Conception d'un Système Embarque Sur et Sécurisé

Michele Portolan

▶ To cite this version:

Michele Portolan. Conception d'un Système Embarque Sur et Sécurisé. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2006. Français. NNT : . tel-00163980

HAL Id: tel-00163980

https://theses.hal.science/tel-00163980

Submitted on 19 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__/__|

T H E S E

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de

l'Ecole Doctorale d'« Electronique, Electrotechnique, Automatique, Télécommunications, Signal »

présentée et soutenue publiquement
par

Michele PORTOLAN

Le 06 Décembre 2006

Titre :

CONCEPTION D'UN SYSTEME EMBARQUE SUR ET SECURISE

Directeur de Thèse : Régis Leveugle

JURY

Mr. Guy MAZARE , Président
Mr .Eric MARTIN , Rapporteur
Mr. Michel ROBERT , Rapporteur
Mr. Régis LEVEUGLE , Directeur
Mr. Michel PIGNOL , Examinateur

« Tout a une fin, sauf la banane qui en a deux »

Proverbe Bambara

Remerciements

Il est impossible de synthétiser dans quelque petite ligne ma gratitude pour toutes les personnes qui, chacune de sa façon, m’ont aidé, soutenu ou simplement accompagné pendant ces trois longues années. Le seul fait de faire une liste implique d’indiquer une priorité et une préférence, quand en réalité on voudrait pouvoir donner à tout le mode la première place.
Je remercie Mr. Bernard Courtois, qui m’a permis d’effectuer cette thèse dans son laboratoire.
Je remercie spécialement mon directeur de thèse Mr. Régis Leveugle qui a eu confiance en moi en me proposant ce doctorat et en me donnant pleine liberté et indépendance dans l’organisation de mon travail. Indépendance que ne lui a jamais empêché de me suivre de près, en me donnant toujours conseil, guidance et soutien quand j’en ai eu besoin. Merci aussi pour avoir eu le courage de corriger mon français.
Merci à Mr. Guy Mazare, Professeur à l’INPG-ENSIMAG et Vice-président de l’INPG, pour m’avoir fait l’honneur de présider mon Jury de Thèse.
Je remercie Mr. Eric Martin, Professeur et Président de l’Université Bretagne Sud, pour avoir accepté d’être mon Rapporteur malgré les lourdes tâches attachées à sa fonction.
Merci à Mr. Michel Robert, Professeur à l'Université de Montpellier et Directeur du LIRMM, pour avoir été mon Rapporteur et m’avoir aidé avec ses remarques à valoriser au mieux mon travail dans mon rapport.
Je remercie Mr. Michel Pignol, Ingénieur R&D au CNES à Toulouse, pour avoir apporté son expérience dans le Jury et pour son appréciation de mon travail.
Un remerciement à toutes les personnes qui m’ont aidé dans mon travail. En vrac : Alexandre, Robin, Hubert, Abdelmajid, Frédéric, Vincent, Pierre.

Avant tout, merci à ma Nad d’être là. Aucun mot ne pourra jamais suffire pour dire tous ce que j’ai dans mon cœur, bien au-delà de ces pages. Et bien sûr pour avoir résisté et m’avoir soutenu tout au long de ces derniers mois de rédaction!
Grazie a Mamma e Papà: senza la vostra presenza, il vostro sostegno, il vostro affetto non sarei mai andato così lontano. Nei miei viaggi, nei miei sogni, nelle mie piccole follie.
Grazie alla mia cara Sorellina, che riesce sempre ad essere presente anche sommersa dai suoi ventimila impegni.
Grazie a mio Fratello, così lontano ma per fortuna sempre così vicino grazie ai miracoli della tecnologia. Che gli omini verdi siano sempre con te!
Merci à Françoise et Jean-Michel, qui ont accepté l’arrivé de ce petit italien bizarre à coté de leur fille.
Merci à tous les amis que j’ai trouvé au TIMA et qui m’ont rempli mes journées en m’aidant à m’échapper aux radiations de mon écran : Marçin, Guillaume, Karine, Hella, Yassin, Benoît, Luis, Paolo….pardon à tous ces dont j’ai oublié le nom, mais pas le visage.
Merci à tous les amis que j’ai rencontré aussi au dehors du travail : Salvo, Luca, Luca piccolo, Ema, Dave, Anna, Romano, Rosilde, Kristell, Khalil, Sam, Bast, Jérémy, Amandine….
Grazie ai miei amici italiani, che non dimentico e non mi hanno dimenticato nonostante le Alpi (e molto di più per alcuni): Toni, Andrea, Isa, Marco, Tex, Ciccio, Silvia, Mauri, Nick, Vale….
Merci aussi aux nombreuses bestioles qui étaient témoins de mes pauses de rédaction : les deux Choups, les 4 paires d’oreilles de Coulouvre, le chat de voisins, ~~Stefano~~ (ah no, l’ho già contato prima…)

Table des Matières

Introduction 1

1 Etat de l'art 3

1.1 Systèmes embarqués 3

1.2 Architecture de processeurs : principes 4

1.2.1 Pipeline 5

1.2.2 Caches 7

1.3 Systèmes d'Exploitation Embarqués 8

1.4 Sûreté de fonctionnement 10

1.4.1 Historique et terminologie 10

1.4.2 Fautes et modèles 13

1.4.3 Protection niveau matériel 15

1.4.4 Protection niveau système d'exploitation 24

1.4.5 Protection niveau logiciel 26

1.4.6 Approches mixtes 29

1.5 Sécurité - confidentialité 30

1.5.1 Chiffrement et sécurité 30

1.5.2 Cryptanalyse et attaques par fautes 32

1.6 Conclusions 35

2 Système embarqué cible et approche globale de protection 37

2.1 Méthodologie et exemple représentatif 37

2.1.1 Le processeur Leon2 38

2.1.2 Le système d'exploitation eCos 39

2.1.3 IPs ad hoc : AES et interface vidéo 40

2.1.4 Les applications logicielles 41

2.2 Protection Niveau Système d'Exploitation 41

2.2.1 Recouvrement à coût minimal : principes 41

2.2.2 Contraintes et limitations 44

2.2.3 Les communications inter - tâches 48

2.2.4 Fonctionnalités avancées du système d'exploitation 50

2.3 Protection niveau matériel : approche RTL 50

2.3.1 Approche micro-architecturale 51

2.3.2 Redondance d'information : codage des données 51

2.3.3 Protection des éléments séquentiels 52

2.3.4 Protection de la logique combinatoire 54
2.3.5 Cas d'un pipeline : détection et recouvrement 56
2.3.6 Redondance temporelle : protection de l'IP AES......................... 58
2.4 Protection niveau logiciel 58
2.5 Coopération entre niveaux 59
2.6 Conclusions.. 61
3 Implantation pratique des approches proposées......................63
3.1 Mesures de complexité 63
3.2 Ajout du recouvrement dans le système d'exploitation eCos............................ 64
3.2.1 Implantation du recouvrement.. 64
3.2.2 Modification des sources eCos 68
3.2.3 Critères de validité dans l'application vidéo 69
3.3 Durcissement du processeur Leon2 72
3.3.1 Protections ajoutées et surcoûts... 72
3.3.2 Automatisation des protections... 78
3.4 Implantation collaborative entre mémoire cache et OS 80
3.5 Sécurisation du coprocesseur AES 81
3.6 Sécurisation de la version logicielle de l'AES 83
3.7 Conclusions.. 83
4 Prototype global : évaluation des coûts et de l'efficacité..........85
4.1 Flot de conception global du prototype.. 85
4.1.1 Plateformes cible 85
4.1.2 Flot de conception matériel 88
4.1.3 Flot de conception logiciel.. 89
4.1.4 Versions du prototype.. 91
4.2 Stratégie de mesure 92
4.2.1 Mesures de performances 92
4.2.2 Couverture de fautes 93
4.3 Performances : analyse statique 94
4.3.1 Point communs 94
4.3.2 Application au prototype : performances de eCos........................ 95
4.4 Cas particulier : comparaison des implantations logicielle et matérielle du chiffrement AES.. 98
4.5 Couverture de fautes : analyse statique.. 98
4.6 Efficacité de la collaboration OS-matériel : analyse dynamique...................... 100

4.7 Performances et couverture de fautes : analyse dynamique automatisée 104
4.7.1 Environnement d'injection de fautes 104
4.7.2 Implantation de Leon2 dans l'environnement 106
4.7.3 Spécification des campagnes d'injection de fautes 108
4.8 Conclusion 112
5 Conclusions 113
6 Bibliographie 115
7 Publications obtenues pendant la thèse 119
8 Glossaire 120

Table des Figures

Figure 1-1 Machine de Turing ... 4

Figure 1-2 Schéma de base d'un processeur ... 5

Figure 1-3 Pipeline du processeur DLX. On peut remarquer comment chaque étage correspond à de la logique indépendante [Henne96] ... 6

Figure 1-4 Remplissage du pipeline du processeur DLX. Noter au cycle 5 le moment idéal quand chaque étage est en train d'élaborer une instruction différente [Henne96] ... 7

Figure 1-5 Cas typique de hiérarchie mémoire, avec ordres de grandeur des tailles et des vitesses ... 7

Figure 1-6 Flot d'exécution global typique d'un système multitâche ... 9

Figure 1-7 Flot d'exécution d'une tâche dans un système multitâche ... 10

Figure 1-8 Schéma de la succession faute-erreur-défaillance ... 11

Figure 1-9: Ionisation produite par une particule incidente dans une jonction PN ... 13

Figure 1-10 Triplication du type Von Neumann ... 16

Figure 1-11 Triplication des bascules avec horloge décalée ... 17

Figure 1-12 Masquage d'un SET grâce au décalage d'horloge ... 17

Figure 1-13 Réplication m-n modulaire avec réserves avant (a) et après (b) reconfiguration ... 18

Figure 1-14 Duplication avec détection ... 19

Figure 1-15 Exemple de circuit auto-contrôlable implanté en logique Dual Rail ... 19

Figure 1-16 Exemple de triplication par redondance temporelle ... 20

Figure 1-17 Duplication et détection par redondance temporelle ... 20

Figure 1-18 Détection par calcul inverse avec module inverse (a) et en redondance temporelle pure (b) ... 21

Figure 1-19 Exemple de code correcteur : m' peut être reconduit à m parce que c'est le mot de code le plus proche. ... 22

Figure 1-20 Schéma typique d'une mémoire RAM protégée par code de Hamming. ... 23

Figure 1-21 Organisation du système STAR, prise de [Pradh96] ... 23

Figure 1-22 Exemple de recouvrement par « Checkpoint and Rollback » ... 27

Figure 1-23 Le processeur primaire envoie les informations de sauvegarde au processeur secondaire dans une architecture Tandem. Image prise de [Tande90]. ... 29

Figure 1-24 Schéma de cryptographie à clé secrète ... 31

Figure 1-25 Schéma de cryptographie à clé publique ... 32

Figure 1-26 Récapitulatif des techniques utilisées en cryptanalyse ... 35

Figure 2-1 Schéma du système cible ... 37

Figure 2-2 Schéma bloc du processeur Leon2 ... 39

Figure 2-3 Changement de contexte dans une machine de Turing ... 43

Figure 2-4 Checkpoint & Rollback basé sur le changement de contexte ... 44

Figure 2-5 Périmètre de modification des données avec un cache "write-back" ... 46

Figure 2-6 Exemple de déviation par rapport à la référence dans une application de traitement de signal ... 47

Figure 2-7 Bascule typique et son chronogramme en situation normale (a) et en présence d'une faute (b) ... 52

Figure 2-8 Protection du registre R à travers des blocs de codage C, avec contrôleur double rail ... 53

Figure 2-9 Protection d'une RAM embarquée à travers des blocs de codage C, avec contrôleur double rail ... 54

Figure 2-10 Exemple de protection d'un bloc combinatoire par redondance d'information ... 55

Figure 2-11 Protection par prédiction de parité automatisée ... 55

Figure 2-12 Protection par prédiction de code générique avec contrôleur double rail ... 56

Figure 2-13 Protection par prédiction de code d'un étage i d'un pipeline ... 57

Figure 2-14 Recouvrement par prédiction de code et gel du pipeline ... 57

Figure 2-15 Schéma de base d'implantation d'un cache des victimes modifiées ... 60

Figure 3-1 Schéma interne d'une cellule de base dans un FPGA Virtex-II Pro (pris de la documentation Xilinx) ... 64

Figure 3-2 Exemple d'enchaînement setjmp-longjmp avec conservation (a), avec perte (b) et avec réécriture (c) de la pile ... 65

Figure 3-3 Algorithme de base de l'ordonnanceur (a) et insertion du setjmp (b) ... 66

Figure 3-4 Boucle de compression/Décompression JPEG (image prise de [Wikip06]) ... 69

Figure 3-5 Image originale "Lena" en bitmap (a) et compessée en JPEG avec une qualité de 20/100 (b) ... 70

Figure 3-6 Lena reconstituée à partir du fichier JPEG compressé sans (a) et avec simulation de recouvrement (b) ... 71

Figure 3-7 Lena comprimée en JPEG visualisée à l'écran par le prototype ... 72

Figure 3-8 Structure de l'unité centrale du Leon 2. Les parties protégées sont hachurées ... 72

Figure 3-9 Propagation d'un signal indéfini dans un arbre de XOR ... 73

Figure 3-10 Protection des caches par parité ... 75

Figure 3-11 Implantation d'une mémoire RAM dans une FPGA ... 77

Figure 3-12 Modélisation d'une entité générique E ... 79

Figure 3-13 Entité de la Figure 3-12 protégée par réplication automatique ... 79

Figure 3-14 Implantation de la mémoire de victimes modifiées ... 80

Figure 3-15 Algorithme du bloc de multiplexage entre IU, CD et DVC ... 81

Figure 4-1 Implantation du prototype sur la carte Avnet ... 86

Figure 4-2 Prototype Vidéo sur la carte Avnet ... 86

Figure 4-3 Prototype Vidéo sur la carte Multimedia ... 87

Figure 4-4 Interface graphique de configuration des protections matérielles ... 88

Figure 4-5 Flot de développement matériel ... 89

Figure 4-6 Outil de configuration de eCos, avec les options de protection ... 90

Figure 4-7 Flot de développement logiciel ... 91

Figure 4-8 Classification des fautes ... 93

Figure 4-9 Comparaison des grandeurs de base de eCos en configuration originale et avec C&R ... 96

Figure 4-10 Comparaison des grandeurs de base de eCos en configuration originale et avec détection matérielle ... 96

Figure 4-11 Comparaison des grandeurs de base de eCos en configuration originale et avec recouvrement matériel ... 97

Figure 4-12 Comparaison des grandeurs de base de eCos en configuration originale et avec recouvrement matériel à la même fréquence de travail 97

Figure 4-13 Reproduction de la figure 2-11 "Protection par prédiction de parité automatisée" 99

Figure 4-14 Exemple d'injection de fautes à l'aide de portes XOR 101

Figure 4-15 Schéma de la plate-forme d'injection de fautes développée par Pierre Vanhauwaert 105

Figure 4-16 Algorithme de l'interface PPC-CUT en modalité prototypage (a) et injection de fautes (b) 106

Figure 4-17 Duplication des registres de l'unité entière pour simuler l'injection des SETs 107

Tableaux

Tableau 3-1 Surcoûts statiques pour la protection du système d'exploitation... 67

Tableau 3-2 Résumé des protections dans l'unité centrale du Leon 2.. 73

Tableau 3-3 Surcoût statique des protections pour l'Unité Entière du Leon2... 76

Tableau 3-4 Surcoût statique des protections pour l'Unité Entière et le système de caches du Leon2........ 76

Tableau 3-5 Synthèse de Leon2 sans RAMs sur cellules AMS 3.60 à 80MHz 78

Tableau 3-6 Résultats de synthèse par Leonardo pour le coprocesseur AES sur un VirtexII-Pro VP20.... 82

Tableau 3-7 Surcoût des protections pour le coprocesseur AES.. 82

Tableau 3-8 Comparaison de la taille en octets de différentes implantations de l'algorithme AES 83

Tableau 4-1 Temps d'exécution de l'algorithme AES à 27Mhz avec une clé de 128 bits.............................. 98

Tableau 4-2 Estimation des bits pris en compte selon la stratégie de filtrage... 100

Tableau 4-3 Résumé des protections matérielles dans Leon2 et l'IP AES 108

Tableau 4-4 Résumé des protections logicielles et OS.. 109

Tableau 4-5 Résumé des applications disponibles... 109

Introduction

Un nombre croissant d'applications repose aujourd'hui sur l'utilisation de circuits intégrés complexes et, plus largement, sur l'utilisation de systèmes embarqués incluant matériel et logiciel (système d'exploitation et logiciel d'application) intégrés dans un même circuit. La confiance pouvant être accordée à de tels systèmes dépend de leur capacité à réagir de façon sûre (c'est à dire, non dangereuse pour l'application) lorsque des fautes surviennent pendant l'exécution.

L'évolution des technologies CMOS se traduit par un fort accroissement de la sensibilité des circuits à différents parasites (rayonnements ou particules, même au niveau de la mer). La probabilité de fautes transitoires perturbant le fonctionnement des circuits tend donc à devenir inacceptable, même dans des applications grand public. Ceci rend nécessaire un durcissement des circuits et des applications. Classiquement, au niveau circuit, des dispositifs de protection sont intégrés et permettent soit la réalisation d'un test en ligne (détection d'une erreur survenant pendant le fonctionnement de l'application), soit la tolérance de certaines fautes (maintien du comportement nominal en présence de ces fautes). Des méthodes similaires peuvent être employées pour durcir le logiciel embarqué.

