot affect the correctness of another kernel. The only resource shared by multiple kernels is the NoC. Therefore, one kernel can only affect the execution time of another kernel.

The compiler ensures that the unordered HyperOps are data-race free. This is required to ensure a deterministic parallelism: the output of a kernel must not depend on the order in which the unordered HyperOps are executed.

Nevertheless, the execution time of a HyperOp may be affected by another HyperOp execution if they share the same communication resources.

2.2.3 Network On Chip

The Network on Chip can be a source of interference. Depending on the traffic on the NoC, there might be time delays associated with the data communication between Compute Resources, or between the fabric and the Resource Manager. Since the intra-HyperOp communication does not use the NoC, this cannot happen within a HyperOp.

On REDEFINE, the NoC is used:

- by the Resource Manager to load input data and the executables of the kernels to Compute Resources,
- by the Compute Resources for inter-HyperOp communication during kernel execution,
- by Compute Resources for L1 cache fills and write-backs during kernel execution,
- by the Resource Manager to read the outputs of the kernels from the Distributed Shared Memory.

2.2.4 Running Safety-Critical Applications on REDEFINE

REDEFINE allows for real-time reallocations of applications on the fabric. This motivates the study of its suitability for safety-critical applications.

Indeed, provided that spatial partitioning is enforced on the Network on Chip throughout the execution of the applications, interference between the applications can be avoided.

Furthermore, in case of a hardware fault, provided that the fault is detected, the Resource Manager could reallocate the applications so that the system keeps running as long as enough Compute Resources are available.

The following chapter addresses the development of a centralized task allocation algorithm that can:

- handle applications of different criticality levels,
- reallocate the applications in real-time when a hardware fault is detected,
- constantly enforce spatial partitioning on the Network on Chip.

CHAPTER 3
TASK ALLOCATION ALGORITHM

As explained in the previous chapter, the dynamic features of REDEFINE can be exploited to design a safety-critical embedded system that is tolerant to faults. Indeed, in case of a hardware fault on the fabric, the Resource Manager could reconfigure the applications in real-time to ensure that the system keeps running.

To fulfill this objective, the task allocation problem on REDEFINE has to be mathematically formulated and solved each time a hardware fault occurs.

In this chapter, a high-level mathematical formulation of the problem is given. The goal is to allow the reconfiguration of the applications in real-time when hardware faults occur on the platform, with the constraint of enforcing spatial partitioning on the Network on Chip.

3.1 Description of the Problem

Our interest lies in the dynamic reallocation of tasks when hardware faults occur on the platform. Throughout the execution, spatial partitioning must be enforced on the Network on Chip between the applications, to avoid interference, i.e. timing delays.

This thesis only addresses two types of high-level hardware faults within the fabric tiles:

- faulty Compute Resources,
- faulty routers.

The algorithm that is developed is a centralized task allocation algorithm that would eventually be implemented at the REDEFINE Resource Manager level.

3.1.1 Assumptions

As this work addresses the real-time task allocation problem on REDEFINE, the following assumptions are made:

- hardware faults are detected when they occur,
- the health of the architecture is known to the Resource Manager.

The health monitoring system and the hardware fault models are not discussed in this thesis. Under these two assumptions, the task allocation algorithm is formulated.

3.1.2 Approach

The task allocation problem can be formulated as an Integer Linear Programming (ILP) optimization problem, using incidence matrices to describe the compute resources, the topology of the NoC and the routing algorithm [28]. The decision variables are also incidence matrices, describing the mapping of the applications onto the compute resources and the mapping of the communication requirements between the applications onto the communication paths of the NoC. The constraints are used for instance to enforce spatial partitioning between applications, or to avoid allocating applications to faulty compute resources. The objective function can for example be used to minimize the total traffic on the Network on Chip. The algorithm has to be efficient enough to compute a suitable reconfiguration without affecting too much the ability of the system to meet hard real-time constraints.

There is also interest in executing a safety-critical application such as an engine controller (DAL A) in parallel with a non-critical application, for instance an engine health-monitoring software. Indeed, safety-critical applications are typically low-demanding in terms of computational power and network traffic, leaving a substantial amount of available resources on the multi-core platform unused. These remaining resources could be used to execute computational demanding applications and thus to better exploit the multi-core architecture's abilities. However, if hardware faults occur, the execution of these non-critical applications can be stopped in order to free compute resources and communication paths for the safety-critical applications, using a prioritization system. This can also be handled by an Integer Linear Programming approach.

3.2 Integer Linear Programming Formulation

3.2.1 Modeling the Architecture

As stated above, the online task-allocating problem on the Network on Chip can be formulated as an Integer Linear Programming problem [29]. Previous work is adapted and extended to address issues that are specific to the REDEFINE architecture and safety-critical systems, in particular spatial partitioning on the NoC.

The Network on Chip's topology is described by an undirected graph. Each vertex represents a Compute Resource and each edge represents a communication path between two Compute Resources. The matrix G is a $N_{CR} \times N_{paths}$ incidence matrix representing this graph, where N_{CR} is the number of CRs on the fabric and N_{paths} is the number of communication paths.

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

3.2.2 Modeling the Applications

Several undirected graphs represent the N_{apps} different applications that are to be executed on the platform. On REDEFINE, the compiler computes the number of Compute Resources (and their spatial configuration) that each application requires, with the objective of maximizing parallelism (as shown on Figure 2.5). Therefore, each vertex of a graph represents a Compute Resource that will be assigned to the application and each edge represents a communication path that links two of these Compute Resources. The matrix A^k is a $N^k_{nodes} \times N^k_{links}$ incidence matrix representing the application k, where N^k_{nodes} is its number of nodes and N^k_{links} its number of links.

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

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

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

3.2.3 Modeling an Heterogeneous Architecture

Other incidence matrices can be used to specify the type of each Compute Resource and communication path on the NoC. The different types of Compute Resources can be used to describe the ability of some of the CRs to execute specific operations (linear algebra, encryption, etc.). The different types of communication paths can for instance be used to describe the maximum capacity of the communications paths, or to describe which path can realize which type of communication. Both the communications paths and the CRs can be of several different types.

T^{CR} is a $N_{CR} \times N^{CR}_{types}$ incidence matrix describing the types of each Compute Resource. N^{CR}_{types} is the number of different CR types. T^{paths} is a $N_{paths} \times N^{paths}_{types}$ incidence matrix describing the types of each NoC path. N^{paths}_{types} is the number of different communication paths types.

$$T^{CR}_{ij} = \begin{cases} 1 & \text{if the CR } i \text{ is of type } j \\ 0 & \text{otherwise} \end{cases}$$

$$T^{paths}_{ij} = \begin{cases} 1 & \text{if the NoC path } i \text{ is of type } j \\ 0 & \text{otherwise} \end{cases}$$

For each application A^k, the type of each node is described in the $N^k_{nodes} \times N^{CR}_{types}$ incidence matrix TN^k. If a given node is of a certain type, it will need to be allocated to a CR of the same type.

$$TN^k_{ij} = \begin{cases} 1 & \text{if the node } i \text{ is of type } j \text{ in application } k \\ 0 & \text{otherwise} \end{cases}$$

For each application A^k, the type of each link is described in the $N^k_{links} \times N^{paths}_{types}$ incidence matrix TL^k.

$$TL^k_{ij} = \begin{cases} 1 & \text{if the link } i \text{ is of type } j \text{ in application } k \\ 0 & \text{otherwise} \end{cases}$$

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

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

The same can be done with the TN^k and TL^k $(k = 1, \ldots, N_{apps})$ matrices:

$$TN = \begin{bmatrix} TN^1 \\ \vdots \\ TN^k \end{bmatrix} \quad \text{is a } N_{nodes} \times N_{types}^{CR} \text{ matrix.}$$

$$TL = \begin{bmatrix} TL^1 \\ \vdots \\ TL^k \end{bmatrix} \quad \text{is a } N_{links} \times N_{types}^{paths} \text{ matrix.}$$

3.2.4 Decision Variables

The decision variables of the problem describe the mapping of the applications' vertices on the NoC Compute Resources and the mapping of the applications' edges on the NoC communication paths.

$$X_{ij}^{CR \to apps} = \begin{cases} 1 & \text{if the CR } i \text{ is allocated to the application node } j \\ 0 & \text{otherwise} \end{cases}$$

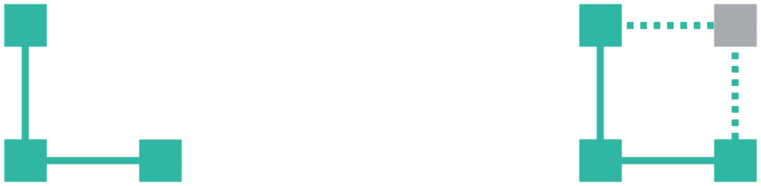
$$X_{ij}^{paths \to links} = \begin{cases} 1 & \text{if the NoC path } i \text{ is allocated to the application link } j \\ 0 & \text{otherwise} \end{cases}$$

$X^{CR \to apps}$ is a $N_{CR} \times N_{nodes}$ matrix and $X^{paths \to links}$ is a $N_{paths} \times N_{links}$ matrix.

