N° d'ordre : 3970 ANNÉE 2010

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et Télécommunications

École doctorale MATISSE

présentée par

Hung-Manh Pham

préparée à l'unité de recherche UMR6074 IRISA
Institut de recherche en informatique et systèmes aléatoires - CAIRN
École National Supérieure des Sciences Appliquées et de Technologie

---

Embedded computing architecture with dynamic hardware reconfiguration for intelligent automotive systems

Thèse soutenue à Rennes
le 15 Decembre 2010

devant le jury composé de :

Jean ARLAT
Directeur de recherche CNRS
LAAS / examinateur

Michel RENOVELL
Directeur de recherche CNRS
LIRMM / rapporteur

Guy GOGNIAT
Professeur, Université de Bretagne-Sud / rapporteur

Stanisław J. PIESTRAK
Professeur, Université de Metz / examinateur

Didier DEMIGNY
Professeur, Université de Rennes 1/ directeur de thèse

Sébastien PILLEMENT
Maître de Conférences,
Université de Rennes 1 / co-directeur de thèse

"When you are courting a nice girl, an hour seems like a second. When you sit on a red-hot cinder, a second seems like an hour. That's relativity."

—Albert Einstein

Acknowledgements

I take this opportunity to acknowledge and thank all those people who supported me during my PhD.

First of all, I would like to thank first my advisor Prof. Dr. Didier Demginy for his support throughout my PhD work. His suggestions and remarks have always been productive and constructive that are invaluable for my research. I would like to express my sincere gratitude to my co-advisor Dr. Hab. Sébastien Pillement. His continuous support during the PhD work, his constructive feedbacks, comments and suggestions at various stages and his gentleness have been significantly useful in shaping the thesis up to the completion. Without any of these, it would not have been possible to complete this PhD thesis.

I also want to thank all the members of the jury who evaluated this work: Jean Arlat (Director of Research CNRS, LAAS, Toulouse), Michel Renovel (Director of Research CNRS, LIRMM, Montpellier), Guy Gogniat (professor at Université de Bretagne-Sud) and Stanisław J. Piestrak (Professor at University of Metz).

I wish to thank all the member of IRISA/CAIRN Lab who make my stay a memorial period. It is a great pride for me completing the PhD thesis in this dynamic Lab. That is the reason I want to thank the head of Lab Prof. Dr. Oliver Sentieys for his kind attitude, generous help and prestigious suggestions during my PhD thesis. I am equally thankful to Prof. Dr. Stanisław J. Piestrak for his advices, his scientific open-mind and his politeness that are really helpful for my research activity.

I am equally thankful to the partners of the CIFAER project for all the scientific exchanges and discussions. I am extremely grateful to the Conseil Général de Côtes d'Armor (CG22) and the Conseil Régional de Bretagne for their finance supports of this thesis.

Finally, I would like to thank my parents for their never ending supports, that help me stay concentrated on my thesis.

Contents

List of Figures

List of Tables

Résumé en français

Problématique

Dans les domaines tels que le militaire, l'aérospatial, l'automobile et le médical, les applications critiques nécessitent la mise en place de stratégies de tolérance aux fautes pour assurer le fonctionnement fiable du système. En effet dans ces domaines, une faute pendant le fonctionnement de ces systèmes peut mettre en danger la vie humaine ou l'environnement. Les exigences de fiabilité posent de nouveaux défis pour la conception de systèmes embarqués électroniques. L'industrie automobile a observé une transition importante de la mécanique vers la mécatronique. Les contrôles mécaniques sont de moins en moins présents et sont remplacés par des composants électroniques (Contrôle X-by-wire) appelé ECU (unité de contrôle électronique). L'omniprésence des calculateurs a conduit l'industrie de l'électronique automobile à faire face aux mêmes défis des exigences de sécurité que le domaine spatial.

Les solutions classiques pour améliorer la fiabilité et la sécurité s'appuient sur la multiplication des ressources et la notion de redondance des éléments de traitement. Le fonctionnement correct de l'ensemble du système est assurée par le fait que les erreurs ne concernent qu'une seule instance du système. Le principe de la redondance permet d'identifier une instance du circuit qui donne un résultat différent, permettant de fait d'identifier un fonctionnement anormal. Ces techniques de tolérance aux fautes ont un impact fort en terme de coût sur la conception des systèmes. La redondance matérielle a évidemment besoin de plus de ressources matérielles pour mettre en œuvre une fonction. Hormis les questions de coût, la redondance des ressources de calcul pose le problème du taux d'utilisation des circuits. En effet, la multiplication des modules peut limiter la

mise en œuvre d'autres modules dans une plateforme donnée et donc limiter la puissance de traitement de l'ensemble du système.

Dans le même temps, les applications sont de plus en plus complexes pour les systèmes électroniques en général et pour les contrôles électroniques de l'automobile en particulier. Cette tendance induit la conception de systèmes de calcul parallèles pour résoudre des problèmes complexes avec des contraintes de temps fortes. Un système de calcul parallèle est constitué de différents éléments de traitement (PE pour Processing Element) qui travaillent en collaboration pour résoudre un problème. Les systèmes embarqués, qui contiennent des processeurs combinant la flexibilité de la programmation logiciel avec la puissance de traitement des composants matériels dédiés, ne sont plus utilisés comme simple contrôleurs. Ces systèmes doivent offir de plus en plus de puissance de calcul pour satisfaire les demandes, comme l'encodage/décodage audio/vidéo, le traitement d'image, etc. et les systèmes multiprocesseurs sur puce (MPSoC) sont une option pour faire face à cette augmentation des besoins en performance. Cette approche offre une certaine flexibilité grâce à la reprogrammation du logiciel, tout en supportant de haute performance grâce à l'exécution de nombreuses fonctionnalités en parallèle.

Les récentes évolutions technologique au sein de l'industrie des semiconducteurs permettent aux fabricants d'équipement de développer de nouvelles technologies pour améliorer la qualité, la performance et la fiabilité. Des fonctions spécifiques sont souvent mises en œuvre dans les circuits intégré spécifique à l'application (ASIC). Cependant ces circuits n'offrent aucune flexibilité permettant une adaptation des circuits post-fabrication.

A côté des exigences accrues de performance, de qualité et de fiabilité, l'industrie électronique doit de plus travailler sur la réduction du coût du produit. L'attente sur la réduction des coûts oblige les fournisseurs à développer de nouvelles technologies tout en améliorant l'efficacité de fabrication. La solution ASIC nécessite souvent un important délais de conception et des coûts de fabrication prohibitif pour des circuits en petite série. Ceci est en contradiction avec les tendances industrielles. Une solution aux problèmes de coût et de besoin en flexibilité est l'utilisation de circuits reconfigurable de type Field Programmable Gate Arrays (FPGA).

Les fournisseurs de FPGA ont profité des dernières évolutions technologiques pour offrir

des circuits qui peuvent inclure des millions de portes programmable intégrées avec plusieurs méga-octets de mémoire interne et des cœurs de processeur adaptés à la personnalisation d'une grande variété d'applications. La combinaison de propriétés intellectuelle réutilisables (IPs), de faibles coûts unitaires et une relative facilité de mise en œuvre ont entraîné une utilisation grandissante des FPGA dans les domaines critiques comme l'automobile par exemple.

Hormis les aspects financier, les FPGA permettent aussi une mise oeuvre et des temps de développement très rapide. L'utilisation de FPGA peut permettre d'introduire rapidement de nouvelles fonctionnalités ou d'adapter les systèmes après leur déploiement.. Cette évolution est supportée par les accords entre les fabricants de circuits FPGA et des fournisseurs tiers afin d'offrir des IPs dédié, comme les contrôleurs de communication ciblées spécifiquement aux applications automobiles (CAN, MOST, FlexRay, etc.). D'un autre point de vue les FPGA offrent des niveaux de performances très supérieur à l'utilisation de processeur classique. Ainsi ces technologies sont de plus en plus utilisées dans des applictions critiques, mais nécessite la prise en comptes de nouvelles contraintes lors de la conception des systèmes.

Il est maintenant reconnu que les circuits logiques reconfigurables répondent aux exigences de performances de traitement et au défi de l'évolutivité des applications. Le principal avantage de ce type de solution réside dans la possibilité de ne pas augmenter systématiquement les ressources de calcul, mais plutôt de profiter de la généricité de l'équipement proposé pour optimiser l'utilisation des ressources, notamment en fonction des temps creux.

Contrairement à un ASIC, où le coût de fabrication depasse le million de dollars, le coût de conception d'un FPGA est relativement faible.

Ces temps de développements faibles permettent alors d'itérer plusieurs fois afin de concevoir un système complexe répondant aux contraintes des applications. En outre, les FPGA modernes supportent la reconfiguration dynamique partielle, qui est une fonctionnalité avancée permettant de renforcer encore la flexibilité et, éventuellement, la fiabilité du système ciblé. La reconfiguration dynamique partielle permet aux modules de partager temporairement des ressources physiques. Cette caractéristique permet de modifier la fonctionalité d'une partie du FPGA tandis que les autres blocs continuent de fonctionner. En utilisant la reconfiguration partielle, les concepteurs peuvent considérablement

augmenter les fonctionnalités d'un seul FPGA, permettant de mettre en œuvre un système avec des circuits plus petits. Exploiter la reconfiguration partielle permet de mettre en œuvre les techniques traditionnelles de tolérance aux fautes en utilisant les ressources reconfigurables disponibles comme support à la redondance classique. Par ailleurs, la reconfiguration dynamique apporte des avantages supplémentaire aux approches classiques de tolérance aux fautes. En effet, la reconfiguration dynamique partielle facilite la récupération du système aprés apparition d'une faute par, resynchronisation partielle du module défectueux sans perturber les autres modules, ou en modifiant le système afin de migrer les tâches du module fautif. En conclusion, non seulement les FPGA introduisent un plus haut niveau de flexibilité, mais ils permettent aussi une continuité de service plus important. Grâce à l'aspect dynamique et partiel de la reconfiguration des FPGA, le fonctionnement du système est garanti sans interruption, ce qui peut faciliter le passage en mode dégradé avec le maintien services les plus essentiels.

Une tendance actuelle est d'exploiter la reconfiguration dynamique partielle dans les systèmes multi-processeurs pour créer des MPSoCs dynamiques. Cette coordination introduit un nouveau degré de liberté à la conception du système et au comportement du systèmependant l'exécution. La flexibilité est alors supportée par une approche logicielle, comme dans les MPSoC classiques, mais le matériel peut aussi être adapté en cours d'exécution pour supporter des changements de conditions d'exécution. Cette adaptation de l'architecture matérielle pendant l'exécution offre au concepteur un nouveau degré de flexibilité afin d'assurer une répartition optimisée des tâches de calcul sur les cellules de traitement et de répondre aux contraintes de performances, de puissance et de consommation de surface.

Objectifs

Dans ces travaux de thèse, nous proposons de démontrer l'intérêt des nouvelles technologies reconfigurables dynamiquement dans le domaine de l'automobile et plus généralement dans le domaine de la sûreté de fonctionnement. L'utilisation de calculateurs reconfigurables dynamiquement a un double apport :

1. la **réduction du nombre de calculateurs** diminuant ainsi les coûts de mise en œuvre, et permettant d'offrir une certaine redondance du matériel. En outre

ces architectures offrent un support d'exécution unifié pour l'ensemble des tâches à exécuter, qu'elles soient critiques ou non. Le taux d'occupation du matériel est amélioré par réutilisation des calculateurs pendant leurs périodes oisives ;

2. le support efficace des approches classiques et industrielles de **la sûreté de fonctionnement**. En réalisant une détection de fautes sur les calculateurs reconfigurables et en la couplant avec un mécanisme de migration des tâches (matérielles et logicielles), il est alors possible d'augmenter de façon significative la robustesse du système, tout en conservant des performances optimales. Nous utiliserons dans ces travaux les techniques classiques pour les aspects détections de fautes.

Le premier point dur vient de l'étude des architectures matérielles des calculateurs reconfigurables, compatible avec les contraintes de coût liées au domaine, ainsi qu'aux contraintes de complexité des algorithmes et de temps-réel. Le deuxième point dur concernera alors l'interface entre le calculateur et le véhicule si l'on implémente de la migration de tâches suite à la détection d'une faute.

Sur la base de l'architecture matérielle développée et de son logiciel de contrôle, un démonstrateur mettant en œuvre la reconfiguration dynamique partielle, ainsi que les services gérant la sûreté de fonctionnement, a été réalisé durant la thèse.

Contributions

En général, les mécanismes de tolérance aux fautes nécessitent des ressources matériels supplémentaires pour supporter la redondance ce qui limite le nombre d'éléments de traitement implémentable et par conséquent, limiter la capacité du système. Ainsi, des défis émergent de l'application de techniques de tolérance aux fautes peuvent réduire la puissance de calcul du système final. Durant cette thèse, nous avons maîtrisé la reconfiguration dynamique partielle et nous l'avons combiné avec des techniques de tolérance aux fautes. La reconfiguration dynamique, d'une part, contribue à améliorer la fiabilité du système et à améliorer les performances en minimisant l'impact de la tolérance aux fautes, d'autre part, elle permet d'accroître la flexibilité en incluant la possibilité de changer la fonctionalité des circuist à la demande.

Prémièrement nous introduisons un système multi-processeur complétement dynamique et tolérant aux fautes (FT-DyMP SoC). Ce système implémente l'ensemble des caractéristiques d'un MPSoC (performance et flexibilité) et intègre également des mécanismes de tolérance aux fautes. La détection de fautes est réalisée à différents niveaux : au niveau de chaque processeur par l'implémenttion de mécanisme de type LockStep et au niveau système par l'implémentation d'une matrice de connexion. Tous les processeurs dans FT-DyMPSoC sont dynamiques, c'est à dire que n'importe quel processeur peut être reconfiguré par un autre processeur en cas d'erreurs détéctées. L'originalité de notre système vient du fait qu'il n'y a aucun processeur statique comme dans les autres systèmes. Ce processeur est alors un point faible du point de vue de la tolérance aux fautes. Une autre difficulté provient de la construction du système. En effet, il faut pouvoir reconfigurer l'ensemble des processeurs incluant l'ensemble de leurs périphériques. Cette approche n'est actuellement pas supporté par le flot de conception classique proposé par le fabricant de circuit. Nous introduisons donc une modification du flot de conception qui facilite la création de processeurs entièrement dynamique. Le flot modifié nécessite la définition du concept de *wrappers* et *sockets*. Le support de ces composants (*wrappers* et *sockets*) permet de simplifier le processus de synthèse lors de la construction de systèmes complexes. Ils permettent de palier au manque des outils en introduisant des connexions virtuelles entre les processeurs et les périphériques dynamiques du système.

L'implémentation de méthodes de tolérances aux fautes dans les systèmes embarqués s'accompagnent toujours de surcoût matériel et/ou temporel. Il est donc important de pouvoir évaluer au plus tôt les pénalités introduites par ce type d'approches. Ainsi, un modèle analytique est proposé pour aider à accélérer l'évaluation du compromis performance/fiabilité tout en incluant la technique de tolérance aux fautes dans les systèmes cibles. Nous avons appliqué notre modèle à notre système mais aussi à une technique classique de la tolérances aux fautes, le *scrubbing*. Cette technique vise à reconfigurer régulièrement le FPGA afin d'éviter les changements de valeur des bits de configuration. Les résultats obtenus et validés montre que notre système supporte un meilleur compromis performance/fiabilité que le *scrubbing*. En outre, notre modèle n'est pas seulement limité aus systèmes utilisant des architectures reconfigurable, il peut être utilisé de manière plus générale pour des système tolérants aux fautes en adaptant les paramètres à l'architecture cible.

La duplication du matériel présente un avantage en terme de surcout matériel par rapport à la triplication qui nécessite de reproduire trois fois un même chemin de données. A contrario la duplication ne permet pas de corriger une faute dans le système mais ne permet que la détection de cette dernière. Dans nos travaux nous avons alors proposer un système utilisant la reconfiguration dynamique permettant de corriger les fautes dans un système lockstep (duplication) de processeur reconfigurable dynamiquement. Le système lockstep contient deux copies d'un même processeur capable de détecter les erreurs grâce à un indicateur de différence. Notre proposition renforce le lockstep en ajoutant la capacité de localiser la faute et de ce fait de permettre une correction. Un moteur de configuration (Configuration engine) proposée surveille le système en arrière-plan. Ce composant permet par relecture des frames de configuration et par calcul de CRC d'identifier la frame fautive éventuelle permettant de locaiser l'erreur. Nous avons développé le COMP_MUX (Comparateur/Multiplexer) qui contien un comparateur standard permettant de détecter toute incohérence entre les deux processeurs identiques, dans cette éventualité l'opération de localisation des fautes du configuration engine est lancé. Après localisation de la faute, la sortie du processeur non fautif est immédiatement passée à la sortie finale grâce au multiplexeur , de ce fait les sorties fausses sont bloquées mais le système continue d'être opérationnel. Les composants ajoutés au système (COMP_MUX et Fault-Tolerant Configuration Engine) sont durcis pour être eux même tolérants aux fautes. Ainsi le COMP_MUX est intégré dans une zone de reconfiguration dynamique, et le FTCE est tripliqué. Cette triplication est cependant moins couteuse que la triplication des processeur beaucoup plus gros. Un autre avantage de notre proposition est sa généricité puisqu'elle peut être appliqué à n'importe quel coeur de calcul qu'il suffit de dupliquer.

Chapter 1

Introduction

1.1 Problem statements

Critical applications from particular domains such as military, aerospace, automotive and medical imaging require the implementation of fault-tolerance strategies to ensure reliable operations during system life-time. A defect during operation of safety critical systems may endanger human lives or the environment. The stringent requirements in term of reliability pose new challenges to electronic designers. Remote space applications have encountered this problematic due to the impossibility to repair systems after launch in space. State-of-the-art research and industry solutions attempt to alleviate the penalty of implementing fault-tolerance schemes. Recently, the automotive industry has observed a significant transitions from mechanical engineering towards mechatronical products. The mechanical control is less present, and is being replaced by electronic components (X-by-wire control) called ECU (electronic control unit). The omnipresence of ECUs has led the electronic automotive industry to face the similar challenges of safety requirements for electronic devices.

Also, due to the process variation and aging, after long time of operation, electronic circuits accumulate harmful physic phenomena which can provokes faults. The effect of this type of fault is permanent and it is not possible to deal with by using strategies for transient faults.

Conventionally, the solutions adopted to deal effectively with the safety problem tend to the multiplication of computing resources and the concept of processing element redundancy. The correct functioning of the overall system is ensured by the fact that single errors affect only one of the design instances. Hence, one instance who react differently from the others can be distinguished and supposed to be faulty. Unfortunately redundancy becomes a major issue of fault-tolerance techniques in term of cost overhead. The hardware redundancy obviously needs more resources to implement a function. Besides the cost issues, the redundancy of computing resources poses the problem of their utilization ratios due to the resources overhead. The resources overhead of a module can limit the implementation of other functions in a given platform, so potentially limit the processing power of the global system.

This limitation of processing performance can have dramatic impact on modern application. More and more complex applications for electronic systems in general and for electronic automotive controls in particular led to a large-scale integration of electronic components. This trend induces parallel computing systems for solving complex problems with specific time constraints. A parallel computing system is made up of various processing elements (PE) that work cooperatively to solve a problem. Embedded systems contain general-purpose processors for flexibility of software programming and dedicated hardware components for processing peformance. Embedded systems need more computational power to satisfy today's applications' needs, like audio/video encoding/decoding, image processing, etc. Multiprocessor systems-on-a-chip (MPSoCs) are an option to deal with this increasing computational requirements. This approach offers certain flexibility thanks to the software reprogrammability. Moreover, a high-performance system can be built-up with the execution of many functionalities in parallel on different processor cores.

Recently, the technological and process advances within the semiconductor industry allow the manufacturers to develop and support new technologies to improve the quality, the performance and the reliability of systems. Dedicated hardware circuits (ASICs) can respond to real-time constraints and permit ultimate reliability schemes, but they do not offer the required flexibility. Thus a flexible, programmable hardware acceleration is needed.

Along with the increased performance, quality and reliability requirements, the electronic

industry has also a significant focus on product cost reduction. The expectation on cost reduction forces the manufacturers to develop new technologies while improving manufacturing efficiencies in order to meet targeted cost reductions. The ASIC solution often requires a significant lead-time and nonrecurring engineering (NRE) costs to bring the design to fabrication. This is in conflict with industry trends. A solution to this problem, gaining acceptance in the automotive industry, is Field Programmable Gate Arrays (FPGAs) which permits to add flexibility, reduce the costs and support required performances.

FPGA suppliers have taken advantage of technology evolution to offer reconfigurable devices that can include millions of programmable gates along with megabytes of internal memory and processor cores suitable for customizing a large variety of applications. The combination of reusable intellectual properties (IPs), low unit costs and relative ease of implementation has led to increased FPGA appearance in the industry. Designers are turning to FPGA solutions to enable the required features and functions not currently available with standard components. This technology acceptance is partially due to increased marketing efforts from FPGA manufacturers, but largely due to the lower prices of the products.

However, it is not just only price that makes FPGA an attractive solution. Also a new effort toward reducing the time-to-market cycles of electronic products is introduced. The use of FPGAs can allow for the rapid introduction of new functionalities without the long lead times typical in the development of custom ASICs. This is evident by the combined effort of FPGA suppliers and third-party vendors to provide dedicated specific content, such as communication controller cores targeted specifically for automotive applications (CAN, MOST, FlexRay, etc.). Moreover, the cost advantages of an FPGA vs. an ASIC tip the scale especially as the actual process technology nodes decrease, where the mask costs alone can exceed one million dollars [4].

It is now recognized that the reconfigurable logic circuits can meet the processing performance requirements and the challenge of scalability of applications. The main advantage of this type of solution lies in the possibility of not systematically increase the number of processing elements, but rather to enjoy the genericity of the proposed architecture to optimize the utilization of resources particularly following the idle time.

The concept of re-programmability has dictated much of the approach to FPGA design. Unlike an ASIC, where the cost of a redesign can exceed a million dollars, the cost to fix an FPGA design is perceived to be relatively low. Hence, FPGA designs appear to allow fast creation and system-level integration testing of the design, with fine-tuning and design modification being completed through multiple reprogramming of the device. Moreover, recent advanced feature present in modern FPGAs—dynamic partial reconfiguration (DPR), provides further opportunities to enhance the flexibility and possibly the reliability of the target system. DPR allows multiple design modules to time-share physical resources. This feature enables run-time modification of one portion of the FPGA while the rest of the circuit keep running. By using DPR, designers can dramatically increase the functionality of a single FPGA, allowing a system to be implemented with fewer and smaller devices than is otherwise required. Exploiting DPR for automotive electronic system enables to implement traditional fault-tolerance techniques like classical redundancy using available reconfigurable resources. Furthermore, DPR paradigm permits to developped new strategies for fault-tolerant system design. For example, the DPR eases the system recovery mechanism after a fault occurrence by, either bringing the defected module to the correct functioning with its tasks without disrupting other modules, or modifying another module to carry out interrupted tasks. Not only a higher level of flexibility is introduced, but also the service continuity is considerably increased. Thanks to the non-restart of the whole FPGA, the operation state of the system is guaranteed, which can permit to easily switch to a degraded mode with the maintain of most critical services.

A new trend is to exploit the DPR feature into multiprocessor platform design to form dynamic MPSoCs. This coordination introduces a new degree of freedom for system design and run-time behavior. In a such system not only the software, like in static MPSoCs, but also the hardware can be adapted at run-time. This dynamic adaptation of the hardware architecture gives the designer a new degree of flexibility to ensure an optimized distribution of computing tasks on the processing cells and to fulfil constraints such as on performance, power and area consumption.

1.2 The CIFAER project context

The automotive industry has observed a significant transitions from mechanical engineering towards mechatronical products. The mechanical control is less present, being replaced by electronic components called ECU (X-by-wire control). With respect to the development of electronic domain, the evolution of electronic automotive industry has recently released the appearance of a large number of ECUs in recent automobiles. As presented in Fig. 1.1, since the 1950's, the omnipresence of ECUs in vehicles has led the industry to face a great challenge in term of increasing requirements about complexity, functionality, performance, communication and safety. The ECUs themselves as well as their communication buses require security for passenger safety. The automotive electronic systems demand the more and more computation capacity as well as reliability.

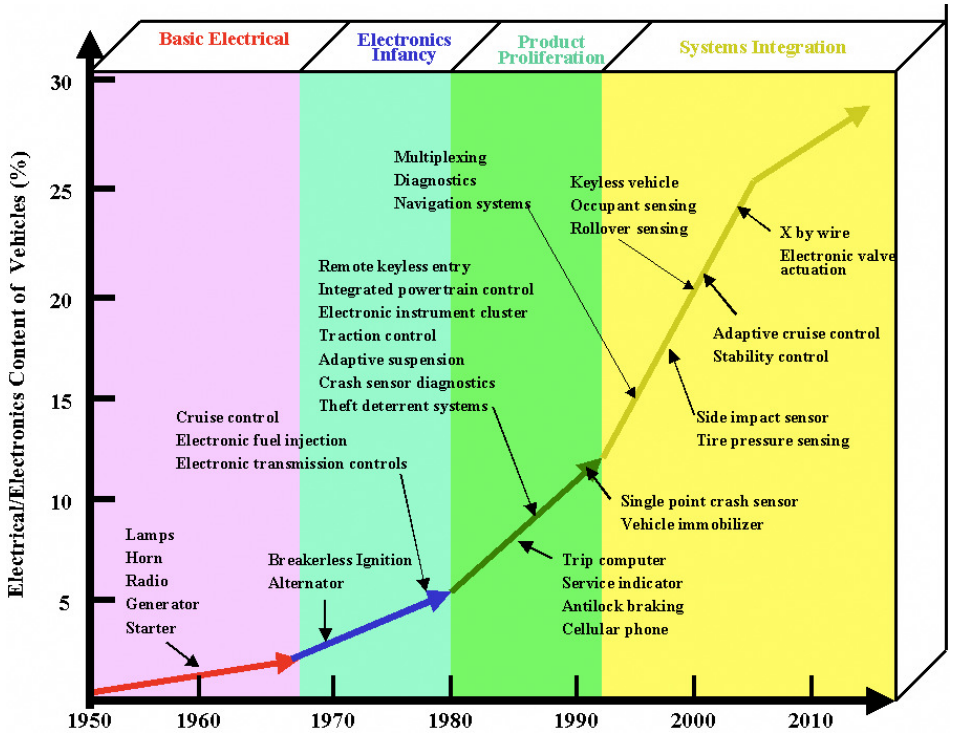


FIGURE 1.1: Trends in automotive electronics industry (data provided by TRW Automotive [2])

In recent years, the application of wireless and information technology products such as rear seat entertainment, digital satellite radio and navigation systems have presented the potential for significant new growth opportunities in the automotive entertainment domain.

In various application domains emerging requirements leads to the definition of new architectures for electronic embedded systems, both for software and hardware parts. For example, in the automotive context, investigated solutions correspond to network of processing elements, distributed in the vehicle. Notably, these substructures should support multimedia applications, still delivering working fail safe and fault tolerance (for example, driving assistance using GPS or image processing, etc.). Besides, considering the economical context, interoperable and easily updatable solutions should be proposed. One potential solution corresponds to the use of networks made of adaptable communication schemes, where each processing node could be reconfigured according to system behaviour. Thus, such targeted reconfigurable architectures should make possible to optimize processing execution and then match to functional requirements. System reliability and fault tolerance should also be improved by defining an efficient management able to modify and adapt the network property during system execution.

The research activity envisaged within the CIFAER project (Communication Intra-véhicule Flexible et Architecture Embarquée Reconfigurable) [5] (Fig. 1.2) concentrates on the definition of an architecture built around a processing unit supporting DPR (generic processor associated with a reconfigurable area) and supporting flexible communication interfaces. This flexible network will be based on radio frequency (RF) or power line communication (PLC) [6] technologies. This architecture will play the role of a processing node used for the flexible, tailored network to meet the constraints of bandwidth and dependability according to different working scenarios.

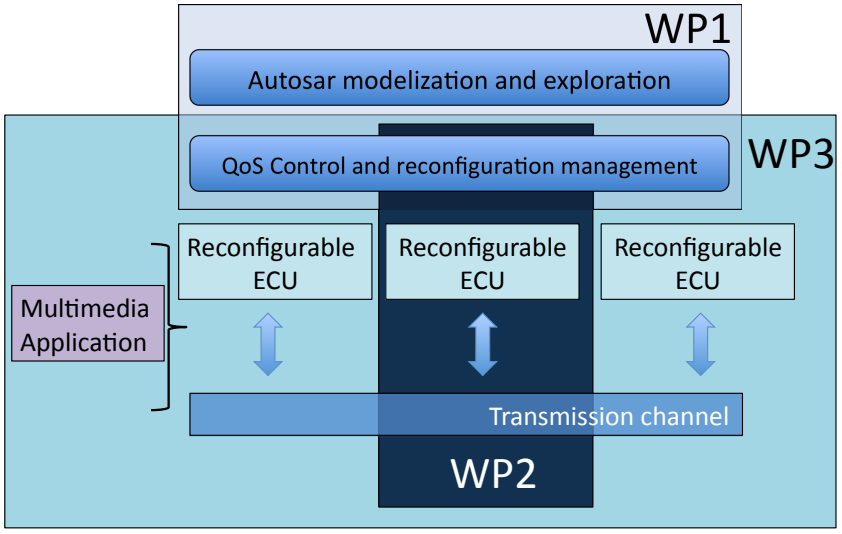


FIGURE 1.2: CIFAER project organization

The construction of a reconfigurable ECUs needs to exploit some of the benefits of PLC or RF communication. Indeed, these connections will allow to add, at low cost, functionalities in the vehicle or may carry data previously confined to a single ECU in the vehicle. The reconfiguration aspect would then have three major objectives:

1. support these new communications channels;
2. support the update of new features in the vehicle;
3. ensure the dependability by accessing data of a faulty ECU, for example.

The work of this thesis fits into the context of the CIFAER project. The project aim is to provide a flexible network of communication to consider the task migration. Critical tasks need to be maintained by migrating them from defected ECU to fault-free one. We focus initially on multimedia network (infotainment) of the vehicle, and thus the dependability aspects mainly concern aspects of "qualities of service" or "denial of service". We will not deal for the moment with safety aspects. In a second step, the study of an ADAS application will require methodologies and dependability to support more advanced level of reliability.

1.3 Objectives

In this thesis, our contribution is to propose and realize fault-tolerance services to the automotive system using dynamic partial reconfiguration. We propose to demonstrate the relevance of new dynamically reconfigurable technologies in the automotive electronic systems and more generally in the dependability domain. The use of dynamically reconfigurable computing has a double-objective:

1. reducing the number of ECUs thus reducing development costs and providing some hardware redundancy as well as a unified implementation for all tasks, whether critical or not. The occupation rate of the equipment is improved by the reutilization of processing elements during their idle periods;

2. supporting effectively classical and industrial approaches for dependability aspect. By performing a failure detection using reconfigurable computing and the coupling to a migration of tasks (hardware and software), it is possible to significantly increase the

robustness of the system, while maintaining optimized performance. We will use in this work the fault detection, fault correction and recovery mechanisms.

The first difficulty comes from the study of the hardware architecture of reconfigurable ECU compatible with the cost constraints, as well as the constraints of algorithm complexities and real-time. A complex problem is the interface between the computer and the vehicle when migrating tasks following the failure detection.

Finally, the management of the security operation will be optimized by reusing the hardware. For non-critical or complex medium, a time division multiplexing equipment will optimize the utilization of computers.

Based on the developed hardware architecture and its control software, a demonstrator performing the dynamic partial reconfiguration, as well as the services managing the fault-tolerance strategies, was realized during the thesis.

1.4 Contributions and content

In general, the fault-tolerance mechanisms require hardware overhead for implementing redundancies, which limit the number of processing elements deployed, and hence limit the system capacity. So the challenges spring from the application of fault-tolerance techniques that reduces the computational power of target system. During the thesis, we have managed to exploit the dynamic partial reconfiguration in order to prpose efficient and non intrusive fault tolerance scheme. The dynamic partial reconfiguration, on the one hand, helps improve the system reliability and amend the performance by reducing the negative impact of fault-tolerance, on the other hands enhance the flexibility by including the change on-demand capacity.