Pour les applications nécessitant une sécurité élevée, des fautes volontairement générées par des techniques d'attaque viennent s'ajouter aux fautes "naturelles" précédemment mentionnées. Ces fautes intentionnelles ont pour objectif de saturer les dispositifs de sécurisation implantés afin d'accéder aux informations sensibles présentes dans le circuit (par exemple, une clé privée associée à un algorithme de chiffrement). Ce type de fautes peut avoir des caractéristiques très différentes des fautes "naturelles" et demande donc des techniques de protection adaptées.

Cette thèse s'est donc attachée à définir une méthodologie globale permettant d'atteindre un niveau de sûreté et de sécurité donné face à des fautes logiques (naturelles ou intentionnelles) survenant dans un système intégré monoprocesseur, représentatif par exemple d'une carte à puce. La recherche n'a toutefois pas été limitée à ce domaine ; les résultats peuvent être appliqués à tout circuit construit autour d'un cœur de microprocesseur et d'un ensemble de périphériques spécialisés. Les méthodes de protection portent simultanément sur le matériel, le logiciel d'application et les couches d'interface (en particulier, le système d'exploitation). Les approches à haut niveau d'abstraction ont été privilégiées parce que leurs caractéristiques les rendent très intéressantes du point de vue de la généralité, configurabilité et adaptabilité des interventions. Le cas des systèmes multiprocesseurs est nettement différent et n'a pas été abordé dans cette thèse.

Une étude bibliographique couvrant l'ensemble des aspects cités a permis de résumer l'état de l'art du domaine dans le Chapitre 1 de ce manuscrit. Cette étude a conduit à dégager des pistes intéressantes pouvant conduire à une approche basée sur la coopération des différents composants du système.

Le champ d'analyse étant très vaste, on a choisi de cibler la recherche sur un sous-ensemble précis de menaces : les fautes transitoires localisées. Ceci permettra de faire un certain nombre d'hypothèses, notamment l'absence de simultanéité de modifications dans deux copies d'une même donnée, mémorisées dans des blocs fonctionnels nettement séparés. Ceci sera utilisé par exemple lorsque nous traiterons des mémoires cache et de la mémoire centrale. De la même façon, ceci permet de supposer qu'il n'y a pas simultanément erreur dans le processeur et dans les échanges avec la mémoire.

Pour éviter le risque de rester limité à des discussions théoriques, on a décidé dès le début de spécifier et de réaliser un système embarqué le plus représentatif possible, à utiliser comme cible pour démontrer la faisabilité et l'efficacité des techniques de protection développées. Pour cela, on a défini un système monoprocesseur pouvant exécuter de nombreuses applications grâce à un système d'exploitation. Le coeur de traitement est par ailleurs connecté à des blocs fonctionnels ad hoc (IPs) dédiés aux communications avec l'environnement ou à des accélérations matérielles. Le tout est implanté sur une carte de prototypage à base de FPGA. Ce choix permet d'obtenir un système de complexité importante et très représentatif d'un équipement réel, d'autant plus que pour les deux composants les plus importants, le cœur de processeur et le système d'exploitation, on a choisi d'utiliser des produits Open-Source : le processeur Leon2 et le système d'exploitation eCos. Comme IP spécifiques, on a choisi un coprocesseur cryptographique réalisant l'algorithme AES et une interface d'acquisition vidéo. Ce système peut être facilement configuré pour l'adapter aux besoins d'une application et la licence Open-Source nous donne pleine liberté d'accès et modification des sources. Le Chapitre 2 décrit tout d'abord en détail ce système. Il présente ensuite le principe des méthodes proposées pour améliorer la sûreté en intervenant aux différents niveaux. En particulier, des approches originales seront présentées au niveau du système d'exploitation ou en couplant des modifications du système d'exploitation et un support au niveau matériel. Il faut souligner que l'objectif du travail n'est pas d'obtenir une protection parfaite, mais le meilleur compromis entre robustesse et coûts d'implantation. Concernant le système d'exploitation, l'approche proposée ne permet pas de protéger l'exécution des fonctions du système lui-même (noyau et appels système). Si ces fonctions correspondent à une part notable du temps d'exécution, il est donc nécessaire de compléter l'approche qui sera présentée dans ce document.

L'implantation sur FGPA est un choix de souplesse, qui permet de développer rapidement un prototype, modifiable ensuite à souhait. Cependant, les techniques de protection proposées ne sont pas spécifique pour cette cible technologique et peuvent être employées dans d'autre cas, notamment une implantation ASIC à base de cellules précaractérisées. Par ailleurs, les fautes spécifiques aux FPGA, comme par exemple les fautes dans la mémoire de configuration, sont en dehors des objectifs de cette thèse et ne seront pas prises en compte.

Le passage de l'analyse théorique à une réalisation réelle n'est pas toujours élémentaire. Le Chapitre 3 est dédié à ce passage, avec une analyse détaillée des cas où les hypothèses théoriques ont pu être directement appliquées, mais aussi des modifications causées par les contraintes d'une application réelle.

Le Chapitre 4 est ensuite consacré à l'analyse des résultats obtenus. Une attention particulière est apportée à la comparaison entre le système de base et le système protégé, plutôt qu'aux caractéristiques absolues. Il faut en effet se souvenir que l'objectif est le développement d'un système représentatif sur lequel sont appliquées des techniques de protection générales et non pas spécifiquement développées pour ce système particulier. L'analyse des variations introduites permet donc d'évaluer l'effet que ces mêmes techniques peuvent avoir si elles sont appliquées à un autre système utilisant d'autres composants de base.

1 Etat de l'art

Ce chapitre a pour but de présenter un état de l'art du domaine de la thèse, afin de fournir au lecteur toutes les bases nécessaires pour appréhender le travail qui sera détaillé dans les chapitres suivants. Notamment, les termes techniques essentiels vont être introduits. Compte tenu du large spectre concerné, ce chapitre va être divisé en plusieurs parties en apparence disjointes. Les interactions entre ces notions, permettant de construire un ensemble cohérent, deviendront apparentes et seront explicitées dans le chapitre 2.

Le paragraphe 1.1 introduit tout d'abord le contexte général de l'étude, à savoir les systèmes embarqués, en donnant une vue rapide de leur évolution historique et de leur importance dans le monde contemporain. Les paragraphes 1.2 et 1.3 sont ensuite consacrés aux entités les plus importantes sur lesquelles on va être amené à travailler, c'est-à-dire respectivement les processeurs et les systèmes d'exploitation embarqués. Le paragraphe 1.4 a pour rôle de présenter le sujet clé du travail, c'est à dire le concept de sûreté de fonctionnement, avec son cadre théorique, son historique et les techniques principales mises en œuvre pour satisfaire ce type de contraintes. Enfin, le paragraphe 1.5 introduit le concept de sécurité de l'information, en expliquant ses principes et notamment les liens avec les autres sujets de base introduits dans ce chapitre.

Un grand nombre de termes techniques va être introduit, en général avec leur équivalent anglais s'il n'est pas trivial. Ce choix est dû au fait que la discipline est relativement jeune (une soixantaine d'années) et la terminologie de référence est essentiellement internationale, donc en anglais. Les terminologies utilisées dans les autres langues (y compris le français) sont essentiellement des traductions, parfois ambiguës (plusieurs notions distinctes pouvant avoir une même traduction) et souvent très variables suivant l'auteur et son origine scientifique. La terminologie présentée dans ce chapitre permettra de bien préciser les aspects étudiés dans les chapitres suivants.

1.1 Systèmes embarqués

Les ordinateurs et, plus généralement, les systèmes électroniques et informatiques sont désormais devenus omniprésents dans la vie de tous les jours. Téléphones portables, baladeurs, PDAs, cartes de paiement, il est presque impossible de tourner la tête sans voir au moins une application. Ceci est sans parler des applications moins « voyantes » mais pas moins répandues : conduite assistée pour voitures, monitorage du trafic, contrôles d'accès, systèmes de sécurité, etc. … Pour atteindre un tel résultat, l'électronique a dû progressivement s'adapter pour devenir portable, puis ultra-miniaturisée : c'est la naissance des systèmes embarqués.

Taille, poids et consommation sont les facteurs clés d'un système qui a la caractéristique d'être portable. Embarqué dans un équipement (voiture, avion, etc. …) il peut interagir avec son environnement et effectuer un grand nombre de tâches. Une des plus anciennes et connues est sans doute le pilotage automatique d'avions et de fusées : développés dès le début du vingtième siècle et utilisés dans les Deux Guerres, les systèmes de pilotage automatique ont évolué à partir de systèmes hydromécaniques capables seulement de garder une route à hauteur constante vers des systèmes capables d'aider le pilote dans toutes les phases cruciales du vol, décollage et atterrissage compris. Cette évolution fut possible grâce à l'évolution des systèmes électroniques embarqués dans l'avion.

Au début, les applications des systèmes embarqués étaient surtout dans le milieu militaire à cause d'un ensemble de facteurs, notamment leur prix et l'avantage stratégique

d'une telle technologie. Cependant, leur grande potentialité leur a rapidement ouvert les portes vers des applications civiles (i.e. pilotes automatiques pour avions de ligne, etc. …). De plus, le progrès technologique a constamment baissé leur prix et augmenté leurs possibilités ; aujourd'hui, les systèmes embarqués intégrés trouvent des applications partout où taille, poids et consommation sont des facteurs clés. Un bon exemple est l'évolution des téléphones mobiles, depuis les « valises » des années 80 jusqu'aux minuscules modèles actuels.

Un système embarqué est composé typiquement de matériel exécutant du logiciel, ce qui implique un ou plusieurs processeurs, des mémoires et éventuellement des circuits dédiés à des applications spécifiques (appelés en général ASIC, « Application Specific Integrated Circuits »). Ces dernières années, la capacité des circuits intégrés a augmenté considérablement : les systèmes qui autrefois étaient composés de plusieurs puces reliées entre elles sur un circuit imprimé peuvent maintenant être directement implantés dans un seul composant. Les systèmes embarqués deviennent donc souvent des Systèmes sur Puce (en anglais SoC, System on Chip). Tous les avantages des systèmes embarqués (vitesse, taille, consommation, etc. …) sont augmentés, mais aussi la complexité du développement. En pratique, même si un certain nombre de leurs caractéristiques peuvent être différentes, surtout du point de vue technologique, systèmes intégrés et systèmes embarqués tendent souvent vers des architectures semblables basées sur les mêmes éléments (processeurs, mémoires, logique dédiée, etc. …). Pour cette raison, une grande partie des propositions faites dans ce document est également applicable dans le contexte des SoCs.

1.2 Architecture de processeurs : principes

Dans la section précédente, on a introduit le concept de système embarqué, en indiquant que le cœur d'un tel système est composé le plus souvent par un ou plusieurs processeurs. Les principes de base du fonctionnement d'un ordinateur moderne sont bien connus, mais il est quand même utile de rappeler les points principaux, qui seront utiles pour comprendre les propositions faites dans les chapitres suivants.

La première et plus simple définition d'un processeur est celle de la machine de Turing, présentée pour la première fois en 1936 et adaptée plusieurs fois les années suivantes. La formalisation présentée ici, qui a la remarquable qualité d'être extrêmement claire, est tirée de [Pinke95]. La machine illustrée en Figure 1-1 est composée d'une tête qui peut lire et écrire des mots dans une mémoire, représentée par un ruban. La machine lit la valeur contenue à la case active et, suivant cette valeur et son état interne, peut déplacer la tête vers une autre case, écrire une nouvelle valeur ou modifier son état interne.

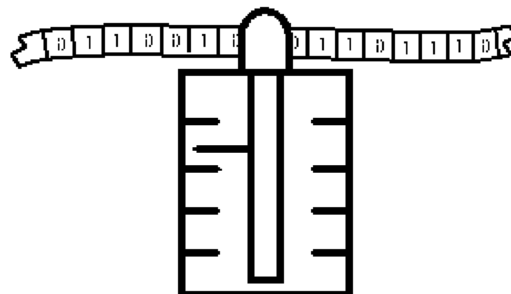


Figure 1-1 Machine de Turing

L'intérêt de la machine de Turing est que n'importe quel algorithme peut être formalisé et exécuté avec elle, et tous les processeurs peuvent être représentés par une machine de Turing équivalente.

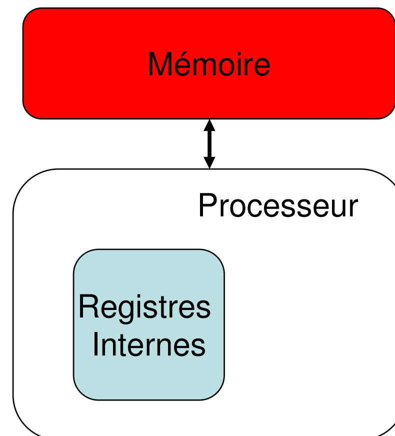


Figure 1-2 Schéma de base d'un processeur

Dans les processeurs modernes (Figure 1-2) la mémoire à ruban est remplacée par les mémoires RAM ou ROM et l'état interne est mémorisé par les registres internes. Au lieu de lire un bit à fois, les processeurs lisent des mots de taille donnée (typiquement 8,16 ou 32 bits) qui indiquent l'opération à faire. L'ensemble des opérations réalisables est appelé **jeu d'instructions** et définit totalement les possibilités du processeur. Ici, la pratique commence à différer de la théorie. Quand on implante un tel processeur, on peut remarquer trois points importants :

1) Souvent, les instructions ne prennent pas toutes le même temps pour s'exécuter ;
2) Les accès mémoire peuvent être extrêmement longs ;
3) Toutes les ressources du processeur ne sont pas utilisées en même temps.

Ces considérations sont à la base des deux principes les plus importants utilisés dans l'architecture des processeurs modernes : le pipeline et les mémoires caches. Le pipeline, introduit en section 1.2.1, répond aux observations 1) et 3). Les mémoires caches, introduites en section 1.2.2, répondent à l'observation 2). Pour cette présentation, nous nous baserons sur les exemples présentés dans l'un des principaux livres de base du domaine [Henne96].

1.2.1 Pipeline

Si on analyse l'exécution d'une instruction quelconque on peut normalement repérer 5 phases principales :

1) avant tout, l'instruction est lue dans la mémoire : c'est ce que l'on appelle la phase extraction ou **fetch** ;
2) l'instruction lue doit ensuite être interprétée : c'est la phase de décodage ;
3) les opérations requises doivent être commandées aux éléments opératifs : c'est la phase proprement dite de l'**exécution** ;
4) il peut y avoir besoin de lire ou écrire des données en mémoire: c'est la phase des **accès mémoire** ;
5) si une opération a été réalisée il faut mémoriser son résultat dans un registre : c'est la phase d'**écriture.**

Si on conçoit soigneusement l'architecture, on peut facilement faire exécuter chacune de ces tâches par du matériel séparé, comme on peut le voir dans l'exemple de la Figure 1-3. Il faut noter que toutes les figures de cette section font référence à un processeur appelé DLX, qui correspond à un modèle didactique simplifié, permettant d'illustrer de façon claire toutes les problématiques d'un processeur réel. Les notions présentées sous cette forme seront utilisées dans les chapitres suivants sur des processeurs réels.

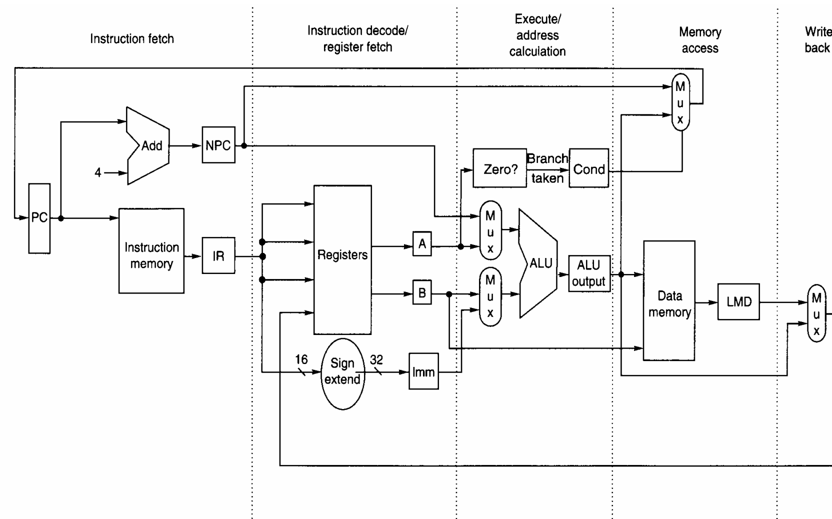


Figure 1-3 Pipeline du processeur DLX. On peut remarquer comment chaque étage correspond à de la logique indépendante [Henne96]

La logique des étages étant séparée, il suffit de mettre les étages ainsi construits en série pour obtenir l'exécution complète de l'instruction. Si on insère des registres entre chaque étage, l'exécution va se diviser en plusieurs cycles d'horloge, permettant donc d'atteindre des fréquences beaucoup plus élevées. Chaque instruction va entrer dans la série d'étages, passer par chacun et enfin sortir terminée, un peu comme de l'eau qui entre dans une conduite et passe dans chaque segment avec de sortir : c'est la raison pour laquelle une telle structure est appelée pipeline (tuyauterie en anglais). En plus, quand un étage/segment est libéré d'une instruction I, il peut directement prendre en compte la suivante, sans devoir attendre que les étages suivants aient fini de traiter l'instruction I. Quand le « tuyau » est plein on a donc, pour l'exemple de découpage présenté, 5 instructions qui sont exécutées en parallèle et une instruction qui se termine à chaque cycle.