3.2.5 Spatial Partitioning

The routing algorithm implemented on REDEFINE is a XY deterministic algorithm (see Figure 2.4). To reach its destination, a packet first moves along the X-axis to reach the destination's column and then along the Y-axis to reach its row. Therefore, an application can use the router of a tile while not using the CR of the same tile.

To enforce spatial partitioning on the Network on Chip, a tile whose router is used by an application will be considered as dedicated entirely to this application. This is symbolized by the gray square on Figure 3.1. The application does not use the CR of the upper right tile but it uses its router for intra-application communication, for instance from the upper left tile to the bottom right tile. We will refer to these nodes as "ghost" nodes.



(a) The spatial configuration of an application as computed by the compiler.

(b) A "ghost" node (in gray) is added because the top right tile's router is used by the application for intra-application communication.

Figure 3.1: Every tile that is used by an application for intra-application communication is considered as a node of this application.

To avoid interference between the applications on the NoC, we must ensure that two applications never share communication paths and routers. Therefore, in our formulation, we consider that the ghost nodes of the applications belong to the set of applications' nodes. The difference with the other colored "true" nodes is that the application can still run if the Compute Resource of the tile allocated to a "ghost" node is faulty while the router of this tile is healthy.

3.3 Constraints

The task allocation problem can now be formulated. The basic set of constraints is the following:

1. $\forall i = 1, \dots, N_{CR},\ \sum_{k=1}^{N_{nodes}} X_{ik}^{CR \rightarrow apps} \leq 1 \iff$ a CR can be allocated to at most one application.

2. $\forall i = 1, \dots, N_{nodes},\ \sum_{k=1}^{N_{CR}} X_{ki}^{CR \rightarrow apps} = 1 \iff$ each application node must be assigned to exactly one CR.

3. $\forall i = 1, \dots, N_{paths},\ \sum_{k=1}^{N_{links}} X_{ik}^{paths \rightarrow links} \leq 1 \iff$ a NoC path can be allocated to at most one application link.

4. $\forall i = 1, \dots, N_{links},\ \sum_{k=1}^{N_{paths}} X_{ki}^{paths \rightarrow links} = 1 \iff$ each application link must be assigned to exactly one NoC path.

5. $X^{CR \rightarrow apps}\ A = G\ X^{paths \rightarrow links} \iff$ 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.

6. $(X^{CR \rightarrow apps})^T\ T^{CR} = TN \iff$ an application node of type i must be assigned to a CR of type i.

7. $(X^{paths \rightarrow links})^T\ T^{paths} = TL \iff$ an application link of type i must be assigned to a NoC path of type i.

8. All the decision variables are binary.

The constraints 1 and 3 are sufficient to ensure that spatial partitioning is enforced on the NoC, as the "ghost" nodes are considered to be part of the applications.

However, more constraints need to be added to fulfill the objectives of the task allocation problem.

3.3.1 Faulty Compute Resources and Routers

More constraints need to be added to take into account that some Compute Resources or routers may become faulty. Several cases are to be considered. Within a tile:

- If both the Compute Resource and the router are healthy, any application node can be mapped on the tile.
- If the Compute Resource is faulty but the router is healthy, only "ghost" application nodes can be mapped on the tile.
- If the router is faulty, regardless of the health of the Compute Resource, then no application nodes can be mapped on the tile.

Therefore, to prevent the algorithm from allocating an application node to a tile i that has a faulty router, the following constraint is used:

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

To prevent the algorithm from allocating a true application node to a tile i that has a faulty Compute Resource, the following constraint is used:

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

3.3.2 Spatial Orientation of the Applications

For each application it compiles, the REDEFINE compiler outputs the spatial configuration and orientation of the Compute Resources to which the application will be mapped. Intra-application data communication is achieved using the relative (x, y) positions of the CRs. Therefore, the applications cannot be rotated and must be strictly implemented in the spatial configuration given by the compiler.

To enforce this requirement in the ILP formulation, we use the fact that the CRs are numbered, as shown on Figure 3.2.

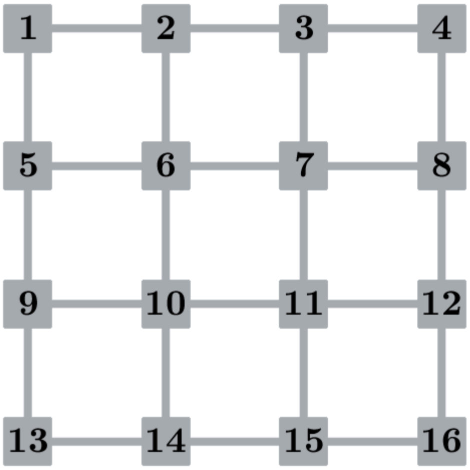


Figure 3.2: Example of a 4 × 4 mesh topology. Each gray square represents a tile composed of a Compute Resource and a router. The edges represent communication paths.

Therefore, using this numbering, we can ensure that the applications are strictly mapped in the spatial orientation that the compiler computed. Indeed, the relative orientation of two Compute Resources can be specified by ensuring that the difference between the two CRs' numbers is equal to a certain value, as shown on Figure 3.3.

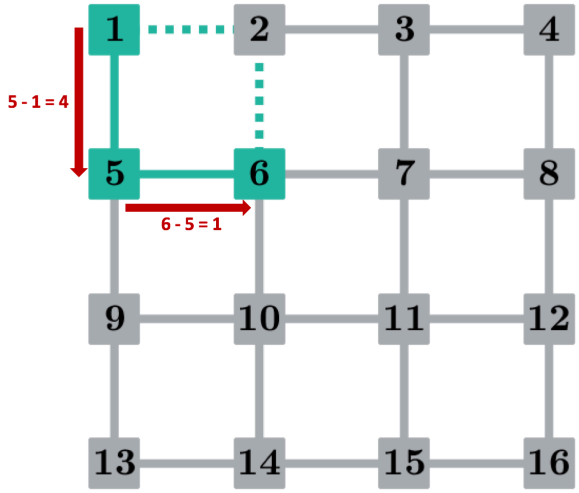


Figure 3.3: By ensuring that the difference between two CRs' numbers allocated to an application is equal to a specific number, the spatial orientation of the application can be enforced. Here, only two of those constraints are necessary.

In the example above (Figure 3.3), two constraints are necessary to ensure that the application has the right orientation:

- the difference between the number of the CR allocated to the bottom left CR and the number of the CR allocated to the upper left CR must be equal to 4,
- the difference between the number of the CR allocated to the bottom right CR and the number of the CR allocated to the bottom left CR must be equal to 1.

A set of constraint is added to the problem to ensure that the difference between the numbers of the contiguous pairs of CRs allocated to an application matches the orientation of this application, as computed by the compiler.

3.3.3 Clusters

Using a set of constraints, we can also ensure that an application is only mapped to a certain subset of CRs on the NoC, and that the other applications cannot be mapped on this subset. This has the effect of creating a cluster of CRs that is dedicated to an application.

For instance, to make sure that one application named *critical* is mapped on a set of contiguous CRs named *cluster* and that no other application is mapped to this set, the following constraints must be added:

- $\forall i \in cluster,\ \sum_{k \in apps \setminus critical} X_{ik}^{CR \to apps} = 0$ where $apps \setminus critical$ is the set of application nodes that do not belong to $critical$,
- $\forall i \in CRs \setminus cluster,\ \sum_{k \in critical} X_{ik}^{CR \to apps} = 0$ where $CRs \setminus cluster$ is the set of CRs that are not in $cluster$.

3.3.4 Reallocating only one Application

As the reconfiguration process affects the behavior of an application for a certain amount of time, constraints are added so that only the application that is affected by the hardware fault is reallocated.

This can be done if the decision matrix $X^{CR \to apps}$ is stored after each reconfiguration. If the previous value of $X^{CR \to apps}$ is equal to $X_{old}^{CR \to apps}$, when a hardware fault occurs and affects the application *affected*, then the following constraint can be enforced for the next reconfiguration:

$$\forall i = 1, \ldots, N_{CR},\ \forall j \in apps \setminus affected,\ X_{ij}^{CR \to apps} = X_{old\ ij}^{CR \to apps}$$

where $apps \setminus affected$ is the set of application nodes that do not belong to the affected application.

3.4 Real-Time Reconfiguration

This section details the functioning of the centralized task allocation algorithm under the assumptions given above.

In real-time, there are only two cases of hardware fault in which a reconfiguration needs to be computed:

1. if a true application node is mapped on a tile which CR becomes faulty,

2. if any application node is mapped on a tile which router becomes faulty.

Indeed, a faulty CR within a tile does not affect an application which has one of its “ghost” application nodes mapped to this same tile – it only uses its router.

3.4.1 Priorities

As stated above, as there is interest in executing one safety-critical application along with non safety-critical applications simultaneously on the platform, a prioritization system is implemented in the algorithm.

The prioritization of the applications allows to drop the non safety-critical applications if not enough resources are available for the critical one.