FT-DyMPSoC is a system that coordinates all the features of a dynamic MPSoC and also integrates fault-tolerance schemes to deal with potential errors. In our system, the fault detection is carried out at various levels: at processor level a lockstep scheme is implemented, while at system level we use the connection matrix approach. About the fault correction, all the processors inside FT-DyMPSoC are implemented in dynamically reconfigurable zones, and any processor can be reconfigured by another one in case of detected errors. This feature fulfills the drawback of state-of-the-art MPSoC systems

which usually contain a static processor as a master processor in charge of reconfiguring the others. Another challenge comes from the system building-up. Because the whole processor including the processor core as well as its peripherals needs to be reconfigured to deal with errors. Nevertheless this feature is restricted if the standard design flow is applied. We then introduce a modification of the design flow which eases the creation of fully dynamic processors. The modified flow necessitates the definitions of *wrappers* and *sockets* concepts. The support of these components (*wrappers* and *sockets*) alleviates complicated processes while constructing complex systems as all the structure complexities are managed by the design tools.

It is necessary to include fault-tolerance schemes to prevent electronic systems from being defective during life-time. However penalties always accompany fault-tolerance techniques, an achievement in reliability implies a lost in system performance. There lacks an effective method to estimate the impacts of fault mitigation schemes on the system performance. Thus, an analytical model is proposed in Chapter 4 to help ease the evaluation of performance/reliability trade-off while including fault-tolerance technique into the target systems. Then our FT-DyMPSoC is introduced into the model to evaluate the impact of fault-tolerance on the system performance. Additionally, one of the most popular state-of-the-art fault-tolerance technique for FPGAs—*scrubbing* [7] is also applied in the model for comparison. The obtained results demonstrate the better performance/reliability trade-off of our FT-DyMPSoC than *scrubbing*. Furthermore, the model is not only restricted to system using FPGAs, as the model has a wide applicability on various fault-tolerant system using different architectures by defining appopriate parameters. Also in this chapter, a simulation and verification model for fault-tolerant systems proposed. This model helps to quickly validate complex systems including fault-tolerant feature. The simulation models allow for first, quickly verifying the e?ectiveness of the fault-tolerance schemes applied in the system, secondly speeding-up the period of user-application design and debug. Implementation and analysis results are given to demonstrate the interest of this simulation/verification model.

Duplex system is more appreciated than a triple redundancy system in term of required hardware overhead. On the contrary, duplex system lacks the fault identification localization, and hence correction capabilities which are present in triplication system. Ameliorating existing fault-tolerant schemes based on duplication using dynamically reconfigurable architectures is presented in Chapter 5. In this chapter, our fault-tolerant

low overhead softcore processor system based on lockstep scheme is proposed and realized. The lockstep system contains a duplex copy of processor which is able to detect error in the dual processor thanks to a mismatch indicator. Our proposal enhance the lockstep scheme by adding the fault identification capability. A proposed configuration engine supervises the system in back-ground. The fault localization action detects which processor within the duplex copy is defected by error. The proposed COMP_MUX (Comparator/Multiplexer) contains a standard comparator that detects any inconsistency between the two identical processors, then the fault localization operation of the configuration engine is launched beyond the fault detection (reported by the comparator). Afterwards the correct output of the fault-free processor is instantly switched to the final output by the multiplexer. That prevents the erroneous results from being introduced to the environment and thus avoids any potential catastrophic results propagation. The operation disruption due to fault occurrence is minimized offering a big advantage to the system safety. Moreover the generality of configuration engine and COMP/MUX concepts do not prevent them from being implemented in diverse types of systems. It is enough to implement the target module in duplex, then add the configuration engine and the COMP/MUX into the system. The big hardware overhead of triplication is relieved while still providing the fault identification capacity.

Chapter 2

Background and Related Works

2.1 Reconfigurable architecture

2.1.1 Introduction

Fine-grained reconfigurable devices allow for fast functionality modification at a low level of granularity. For example, the device can be modified such as to add or remove a single inverter or a single logic gate. Fine-grained reconfigurable devices are mostly represented by complex programmable logic devices (CPLD) and the field programmable gate arrays (FPGA). Because of higher density integration of logic blocks in one device, using FPGAs allow for building up high-performance system with low development cost, short time-to-market and reprogrammability. Especially, FPGAs provide high flexibility at both design-time and run-time.

Figure 2.1 provides a simplified structure of an FPGA. The basic architecture of FPGAs consists of three kinds of components: logic blocks, routing, and input/output blocks. Generally, FPGAs consist of an array of configurable logic blocks (CLBs) that can be interconnected each other as well as to the programmable I/O blocks through some sort of programmable routing architecture. FPGAs can be programmed (configured) to realize the required functionality. The CLB is composed of Lookup Tables (LUTs), multiplexers (MUXs), Flip-Flops and registers. The contents of CLBs are programmed to control the functionality of the logic blocks, while the routing switches (switch boxes) are programmed to realize the desired connections between the logic blocks.

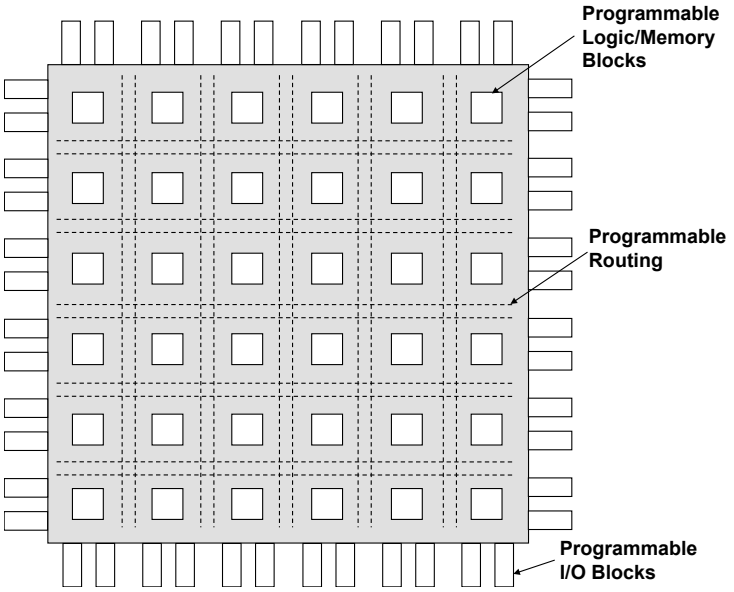


FIGURE 2.1: Generic FPGA architecture

There are different methods to store this program information, ranging from the volatile SRAM method [8] to the antifuse technology [9]. The area of an FPGA is mainly due to the programmable components. Hence, the programming technology can also affect the area of the FPGA. Another factor that has to be considered is the number of times the FPGA has to be programmed (configured). Antifuse-based FPGAs can be programmed only once, while SRAM-based FPGAs have no limit to the number of times that the array can be reprogrammed.

The configuration details of SRAM-based FPGA are stored in SRAM memory cells. The lookup tables (LUT) used in the logic block are also stored in the SRAM cells. When the interconnect network is implemented using pass-transistors, specific SRAM cells switch the transistors on or off. This storage method is volatile and the configuration has to be written into the FPGA each time on power-up. So for systems using SRAM-based FPGAs, the configuration is usually fetched to an external non-volatile storage device. The SRAM technology offers the convenience of reusing a single device for implementing different applications by loading different configurations. This feature has made SRAM-based FPGAs popular in reconfigurable platforms, which strive to obtain performance gains by customizing the implementation of functions to the specific application. But using this technology, each cell requires at least five transistors. Due to the relatively large size of the memory cells, the area of the FPGA is dominated by the configuration storage.

With the low granularity of the function generators (LUT and MUX), FPGA allows for programming any kind of function as far as this can fit onto the device. However, the programmable interconnections between the logic blocks reduce the performance of FPGAs. To overcome this, one potential solution is to embed a frequently-used module as hard macro in the device, as it is the case in hybrid FPGAs. Therefore, it allows programmable interconnections only between processing elements available as hard macros on the chip. Coarse-grained reconfigurable devices follow this approach. In general, those devices are made upon a set of hard macros (8-bit, 16-bit or even a 32-bit ALU) carrying few operations such as addition, subtraction or even multiplication. The interconnections are realized either through switching matrices or dedicated buses. The configuration is done by defining the operation mode of the hard macros and programming the interconnection between the processing elements.

Large FPGA companies also provide many coarse-grained elements embedded in their hybrid devices according to the market need. For instance, Xilinx embed in some Virtex family hard-wired 32-bit RISC processor PowerPC [10] which can construct high-performance embedded system combining with high flexible reconfigurable resources.

2.1.2 Reconfigurable processor

Modern FPGAs, besides customary high-density reconfigurable resources, offer the designers the possibilities of implementing reconfigurable processors, having features of Commercial Off-The-Shelf (COTS) components (no need to modify processor architecture or application software). Processors are in charge of collecting the data from peripherals and from the memory, process the data and send them to the memory and to the peripherals. Also, processors manage the memory and initialize the peripherals. Xilinx FPGA devices include two categories of processors: the hardcore embedded processor (PowerPC [10]) and softcore processors (MicroBlaze [11], PicoBlaze [12]). Altera also provides two processor type: ARM-based hardcore processor (Excalibur) [13] and softcore processor (Nios) [14]. Hardcore embedded processors are hard-wired on the FPGA die and their number is limited on each device (1, 2, 4 or no hardcore processor). On the other hand, softcore processors use reconfigurable resources, so the number of processors that can be actually implemented depends on the device size only.

2.2 Dynamic reconfiguration

2.2.1 Definition

Currently, the main interest in the granularity of the FPGA programming data is related to the property provided by some recent FPGAs like Virtex families of Xilinx to perform on-line programming (dynamic reconfiguration) of a portion of their logic (partial reconfiguration). The system can change its behavior according to its environment or external events during run time. Dynamic reconfiguration even permits a system to change partially his logic resources without affecting the rest of the system. Among the FPGA companies, Xilinx is known as the biggest one who offers the commercial Virtex families with partial reconfiguration capability.

The behavior of the FPGAs is determined by a configuration bitstream that consists of a sequence of *commands* and *control signals* bits data. The reconfiguration processes, by downloading this sequence, program the FPGA to perform the design functions. The reconfiguration process itself can be either done completely or partially depending on the type of the bitstream to be downloaded. The design and its bitstream has a nonreciprocal relation: it is not possible to extract the design structure and implementation on the FPGA from the bitstream.

Modular or module-based reconfigurable system [15] is the system which has at least one entire block is dynamically reconfigurable. In this kind of system, the FPGA fabric is partitioned into one static logic and one or more partially reconfigurable regions (PRRs). The static logic is part of the design which does not change and stay stable during all the execution of a configuration. It can be operational, and include logic controlling the reconfiguration process. PRR contain logic that can be partially reconfigured independently of the static logic and other PRRs [16]. The resources occupied by the PRR have to be more than the resources required for the related module. That makes the occupation rates of the PRRs are inferior to 100%. Each PRR has a related partial bitstream and the reconfiguration process can be done by sending this partial bitstream to the reconfiguration port—Internal Configuration Access Port (ICAP) [3]. And the size of the bitstream depends on the area of the PRR.

In one PRR, several PRM (Partially Reconfigurable Module) could be loaded (one at a time). Each PRM is designed and implemented individually using the partial reconfiguration design flow [16]. All PRMs for a given PRR must be pin compatible with each other, i.e., have the same port definitions and entity names.

The creation of PRRs must satisfy place and route constraints of reconfigurable resources. One of these constraints is the reconfiguration granularity of each FPGA decided by the smallest amount of data that can be reconfigured—a reconfiguration frame. To partially reconfigure the FPGA-logic at run-time, an entire reconfiguration frame must be rewritten to the configuration memory for the smallest change. The reconfiguration of the whole frame that does not disturb other frame is called configuration scrub, which is sometimes referred as frame-based reconfiguration (differ with module-based reconfiguration).

Virtex architectures have configuration memory arranged in frames that are tiled across the device. These frames are the smallest addressable segments of the device configuration memory space that physically correspond to base regions in the device matrix. The dimension of base regions varies on different devices.

- Base regions in Virtex-6 are 40 CLBs high by 1 CLB wide [17].
- Base regions in Virtex-5 are 20 CLBs high by 1 CLB wide [3].
- Base regions in Virtex-4 are 16 CLBs high by 1 CLB wide [18].

Two PRRs must not overlap vertically inside one base region. However, since recent FPGAs (Virtex-4, 5 and 6) can have multiple base regions along one device column, two PRRs may overlap vertically in the FPGA as long as they do not share any base region.

Another important notion in designing a dynamically reconfigurable system is Bus Macro [16]. Bus Macros are physic ports which connect PRMs to the static logic. Any connection between a PRM and the static logic must pass through Bus Macros. Furthermore, Bus Macros serving as outputs from the PRM should be disabled prior to the partial reconfiguration as these signals can toggle unpredictably during the reconfiguration process, hence affect other logics. Therefore, enable control signal for Bus Macros outputs is provided. The Bus Macros should be disabled by de-asserting enable signal prior to

reconfiguration process, and then re-enabled after the last bit of the partial bitstream is loaded.

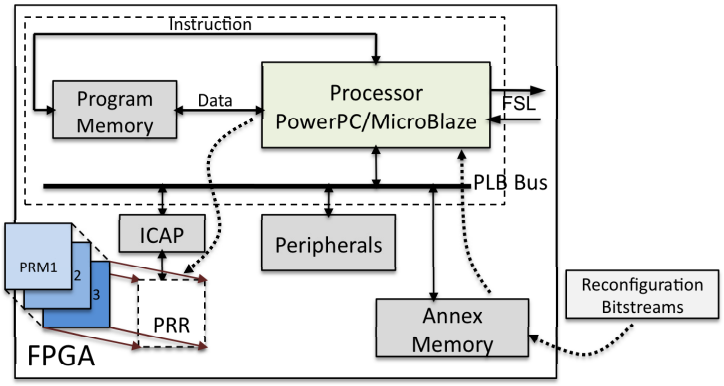


FIGURE 2.2: Microprocessor-based system controlling reconfigurable resources

Conventionally, in a dynamically self-reconfigurable system, there is a central processor who connects to the reconfiguration port and controls the reconfiguration process by sending partial bitstreams to this port. In the partial bitstreams, there is already all the information concerning the reconfigurable zones. Figure 2.2 depicts a typical structure of a processor-based system in one Xilinx Virtex device. The heart of the system is the processor who reads the configuration bitstreams in the annex memory and controls the reconfiguration process by sending partial bitstreams of different Partially Reconfigurable Regions (PRRs) to the Internal Configuration Access Port (ICAP) [3]. The partial bitstreams are stored in the annex memory that can be external like the Compact Flash or internal like the BRAM. The software program of the central processor resides in the program memory and drives the reconfiguration via the ICAP controller. The processor is usually considered in the static logic to dynamically control the other reconfigurable resources.

2.2.2 Design flow

Standard design flow for modular reconfiguration using Xilinx FPGAs is based on provided tools: Embedded Development Kit (EDK) [19], Integrated Software Environment (ISE) [20] and PlanAhead [21] as shown on Figure 2.3. As stated before, the self run-time reconfiguration of a FPGA is usually driven by a reconfigurable processor on the same device as shown on Fig. 2.2. So the design flow of a reconfigurable system is as follow.

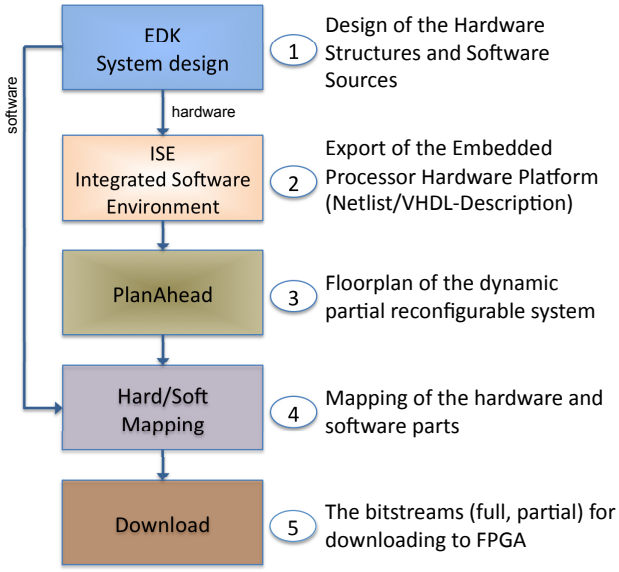


FIGURE 2.3: Standard design flow for dynamically reconfigurable system

First, the whole embedded system of Fig. 2.2 including hardware structures (Processor, ICAP controller,...) and processor software sources are constructed using EDK. Then the hardware parts are synthesized using ISE. The floorplan stage is managed by using PlanAhead. During this stage, the PRRs are physically constrained on the FPGA matrix and the Bus Macros are also fixed at the boundary of the PRRs. Afterwards, the floorplaned hardware parts are combines with the software part to form a complete system including full and partial bitstream ready for download to FPGA. The main interest of the design flow concerns auto-insertion of Bus Macros into the initial design. With the old design flow relating to the version 8.2 and 9.2 of Xilinx tools (ISE, EDK, PlanAhead), the designer must manually add the Bus Macros into the design at the first stage of the design flow. With the new design flow of Xilinx tools version 12, Bus Macros are automatically added and placed at the stage 3 without the need of manual intervention.

Field Programmable Gate Arrays (FPGAs) are very popular for design solutions because of the high flexibility and reconfigurability feature, which reduces the time to market. They are also an attractive candidate for space applications in terms of high density, high performance, low NRE (Non-Recurring Engineering) cost and fast turnaround time. SRAM-based FPGAs can offer an additional benefit for remote missions. For instance, by allowing in-orbit design changes thanks to reprogrammability, with the aim of reducing the mission cost by correcting errors or improving system performance after launch. The reliability and human safety constraints of certain critical domains require the fault-tolerance schemes to be implemented in the system.

2.3 Fault-tolerance in reconfigurable architectures

Fault-tolerance on semiconductor devices has been a meaningful matter since upsets were first experienced in space applications several years ago. Since then, the interest in studying fault-tolerant techniques in order to keep integrated circuits (ICs) operational in such hostile environment has increased, driven by all possible applications of radiation tolerant circuits, such as space missions, satellites, high energy physics experiments and others [22]. Similarly, in automotive domain, the electromagnetic environment around the vehicle can vary really fast over time by passing across various sorts of objects: other vehicles, electric panels, traffic lights. Working in that harsh environment requires critical electronic circuits to ensure highly reliable operation for driver and pedestrian safety. Automotive systems include a large variety of analog and digital components that are potentially sensitive to radiation and must be protected or at least qualified for correct operation.

The fault-tolerant techniques can be classified as 2 types: the ones that change the technology of the fabrication process, and the ones that change the design structure of a system. The first possibility is to design new FPGA matrix composed of fault-tolerant elements. These new elements can replace the old ones in the same architecture topology or a new architecture can be developed in order to improve robustness. The cost of these two approaches is high and it can differ according to the development time, number of engineers required to perform the task and the foundry technology used. Actel offers the Radiation Tolerant FPGA families [23] in which flip-flops are protected by Triple Modular Redundancy (TMR). Xilinx, besides the commercial families, also offers military families that are radiation hardened. The latest family is Radiation-Hardened Space-Grade Virtex-5QV [24] which is compatible with commercial Virtex-5 FPGAs.

The second possibility is to protect the high-level description of the module by using some sort of redundancy, targeting the FPGA architecture. In this way, it is possible to use a commercial FPGA part to implement the design and the SEU mitigation technique is applied to the design description before being synthesized in the FPGA. The cost of this approach is inferior to the previous one because in this case the user is responsible for protecting his own design and it does not require new chip development and fabrication. The user has the flexibility of choosing the fault-tolerant technique and consequently the overheads in terms of area, performance and power dissipation.

All of these two solutions have their own space in the market, as each application has its own constraints. But because the semiconductor industry trends to reduce time-to-market and low-cost production, the implementations based on high-level design seem more interesting.

2.3.1 Fault models in reconfigurable architectures

The damage provoked by radiations may be classified in two principal categories:

- Single Event Effects (SEE) of which the classification is presented in Fig. 2.4
- Long terms cumulative degradation that may raise permanent faults

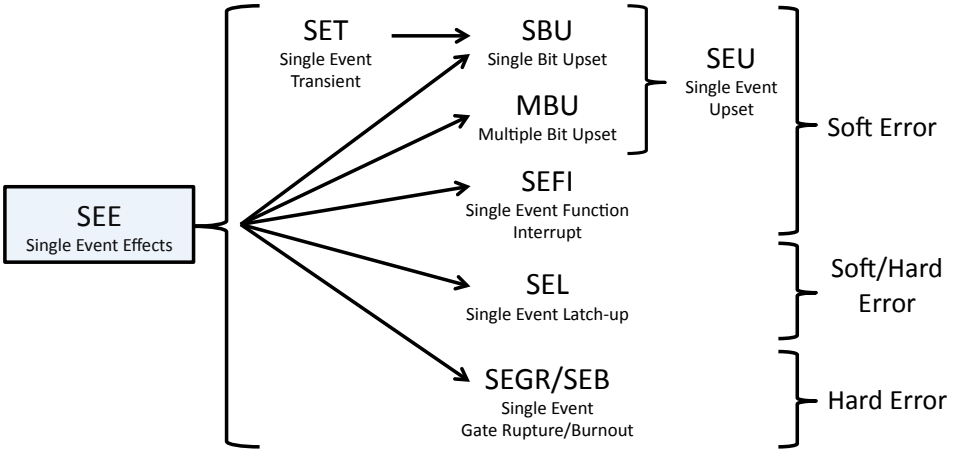


FIGURE 2.4: SEE classification

With potentially serious consequences for the application, including information loss and functional failure, Single Event Effect (SEE) is the major concern in mission-critical applications. SEE occurs when charged particles hit the silicon transferring enough energy in order to provoke a fault in the system. SEE can have a destructive or transient effect, according to the amount of energy deposited by the charged particles and the location of the strike in the device. The main consequences of the transient effect, also called Single Event Upset (SEU), are bit flips in the memory elements. According to the logic fan-out, SEE can also produce multiple transient current pulses at the output. Consequently, SETs in the logic can also provoke multiple bit upsets (MBU) in the memory elements.

2.3.1.1 Single Event Effect

Single Event Upset SEU has been constantly magnified in the past years, caused by the continuous technology evolution that has led to more and more complex architectures, with a large integration of embedded memories, followed by an amazing scaling down process of transistor dimensions following Moore's Law [25]. Figure 2.5.a represents a

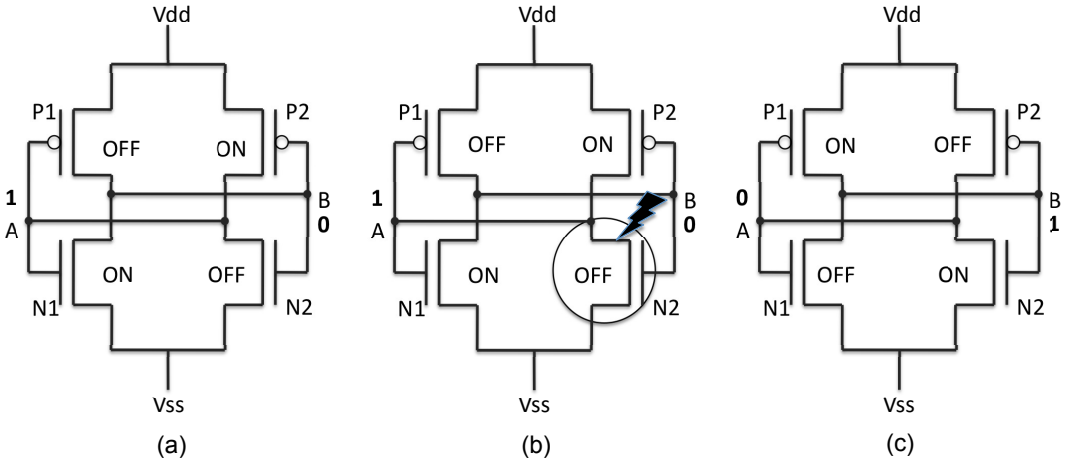


FIGURE 2.5: SEU effect on the state of a memory cell

SRAM memory cell which has two stable states '1' and '0'. In each state, two transistors are 'ON' and two others are 'OFF'. A bit-flip happens when an high-charged particle provoke the inversion of the circuit transistor state. This phenomena is called Single Event Upset (SEU) and is one major concern in integrated circuits. In SRAM-based FPGAs, SEU can affect the configuration memory by flipping a bit and corrupting the functionality of the target design. SEU can also attack the data path of the design that modify the data to processed, so produce wrong values at the output.

Single Event Functional Interrupt The malfunction provoked by an SEU is classified as Single Event Functional Interrupt (SEFI). The SEFI is the first anomaly within integrated circuits provoked by a bump of a single ion, similarly to the SEU, that introduces a temporary malfunction or interruption of the device standard operations. While the SEU is a phenomena that produces a temporary change of the device physical conditions, the SEFI is a phenomena that happens in the temporary change of the implemented functionality and may remain until the power supply is interrupted. The SEFI are observable in several devices, however until it is not related to a single cause, this phenomena remains hardly definable [26].

Single Event Latch-Up The ionizing radiations may provoke other kinds of effects called Single Event Latch-up (SEL), that is produced activating the parasitic transistor present between the junctions N-P of the CMOS transistors. The activation of such kind of transistor create a low frequency path between the power supply (Vcc) and the ground, crossed by an high current. If it is detected early enough after occurrence, it is considered as soft error requiring reconfiguration or reset to correct. However, the SEL effects are potentially destructive for an electronic circuit, hence it can become a hard error.

Single Event Gate Rupture/Burnout Single Event Gate Rupture/Burnout refer to destructive failures of transistors. An ion traverses the transistor structure, potentially damages (increase gate leakage) or ruptures the gate oxide insulation. This phenomena leads to device destruction if sufficient short-circuit energy is available.

2.3.1.2 Long terms cumulative degradation

The phenomena of the performance degradation of electronic circuit overtime is labelled "aging" [27]. This cumulative degradation during long term could be induced by radiation or by process variation [28] or both. The process variation may produce delay and power leakage differences between transistors on the same circuit. The high-charged radiation can provoke hot-carrier effect (HCE) [29] degrading the functional quality of the transistor. All these delay and power variations due to process variation and HCE may provoke permanent faults after a long-term operation, which cannot be eliminated by reconfiguration.

2.3.2 SEU effects on configuration memory

The design functionality of FPGA is defined by configuration SRAM contents which are stored in configuration memory cells. As shown on Fig. 2.6, these memory cells define the states of LUTs, MUXs, Flip-Flop and switch boxes of the CLBs. The FPGA configuration memory consists of these cells. When an error produced by an SEU occurring in the configuration memory, it may modify the circuit functionality, e.g:

- Modify a LUT content: modify the combinatorial function

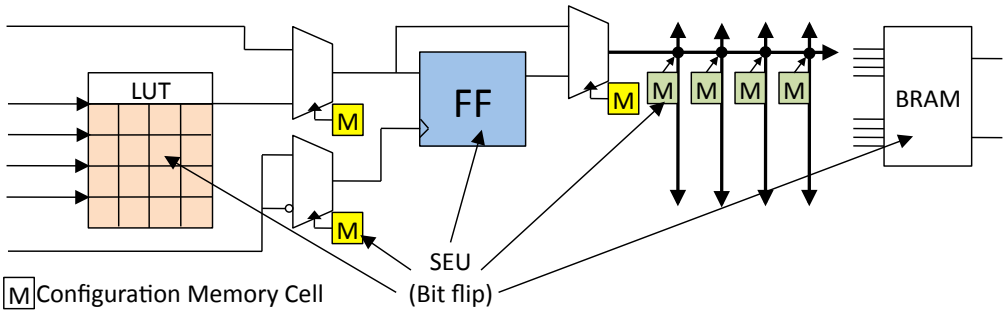


FIGURE 2.6: SEU effects on Configuration Memory of Xilinx Virtex

- Change I/O configuration: revert I/O direction
- Modify connecting matrices: cause an open connection or a short circuit

These functional modifications may cause the functional failure or loss of logical functionality. Since the configuration memory upsets still remain latched, their effects are permanent until the configuration memory is refreshed with correct configuration data.

Configuration upset classification In many cases, the functional errors that occur after an SEU are temporary. Once the configuration fault has been repaired, there is no sign of failure. In other cases, the functional output errors persist indefinitely beyond repair. The concepts of persistence and non-persistence are based on this idea that, in a system with configuration scrub, the duration of some functional errors is finite.

- non-sensitive bit: non-sensitive bits act as "don't care" configuration bits for a design. A bit-flip in non-sensitive bits does not affect the functionality of that particular mapped design. Hence, the sensitivity of each configuration bit is *application-dependent*.
- sensitive bit: Any bit-flips in sensitive bits will eventually affect the user-bits (system state). Frames containing sensitive configuration bits are defined as sensitive frames.
  - non-persistent bit: The sensitive configuration bits provoke functional errors which do not persist in a design and are flushed after configuration scrub. Once the errors have flushed, the system exhibits no signs of failure. These transient errors represent a temporary interruption of service that do not require a reset to recover.

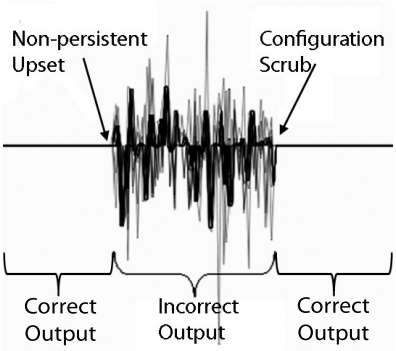


FIGURE 2.7: Non-persistent upset

Fig. 2.7 illustrates the arithmetic difference between two data streams generated by identical designs. The figure depicts a non-persistent error. After the upset reparation, there is no difference between the two streams.

– persistent bit: Sometimes an SEU-induced fault within the configuration memory of an FPGA will introduce functional errors which indefinitely propagate within a circuit, even after configuration scrub repairs the fault. Unlike non-persistent errors, persistent errors do not disappear after configuration scrub. Although configuration scrub repairs the circuit structure, the temporary circuit failure inserts incorrect state into the system that cannot be corrected, and will not self-correct without a global system reset. Persistent functional errors react like a SEFI. However, persistent functional errors are specific to the configuration programmed into the FPGA, not to the specific device. In addition, persistent errors can be removed with a global system reset while a SEFI usually requires a system power off/on to recover [30].

  Persistent errors are caused by an SEU within the configuration memory corresponding to circuit structures that contain feedback and store internal state. The feedback structures "trap" the incorrect state and store this erroneous state until appropriate reset measures are taken.

  Fig. 2.8 illustrates a persistent error, or permanent service interruption. Even the configuration scrub repairs the upset, the output difference still persists.

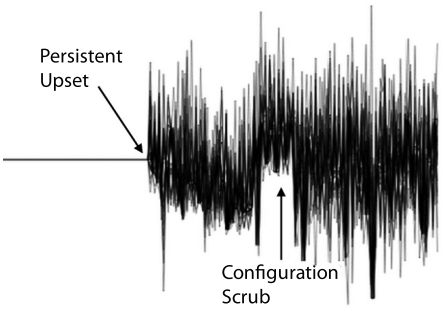


FIGURE 2.8: Persistent upset

2.4 Classical fault mitigation schemes

Since upsets in electronic devices were first discovered, the interest in applying fault-tolerance techniques to ensure the operation of target devices in harsh environment has increased. Besides hardening in fabrication process, the electronic designers have also applied classical approaches at different level to keep the circuit operational.

2.4.1 Architectural level

At architectural level, the popular approaches of fault detection and tolerance approaches are based on the redundancy. These approaches do not require any modification in the internal structure of the target block.

2.4.1.1 Hardware redundancy

Using the hardware redundancy, the target blocks are replicated so that the system could monitor its functioning state itself. Generally, there are two approaches: Duplication with Comparison (DWC) where the hardware resources are doubled and Triple Modular Redundancy (TMR) where the hardware resources are triplicated.

In DWC (Fig. 2.9), the original module is replicated twice and the results produced by the original and the replicated modules are compared to detect faults. However the DWC scheme cannot identify the faulty module. DWC allows to tolerate temporary faults, provided that it is supported by re-execution. Then, any disagreement between the dual modules outputs triggers a few attempts to repeat the last operation hoping that the error was due to temporary fault (in case of failure, a permanent fault is declared).

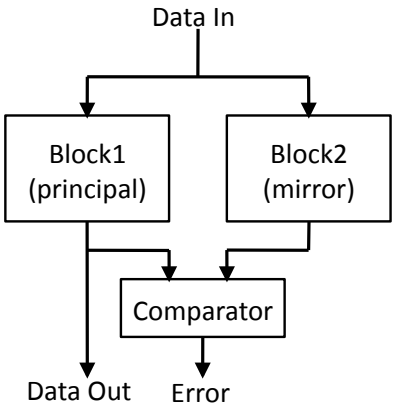


FIGURE 2.9: Duplication with comparison (DWC)

Lockstep scheme is the implementation of DWC at the processor level, supported by some Xilinx FPGAs [31]. Two identical processors μP1 and μP2 receive the same inputs, simultaneously execute the same instructions, and their results are compared step-by-step at each clock cycle. μP2 generates the reference results to be compared against those of μP1 that provides the system output. This system is able to detect but not to correct error, because it cannot point out the faulty processor. In case of error, the whole system need to be refreshed to recover correct functionalities of both processors.