L'unité qui contient le pipeline est généralement appelée Unité Centrale de Calcul, ou CPU de l'anglais Central Processing Unit. Tous les processeurs à hautes performances sont basés sur une structure de pipeline, complétée par différentes améliorations architecturales. Une fois encore [Henne96] en décrit un ensemble bien représentatif, mais il faut souligner que aujourd'hui encore la bonne gestion du pipeline est le point clé pour l'optimisation des performances.

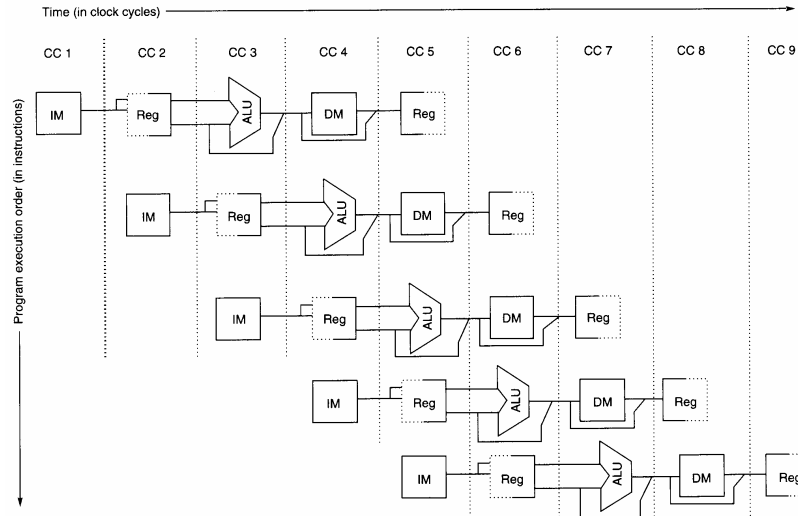


Figure 1-4 Remplissage du pipeline du processeur DLX. Noter au cycle 5 le moment idéal quand chaque étage est en train d'élaborer une instruction différente [Henne96]

1.2.2 Caches

Un des principaux compléments apportés à une architecture pipeline est l'utilisation de plusieurs niveaux de mémoire. Cela est dû au fait que les temps d'accès aux mémoires de grande taille est notablement supérieur au temps de cycle atteint par les processeurs dans les technologie modernes et donc les opérations de lecture/écriture sont devenues les vrais pierres d'achoppement des architecture modernes. Le coût des mémoires embarquées à haute vitesse a baissé, mais pas assez pour pouvoir les utiliser comme seule mémoire centrale, sans compter les contraintes technologiques et notamment la consommation. L'idée de base est donc d'interposer entre le CPU et la Mémoire Centrale (MC), grande mais lente, une mémoire plus petite mais nettement plus rapide, appelée mémoire cache. Le résultat est une hiérarchie de mémoires qui deviennent plus grandes et plus lentes au fur et à mesure que l'on s'éloigne du CPU, comme illustré dans la Figure 1-5.

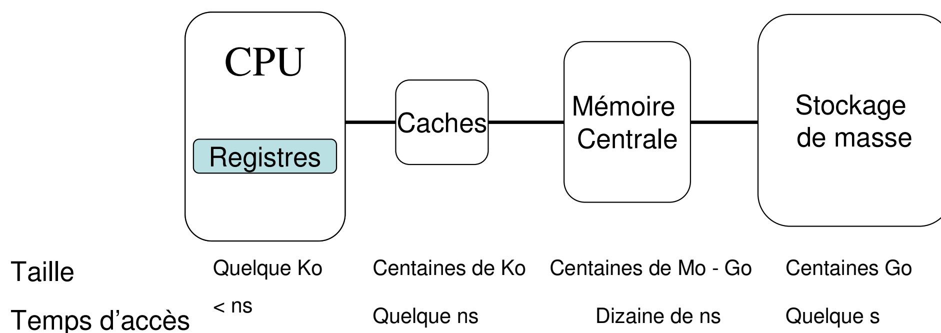


Figure 1-5 Cas typique de hiérarchie mémoire, avec ordres de grandeur des tailles et des vitesses

Comme son nom l'indique (en anglais, cache = cachette), une mémoire cache contient la copie des instructions que le CPU doit exécuter et/ou les données sur lesquelles le processeur travaille. De cette façon, le pipeline interagira idéalement toujours avec une mémoire rapide et les accès à la mémoire plus lente seront faits par des unités indépendantes. Cependant, la mémoire cache est plus petite que la MC et pourra donc en copier seulement une petite partie. Il est donc vital de bien la remplir en choisissant les bonnes plages

d'adresses à copier. Si le processeur ne trouve pas la donnée ou l'instruction qu'il cherche dans le cache, il est forcé à attendre son transfert depuis la MC, avec la pénalité en vitesse qui en résulte : on appelle cela un « cache miss ».

Les besoins étant différents pour les instructions et les données, on a le plus souvent deux mémoires caches séparées, correspondant à un système processeur-mémoire avec une architecture externe du type Von Neumann mais une architecture du type Harvard entre le processeur et le premier niveau de cache. A noter que les caches instructions vont être en lecture seule pour le CPU, mais que les caches données sont aussi en écriture (pour stocker les résultats). Cela implique des problèmes de cohérence entre cache et MC, qui sont résolus soit en écrivant directement la donnée modifiée en MC (politique appelée write-through, « écrire à travers » parce que la donnée passe à travers le cache) soit en marquant la donnée modifiée dans le cache et en ne l'écrivant en MC que lorsque le bloc correspondant du cache est remplacé (politique appelée write-back car c'est une opération qui est retardée).

1.3 Systèmes d'Exploitation Embarqués

Quand on parle de systèmes complexes à base de processeurs la clé pour obtenir de bonnes performances et une bonne flexibilité est le système d'exploitation (OS de l'anglais Operating System). Cependant les systèmes embarqués ont des caractéristiques propres à eux qui ont des répercussions sur l'OS, les plus importantes étant :

- les ressources (mémoire, puissance de calcul, etc. …) sont typiquement réduites ;
- les contraintes temporelles peuvent être importantes ;

C'est la raison pour laquelle on parle en général de Systèmes d'Exploitation Embarqués (EOS de l'anglais Embedded OS). La grande variété des solutions embarquées existantes a bien sûr conduit à un grand nombre de EOS, chacun adapté pour mieux exploiter certaines des ressources disponibles. La première conséquence a bien sûr été une presque totale incompatibilité entre les différents systèmes, parfois même entre solutions proposées par un même fournisseur. Cela entraînait des coûts énormes de développement pour le logiciel, qui devait être presque complètement ré-écrit et re-validé à chaque fois. Heureusement, le progrès technologique a permis d'augmenter significativement les capacités des systèmes embarqués, en atteignant des niveaux de complexité et de puissance de calcul comparables à ceux des ordinateurs d'il y a dix ou vingt ans. Le choix de sacrifier un peu de ressources pour augmenter la compatibilité a donc été naturel, avec de grands avantages pour les temps de développement. Dans un premier temps, chaque fournisseur tendait à développer son propre EOS, mais la difficulté d'une telle tâche ainsi que les limitations implicites sur la compatibilité ont engendré une volonté de standardisation, ainsi qu'une grande attention vers les logiciels libres. On est encore bien loin d'une uniformisation totale, mais cependant deux tendances intéressantes peuvent être observées :

- le développement de systèmes d'exploitations conçus pour la portabilité, qui sont donc très faciles à adapter à n'importe quelle machine grâce à des couches d'abstraction ajoutées au moment de la conception de l'OS. C'est le cas, par exemple, de eCos de Red Hat, détaillé dans le paragraphe 2.2.3.
- l'attention croissante vers les versions embarquées de Linux, à la suite de son succès dans les serveurs, les ordinateurs personnels et les services Web en général, ainsi que pour l'excellente qualité et quantité du logiciel disponible. Plusieurs distributions sont

maintenant disponibles, notamment le classique µC-Linux (pour machines sans unité de gestion de la mémoire ou MMU) [Micro05] ou le Snapgear [Snapg04].

L'évolution vers des systèmes d'exploitation « complets » permet de mettre en évidence des mécanismes de base communs qui offrent des possibilités d'intervention très intéressantes : si on se concentre sur ces caractéristiques communes, les modifications seront facilement adaptables à plusieurs OS.

L'exemple le plus important est sans doute le **changement de contexte**, qui permet d'exécuter plusieurs programmes sur la même machine en parallélisme simulé, en réalisant ce que l'on appelle un système multitâche. Le principe est très simple : chaque programme, généralement appelé tâche, est enrobé dans une « coquille » qui lui permet de s'exécuter de façon indépendante. Cette « coquille », normalement appelée processus, est en effet une série de ressources (notamment espace mémoire) que le système d'exploitation lui réserve de manière exclusive. Cette astuce permet de suspendre à presque chaque moment l'exécution de la tâche T1 et de démarrer une autre tâche T2. T2 utilisant des ressources différentes (sa propre « coquille »), T1 ne va pas être gênée et pourra être redémarrée à un autre moment. Toutes les informations qui permettent d'identifier et d'exécuter une tâche donnée sont appelées « contexte » et varient beaucoup suivant le processeur et le système d'exploitation, mais dans la plupart des cas le contexte est composé principalement par les registres d'état du processeur (compteur de programme, pointeur de pile, etc. …) et par quelques variables de contrôle de l'OS. De cette façon, le processeur va exécuter en série toutes les tâches, avec entre elles un changement de contexte, comme illustré en Figure 1-6. Il est très important de souligner les deux grandeurs qui vont être déterminantes pour les performances du système :

T_e : le temps que chaque tâche a à sa disposition pour s'exécuter, usuellement appelé avec le terme anglais « timeslice » ;

T_c : le temps nécessaire pour effectuer un changement de contexte, c'est-à-dire pour sauvegarder les informations sensibles de la tâche en phase d'exécution afin de pouvoir la suspendre et pour restaurer celles de la tâche suivante.

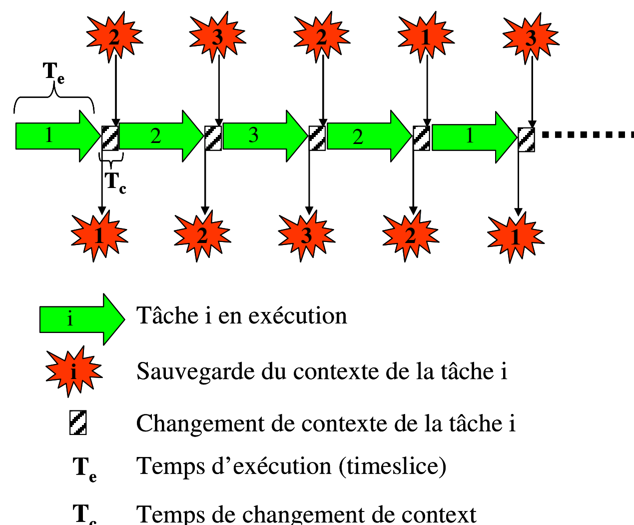


Figure 1-6 Flot d'exécution global typique d'un système multitâche

Il est facile de comprendre que T_c est du temps mort du point de vue des performances, car le processeur est occupé par des opérations qui ne sont pas liées à sa charge de travail. La somme des temps T_c doit donc être en général le plus petit possible. Par ailleurs, si T_e est trop long, les tâches suspendues risquent de devoir trop attendre avant de s'exécuter :

du point de vue d'une tâche donnée, l'exécution va être fragmentée entre phases d'exécution effective et phases d'attente comme illustré dans la Figure 1-7, le temps de complétion étant la somme des temps d'exécution, ΣT_{ei} et des temps d'attente, ΣT_{si}, c'est-à-dire $\Sigma(T_{ei}+T_{si})$.

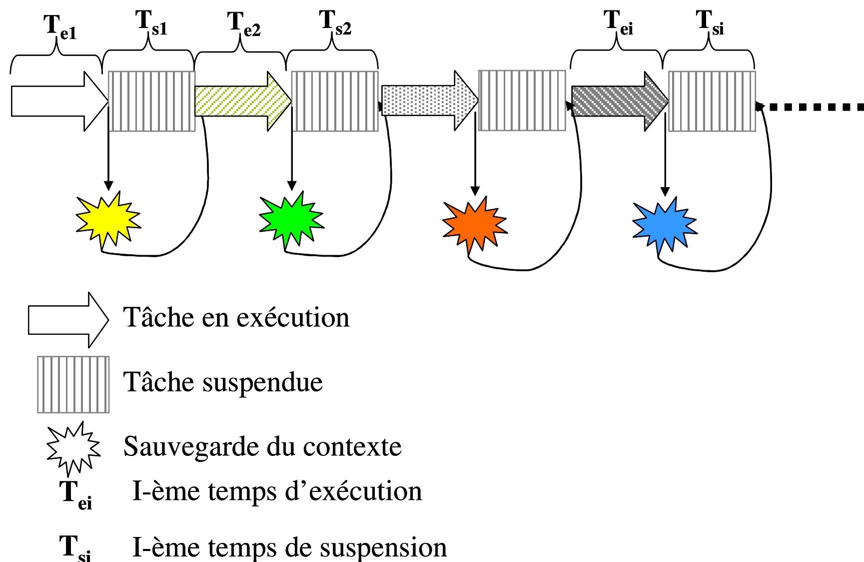


Figure 1-7 Flot d'exécution d'une tâche dans un système multitâche

Pour faire un lien avec la Figure 1-6 il faut noter que le temps de suspension T_{si} est le temps entre deux exécutions successives de la même tâche, pas forcément constant. Par exemple la tâche 2 doit attendre l'exécution de la tâche 3 : $T_{s2}= T_c+ T_{e3}+ T_c$. Pour la tâche 1 l'attente est bien plus longue : $T_{s1}= T_c+T_{e2}+T_c+ T_{e3}+ T_c+T_{e2}+T_c$. Une tâche a donc besoin de plus de temps pour se compléter que si elle avait pleine possession du processeur et ce surcoût dépend directement de l'ordre dans lequel les tâches ont accès au processeur. Le module qui gère le changement de contexte et décide quelle tâche doit être exécutée est appelé ordonnanceur (ou plus couramment « **scheduler** » en anglais) et sa mise au point est un des facteurs les plus importants et difficiles déterminant la qualité et les performances d'un système d'exploitation. Une discussion détaillée de ce sujet sort du cadre de ce document, mais la référence [Tanne87] présente une introduction de qualité et très exhaustive.

1.4 Sûreté de fonctionnement

La sûreté de fonctionnement est un thème qui est présent depuis le tout début de l'histoire de l'électronique, même si elle reste méconnue par le grand public ; un rapide historique de la discipline est présenté dans le paragraphe 1.4.1. ainsi que la terminologie utilisée dans le domaine. Le paragraphe 1.4.2 introduit ensuite les modèles de fautes considérés, puis les techniques et applications principales sont détaillées dans les paragraphes 1.4.3, 1.4.4, 1.4.5, chacun consacré à l'un des trois niveaux dont un système embarqué est composé : matériel, système d'exploitation et logiciel respectivement.

1.4.1 Historique et terminologie

Von Neumann (le « père » des ordinateurs) et Shannon parlaient déjà de sûreté de fonctionnement en 1956 dans leurs articles [Neuman52] et [Moore56]. L'attention était surtout focalisée sur les défauts de production d'une technologie à son stade embryonnaire et donc imparfaitement maîtrisée, ainsi que sur les effets du vieillissement des dispositifs. Dans

les années suivantes, une branche spécifique s'est développée pour étudier et identifier ces phénomènes : on l'appelle généralement « test » (« testing ») et elle se focalise sur la vérification de la bonne santé d'un circuit envers ces défauts qui sont modélisés comme des « fautes **permanentes** ». Cependant, plus récemment, une nouvelle catégorie de fautes a retenue l'attention générale : les fautes **transitoires**, qui vont être détaillées dans le paragraphe 1.4.2.

Avant de rentrer dans les détails, il est préférable d'expliciter la terminologie partagée par tous les acteurs du domaine. Pour la traduction française, on se basera sur la formalisation donnée par Laprie dans [Lapri88] et étendue ces dernières années.

Un problème physique dans un composant est appelé un **défaut**. Ces défauts peuvent être modélisés sur la base de leur conséquence au niveau de la structure du circuit, par exemple sur la base de leur effet logique. Cette modélisation de plus haut niveau correspond à une **faute** dans le système. Cette faute reste **latente** tant que cette partie du composant est inutilisée et elle est **activée** lorsques certaines opérations sont à réaliser, en générant une **erreur**, c'est-à-dire une déviation de l'état du système par rapport à l'état nominal. Cette erreur peut affecter le comportement d'autres composants dont le fonctionnement dépend des résultats du composant fautif et donc **contaminer** le système. Si la contamination conduit à faire dévier le comportement externe du système d'une manière inacceptable du point de vue de l'application, on a une **défaillance** (**failure**). La Figure 1-8 explicite cette succession.