To do so, the algorithm takes a variable *apps* as a parameter, describing the set of applications it should allocate on the fabric. When the algorithm fails to reconfigure all the applications on the fabric, one or several non-critical applications can be removed from the set *apps*.

3.4.2 Algorithm

The algorithm takes several parameters:

- *affected* which is the set of application nodes belonging to the application that is affected by the hardware fault,
- *apps* which is the set of applications that are to be considered for the reconfiguration process,
- $X_{old}^{CR \to apps}$ which is the value of the decision matrix $X^{CR \to apps}$ after the last reconfiguration,
- *cluster* which is a boolean indicating if the safety-critical application must be mapped within a specified cluster or not.

Using these parameters, the Integer Linear Programming problem can be solved in real-time, depending on the health of the architecture.

Figure 3.4 shows an example of flowchart of the algorithm in the case where no cluster has been defined.

In the case where a cluster has been defined for the safety-critical application, meaning that a subset of the fabric tiles is dedicated to the critical application, the flowchart is slightly modified. If a hardware fault affects the critical application, the algorithm tries to reconfigure it inside the cluster. If this is not possible, then all the non-critical applications are dropped and the algorithm allows the critical application to be allocated on the whole fabric. If a hardware fault affects a non-critical application, then it can only be reconfigured outside of the cluster.

The algorithm stops when no solution was found to reallocate the safety-critical application.

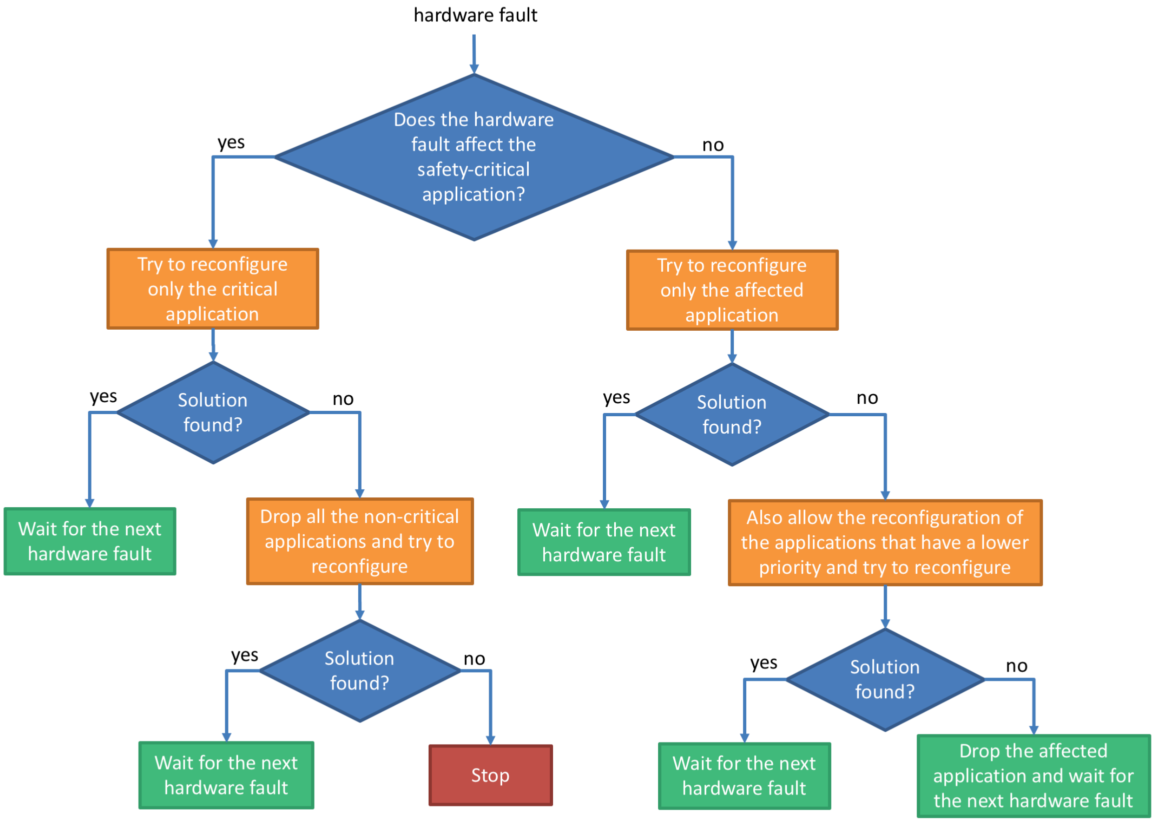

hardware fault
Does the hardware fault affect the safety-critical application?
yes
no
Try to reconfigure only the critical application
Try to reconfigure only the affected application
yes
Solution found?
no
yes
Solution found?
no
Wait for the next hardware fault
Drop all the non-critical applications and try to reconfigure
Wait for the next hardware fault
Also allow the reconfiguration of the applications that have a lower priority and try to reconfigure
yes
Solution found?
no
yes
Solution found?
no
Wait for the next hardware fault
Stop
Wait for the next hardware fault
Drop the affected application and wait for the next hardware fault

Figure 3.4: Flowchart of the proposed algorithm.

3.5 Objective Function

For this high-level task allocation algorithm, the problem is considered as being a feasibility problem rather than an optimization problem. The goal is to determine if a feasible solution exists.

However, objective functions can be used for instance to:

- minimize/maximize the usage of certain tiles,
- minimize/maximize the usage of certain communication paths of the NoC,
- minimize/maximize the spatial distance between different applications on the fabric.

3.6 Solving the Optimization Problem

Using the optimization problem solver Gurobi [30], the Integer Linear Programming problem with the constraints and objective functions formulated above can be solved. The problem is modeled on MATLAB using the modeling framework CVX [31, 32].

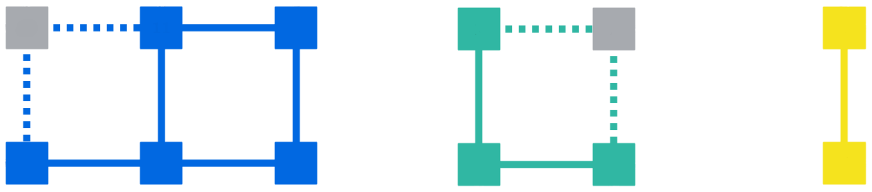
CHAPTER 4

RESULTS

In this chapter, the behavior of the real-time task allocation algorithm is studied in different scenarios, as well as its efficiency.

4.1 Capabilities of the Algorithm

In this section, the behavior of the algorithm in different configurations is demonstrated. In the following examples, the three applications shown on Figure 4.1 are considered.



(a) Application 1 (b) Application 2 (c) Application 3

Figure 4.1: The spatial configurations of the three different applications that are to be executed on the platform. The gray squares represent the "ghost" nodes of the applications, which only use the router of the tile for intra-application communication.

One of them is considered safety-critical (the blue one), and the other two are considered non-critical (see Table 4.1).

Table 4.1: Relative priorities and levels of criticality of the three applications.

	Color	Priority	Criticality
Application 1	Blue	Highest	Critical
Application 2	Green	Intermediate	Non-Critical
Application 3	Yellow	Lowest	Non-Critical

4.1.1 Mesh Topology

First, a 4 × 4 Network on Chip in a mesh topology is considered (Figure 4.2).

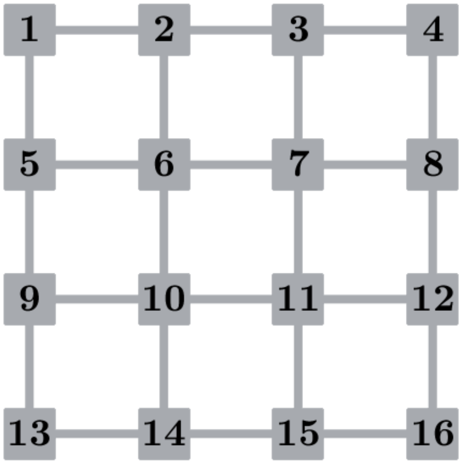


Figure 4.2: Network on Chip in a 4 × 4 mesh topology.

The following objective function is used (1):

Objective 1. $\max J = \sum_{i \in \{1,5,9,13\}} \sum_{k \in true\ nodes} X_{ik}^{CR \rightarrow apps}$

The objective is to maximize the number of "true" application nodes (i.e. nodes that use both the CR and the router of a tile) that are mapped on the leftmost tiles (tiles number $1, 5, 9, 13$ visible on Figure 4.2).

In this example, the architecture is considered homogeneous and all the Compute Resources and the NoC paths have the same type.