One of state-of-the-art solutions in the automotive domain consists in implementing two different processors in one ECU. In the SAPECS (Secured Architecture and Protocols for Enhanced Car Safety) project [32], the main processing element is a 32-bit processor, the second processor is 8-bit to reduce the hardware overhead since it is only for verifying the main 32-bit one (Fig. 2.10).

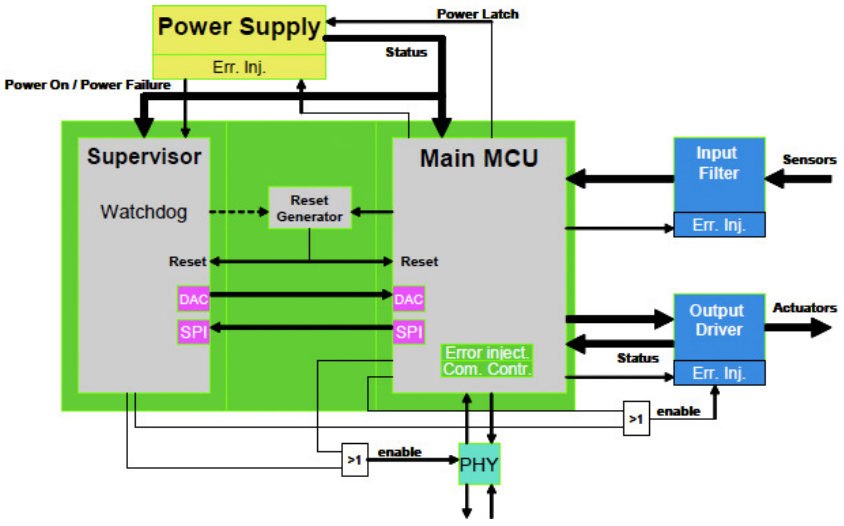


FIGURE 2.10: SAPECS ECU with fault detection

In the TMR scheme, it is possible to identify the error thanks to the resource triplication and the majority voter (Fig. 2.11). If a module output differs from the two others, this module is declared erroneous and the majority voter (Fig. 2.12) will choose the correct value of the two other modules. The scheme protects both combinational and sequential logic against temporary upsets. However, if an error occurs in the voter, the TMR scheme may produce a wrong value in the output. To overcome this issue, Xilinx proposes the XTMRTool.

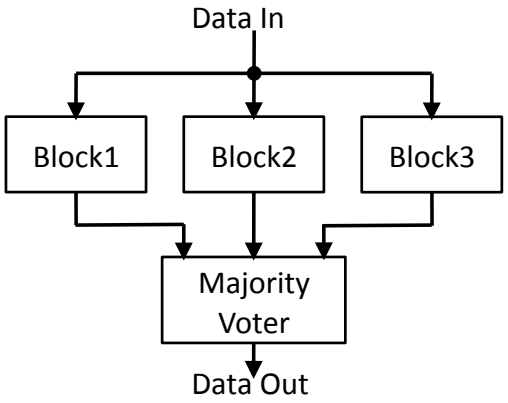
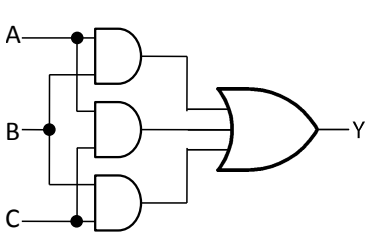


FIGURE 2.11: Triple modular redundancy (TMR)



A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

FIGURE 2.12: Majority voter and truth table

In the XTMR scheme (Fig. 2.13), the majority voter is triplicated and the minority voters are also added to the system. Thanks to these minority voters, the output of the block which behaves differently will be disconnected by the tri-state buffer (TBUF). However, in the recent Xilinx FPGAs, TBUFs are used for connecting to the final I/O pins of the FPGAs, one TBUF controls on I/O pint. So the outputs of TBUFs in XTMR scheme must converge externally i.e., on the PCB trace.

It is difficult to implement this solution in large system due to limited number of TBUFs. Furthermore, I/O pins are usually used for connecting to external controllers which cannot be implemented on the FPGA.

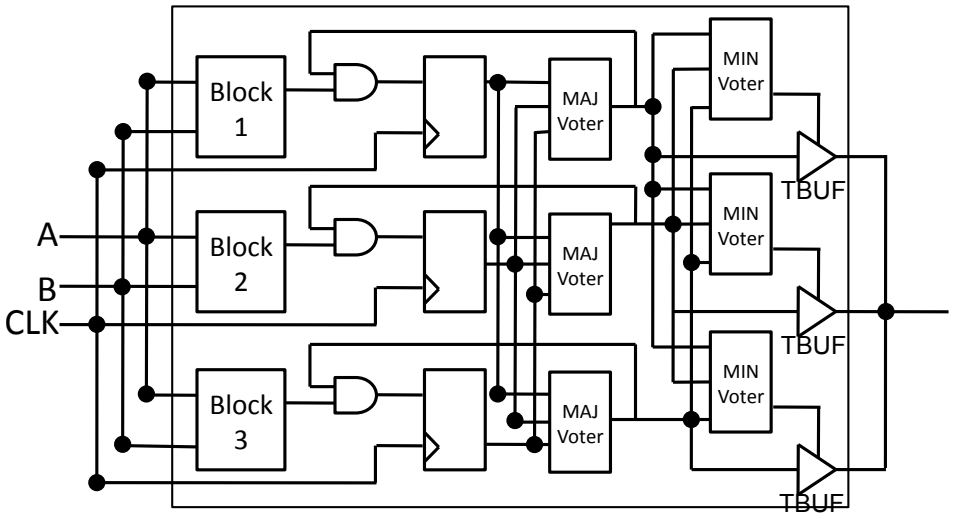


FIGURE 2.13: Xilinx TMR—XTMR scheme

2.4.1.2 Time redundancy

In time redundancy approaches, the register cells are doubled or tripled and the computation of a given signal is made at two or more different instants of time and a majority voter selects the correct output of the registers [33]. This way, a SET with duration smaller than the time delay between the loads of the redundant registers shall not affect the system operation.

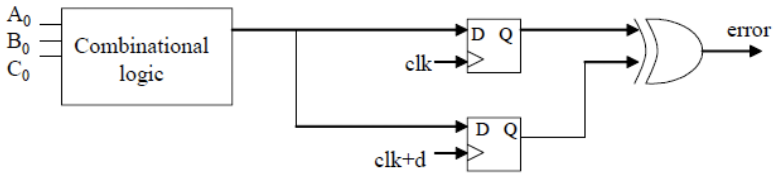


FIGURE 2.14: Time redundancy scheme for combinational logic

Techniques based on time redundancy are usually used to detect a transient effect (SET) in the combinational logic, while hardware redundancy can help to identify an SEU in the sequential logic. In [34], examples of the use of time and hardware redundancy for SET detection is presented. In the case of time redundancy, the goal is to take advantage of the characteristics of the transient pulse generated by the particle strike to compare the output signals at two different moments. The output of the combinational logic is latched at two different times, where the clock edge of the second latch is shifted by time d. A comparator indicates a transient pulse occurrence (error detection). The scheme is illustrated in Fig. 2.14. Nevertheless, the time redundancy approach does not have

much use in FPGA since the faults in the reconfigurable resources become SEFIs which are permanent unless reconfiguration or reset is applied.

2.4.1.3 Error-correcting code

Error-correcting code (ECC) technique [35] is also used to mitigate SEU in integrated circuits. It is usually used in memory. There are many codes to be used to protect the systems against single and multiple SEUs. An example of ECC is the hamming code [36] in its simplest version. It is an error-detecting and error correcting binary code that can detect all single- and double-bit errors and correct all single-bit errors (Single Error Correction-Double Error Detection (SEC-DED)). This coding method is recommended for systems with low probabilities of multiple errors in a single data structure (e.g., only a single bit error in a byte of data).

2.4.2 System level

When a design is highly modular, a fault can be tolerated by the use of a spare functional block, providing degraded performance or by dispatching the failed task to other blocks throughout the system. State-of-the-art solutions in automobile domains explore also the fault-tolerance at system level. In the ReCoNets project [37] reconfigurable nodes are connected together to form a network of reconfigurable computers. Procedures for self-repair and intelligent partitioning were developed to achieve fault tolerance at system level. In order to guarantee short repair times in case of node defects, the placement of tasks is optimized and replicated nodes are created.

The ReCoNets demonstrator contains 4 FPGAs with 1 processor inside each FPGA. If one FPGA is disconnected due to some failure, its tasks will be automatically distributed onto the three others, and the system continue functioning. This system does not exploit the DPR, so it require a high amount of memory to store all the bitstreams. Furthermore, when there is a fault in one FPGA, the three others must be restarted to relocate the tasks which potentially has a long period of function interruption.

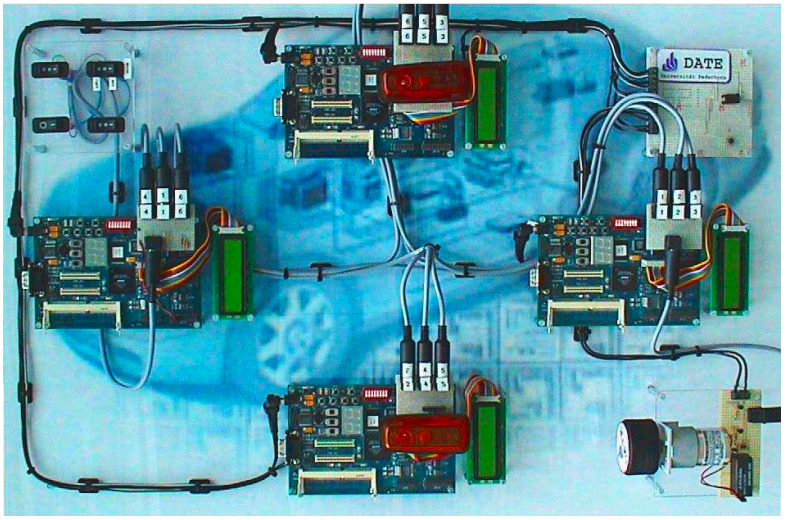


FIGURE 2.15: Multi-FPGA ReCoNets demonstrator

2.4.3 Context recovery strategies

2.4.3.1 Context introduction

In a fault-tolerant system, besides the fault detection and correction, it is necessary to put the faulty module to the same correct state as before the fault occurs, especially when full system or process restart may be unacceptable. So it is required to save the correct context and restore it after the error is corrected. Then the module could resume the execution at the saved point.

The context is a set of information needed to uniquely define the state of the module at a given moment. It could include the states of all the related registers, the cache, the memory, etc. of the module. Saving and restoring all relevant values allow for module context switching and error recovery.

Processor context A processor context is contained in processor registers values. The softcore processor MicroBlaze context is represented by the 32-bit values of 32 General Purpose Registers and two Special Registers: the Program Counter (PC) and the Machine Status Register (MSR) [11].The values of these registers represent the state of the processor.

There exists various context recovery techniques varying on the moment to save and restore the context.

2.4.3.2 Checkpointing and Rollback [1]

During task execution, the module is regularly checked for consistency at so-called checkpoints. If the module passes the check, the correct context is saved at that checkpoint. If fails, the module state is recovered from the correct state of the previous checkpoint. The process of regularly check for consistency and correct context saving is called checkpointing. The context recovery in this case is called rollback.

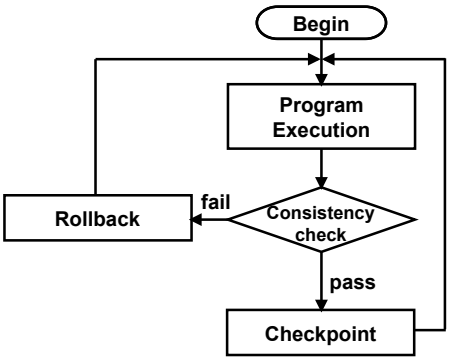


FIGURE 2.16: Rollback recovery using checkpointing

Rollback is usually used in duplex systems without the knowledge of the faulty module. When an error occurs, the checkpoint comparison only detects the inconsistency between the two modules. When no failure occurs in a checkpoint interval, no rollback is necessary. Figure 2.17 depicts a scenario when a fault occurs between two checkpoints. It is not possible to know which module is fault-free to copy its correct context so we cannot copy the context from A to B or vice-versa. So the only option is to rollback both modules to the previous checkpoint Fig. 2.17.

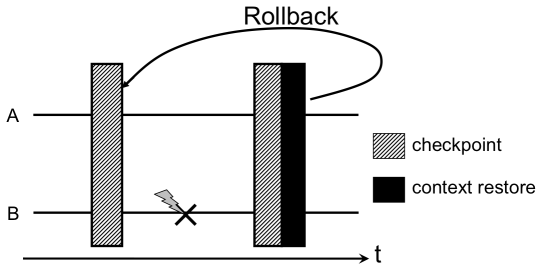


FIGURE 2.17: Rollback in duplex system

2.4.3.3 Rollforward [1]

On the other hand, in a system that is capable of localizing the error, rollforward can be used. In Fig. 2.18, when an error is identified in module B, the modules A and C

continue executing tasks. After the fault correction, the correct state can be copied to module B from module A or C.

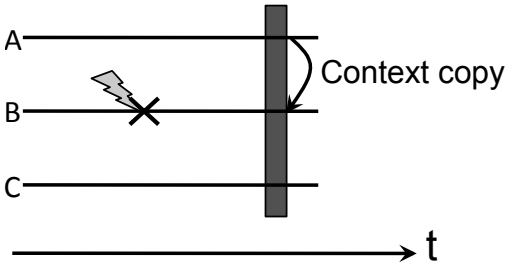


FIGURE 2.18: Rollforward scheme

Rollback can be used in a duplex system without error location capability and rollforward can be used for transient error recovery in TMR or in duplex with error location detection. In most cases, rollback provides more timing overhead than rollforward because of required time of the regular checkpoints combining with rollback. These are not necessary thanks to the knowledge of faulty module for the rollforward strategy.

2.5 Strategies for SEU

SEU is recognized as the main concern of the reliability problem in SRAM-based FPGAs. Dynamic partial reconfiguration feature present in new reconfigurable architectures does not only enhance the flexibility, but also can be used for ameliorating the fault-tolerance degrees of the systems. So besides classical fault-tolerance solutions listed in Section 2.4, some particular techniques using dynamic reconfiguration dealing with SEU were proposed. According to various SEU types in the configuration memory, the appropriate strategy is selected to deal with each particular configuration SEU. So we first classify configuration SEUs following their persistence behaviors (i.e. how the error affects other configuration frames in the FPGA). Then we introduce the potential strategies dealing with these configuration SEUs.

2.5.1 Readback

Readback [38] is a post-configuration read operation of the configuration memory. Once the FPGA is programmed, readback allows for reading the FPGA's programming data and other information through the configuration interface. First, readback can be used to

ensure that the current configuration data stored in the FPGA is correct. It provides the ability to detect SEUs in the configuration memory without disrupting its operations. Second, readback can be used to read the current state of all internal CLBs, connection and switch boxes for hardware debugging, functional verification of reconfigurable computing applications. Since readback provides only the dectection capability, it must cooperate with another correction technique.

2.5.2 Partial reconfiguration

The effect of configuration upset are permanent until the next configuration reload. It could be achieved by global reconfiguration that reloads the configuration of the whole device. This solution may produce a temporary service interruption of the whole system which could be prohibited for some applications. Another solution is to use partial reconfiguration to refresh only the related module without disrupting the rest of the circuit. Obviously, less time consuming overhead is provided with the partial reconfiguration to deal with upset removal.

Configuration scrub

If during readback, the current configuration data is found invalid, the upset should be corrected as fast as possible to avoid any unpredicted result. Since a configuration frame is the smallest reprogrammable segment, it is recommended to reload the whole affected frame with the correct data using configuration scrub. Configuration scrub can be combined with readback to define the bit-flip position and revert it, or the original frame data can be stored in a highly reliable memory where the scrub controller can find the configuration information.

Scrubbing

Upsets can accumulate in the matrix during a long period of time, provoking an error in the system even when the redundancy is applied, it is necessary to clean up all the upsets in such a frequency as to guarantee the correct functionality. The technique of regularly reloading the configuration data with correct information is called *scrubbing* [7]. *Scrubbing* is used to avoid the upset accumulation and *scrubbing* itself does not

contain any detection mechanism. *Scrubbing* can be used alone to clean up the upsets inside the matrix, if it periodically refresh the configuration memory. In this case, the refresh frequency must be superior to the expected appearance rate of the upsets. A throughout study on upset rates should be done before applying *scrubbing*, which requires lots of investigations of the device manufacturing technology as well as the environment condition. Consequently, that could significantly delay the time-to-market of the system.

Throughout this document, we differ configuration scrub as the operation of reloading or modifying one configuration frame with *scrubbing* as the operation of reloading all the configuration frames of the target system.

Module-based reconfiguration

Without identification of the exact faulty configuration frame, an entire defected module need to be reconfigured with correct configuration data to eliminate the error This necessitates the whole system to be designed using module-based partial reconfiguration. The module-based reconfiguration using the design flow explained in Section 2.2.2. The partial bitstreams of the reconfigurable modules are generated using this methodology. The reconfiguration of one module is realized by loading the correspondent partial bitstream using the reconfiguration port.

Tiling

The real permanent faults can appear in the matrix due to long-term accumulation or faulty manufacturing process. This kind of fault may provoke multiple configuration upsets and can be detected by using the combination of multiple strategies. However, permanent fault cannot be removed using configuration data reload. A solution to permanent fault repair in fine-granularity FPGA is presented in [39]. A faulty module can be repaired by reconfiguring the chip so that a damaged configurable logic block (CLB) or routing resource is not used by the design. Many techniques have been presented to provide permanent fault removal for FPGAs through reconfiguration. One solution is to generate at run-time a new configuration after permanent faults are detected. This solution is not feasible because of very complex logic synthesis, place and route strategies which can only be implemented on external computers. Another one called Tiling

is to generate pre-compiled alternative FPGA configurations and store the configuration bitstreams in non-volatile memory. Fig. 2.19 describes the principe of Tiling. The same design has three different configurations (a), (b) and (c) and each configuration has an un-used zone (tile). If a permanent error is found in a tile, the appropriate configuration which has the un-used tile covering the error, is loaded.

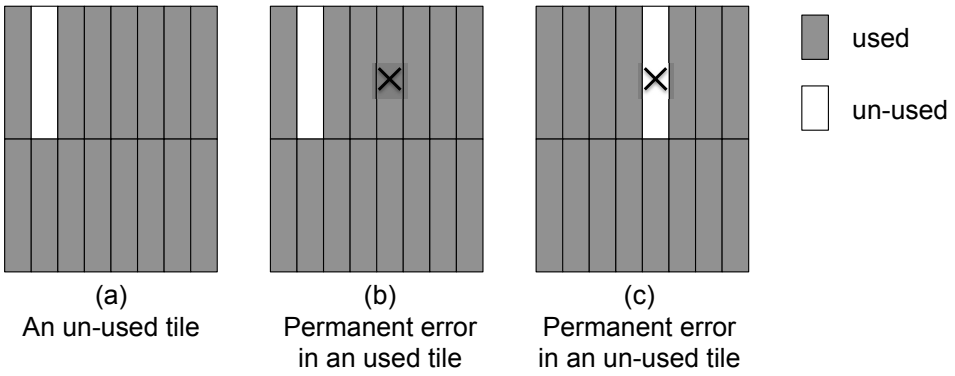


FIGURE 2.19: Tiling principe

Tiling is one of most popular techniques to deal with permanent fault because when permanent faults are detected, a new configuration can be chosen without the delay of re-routing and re-mapping. However, this solution require high memory amount for storing configurations.

The use of Tiling necessitates the application of a permanent fault localization mechanism such as: readback repeatedly the same faulty frame, if the fault persists after continuous correction, the permanent fault is declared and located by readback. Because of the frame-based reconfiguration granularity, it is recommended that one tile should not be smaller than one configuration frame.

2.5.3 Combined approaches

The combinations of several fault-tolerant approaches were exploited. The solutions from [40–42] combine readback with configuration scrub. They have the ability to detect and correct errors using low level approaches that detect undesired bit-flips in the configuration memory. However, these systems are unable to distinguish whether the detected errors really affect the design. Sometimes, the flipped bit does not belong to any design, so correcting this error is pointless, because this non-sensitive bit does not really affect the system functioning.

CRC signature and resources triplication with majority voter are used in parallel in [43]. The CRC signatures generated correspondent to replicated modules, a vote is done based on these signatures to detect the faults. The system can also be restored by copying the configuration memory of the fault-free FPGA to the defected one. Actually, this solution requires intrusive modification of the target design hence may affect the functionality. Moreover, this solution can only work with multi-FPGA system and not be applied into only one device because of the heterogeneity of the internal FPGA structure. Finally, the multi-FPGA system implementing this solution must use the same FPGA device for the entire platform also because of the heterogeneity of different FPGAs.

The method of dynamic partial reconfiguration synchronized with TMR presented in [44] combines the dynamic partial reconfiguration with TMR. The method uses large grain TMR with a special voter that can indicate the faulty module. The definition of checkpoints allows the synchronization of modules. Using dynamic partial reconfiguration, a faulty block is reconfigured and re-synchronization is made while the rest of the system works. This technique makes it possible to reconfigure a block of the TMR on-the-fly, so without loss of information. However, this method is not suitable for large system because of high hardware overhead of TMR. Moreover, big memory capacity is required for saving context at checkpoints. The checkpoints need to be carefully analyzed to maintain the functional services. Finally, this kind of system does not have wide applicability because it is adaptable only to Finite State Machine (FSM) where all the states are well defined during design phases.

2.5.4 Fault-Injection

To estimate the dependability of a system, it is essential to evaluate the effectiveness of fault mitigation schemes applied in the system. This involves the fault-injection experiments for estimating the fault coverage parameter. Generally, there are many ways to perform fault injection. A SEU in a memory cell can be modeled as a bit-flip. It can be easily injected in memory elements described in hardware description language, such as VHDL or Verilog, by performing a XOR operation between the original stored value and a mask, which defines the bit to be flipped [45]. With SRAM-based FPGAs, the bit-flip can occur in any bit of the bitstream, as a result, it is natural to perform the fault injection directly in the FPGA bitstream without understanding the signals design and

topology in the hardware description. The bitstream can be directly modified using any text editor on PC. This modification does not require the design structure understanding but the bitstream structure of related device in order that the modified bits are included in the data not the command parts. The JBits Software Development Kit (SDK) [46] is a set of Java classes which provide an Application Program Interface (API) into the Xilinx Virtex-II FPGA family bitstream. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams readback from actual hardware. This provides the capability of designing, adjusting and dynamically modifying circuits in Virtex devices. The JBits API gives the user the ability to configure CLBs directly. By using JBits classes, it is possible to perform a selective fault injection in the bitstream, which can also reduce the time spent in fault injection. The bitstream manipulations using either text editor or JBits SDK are offline method using external computers. Online method [47] an also be applied using available reconfigurable resources in the device to perform dynamic reconfiguration. A dedicated controller built in the matrix does the readback of the related frame, flip a bit inside the configuration frame data, then write the error-injected frame to the matrix using configuration scrub process. Before launching the next injection, the previous error should be removed to avoid the error accumulation. Error can be eliminated using configuration scrub again for non-persistent error or module-based partial reconfiguration for persistent error. If the design is not partitioned as module-based reconfigurable, the global reset must be applied to deal with persistent error.

2.6 Summary and Conclusions

Due to their flexibility, FPGAs are attractive for mission-critical embedded applications, but they are very susceptible to radiation and electromagnetic noise [48]. So their reliability could be insufficient unless some fault-tolerance techniques capable of mitigating soft errors are used. The major effects caused by them are known as *Single-Event Upsets (SEU)* or *soft errors*, because only some logic state(s) of memory element(s) are changed but the device itself is rarely permanently damaged. In FPGAs, SEUs may directly corrupt computation results or induce changes to configuration memory, that can cause changes in the functionality and performance degradation of the device. There are two ways to implement fault-tolerant circuits in SRAM-based FPGAs, as show in

the flowchart of Fig. 2.20. The first possibility is to design a new FPGA matrix composed of fault-tolerant elements. Certain FPGA companies offer also several radiation tolerant families which have fault-tolerant elements. The fault-tolerance schemes rely on redundancy but built inside the circuit and users do not need to manage that. This approach requires lots of engineers and development time. Furthermore, these special devices only have small markets like automobile and aerospace that is impossible to allow for big-volume production. Hence these particular FPGAs are usually much more expensive than commercial devices.

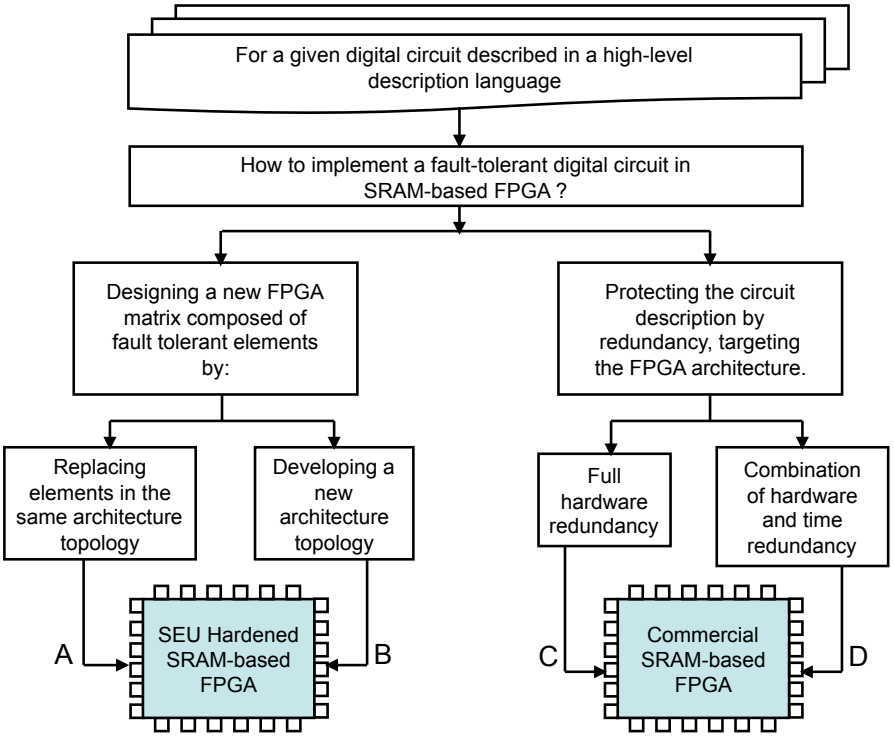


FIGURE 2.20: SEU and SET mitigation design flow

The second possibility is to use commercial FPGAs and apply high-level description techniques of the design before this description is synthesized in the FPGA. The user is responsible for hardening his design which do not require new chip fabrication. Fast turnaround of both hardware and software updates are guaranteed by the manufacturers during device life-time. Obsolete devices can be easily replaced by new ones thanks to the backward compatibility and hence it could attract acceptance from users.

Certain modern FPGAs provide dynamic reconfiguration offering not only a higher degree of flexibility, but also effective solutions for correcting SEU. Using dynamic reconfiguration will allow for on-line error detection during system operation, very fast fault location,

short correction, quick recovery from temporary failures, avoid long-period functional interruption, fast permanent-fault repair through reconfiguration and also maintain the service continuity which are crucial in highly reliable computing.

We propose, in the rest of the thesis document, our proposals combining various approaches with dynamic reconfiguration: effective fault detection, fast correction and recovery from error. We target during the thesis the fault raised by SEU in the configuration memory, in the reconfigurable elements and also single permanent faults. Throughout studies of the proposals are also realized.

Chapter 3

Fault-tolerance in dynamic multi-processor system-on-chip

3.1 Abstract

After studying state-of-the-art fault-tolerance solutions in reconfigurable architectures, we found that the architectural and the system level fault tolerance techniques are more interesting using available commercial FPGAs instead of directly using rad-hard FPGAs.

Currently, parallel computing is an important trend of embedded system. One possible response to increasing requirements in computational power is to distribute tasks over various processors and let these processors operate in parallel. Soft-core processors and FPGAs require low Non-Recurring Engineering costs to develop such multi-processors systems. Furthermore, certain FPGAs allow dynamic partial run-time reconfiguration, but their high sensitivity to electronic defects can cause the system disfunction. This chapter presents a fault-tolerant multi-processor system-on-chip (MPSoC) based on the dynamic reconfiguration of the entire platform. We construct an MPSoC system that combines the fault tolerance techniques at multiple levels: fault-tolerant dynamic MPSoC (FT-DyMPSoC). Also, a modification of the standard methodology of the run-time self-reconfiguration, who facilitates the complex modular concept design, is presented in this chapter.

3.2 Introduction

To meet increasingly complex challenges, the requirements for computing power increases very quickly. There are currently two main strategies to cope with this problem:

1. Operating frequency increase. This solution is significantly restricted by the frequency limit and depends on the manufacturing technology. Furthermore, this strategy is greedy in power consumption.

2. Using parallel processing. This technique increases the number of computing elements in the system. The mutiprocessor topology exploits this mechanism.

Contrary to the increase in frequency, in multiprocessor systems the number of processor can grow depending only on the device size. Many multiprocessor topologies can be implemented on an FPGA processors with *soft-core* available like Xilinx MicroBlaze, or hard-core processors like PowerPC [10]. Additionally new FPGAs offer high density resources for the integration of dedicated processing. There are two main paradigms of communication to exchange data in such a multiprocessor system: message passing and shared memory [49]. Each paradigm has its own pros and cons, but a *multi-cluster* (built by message passing communications) is more suitable for reconfigurable architectures [49]. This integration is facilitated by the existence and availability of point-to-point bus IP.

Several state-of-the-art solutions combine the benefits of parallel processing and reconfigurable architectures. However, none has implemented completely the dynamic partial reconfiguration. The major problem of these solutions comes from the vulnerability of certain parts of the circuit that are not integrated into reconfigurable areas. In [50], the authors construct a fault-tolerant reconfigurable multi-processor. A task can be dynamically *mapped* into the system without needing to restart the design process. However, the platform is deployed onto multiple FPGAs and does not implement the dynamic reconfiguration. In this system, each FPGA implements a processor. The equipment cost is a major issue in this system. The behavior change of nodes (to reduce a system failure) is obtained by reconfiguring totally the appropriate FPGA. This strategy thus leads to long-time reconfiguration. In addition, this system requires a large memory to store the *complete* bitstreams of all FPGAs for all task configurations. If two static tasks

are initially implemented on the FPGA and two tasks do not operate at the same time, the target FPGA chosen must be big enough.

The RAMPSoC architecture [51] implements the dynamic reconfiguration in a MPSoC system. The adaptation of the platform to the environment occurs at different levels. At the processor level, the software elements are modified by replacing the program within the processor memory. The process of dynamic reconfiguration is used for hardware accelerators to optimize performance of the application. The dedicated hardware components, such as the processor itself, the program memory and the memory controller are defined at design time. A change in functionality of RAMPSoC is achieved by reloading the program processors and dynamic reconfiguration of hardware accelerators. This system contains a master processor in charge of reconfiguring other processors. In fault-tolerance view, it is the main issue because if this processor fails, the whole system will not work correctly.

There are other platforms exploiting dynamically reconfigurable multi-processor architecture [52, 53]. But these works consider only the accelerator processors, dedicated hardware components of the processor cannot be changed dynamically. If errors occur in these components, they cannot be removed using partial reconfiguration.

We will introduce in this chapter our fault-tolerant dynamic multi-processor system on chip (FT-DyMPSoC). The system consists of several processors which are all dynamically reconfigurable to: i) deal with errors occurring in the processors; ii) dynamically modify the functionality of processor to adapt to various external events and constraints. The FT-DyMPSoC system is built-up exploiting the modified methodology that will also be presented in this chapter.

3.3 FT-DyMPSoC

The FT-DyMPSoC is constructed using dynamic partial reconfiguration of Xilinx Virtex FPGAs. The standard design flow for partially reconfigurable system is described in Section 2.2.2, p. 16. A typical partially reconfigurable system contains a processor who reads the configuration bitstreams in the annex memory and controls the reconfiguration process by sending partial bitstreams of different PRR to ICAP (Fig. 2.2, p. 16). This processor is usually considered in the static logic to control dynamically reconfigurable

resources. It is a drawback because if this processor is faulty, the whole system will not be able to continue operating. Our FT-DyMPSoC system eliminates this limitation. All the processors in our system are dynamically reconfigurable and can reconfigure any other processor. Each processor which consists of a MicroBlaze core, its program memory and the controller as well as the processor peripherals, is wrapped in to one reconfigurable module. In case of an error in one of these blocks, the processors are reconfigured to correct the error.