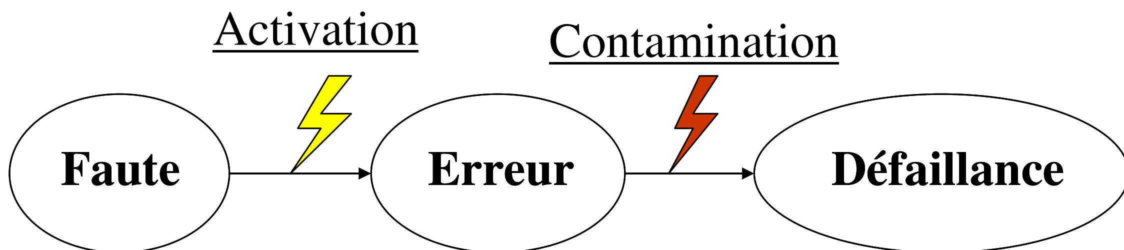


Figure 1-8 Schéma de la succession faute-erreur-défaillance

Du point de vue comportemental, la définition est récursive : une défaillance est l'impossibilité de « délivrer le service demandé par les spécifications », une erreur est un « état susceptible de générer une défaillance » et la faute est la « cause de l'erreur ». Ceci peut être appliqué à différents niveau d'abstraction (système complet, équipement, composant, …). Cette vision descendante est intéressante parce qu'elle permet de trouver les parties à protéger des fautes à partir des spécifications de haut niveau du comportement du système, ainsi que de comprendre quelles erreurs doivent être complètement évitées/corrigées et lesquelles peuvent être jugées acceptables (parce qu'elles ne vont pas engendrer de défaillances).

Certaines fautes peuvent être **évitées** ou **éliminées** au niveau de la conception, mais certaines autres (notamment celles définies dans le paragraphe suivant) doivent être prises en charge au moment de l'exécution. Les choix sont multiples : on peut se contenter de les **détecter**, on peut chercher à les **masquer** pour continuer l'exécution malgré leur présence ou on peut les **tolérer**, par exemple en déclenchant après détection des **opérations de recouvrement** pour corriger l'erreur avant qu'elle ne se propage et qu'elle cause une défaillance. Le choix est fait en fonction du **niveau de sûreté de fonctionnement** (**dependability**) que le système doit assurer, défini par des attributs dont les plus typiques sont :

- **fiabilité** (**reliability**) : continuité du service délivré pendant une durée minimale. Si cette periode est très importante (plusieurs années) on parle alors de systèmes à **longue vie** (**long-life**). Par exemple, le système STAR (voir fin du chapitre) a une probabilité de fonctionnement de 0,90 après 10 ans de service
- **sécurité–innocuité** (**safety**) : la probabilité d'occurrence des défaillances jugées critiques est inférieure à un certain seuil. Si en plus aucune défaillance ne peut causer un événement catastrophique on est dans une situation de **défaillance sûre** (**fail-safe**).
- **sécurité–confidentialité** (**security**) : confidentialité et intégrité des données sont assurées avec une probabilité fixée. Ceci est très différent de la sécurité au sens innocuité, même si le terme sécurité est utilisé en français dans les deux cas.
- **disponibilité** (**availability**) : le pourcentage du temps pendant lequel le système est prêt à l'utilisation est supérieur à un certain seuil. Dans ce cas, l'exactitude des résultats est moins importante que la possibilité d'utiliser le système : certaines défaillances peuvent être acceptées tant qu'elles permettent de garder le système actif.

Il faut souligner que plusieurs attributs peuvent être simultanément pertinents pour un système donné.

A titre d'exemples concrets, un ordinateur doit généralement être fiable et/ou disponible (il ne doit pas être toujours en panne), et s'il est utilisé pour travailler il devra probablement être aussi sécurisé au sens de la confidentialité (pour protéger les données manipulées). Le système de contrôle d'un satellite doit être fiable (pour être sûr de ne pas perdre le contrôle) et le système de pilotage automatique d'un avion doit être aussi à défaillance sûre car le pilote va lui confier la vie des passagers et de l'équipage. Un exemple typique de système à haute disponibilité est un central téléphonique : il doit être toujours prêt quand quelqu'un en a besoin.

Une application est indiquée comme **critique** quand au moins certains types de défaillances ne devraient jamais survenir. En fonction du niveau de criticité et des autres contraintes de l'application, on peut souvent se contenter de savoir si le résultat est correct ou non, sans forcément devoir le corriger : dans ces cas une simple **détection** d'erreur est suffisante. Dans d'autres cas, la correction des résultats est nécessaire pour satisfaire les attributs de sûreté de fonctionnement.

Le livre de Siewiorek ([Siewi82]) est encore maintenant le point de référence pour ce qui concerne les techniques permettant d'assurer la sûreté de fonctionnement. Dès le début, le principe d'intervention a été bien défini : de la redondance (matérielle, temporelle ou d'information) est ajoutée au système à protéger pour pouvoir vérifier l'exactitude du déroulement des opérations et, si possible, le corriger. Pour faire une synthèse extrême, l'évolution des techniques a toujours été de réduire cette redondance tout en gardant un bon degré de protection. En général, un composant est dit « protégé » ou « durci » quand sa sûreté de fonctionnement a été assurée par une intervention ajoutant un certain niveau de redondance (approche architecturale) ou évitant l'occurrence de fautes (approche technologique). Cette thèse se limite au premier type d'intervention.

1.4.2 Fautes et modèles

Nous nous limiterons ici au cas des fautes transitoires. Une faute transitoire est une perturbation du comportement habituel du circuit à la suite d'interférences. Sa caractéristique principale est justement d'être éphémère : une fois la perturbation passée, le circuit peut continuer à fonctionner, aucun dommage permanent n'étant généré. Cependant, des résultats erronés peuvent avoir été générés pendant cet intervalle de temps, dont la gravité peut être très importante. Les causes d'une telle faute peuvent être multiples (radiations, couplage entre lignes de transmission proches, etc. …), mais nous considérerons ici plus particulièrement le cas des **effets singuliers** (non destructeurs) causés par des **impacts de particules**. Etudiés de façon théorique pour la première fois en 1962 par Wallmark et Marcus ([Wallm62]), ils ont été observés pour la première fois en 1975 sur un satellite en vol ([Binde75]). Le principe de base est simple : une particule incidente sur un transistor peut créer une zone d'ionisation qui interfère avec la polarisation normale du dispositif et altère son comportement, comme illustré en Figure 1-9.

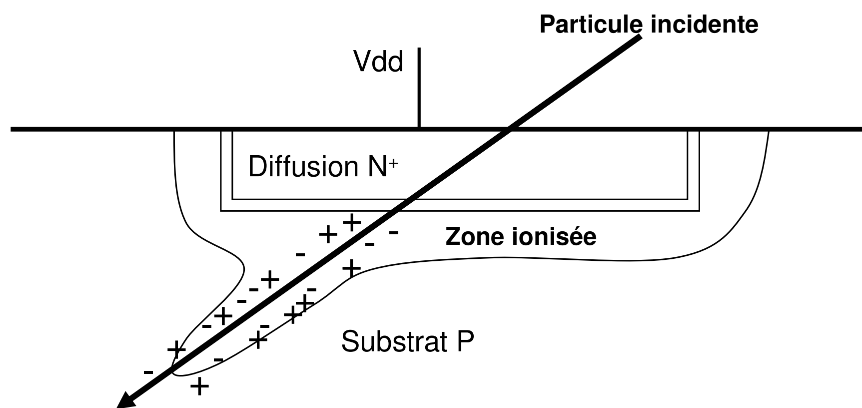


Figure 1-9: Ionisation produite par une particule incidente dans une jonction PN

Beaucoup de travaux ont été effectués sur la modélisation de ces phénomènes pour étudier leur incidence sur les différentes technologies : à partir du susmentionné [Wallm62], en passant par les travaux sur les particules alpha de May et Woods dans [May79], et arrivant jusqu'à la volumineuse littérature contemporaine dans les actes de nombreuses conférences (Radecs, IOLTS, DATE, etc.…).

Tous ces travaux ont traditionnellement été réservés aux milieux dits difficiles, comme l'espace ou les centrales nucléaires, caractérisés par des concentrations de radiations ou de particules très élevés. Cependant, le progrès technologique a porté vers une miniaturisation très poussée des composants et une diminution de la tension d'alimentation ; ces deux aspects ont augmenté considérablement la sensibilité des circuits vis à vis des fautes transitoires. D'un coté, des composants plus petits sont intrinsèquement plus sensibles aux effets singuliers, par réduction de la valeur de capacité sur chaque nœud électrique. D'un autre côté, une tension de travail plus faible contribue à réduire encore la quantité de charges stockée. En conséquence, des effets qui étaient insignifiants sur des technologies anciennes peuvent maintenant engendrer des problèmes notables. Cela signifie que désormais les effets transitoires sont un **problème réel** même en **condition d'utilisation normale** et doivent être pris en compte dans la conception de nombreux circuits, même destinés à fonctionner au niveau de la mer. Pour donner un exemple concret, Sun Microsystems utilise régulièrement des techniques de

protection contre les fautes transitoires dans ses serveurs à haute disponibilité [Parul05], conçus pour être utilisés au niveau terrestre.

Comme pour tous les phénomènes physiques, il est nécessaire de définir une modélisation pour pouvoir en tenir compte dans nos analyses. Cette modélisation sera basée sur la formalisation proposée par [Anghe05]. D'un point de vue électronique, il est possible de définir plusieurs niveaux d'abstraction, le plus bas étant celui de la physique des dispositifs (comme dans la Figure 1-9), et le plus haut celui du comportement général du système cible (interruption de service, résultats erronés, etc. …). Entre ces deux extrêmes, il existe de nombreux modèles pouvant s'adapter au niveau de précision souhaité : impulsions de courant ou de tension (appelées aussi glitch en anglais), inversion de bits, etc. La modélisation au niveau portes est le niveau le plus bas où ces phénomènes physiques passent de leur nature intrinsèquement analogique à une représentation **numérique**. Toutefois, la modélisation au niveau Transfert de Registres (appelée usuellement RTL, acronyme de l'anglais Register Transfer Level) est particulièrement importante pour nous parce que c'est le niveau le plus adapté pour une analyse efficace de l'effet des fautes sur des circuits logiques aussi complexes que ceux visés dans notre étude. En effet, nous nous attacherons par exemple à l'étude du comportement d'un système (processeur et logiciel associé) suite à la modification de la valeur d'un registre par une faute. Dans ce cas, il n'est pas nécessaire de connaître les causes physiques exactes ayant entraîné le changement de valeur.

Le plus souvent, un impact de particule est très localisé. On a donc un effet singulier ou SEE (Single Event Effect), qui peut être de deux types principaux (en faisant abstraction des effets de latchup, dont nous ne parlerons pas ici) :

- une inversion de bit dans la logique séquentielle (registres, mémoires, etc. …), appelée SEU (Single Event Upset) ;
- une inversion de signal dans les parties analogiques ou la logique combinatoire, appelée SET (Single Event Transient).

L'hypothèse d'un effet singulier peut être trop forte dans certaines conditions : une particule peut aussi, par exemple, toucher dans sa course plusieurs transistors associés à plusieurs cellules mémoire. Dans ce cas, on sort du cadre des effets singuliers et on tombe dans les effets multiples tels que les MBU (Multiple Bit Upset). Par ailleurs, il est important de noter que la propagation d'un SET dans de la logique combinatoire peut conduire sous certaines conditions à la mémorisation de plusieurs bits erronés sur les sorties du bloc, cet effet devenant d'autant plus probable avec l'augmentation des fréquences d'horloge. De telles erreurs sont classiquement appelées "soft errors" car le circuit n'est pas endommagé, et qu'un comportement correct peut être rétabli par une ré-initialisation. Que l'origine de l'erreur multiple soit la propagation d'un SET ou un effet multiple direct de type MBU importe en fait peu pour les études au niveau RTL. Nous considérerons donc dans la suite un modèle nommé MBF (Multiple Bit Flip), qui englobe ces différentes causes ainsi que les SEU, comme cas particulier d'une multiplicité égale à 1. Il est important de souligner que les MBF tendent à remplacer les SEE dans la spécification des techniques de protection à implanter dans les circuits.

Pour conclure cette partie, il faut noter que le modèle des MBF peut aussi englober d'autres causes d'erreurs telles que des interférences électromagnétiques ou des phénomènes de bruit internes au circuit, par exemple dus à des couplages. Nous verrons, un peut plus loin dans ce chapitre, que le modèle MBF peut aussi être utilisé pour représenter l'effet d'attaques volontaires sur un circuit.

L'unité de mesure la plus typique liée à la sensibilité aux fautes est le FIT (Failures in Time) qui indique le nombre de défaillances attendues causées par effets singuliers (ou multiples) dans le circuit dans un intervalle de 10^9 heures. Les contraintes de conception sont souvent exprimées par le SER (Soft Error Rate), qui est le taux d'occurrence de telles erreurs par unité de temps, ou le MTBF (Mean Time Between Failures), qui indique le temps moyen entre deux défaillances successives du système.

1.4.3 Protection niveau matériel

Le niveau matériel à été le premier a être pris en compte pour la sûreté de fonctionnement et a donc atteint un degré d'évolution très élevé : comme on le verra dans les paragraphes suivants, la plupart des techniques appliquées à n'importe quel niveau ont été à l'origine développées pour le matériel et ensuite adaptées à d'autres niveaux.

Quand on parle de protection matérielle, il faut bien clarifier le concept de **niveau d'intervention**. Un composant peut être décrit à plusieurs niveaux d'abstraction, et les différentes façons d'intervenir sont résumées ici en niveau d'abstraction croissant :

- modification des structures 3D des transistors ou du processus de fabrication pour rendre le circuit moins sensible aux effets transitoires. Ce sont des techniques qui concernent le technologue plus que le concepteur de circuits ;
- développement de cellules technologiques intrinsèquement résistantes à travers une redondance dans la structure électrique au niveau transistors et/ou une modification des caractéristiques géométriques des transistors (dimensionnement) ;
- protection au niveau portes : fonctions logiques et éléments mémoire de base (bascules) ;
- modification au niveau des primitives de calcul ;
- modification au niveau micro-architectural ;
- modification de l'architecture globale.

Les approches les plus pertinentes pour notre étude sont celles intervenant aux niveaux RTL et portes. Les approches RTL (niveau micro-architecture) seront plus particulièrement considérées pour le développement de notre système dans le chapitre 2.

Le fil conducteur de toutes les techniques qui vont être introduites est la **redondance** : pour assurer les caractéristiques de protection voulues on va ajouter quelque chose au composant. Les approches possibles sont fondamentalement au nombre de trois :

1) redondance **matérielle** : la plus ancienne et la plus directe. Des composants sont implantés plusieurs fois dans le système pour détecter des fonctionnements incohérents et/ou remplacer des composants défaillants ;
2) redondance **temporelle** : à la place d'ajouter des composants, des opérations supplémentaires sont effectuées pour vérifier/corriger les opérations normales ;
3) redondance **d'information** : des données supplémentaires sont ajoutées au système pour améliorer ses propriétés de protection (codage redondant).

1.4.3.1 Redondance matérielle

La première technique à avoir été développée, la **triplication** ou **TMR**, a été proposée par Von Neumann dans [Neuman52] et ses principes de base sont encore très utilisés aujourd'hui. Le composant à protéger est implanté en trois exemplaires qui travaillent en parallèle sur les mêmes données d'entrée. Les trois résultats sont ensuite comparés et la sortie est décidée par un vote majoritaire. La Figure 1-10 montre le schéma typique de la triplication. C'est une technique de **tolérance par masquage** car la sortie est correcte même si il y a une erreur dans une des répliques et donc de l'extérieur on ne peut pas savoir si une erreur est effectivement survenue ou non.

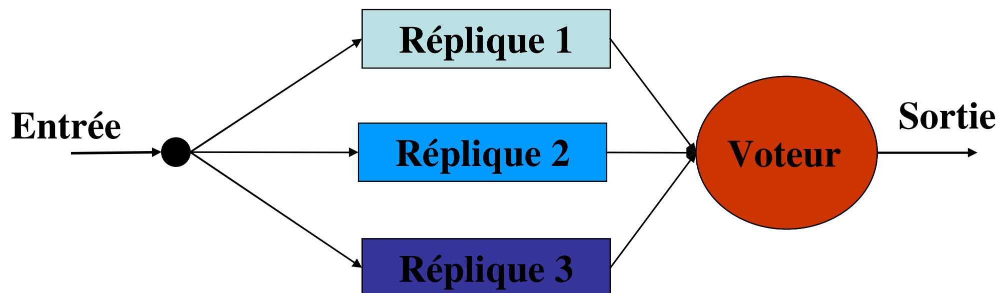


Figure 1-10 Triplication du type Von Neumann

Une grande simplicité d'implantation et d'excellentes propriétés de protection ont fait la popularité de cette technique. Si par exemple chaque réplique a une fiabilité F=0,90 (i.e. la probabilité que le composant fonctionne correctement est 90%) le système global va avoir, en supposant le voteur toujours fonctionnel, une fiabilité F_3=0,972 (il faut qu'au moins deux répliques soient défaillantes en même temps pour entraîner une défaillance du système : $F_3=1-(P_{défaillance})=1-(P_{3défaillantes}+3\ P_{2défaillantes})=1-[(1-F)^3+3F(1-F)^2]$), d'où un gain de presque un ordre de grandeur. Le prix est un surcoût très élevé, avec trois fois plus de composants, sans compter les ressources demandées pour le voteur. La principale faiblesse vient du voteur lui-même, qui devient un élément très critique : si c'est lui qui vient à être défaillant, la sortie sera toujours incorrecte.