The first execution of the algorithm gives the following configuration (Figure 4.3):

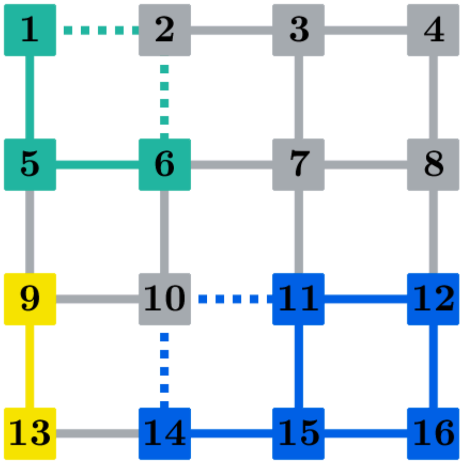


Figure 4.3: Result of the first execution of the algorithm. Spatial partitioning is enforced on the fabric.

Spatial partitioning is enforced on the fabric: no Compute Resources, routers or NoC communication paths are shared by different applications.

4.1.2 Toroidal Mesh Topology

The topology of the NoC on REDEFINE is a 4×4 toroidal mesh, as seen on Figure 2.3. The formulation also handles this topology. With this topology and the same objective 1, the first execution of the algorithm gives the configuration shown on Figure 4.4.

To facilitate the visualization of the NoC, only the wrap-around communication paths that are actually used by an application are shown on Figure 4.4.

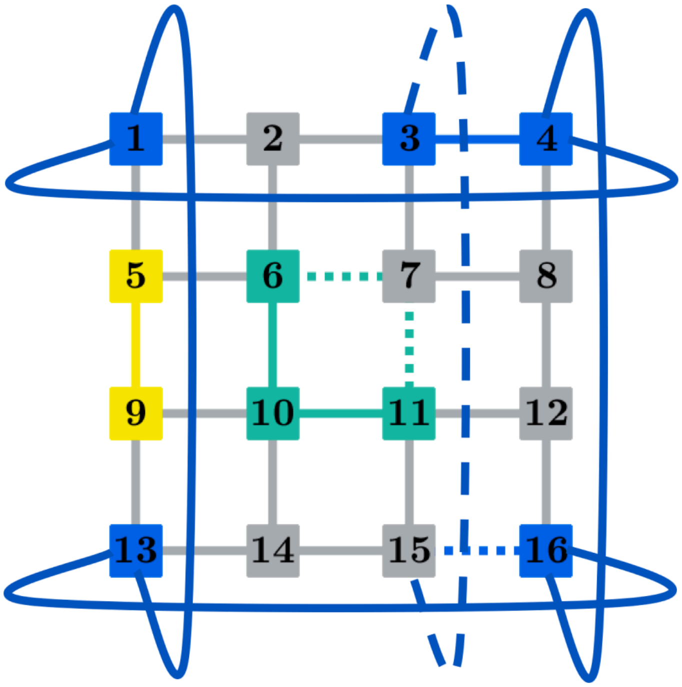


Figure 4.4: Result of the first execution of the algorithm. The blue application is mapped to the wrap-around NoC paths, and spatial partitioning is enforced on the fabric.

As seen on Figure 4.4, the safety-critical application (in blue) is using the wrap-around NoC paths. Spatial partitioning is still enforced on the fabric.

4.1.3 With a Cluster

In this example, a mesh 8 × 8 topology is considered and a cluster is defined. The upper left 5×5 square of Compute Resources is dedicated to the safety-critical application (in blue). The CRs of the cluster are represented by larger squares on Figure 4.5.

The result of the first execution of the algorithm is shown on Figure 4.6. The critical application (in blue) is allocated within the cluster while the non-critical applications are allocated outside of it.

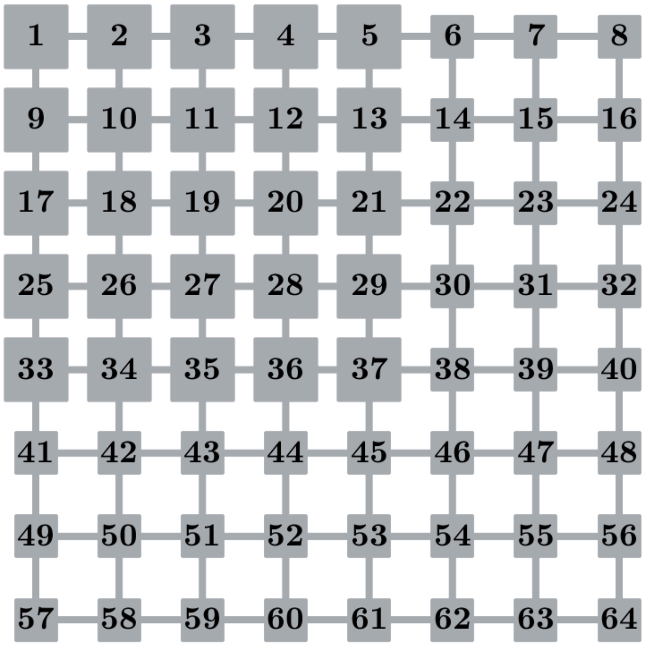


Figure 4.5: A mesh 8 × 8 topology. The cluster (upper left 5 × 5 square) is represented by larger squares.

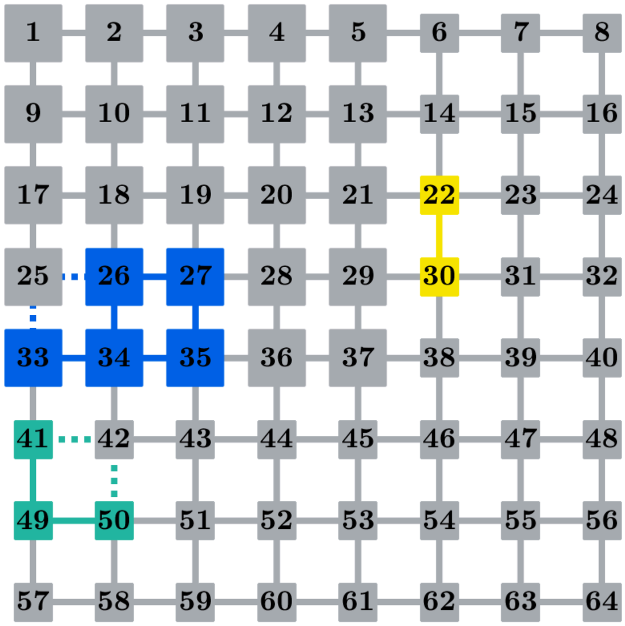


Figure 4.6: Result of the first execution of the algorithm. The cluster is dedicated to the safety-critical application (in blue).

4.2 Graceful Degradation

In this section, the behavior of the algorithm when hardware faults are detected is demonstrated. We consider the three same applications as in the previous section (see Figure 4.1 and Table 4.1).

The following representation of the health of the fabric tiles (Figure 4.7) is used:

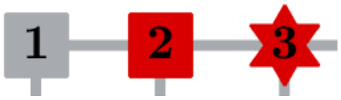


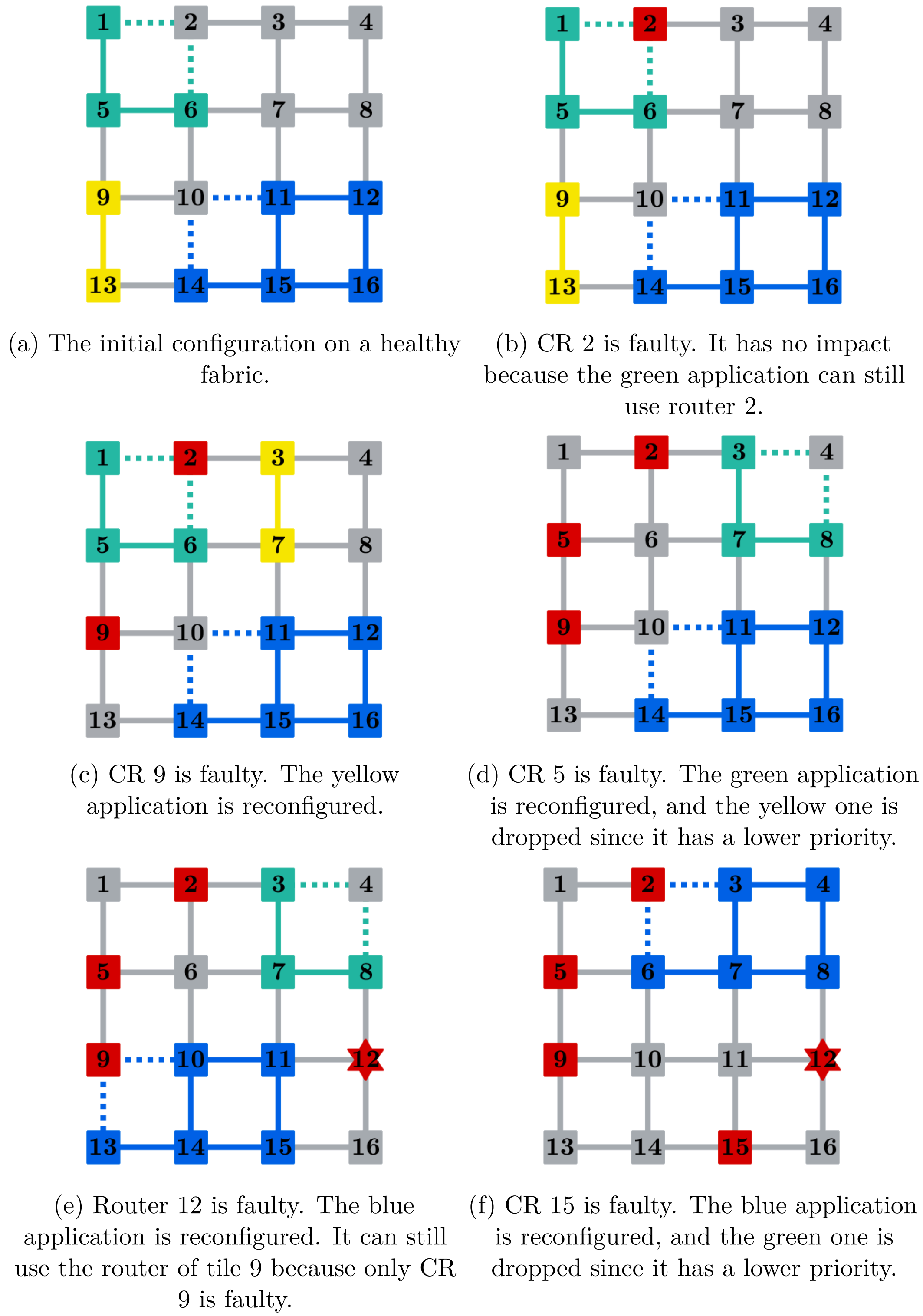
Figure 4.7: Tile number 1 is unused. Tile number 2 has a faulty Compute Resource. Tile number 3 has a faulty router.