The FT-DyMPSoC system (Fig. 3.1) consists of several softcore processors MicroBlaze (μP1, μP2, μP3, etc.), a shared DDR2 SDRAM [54], an interrupt controller (INT) and a non-volatile memory Compact Flash. The processors communicate between them in a meshed topology using point-to-point Fast Simplex Links (FSL) [55]. The main advantage of this system is that all the processors are dynamically reconfigurable. This dynamically reconfigurable distributed MPSoC is able to handle the correct functionality of given tasks as well as the possibility to introduce new tasks of the entire system. The reconfiguration of a processor takes place when this processor fails or if we want to migrate some tasks to this processor. The processor reconfiguration can either change the program inside the MicroBlaze, or the hardware peripherals of MicroBlaze. It means that the processor's behavior can be changed at any time as long as the PRM's interfaces stay stable. If the reconfiguration of the processor 2 is required, the processor 1 will access the ICAP and control the appropriate bitstreams and vice-versa. It is the same mechanism with processor 3 and 4, which can reconfigure each other. Since there are only two ICAPs in a Virtex 5 device (ICAP0 and ICAP1 in **Fig. 3.1**) and four processors are capable of driving ICAPs, two controllers are required and instantiated in our design. Each controller is in fact a 2-inputs multiplexer, whose inputs are two MicroBlaze's connections, and the controller's output connects to ICAP. Therefore, the MicroBlaze 1 and MicroBlaze 2 can control ICAP0, and ICAP1 can be driven by MicroBlaze 3 and MicroBlaze 4. These 2 ICAPs cannot operate simultaneously. Thanks to the controller, we switch from one ICAP to the other by writing a configuration sequence using the active ICAP and gives the reconfiguration control to the other ICAP.

The interrupt controller itself is dynamically reconfigurable module. The duration interrupt strobes can be easily modified by reconfiguring the interrupt controller module with a new duration.

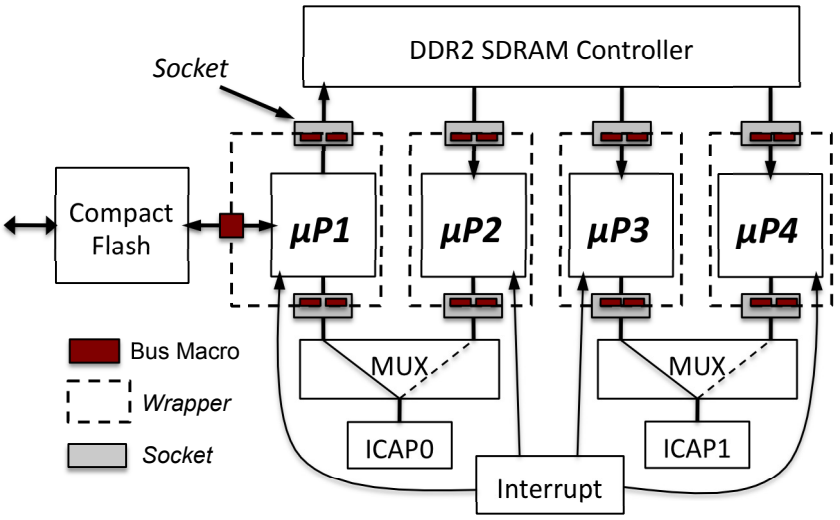


FIGURE 3.1: FT-DyMPSoC structure

The bitstream stock in the system is also indispensable. We use the ML506 card [20] which has a Virtex 5 XC5VSX50T device, and several on-board memory devices like a non-volatile CompactFlash (CF) Card and a DDR2 SDRAM. While partially reconfiguring the FPGA, each MicroBlaze needs to access partial bitstreams of the other processors, who are often in the CompactFlash memory. Since only one MicroBlaze is connected to the CF card (**Fig. 3.1**), there must be another shared memory to contain the partial bitstreams. Due to the limited size of available BRAM in the FPGA, the DDR2 SDRAM is selected.

In FT-DyMPSoC, the fault detection scheme is performed at two levels:

- Processor level: Each processor is hardened by lockstep mechanism of the processor core [56]. The processor core is duplicated, the 2 cores receive the same inputs and a comparator is added to report if there is a mismatch between outputs. This mismatch takes place when an SEU affects the computation results.

- MPSoC level: At this level, the fault mitigation strategy is realized by synchronizing among processors on exchanging detection frames via their communication (Figure 3.3). While the processors execute their tasks (T_1, T_2, T_3,...), the interrupt (INT) drives all the processors into the interrupt routine to exchange detection data, to build the connection statuses in form of a connection matrix and to share all the matrices among processors. The connection matrices are deployed to indicate if there are errors in the system. By varying the interrupt interval, the

system can auto-adapt to various constraints such as fault-tolerance constraints or real-time constraints of related user-applications.

The fault-tolerance strategy combining the processor level (block level) technique with MPSoC level (system level) scheme can provide a throughout fault diagnostic capability. If an error takes place in the processor (reported by the comparator) or in the communication link (via the synchronization process), the partial reconfiguration of the defected module will be launched to correct error. The error in one processor can be reported immediately to the others, but for coherence purpose among processors, it is announced at the synchronization phase. At the same time, the synchronization phase is able to detect if there are errors in the communication links.

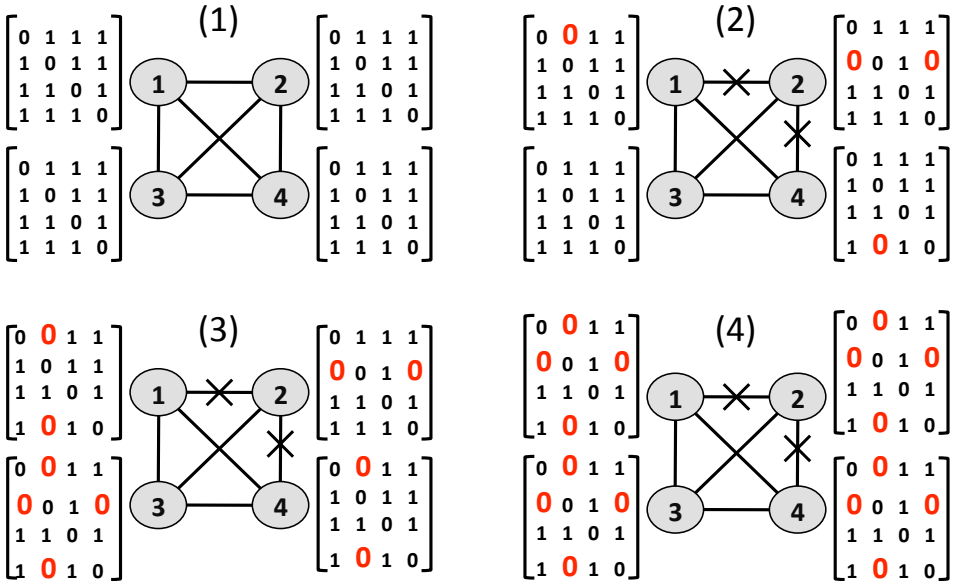


FIGURE 3.2: Connection matrices algorithm

The algorithm of connection matrices [50] used in the synchronization process allows for the error detection in FT-DyMPSoC. The principe of connection matrices is explained in Fig. 3.2. Each processor has its own connection matrix representing the connection status of the network. Fig. 3.2(1) shows four fully connected matrices. Fig. 3.2(2) shows the situation where two connections fail. Each processor detects the error and updates its own matrix by changing 1 to 0. In the next step in Fig. 3.2(3), all the neighbor processors are informed about the network topology change. To note that each processor store a different matrix. Finally, the step in Fig. 3.2(4) achieves the same matrix in all the processors. If all the connections of one processor fail, it is highly probable that it is the processor which fails.

The Gantt diagram of this process for four processors is depicted in Fig. 3.3. At start-up, the processor 1 copies the partial bitstreams from the CF Card to the DDR2 SDRAM in order that all processors can access the data thanks to the DDR2 multi-port memory controller (MPMC) [54]. Perodically the processors synchronize themselves via their FSL connections. To avoid timing delay of the synchronization, we use a unified interruption source for all the processors. The interrupt ticks are independent of the processor clocks so even if the processors operate at different frequencies or use different operating systems with other timing constraints, they can be synchronized.

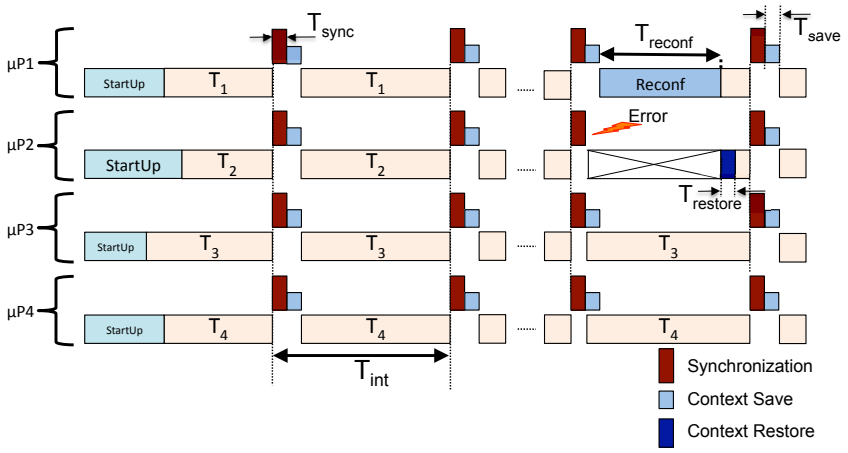


FIGURE 3.3: Timing diagram of FT-DyMPSoC

The synchronization processes are done regularly in the interrupt routines. Each interrupt strobe (T_{int} in Fig. 3.3), the Interrupt Controller generates a signal in order that all the processors enter the interrupt handler. In the handler, each processor sends to the others its synchronization frames and receives frames from the others. If a processor does not receive the frame from another for more than 100 execution cycles, that means that connection is corrupted due to an error. Via the operational links, the connection matrices are updated until they are identical in all the processors. If all the links of one processor fail, this processor is faulty. The reconfiguration is realized in round-robin order: μP1 reconfigures μP2 (**Reconf** phase in **Fig. 3.3**), μP2 reconfigures μP3, μP3 reconfigures μP4 and μP4 reconfigures μP1 The strobe length (T_{int}) is chosen at the design phase, but it can be modified by the reconfiguration to adapt to various application scenario and different bitstream sizes. To do so, all the processors have to de-activate theirs interrupt routines while one processor reconfigures the interrupt controller with a new strobe period.

The status of the two ICAPs as well as the two multiplexers are synchronized thanks to the same mechanism as the connection matrix. Which processor connects to which ICAP, which ICAP is actually active, all this information are known and determined through out the network during the synchronization phase.

Depending on the error type, different reconfiguration techniques are selected: normal partial reconfiguration for a temporary fault or *tiling* technique [57, 58] for a permanent fault. If the error persists after completing the fault correction process (i.e. for a temporary fault), the permanent fault is declared. The principle of *tiling* avoids usage of the faulty zone of the FPGA by pre-compiling the same design with various configurations. Each configuration has a blank zone and the permanent fault can be masked by charging the appropriate configuration in which the blank zone overlaps the faulty area. Each configuration of the same design has its own bitstream and the fault masking process is done by downloading the appropriate bitstream through ICAP in a partial reconfiguration procedure. The bitstream generation for various configurations is done in our system using a basic placement constraint PROHIBIT [59] which is assigned through the user placement constraint file provided to the synthesis tool. The blank rectangle zone is defined by the coordinates X and Y of two points A and B. By varying these coordinates, the blank zone will be moved inside the PRR (Fig. 3.4).

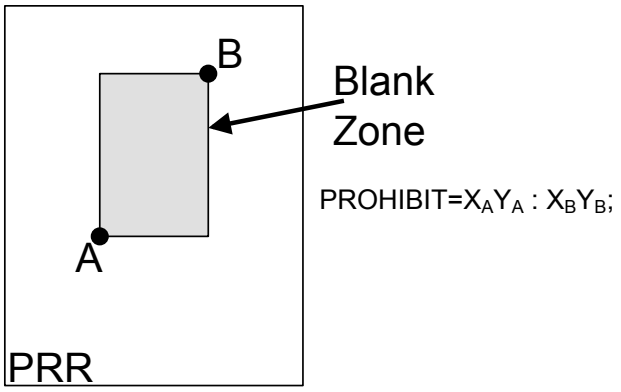


FIGURE 3.4: *Tiling* technique using PROHIBIT

Figure 3.5 illustrates the fault-tolerance flow chart of the FT-DyMPSoC system. Periodically the processors synchronize to detect errors. If there is no error, the processor contexts are saved into the DDR2 SDRAM (Checkpointing in Section 2.4.3.2 p. 30). If an error is detected, the fault diagnosis is launched to identify if the error is permanent or not. If it is a temporary error, the partial reconfiguration corrects it by downloading the appropriate bitstream to the ICAP. If the permanent error is identified, *tiling* technique

is used to deal with this error. After that, the last saved processor context is restored to resume the task in this processor (Roll-back).

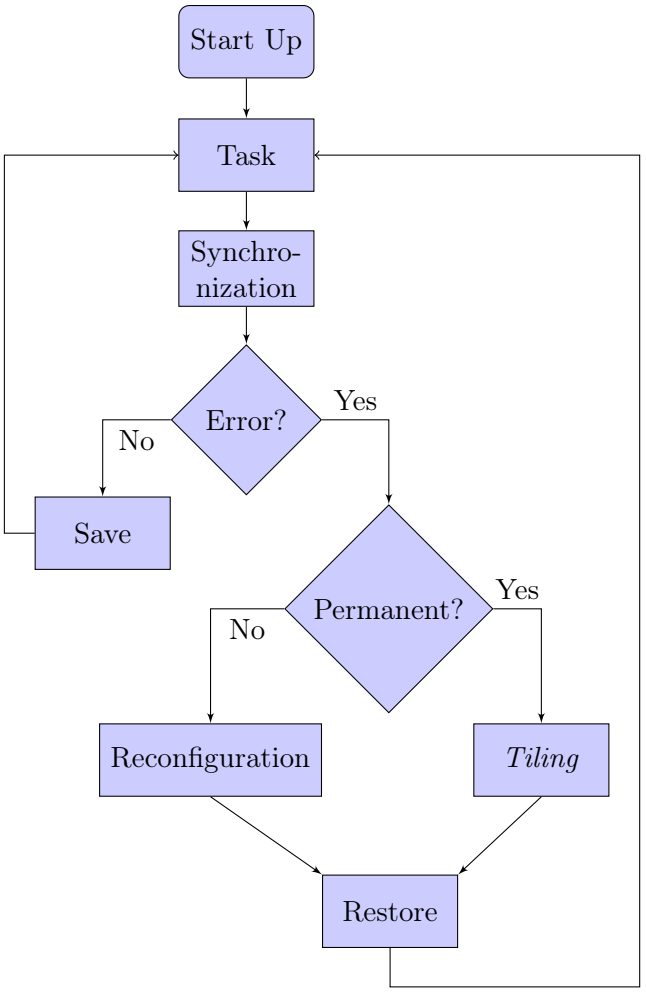


FIGURE 3.5: Fault mitigation scheme

3.4 Design flow modification

However, the Xilinx PlanAhead tool does not allow to set a group of several IPs (who are all subset of top level instance) as one reconfigurable module. Need-by, the design flow needs to be modified to respond to the proposed system requirements.

To cope with the standard design flow drawback, we have exploited and modified this flow. The modified design methodology is based on modular design concept which is presented in [16]. This feature allows designs to be splitted into portions that are independently synthesized, coded, placed, routed, and mapped. The modification necessitates the definition of the *Socket* and the *Wrapper* components.

3.4.1 Design flow modification

The first challenge springs from the structure of the MicroBlaze IP cores. The MicroBlaze IP is reconfigured with its peripherals such as: BRAM program memory, BRAM controller, PLB bus [20] and so on (Fig. 2.2). At this moment, all the IPs are "equal" subset of top level instance (Fig. 3.6.a).

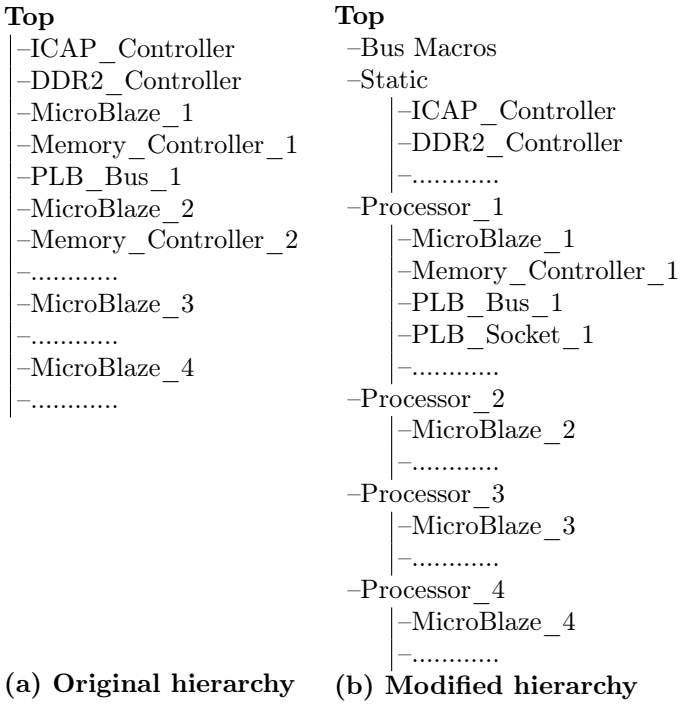


FIGURE 3.6: Design hierarchy

But the Xilinx's PlanAhead tool does not allow to set a group of several IPs (who are all subset of top level instance) as one reconfigurable module. So the system hierarchy needs to be modified as shown in Fig. 3.6(b).

At the first design phase (Fig. 3.7), a complete static design is constructed to verify the operation of the whole system. The design procedure of this stage follows the standard design flow for Xilinx devices (Section 2.2.2, p. 16). After validating the functionalities of the shared DDR2 SDRAM controller, ICAP controllers and FSL connections, we have an operational system who is still fully static. The design has all the functional blocks and the required elements of the fully dynamic system that we want to construct afterwards.

The second design phase breaks the system into several sections consisting of all reconfigurable parts and the static portion. Each module is coded, synthesized and mapped following the standard Xilinx design flow (Phase III of Fig. 3.7).

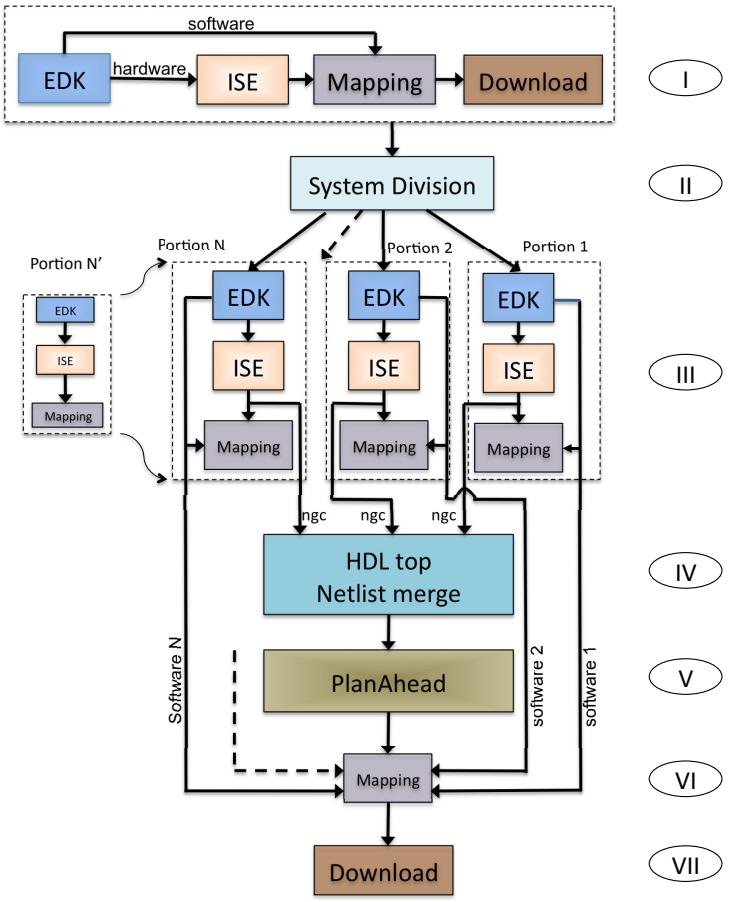


FIGURE 3.7: Modified design flow for complex dynamically reconfigurable system

After the system fragmentation, some components, such as ICAP controller or DDR2 controller, are no longer connected to the PLB bus. Because these controllers are in the static part, while the PLB buses are attached to their own MicroBlaze in the reconfigurable module. But the connections of PLB buses cannot be open, so we need to add PLB_Socket presented below.

At the end of stage III, the netlists of all PRMs and the static logic are generated. Then at stage IV, we instantiate these modules at top level by creating the module wrappers. After the wrappers of four processors and the static part are realized, the Bus Macros are instantiated in order to connect proper signals between system parts. Now the Top level instance has five subsets: static part and four processors (Fig. 3.6.b). These subsets are black boxes and their netlists will be invoked in PlanAhead (Stage V). At this stage, the four processors can be set as reconfigurable modules. They can be floorplanned to generate hardware bitstreams of the whole system. The step VI of the flow merge the hardware parts with modified MicroBlaze programs to form the final bitstreams including the initial full bitstream and all partial bitstreams.

The presence of static logic is also essential in the design. In the static region, the Bus Macros are instantiated, but they are located in the PRRs at the place and route stage to keep the interface logic unchanged during reconfiguration. The reconfiguration controller is mapped in this region. The two multiplexers are instantiated in the static part of the design.

3.4.2 *Socket*

The objective of building *socket* is to introduce the presence of peripherals who exist in other subsystems. Since the DDR2 Controller, ICAP Controller (who connected to PLB Bus [20]) and FSL Buses are shared among four processor subsystems, they must be instantiated in another subsystem. So at the subsystem design phase, each processor has no longer direct connections with these controllers. This generate hardware synthesis errors because the connections of PLB and FSL bus must not be open. A *socket* is a mask that has one end connected to the bus to conserve the connection, the other end is open in order to be connected to Bus Macros afterwards. Hence, the *socket* is now a new peripheral that helps filling up the complete component list of a processor. Especially for the PLB bus, since it is a multi-slave bus, so there is an address translator implemented in the *socket* so that the processor can communicate with the appropriate peripheral of the PLB bus. Another role of *sockets* is to make a fake peripheral appear in the address memory of the processor that helps the processor to drive the missing peripheral with the software program. This requires a little modification in the program of the processor.

3.4.3 *Wrapper*

All the connection interfaces of each reconfigurable modules are declared in its *wrapper*, so the *wrappers* of different PRMs for a PRR must be the same. A *wrapper* is in fact a black-box written in HDL language and describes all the interface of a PRM with the static logic. We have written four *wrappers* for the four processors respectively (dashed rectangles in Fig. 3.1) in order that each processor can be designed independently using standard design flow (Sec. Intro). A big advantage of using *wrappers* is that we do not have to take care about the structure complexity of a reconfigurable module. All the difficulties while building a complex module is managed by the Xilinx FPGA CAD tools, we just need to keep the substitute PRM's *wrapper* to be the same as the original one.

One advantage of this flow is the facility of the HDL *wrappers* creation for all the reconfigurable modules that eases the design of complex modular system. Only a little modification of the *wrappers* is needed to adapt a substitute reconfigurable module to the system. If we want to replace the portion N by the portion N' (regardless of the module complexity), we just need to keep the connection name unchanged in order that its HDL wrapper is the same as the original one. All the structure complexity of the module N' is managed by the CAD tools.

3.5 FT-DyMPSoC amelioration

We have enriched our FT-DyMPSoC system using a NoC instead of point-to-point links. The Reliable and Reconfigurable Dynamic Architecture (Re^2DA) system uses a NoC-based network called DRAFT (Dynamic Reconfigurable Adaptive Fat-Tree) [60]. The fault-tolerance at multi-platform level has also been taken into concern providing a highly reliable system based on FT-DyMPSoC. The multi-platform consists of several FPGAs and each FPGA is an FT-DyMPSoC system. The fault-tolerance is managed at MPSoC level as well as multi-platform level (Overall system level).

3.5.1 Re^2DA system

Since the complexity of MPSoC systems continuously increase with the parallelism requirements of the applications, interconnection becomes a concern leading to the use on Networks on Chip (NoCs). NoCs, like Dynamic Reconfigurable Adaptive Fat-Tree (DRAFT) [60], provide flexibility and scalability to the applications. However, shared communication architecture directly impacts the fault tolerance capability of the system. We hence use DRAFT as the central communication architecture among the processors in an MPSoC system. Beside the user-application, DRAFT is also used for fault-tolerance objective. The envisaged user-application is constituted of 4 automotive-aimed tasks that have different security levels. The fault-tolerance mechanism guarantees the system functionality depending on these security levels, the highest security task will be managed first.

Currently, a lot of interconnection architectures were designed but their topologies can be classified in a few basic families [61]. Main families are the bus-based architectures,

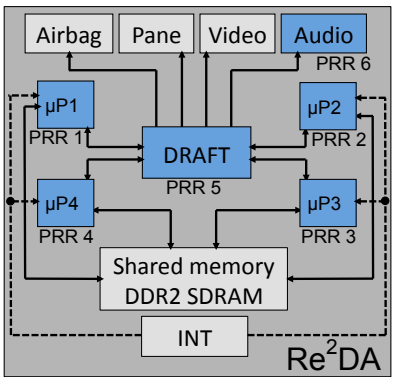


FIGURE 3.8: Internal structure of Re^{2}DA

the matrix-based structures like meshes and torus, the tree-based topologies like fat-trees, the rings based ones, and finally the custom networks. However, the most popular structures for an implementation into the framework of systems using dynamic reconfiguration are the bus-based, the meshes, and the fat-trees [62]. We use in this work the DRAFT network [60] which is adapted from a fat-tree based structure and was designed especially to be implemented in system performing dynamic reconfigurations. Futhermore, it provides best network performances while consumming less hardware resources than other mesh or fat-tree based networks. Its performances in terms of latencies and aggregative bandwidth are interesting for both the transmission of a video flow and the transit of control words which are used for fault tolerance purpose. That is the reason why the DRAFT network was choosen for our dynamically reconfigurable system.

The fault detection mechanism of this system is also based on the connection matrices. However, the matrices are adapted to the DRAFT network. Using these modified matrices, the system is capable of detecting errors whether in the processors, in the DRAFT or in the connections of DRAFT. Once an error is detected, the dynamic reconfiguration of the related module is launched to correct the error. If via the connection matrices using DRAFT, one processor is reported to be disconnected from the other, it is possible that the processor fails or the link is erroneous. In that case, the DDR is used to distinguish the error. All the processors write a flag in their specific locations in the DDR and read the flags of the others. If the flag of one processor is not found, this processor is faulty. If the flag is successfully found, the processor is still working, so the correspondent link fails.

The processors periodically synchronize themselves, share the connection matrices, and

save the processor contexts for error detection purpose during the saving phases (Checkpointing). The contexts that relieve the state of the processors are regularly saved in the DDR by the processors themselves. After the reconfiguration of a faulty processor, the state of this processor needs to be recovered to the last saved state before the error occurrence. So the last correct context will be flashed back to the processor just after its reconfiguration (Rollback).

3.5.2 Multi-FPGA platform

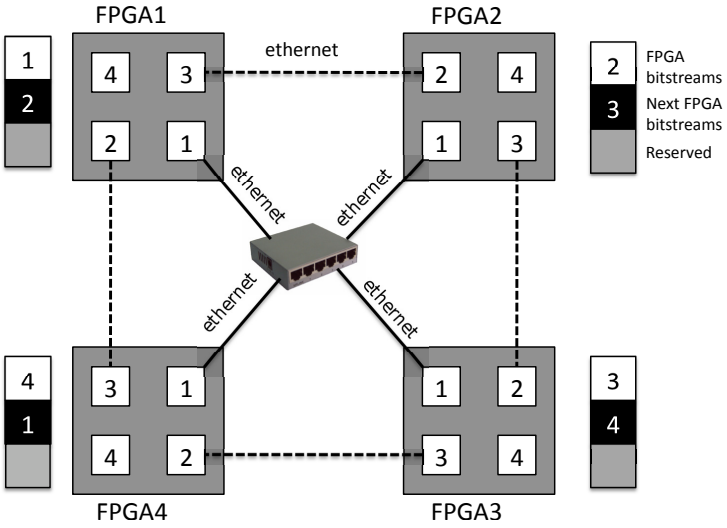


FIGURE 3.9: Fault-tolerant multi-FPGA platform

We have started to implement the fault-tolerance schemes at multi-FPGA level (Figure 3.9). The system consists of four FPGAs connected together using two Ethernet communication (in future development one will be based on PLC interface, while the other will be constructed on RF connections as studied in the CIFAER project [5]). The first network is routed via a network switch, the other network form a ring topology for the fault-tolerance purpose. The Ethernet protocol is built by Ethernet controller as MicroBlaze hardware peripherals and LightWeight IP [63] as the software library. The lwIP is an open-source stack using TCP/IP protocol, which can be easily adapted to PLC and wireless modem. Each FPGA contains an FT-DyMPSoC system.

On the overall system, each FPGA is interfaced with a memory that can be accessed by all the processors inside the same FPGA. This memory is partitioned into three segments (Fig. 3.9):

- One for saving all the bitstreams and the software contexts of all the processors of this particular FPGA.
- One for saving all the bitstreams of the next FPGA in the ring network.
- One reserved and used in case of failure occurrence in the system. This segment helps to transfer the bitstreams and contexts between different FPGAs.

The memory segmentation guarantees the existence of at least one copy of all the bitstreams over the whole network.

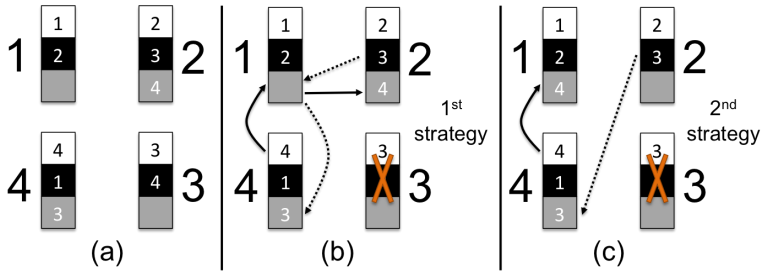


FIGURE 3.10: Fault recovery strategies.

As we can see in Fig. 3.10(a), the bitstream of each FPGA is present in its local memory and also in the local memory of the previous FPGA in the ring topology. For example, FPGA1 stores its own bitstream 1 and the bitstream 2, FPGA2 stores bitstream 2 and bitstream 3 and so on. These copyies will be used in case of system failure, and permit fast context switching.

The fault-tolerance degrees are maintained at two levels in the system. The Intra-FPGA level corresponds to the fault-tolerance strategy inside each FPGA, and is related to the design of the FT-DyMPSoC system. The fault-mitigation strategy is realized using the connection matrices algorithm, and fault are mitigated by using dynamic reconfiguration at the processors level. The second level called Inter-FPGA level corresponds to the overall system presented in Fig. 3.9. To detect error in the overall network, all the FPGAs exchange frequently among them detection frames. These frames contain the software contexts of the four MicroBlazes of each FPGAs. On one hand, this helps detecting error in the network. On the other hand, including the contexts within the detection frame will help to resume the tasks of a faulty FPGA on another FPGA. During the exchange if the contexts of one FPGA (i.e. FPGA3 in Fig. 3.10) are not received by the others circuits, the FPGA3 is declared faulty. There are 2 possibilities: the

MicroBlaze 1 (supporting the interface to the network) of FPGA3 is faulty, causing the communication lost of this FPGA, or the whole FPGA3 is faulty. In order to distinguish these 2 possibilities, the secondary ethernet links is used. FPGA2 and FPGA4 try to communicate with MicroBlaze 2 and 3 of FPGA3. If these communications fails, the whole FPGA3 is declared defected, if not, only the MicroBlaze 1 is defected.