Le TMR est bien adapté pour des hauts niveaux d'abstraction car les modules sont vus comme des « boîtes noires ». Pour cette même raison, cette technique reste applicable à tous les niveaux où on a à faire à des blocs : composants, portes, transistors, etc., tous sont facilement réplicables. Chaque niveau a ses avantages et désavantages : plus l'abstraction est faible, plus on peut avoir une meilleure optimisation mais plus la complexité est grande.

Un exemple très intéressant et récent d'application de triplication au niveau portes logiques est décrit dans [Anghe00] : toutes les bascules sont tripliquées pour les protéger des SEUs, et la logique combinatoire est protégée des SETs grâce à un décalage de l'instant d'échantillonnage entre les trois répliques (il faut noter que ce décalage correspond à une redondance temporelle, qui est associée à la redondance matérielle des bascules). Si un SET survient, il va générer une impulsion de durée limitée sur le signal et si les trois instants d'échantillonnage sont assez espacés le SET va affecter seulement une des trois répliques et va donc être filtré par le voteur au moment de la lecture. La Figure 1-11 montre le schéma de base de réalisation, et la Figure 1-12 montre à travers un chronogramme le filtrage d'un SET.

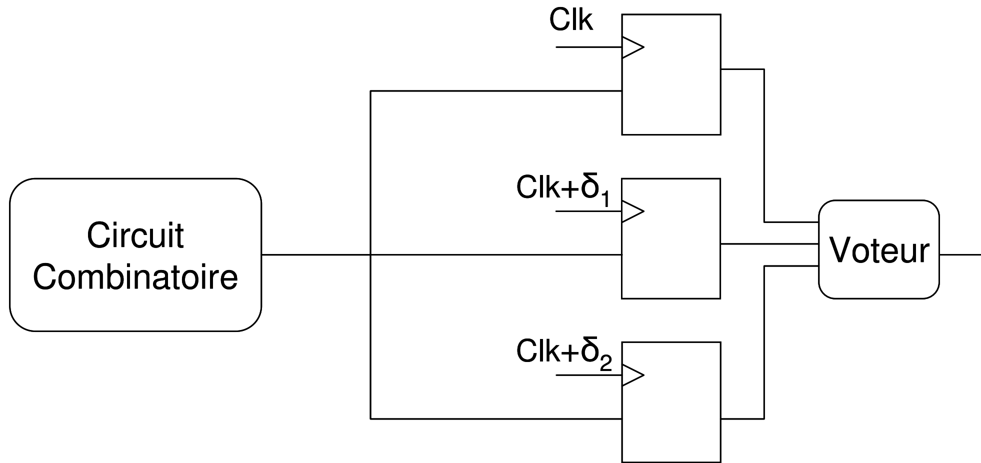


Figure 1-11 Triplication des bascules avec horloge décalée

Les décalages δ_1 et δ_2 sont décidés par le concepteur en fonction de la caractérisation de la technologie utilisée et de l'environnement de travail du circuit, et ne doivent pas forcément être multiples (même si, bien sûr, cela simplifie l'implantation). On peut les réaliser en introduisant des éléments de délai avant d'appliquer l'horloge sur les répliques, ou en développant directement des arbres de distribution d'horloge séparés (même si cette deuxième option est en fait très coûteuse en termes de routage).

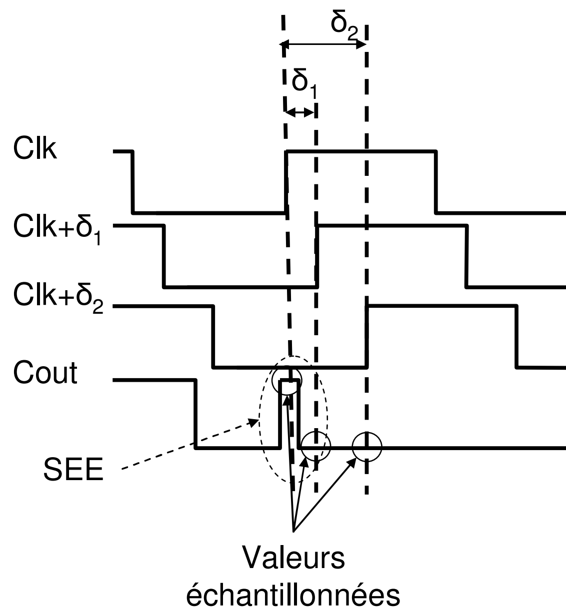


Figure 1-12 Masquage d'un SET grâce au décalage d'horloge

Cette technique a l'avantage d'avoir des coûts très maîtrisés du point de vue matériel (bascules tripliquées + éléments de délai) et fréquentiel (le cycle d'horloge doit juste être augmenté de δ_2). Elle a été appliquée par exemple par ATMEL dans son dernier processeur pour applications spatiales, le AT697E [Atmel06]. Cependant, elle demande de router le signal d'horloge avec une précision redoutable pour assurer toujours les bons délais à chaque point mémoire. Cette tâche peut être très lourde pour des circuits de complexité importante tels que le AT697E, et peut entraîner des retards notables à la mise sur le marché.

Développée pour une description au niveau portes, cette technique pourrait théoriquement être généralisée pour des niveaux d'abstraction plus élevés, où par exemple à

la place des bascules il y aurait des bancs de registres et/ou des mémoires. Cependant ses caractéristiques typiquement temporelles rendraient des vérifications à haut niveau très difficiles.

Une extension de la triplication est la réplication m-n modulaire avec réserves et reconfiguration : le système présente m répliques dont seulement n travaillent en parallèle, les autres étant des réserves en attente hors ligne. Au moment où le voteur identifie une réplique défaillante (i.e. sa sortie est différente des autres) il prévient un composant de reconfiguration qui va mettre hors ligne l'élément fautif et le substituer par une des répliques de réserve.

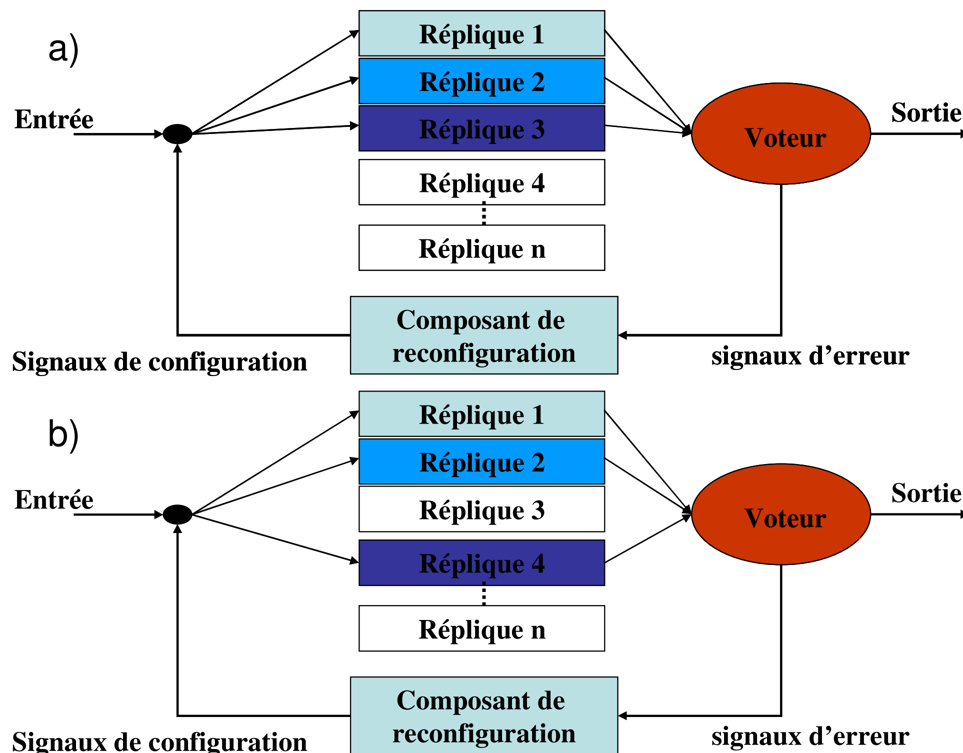


Figure 1-13 Réplication m-n modulaire avec réserves avant (a) et après (b) reconfiguration

La Figure 1-13 représente un système de ce type, avec un exemple de reconfiguration : les répliques en ligne sont colorées, celles hors-ligne sont blanches.

L'avantage d'une telle approche est de pouvoir mettre hors-ligne les composants supposés fautifs pour pouvoir essayer de les réparer ou de les substituer, et ainsi éviter l'accumulation des fautes conduisant plusieurs répliques à donner des résultats faux. Mais le surcoût est encore plus élevé que pour la triplication « Von Neumann » et les faiblesses viennent toujours des éléments sensibles non répliqués : le voteur et le composant de configuration.

En ce qui concerne ses niveaux d'application la réplication m-n modulaire a bien sûr les mêmes propriétés que la triplication, même si elle est en générale mieux adaptée au niveau de blocs architecturaux. Elle est excellente, par exemple, pour obtenir une architecture à haute sûreté à partir de blocs de base non protégés ou, en utilisant une terminologie récente, « composants sortis de l'étagère » (COTS, « Commercial Off-The-Shelf »). Ce type de réalisation permet de diminuer de façon très significative les coûts de développement en s'appuyant sur des composants disponibles.

Une autre approche très utilisée est la **duplication** : les répliques sont seulement au nombre de deux et un comparateur génère un signal d'erreur si les deux sorties sont différentes. Cette technique, illustrée dans la Figure 1-14, permet seulement la **détection**, mais a l'avantage d'être plus simple et moins gourmande que la triplication et peut être plus que suffisante dans certaines applications. Il faut aussi souligner que la détection peut être utilisée pour déclencher des actions de recouvrement. Une fois encore le comparateur est un élément sensible car il n'est pas protégé lui-même.

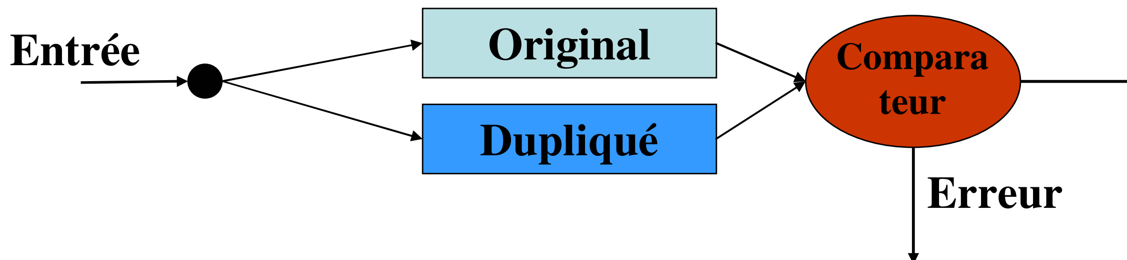


Figure 1-14 Duplication avec détection

Un exemple de système tolérant aux fautes grâce à la duplication avec comparaison est le processeur UNIVAC, où le bus de données était dupliqué, ainsi que tous les sous-processeurs : chaque cycle, les résultats sur les deux bus (celui de l'original et celui de la réplique) étaient comparés et en cas d'inégalité les opérations étaient bloquées et une routine de recouvrement était déclenchée.

Un cas particulier de duplication est la logique dite **Dual Rail** : ici le composant dupliqué est en fait réalisé en **logique duale** par rapport à l'original, sa sortie est l'inverse de la sortie de l'original et il y aura erreur si elles ne sont pas différentes. Le résultat est une meilleure protection contre les fautes réelles (surtout pour les fautes transitoires, voir 1.4.2) parce que en étant physiquement différents l'original et la réplique vont répondre de façon différente aux perturbations et donc il y a une probabilité plus faible que la même interférence cause la même erreur dans les deux composants, faussant la détection.

Une application très intéressante est le développement de composants **auto-contrôlables** (self-checking) : ces circuits sont capable de veiller sur eux mêmes et de prévenir en cas d'erreur interne, comme par exemple le circuit de la Figure 1-15.

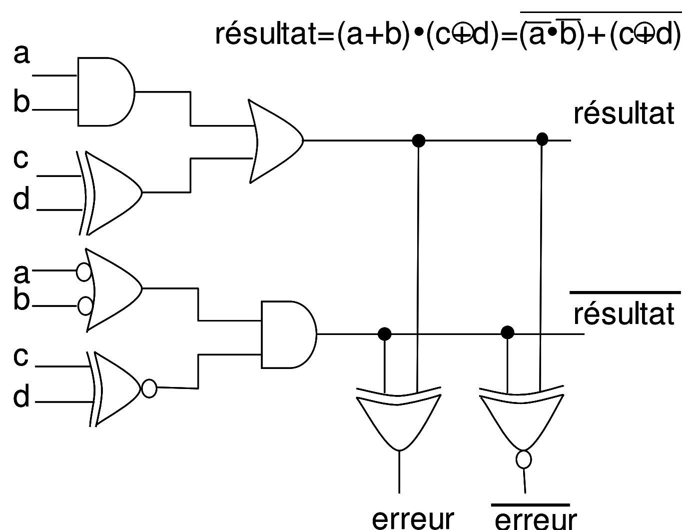


Figure 1-15 Exemple de circuit auto-contrôlable implanté en logique Dual Rail

Souvent les éléments critiques sont donc protégés avec de la duplication et comparaison ou, encore mieux, en **Dual-Rail**. Ces techniques sont très intéressantes notamment pour les éléments les plus sensibles dans les techniques de redondance précédemment introduites : voteurs, comparateurs etc.… C'est une solution plus efficace que de protéger ces éléments de la même façon que les composants d'origine, parce qu'elle évite de trop augmenter la complexité du circuit, sans risquer de tomber dans un cercle vicieux où pour chaque protection on ajoute de nouveaux points faibles à protéger. L'introduction de composants auto-contrôlables permet d'arrêter cette récursivité en donnant un point précis où l'erreur est détectée. Mais cela s'appuie sur un modèle de fautes précis, qui doit être parfaitement représentatif des fautes réelles pouvant survenir.

1.4.3.2 Redondance temporelle

Les techniques de redondance temporelle ont comme but principal de réduire le surcoût matériel, au prix d'une perte de performances. Typiquement ces techniques sont dérivées des approches matérielles, où à la place d'ajouter des répliques on ajoute des cycles de calcul. La Figure 1-16 montre le schéma de réalisation typique d'une triplication par redondance temporelle.

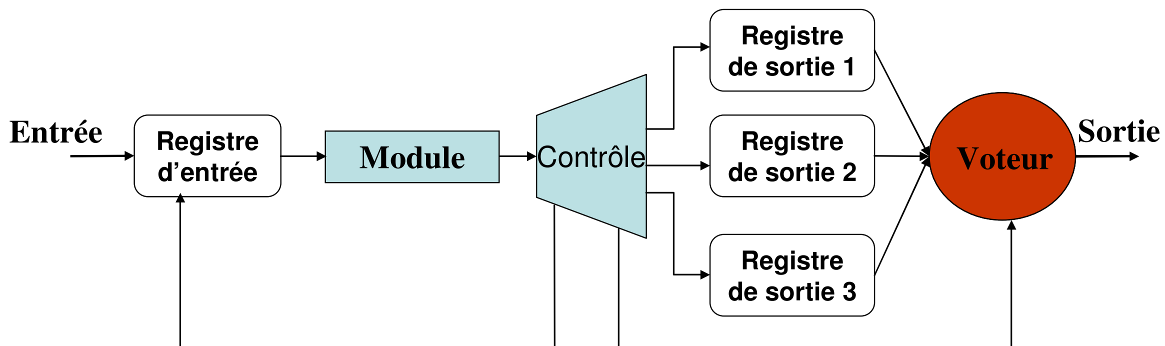


Figure 1-16 Exemple de triplication par redondance temporelle

De la même façon on peut facilement obtenir de la duplication avec détection, comme illustré dans la Figure 1-17.

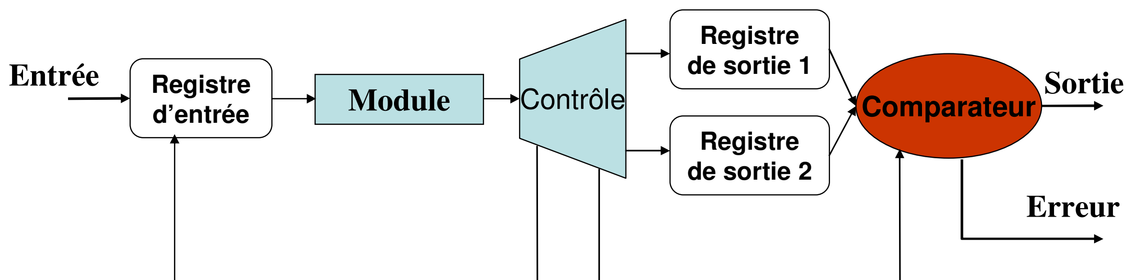


Figure 1-17 Duplication et détection par redondance temporelle

Si les opérations à effectuer sont inversibles il est possible d'effectuer un type particulier de redondance temporelle : après avoir effectué le calcul « direct » on effectue son inverse. Si le résultat ainsi obtenu n'est pas équivalent aux données en entrée alors une erreur est survenue. La Figure 1-18 montre une réalisation possible de ce schéma, où pour simplifier les signaux de contrôle du multiplexeur et des registres ne sont pas indiqués.