As a reminder, a true application node can be mapped to a tile only if both the router and the Compute Resource are healthy. A ghost application node can be mapped to a tile as long as the router is healthy, regardless of the health of the CR.

4.2.1 Without a Cluster

In this subsection, we give an example of the reconfigurations that occur when hardware faults are detected on a Network on Chip in a mesh 4 × 4 topology.

The three applications can be mapped on the whole fabric since no cluster is defined. Figure 4.8 shows the consecutive reconfigurations on the fabric when several hardware faults occur.



(a) The initial configuration on a healthy fabric.

(b) CR 2 is faulty. It has no impact because the green application can still use router 2.

(c) CR 9 is faulty. The yellow application is reconfigured.

(d) CR 5 is faulty. The green application is reconfigured, and the yellow one is dropped since it has a lower priority.

(e) Router 12 is faulty. The blue application is reconfigured. It can still use the router of tile 9 because only CR 9 is faulty.

(f) CR 15 is faulty. The blue application is reconfigured, and the green one is dropped since it has a lower priority.

Figure 4.8: First scenario of graceful degradation on a mesh 4 × 4 fabric.

As seen on Figure 4.8, the task allocation algorithm allows for graceful degradation by reconfiguring the architecture when a hardware fault is detected. All along, spatial partitioning is enforced between the applications that are implemented on the fabric.

Figure 4.9 shows that when the safety-critical application can no longer be executed, the algorithm stops.

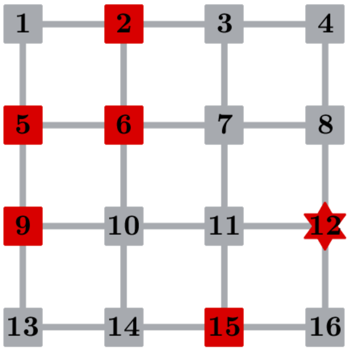
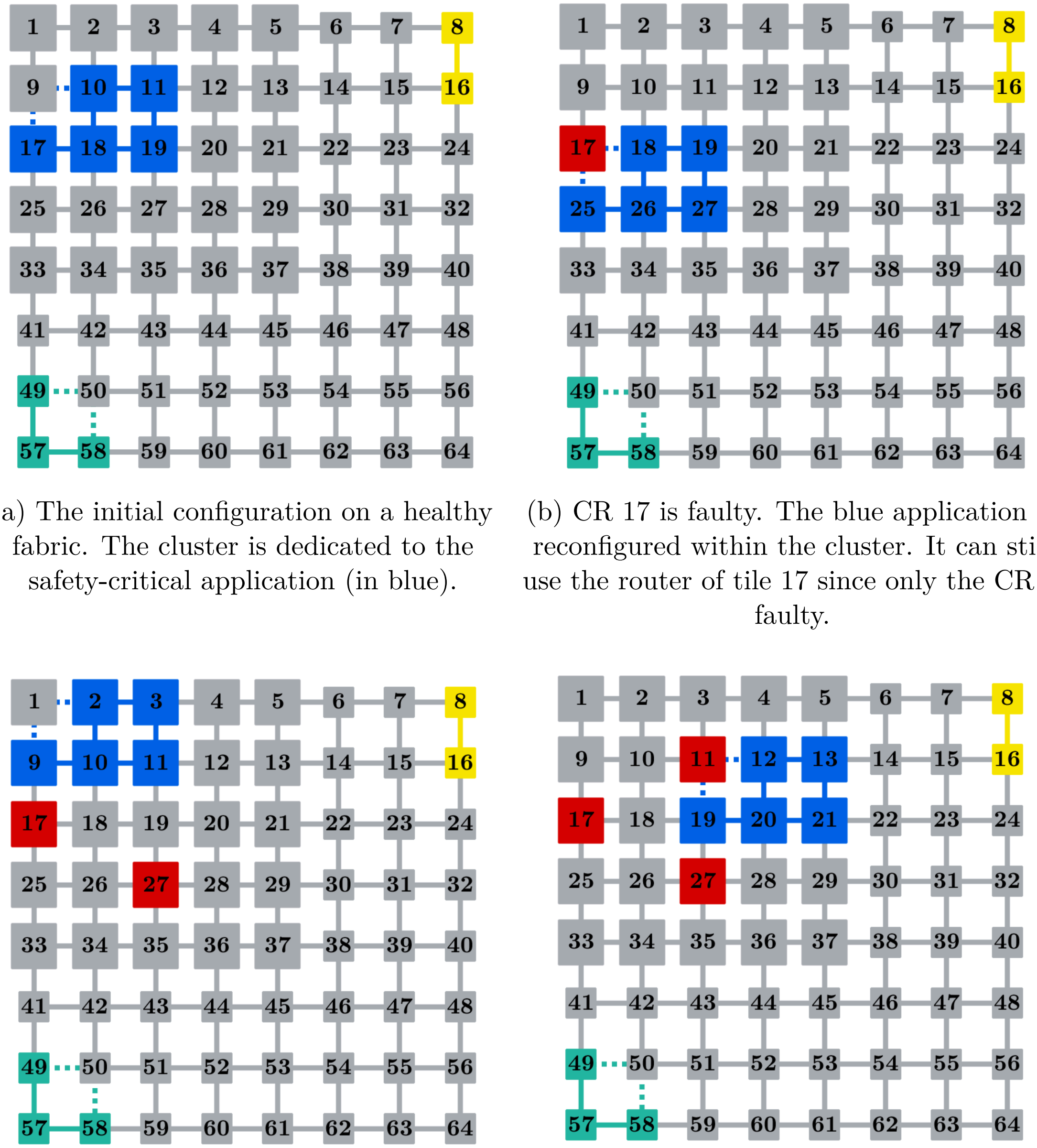


Figure 4.9: CR 6 is faulty. The safety-critical application can no longer be executed, and the algorithm stops.

4.2.2 With a Cluster

In this subsection, we give an example of the reconfigurations that occur when hardware faults are detected on a Network on Chip in a mesh 8 × 8 topology. A cluster dedicated to the safety-critical application is defined: it can only be mapped to a subset of fabric tiles. The cluster is consists of the upper left 5 × 5 square of Compute Resources, as seen on Figure 4.5 where it is represented by larger squares.

Figure 4.10 shows the consecutive reconfigurations on the fabric when several hardware faults occur.