If only one MicroBlaze inside one FPGA fails, this error is managed thanks to dynamic reconfiguration of this processor or by using task migration within the MPSoC system. The error is managed at the FPGA level. If the whole FPGA fails the task migration concerns the overall circuit. In this case, the tasks of the FPGA3 need to be dispatched across the remaining circuits. If the system cannot manage all the tasks with one missing FPGA, task priority needs to be defined and used to maintain critical services for example. In this case, arbitration on the running tasks needs to be executed, and reconfiguration of the remaining FPGA is launched. If one FPGA is lost, we need to maintain the two bitstreams copy stored in the faulty FPGA. For example, if the FPGA3 is lost (Fig. 3.10), the copies of bitstream 3 and 4 are inaccessible requiring a clone of bitstream 3 and bitstream 4. We propose here 2 strategies delivering the bitstream 3 and 4 to other FPGAs:

1. The first strategy uses only the secondary communication media. We need to use FPGA1 reserved segment as intermediate medium. First the bitstream 4 is copied from FPGA 4 to FPGA1 reserved segment, then to FPGA2. Afterwards, bitstream 3 is copied from FPGA2 to FPGA1, then to FPGA4.

2. The second strategy requires both communication media. Bitstream 4 is copied from FPGA4 to FPGA1 using direct Ethernet link. Simultaneously, bitstream 3 is copied from FPGA2 to FPGA4 using the primary Ethernet via the switch.

In case the Ethernet switch fails, all the primary Ethernet connections are defected; this leads to a connection loss between all the FPGAs. At this moment all circuits switch to the ring topology. The second network will then ensure proper operation of the overall system. The use of redundancy of the network, coupled with the new dynamically reconfigurable paradigm permits to construct highly reliable system.

3.6 Implementation details

We have implemented the FT-DyMPSoC system on the Virtex-5 XC5VSXT50T. An applicative system is constructed with three dynamic MicroBlaze processors and two audio filters also in dynamically reconfigurable zones. Three processors operating at 100MHz. The processor 1 connects to a camera, controls the video acquisition and then displays it on a screen. This processor is used for **ADAS** (Advanced Driver Assistance Systems) objectives such as collision avoidance, night vision or blind spot detection. The processor 2 controls the audio configurations by choosing the appropriate filters (all-pass, high-pass, low-pass or band-pass) and reconfigures them on-the-fly. The processor 3 controls the vehicle window pane by the user buttons and releases the airbag in case of collision detection.

The implementation view of the system is displayed in Figure 3.11. The largest component is the DDR2 SDRAM controller because it carries out big tasks such as for saving processor contexts, for buffering video frames and for the reconfiguration processes. This controller is static and manages 512 MBytes of DDR2 SDRAM.

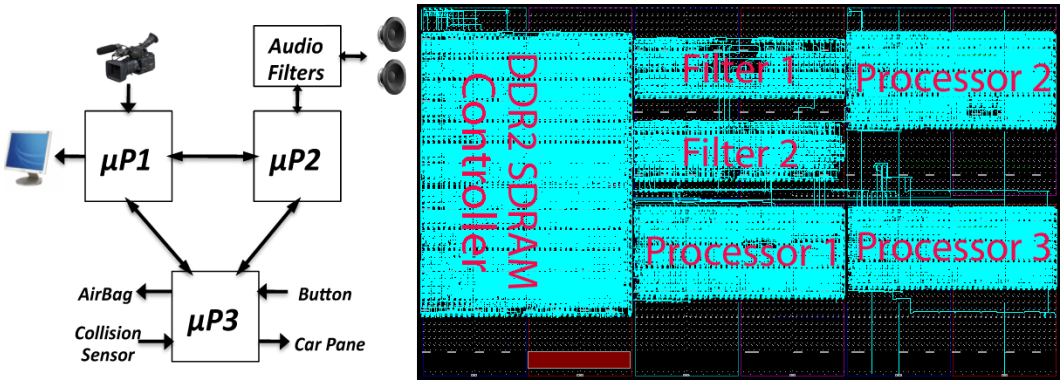


FIGURE 3.11: FPGA Editor view of implemented system with automotive applications

As shown in Table 3.1, the processor bitstream sizes are respectively 194, 177 and 158 KBytes for processor 1, 2 and 3. The different bitstream sizes of processors are due to the different implemented functionalities, hence induce different hardware requirements for each processor. With the same functionality, the bitstream size depends significantly on the PRR size. The bitstream size does not depend on the PRR position in the FPGA matrix. However, it depends on the number of affected clock regions (base regions), even if the PRR size stays the same. To minimize the bitstream size of a given amount of

resources, it is necessary to span the PRR on the clock region boundaries, thus reduce the reconfiguration frames to be reconfigured.

TABLE 3.1: System hardware resources

	LUT	FF	Slice	BRAM	Bitstream Size (KBytes)
XC5VSX50T	32640	32640	8160	132	
Processor 1	3840	3840	960	16	194
Processor 2	3360	3360	840	16	177
Processor 3	2880	2880	720	16	158
Socket	8	8	2	0	
Wrapper	0	0	0	0	

A *socket* in FT-DyMPSoC requires only 8 LUTs because it only plays the role of connecting elements, without realizing the computational functionalities. Moreover, a*wrapper* does not consume any resource since it is just a virtual component that helps easing the design flow. Consequently, the proposed flow modification requires very few hardware resources, but considerably accelerate the system construction phase.

Recall that the bitstreams need to be copied in the DDR2 so as to be accessed by all processors. By varying the size of the PRRs, we get different bitstream sizes. We have measured the reconfiguration duration from the CF (CF2ICAP in Table 3.2) with different sizes of bitstream, also the time to copy the bitstreams to the DDR2 (CF2DDR) and the duration to reconfigure the system from the DDR2 (DDR2ICAP). The reconfiguration duration from the DDR2 is about 75 % faster than from the CF card. But the procedures that copy the bitstream to the DDR2 and read from the DDR2 can increase the fault probabilities during bitstream manipulations. To respond to this issue, the error detection and correction codes (like parity, Hamming or CRC codes) can overcome this issue. However, the overhead can raise the bitstream size so the needed time to copy to and reconfigure from the DDR2 as well.

The required reconfiguration times using DDR2 memory for three processors are respectively 99, 91 and 80 ms. So the interrupt interval is chosen at 100 ms that is superior to the reconfiguration times of processors in order that the interrupt routine does not disrupt the reconfiguration process. For other systems which have other fault-tolerance or real-time constraint, FT-DyMPSoC can quickly adapt by reconfiguring dynamically the interrupt controller with appropriate intervals.

TABLE 3.2: Bitstream manipulation time

Bitstream Size (KBytes)	CF2ICAP (ms)	CF2DDR (ms)	DDR2ICAP (ms)
158	**326**	**281**	**80**
177	**360**	**310**	**91**
194	**414**	**357**	**99**
214	448	386	109
253	529	456	129
384	797	692	196
560	1167	1009	286
796	1654	1434	407

The process copying bitstreams of all the processors is extremely long (281 + 310 + 357 = 948 ms), but this stage takes place only once when the system boots, afterwards the processors reconfigure from the DDR2 with shorter required time. In case of error in one processor, only 2 processors hang their tasks for nearly one interrupt interval to correct the error, while the 2 other processors continue executing their tasks.

The time needed to save one MicroBlaze context to the DDR2 is 4,5 μs, and to restore is 7,65 μs. So in correct function of the whole system, one checkpointing process needs 5,5 μs.

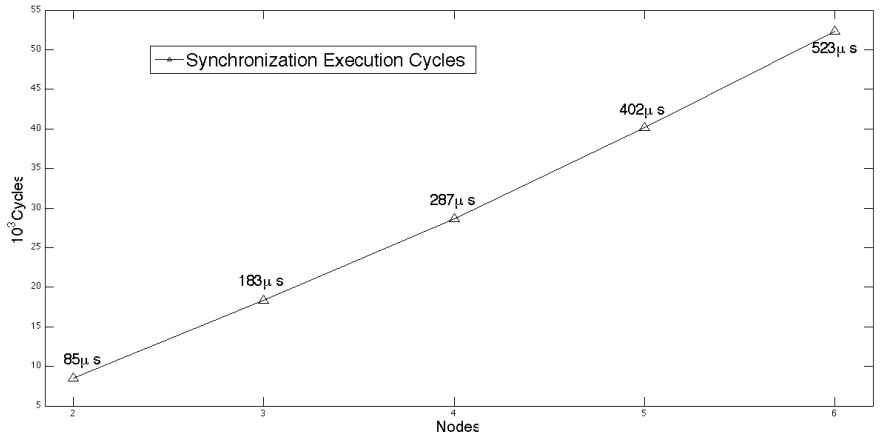


FIGURE 3.12: Synchronization duration

The synchronization routine in a system of 2 processors running at 100 MHz frequency needs 85 μs (8492 cycles). With 3, 4, 5 and 6 processors, it takes 183 μs, 287 μs, 402 μs and 523 μs respectively (Fig. 3.12). Even the required synchronization time rises considerably when the number of processors increases, the synchronization duration is still significantly inferior to the interrupt interval (183 μs with 3 processors compared to

100 ms of the interrupt interval). This synchronization process plus the context recovery process have not much timing contribution to the system.

In order to measure the execution times of different operation in the processors, a timer [64] is added to the system. This timer is a hardware peripheral and controlled by the processor 1. The number of clock cycles that is counted by the timer can be easily read using the processor. Thus the measured time of the required operation is achieved by starting and stopping the timer respectively, at the beginning and the end of the related operation.

Table 3.3 compare our FT-DyMPSoC system with other fault-tolerance techniques for reconfigurable architectures. We take the reference of the target systems with the same functionalities as FT-DyMPSoC. The fault coverage evaluates the ratio of mitigated faults. The continuity depicts the capability of resuming the task in progress at the paused point due to fault occurrence. The last column represents the ability to deal with permanent fault.

TABLE 3.3: Comparison of different fault-tolerance techniques

	Hardware resources	Fault coverage	Continuity	Permanent fault
scrubbing	>1x	<100%	N.A.	No
TMR	>3x	100%	+++	No
DWC	>2x	100%	-	No
FT-DyMPSoC	<1.6x	~100%	++	Yes

Scrubbing requires obviously the less hardware resources because this technique necessitates only a small controller executing the task of *scrubbing*.

Event FT-DyMPSoC uses lockstep mechanism with resource duplication for processors, the areas using for processors are not duplicated because of higher utilization rate in the PRR. Comparing to the proposal in [65] in which the processor is triplicated in the MPSoC context, big hardware overhead could be an high bottleneck. Additionally, the communication links between processors are not duplicated thanks to the connection matrix algorithm. So, the hardware resources required for FT-DyMPSoC is 1.6 times compared to a basic system.

The hardware redundancy techniques require significantly much hardware resources because of their triplication or duplication as well as the voter or comparator.

In term of fault coverage, the redundancy techniques provide the best results as the fault is detected and corrected instantly. Our FT-DyMPSoC system offers a fault coverage nearly 100% depending on the synchronization rate. Anyway FT-DyMPSoC offers better fault coverage than *scrubbing*, since the synchronization rate is always superior than the *scrubbing* rate.

It is difficult to identify the continuity of *scrubbing*. The TMR scheme evidently provides the best continuity because the task can continue its execution in spite of the fault occurrence. Whereas DWC scheme offers a low continuity since both the two modules in the duplication needs to be refreshed to correct the error. FT-DyMPSoC provides a good continuity with the rollback strategy applied in the system.

And finally, among these techniques, FT-DyMPSoC is the only system that can deal with permanent faults thanks to the *tiling* (p. 33) implemented.

Re^{2}DA system In this implementation, there are 4 MicroBlazes and a DRAFT that operate at 100 MHz while the DDR2 operates at 200 MHz. The PRR containing DRAFT takes 1920 LUTs of which needs 53 ms to be reconfigured while PRR for each processor consumes 2240 LUTs (the reconfiguration time is 59 ms). In good condition that all the 4 processors and DRAFT function correctly, the synchronization process takes 1 μs to finish. In case of error occurring either in the DRAFT or in a processor, the processor task rupture duration is less than 60 ms which is acceptable for the application scenario.

Multi-FPGA platform A platform composed of three Virtex-5 XC5VSXT50T using ethernet communication is implemented. Each FPGA contains an FT-DyMPSoC system of 4 MicroBlazes. The MicroBlazes run at 100 MHz. The inter-FPGA communication is done via the TCP/IP protocol of lwIP library controlled by a MicroBlaze in each FPGA. The lwIP can operate in two modes: RAW mode and socket mode.

The RAW API provides a callback style interface to the application. Applications using the RAW API register callback functions to be called on significant events like accept, read or write. The socket mode provides a simple API that blocks on socket reads and writes until they are complete. However, the socket API requires many pieces to achieve this and this API contains significant overhead for all operations, so it is slow.

Although the RAW API is more complex than the SOCKET API, it provides much higher throughput because it does not have a high overhead.

TABLE 3.4: Ethernet performance measurement

RAW Mode		**Socket Mode**	
RX	TX	RX	TX
120 Mbps	104 Mbps	20 Mbps	31.2 Mbps

To help the platform exploration, performance measurements are applied in the multi-FPGA platform in both two modes (Table 3.4). The receive and transmit throughput using the RAW mode are much higher than the socket mode. The bitstream size of a MicroBlaze is about 170 KBytes which needs $\frac{170KBytes \times 8bit}{104Mbps \times 10^3} = 13$ ms to transfer from one FPGA to another one using the RAW mode.

32 KBytes software code of a MicroBlaze corresponds to a 5 Kbit bitstream. Reconfiguring using this bitstream will only change the software running on the MicroBlaze without affecting the hardware part. The software context of a MicroBlaze has the size of 1 Kbits. So, if we want to resume a task of a MicroBlaze from an FPGA on another FPGA, only the software bitstream (5 Kbits) and the software context (1 Kbits) needs to be transfered via ethernet network which requires only 58 μs.

3.7 Conclusion

In this chapter, we have presented a fully dynamic multi-processor system in context of dynamically reconfigurable architecture who can deal with the possible faults in the reconfigurable architecture with low timing overhead. We have also proposed and validated a CAD extension for managing the reconfiguration of complex reconfigurable modules. The methodology facilitates the creation of the reconfigurable module *wrappers*. We have developed the bus *sockets* and also the ICAP controller.

An alternative version of FT-DyMPSoC using DRAFT—Re^{2}DA is also presented in this chapter. No more hardware overhead is required to maintain the fault tolerance technique in the system. All the needed hardware resources are used for constructing basic dynamic system. The fault tolerance schemes lead to a slight software overhead which have small effects in the system function. Using DRAFT as a centralized interconnection guarantees the system communication capacity, while contributing actively to the system services

continuity with low overheads and provide considerable flexibility and scalability for future larger system.

The proposed multi-FPGA platform provides a high-performance, flexible solution. The cross-level fault-tolerance strategy ensures the correct functioning of the entire platform in spite of fault occurrence. This system exploiting the dynamic reconfiguration can ameliorate the ReCoNets system: lower hardware overhead, more flexible, shorter functional interrupt, better service continuity.

Throughout the duration of the thesis, an ADAS application for automotive domain was considered and realized in the MPSoC systems. Each processor carries out an automotive task with a particular priority. The tasks are: airbag management, window pane control; blind-spot detection; and infotainment missions (audio and video management). In case of failure, the critical service should be maintained depending on the task priority; the airbag management and window pane control should be prior to the others.

All proposed system topologies can auto-adapt to either various fault-tolerance constraints: different error rates, reliability requirements, etc. or many user-application constraints by regulating the interrupt interval on-the-fly.

Chapter 4

Analytical Model

4.1 Abstract

Implementing dynamic multi-processor system-on-a-chip (MPSoC) using Commercial Off-The-Shelf (COTS) partially reconfigurable architectures is one feasible solution to respond to the computational power needs with low Non-Recurring Engineering costs requirement. However, the high sensitivity of commercial FPGAs to electronic defects urges system designers to include fault-tolerance schemes to prevent their architectures from being defective during products life-time. This additional item can decrease the system computing power. Therefore, the need of an analytical model to analysis the effect of fault mitigation schemes to system performance is one urgent requirement when building such fault-tolerant system. This chapter presents an analytical approach for the fault-tolerant dynamic multiprocessor system-on-a-chip (FT-DyMPSoC) which is able to resist to Single Event Upset—predominant fault in FPGAs. The analytical model is introduced to assess the performance, the reliability and the trade-off of a fault-tolerant MPSoC system. Also, some comparisons with classical fault-tolerance solutions to enhance our solution advantages are given.

4.2 Introduction

The increasing needs of performance and scalability orient designers toward using commercial reconfigurable architectures such as available commercial FPGAs. Furthermore,

applying dynamically reconfigurable architectures can provide a higher degree of flexibility, scalability and simultaneously maintain the computing power.

However, deep sub-micron system-on-a-chip aggravates the reliability problem in which Single Event Upsets (SEU) are one major source of concern. An SEU takes place when the radiation causes a bit-flip in some latches (1 to 0 or vice versa). In the available commercial FPGAs such as Virtex 5 Series, predominant fault source is SEU in configuration memory [66] that inverts one bit in the configuration memory, this undesired modification may cause the dysfunction of the target design. These susceptibilities delay their uses in consumer and industrial products (automotive, aerospace,...) unless they apply fault-tolerance techniques to mitigate the error effects. There are various solutions to mitigate, mask, detect and correct error like hardware redundancy (Duplication with Comparison, Triple Modular Redundancy) [67], time redundacy or some configuration memory techniques like readback [38], *scrubbing* or readback and partial reconfiguration combination [40].

Nevertheless, the fault-tolerance itself on the one hand can increase the system reliability, on the other hand can affect the overall system in term of hardware and timing/software overhead. The hardware overhead may restrict the available resources for designing user applications. Besides, the timing overhead limits the execution time for application tasks because of executing fault-tolerance schemes, consequently affects the entire system performance.

On designing a fault-tolerant system, the trade-off between performance and reliability becomes a considerable factor. So the need of an analytic model to evaluate the effects of fault-tolerance schemes on the system performance becomes important.

This chapter presents our analytical approach to evaluate the system performance and reliability of our FT-DyMPSoC system that is able to deal with SEU by exploiting the dynamic partial reconfiguration. The analytical model is also used to compare with the classical *scrubbing* solution.

4.3 Analytical Model

4.3.1 General definitions

Definition 1. System availability is defined as:

$$Sys\_Avail = \{1 - X_{detect} - X_{correct} - X_{recover} - X_{correlation}\} \tag{4.1}$$

The system availability Sys_Avail is the percentage of available system execution time per total time. X_{detect}, $X_{correct}$ and $X_{recover}$ are process factors respectively representing the percentage of time that have been spent for detecting, correcting and recovering from errors. In a non-fault-tolerant system, the system availability is equal to 1 since all the time is reserved for executing user-application tasks (X_{detect}, $X_{correct}$ and $X_{recover}$ are equal to 0). Besides, there is sometimes a correlation factor ($X_{correlation}$) among processing elements. The correlation is due to the fact that the detection, correction or recovery processes in one module may affect other modules operations in the system, hence that could affect the overall system availability.

Definition 2. System computation power is defined as:

$$Sys\_Power = Sys\_Avail(\%) \times Sys\_Power_{nom} \tag{4.2}$$

where Sys_Power_{nom} is the total nominal computation power that the system can provide.

In a first approach, we consider that in an MPSoC system consisting of NUM processors, the system power is the sum of all the processor computational powers:

$$\begin{aligned} Sys\_Power_{mpsoc} &= \sum_{i=1}^{NUM} \mu P\_Power_i \\ &= \sum_{i=1}^{NUM} \{\mu P\_Avail_i \times \mu P\_Power_{nom_i}\} \end{aligned} \tag{4.3}$$

where μP_Avail_i is the availability percentage of the processor$_i$ and $\mu P\_Power_{nom_i}$ is the nominal processor computational power of the processor$_i$, represented in MIPS/-MOPS (Mega Instructions/Operations Per Second).

Definition 3. The MPSoC system reliability is the product of all the processor reliabilities:

$$Sys\_Reliability = \prod_{i=1}^{NUM} Proc\_Reliability_i \tag{4.4}$$

The processor reliability is calculated based on the fault tolerance precision of the processor:

$Proc\_Reliability_i = 1 - Proc\_FT\_pre_i$, where $Proc\_FT\_pre_i$ is the fault tolerance precision inside the processor$_i$ which represents the probability that the system can miss an error occurrence in this processor. This precision depends on the fault probability in the related processor and the detection interval.

Definition 4. The fault probability of a module is defined by the probable number of fault occurrences in the module. So the Fault Probability within the processor$_i$ is defined as:

$$FP_i \text{ (Fault/s)} = FR \text{ (Fault/(s.Mb))} \times Sens\_Bit_i \text{ (Mb)} \tag{4.5}$$

where FR is the nominal *Failure Rate* in time per Megabit of configuration bits. This *Failure Rate* is device and environment dependent,

The fault probability of a processor is defined by probability of fault appearance per an area unit during a time unit, multiplied by the sensitive bits of the processor.

Definition 5. $Sens\_Bit_i$ is the size representation of processor$_i$. It's all the bits of which any change will lead to a processor$_i$ malfunction.

The actual dynamic reconfiguration design comes with the declaration of Partially Reconfigurable Region (PRR) [16]. In a dynamic zone (PRR), we put the whole processor inside, which does not occupy all the reserved resources. As a consequence, the sensitive bits of a dynamic processor is defined by the occupation ratio (% *Occupation*) of the processor within the declared region, as follows:

$$Sens\_bit = BS\_size \times \%\ Occupation \tag{4.6}$$

where the bitstream size BS_size of the dynamic processor is defined by the size occupied by the PRR at design time.

Definition 6. T_{detect} is the detection time interval.

$f_{detect} = \frac{1}{T_{detect}}$ is the frequency that the overall system hangs to check the statuses of all components. After this process, the system level decision will be given according to the error to maintain the functionalities.

t_{detect} is the duration of one system level detection process.

$t_{correct}$ is the duration for correcting one processor.

t_{save} and $t_{restore}$ is the duration for saving and restoring the context of one module, these two recovery parameters are application dependent.

Definition 7. N_i is the number of detection processes that have taken place upon a fault occurrence in the processor$_i$. N_i depends significantly on the Fault Probability in the processor$_i$ and the detection interval T_{detect}.

These above definitions will be applied to our FT-DyMPSoC system using COTS FPGAs. Nevertheless, these definitions have general applicability which is not only restricted to FPGA.

4.3.2 Analytical model for FT-DyMPSoC

We evaluate the affectation of fault-tolerance scheme to the system performance and reliability in our MPSoC system consisting of NUM processors. The system timing diagram is shown in Fig. 3.3 (p. 45).

T_{int}: the interrupt tick, that is the detection interval T_{detect},

T_{sync}: the synchronization duration, is the detection duration t_{detect},

T_{reconf}: the reconfiguration times, corresponds to the correction duration $t_{correct}$,

T_{save}, $T_{restore}$: context saving and restoring durations.

The processor$_i$ availability is calculated as follows:

$$\mu P\_Avail_i = \begin{cases} \frac{T_{int}-T_{sync}-T_{save}}{T_{int}} & \text{(I)} \\ \frac{T_{int}-T_{sync}-T_{reconf}-T_{restore}}{T_{int}} & \text{(II)} \end{cases} \tag{4.7}$$

In Eq. 4.7, the case I represents the processor availability when there is no error. This process, which has detection and context saving phases, takes place (N_i - 1) times. N_i is the probable number of synchronization processes upon one fault occurrence (one reconfiguration process), calculated by the probable time interval that a fault appears ($\frac{1}{FP_i}$) divided by the interrupt interval (T_{int}):

$$N_i = \frac{\frac{1}{FP_i}}{T_{int}} = \frac{1}{T_{int} \times FP_i} \tag{4.8}$$

During (N_i-1) processes (normal operation without error), there is a fault occurrence which corresponds to a reconfiguration process (Case II).

then:

$$\begin{aligned} \mu P\_Avail_i &= \frac{N_i - 1}{N_i} \times \frac{T_{int} - T_{sync} - T_{save}}{T_{int}} + \frac{1}{N_i} \times \frac{T_{int} - T_{sync} - T_{reconf\_i} - T_{restore}}{T_{int}} \\ &= \frac{T_{int} - T_{sync} - T_{save}}{T_{int}} - \frac{1}{N_i} \times \frac{T_{reconf\_i} + T_{restore} - T_{save}}{T_{int}} \\ &= \frac{T_{int}}{T_{int}} - \frac{T_{sync}}{T_{int}} - \frac{T_{save}}{T_{int}} - \frac{1}{N_i} \times \frac{T_{reconf\_i} + T_{restore} - T_{save}}{T_{int}} \\ &= 1 - \frac{T_{sync}}{T_{int}} - \frac{T_{save}}{T_{int}} - \frac{1}{N_i} \times \frac{T_{restore} - T_{save}}{T_{int}} - \frac{1}{N_i} \times \frac{T_{reconf\_i}}{T_{int}} \end{aligned} \tag{4.9}$$

where $\frac{T_{sync}}{T_{int}}$ is the detection factor (X_{detect}),

$\frac{1}{N_i} \times \frac{T_{reconf\_i}}{T_{int}}$ is the correction factor ($X_{correct}$),

and $\frac{T_{save}}{T_{int}} + \frac{1}{N_i} \times \frac{T_{restore} - T_{save}}{T_{int}}$ is the recovery factor illustrating how the save and restore processes affect the system performance ($X_{recover}$).

Besides, it exists a correlation factor between processors ($X_{correlation}$). This factor is due to the processor$_i$ that will correct the fault occurring in the processor$_{i+1}$.

So $X_{correlation} = \frac{1}{N_{i+1}} \times \frac{T_{reconf_{i+1}}}{T_{int}}$

Finally, the processor availability is

$$\begin{aligned} \mu P\_Avail_i &= 1 - \frac{T_{sync}}{T_{int}} - \left[\frac{T_{save}}{T_{int}} + \frac{1}{N_i} \times \frac{T_{restore} - T_{save}}{T_{int}}\right] \\ &- \frac{1}{N_i} \times \frac{T_{reconf_i}}{T_{int}} - \frac{1}{N_{i+1}} \times \frac{T_{reconf_{i+1}}}{T_{int}} \end{aligned} \tag{4.10}$$

According to Eq. 4.3, we have the overall system performance:

$$\begin{aligned} Sys\_Power = \sum_{i=1}^{NUM} \Bigg\{ \Bigg[1 - \frac{T_{sync}}{T_{int}} - \frac{T_{save}}{T_{int}} - \frac{1}{N_i} \times \frac{T_{restore} - T_{save}}{T_{int}} \\ - \frac{1}{N_i} \times \frac{T_{reconf_i}}{T_{int}} - \frac{1}{N_{i+1}} \times \frac{T_{reconf_{i+1}}}{T_{int}} \Bigg] \times \mu P\_Power_{nom_i} \Bigg\} \end{aligned} \tag{4.11}$$

In a homogeneous MPSoC system, the nominal processor computational power is the same so:

$$\begin{aligned} Sys\_Power = \sum_{i=1}^{NUM} \Bigg\{ 1 - \frac{T_{sync}}{T_{int}} - \left[\frac{T_{save}}{T_{int}} + \frac{1}{N_i} \times \frac{T_{restore} - T_{save}}{T_{int}}\right] \\ - \frac{2}{N_i} \times \frac{T_{reconf_i}}{T_{int}} \Bigg\} \times \mu P\_Power_{nom} \end{aligned} \tag{4.12}$$

The processor fault tolerance precision is the probability that the system can miss a fault inside a processor:

$$Proc\_FT\_pre_i = FP_i(Fault/s) \times T_{int}(s) \tag{4.13}$$

hence the reliability inside a processor is:

$$\begin{aligned} Proc\_Reliability_i &= 1 - Proc\_FT\_pre_i \\ &= 1 - FP_i \times T_{int} \end{aligned}$$

Following Eq. 4.8, we get:

$$Proc\_Reliability_i = 1 - \frac{1}{N_i} \tag{4.14}$$

The processor reliability is the probability that the processor executes its tasks successfully. Consequently, the system reliability is the product of all the processor reliabilities:

$$\begin{aligned} Sys\_Reliability &= \prod_{i=1}^{NUM} \{1 - FP_i \times T_{int}\} \\ &= \prod_{i=1}^{NUM} \{1 - \frac{1}{N_i}\} \end{aligned} \tag{4.15}$$

The system reliability depends on the fault probability FP_i in each reconfigurable component which is constant in a specific environment and at the same time depends on T_{int}-the synchronization rate. If we decrease this rate (prolong T_{int}), the system reliability decreases (Equation 4.15), on the contrary, the system computational power increases (Equation 4.12).

4.3.3 Model application for *scrubbing*

Scrubbing [7] is a classical solution that is widely used in industry for reconfigurable architectures. This technique uses the reconfiguration scrub to periodically reload a configuration frame at a chosen time interval. So in order to mask the possible errors, all the configuration frames within the FPGA have to be reloaded that means reconfiguring the whole FPGA but using partial reconfiguration of configuration frame one by one. The reconfiguration frequency must be much more superior to the error rate. To recall that reconfigurations scrub cannot deal with neither persistent error nor permanent upset. To be sure that each module is refreshed correctly, all the frames of one module have to be reloaded to return to the startup state and two different modules must not share

one configuration frame. So we choose the "modified" *scrubbing* scheme in which all the modules (processors) are in dynamic zone that do not share any configuration frame. Also, we can reconfigure the whole processor without interrupting the others. All the dynamic processors are reconfigured at the frequency superior to the error rate of these processors. After the reconfiguration, the processor returns to the startup state without the need of saving and restoring the processor context. So as to be easily deployable and able to compare with our method, we choose the *modified scrubbing* scheme. Instead of partially reconfiguring the configuration frames one after another consecutively, all the dynamic modules are reconfigured at the frequency superior to the error rate of these modules. The duration for module-based reconfiguration of the processor is approximately equal to the duration of applying configuration scrub to all the frames of this processor.

Applying the proposed analytical model to the *scrubbing* system, the reconfiguration rate of processor$_i$ is defined by factor n_i - one configuration process is launched among n_i synchronization cycles. The availability of the processor$_i$ is:

$$\begin{aligned} Proc\_Avail_i^* &= \frac{n_i - 1}{n_i} + \frac{1}{n_i} \times \frac{T_{int} - T_{reconf_i} - T_{reconf_{i+1}}}{T_{int}} \\ &= 1 - \frac{1}{n_i} \times \left\{ \frac{T_{reconf_i}}{T_{int}} + \frac{T_{reconf_{i+1}}}{T_{int}} \right\} \\ &= 1 - \frac{1}{n_i} \times \frac{T_{reconf_i}}{T_{int}} - \frac{1}{n_i} \times \frac{T_{reconf_{i+1}}}{T_{int}} \end{aligned} \tag{4.16}$$

with $n_i = \frac{1}{R_{scrub}} \times \frac{1}{T_{int} \times FP_i} = \frac{1}{R_{scrub}} \times N_i$, where R_{scrub} is the *scrubbing* ratio:

$R_{scrub} = \frac{F_{Scrub}}{Failure\ Rate}$, the *scrubbing* frequency is R_{scrub} times superior to the failure rate).

in the Eq. 4.16, $\frac{1}{n_i} \times \frac{T_{reconf_i}}{T_{int}}$ is the correction factor,

and there is no detection and recovery factors because we reconfigure regularly without using the synchronization and context recovery processes ($T_{sync} = T_{save} = T_{restore} = 0$).

Consequently, the computational power of *scrubbing* system is:

$$Sys\_Power_{Scrub} = \sum_{i=1}^{NUM} \left\{ 1 - \frac{2}{n_i} \times \frac{T_{reconf_i}}{T_{int}} \right\} \times \mu P\_Power_{nom} \tag{4.17}$$

The fault tolerant precision of *scrubbing* within the processor$_i$:

$$FT\_pre_i^* = \frac{n_i}{N_i} = \frac{1}{R_{scrub_i}} \tag{4.18}$$

Then the system reliability using *scrubbing*:

$$Sys\_Reliability_{scrub} = \prod_{i=1}^{NUM} \{1 - \frac{1}{R_{scrub_i}}\} \tag{4.19}$$

The *scrubbing* system reliability is constant in a fault stationary environment and depends only on the *scrubbing* rate that is R_{scrub} times faster than the probable error rate. In order to maintain the *scrubbing* reliability, the error rate of the environment must be thoroughly studied before.

4.4 Experimentation details and comparisons

4.4.1 Implementation

We only take into account errors in dynamic processors, not the filters since these tasks are not very critical and they can be harden by classical hardware redundancy technique. The DDR2 SDRAM is used for saving processor contexts, for buffering video frames and for accelerating the reconfiguration processes.