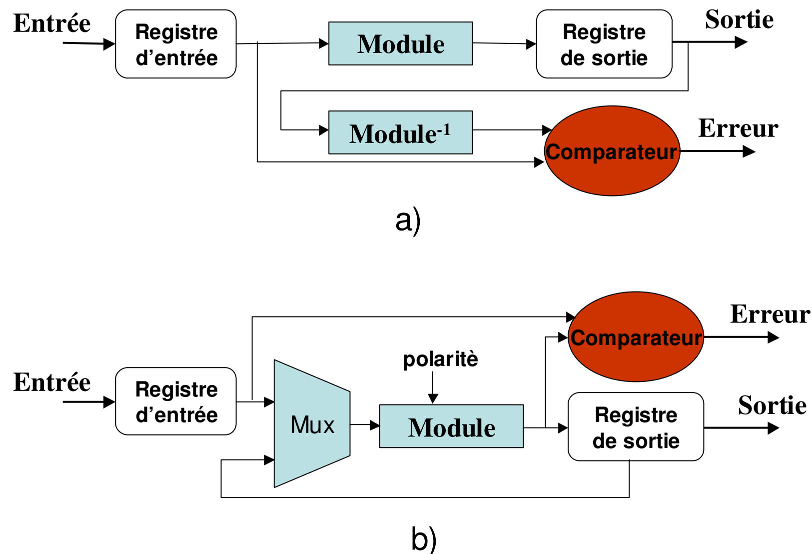


Figure 1-18 Détection par calcul inverse avec module inverse (a) et en redondance temporelle pure (b)

L'intérêt de la redondance temporelle est que souvent les systèmes sont beaucoup plus rapides que l'environnement, donc on peut sacrifier une certaine partie de cet « excès de performances » pour mieux satisfaire les contraintes de protection. En plus dans les systèmes complexes il y a souvent des points de synchronisation où les éléments plus rapides attendent le résultat des composants plus lents avant de continuer : ces pauses sont exploitables pour de la redondance temporelle.

On peut citer également pour mémoire l'existence de techniques de redondance temporelle utilisant des opérandes décalés ou échangés (RESO, RESWO, etc.), techniques particulièrement adaptées aux éléments de calcul arithmétique.

1.4.3.3 Redondance d'information

La redondance d'information a été présente dès le début, sous la forme des codes détecteurs/correcteurs. Déjà dans les années '40 Shannon et Hamming ont posé les bases théoriques, pratiques et statistiques de cette discipline, qui n'a pas cessé d'évoluer au cours des années grâce aussi au développement des télécommunications, qui l'utilisent massivement. Le sujet est très complexe et touche des concepts mathématiques souvent complexes : ce n'est pas l'objectif de ce document d'entrer dans les détails, mais plutôt d'en donner un résumé et d'expliquer leur application. Notre approche est celui d'un « utilisateur » qui doit exploiter les techniques, pas celui d'un « développeur » qui doit en créer. On citera ici juste [Chen84] qui avec ses abondantes références peut être un bon point d'entrée dans le domaine.

Tous les codes sont basés sur le même principe : à partir des bits d'information à protéger (que l'on appellera « **mot** ») on va obtenir des **bits de code** à travers un calcul mathématique souvent très complexe, mais correspondant toujours à des opérations entre bits. Au moment du contrôle on va vérifier que les bits de code sont toujours compatibles avec les bits du mot : en cas contraire une erreur est **détectée** et, si le code le permet, **corrigée**. Une fois encore on introduit des composants critiques (codeurs et décodeur), qui sont donc souvent implantés de façon auto-contrôlable (en Dual Rail par exemple).

Le code le plus connu est sans doute le **code de parité** : un bit de code est ajouté au mot de telle façon que le nombre de 1 soit toujours pair ou impair. Cet artifice permet de

détecter toutes les transitions erronées d'un nombre impair de bits, mais pas les paires car elles ne modifient pas la parité. Le grand avantage de ce code est sa **simplicité d'implantation** (un arbre de XOR ou XNOR suffit) et son **surcoût très modéré** (un seul bit est ajouté par mot, avec le codeur et le décodeur).

Une autre famille de codes très répandue est celle des **codes de Hamming**. Ici les **mots de code** sont choisis de façon à avoir une certaine **distance minimale d** entre eux. Cette distance est calculée par une fonction P (**poids de Hamming**), qui correspond au nombre de bits différents. Pour être plus précis, pour un mot d'information de k bits on ajoute r bits de contrôle, en obtenant un mot de code de n=r+k bits, dont seulement un sous-ensemble *H* de 2^k mots appartient au code sur les 2^n mots possibles. Une erreur dans un mot $m \in H$ va donc générer un mot $m' = (m+e) \notin H$, où e est le **vecteur d'erreur**, composé par les bits qui ont changé. Si **P(e)<d/2** il est possible de comprendre quel était le mot m originaire, le restaurer, et donc **corriger** l'erreur, comme montré dans la Figure 1-19. Il reste toujours possible de détecter une erreur jusqu'à ce que **P(e)<d**. Par exemple un code de distance 3 pourra corriger une erreur simple et détecter une erreur double.

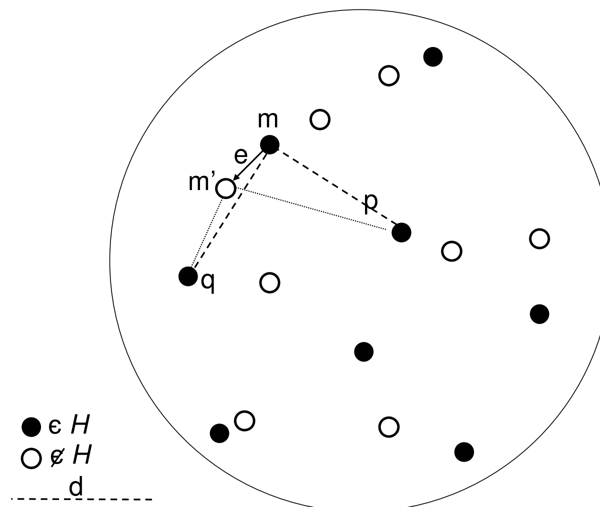


Figure 1-19 Exemple de code correcteur : m' peut être reconduit à m parce que c'est le mot de code le plus proche.

Le décodage est effectué en calculant le **syndrome s**, un vecteur dont le poids P(s) donne la magnitude de l'erreur (le nombre de bits erronés) : bien sûr si P(s)=0 il n'y a pas eu d'erreur et si P(s)<d/2 l'erreur est corrigeable. Un simple composant pourra dont lire s et décider si le mot reçu m_v a besoin d'être corrigé pour obtenir la sortie m_c ou si une erreur doit être signalée.

Un code de Hamming **H(n,k,d)** est donc défini par la **taille de son mot de code** (n), la **taille de données à protéger** (k) et sa **distance minimale,** à partir de laquelle on peut facilement calculer sa **capacité de détection** (d), et sa **capacité de correction** (d-1/2). Le surcoût d'un code de Hamming est composé par les r bits de contrôle supplémentaires : comme on peut facilement l'imaginer, le nombre de bits à ajouter sera d'autant plus grand que l'on souhaite corriger ou détecter des erreurs de poids plus grand. Une grande qualité des codes de Hamming est que toutes les opérations correspondent à des multiplications de matrices, faciles à implanter :

- la génération des bits de contrôle se fait en multipliant la matrice H caractéristique du code par le vecteur d'origine m : h_m=H*m ;

- le syndrome est calculé de la même façon avec le vecteur reçu v : s=H*v

Les codes de Hamming les plus utilisés sont sans doute ceux avec d=3, souvent appelés **SEC/DED** (Single Error Correction/Double Error Detection), et parfois implantés dans des mémoires RAM suivant le schéma de la Figure 1-20.

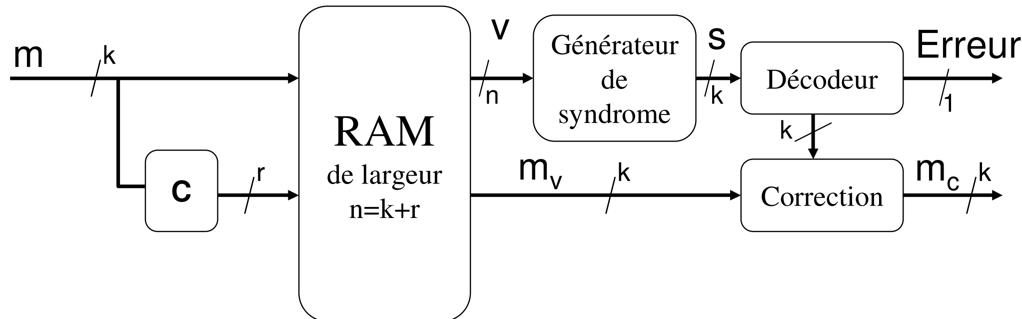


Figure 1-20 Schéma typique d'une mémoire RAM protégée par code de Hamming.

Presque tous les codes développés plus récemment suivent le même principe que les codes de Hamming : l'évolution consiste principalement à trouver des règles mathématiques qui permettent de définir des codes avec de bonnes distances minimales qui demandent peu de bits de contrôle ou qui soient faciles à calculer/vérifier pour augmenter les performances. On cite par exemple les Codes à Redondance Cyclique ou CRC, les codes de Berger et les codes à résidu.

Les code **m parmi n** sont un cas à part digne d'être noté : dans ces codes, pour un mot de n bits, seulement m peuvent être à 1, les autres doivent être forcément à 0. Leur surcoût en terme de bits additionnels est assez élevé, mais dans des applications spécifiques ces codes peuvent être très simples à former et à décoder : le code 1 parmi n (« One hot ») est par exemple très utilisé dans les machines à états finis pour coder les états.

1.4.3.4 Approches matérielles mixtes

Un exemple représentatif de système tolérant aux fautes à travers des techniques introduites dans les paragraphes précédents est le STAR (Selt-Test And Repair) présenté dans le chapitre 2 de [Pradh96] et reproduit dans la Figure 1-21.

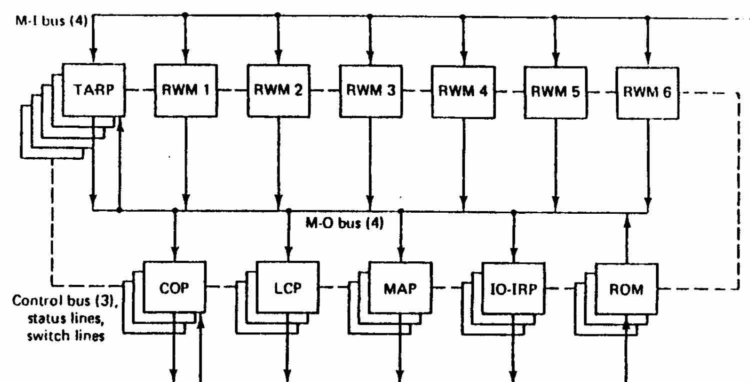


Figure 1-21 Organisation du système STAR, prise de [Pradh96]

Conçu pour des applications spatiales au début des années 70, ce système est un exemple d'utilisation intensive de la m-n modularité : tous les composants du système visibles en bas de l'image sont en 3-1, avec un exemplaire en ligne et deux réserves hors-ligne.

Chaque copie est dotée de systèmes de détection adaptés à sa structure interne et communique d'éventuelles détections d'erreurs à l'unité de reconfiguration TARP (Test And Repair Processor), qui se charge de déconnecter une unité fautive et d'activer une des copies disponibles. Etant donnée la criticité de cette tâche, le TARP est protégé par une modularité en 5-3, pour avoir un masquage complet des erreurs possibles. En plus, les données qui transitent sur les bus sont protégées par un code à résidu, en obtenant 28 bits utiles pour un mot de 32 bits, transmis en 8 fragments successifs sur le bus de 4 bits. Pour augmenter encore la sûreté les fragments sont codés en 2 parmi 4, en obtenant donc 15 bits utiles par mot de 32 bits. Le STAR a été l'une des premières tentatives d'application intensive de la tolérance aux fautes pour obtenir des systèmes à longue vie : ce système affiche une probabilité de 0,95 d'être opérationnel après 10 ans d'activité et une probabilité de 0,90 d'être opérationnel après 20 ans. Pour avoir un élément de comparaison, un système non tolérant à la même époque assurait seulement une probabilité de 0,019 d'être actif après 20 ans : le STAR apporte un gain de presque deux ordres de grandeur.

1.4.4 Protection niveau système d'exploitation

Les systèmes d'exploitation sont très difficiles à protéger car souvent on ne peut pas connaître à l'avance leur comportement, surtout si on ne connaît pas a priori et en détail les tâches qui vont être exécutées. Cependant leur importance dans le fonctionnement global des systèmes complets est grande et ils ne peuvent pas être oubliés. Dans [Denni76] on considère qu'un système d'exploitation est sûr s'il vérifie les propriétés suivantes :

- **Confinement d'erreur** : les erreurs générées par une tâche de doivent pas se propager aux autres tâches ou à l'OS lui-même ;
- **Détection** et **Catégorisation** : les erreurs sont détectées et identifiées le plus rapidement et précisément possible ;
- **Reconfiguration** : le système doit être capable de retourner vers un état cohérent ;
- **Redémarrage** : une fois l'erreur corrigée/détectée, le système doit pouvoir redémarrer ou poursuivre l'exécution normale.

Si on analyse les systèmes d'exploitation actuels, bien catalogués et expliqués dans [Tanne87], on peut remarquer que ces propriétés font désormais partie des caractéristiques fondamentales. Dans 1.3 on a introduit le concept de processus, cette « coquille » dans laquelle les tâches à exécuter sont enfermées, et de l'ordonnanceur, le composant qui se charge de choisir le processus à exécuter. Ensemble ces deux éléments peuvent déjà garantir la plupart des propriétés mentionnées :

- **Confinement d'erreur** : un processus peut accéder seulement à un ensemble fermé de ressources, et donc ne peut pas corrompre les ressources des autres;
- **Reconfiguration** : si un processus tombe dans un état erroné l'ordonnanceur reprend en main l'exécution et peut donc changer à volonté la configuration du système d'exploitation;
- **Redémarrage** : c'est toujours l'ordonnanceur qui peut décider de redémarrer un processus erroné ou de continuer l'exécution d'un autre processus dans la liste.

Pour ce qui concerne la « détection et catégorisation » la plupart des processeurs fournissent des fonctionnalités matérielles qui aident les OS dans cette tâche, par exemple en détectant les instructions incorrectes en phase de décodage (pour un PC Intel erreur « SIGINT : Instruction inexistante »), en signalant les erreurs dans les opérations

arithmétiques (division par zéro, etc.…) ou en permettant de définir des zones de mémoire à accès réservé (« Segmentation Fault »). Ces mécanismes, déjà préconisés dans [Denni76] sont devenus rapidement incontournables parce que les systèmes d'exploitation doivent faire tourner des logiciels qui ne sont pas forcément fiables et peuvent présenter plusieurs bugs (voir paragraphe suivant) ; c'est la tâche de l'OS de permettre un bon fonctionnement même dans ces conditions.

Le problème principal des processus est qu'ils sont souvent assez lourds pour le système d'exploitation : ils demandent beaucoup de ressources, et donc ils sont longs à créer, à suspendre et à redémarrer parce qu'il faut sauvegarder/restaurer beaucoup d'informations. Pour ces raisons on a introduit le concept de **thread**(« fil » en anglais), qui est un processus « léger » : plusieurs threads peuvent partager le même espace mémoire, donc les opérations de création/suspension/redémarrage sont beaucoup plus rapides. L'OS perd un peu en terme de confinement d'erreur car il y a la possibilité qu'un thread aille écrire dans l'espace d'un autre, sans que l'OS puisse s'en apercevoir, mais le gain de vitesse et l'économie de ressources sont considérables, raisons pour lesquelles plusieurs OS utilisent les threads, même en dehors du domaine des systèmes embarqués (par exemple Linux et Windows supportent processus et threads).

Du point de vue de la tolérance aux fautes, le système d'exploitation doit maintenir les propriétés mentionnées auparavant et donc en général l'attention va tomber sur l'élément clé du système à processus : l'ordonnanceur. La solution la plus employée est sans doute la **réplication des tâches** : plusieurs copies de chaque tâche sont exécutées en parallèle, suivant un schéma similaire à la redondance matérielle introduite dans le paragraphe précédent. Cette technique demande bien sûr un grand nombre de ressources additionnelles et est donc le plus souvent appliquée dans un contexte multiprocesseurs, où elle est particulièrement efficace : on peut typiquement avoir un système où un processeur dédié s'occupe de l'ordonnanceur et envoie les tâches à exécuter vers d'autres processeurs, comme indiqué par exemple dans [Aloma01]. Plusieurs études ont été menées sur le sujet, par exemple en définissant des tâches « fantômes » qui sont exécutées seulement si nécessaire ([Krish86]), en divisant les tâches en parties obligatoires et optionnelles ([Mejia00]), ou encore en développant des algorithmes adaptés pour optimiser l'ordonnanceur dans ces circonstances ([Aloma01], [Burns91], [Kanda99]). Cependant dans le cadre des systèmes monoprocesseurs ces approches restent encore trop lourdes et ne sont pas très répandues.