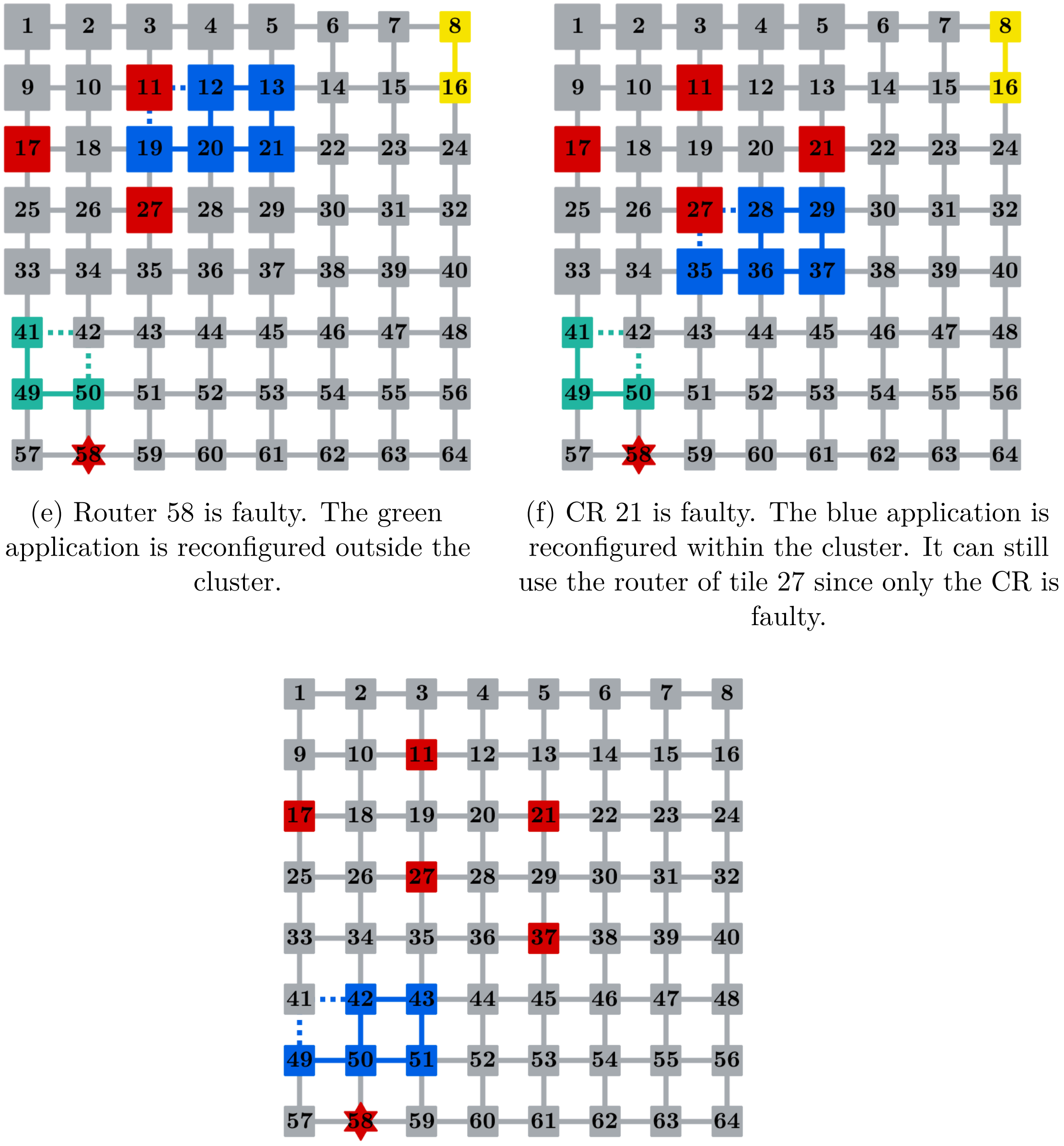


(a) The initial configuration on a healthy fabric. The cluster is dedicated to the safety-critical application (in blue).

(b) CR 17 is faulty. The blue application is reconfigured within the cluster. It can still use the router of tile 17 since only the CR is faulty.

(c) CR 27 is faulty. The blue application is reconfigured within the cluster.

(d) CR 11 is faulty. The blue application is reconfigured within the cluster. It can still use the router of tile 11 since only the CR is faulty.



(e) Router 58 is faulty. The green application is reconfigured outside the cluster.

(f) CR 21 is faulty. The blue application is reconfigured within the cluster. It can still use the router of tile 27 since only the CR is faulty.

(g) CR 37 is faulty. The safety-critical application cannot be reconfigured inside the cluster, so all the non-critical applications are dropped. From now on, the critical application can be reconfigured on the whole fabric.

Figure 4.10: Second scenario of graceful degradation on a mesh 8 × 8 fabric, with a subset of fabric tiles dedicated to the safety-critical application (in blue).

As seen on Figure 4.10, the task allocation algorithm enforces spatial partitioning on the Network on Chip through the reconfigurations. It also maps the safety-critical application (in blue) to the cluster as long as it is possible. If it is not, all the non-critical applications are dropped and the critical application is mapped anywhere on the fabric, provided that enough Compute Resources and routers are healthy.

4.3 Fault Tolerance

In this section, the number of hardware faults the task allocation algorithm can handle until the safety-critical application can no longer be executed is studied.

We generate 100 sequences of hardware faults to simulate the degradation of the architecture. They can be either router faults or Compute Resource faults. We then launch the algorithm on those sequences to study its results and efficiency.

For those trials, the three applications described on Figure 4.1 ar considered, with the priorities shown in Table 4.1.

4.3.1 4 × 4 Fabric

First, a 4 × 4 fabric is considered. Both the mesh and the mesh toroidal topologies are studied.

As seen on Figure 4.11, for certain sequences, the two topologies can handle the same number of hardware faults. Furthermore, the toroidal mesh topology is either equally or more resilient than the mesh topology. This result is consistent with the fact that the wrap-around NoC paths of the toroidal mesh offer more mapping solutions for the applications links.

On average, the toroidal mesh topology tolerates 7.32 hardware faults while the mesh topology tolerates 5.50 hardware faults.

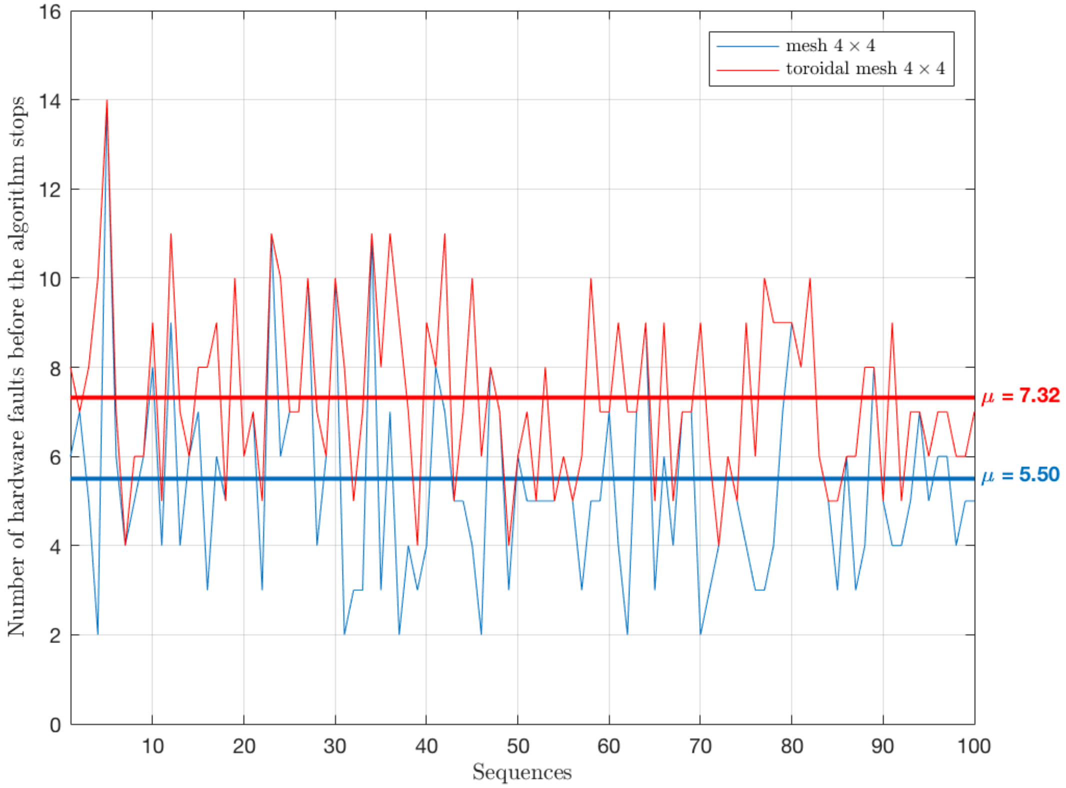


Figure 4.11: Number of hardware faults before the safety-critical application can no longer be executed on the 4 × 4 fabric, for the 100 trials. Mesh topology in blue and toroidal mesh topology in red.

4.3.2 8 × 8 Fabric

A 8 × 8 fabric is now considered. Both the mesh and toroidal mesh topologies are studied.

As seen on Figure 4.12, the results obtained for the 8 × 8 fabric are similar to those obtained for the 4 × 4 fabric. The toroidal mesh fabric tolerates more hardware faults than the mesh fabric.

On average, the toroidal mesh topology tolerates 41.20 hardware faults while the mesh topology tolerates 38.57 hardware faults. The larger size of the fabric allows for more reallocation possibilities than in the 4 × 4 case when faults occur.