The processor bitstream sizes with the occupation percentages and the required reconfiguration times are shown in Table 4.1. So the interrupt interval is chosen at 100ms that is superior to the reconfiguration times of processors in order that the interrupt routine does not disrupt the reconfiguration process.

The three processors nearly have the same resource requirements but different sizes of declared regions for the dynamic reconfiguration purpose. Consequently they have different occupation rates (Table 4.1). If we calculate the occupation rate based on SLICE (Virtex Configurable Logic Block - CLB) utilization, the sensitive percentage must approximate this rate (Required SLICEs for processor upon available SLICEs of

the reserved region). Therefore, the calculated sensitive bits of the three processors (Eq. 4.6) are almost equal (about 134 KBytes), then the synchronization processes numbers N_i are identical for all three processors.

TABLE 4.1: Experimental results

	Bitstream Size	Occup. Ratio	T_{Reconf}
μP1	194(KBytes)	69%	99 ms
μP2	177(KBytes)	76%	91 ms
μP3	158(KBytes)	85%	80 ms

Based on [68], the Virtex 5 has a nominal Failure Rate of 151 FIT/Mb at 95% confidence range. One FIT is one failure in 10^9 hours, then it comes:

$$\begin{aligned} N_i &= \frac{1}{T_{int} \times FP_i} = \frac{1}{T_{int} \times FR \times Sens\_Bit_i} \\ &= \frac{1}{100ms \times 151FIT/Mb \times 134KBytes} \\ &= \frac{1}{\frac{0.1s \times 151FIT/Mb}{10^9 \times 3600s} \times \frac{134KBytes}{1024} \times 8bits} \\ &= 2 \times 10^{14} \end{aligned} \tag{4.20}$$

The synchronization phase of 3 processors takes 200 μs (T_{sync} = 200 μs) (Fig. 3.12 p. 58). After each synchronization, the processor contexts are saved in a reserved place in the SDRAM. The processor contexts are contained in 34 registers including 32 General-Purpose Registers and 2 Special-Purpose Registers (Program Counter Register and Machine Status Register). The roles of these 34 registers are explained thoroughly in [11]. Each register is 32-bit so the state of one processor is contained in about 1 Kbit which needs 4.5 μs to be saved to and 7,65 μs to be restored from the SDRAM (T_{save} = 4.5 μs, $T_{restore}$ = 7.65 μs).

The nominal computational power of one MicroBlaze at 100MHz is about 120 Dhrystone Mega Instructions Per Second (DMIPS) [11] (μP_Power_{nom}=120 DMIPS), so maximum performance of the system containing 3 processors Sys_Power_{nom} = 360 DMIPS.

4.4.2 Comparison

Applying all above data to Equation 4.12 and Equation 4.15, we get the system availability of 359 DMIPS (99.78% of 360 DMIPS) and the system reliability of $(1 - 2 \times 10^{-11}) \times$

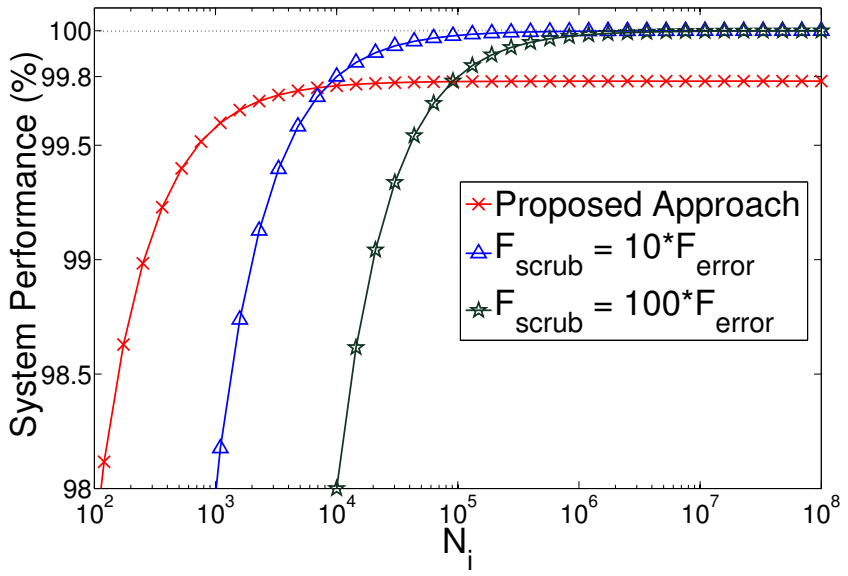


FIGURE 4.1: System performance with three processors and T_{int} = 100ms

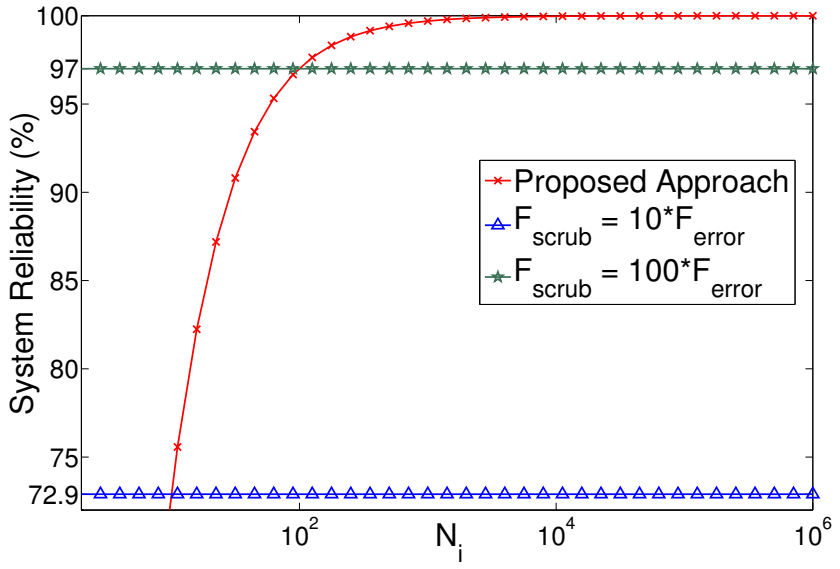


FIGURE 4.2: System Reliability with three Processors and T_{int} = 100ms

100% at 95% confidence range.

Compared to the *scrubbing* system (Equation 4.17 and 4.19), the availabilities are 99.99999999% and 99.9999999% of 360 DMIPS, the reliabilities are 72.9% and 97% (Figure 4.2), with respectively the *scrubbing* rate 10 times (R_{Scrub} = 10) and 100 times (R_{Scrub} = 100) superior to the error rate.

With our proposed approach, the system performance very slightly decreases 0.2% (359 comparing to 360 DMIPS) but the system reliability is significantly improved, our approach provides the reliability of 100% with precision of 10^{-11} comparing to two *scrubbing*

system: 72.9% with $R_{scrub} = 10$ and 97% with $R_{scrub} = 100$.

And since the Mean Time Between Failure (MTBF) of a Virtex-5 device is about dozens of years [68]. So the time between two *scrubbing* operations could be years, months or days. *Scrubbing* cannot be applied very regularly because of the non-capability of detecting error and the long functional interruption due to long reconfiguration time. Thus the reliability of systems using *scrubbing* is so low (72.9% or 97%). On the contrary, applying the connection matrices algorithm, we got a better reliability. The connection matrices exchange plays the role of the self-checking process in the processor network. This process is much faster than the reconfiguration of *scrubbing*. And especially, thanks to its short period, this exchange can be done regularly leading to a great improvement of reliability.

Figure 4.1 and Figure 4.2 show the system performance and reliability comparison between the proposed approach and 2 *scrubbing* systems with *scrubbing* rates respectively 10 times and 100 times faster than the failure rate, in the variation of N_i that means also the variation of fault probability (Equation 4.8). With $N_i > 10^5$ (or a Mean Time Between Failure MTBF $<$ 41 days), the system performance begins to be less than the *scrubbing* system with $R_{scrub} = 100$, but the availability is always more than 99.75%, while the system reliability really dominates the two *scrubbing* systems (Figure 4.2).

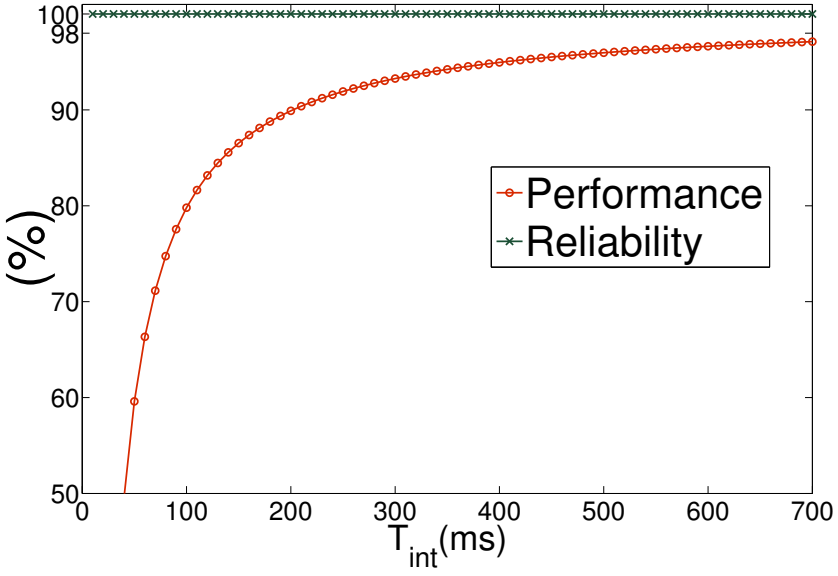


FIGURE 4.3: Performance vs. reliability with Failure Rate=151 FIT

By knowing the N_i parameter using a counter triggered by the interrupt and stops each time a reconfiguration is launched (a fault is detected), the failure rate of the

device in the environment can be easily deduced so that the system chooses the optimal interrupt interval T_{int} for the specific system. Figure 4.3 shows the variation of system performance and reliability in function of T_{int} variation. The value of Failure Rate is constant at 151 FIT [68] ($FP_i = 44 \times 10^{-11}$ (Fault/Second)). Applying to Equation 4.12 and Equation 4.15, the fault tolerance scheme almost achieves 100% reliability while the system performance increases with the rise of interrupt time interval.

4.5 Conclusion

In a fault-tolerant MPSoC system, a trade-off always exists between performance and reliability: performance gain is achieved in addition with a loss in reliability. This chapter presents an analytic approach to evaluate the performance and reliability of fault-tolerant MPSoC system. A trade-off between performance and reliability exists in a fault-tolerant MPSoC system. The proposed analytical model affirms this statement. This model is applied to our FT-DyMPSoC system and state-of-the-art approaches to enhance our approach advantages: better performance/reliability trade-off, higher hardware resource utilization. This model allows system designer to choose system specification according to pre-design requirements. Furthermore, the model is not only limited to commercial FPGAs, a wide applicability can be extended by feeding proper parameters to the model.

The validation of the proposed analytical model consists in the deliberate introduction of faults into a system an effective fault-injection campaign should be realized. An effective fault injection campaign allows for the dependability assessment of the system so helps to evaluate the accuracy of the analytical model. For that purpose, a fault injection engine is being taken into account.

4.6 Simulation and verification model for fault-tolerant MPSoC

The inherent complexity is the development and validation of MPSoC systems with fault-tolerance feature. This feature should be scheduled as early as possible in the design flow. Designing and validating these complex systems in real hardware implementation usually require lots of efforts and may delay the appearance of final product. Using system-level modeling language to model such platform can quickly validate the concept of fault-tolerance feature in complex systems at various levels of abstraction.

State-of-the-art research concerns modeling complex systems at high-level. The methodology in [69] presents modeling and simulation of MPSoC that includes dynamic reconfiguration. This approach does not neither implement any fault-tolerance mechanism nor consider the dynamic reconfiguration. In [70], a methodology is proposed that performs design space exploration and models partially reconfigurable hardware using TLM but without MPSoC and fault-tolerance modeling. Another methodology in [71] supports multi-MicroBlaze system modeling using StepNP simulation and exploration platform for system level architectural exploration. This approach does not implement any dynamic reconfiguration or any fault-tolerance mechanism. In [72] a system-level cycle-based framework called *GRAPES* has been proposed for the modeling and designing heterogeneous MPSoC that allows structural and modular models to cope with modeling, simulation and design challenges for MPSoC systems. It provides fast simulation speed, while maintaining cycle level accuracy. This work also supports dynamic reconfiguration but without fault-tolerance mechanism.

It is clear from the analysis of the existing models that neither implements any fault-tolerant mechanisms. Thus a modeling methodology of a fault-tolerant MPSoC has been proposed.

Fig. 4.4 shows the proposed model containing processor modules (μP), point-to-point connections between processors based on FIFO channels, a shared memory (SM) and a Fault-tolerant Engine. Each processor module which is not architectural dependent, is dynamically reconfigurable and handles the execution of both normal task and fault-tolerant task. Each processor has its respective local memory (LM) for storing its own software context. The SM is used for storing bitstreams of the reconfigurable processors

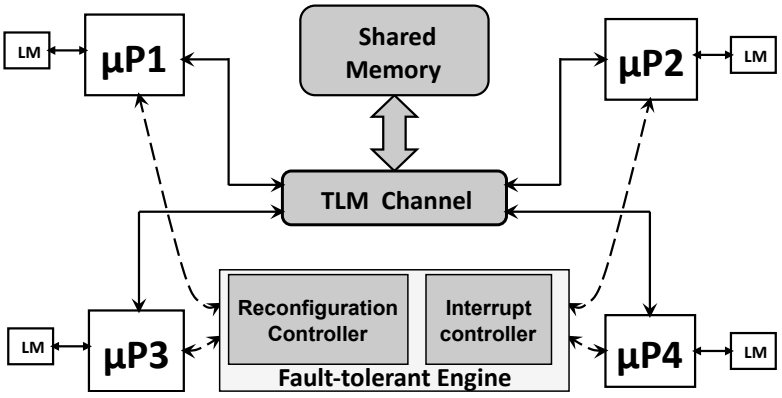


FIGURE 4.4: Proposed model

that are pre-loaded as well as for containing the software contexts of all the processors during execution. Multiple accesses to the SM are carried out via the TLM (Transaction-Level Modeling) channel.

Fault-tolerant Engine consists of two blocks: a Interrupt Controller and a Reconfiguration Controller. This engine has three intrinsic functionalities: fault detection, fault correction and context recovery after fault elimination.

The concept of fault-tolerant mechanism is implemented in the interrupt routine driven by the Interrupt Controller. All the processors are interrupted at the same time and exchange connection matrices to determine the faulty processor or the defected link to be reconfigured to eliminate the fault. The flow of this fault-tolerance mechanism is given in Fig. 4.5. Faulty processor or link is detected and reconfigured within the interrupt routine.

The execution model of a processor is illustrated in Fig. **??**. The reconfiguration controller is connected to all the processors and it does the reconfiguration of the faulty processor. It is sensitive to the data received from the processor that initiates the reconfiguration. The received reconfiguration data is analyzed to determine which processor needs to be reconfigured. The execution of the faulty processor is halted and it waits for the event generated by the reconfiguration controller once the reconfiguration has been completed. It can be seen from Fig. 4.6, if bitstream **PBS_M** value is correct, it means the processor is not faulty and it executes its task and interrupt routine as usual. If the value has changed, it means the processor is faulty and it will wait for the reconfiguration. After the reconfiguration, **PBS_M** is restored to the correct initial

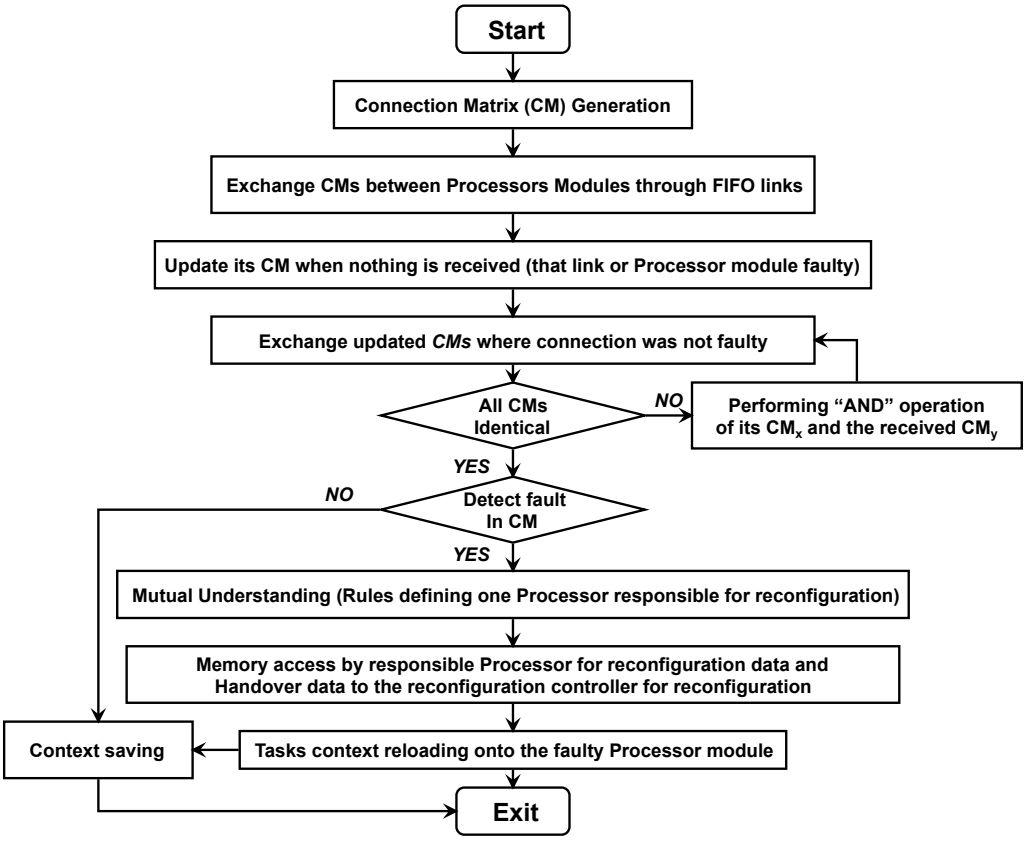


FIGURE 4.5: Fault-tolerance mechanism

value. Now, the processor will wait for the context to be reloaded from the SM that will be used during the task execution.

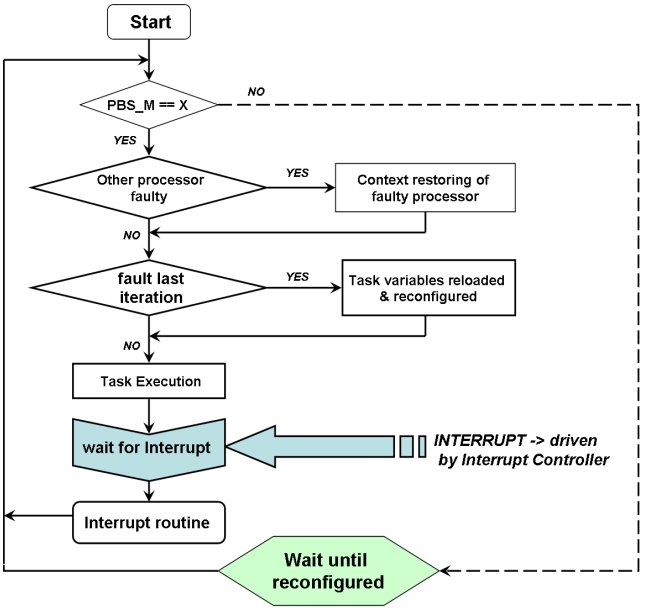


FIGURE 4.6: Execution model

- When there is no fault, processor checks if any other processor needs context reloading. If another processor is just reconfigured and needs context reloading, then the processor reloads the context. It executes its task and the result of the task is sent during the interrupt routine to all the other processors which store the result as task context in their local memories.

- In case of fault, the processor is reconfigured, the context of the task is reloaded that was previously saved in the SM when no fault occured and the task is re-executed.

4.6.1 Implementation

All the components are modeled in SystemC. The processor is implemented as a SystemC module constructed with one SC_THREAD process that handles the execution of both normal task and interrupt routine and would be able to access bitstream from memory in case of fault. The SystemC sample code for the processor module is shown in Fig. 4.7(a) where *execution_run* implements the execution model of the processor.

```
#include "systemc"
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"

SC_MODULE (processor1)
{
    sc_port<sc_fifo_in_if<unsigned int> > M1_rcvM2;
    sc_port<sc_fifo_out_if<unsigned int> > M1_txM2;
    sc_port<sc_fifo_in_if<unsigned int> > M1_rcvM3;
    sc_port<sc_fifo_out_if<unsigned int> > M1_txM3;
    sc_out<unsigned int> bit_stream_in_M1;
    sc_out<unsigned int> bit_stream_out_M1;

    tlm_utils::simple_initiator_socket<master1> socket;

    SC_HAS_PROCESS (processor1);
   processor1(sc_module_name name_): sc_module(name_)
   {
     SC_THREAD(execution_run);
   }

    void execution_run();

    private:
      void interrupt_rout1();

      int task(int);
      int update_bit1(int data, int pos, int val);
      int data;
};
```

(a) Processor

```
#include "systemc"
#include "tlm.h"
#include "tlm_utils/simple_target_socket.h"


SC_MODULE(Memory)
{
 tlm_utils::simple_target_socket<Memory> socket1,socket2;  // target socket

 SC_CTOR(Memory): socket1("socket1"),socket2("socket2")
 { socket1.register_b_transport(this, &Memory::b_transport);   // registering transport call
   socket2.register_b_transport(this, &Memory::b_transport); }

 virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )  //blocking interface method
 {
  tlm::tlm_command cmd = trans.get_command();
  unsigned int     adr = trans.get_address() ;
  unsigned char*   ptr = trans.get_data_ptr();
  unsigned int     len = trans.get_data_length();
  unsigned char*   byt = trans.get_byte_enable_ptr();
  unsigned int     wid = trans.get_streaming_width();

  if ( cmd == tlm::TLM_READ_COMMAND )  // implementing read operation
    memcpy(ptr, &mem[adr], len);

  trans.set_response_status( tlm::TLM_OK_RESPONSE );  //set response status to indicate successful
      completion
 }

 int mem[SIZE];   // memory as an array
};
```

(b) Shared Memory

FIGURE 4.7: Sample Codes

In the model, the shared memory (SM) is implemented as a SystemC module and holds processor bitstreams. The sample SystemC for this memory is shown in Fig. 4.7(b). Two target TLM sockets have been declared and they must be registered with the TLM blocking interface. Memory implements the b_transport blocking interface.

The interrupt controller is a SystemC module modeled as a counter and sensitive to clock. When the value of the counter reaches some predetermined value, an event (SystemC construct) is notified which activates all the suspended processes in all the processor modules executing their respective interrupt routines. The reconfiguration controller is a SystemC module and performs reconfiguration of the faulty processor. It uses sc_method process which is sensitive to the data received from the processor to initiate the reconfiguration.

4.6.2 Analysis

Concept verification of the fault-tolerance methodology is performed by introducing fault in the processor module and later checked, if the system detects and corrects the fault or not. The fault type was a change inside the representation bitstream of the related processor model. From the analysis point of view, the interrupt interval has been varied and the effect on performance and reliability has been observed. Three interrupt scenarios have been taken i.e. 100 ms, 150 ms and 300 ms. The system with interrupt period of 100 ms is more reliable than the other two systems because the fault in the system is detected and corrected earlier than the other two systems. The execution times of the model have been measured 1000 consecutive times and the results have been attained by taking the average of these samples.

- Interrupt routine takes 1595.6 μs when there is no fault in the system.
- Interrupt routine takes 2969.25 μs when there is fault in the system.

When fault arises, interrupt routine takes more time as time required to make their connection matrix equivalent will increase and also it includes the reconfiguration time. Increasing interrupt frequency will have a negative effect on system performance because the interrupt routine will be invoked more frequently and consequently the time for executing tasks will be less . This will be even worse in case of fault. There has to be some trade off in the selection of interrupt ticks, so that reliability and performance are not effected negatively. This depends on the critical application running on MPSoC.

4.6.3 Conclusions and Future Works

In this paper, a new model has been proposed that validates successfully the fault-tolerance mechanisms through dynamic reconfiguration using SystemC. The proposed facilitates the fast evaluation and verification of fault-tolerance feature in complex system. This model has a lot of research potential from future point of view. The simple processor model can be replaced by an Instruction Set Simulator (ISS) [73] to have more timing accuracy. New techniques of fault tolerance can also be tested with this model such as *scrubbing* [7]. Scalability of such model can be increased by replacing FIFO with an efficient Network-on-Chip (NoC). A more sophisticated controller handling the reconfiguration can also be tested.

Chapter 5

Low overhead fault-tolerant reconfigurable softcore processor

5.1 Abstract

Modern FPGAs, like Xilinx Virtex-5, besides customary reconfigurable resources, offer the designers programmable softcore processors having features of Commercial Off-The-Shelf (COTS) components. Because these FPGAs are SRAM-based devices particularly sensitive to radiation, their use in mission-critical embedded applications would not be feasible due to insufficient reliability unless some fault-tolerance techniques capable of mitigating radiation-induced temporary faults (soft errors) are used. In this chapter, we consider the possibility of implementing techniques which would allow to tolerate temporary configuration faults of the softcore processors at low hardware and time overhead. An enhanced lockstep scheme allows to detect and eliminate errors in real-time, without interrupting the functioning of the system. The efficiency of the proposed approach was validated through the fault-injection experiments.

5.2 Introduction

Modern FPGAs, besides customary reconfigurable resources, offer to the designers the possibilities of implementing programmable processors offering features of Commercial

Off-The-Shelf (COTS) components (no need to modify processor architecture or application software). Xilinx FPGA devices include two categories of processors: the hardcore embedded processor (PowerPC) and softcore processors (MicroBlaze, PicoBlaze) [10]. Hardcore embedded processors are hard-wired on the FPGA die and their number is limited on each device (1, 2, 4 or no hardcore processor). On the other hand, softcore processors use reconfigurable resources, so the number that can be actually implemented depends only on the device size and the configuration of softcore processors.

Remind that lockstep scheme is the implementation of DWC at the processor level (Section 2.4.1.1). Two identical processors μP1 and μP2 receive the same inputs, simultaneously execute the same instructions, and their results are compared step-by-step at each clock cycle (Fig. 5.1). μP2 generates the reference results to be compared against those of μP1 that provides the system output. This system is able to detect error, but it cannot point out the faulty processor. In case of error, the whole system need to be refreshed to recover correct functionalities of both processors.

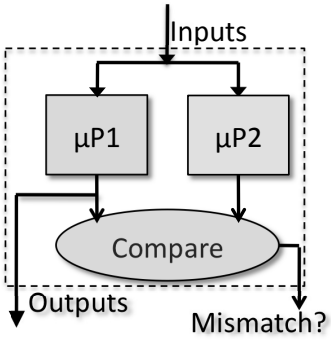


FIGURE 5.1: Basic lockstep scheme

Here, we are interested in designing and implementing a fault-tolerant softcore processor using Virtex-5 FPGA. Only relatively few works can be found on FPGA implementations of fault-tolerant softcore processors. In [31], a lockstep system using dual hardcore processors PowerPC found in certain FPGAs is constructed. However, a limited number of available PowerPC in FPGAs restricts their utilization to build e.g. a fault-tolerant Multi-Processor System-on-Chip (MPSoC). Therefore one option to implement larger number of lockstep modules in a single FPGA is to employ softcore processors that can use available reconfigurable resources of the device.

In [56], the authors claim proposing a lockstep scheme using two hardcore PowerPC processors embedded in Xilinx Virtex II Pro FPGA. However, this scheme is rather a

handshake scheme than a lockstep one, because the processors execute the same program but not simultaneously and use checkpoints to verify the system consistency. As a result, the time overhead is relatively large, because of the sequential execution of two identical tasks on two different processors, which might be prohibitively long in some real-time applications.

Because in the basic lockstep it is not possible to identify the faulty processor without extra diagnosing support, the error recovery is achieved through reconfiguration of both processor cores. However, this needs regular checkpointing for processor context saving and rollback for context restoring to the last good processor state, which can be time-consuming [56] (Fig. 5.2). Hence, it is worthwhile to consider the possibility of context saving and restoring only in case of errors, although it could require some extra hardware.

Indeed, the TMR softcore processor architecture from [74] is able to correct single error in a faulty module through rollforward error recovery. Although there is no need to regularly save the context when no errors occur, it consumes over 200% of extra hardware resources, which could preclude to build more powerful system like MPSoC. Also, the major disadvantage of this scheme is the lack of autonomy, because it requires external computer host to perform reconfiguration.

In this chapter, we propose a new architecture of a fault-tolerant reconfigurable system which can be implemented on any FPGA with integrated softcore processors at a reduced hardware and time cost. Our actual implementation on Xilinx Virtex-5 FPGA contains an enhanced lockstep scheme built using a pair of MicroBlaze cores. The faulty core can be identified by a fault-tolerant configuration engine (FT Configuration Engine) built using PicoBlaze. Once the exact error location is determined by the *Scan Motor* and the *Bitstream Parser* specially designed, the error is corrected through partial reconfiguration (frame-based or module-based) combined with rollforward recovery technique. As a result, there is no problem of fault latency, because faults are detected immediately once

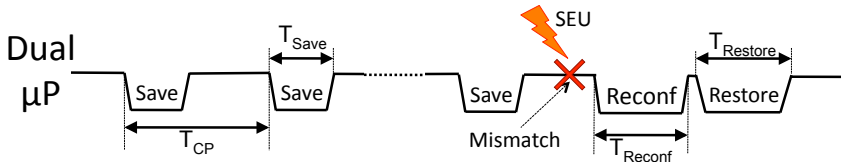


FIGURE 5.2: Checkpointing and rollback for basic lockstep recovery

they cause an error, which is unlike in [56], where it exists a delay between two separate executions in two different hardcore embedded processors.

5.3 New fault-tolerant architecture

The basic lockstep scheme is the realization of DWC at the processor level that can be implemented using PowerPC processors available in certain Xilinx FPGAs [31]. Due to very limited number of PowerPC processors available in one FPGA, it can be used to build only systems with small number of processors. The basic lockstep scheme can be implemented using softcore processor MicroBlaze to build MPSoC systems. Unfortunately, it can only detect errors without indicating the faulty module. Our architecture of the enhanced lockstep scheme (shown in Fig. 5.4) eliminates this limitation. It use a special FT Configuration Engine and two Virtex-5 hardware primitives: the reconfiguration port ICAP and the FRAME_ECC [3] to detect the faulty processor and then to continue execution with the fault-free processor.

The architecture of the fault-tolerant system to be implemented is shown in Fig. 5.3. It consists of two main blocks: the enhanced lockstep scheme and the fault-tolerant (FT) Configuration Engine (CE), which uses ICAP and FRAME_ECC [3] as well as a highly reliable external Golden Memory to store the configurations.

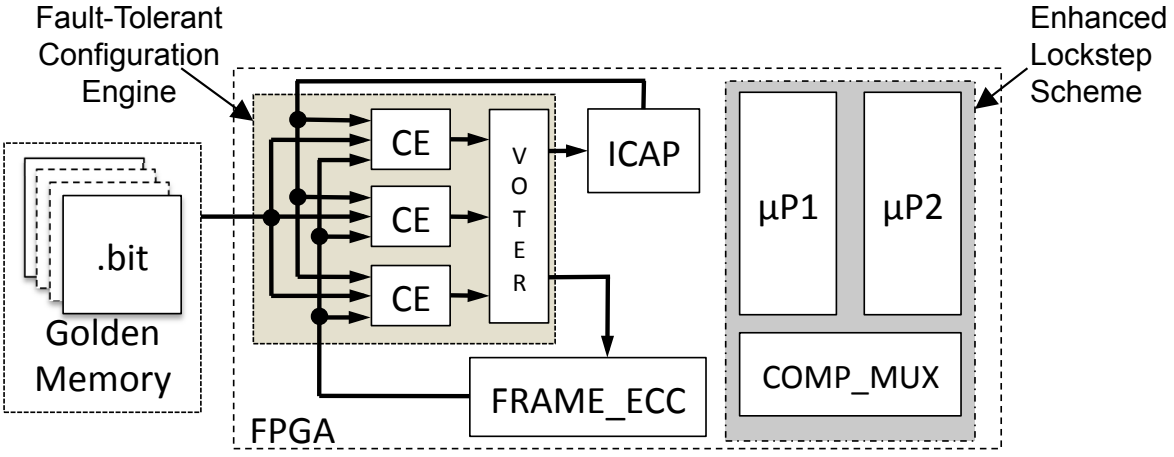


FIGURE 5.3: Block scheme of the fault-tolerance architecture

5.3.1 Enhanced lockstep scheme

Two identical softcore processors MicroBlaze μP1 and μP2 are the heart of the enhanced lockstep scheme. Each of them is dynamically reconfigurable and is a complex system consisting of a 32-bit processor core, central bus and peripherals (Fig. 2.2 p. 16), designed using the complex modular reconfiguration concept flow proposed in [75]. Their outputs are identical during fault-free functioning, any disagreement indicating error(s). A total of 200 bits of output signals of the PLB bus and the peripheral outputs are compared by the Comparator/Multiplexer (COMP_MUX) which is also dynamically reconfigurable. To note that each dynamically reconfigurable module should be fixed to a specific location in a reconfigurable zone. Without this restriction, all the resources of the two cores might be blended together and it could not be possible to distinguish the faulty module when an error is detected. Also, the location fixation allows for reconfiguration of one of the modules without interrupting the others.