Une autre catégorie d'approche souvent développée au niveau des OS est de ne pas le protéger lui-même, mais d'**offrir des fonctionnalités** de protection pour les tâches qu'il fait exécuter. L'hypothèse de base est que le temps d'exécution de l'OS est raisonnablement très inférieur au temps d'exécution des tâches (voir par exemple la Figure 1-6), et qu'il est donc beaucoup plus probable pour une faute de se produire pendant le déroulement des tâches. Le cas le plus typique est l'implantation niveau OS du « Checkpoint and Rollback » (voir paragraphe suivant), dont le system TANDEM est un bon exemple : il sera présenté en détail dans le paragraphe 1.4.6.

1.4.5 Protection niveau logiciel

Le logiciel est devenu ces dernières années un des éléments les plus critiques dans un système. La raison principale est la grande complexité que les programmes peuvent atteindre, rendant très difficile leur vérification et leur validation. Tous les programmeurs et utilisateurs des ordinateurs savent que tous les programmes sont hantés par des erreurs de programmation ou d'algorithme, regroupées en général sous le terme anglais de « **bug** » (une « punaise » qui se cache dans le code). Ces bugs peuvent causer des problèmes plus ou moins graves, être systématiques ou plutôt imprévisibles, mais ils ont le même résultat : depuis longtemps les programmes à exécuter doivent être traités comme des éléments non fiables. La première conséquence est un haut degré de cloisonnement pris en compte dans tous les OS, comme mis en évidence dans le paragraphe précédent. La deuxième conséquence est une grande attention portée au processus de vérification des logiciels pendant leur cycle de développement. Ces dernières années, une attention croissante a également été portée aux techniques de tolérance aux fautes pendant l'exécution, due à l'augmentation de la probabilité des fautes transitoires dans un système, pouvant corrompre l'intégrité des données ou même des instructions stockées en mémoire ou manipulées par le programme. Lorsqu'il n'est pas possible d'implanter suffisamment de protections au niveau du matériel (pour des raisons de coût, d'utilisation de composants standard ou de flexibilité), il devient nécessaire d'implanter ces protections dans le logiciel d'application. On va se concentrer ici sur trois approches représentatives : deux grands classiques comme la réplication et l'utilisation de points de reprise (Checkpoint and Rollback) et une technique basée sur la protection au niveau instructions.

Pour reprendre la terminologie utilisée pour le matériel les bugs sont assimilables aux défauts de fabrication ou aux fautes permanentes. Ils doivent donc être détectés et résolus avec des techniques de test ou de vérification, et restent en dehors du sujet de travail de ce document. Dans les paragraphes suivants on présente donc des techniques logicielles contre les fautes transitoires.

La **réplication** est bien sûr une application au niveau logiciel des techniques de duplication/triplication (voire de réplication m-n modulaire) présentées pour le niveau matériel dans le paragraphe 1.4.3. Le problème principal est le coût de la technique, non seulement au niveau du temps d'exécution (ou de la puissance de calcul nécessaire) mais aussi, selon le type d'implantation, au niveau matériel à cause de l'augmentation de la quantité d'informations à stocker. Une variation est la **programmation à n-versions**, où chaque réplique est implantée de façon indépendante, avec des langages de programmation différents et/ou des algorithmes différents. Le but est de diminuer la probabilité d'avoir la même réaction des différentes versions envers des fautes transitoires et de réduire la probabilité d'effets semblables dans les répliques à cause des « bugs ». Le principal problème alors est d'obtenir plusieurs algorithmes sensiblement différents pour effectuer le même traitement, sans parler du temps de développement supplémentaire.

La technique des points de reprise (« **Checkpoint and Rollback** », souvent indiqué **C&R**) est une technique bien adaptée au logiciel : l'idée est de sauvegarder à des instants bien définis (appelés « checkpoints » en anglais) l'état de la tâche en phase d'exécution pour pouvoir le restaurer ultérieurement (rollback) si nécessaire. Un rollback est déclenché quand la tâche ne satisfait pas un test d'acceptation qui est chargé de vérifier que l'exécution s'est passée correctement, comme illustré dans la Figure 1-22.

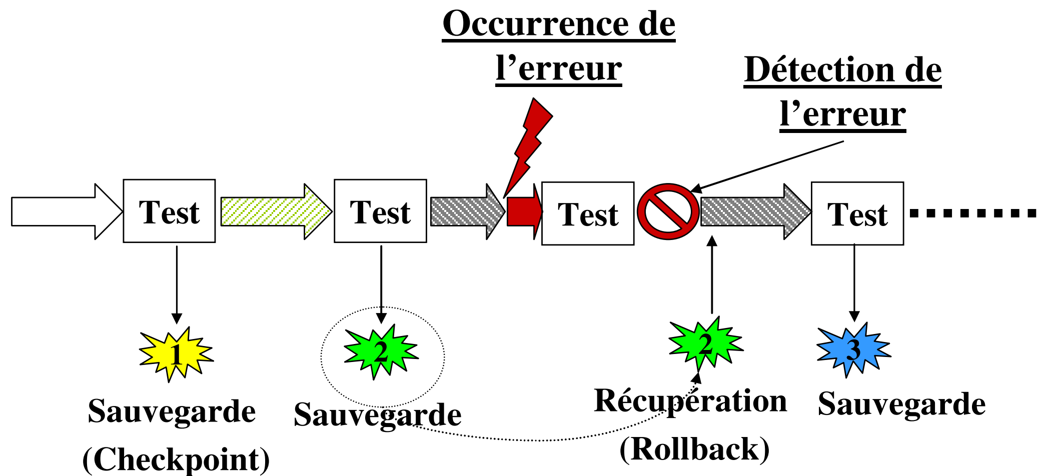


Figure 1-22 Exemple de recouvrement par « Checkpoint and Rollback »

La principale difficulté pour implanter cette protection est justement de pouvoir définir ces tests d'acceptation, donc d'avoir une profonde connaissance du comportement de la tâche, qui doit donc avoir de bonnes qualités de prédictibilité. Les opérations de test et de sauvegarde peuvent aussi être très lourdes en termes de surcoût d'exécution car elles peuvent nécessiter un grand nombre d'accès mémoire, sans parler de l'espace requis pour stocker les sauvegardes. La stratégie d'ordonnancement doit être adaptée à ces nouvelles conditions, comme présenté par exemple dans le très complet [Punne01] ou dans le tout récent [Sang05], démontrant que le sujet est encore très ouvert.

Le C&R est une des approches les plus efficaces pour la tolérance aux fautes logicielles parce qu'il peut corriger toutes les fautes transitoires, même multiples, qui sont détectées par les tests d'acceptation. Les fautes permanentes sont son principal point critique car elles peuvent engendrer des boucles infinies de détection/recouvrement. Le temps de reprise assez long en cas de recouvrement rend aussi le C&R difficile à accepter en cas de systèmes temps réel avec contraintes temporelles assez stricte.

Cette technique est souvent intégrée avec le système d'exploitation, qui fournit au développeur une série de fonctions dédiées qui lui permettent de définir les procédures de sauvegarde, les tests de validité et les routines de recouvrement. Il y a cependant toujours besoin d'une intervention du programmeur au moment de la conception de l'application, raison pour laquelle on a préféré cataloguer cette technique comme une approche « niveau logiciel » plutôt que « niveau système d'exploitation ». Il faut noter aussi que des techniques de type C&R peuvent être dans certains cas aidées par des dispositifs implantés au niveau matériel, voire même être adaptées pour être réalisées sans intervention du logiciel ; un exemple en est la technique du « micro-rollback » appliquée à des structures de pipeline [Tamir90]. Toutefois, le C&R est très majoritairement implanté sous forme logicielle.

Des approches appliquées au niveau du logiciel d'application ont également été développées récemment, basées sur des modifications automatiques du code source qui permettent d'obtenir de la tolérance aux fautes transitoires [Rebau00] [Nicol04]. L'idée de base est de répliquer les accès mémoires et les opérations exécutées et d'insérer des assertions pour vérifier l'état de l'exécution, le tout de façon automatique et transparente au développeur. Un autre élément fondamental pour le bon déroulement d'un programme est le flot d'exécution, c'est à dire la séquence d'instructions exécutées. Pour contrôler ce genre de fautes le programme est en général divisé en plusieurs blocs de base, dont l'enchaînement est validé par des techniques de signature plus ou moins sophistiquées selon le niveau de protection souhaité. Les approches se différencient principalement par la façon dont ces

contrôles sont implantés et par le niveau d'intervention : par instruction assembleur simple ou en groupe, par instruction de langage de haut niveau, par blocs fonctionnels, etc… Un autre élément fondamental est le niveau d'automatisation : les logiciels contemporains sont des éléments de grande complexité et les analyser à la main serait une tâche très onéreuse et difficile, capable d'augmenter sensiblement le coût total et le TTM, sans compter le risque d'introduire de nouvelles fautes ou bugs. Cependant écrire un parseur capable d'interpréter des codes source d'une certaine complexité est très difficile, élément qui a longtemps ralenti le développement de ces méthodes. En plus dans les milieux embarqués les codes sont souvent très optimisés, ce qui rend la tâche encore plus difficile. Heureusement les développeurs peuvent désormais compter sur plusieurs années d'expérience, ce qui a permis aux techniques de SIFHT (« Software Implemented Hardware Fault Tolerance », ou « Tolérance aux fautes matérielles implantée en logiciel »), d'aboutir à un bon degré de maturité. La plupart des problèmes d'interprétation du code source a été résolue grâce à des parseurs de plus en plus évolués et l'attention est désormais portée sur l'optimisation des capacités de couverture, la réduction des surcoûts et l'optimisation des techniques de protection.

On va ici présenter plus en détail l'approche introduit dans [Rebau00], [Cheyn00], et [Nicol04], dont les dernières évolutions sont présentées dans [Berna06] et qui est une bonne synthèse du savoir-faire développé au fil des années par la communauté de spécialistes.

Les données sont protégées par une duplication systématique : toutes les variables sont présentes en deux répliques et leur cohérence est vérifiée après chaque lecture. Le flot d'exécution est protégé en donnant à chaque bloc fonctionnel un identifiant unique, dont la valeur est sauvegardée au début du bloc dans une variable qui est contrôlée à la fin du bloc. Si une faute cause un saut non voulu entre deux blocs la variable de contrôle n'aura pas été modifiée, permettant donc la détection. Les sauts conditionnels sont placés sous une surveillance particulière vu leur importance dans le déroulement du flot d'exécution. Toutes les instructions du type « si, alors, sinon » sont transformées pour permettre de détecter toute erreur ou faux branchement. Les surcoûts introduits sont importants en termes d'occupation mémoire et des performances, mais les analyses présentées par [Cheyn00] montrent que la couverture des fautes est importante, proche du 100% en simulation selon le modèle de fautes employé (bit-flips dans la mémoire), et autour de 86% dans des tests sous radiation (toujours que la mémoire a été irradiée). Malgré les évolutions technologiques, les coûts induits peuvent être rédhibitoires pour des systèmes embarqués avec de fortes limitations des ressources disponibles.

La limitation principale de ces approches est justement dans la modélisation des fautes, qui sont en général limitées à des inversions de bits en mémoire. Cependant une catégorie de fautes très importantes est constituée par les fautes internes au processeur dont les effets en termes logiciels sont très difficiles à modéliser a priori. Si en plus le système n'exécute pas un seul programme mais un système d'exploitation avec plusieurs threads en compétition et des interruptions externes, le cadre est encore plus compliqué. L'implantation de ces protections dans notre prototype permettra d'analyser leur comportement à partir de fautes injectées au niveau matériel, comme cela sera détaillé dans le Chapitre 3.

1.4.6 Approches mixtes

De nombreux systèmes tolérants aux fautes ont été développés au fil de années, en faisant appel aux techniques les plus avancées disponibles au moment de la conception. Les résultats ont été souvent très déséquilibrés, avec un seul paradigme de protection poussé à l'extrême. Cependant, il y a eu de remarquables exceptions où une bonne interaction entre différentes techniques a donné de très bons résultats. Un exemple est sans doute le système STAR présenté dans le paragraphe 1.4.3, où plusieurs techniques de détection/correction sont appliquées ensemble : réplication n-m modulaire au niveau de l'architecture, codes pour les données et communications, et techniques ad hoc dans chaque composant. Il dispose aussi d'une liaison avec le logiciel très intéressante, qui n'avait pas été mentionnée auparavant : le logiciel peut indiquer des points de sauvegarde qui seront exploités par le système s'il y a besoin d'un recouvrement, obtenant un C&R aidé par le matériel.

Le C&R est justement l'exemple le plus typique des interactions du logiciel avec les autres niveaux : le programmeur du logiciel d'application peut exploiter des fonctionnalités spécifiques du matériel (comme pour le STAR) ou plus typiquement du système d'exploitation pour définir les point de sauvegarde, les tests d'acceptation et les routines à utiliser en cas de recouvrement. Un exemple très intéressant d'interaction sur les trois niveaux est donné par le système Tandem ([Siewi91], [Tande90] et [Tande91]) : le système d'exploitation Guardian OS exploite la duplication matérielle pour offrir au logiciel des fonctionnalités de C&R à hautes performances. L'architecture Tandem est, comme on peut le voir sur la Figure 1-23, basée sur deux processeurs à « couplage léger » : un processeur, appelé « primaire », exécute le programme et envoie au deuxième processeur, le « secondaire », seulement les informations de sauvegarde.

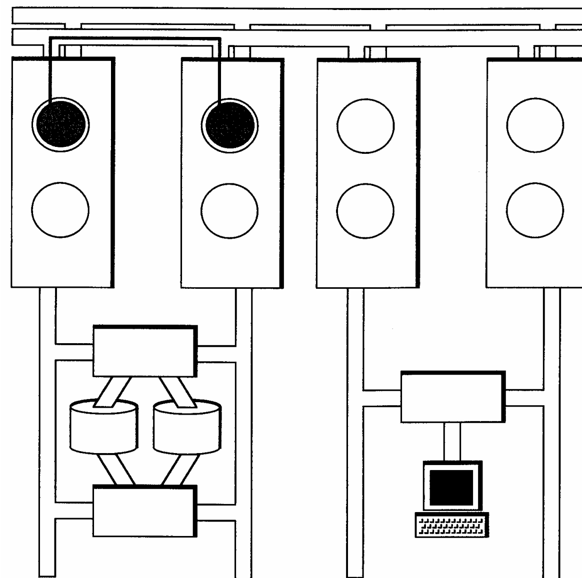


Figure 1-23 Le processeur primaire envoie les informations de sauvegarde au processeur secondaire dans une architecture Tandem. Image prise de [Tande90].

Le deuxième processeur est en modalité passive et se contente d'enregistrer les informations qu'il reçoit, avec une bonne économie d'activité et donc de consommation. En cas de défaillance du processeur primaire, le secondaire peut directement prendre le relais et continuer l'exécution à partir du dernier point de reprise, avec un temps de réaction très court comparé à un recouvrement classique. C'est le Guardian OS qui s'occupe de ces opérations, en exploitant l'architecture Tandem et en particulier le bus dédié DYNABUS™ pour

transférer à haute vitesse les informations entre les deux processeurs. Le système offrait donc une solution complète pour obtenir de la tolérance aux fautes à haute performance et un système à haute disponibilité, mais demandait une duplication matérielle complète avec en plus des composants dédiés (et protégés par copyright) comme le DYNABUS^{TM}, le système d'exploitation propriétaire Guardian et du logiciel adapté.

1.5 Sécurité - confidentialité

La sécurité - confidentialité est l'un des attributs cités lors de la présentation de la sûreté de fonctionnement au paragraphe 1.4.1. Une partie spécifique lui est consacrée car cette sécurité est en fait basée sur des aspects très spécifiques. Nous verrons que les protections précédemment citées peuvent être aussi utilisées pour atteindre des objectifs de sécurité au sens confidentialité. Notamment, ce type de technique peut s'appliquer pour protéger des procédures de chiffrement contre certains types d'attaques. Cette section présente donc rapidement les principes d'un chiffrement et les différents types d'attaques. Dans le contexte de la sécurité, nous verrons que l'origine et les caractéristiques des fautes peuvent différer des fautes mentionnées jusque là et les contraintes sur les réactions du système lorsque des fautes surviennent sont également différentes.

1.5.1 Chiffrement et sécurité

L'utilisation croissante des systèmes électroniques dans la vie de tous les jours a amené de plus en plus d'informations sensibles dans ces équipements. Désormais il est typique d'avoir de nombreuses données personnelles et/ou confidentielles stockées dans des appareils électroniques : informations bancaires (carte bancaire avec puce, sauvegardes dans l'ordinateur ou même dans le téléphone portable pour achats en lignes), mots de passe (ordinateur personnel), courriers et historique de navigation (ordinateur), musiques ou vidéos protégées par copyright, etc.… Ce phénomène est encore plus marqué par le développement des systèmes embarqués, qui génèrent presque chaque jour de nouveaux gadgets avec de nouvelles possibilités et de nouveaux risques d'intrusion. En effet, plus la connectivité augmente et plus on offre de points d'entrée pour violer nos secrets : dans [Hager03] on peut voir comment Bluetooth ou IEEE 802.11, les deux standards de communications sans fils les plus utilisés aujourd'hui, sont intrinsèquement vulnérables aux attaques.