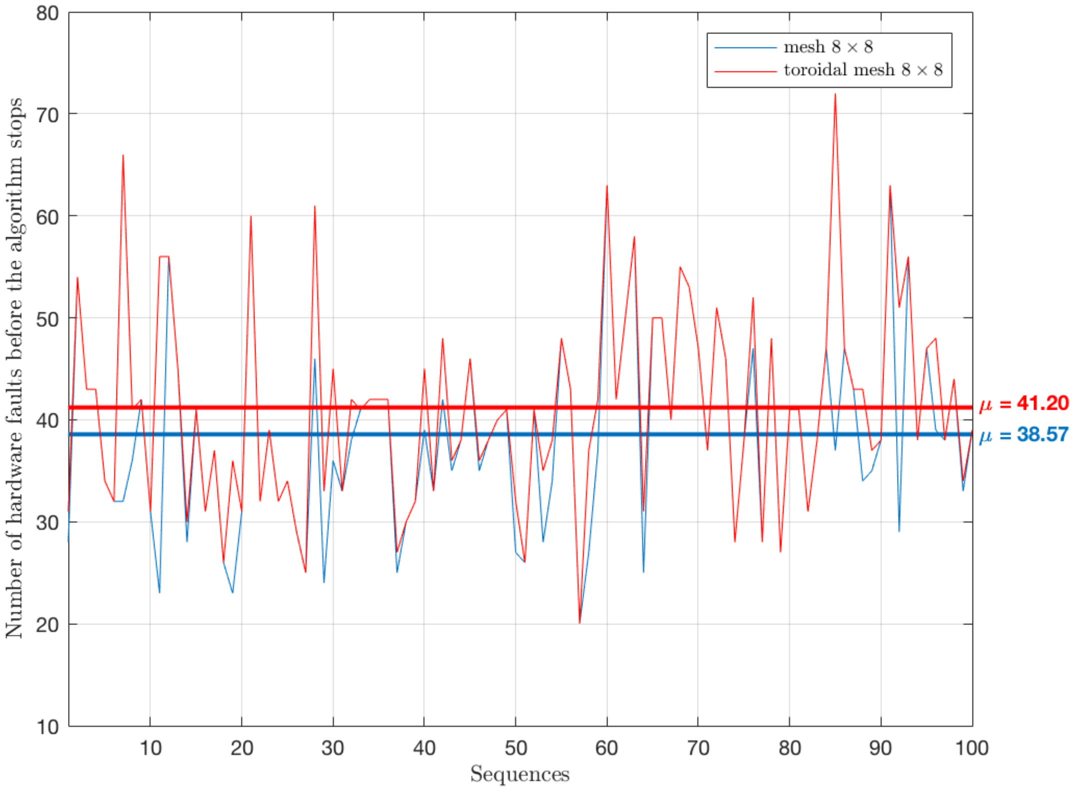


Figure 4.12: Number of hardware faults before the safety-critical application can no longer be executed on the 8 × 8 fabric, for the 100 trials. Mesh topology in blue and toroidal mesh topology in red.

4.4 Computation Time

In this section, the computation time needed by each call of the solver is studied. As the reconfigurations must take place in real-time when hardware faults are detected, the algorithm must be efficient enough to compute the reconfigurations.

The computation time is measured on an Apple MacBook Pro (CPU 3.1 GHz Intel Core i7). The problem is modeled on MATLAB using the modeling framework CVX, and solved with Gurobi. The same 100 sequences of hardware faults and the same three applications as in the previous section are used.

4.4.1 4 × 4 Fabric

First, a 4 × 4 fabric is considered. Both the mesh and the mesh toroidal topologies are studied.

The first solver call is not shown on Figure 4.13 since it corresponds to the initial allocation of the applications on the healthy fabric. We are interested in studying the reconfiguration overhead in case of hardware faults.

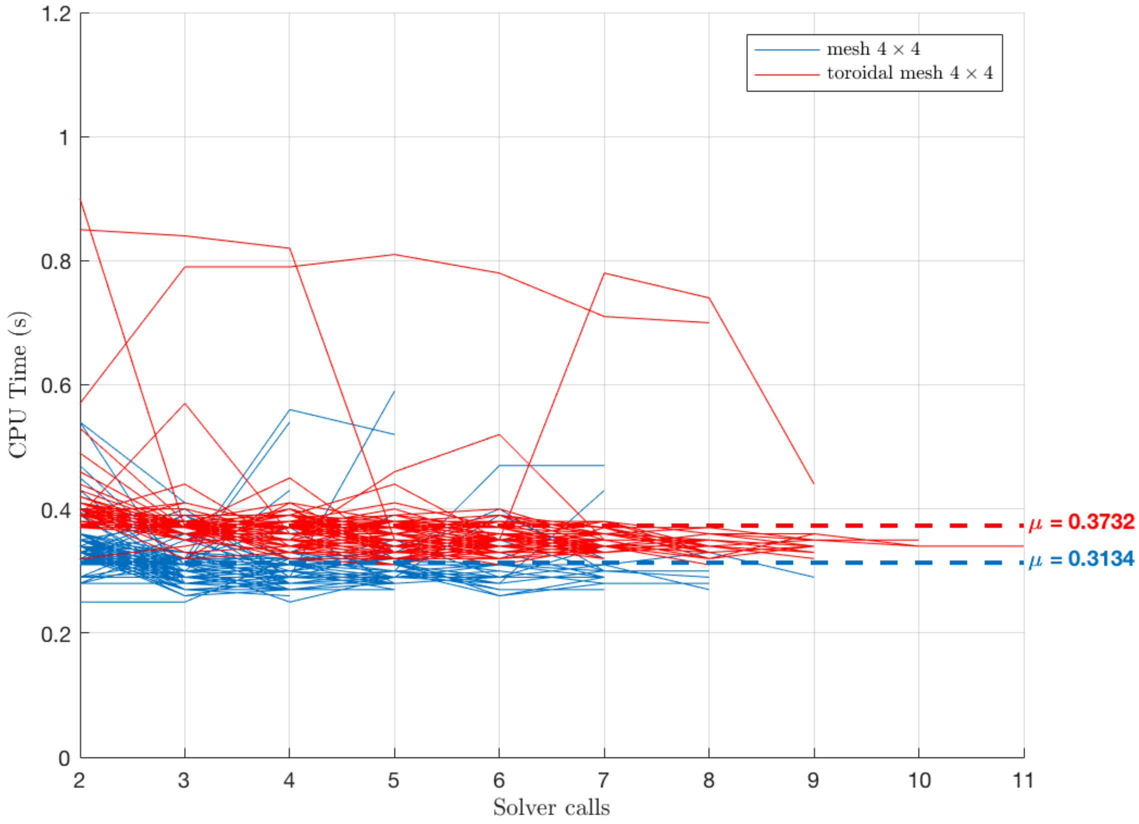


Figure 4.13: CPU time used by each solver call for the 100 trials, on a 4 × 4 fabric. Mesh topology in blue and toroidal mesh topology in red.

As seen on Figure 4.13, on average solving the optimization problem takes more time in the case of a toroidal mesh topology. Indeed, the wrap-around NoC paths of the toroidal mesh add binary decision variables to the problem.

4.4.2 8 × 8 Fabric

A 8 × 8 fabric is now considered. Both the mesh and the mesh toroidal topologies are studied.

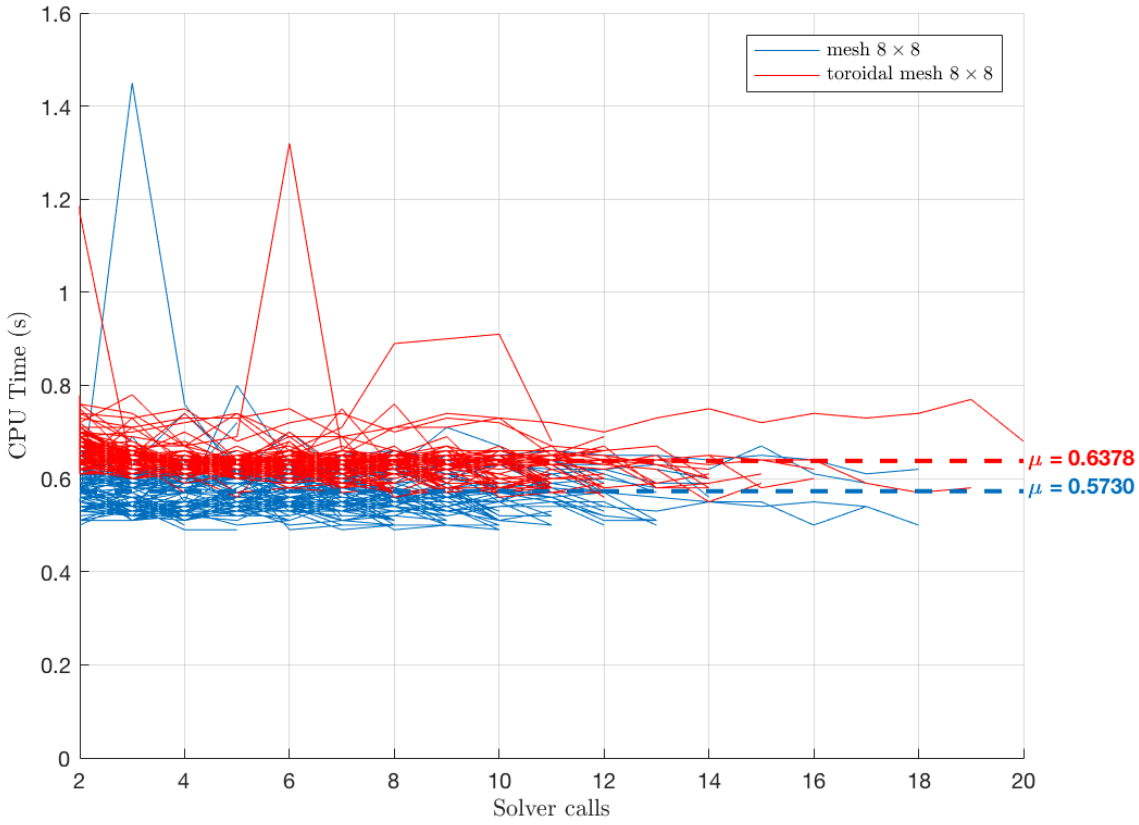


Figure 4.14: CPU time used by each solver call for the 100 trials, on a 8 × 8 fabric. Mesh topology in blue and toroidal mesh topology in red.