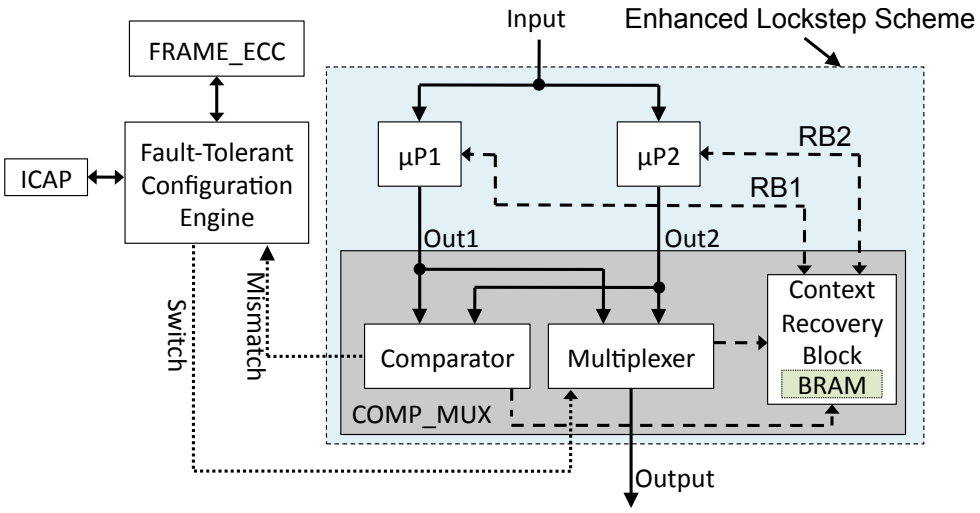


FIGURE 5.4: Enhanced lockstep scheme

Figure 5.4 describes more details our enhanced lockstep scheme. The COMP_MUX component consists of three blocks. One is the Comparator that indicates any mismatch between μP1 and μP2 for the PLB and final output signals. Another is the Multiplexer which connects one of the processors to the system output. If one of them is reported to be faulty, the Multiplexer will switch the other processor to the output. The switching is an atomic operation executed in one clock cycle. Once the error is located by the FT Configuration Engine presented in Section 5.3.2, the corresponding core is reconfigured to eliminate its configuration error. Then, the two cores need to be synchronized to put the newly reconfigured core to the same state as the correct one, so the two softcore processors

would be able to continue executing the same task in lockstep again. This is done by the *Context Recovery Block* which handles the recovery process of the enhanced lockstep scheme. The *Context Recovery Block* has a memory shared by both softcore processors to store their context using on-chip BRAM that has dual-port connection providing simultaneous access to two Recovery Buses RB1 and RB2. These buses serve to save and restore the softcore processors context to the BRAM as well as to control the *Context Recovery Block* during the context switching process. To note that the COMP_MUX itself is also a dynamic module that can be reconfigured by the FT Configuration Engine presented below in case of error on its logic.

In case of error affecting functional block, not the configuration memory of the enhanced lockstep scheme, the FT Configuration Engine is not capable of detecting the faulty module. In this circumstance, the COMP_MUX still reports an error. Our system can not identify faulty core but the enhanced lockstep scheme can operate as a classical lockstep scheme. The whole enhanced lockstep module consisting of the two processors and the COMP_MUX needs to be reconfigured at the same time to remove the error.

5.3.2 Fault-tolerant configuration engine

Once a disagreement is detected by the Comparator, the task of localizing the faulty softcore is executed by the Configuration Engine (CE) (Fig. 5.5) which consists of a *Scan Motor*, a *Bitstream Parser*, and an 8-bit softcore processor PicoBlaze (chosen because of its small size). Reliable uninterrupted operation of the Configuration Engine is crucial for the correct functioning of the whole lockstep system. Because it is also exposed to errors, to avoid a single point of failure, we made it fault-tolerant using TMR. Moreover, if the FT Configuration Engine detects an error in one of its modules, the faulty module will be disconnected. Then proper reconfiguration (frame-based or module-based) will be launched to correct the error depending on the persistence of the erroneous bit. If the bit is non-persistent, configuration scrub (frame-based reconfiguration) is launched, else the FT Configuration Engine will reconfigure the defected module by transferring the correct bitstream from the golden memory through the ICAP. This is achieved by implementing each Configuration Engine in separate PRR withing the modular reconfigurable concept. The hardware overhead of the TMR scheme in this case can be alleviated by the small size of the Configuration Engine

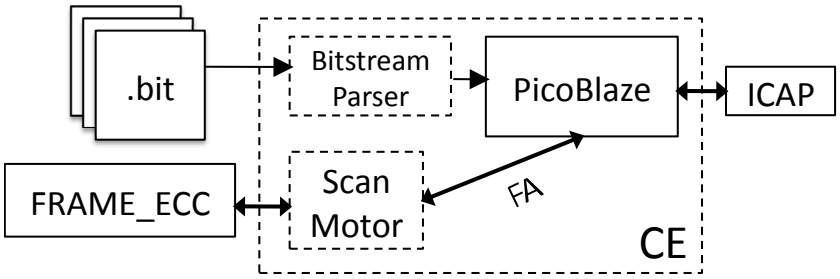


FIGURE 5.5: Configuration engine

5.3.2.1 Scan Motor

The *Scan Motor* continuously works in background to point out the physical position of the error, hence, it does not affect the user design: it cyclically reads the configuration frame by frame using readback through ICAP and checks all frames for errors. Every configuration frame contains already built-in error correction code (ECC) bits. Any change to the ECC bits caused by an SEU has no effect on the active design. During readback, the FRAME_ECC primitive automatically computes the ECC bits based on the readbacked configuration data and compares with built-in ECC bits. In case of change in the configuration memory due to soft errors, the Scan Motor localizes it and reports the erroneous Frame Addresses (FA) to the PicoBlaze, although it is not yet able to identify the faulty module. The actual address of the erroneous frame of the lockstep module is decoded by the *Bitstream Parser*. The reconfiguration of the faulty block takes place by sending the appropriate configuration data to the ICAP, realized by the Configuration Engine.

In a partially reconfigurable design, the reconfigurable modules are physically located in specific regions (PRRs). Because we do not have available tools to extract the design directly from the bitstream, we have designed the *Bitstream Parser* component to enumerate all the concerned frames within one dynamic module.

5.3.2.2 Bitstream Parser

A bitstream is a sequence of controls and data that determine the functionality of the circuit by defining certain configuration frames. However, it is a one-way relation: defining the system functionality generates the bitstream file, but it is not possible to extract the design from the bitstream. The full bitstream composition is described in [3]. For

Xilinx Virtex-5 FPGAs, some related parts of the partial bitstream resemble the full one, shown in Fig. 5.6. Beside a number of instructions, the bitstream contains the configurations of all the frames concerning the well-defined zone which contains the module function related to the bitstream. In the bitstream, all frames appear sequentially, each one consists of a command Write words to Frame Address Register (**FAR**) (30002001) followed by the FAR value.

Configuration Data (HEX)	Explanation
FFFFFFFF	Dummy Word
20000000	No Operation
...	...
30002001	Write words to FAR
xxxxxxxx	**FAR Value 1**
xxxxxxxx	**Data 1**
30002001	Write words to FAR
xxxxxxxx	**FAR Value 2**
xxxxxxxx	**Data 2**
⋮	⋮
30002001	Write words to FAR
xxxxxxxx	**FAR Value n**
xxxxxxxx	**Data n**
⋮	⋮
20000000	No Operation
	End

FIGURE 5.6: Bitstream composition (Table 6.15, p. 129 in [3])

The *Bitstream Parser* analyzes the pre-generated partial bitstreams and lists all the frame addresses related to reconfigurable blocks of our architecture. Therefore, by parsing all the partial bitstreams of these blocks, we can find all their frame address ranges. Because each reconfigurable blocks has its own list of frames, if the erroneous frame is found on the list, the error must be in the corresponding block.

With Xilinx synthesis tools, it is possible to analyze the bitstream to get address range for each frame included. However, their use requires a lot of manual steps that can only be done off-line at design time using dedicated tools on an external computer. The proposed *Bitstream Parser* can be easily used either off-line by using external computer or on-line by a processing element (which can be implemented on FPGA) using dedicated reconfigurable resources. Once the *Bitstream Parser* is realized, it can be reused for all other partial bitstreams without any modification. In the proposed system, the *Bitstream*

Parser is a software application program running once, at initialization time on the PicoBlaze of the Configuration Engine.

5.4 Fault mitigation strategy

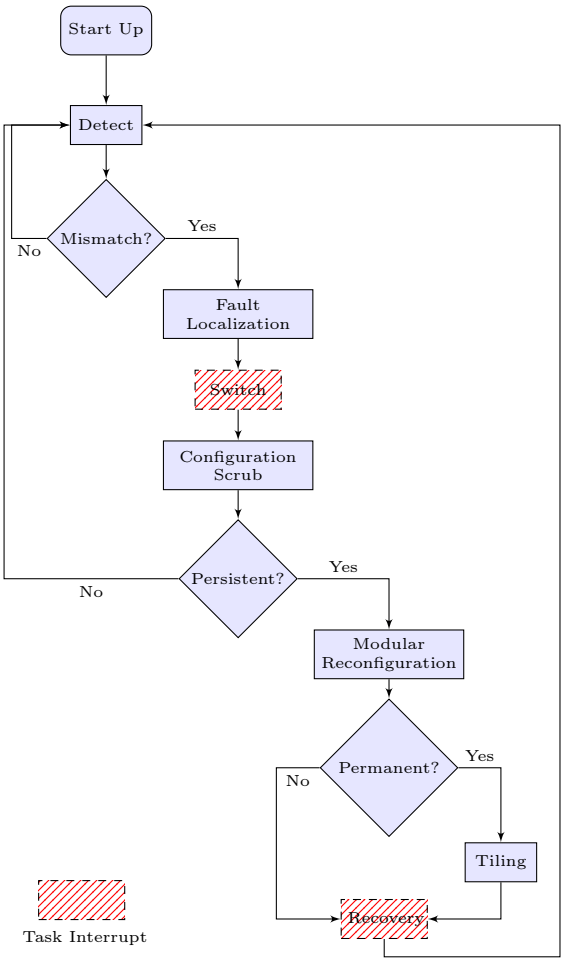


FIGURE 5.7: Fault mitigation strategy for the enhanced lockstep scheme

Figure 5.7 illustrates the fault mitigation strategy applied to deal with errors occurring in the enhanced lockstep scheme. During system operation, the COMP_MUX continuously supervises the dual-core lockstep module, while the FT Configuration Engine scans the FPGA in background. If the COMP_MUX detects a mismatch between the two cores, a fault causing an error can be either in one softcore processor or in the COMP_MUX itself. The mismatch signal of the COMP_MUX is connected to the FT Configuration Engine to launch the fault localization in the enhanced lockstep only. The FT Configuration Engine starts to scan the COMP_MUX immediately and, if a fault is found, the FT Configuration Engine will reconfigure it. If the COMP_MUX is fault-free, the error must

be due to a fault in one of the two cores, so the FT Configuration Engine only needs to scan one core to identify the source of error. The COMP_MUX switch its output to the correct processor output. The scan itself does not affect the core's task, although during this special scan, the enhanced lockstep needs to be paused to prevent any catastrophic results.

Either the error is in the COMP_MUX or in a processor, a configuration scrub is first launched by the Configuration Engine to try to correct the error. If the configuration scrub can repair this error, the error is non-persistent. Then the whole system return to its normal operation without processor context recovery, since the faulty processor will perform correct synchronized functioning with the other after configuration scrub. On the contrary, if the error cannot be repaired by the configuration scrub, the error is either persistent or permanent. At that time, the partial reconfiguration is launched to try to eliminate it. Again, if the error still persists, the fault caused by the error is permanent, consequently Tiling is launched to correct this permanent fault. After the fault correction, the dual processors enter the recovery process to synchronize their states. Recall that the configuration scrub does not interrupt the operation of the design. So the non-persistent bit elimination using configuration scrub does not affect operation delay in the related processor, hence does not require context recovery to resynchronize the two processors.

5.5 State recovery procedure for enhanced lockstep scheme

Once a core is found faulty, it is reconfigured to bring it back to its initial state as at the start-up, which is followed by loading in it the same state as the other core. Recall that the Comparator can signal an error caused by a fault in the dual core, without the capability of indicating the faulty core. However, thanks to the fault diagnostic process running in background (configuration scan and bitstream parsing), it is possible to diagnose the faulty core. It is also likely that the fault diagnosis would report an error dispite the comparator indicates no mismatch, which would only mean that a non-sensitive bit was flipped (i.e., one which has no effect on the design). After the core reconfiguration, its state recovery process is launched, as shown in Fig. 5.8.

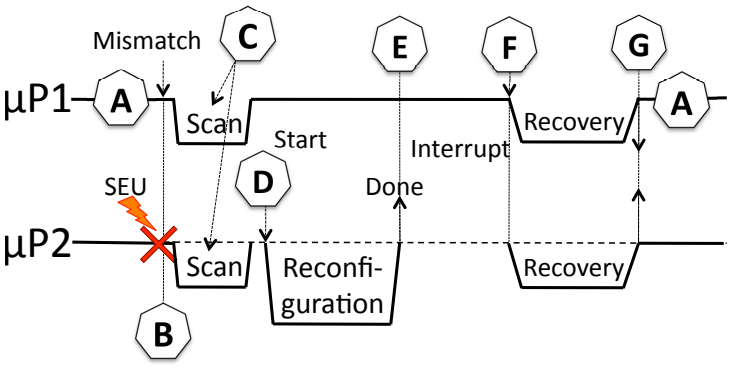


FIGURE 5.8: Rollforward state recovery process for enhanced lockstep scheme

We use the following rollforward procedure to recover and synchronize the states of the two softcore MicroBlazes μP1 and μP2. The context of each consists of the contents of 32 General Purpose Registers and two Special Registers (each has 32 bits). During normal execution of both cores (**A**), if an SEU is detected by a mismatch (**B**), the scan process (**C**) is launched immediately to localize the error (here it is in μP2) and the reconfiguration process corrects the error (**D**). During that time, the non-faulty processor continues the task execution. Once the reconfiguration is completed (**E**), μP2 signals its ready status for the state recovery procedure to the *Context Recovery Block*. The moment of recovery is decided by μP1 (e.g., it can wait until the end of a critical task that should not be interrupted). To start the recovery process, μP1 signals the *Context Recovery Block* inside the COMP_MUX to start recovery via RB1 bus. The *Context Recovery Block* sends immediately an interrupt signal via buses RB1 and RB2 to announce the two cores to begin recovery (**F**). After the recovery process, the two cores resynchronize themselves (**G**) and continue executing correctly the task (return to **A**).

The recovery process is detailed in Fig. 5.9. The *Context Recovery Block* launches an interrupt (**INT**) to force both cores to enter the recovery routine (**Start**). At the beginning, μP1 saves its context in the shared memory BRAM of the *Context Recovery Block*, while μP2 is waiting for the context saving process to finish (**Wait for SAVE_DONE**). Then, μP2 starts restoring the context (**Restore Context**), while μP1 waits for the synchronization signal from the *Context Recovery Block*. Once μP2 finishes restoring (**RESTORE_DONE**), it signals its ready status for synchronization to the *Context Recovery Block* (**Wait for SYNC**) in order that the *Context Recovery Block* instantly sends a synchronization signal (**SYNC**), and both cores are liberated from recovery to continue tasks execution in a synchronized manner.

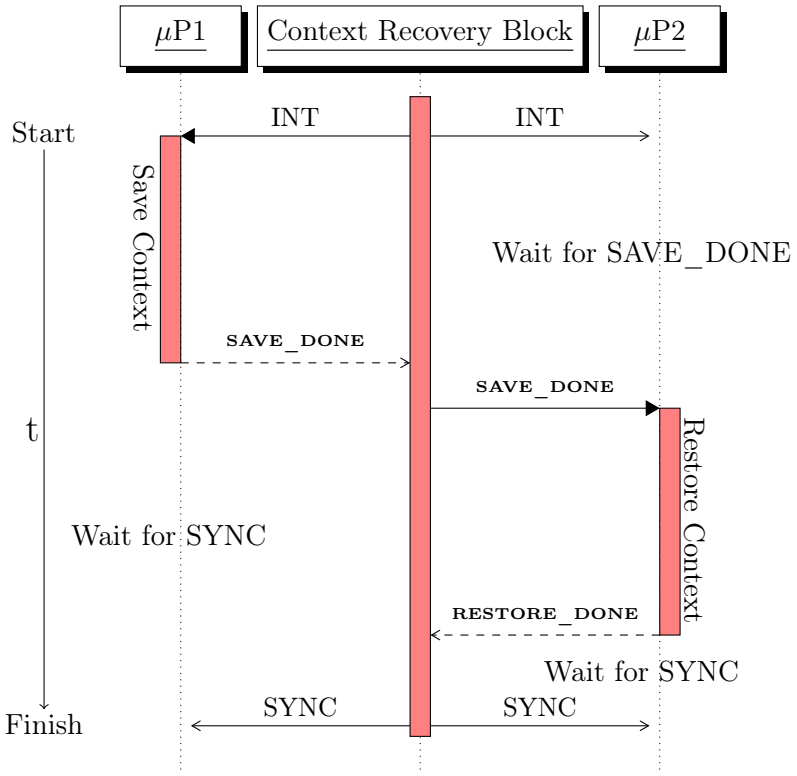


FIGURE 5.9: Recovery process detail for the enhanced lockstep scheme

5.6 Implementation details and comparison

We have implemented the system of Fig. 5.3 using Xilinx Virtex-5 XC5VSX50T device and Xilinx Design Suite v9.2. (Its layout view is shown in Fig. 5.10.) The dual lockstep cores operate at 125 MHz (one cycle takes 8 ns) whereas each of the three PicoBlazes inside the FT Configuration Engine runs at 50 MHz (one cycle takes 20 ns).

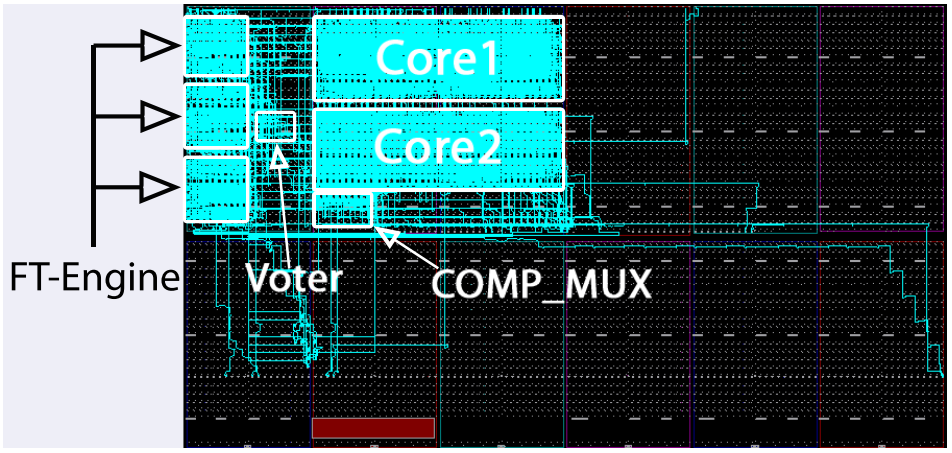


FIGURE 5.10: 90°-rotated view of implemented system

To deal with permanent faults, the *tiling* technique is implemented in our system in the two processors (μP1 and μP2) and in the COMP_MUX. Figure 5.11 shows different implementations using PROHIBIT macro (p. 46) which allow to avoid using a zone inside the PRR of μP1. Notice that in each implementation, the corresponding prohibited zone has a different location. And the classical reconfiguration scrub techniques cannot deal with permanent faults.

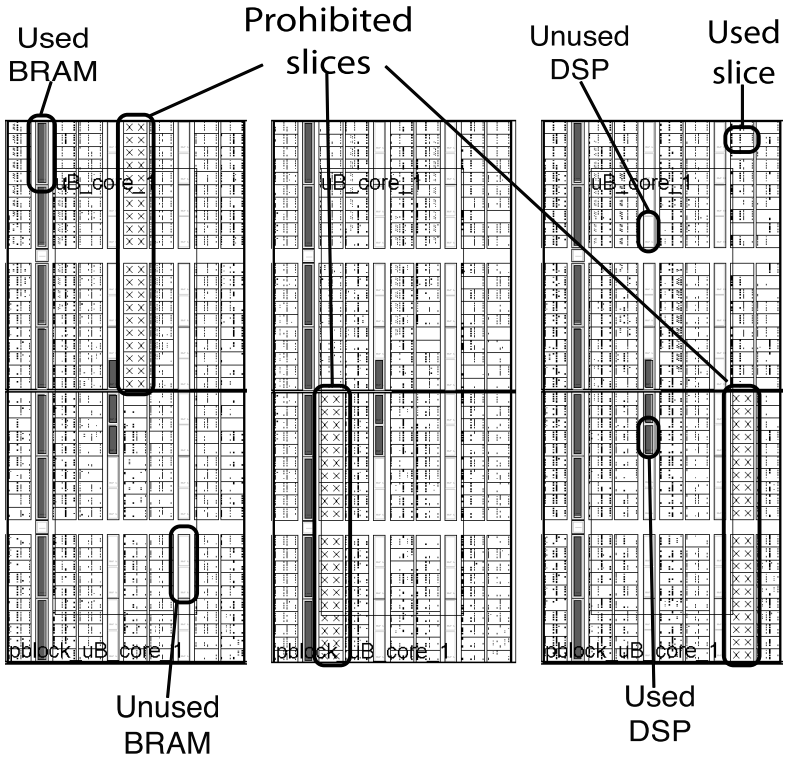


FIGURE 5.11: Three tiling implementations of μP1

Here exists a compromise between the *tiling* granularity against the external memory stocking capacity for *tiling* configuration bitstreams. The finer granularity (the finer prohibited zone) of *tiling*, the more configurations is required to cover the whole PRR, hence the more external memory space is needed to store these configurations. Supposing that a prohibited zone corresponds to a frame, the number of needed configurations is the number of frames occupied by the related processor. To reduce the stocking cost, we can generate only differential bitstream between two neighbor frames. In that case, the bitstream is considerably reduced. There are two major issues needed to be solved using this solution:

1. Between to configurations, except for the two prohibited zone (prohibited frame), all other frames must be the same. So that only 2 related frames need to be

modified when switching from one configuration to another.

2. A big challenge comes with the existence of persistent bits inside the permanently defected frame. In case that some configuration bits of the permanently faulty zone are persistent, the whole processor need to be reseted which requires either the reset signal controlled by the FT Configuration Engine, or the reconfiguration of the whole processor, or the global reset of the entire platform.

TABLE 5.1: FPGA resource occupations of enhanced lockstep modules

Module	Slices	BRAM [KB]
One Configuration Engine	120	
Voter	20	
One MicroBlaze Processor	560	32
COMP_MUX	60	4
XC5VSX50T	32640	132

The system that we have implemented has the following characteristics (Table 5.1). One configuration engine occupies 120 slices, the voter inside the FT Configuration Engine occupies 20 slices. Recall that the FT Configuration Engine that supervises the whole FPGA is hardened by TMR with 2-out-of-3 majority voter. Each Configuration Engine, the dual core lockstep block, and the COMP_MUX (except for the majority voter that drives the output signals of the FT Configuration Engine) are dynamically reconfigurable to be able to deal with soft errors in the configuration memory. Therefore the whole system can cope with soft errors excluding only the voter, which however occupies only 20 Slices, so that its error probability is negligible compared to the whole system. If necessary, this small voter can be implemented using external hardened components.

The hardware overhead of the FT Configuration Engine is $3 \times 120 + 20 + 60 = 440$ Slices (three Configuration Engines, the voter, and the COMP_MUX). This constant overhead appears only once in the FPGA independently on the user design. Besides, there is an overhead of one extra MicroBlaze Processor (560 Slices) to implement the lockstep scheme.

The duration of task interrupt (in case of a single error) depends on the source of error:

- An error in the COMP_MUX:

$$t_{i_{1a}} = t_{sCM} \tag{5.1}$$

if a non-persistent error occurs in the COMP_MUX, and

$$t_{i_{1b}} = t_{sCM} + t_{rCM} \tag{5.2}$$

if a persistent error occurs in the COMP_MUX,

where: t_{sCM} and t_{rCM} are respectively the COMP_MUX scan time and the reconfiguration time.

- An error in a processor core:

$$t_{i_{2a}} = t_{sCM} + t_{sP} + t_{sw} \tag{5.3}$$

if a non-persistent error occurs in the dual-core lockstep module, and

$$t_{i_{2b}} = t_{sCM} + t_{sP} + t_{sw} + t_{rec} \tag{5.4}$$

if a persistent error occurs in the dual-core lockstep module,

where t_{sP}—one processor core scan time,

t_{sw}—the time to switch the COMP_MUX, and

t_{rec}—the time to recover the processor context in case of persistent bit.

The FT Configuration Engine requires 41 clock cycles to complete a frame scan. The maximum time for the fault localization is the time required for scanning the COMP_MUX and one processor core. Because the COMP_MUX and one softcore occupies 3 (60 slices and 4 KBytes BRAM) and 30 (560 slices and 32 KBytes BRAM) configuration frames, respectively, $t_{sCM} = 3 \times 41 \times 20$ ns $= 2.5$ μs and $t_{sP} = 30 \times 41 \times 20$ ns $= 25$ μs. If the FT Configuration Engine detects an error in the COMP_MUX, the reconfiguration of COMP_MUX takes $t_{rCM} = 185$ μs.

The time for COMP_MUX to switch the output of dual-core module is negligible (only one clock cycle—8 ns).

The COMP_MUX contains a 4 KBytes BRAM that is shared by the two cores for the state recovery process even if 1 KBytes is sufficient. These 4 KBytes BRAM are the smallest amount that can be chosen corresponding to an elementary BRAM block in the device matrix.

```
void context_save(void){
        asm volatile(
                "swi R1, R0, 0x83C18000\n\t"    // Save R1
                "swi R2, R0, 0x83C18004\n\t"    // Save R2

                                .
                                .
                                .
                                .

                "swi R31, R0, 0x83C18074\n\t"   // Save R31

                "mfs R3, MSR\n\t"               // Save MSR
                "swi R3, R0, 0x83C1807C\n\t"    // R3 as scratchpad register

                "mfs R3, PC\n\t"                // Save Program Counter
                "addik R3, R3, 0x10\n\t"        // R3 as scratchpad register
                "swi R3, R0, 0x83C18078"
        );
};

void context_restore(void){
        asm volatile(
                "lwi R31, R0, 0x83C18074\n\t"   // Restore R31

                                .
                                .
                                .
                                .

                "lwi R2, R0, 0x83C18004\n\t"    // Restore R2
                "lwi R1, R0, 0x83C18000\n\t"    // Restore R1

                "lwi R3, R0, 0x83C1807C\n\t"    // Restore MSR
                "andni   R3,  R3, 0x02\n\t"     // R3 as scratchpad register
                "mts MSR,R3\n\t"

                "lwi R3, R0, 0x83C18078\n\t"    // Restore Program Counter
                "brad R3"                       // R3 as scratchpad register
        );
};
```

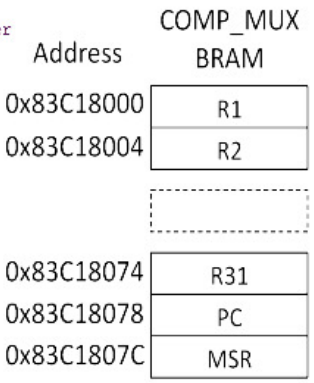


FIGURE 5.12: Context recovery C code

Figure 5.12 shows the C code of the processor context saving and restoring processes. The registers R1-R31, the Machine Status Register (MSR) and the Program Counter (PC) register are saved from the starting address 0x83C18000 of the BRAM inside the COMP_MUX. The context saving and restoring processes need respectively 272 cycles (272×8 ns $= 2.2$ μs) and 325 cycles (2.6 μs). The time of the whole recovery process consisting of the context saving, restoring and control duration of Fig. 5.9 is $t_{rec} = 5.2$ μs.

So the maximum system time overhead results from the duration of task interrupt when a sensitive error occurs, which is:

$t_{i_{1a}} = 2.5$ μs; $t_{i_{1b}} = 2.5 + 185 = 187.5$ μs—if respectively a non-persistent or persistent error occurs in the COMP_MUX, and

$t_{i_{2a}} = 2.5 + 25 = 27.5$ μs; $t_{i_{2b}} = 2.5 + 25 + 5.2 = 32.7$ μs—if respectively a non-persistent or persistent error occurs in the dual-core lockstep module.

The parsing process of the *Bitstream Parser* occurs only once during the system startup and it does not influence the system tasks execution, so its timing is irrelevant. This process, which analyzes the bitstream and enumerates all the concerned frame addresses of the correspondent PRRs, is implemented as a software piece of the PicoBlaze inside the Configuration Engine.

TABLE 5.2: Comparison between approaches (t_{rec} = 5.2 μs)

	Simple Processor	Basic Lockstep	TMR Softcore † [74]	Our Approach
Hardware Resources [%]	100	~200 + 80	~300	~200 + 80
Time Overhead [t_{rec}]	N.A.	20000	1	36
Services Continuity	No	Low	High	High

† An external computer must be used for reconfiguration

Because for the FPGA lockstep system, only a system using hardcore embedded processor PowerPC was suggested [31] and the only fault-tolerant FPGA implementation using the MicroBlaze softcore processor was proposed using TMR in [74], we have also implemented the basic lockstep scheme using softcore processors. The two softcores and the voter are bound to one PRR. We found that the hardware overhead of the basic lockstep is the same as of the proposed enhanced lockstep solution: either system consumes twice of hardware resources than the simple processor. Obviously, there must be a spare processor which controls the reconfiguration process of the dual-core module in case of mismatch. This spare processor itself needs to be fault-tolerant, thus we have chosen the TMR implementation of the 8-bit PicoBlaze like in the FT Configuration Engine. This fault-tolerant spare processor uses the same amount of resources as the FT Configuration Engine (i.e., 80% of those of the simple processor).

Our solution requires significantly lower time overhead. Firstly, because the scan process which pauses the system is much shorter than the reconfiguration process of the whole dual-core block to correct an error according to the basic lockstep scheme. (Recall that the maximum time overhead when an error occurs in the enhanced lockstep is the scan time and the proposed rollforward recovery time.) Secondly, the time overhead of the proposed enhanced lockstep scheme, dominated by t_{scan}, is about 36 times greater than the t_{rec} (the proposed rollforward recovery duration). Furthermore, the proposed rollforward recovery strategy used in our system is always faster than the check-pointing

and rollback strategies used in a basic lockstep scheme. The time overhead when an error occurs in the basic lockstep scheme is dominated by the correction process using dynamic reconfiguration (more than 20000 times longer than one rollforward recovery). Moreover, the major drawback of the basic lockstep is low continuity of services, because after error occurrence, its dual-core module needs to be reconfigured immediately to correct error.

Compared to the TMR softcore approach from [74], our solution enjoys an advantage of smaller hardware overhead. Although the TMR softcore system requires very short time overhead, the reconfiguration process is done externally using computer which connects to the external reconfiguration port. The major advantage of our system is that the reconfiguration is performed in real-time without external intervention and that the enhanced lockstep scheme proposed here provides as high continuity of services as the TMR softcore approach. Especially, a huge advantage of low hardware overhead comes with the design of an MPSoC system due to the presence of only one FT Configuration Engine in one FPGA. An MPSoC system applying our approach incurs a constant hardware overhead of the FT Configuration Engine (440 slices) and 560 slices for each added processor. In an MPSoC system applying TMR, 1120 slices (2×560) overhead is required for each added processor. Fig. 5.13 gives a comparison between the two approaches in term of hardware overhead varying in function of processor (μP) numbers.