As seen on Figure 4.14, the results are similar for a 4 × 4 topology: on average solving the optimization problem takes more time in the toroidal mesh case.

Moreover, solving the optimization problem takes on average 82.8% more computation time in the mesh 8 × 8 case than in the mesh 4 × 4 case. Similarly, it takes on average 70.9% more computation time in the toroidal mesh 8 × 8 case than in the toroidal mesh 4 × 4 case. This is coherent, as more decision variables are required to describe the optimization problem in the 8 × 8 cases.

Overall, the computation time required for the solver calls is well under the second for almost every reconfiguration in the considered scenarios. This time strongly depends on the formulation of the optimization problem, the ILP solver and the platform on which it is implemented. It could therefore be reduced. The results nevertheless show that solving the problem as it was formulated is not computationally heavy, and that this approach is suitable for real-time purposes.

In a real implementation, in case of a detected hardware fault, the system would switch to a healthy redundant system, while the faulty one is reconfigured. It could then still be used as one of the redundant systems, despite its degraded health.

CHAPTER 5
CONCLUSION

5.1 Summary

This thesis aimed at studying a reconfigurable multi-core architecture, discussing its suitability for executing safety-critical embedded software and developing a task allocation algorithm to enforce certain constraints on its Network on Chip (NoC).

First, the REDEFINE architecture developed at the Indian Institute of Science was presented. Its high-level structure, compiler, Network on Chip, memory management and execution model were described. From this analysis, it was argued that the architecture is suitable to execute safety-critical applications, provided that the allocation of the applications meets certain constraints, in particular spatial partitioning on the NoC. In addition, its dynamic features allow for graceful degradation of the system.

Secondly, a centralized task allocation algorithm for REDEFINE was proposed. The allocation problem was formulated as an Integer Linear Programming (ILP) optimization problem. Under the assumption that hardware faults are detected, it manages the reallocation of the different applications when faults occur. The algorithm enforces spatial partitioning on the fabric throughout the reconfigurations. It supports multiple types of NoC topologies, constraints and hardware faults.

Finally, the behavior of the algorithm in different configurations and its efficiency were discussed. Several topologies, sets of constraints and hardware faults scenarios were considered. The performance of the algorithm in terms of computation time indicates that its use in a real-time environment is possible.

5.2 Future Work

In this section, various research leads emerging from the work achieved are discussed.

First, for the development of the presented task allocation algorithm, it was assumed that the hardware faults are detected when they occur, and that the information is made available to the Resource Manager. Therefore, a health monitoring system must also be designed for the architecture. To that purpose, hardware fault models must be defined.

Secondly, the proposed algorithm enforces spatial partitioning on the NoC between the applications at runtime. However, the channels used for input/output (I/O) communication were not considered in this work. The communication between the sensors, the actuators and the applications must also meet real-time constraints, and this constitutes a possibility for future work.

Furthermore, in this work, a centralized task allocation algorithm designed to be implemented at the Resource Manager level was developed. With this approach, the Resource Manager constitutes a single point of failure of the architecture. Consequently, there is interest in developing a decentralized task allocation algorithm.

REFERENCES

[1] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[2] J. Parkhurst, J. Darringer, and B. Grundmann, “From single core to multicore: Preparing for a new exponential,” in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ACM, 2006, pp. 67–72.

[3] 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.

[4] 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.

[5] L. M. Kinnan, “Use of multicore processors in avionics systems and its potential impact on implementation and certification,” in *Digital Avionics Systems Conference, 2009. DASC’09. IEEE/AIAA 28th*, IEEE, 2009, 1–E.

[6] P. Huyck, “ARINC 653 and multi-core microprocessors – Considerations and potential impacts,” in *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, IEEE, 2012, 6B4–1.

[7] “DO-178C: Software Considerations in Airborne Systems and Equipment Certification,” *Radio Technical Commission for Aeronautics Inc. (RTCA)*, 20012.

[8] C. B. Watkins and R. Walter, “Transitioning from federated avionics architectures to integrated modular avionics,” in *Digital Avionics Systems Conference, 2007. DASC’07. IEEE/AIAA 26th*, IEEE, 2007, 2–A.

[9] P. Manolios, D. Vroon, and G. Subramanian, “Automating component-based system assembly,” in *Proceedings of the 2007 international symposium on Software testing and analysis*, ACM, 2007, pp. 61–72.

[10] W. River, “ARINC 653 – an Avionics Standard for safe, partitioned Systems,” in *IEEE Seminar*, 2008.

[11] P. J. Prisaznuk, “Integrated modular avionics,” in *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*, IEEE, 1992, pp. 39–45.

[12] L. Rierson, *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance.* CRC Press, 2013.

[13] R. John, “Partitioning in avionics architectures: Requirements, mechanisms, and assurance,” 1999.

[14] “DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations,” *Radio Technical Commission for Aeronautics Inc. (RTCA)*, 2005.

[15] R. Wolfig and M. Jakovljevic, “Distributed IMA and DO-297: Architectural, communication and certification attributes,” in *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, IEEE, 2008, 1–E.

[16] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert, “Ensuring robust partitioning in multicore platforms for IMA systems,” in *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, IEEE, 2012, 7A4–1.

[17] “Multi-Core Processors,” *Position Paper, Certification Authorities Software Team, CAST-32A, FAA*, November 2016.

[18] X. Jean, M. Gatti, G. Berthon, and M. Fumey, “MULCORS: use of multicore processors in airborne systems,” *European Aviation Safety Agency, Industrial report*, December 2012.

[19] T. J. Todman, G. A. Constantinides, S. J. Wilton, O. Mencer, W. Luk, and P. Y. Cheung, “Reconfigurable computing: Architectures and design methods,” *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[20] J. Williams, C. Massie, A. D. George, J. Richardson, K. Gosrani, and H. Lam, “Characterization of fixed and reconfigurable multi-core devices for application acceleration,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, p. 19, 2010.

[21] R. Hartenstein, “Coarse grain reconfigurable architectures,” in *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, IEEE, 2001, pp. 564–569.

[22] 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.

[23] T. Guillaumet, A. Sharma, E. Feron, M. Krishna, R. Narayan, P. Baufreton, F. Neumann, and E. Grolleau, “Using reconfigurable multi-core architectures for safety-critical embedded systems,” in *Digital Avionics Systems Conference (DASC), 2016 IEEE/AIAA 35th*, IEEE, 2016, pp. 1–6.

[24] M. Krishna, K. T. Madhu, N. S., S. Das, C. Haldar, R. Narayan, and S. K. Nandy, “ReNE: combating dark silicon in polymorphic massively parallel processing cores,” *Poster presentation at ICCAD 2015, Austin, TX*,

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

[26] G. Garga, S. Das, S. Nandy, R. Narayan, C. Haldar, M. P. Jagtap, and S. P. Dash, “A Flexible Crypto-system Based upon the REDEFINE Polymorphic ASIC Architecture,” *Defence Science Journal*, vol. 62, no. 1, p. 25, 2012.

[27] R. D. Blumofe, M. Frigo, C. E. Joerg, C. E. Leiserson, and K. H. Randall, “Dag-consistent distributed shared memory,” in *Parallel Processing Symposium, 1996., Proceedings of IPPS’96, The 10th International*, IEEE, 1996, pp. 132–141.

[28] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, “An ILP formulation for task mapping and scheduling on multi-core architectures,” in *2009 Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2009, pp. 33–38.

[29] M. Srinivasan and G. De Micheli, “Bandwidth-Constrained Mapping of Cores onto NoC Architectures,” in *Design, Automation, and Test in Europe: Proceedings of the conference on Design, automation and test in Europe-*, vol. 2, 2004.

[30] “Gurobi optimizer reference manual,” *Gurobi Optimization, Inc.*, 2016.

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

[32] M. Grant and S. Boyd, “Graph implementations for nonsmooth convex programs,” in *Recent Advances in Learning and Control*, ser. Lecture Notes in Control and Information Sciences, V. Blondel, S. Boyd, and H. Kimura, Eds., `http://stanford.edu/~boyd/graph_dcp.html`, Springer-Verlag Limited, 2008, pp. 95–110.