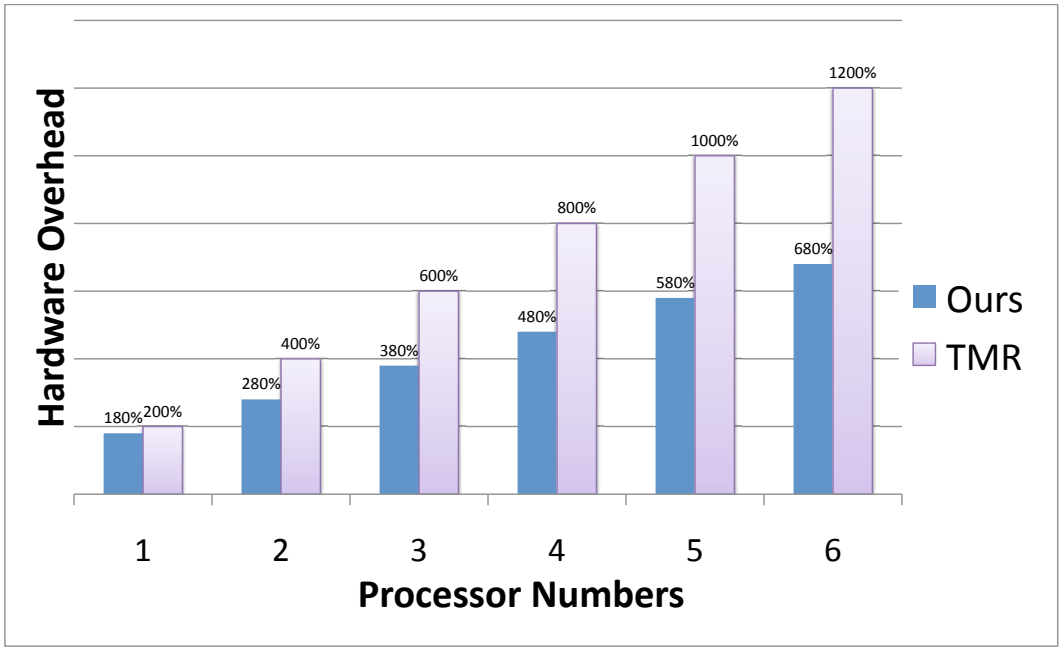


FIGURE 5.13: MPSoC systems with hardware overhead comparison (560 slices = 100%)

To validate the efficiency of the fault-tolerant approach, we have implemented a fault-injection campaigns for μP1 and the COMP_MUX, realized by the FT Configuration Engine. Single configuration faults are injected using the configuration scrub technique (Section 2.5.2 p. 32) through ICAP. First, the chosen configuration frame is read, one bit (indicated by its position inside the frame) is flipped, and then the fault-injected frame is written also using configuration scrub. The configuration upset must be corrected before the fault-injection of the next configuration bit. The automatic fault-injection engine is performed by the Configuration Engine with all the bits of μP1 and the COMP_MUX. μP1 occupies 30 configuration frames, whereas the COMP_MUX occupies 4 frames. Since the malfunction occurred due to the faults in the BRAM of the COMP_MUX are not evident to observe, we do inject faults only in 60 slices of this block. Each configuration frame contains 1312 bits (41 words of 32-bits).

TABLE 5.3: Statistic fault-injection results

Module	Slices	Utilization	Sensitive Bits	Sensitivity	Persitence	Code Size	BRAM Sensitivity	BRAM Persistence
μP1	560	88%	3165	8.6%	2.3%	3.3 KBytes	2.97%	0.59%
COMP_MUX	60	83%	83	8.4%	0.4%			

A matrix multiplication software code ([32][32]×[32][32]), which has the size of 3.3 KBytes, runs in the program memory BRAM of μP1. Also three slice configuration frames of the COMP_MUX are investigated. The BRAM configuration frame of the COMP_MUX is not investigated. The results of the fault-injection campaigns in the slice and BRAM configuration frames of μP1 and slice configuration frames of the COMP_MUX are shown in Table 5.3. The Configuration Engine performs the fault correction process after each fault-injection. The Configuration Engine first uses the configuration scrub to correct non-sensitive and non-persistent sensitive configuration bits. Then, in case of persistent bits, the partial reconfiguration of the whole μP1 is performed to put it back to the normal operation. Thanks to this fault-injection engine, it is possible to find out the sensitive bits of all the configuration frames. So that in case of error, we scan the configuration frames in the order of their sensitivities, the configuration frame which has the most sensitive bits will be scanned first. Based on the fault-injection results given in Table 5.3, the average interrupted duration caused by a configuration error in μP1 is: 32.7 μs × 2.3% + 27.5 μs × (8.6 - 2.3)% = 2.5 μs.

If we take the case of the lockstep without the identification capability. The two processor modules have to be reconfigured at the same time to eliminate the sensitive error which

requires 7.5 ms to complete. The average interrupted duration caused by a configuration error in the lockstep is: 7.5 ms $\times$ 8.6% = 65 μs, which is 26 times longer than our enhanced lockstep.

5.7 Conclusion and Perspectives

We have proposed a new architecture of a fault-tolerant reconfigurable system which can be implemented on any FPGA with integrated softcore processors. An enhanced lockstep scheme built using a pair of MicroBlaze cores was implemented on Xilinx Virtex-5 FPGA. Unlike the basic lockstep scheme, our system allows to identify the faulty core using a configuration engine built using PicoBlaze cores which, to avoid a single point of failure, was implemented as fault-tolerant using TMR. The exact error location is determined by a specially designed *Bitstream Parser* which allows to correct errors through partial reconfiguration combined with rollforward recovery technique. As a result, there is no problem of fault latency, because faults are detected immediately once they cause an error. Compared to similar existing designs, the new architecture enjoys smaller hardware and/or time overhead and does not need any external reconfiguration support. To note that the proposed system can also be adapted to the new Virtex-6 devices which also contains FRAME_ECC and ICAP primitives.

To validate the effectiveness of the fault-tolerant architecture proposed, we have provided the configuration engine with the possibility of on-line fault injection that allows us to carry out automatically fault-injection campaigns and provides statistically meaningful results.

Chapter 6

Conclusion and Perspectives

6.1 Conclusion

The rising computing need of electronic products demands large scale integration of electronic control. The automobile industry has evolved from mechanical control to mechatronic control so-called X-by-Wire. A modern car integrates a lots of additional services i.e.: navigation, infotainment and Advanced Driver Assistance Systems (ADAS) which can contains up to hundreds of ECUs. In particular, automotive control requires high reliability safety of drivers and pedestrians. Hence electronic products need to be hardened either by fabrication technology or by implementation design techniques. Harden circuit by fabrication technology is really costly and this process is only suitable for high-volume production. By doing that process, each particular company has its own Original Equipment Manufacturer (OEM) products which cannot be replaced by another company's solutions. Commercial FPGAs become a good candidate to be appeared in mission-critical environment because of the high-density reconfigurable resources, cheaper prices and low maintenance services. However, FPGAs are sensitive to electronic particles, and they require fault-tolerance schemes to be applied in the system. In particular, using commercial SRAM-based FPGAs are very valuable remote critical missions due to their possibility of being reprogrammed by the as many times as necessary in a very short period. As a result, additional benefits of SRAM-based FPGAs such as short time-to-market, short development time and high service continuity are offered. Moreover dynamically reconfigurable FPGAs allow for online design changes and thus reduce the cost by correcting errors or improving system performance after

deployment. The increase of logic complexity of the programmable logic with more and more embedded reconfigurable logic in one FPGA device can respond to the computation need growth in automotive electronic. The human safety constraints require the integration of fault-tolerance scheme in the electronic products to adapt to the variously harsh environment.

During this thesis, we studied and considered state-of-the-art solutions of the fault-tolerance which does not only target the automotive domain. We have managed to bring some novelties onto many fault-tolerant aspects for safety-aimed solutions. We have implemented a fully dynamic fault-tolerant MPSoC which can deal with temporary and permanent fault in the FPGAs. Several fault mitigation schemes were proposed and implemented in our FT-DyMPSoC system: connection matrix algorithm to detect the fault, partial reconfiguration and *tiling* to correct error, and rollback combined with checkpoint to recover the system software context after error occurrence.

We have proposed and implemented a Computer-Aided Design (CAD) extension that modifies the standard methodology of the run-time self-reconfiguration. This CAD extension facilitates and speed-up the FT-DyMPSoC construction. The concept of *socket* and *wrapper* is to ease the designer modification on original design. The intervention of designer is shorten because the system complexity is already managed by the CAD tools. The design flow modification is not only limited to building-up our FT-DyMPSoC, but also allows for constructing any complex modular system with necessary minor change on initial design.

Analytical model is proposed to evaluate the trade-off of various fault-tolerance schemes. Using this model, we evaluated our FT-DyMPSoC system compared with *scrubbing*. Slight decrease in performance can achieve a considerable gain in reliability. The model has a wide applicability which is not restricted to our system, but can be applied to any fault-tolerant scheme by feeding the appropriate parameters to the model.

The scales of the designs are increasing, and it take much more time for the total design period. Conventionally, verification takes up a large part of the total design period. One method to reduce design and validation time for MPSoCs is to use high-level simulation with modeling languages such as SystemC, OpenVera,.... The model permitting high-level simulation/verification provides designers opportunities to run faster, high-level simulation models, and compared with detailed hardware models. Thus the simulation

models allow for first, quickly verifying the effectiveness of the fault-tolerance schemes applied in the system, secondly speeding-up the period of user-application design and debug. A high-level model for fault-tolerant systems is built to help verify the fault-tolerance aspects as well as develop quickly applications associated with the system.

A low overhead system for reconfigurable softcore processor is also proposed and implemented. The system consisting of dual processor module and two particular modules: fault-tolerant configuration engine and COMP_MUX, enjoy the low hardware and timing overhead advantages. Besides reconfigurable processors, these particular modules have multi-functional capability that can use for any module to enhance the reliability of the target design.

We have implemented a software piece able to extract and write back the software context of softcore processors MicroBlaze, thanks to the capability of read/write/modify the functional registers inside the MicroBlaze. It could be applied also to hardcore processors PowerPC. So when integrating another module into the system with the fault tolerant configuration engine and the COMP_MUX, an effective recovery technique of the related module should be taken into account.

6.2 Perspectives

During the thesis, we have managed to complete all the proposed solutions. There exists anyway several future works to help enhance our proposals and some perspectives beyond the thesis.

In 5.3.2, we introduce the fault-tolerant configuration engine (FT Configuration Engine) that can be integrated in FT-DyMPSoC presented in Chapter 3. The combination with FT Configuration Engine can avoid duplication of processor in the lockstep scheme of FT-DyMPSoC which can save a huge hardware cost. Nevertheless the system may suffer a penalty on performance because of correcting non-sensitive bits which are measured much more numerous than sensitive bits. One proposal is to use an algorithm covering all logic blocks for consistency check. The algorithm is for functional checking that verify the state of all the blocks inside processors. Hence lockstep scheme with processor duplication is no longer necessary. That solution enjoy a lower hardware cost but a performance degradation could appear due to the functional blocks consistency check.

This solution is better than the architectural check such as readback, because the non-sensitive bits which do not affect the computing process are not detected by the functional check, so does not induce system performance decrease.

The proposed analytical model provides a fast evaluation, verification method for fault mitigation schemes. However, the accuracy of the model still needs to be validated which requires different results from various fault-injection campaigns on various fault mitigation schemes. The same scope needs to be taken into account for the high-levle simulation/verification model. A more complete model that can provide more precise results will be carried out. On possibility is to use Instruction Set Simulator (ISS) to provide more timing accuracy. The construction of proposed systems requires lots of manual intervention that demands designers to have certain understandings on the design tools and devices. That could prevent the proposals from being accepted from large public. The envisaged automation will help to facilitate the system built-up with minor intervention required from, hence easily handled by the designers.

We plan to automate the CAD extension of the design flow for complex module-based reconfiguration. The auto-generation of *socket* and *wrapper* will accelerate the design process and minimize the designer intervention period. In this automatic program, we will integrate also the configuration engine which helps detect and also inject any configuration error. The COMP_MUX auto-insertion is also carried out if the designer wants to implement the enhanced schemes like our proposal. A general-purpose fault-injection tool with user-friendly interface will also be implemented in the CAD extension. Instead of using expensive ion-bombardement processes, this tool will help designers to test their fault-tolerance strategies with the particular module as well as with benchmark design under test (DUT) modules.

We implemented in our proposed system software context recovery strategies for soft-core processors. In some cases, the context of hardware peripheral blocks also need to be saved and restored if necessary. That requires the possibility of reading/writing the functional registers defining the block states. The architectural registers of the reconfigurable elements can be read using readback capture. Nevertheless, these values can be only used if the same block with the same place and route implementation are applied onto the FPGA matrix, thus they are useless due to a permanent fault which requires *tiling* technique with a different place and route strategy of the same functional block.

A generic method allowing for extracting functional registers of hardware modules will be studied and realized in the short term.

Regarding the multi-FPGA platform, in stead of exchanging the detection frames with random data, we plan to integrate the software processor context inside the detection frame. Exchanging this frame through the ethernet network, on the one hand can help to detect error in the entire system, on the other hand help recover the processor with the context which is previously exchanged. Thus, it help the system to enhance the service continuity and minimize the functional interrupt.

During the thesis, we take into concern the failure effects of the circuit and are interested in how to overcome these problems. However another failure that can be important in mission-critical application is systematic failure. This kind of failure produced by the error during the design process, not by a circuit defect. An error in programming the processor, a missing signal while coding hardware block in VHDL, all sorts of systematic failure can induces catastrophic consequences on people and environment. This kind of failure can not be corrected using normal homogenous redundancy, since the error appears in all the copies of the redundancy. We mention here the functional safety [76] which requires the design diversity to avoid systematic failures and also manage random hardware failures. A given functionality should be implemented in different ways at the same time, and the results are compared to choose the correct one. For example, instead of implementing a function in the softcore processor only, we implement this function on the softcore processor, on the hardcore processor and also on a DSP processor. A special majority voter chooses the correct results from these three outputs. The function for three targets needs three different ways of programming that can avoid systematic error in one programming. This diverse platform can also deal with temporary hardware failure. It is interesting to imagine to design a new FPGA with dynamically reconfigurable resources. In this FPGA matrix, we could find a DSP processor, a hardcore generic processor like PowerPC or ARM. Several radiation-hardened majority voters are also present in the FPGA matrix for homogenous redundancy as well as diverse redundancy. In this new device matrix, all the proposals during the thesis can be quickly implemented. Furthermore, thanks to the diversity, the coding errors will appear in only one module because the development phase of each module is fully decoupled. Multiple level of safety should be handled using this new FPGA, which really fulfills automotive domain safety requirements.

Hardware platforms are quickly evolving from a single generic processor to multi-processor systems to respond to customer requirements. This trend forces the specific customer functionality to be integrated into system as software rather than hardware. As consequence, the added software functionality is causing an exponential growth rate in the automotive embedded software market. Traditionally, the schedule is carried out due to the availability of real-time operating system, which helps to shorten development phases and ensure the reactivity of the system. Thus software platforms should be compliant with industry standard to help alleviate the integration of new hardware platforms. Autosar [77] has been created to develop an open industry standard for automotive software architectures. To achieve the technical goals of modularity, scalability, transferability, and function reusability, Autosar provides a common software infrastructure based on standardized interfaces for the different layers. In Autosar, software developments are de-correlated with hardware architectures (ECUs) to allow reuse and relocation of vehicle functions. A new constraint in designing an automotive electronic system the software compliance with the standard Autosar.

Personal Publication

- **H-M. Pham**, S. Pillement, and D. Demigny. Reconfigurable ECU Communications in AUTOSAR Environment. In *International Conference on ITS Telecommunications*, Lille, France, October 2009.

- **H-M. Pham**, S. Pillement, and D. Demigny. A Fault-Tolerant Layer For Dynamically Reconfigurable Multi-Processor System-on-Chip. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs*, pages 284–289, Cancun, Mexico, December 2009.

- **H-M. Pham**, S. Pillement, and D. Demigny. FT-DyMPSoC: Analytical Model for Fault-Tolerant Dynamic MPSoC. In *International IEEE Symposium on Field-Programmable Custom Computing Machines*, May 2010.

- **H-M. Pham**, L. Devaux, and S. Pillement. Dynamic NOC-based MPSoC with Fault-Tolerance Support. In *DAC Workshop on Diagnostic Services in Network-on-Chips*, June 2010.

- **H-M. Pham**, S. Pillement, and D. Demigny. Evaluation of Fault-Mitigation Schemes for Fault-Tolerant Dynamic MPSoC. In *International Conference on Field Programmable Logic and Applications*, Milano, Italy, September 2010.

- **H-M. Pham**, S. Pillement, and S. Piestrak. Low Overhead Fault-Tolerance Technique for Dynamically Reconfigurable Softcore Processor. Submitted to *IEEE Transactions On Computers*.

Abbreviations

ADAS	**A**dvanced **D**river **A**ssistance **S**ystem
API	**A**pplication **P**rogram **I**nterface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
CAD	**C**omputer-**A**ided **D**esign
CIFAER	**C**ommunication **I**ntra-véhicule **F**lexible et **A**rchitecture **E**mbarquée **R**econfigurable
CLB	**C**onfigurable **L**ogic **B**lock
COMP_MUX	**COMP**arator/**MU**ltiple**X**er
COTS	**C**ommercial **O**f-**T**he-**S**helf
CPLD	**C**omplex **P**rogrammable **L**ogic **D**evice
CRC	**C**yclic **R**edundancy **C**heck
DRAFT	**D**ynamic **R**econfigurable **A**daptive **F**at-**T**ree
DSP	**D**igital **S**ignal **P**rocessing
DUT	**D**esign **U**nder **T**est
DWC	**D**uplication **W**ith **C**omparison
ECU	**E**lectronic **C**ontrol **U**nit
EDAC	**E**rror **D**etection **A**nd **C**orrection **C**oding
EDK	**E**mbedded **D**evelopment **K**it
FAR	**F**rame **A**ddress **R**egister
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSL	**F**ast **S**implex **L**inks
FSM	**F**inite **S**tate **M**achine
FT-DyMPSoC	**F**ault-**T**olerant **Dy**namic **M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
HCE	**H**ot-**C**arrier **E**ffect
ICAP	**I**nternal **C**onfiguration **A**ccess **P**ort

IP	**I**ntellectual **P**roperty
ISE	**I**ntegrated **S**oftware **E**nvironment
LUT	**L**ook-**U**p **T**able
MBU	**M**ultiple **B**it **U**pset
MPSoC	**M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
MSR	**M**achine **S**tatus **R**egister
NoC	**N**etwork-**o**n-**C**hip
NRE	**N**on **R**ecurring **E**ngineering
OEM	**O**riginal **E**quipment **M**anufacturer
PC	**P**rogram **C**ounter
PLB	**P**rocessor **L**ocal **B**us
PLC	**P**ower **L**ine **C**ommunication
PRM	**P**artially **R**econfigurable **M**odule
PRR	**P**artially **R**econfigurable **R**egion
RAMPSoC	**R**untime **A**daptive **M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
RB	**R**ecovery **B**us
Re^2 DA	**R**eliable and **R**ecofigurable **D**ynamic **A**rchitecture
RF	**R**adio **F**requency
RISC	**R**educed **I**nstruction **S**et **C**omputing
SAPECS	**S**ecured **A**rchitecture and **P**rotocols for **E**nhanced **C**ar **S**afety
SBU	**S**ingle **B**it **U**pset
SDK	**S**oftware **D**evelopment **K**it
SEC-DED	**S**ingle **E**rror **C**orrection-**D**ouble **E**rror **D**etection
SEE	**S**ingle **E**vent **E**ffect
SEFI	**S**ingle **E**vent **F**unctional **I**nterrupt
SEGR/SEB	**S**ingle **E**vent **G**ate **R**upture/**B**urnout
SEL	**S**ingle **E**vent **L**atch-up
SET	**S**ingle **E**vent **T**ransient
SEU	**S**ingle **E**vent **U**pset
TMR	**T**riple **M**odular **R**edundancy
VHDL	**V**ery-High-Speed Integrated Circuit (VHSIC) **H**ardware **D**escription **L**anguage

Bibliography

[1] Dhiraj. K. Pradhan and Nitin. H. Vaidya. Brief Contributions: Roll-Forward and Rollback Recovery: Performance-Reliability Trade-Off. *IEEE Transactions on Computer*, 46(3):372–378, 1997.

[2] TRW Automotive. http://ir.trw.com/.

[3] Xilinx, Inc. Virtex-5 FPGA Configuration User Guide UG191 (v3.6), 2009. URL www.xilinx.com/support/documentation/user_guides/ug191.pdf.

[4] S. Young. Maximizing Silicon ROI: The Cost of Failure and Success, 2002.

[5] http://www.insa-rennes.fr/ietr-cifaer.

[6] P. Tanguy, F. Nouvel, and P. Maziéro. Power Line Communication Standards for in-Vehicle Networks. In *International Conference on ITS Telecommunications*, 2009.

[7] Xilinx, Inc. Correcting Single-Event Upsets Through Virtex Partial Configuration (XAPP216 v1.0), June 2000. URL http://www.xilinx.com/support/documentation/application_notes/xapp216.pdf.

[8] H.C. Hsieh, W. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey, and R. Kanazawa. Third-generation Architecture Boosts Speed and Density of Field-Programmable Gate Arrays. In *IEEE Custom Integrated Circuits Conference*, pages 31.2.1–31.2.7, 1990.

[9] S. Trimberger. Effects of FPGA Architecture on FPGA Routing. In *The 32nd annual ACM/IEEE Design Automation Conference*, pages 574–578, 1995.

[10] Xilinx, Inc. *PowerPC 405 Processor Block Reference Guide*, 2004. URL www.xilinx.com/support/documentation/user_guides/ug018.pdf.

[11] Xilinx, Inc. MicroBlaze Processor Reference Guide UG081 (v10.3), 2009. URL http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf.

[12] Xilinx, Inc. PicoBlaze 8-bit Embedded Microcontroller User Guide UG129 (v2.0), January 2010.

[13] Altera Corporation. Excalibur Devices Hardware Reference Manual (V3.1), 2002.

[14] Altera Corporation. Nios II Processor Reference Handbook, 2005.

[15] Xilinx, Inc. Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations (XAPP290), 2002.

[16] Xilinx, Inc. *Early Access Partial Reconfiguration User Guide UG208*, September 2008.

[17] Xilinx, Inc. Virtex-6 FPGA Configuration User Guide UG360 (v3.1), July 2010.

[18] Xilinx, Inc. Virtex-4 FPGA Configuration User Guide UG071 (v1.11), June 2009.

[19] Xilinx, Inc. Embedded System Tools Reference Guide UG111, 2009.

[20] Xilinx, Inc. http://www.xilinx.com/.

[21] Xilinx, Inc. Xilinx PlanAhead User Guide (UG632 v11.4), Dec 2009. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/cgd.pdf.

[22] Nasa. Radiation Effects on Digital Systems. URL http://radhome.gsfc.nasa.gov/top.htm.

[23] Actel Inc. RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs, August 2010.

[24] Xilinx, Inc. Radiation-Hardened, Space-Grade Virtex-5QV Family Overview DS192 (v1.1), August 2010.

[25] G.E. Moore. Progress in Digital Integrated Electronics. In *Digest of the 1975 International Electron Devices Meeting*, pages 11–13, New York, 1975.

[26] R. Koga, SH Penzin, KB Crawford, and WR Crain. Single Event Functional Interrupt (SEFI) Sensitivity in Microcircuits. In *European Conference on Radiation and Its Effects on Components and Systems*, pages 311–318, 1997.

[27] S. Srinivasan, P. Mangalagiri, Y. Xie, N. Vijaykrishnan, and K. Sarpatwari. FLAW: FPGA Lifetime Awareness. In *The 43rd Annual Design Automation Conference*, page 635. ACM, 2006.

[28] A.A.M. Bsoul, N. Manjikian, and L. Shang. Reliability-and Process Variation-Aware Placement for FPGAs. In *Design, Automation and Test in Europe*, 2010.

[29] S. Mahapatra, R. Rao, B. Cheng, M. Khare, C.D. Parikh, JCS Woo, and J. Vasi. Performance and Hot-Carrier Reliability of 100 nm Channel Length Jet Vapor Deposited Si3N4 MNSFETs. *IEEE Transactions on Electron Devices*, 48(4):679–84, 2001.

[30] RC Baumann. Single-Event Effects in Advanced CMOS Technology. In *IEEE Nuclear and Space Radiation Effects*, 2005.

[31] Xilinx, Inc. PPC405 Lockstep System on ML310 (XAPP564 v1.0.2), 29 January 2007. URL http://www.xilinx.com/support/documentation/application_notes/xapp564.pdf.

[32] Atmel. Secured Architecture and Protocols for Enhanced Car Safety (SAPECS), 2007.

[33] F. Lima, L. Carro, and R. Reis. Designing Fault Tolerant Systems into SRAM-based FPGAs. In *Design Automation Conference*, pages 650–655. ACM New York, NY, USA, 2003.

[34] M. Nicolaidis. Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies. In *IEEE VLSI Test Symposium*. IEEE Computer Society Washington, DC, USA, 1999.

[35] W.W. Peterson and E.J. Weldon. *Error-Correcting Codes*. 1972.

[36] AD Houghton. *the Engineer's Error Coding Handbook*. Chapman & Hall, 1997.

[37] University of Erlangen-Nuremberg. Project ReCoNets. URL http://www.reconets.de/.

[38] Xilinx, Inc. Virtex FPGA Series Configuration and Readback XAPP138 (v2.8), March 2005.

[39] S.Y. Yu and E.J. McCluskey. Permanent Fault Repair For FPGAs With Limited Redundant Area. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 125–133, 2001.

[40] Xilinx, Inc. SEU Strategies for Virtex-5 Devices (XAPP864), 5 March 2009. URL http://www.xilinx.com/support/documentation/application_notes/xapp864.pdf.

[41] M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello. An Efficient and Low-Cost Design Methodology to Improve SRAM-Based FPGA Robustness in Space and Avionics Applications. In *Proc. Int. Workshop on Reconfigurable Computing: Architectures, Tools and Applications. LNCS*, volume 5453, pages 74–84, 2009.

[42] B. Dutton and C. Stroud. Single Event Upset Detection and Correction in Virtex-4 and Virtex-5 FPGAs. In *Int. Conf. on Computers and Their Applications*, pages 57–62, 2009.

[43] H. Castro, A.A. Coelho, and R.J. Silveira. Fault-Tolerance in FPGA's through CRC Voting. In *The 21st Annual Symposium on Integrated Circuits and System Design*, pages 188–192. ACM New York, NY, USA, 2008.

[44] C. Pilotto, J.R. Azambuja, and F.L. Kastensmidt. Synchronizing Triple Modular Redundant Designs in Dynamic Partial Reconfiguration Applications. In *The 21st Annual Symposium on Integrated Circuits and System Design*, pages 199–204, 2008.

[45] H. Zheng, L. Fan, and S. Yue. FITVS: A FPGA-Based Emulation Tool For High-Efficiency Hardness Evaluation. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 525–531. IEEE Computer Society, 2008.

[46] JBits 3.0 SDK. www.xilinx.com/labs/projects/jbits/.

[47] H. Guzmán-Miranda, M.A. Aguirre, and J. Tombs. Noninvasive Fault Classification, Robustness and Recovery Time Measurement in Microprocessor-Type Architectures Subjected to Radiation-Induced Errors. *IEEE Transactions on Instrumentation and Measurement*, 58(5), 2009.

[48] Xilinx, Inc. Single-Event Upset Mitigation Selection Guide (XAPP987 v1.0), March 2008. URL http://www.xilinx.com/support/documentation/application_notes/xapp987.pdf.

[49] A. Grama. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[50] C. Haubelt, D. Koch, and J. Teich. Basic OS Support for Distributed Reconfigurable Hardware. In *Computer Systems: Third and Fourth International Workshops, SAMOS 2003 and SAMOS 2004*. Springer, 2004.

[51] D. Gohringer, M. Hubner, T. Perschke, and J Becker. New Dimensions for Multiprocessor Architectures: On Demand Heterogeneity, Infrastructure and Performance Through Reconfigurability-The RAMPSoC Approach. In *International Conference on Field Programmable Logic and Applications*, pages 495–498, 2008.

[52] A. Montone, V. Rana, M.D. Santambrogio, D. Sciuto, and P. di Milano. HARPE: A Harvard-based Processing Element Tailored for Partial Dynamic Reconfigurable Architectures. In *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[53] A. Klimm, L. Braun, and J. Becker. An Adaptive and Scalable Multiprocessor System for Xilinx FPGAs Using Minimal Sized Processor Cores. In *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[54] Xilinx, Inc. Multi-Port Memory Controller (MPMC) (DS643 v6.00.a), April 2010.

[55] Xilinx, Inc. *Fast Simplex Link (FSL) Bus (DS449)*, June 2007. URL http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf.

[56] F. Abate et al. New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors. *IEEE Transactions on Nuclear Science*, 56(4):1992–2000, Aug. 2009.

[57] A. Kanamaru, H. Kawai, Y. Yamaguchi, and M. Yasunaga. Tile-Based Fault Tolerant Approach Using Partial Reconfiguration. In *Proc. Int. Workshop on Reconfigurable Computing: Architectures, Tools and Applications. LNCS*, volume 5453, pages 293–299, 2009.

[58] Wei-Je Huang and Edward J. McCluskey. Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance. In *Proc. Annu. Int. IEEE Symp. Field-Programmable Custom Computing Machines*, pages 137–146, 2001.

[59] Xilinx, Inc. Constraints Guide (UG625 v11.4), Dec 2009. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/cgd.pdf.

[60] Ludovic Devaux, Sana Ben Sassi, Sebastien Pillement, Daniel Chillet, and Didier Demigny. Flexible Interconnection Network for Dynamically and Partially Reconfigurable Architectures. *International Journal of Reconfigurable Computing*, 2010 (390545):10.1155/2010/390545, 2010.

[61] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38:1–51, 2006.

[62] E. Salminen, A. Kulmala, and T. D. Hamalainen. Survey of network-on-chip proposals. *OCP-IP White Paper,* `http://www.ocpip.org/whitepapers.php`, 2008.

[63] LightWeight IP. http://savannah.nongnu.org/projects/lwip.

[64] Xilinx, Inc. LogiCORE IP XPS Timer/Counter (DS573 v1.02a), April 2010.

[65] S. Tanoue, T. Ishida, Y. Ichinomiya, M. Amagasaki, M. Kuga, and T. Sueyoshi. A Novel States Recovery Technique for the TMR Softcore Processors. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 543–546, 2009.

[66] Xilinx, Inc. Device Reliability Report (UG116), Third Quarter 2009.

[67] K. Kyriakoulakos and D. Pnevmatikatos. A Novel SRAM-Based FPGA Architecture for Efficient TMR-Processor Fault Tolerance Support. In *International Conference on Field Programmable Logic and Applications*, 2009.

[68] Austin Lesea. Continuing Experiments of Atmospheric Neutron Effects on Deep Submicron Integrated Circuits (WP286). Technical report, Xilinx Inc., 2009. URL www.xilinx.com/support/documentation/white_papers/wp286.pdf.

[69] G, Beltrame, et al. High-Level Modeling and Exploration of Reconfigurable MP-SoCs. In *AHS-2008*, pages 330–337, 2008.

[70] C. Amicucci, et al. SyCERS: A SystemC Design Exploration Framework for SoC Reconfigurable Architecture. In *ERSA'06*, pages 63–69, 2006.

[71] S. Xu, et al. A Multi-MicroBlaze Based SoC System: From SystemC Modeling to FPGA Prototyping. In *RSP'08*, pages 121–127, 2008.

[72] M. Monchiero, et al. A Modular Approach to Model Heterogeneous MPSoC at Cycle Level. In *DSD'08*, pages 158–164, 2008.

[73] IBM. Instruction Set Simulator User's Guide (v1.3).

[74] Y. Ichinomiya, S. Tanoue, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi. Improving the Robustness of a Softcore Processor against SEUs by Using TMR and Partial Reconfiguration. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 47–54, 2010.

[75] H-M. Pham, S. Pillement, and D. Demigny. A Fault-Tolerant Layer For Dynamically Reconfigurable Multi-Processor System-on-Chip. In *International Conference on ReConFigurable Computing and FPGAs*, pages 284–289, Cancun, Mexico, December 2009.

[76] Exida. IEC 61508 Overview Report. Technical report, 2006. URL http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=e&wwwprog=seabox1.p&progdb=db1&seabox1=61508.

[77] AUTOSAR GbR. AUTomotive Open System ARchitecture. URL http://www.autosar.org/.

VU :

Le Directeur de Thèse

Didier DEMIGNY

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après autorisation pour autorisation de publication :

Le Président de Jury,