Le principal système pour garantir la confidentialité est dans le **chiffrement** des données : les principes mathématiques mis en jeu sont très complexes, et on ne cherchera ici qu'à en donner un aperçu le plus complet possible sans descendre dans des détails théoriques trop complexes. L'idée de base est assez simple, et est résumée dans la Figure 1-24 : un émetteur, que l'on appellera « Alice » suivant la tradition des textes spécialisés, veut transmettre un message secret P (« Plain Text », ou texte initial « en clair ») à un récepteur, appelé Bob, à travers un canal de communication public non protégé. Pour cela elle utilise un secret commun, la clé K (« Key ») et une fonction de chiffrement E (« Encoding ») pour générer un texte chiffré C (« cipher text »). Bob reçoit donc C et grâce au secret commun K et à une fonction de décodage D il réobtient le texte secret P.

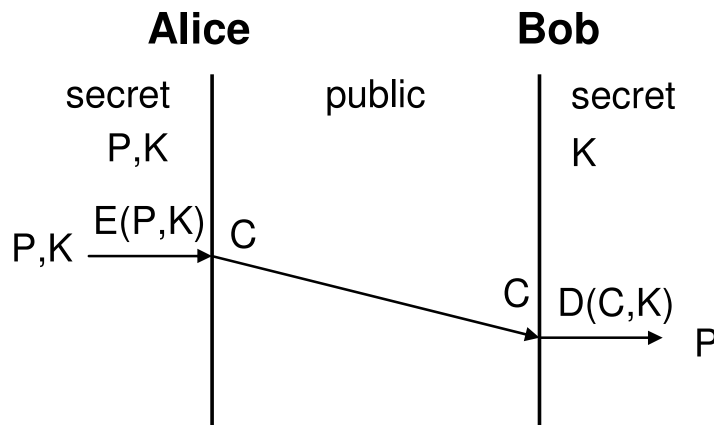


Figure 1-24 Schéma de cryptographie à clé secrète

Le texte C a une propriété très importante : sans le secret commun K, il est mathématiquement difficile, voir impossible, de reconstruire P à partir de C parce que ce dernier est statistiquement indépendant de P (dans le cas optimal). « Mathématiquement difficile » signifie que même si théoriquement l'opération est possible elle demanderait une quantité de calculs trop élevée pour être réalisable dans un temps suffisamment court, avec les puissances de calcul disponibles à un moment donné. Si on a une clé secrète de n bits, il faudrait faire jusqu'à 2^n essais pour obtenir le texte P à partir de C, même si la fonction D est publiquement connue. Pour les tailles les plus typiquement utilisées aujourd'hui, 128 et 256 bits, il faudrait $2^{128}=10^{36}$ ou $2^{256}=10^{75}$ essais respectivement. Ces chiffres correspondent à des attaques « brutales » (i.e. essai de toutes les combinaisons possibles) et peuvent être sensiblement diminués grâce à des stratégies plus raffinées, mais ils sont quand même de bons points de référence. En pratique on considère qu'au-delà de 2^{50} un schéma de cryptographie peut être considéré sûr (face aux attaques de force brute) car il n'existe pas encore de réelles possibilités de faire autant d'opérations en un temps acceptable, sauf si on considère le calcul quantique, supposé pouvoir atteindre des capacités de calcul inimaginables avec les outils classiques (voir [Niels00] pour une référence complète). Mais cela restera encore pour de longues années une proposition théorique plutôt qu'une vraie menace. En revanche, les attaques bien ciblées sur un algorithme de chiffrement donné sont beaucoup plus dangereuses, en permettant de réduire considérablement les efforts de calcul requis et donc en rendant le décodage possible même si une attaque de force brute n'est pas envisageable.

Pour atteindre le niveau de confidentialité requis, on varie la taille de la clé K ou l'algorithme de codage/décodage : on peut trouver beaucoup d'exemples célèbres, dont les plus importants sont sans doutes DES et AES, standardisés par le NIST (plus d'informations sont disponibles dans [Nist06]). Ces algorithmes sont basés sur des transformations/permutations de blocs de bits de P pour produire C. Ils sont faciles à implanter en logiciel ou en matériel, offrent un bon niveau de sécurité (surtout AES, DES souffrant maintenant de son age) et grâce à la standardisation ils ont eu un grand succès. Cependant ils ont le même point faible que tous les algorithmes de cette catégorie : émetteur et récepteur doivent connaître tous les deux la clé secrète K qui doit donc être transmise de façon sûre au récepteur. On dit en général qu'un schéma de ce type est **symétrique** ou à **clé secrète** justement car Alice et Bob doivent avoir les mêmes connaissances.

Pour dépasser cette limitation, des techniques **asymétriques** ou à **clé publique** ont été développées suivant le schéma de la Figure 1-25 : Bob dispose de deux clés, une secrète K^b_s et une publique K^b_p, cette dernière pouvant tranquillement être transmise à Alice à travers le canal non sûr. Alice utilise cette clé et une fonction de codage commune E pour générer le

texte chiffré C, qu'elle envoie à Bob. A ce point là, Bob peut utiliser la fonction de décodage D avec sa clé secrète K^b_s pour retrouver le texte P.

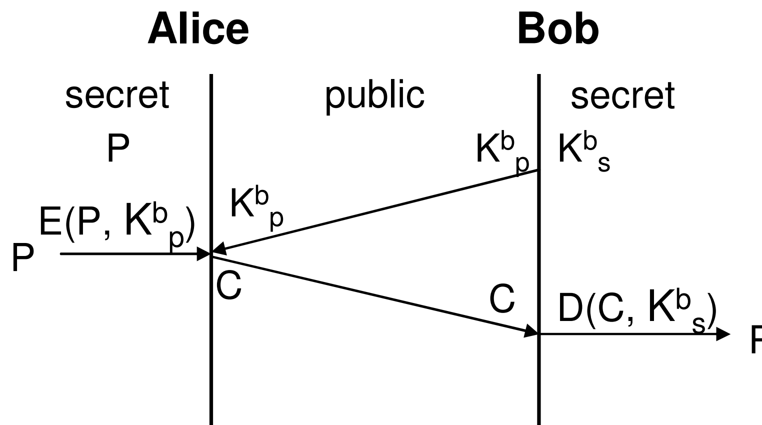


Figure 1-25 Schéma de cryptographie à clé publique

Le grand avantage d'un schéma de ce type est qu'il n'y a aucun secret qui doit être partagé entre les deux, donc il n'y a pas besoin d'une communication sécurisée préalable. Le coût à payer par contre est l'utilisation de mathématiques beaucoup plus complexes, basées sur les isomorphismes entre espaces polynomiaux à travers des fonctions « à sens unique ». La théorie est très complexe et on n'a pas la présomption d'expliquer ici tous les détails. Dans le cadre de cette thèse, l'approche à la cryptographie a été de type « utilisateur » plutôt que « développeur » : les détails théoriques peuvent être trouvés dans des textes spécialisés comme [Stins96], [Masse98a] et [Masse98b]. A la base, une « fonction à sens unique » est une fonction f(x) qui est mathématiquement difficile à inverser. Encore plus importantes, certaines fonctions à sens unique sont dites « avec trappe » : pour ces fonctions $f^{-1}(x)$ est difficile à calculer sauf si on a une information supplémentaire qui permet d'« ouvrir la trappe ». Sur cette base, on peut construire des schémas de cryptographie asymétrique où la clé publique est un paramètre qui permet de définir f(x) et la clé secrète est l'information qui permet d'ouvrir la trappe. La fonction à sens unique classique est le logarithme entier et elle est utilisée par exemple pour l'algorithme RSA ([Rives78], [Masse98b]) : deux grands nombres premiers (512,1024 ou 2048 bits de taille) indépendants sont utilisés comme paramètres pour définir les clés et les méthodes de codage/décodage. C'est un système qui offre une très bonne sécurité avec des coûts d'implantation raisonnables, même si bien sûr la complexité est plus grande que pour des schémas symétriques comme l'AES.

1.5.2 Cryptanalyse et attaques par fautes

Dans le paragraphe précédant on a affirmé qu'avec les moyens actuels les codes cryptographiques assez complexes peuvent être considérés « sûrs » contre les attaques traditionnelles. Laissant de côté la chimère quantique ([Niels00]) on va ici introduire les vraies menaces actuelles contre la sécurité, et notamment les attaques par fautes. Le concept de base est que si les algorithmes ne sont pas attaquables en fonctionnement « normal » on va provoquer des erreurs et donc des situations « anormales » à partir desquelles il est possible de recueillir de précieuses informations. Pour obtenir cela, on va exploiter le fait que ces algorithmes sont implantés sur un support réel et physique qui n'est pas parfait et qui peut être modifié, contrairement aux formules théoriques sur du papier. Si on **introduit des fautes** dans

le système **pendant le calcul** on va faire « dérailler » l'algorithme et donc provoquer des conditions anormales qui peuvent se révéler très intéressantes. Le cas le plus illustre est l'attaque contre le RSA implanté selon le Théorème du Reste Chinois (CRT, de « Chinese Reminder Theorem »). L'algorithme RSA est basé sur l'exponentiation de nombres de grande taille, une opération assez lourde qui peut être simplifiée grâce à ce théorème qui permet de diviser le calcul en deux parties qui utilisent des nombres de taille plus petite et sont donc plus simples, rapides et ouvertes à des optimisations importantes (ex : parallélisme). Dans [Lenst96], Lenstra a montré comment une faute générée dans certaines phases du calcul permet aisément d'obtenir la clé secrète utilisée pour le décodage : si jamais un des deux calculs de CRT est perturbé et donne un résultat incorrect, il est possible de déduire la clé secrète en comparant ce résultat faux avec le résultat correct. Un des aspects les plus préoccupants d'une telle attaque est la relative facilité et le relatif faible coût de réalisation ; pour effectuer une attaque de ce type contre un circuit de décodage RSA, il suffit de suivre cette démarche :

- identification des deux phases de calcul du CRT, grâce à un simple analyseur de courant : en général on peut trouver deux paliers de consommation très évidents ;
- génération d'une faute synchronisée sur l'un des deux paliers et récupération du résultat erroné ;
- répétition du calcul sans interférences pour récupérer la valeur correcte ;
- analyse mathématique simple pour en déduire la clé.

Une attaque par fautes combine ainsi souvent l'obtention d'un résultat erroné et une analyse mathématique de l'algorithme de chiffrement (cryptanalyse). Ceci vient en complément de la branche de la cryptanalyse qui se concentre directement sur les faiblesses liées au protocole de cryptage, en recherchant les séquences spécifiques de messages à crypter/décrypter pour extraire des informations secrètes grâce à des techniques de corrélation statistique ou similaire. C'est la tâche des mathématiciens d'identifier ces combinaisons dangereuses et de les éviter ; cet aspect ne sera pas davantage détaillé ici.

Le principe de l'attaque de Lenstra a été appliqué aux autres systèmes cryptographiques, en générant une nouvelle catégorie d'attaques basée sur l'analyse du comportement des algorithmes soumis à différents types de fautes et la comparaison de résultats corrects et erronés. Ce type d'attaque est appelé Analyse Différentielle des Fautes (DFA, de « Differential Fault Analysis »).

Il y a plusieurs techniques pour générer une faute, basées sur différents types de perturbations du système : on en donne ici un résumé, basé en partie sur [Renau04]. Une première possibilité, si l'algorithme est implanté par programmation (par exemple, dans une carte à puce) est d'agir au **niveau logiciel**, en exploitant des bugs ou faiblesses de conception du système d'exploitation ou de l'application. C'est le même principe que celui utilisé par la vague des virus et vers de dernière génération qui ont fait des ravages sur Internet ces dernières années. En général, ils exploitent des débordements de pile ou autres effets de bord pour accéder à la zone de mémoire protégée et ils sont normalement contrés par le système d'exploitation lui-même grâce à des techniques de protection comme celles présentées dans le paragraphe 1.4.4. Les attaques de ce type sont surtout dangereuses pour des systèmes de grande taille (et donc avec des OS très complexes et difficiles à vérifier dans tous leurs détails) et demandent une grande connaissance du système victime pour pouvoir être menées à terme. Si au niveau des ordinateurs de grande diffusion (notamment pour les PC avec Windows) elles sont une des menaces les plus graves, ces attaques restent limitées à des cibles particulières pour les systèmes embarqués (notamment, les cartes à puce contenant des

données sensibles, comme un code d'accès à un service particulier : banque, GSM, télévision à péage, accès physique à un bâtiment, …). Cependant les EOS renoncent souvent à certaines contraintes de sécurité pour des raisons d'optimisation (ex : les threads sont plus rapides que les processus mais offrent moins de confinement mémoire) et il faut donc faire très attention si le système embarqué cible va être inséré dans un environnement à forte connexion (ex : lié à l'Internet ou avec des fonctionnalités d'accès à distance).

Les attaques les plus répandues et les plus dangereuses pour les systèmes embarqués sont toutefois des attaques au **niveau matériel** : le fonctionnement physique du système est perturbé pour engendrer une faute. La principale classification est dans la façon de mener ces attaques :

a. les approches **invasives** modifient généralement le circuit de façon permanente, voire destructive (par exemple, en créant un court circuit ou en coupant une interconnexion). Il est aussi possible d'insérer des sondes sur certaines connexions internes ;

b. les approches **non-invasives** ne modifient pas le circuit et cherchent à analyser ou perturber son comportement en situation de travail normale ;

c. de façon intermédiaire, les approches **semi-invasives** se contentent d'une intervention modérée sur le circuit pour le préparer à des attaques non invasives. Le plus typique est, par exemple, l'ouverture du boîtier et l'enlèvement d'une couche d'oxyde de protection opaque pour préparer une attaque optique.

Les approches non invasives ou semi-invasives sont très intéressantes car elles sont souvent plus faciles à mettre en oeuvre et relativement peu chères, multipliant les attaquants possibles. En ce qui concerne la DFA, les techniques les plus répandues sont basées sur la génération de perturbations (**glitchs)** sur l'alimentation ou l'horloge (pour un circuit synchrone), sur des attaques optiques à base de flash lumineux ou de rayon laser, voire sur l'exposition de la puce à des **radiations** (notamment, sources alpha pour les attaques les plus simples à mettre en œuvre).

Désormais la plupart des circuits développés pour des applications sécurisées sont résistants aux types d'attaques **globales**, grâce à des capteurs de lumière, des générations d'horloges internes ou des moniteurs de courant permettant de détecter les perturbations dues à une attaque. Il faut noter ici qu'un circuit sécurisé au sens confidentialité n'a pas les mêmes contraintes de fonctionnement qu'un circuit développé pour des applications ayant des contraintes de sécurité au sens innocuité. En effet, l'objectif est la conservation du secret, pas forcément le maintien de la capacité opérationnelle. En d'autres termes, après détection d'une attaque, la tolérance ou le recouvrement n'est pas forcément nécessaire : il est possible de stopper le déroulement de l'application, voire dans certains cas de détruire les informations sensibles contenues par le circuit. Ceci doit cependant être réalisé de manière à ne pas donner d'informations indirectes à l'attaquant … La définition de la stratégie de sécurisation sort toutefois du cadre de cette thèse, qui s'intéressera uniquement à la détection des attaques par fautes. Par ailleurs, nous nous intéresserons particulièrement à la menace la plus importante aujourd'hui, à savoir les attaques **localisées**, qui permettent de générer des fautes sans déclencher les capteurs. Pour cela, les **lasers** sont probablement les outils les plus dangereux à disposition des attaquants : ils permettent de focaliser une grande intensité sur une zone de la puce très limitée, en évitant donc les capteurs, et en plus les impulsions sont de durée très faible et facilement synchronisables avec le calcul.

Il faut noter que d'autres types d'attaque existent, basés sur l'observation de caractéristiques physiques pendant le fonctionnement du circuit. Le principe de base de ces attaques dites par **canaux cachés** est d'observer des variables qui offrent une grande **corrélation** avec les données manipulées pendant les calculs : en observant leurs variations on peut donc extraire des informations sensibles qui devraient être cachées. Les attaques les plus utilisées sont basées sur la consommation électrique du circuit (Analyse de Puissance Simple ou Différentielle - SPA et DPA) et sur les émissions électromagnétiques liées à cette consommation (Analyse Electromagnétique Simple ou Différentielle - EMA, DEMA). Des **attaques temporelles** (Timing Attack – TA) sont également possibles : si la durée des calculs dépend des données traitées, mesurer le temps des opérations permet d'identifier ces données. Les protections (appelées contre-mesures) envers ces techniques d'attaque peuvent être à différents niveaux : mathématiques, algorithmiques mais aussi architecture des circuits. La Figure 1-26 récapitule les techniques d'attaque principales. Seules les techniques basées sur l'injection de fautes localisées seront prises en compte dans le cadre de cette thèse

1.6 Conclusions

L'analyse de l'état de l'art a permis de montrer un domaine en plein développement, celui des systèmes embarqués, qui est porté par sa propre évolution à se croiser avec la cryptographie, autre domaine d'importance croissante, et avec la tolérance aux fautes, une « doyenne » dont l'importance croît avec l'augmentation de la probabilité des fautes transitoires. Cette rencontre de plusieurs domaines demande la fusion d'expertises très différentes, qui doivent co-exister dans le même système.

Cette thèse s'est donc attachée à la recherche d'une stratégie d'implantation matérielle/logicielle dépassant la simple co-existence en faveur d'une bien plus profitable et intéressante coopération. La méthode proposée cherchera à combiner efficacement des protections complémentaires aux différents niveaux présentés : matériel, logiciel d'application et intergiciel (système d'exploitation), en visant un bon compromis entre le niveau de robustesse atteint et les coûts induits, aussi bien au niveau matériel qu'au niveau performances.

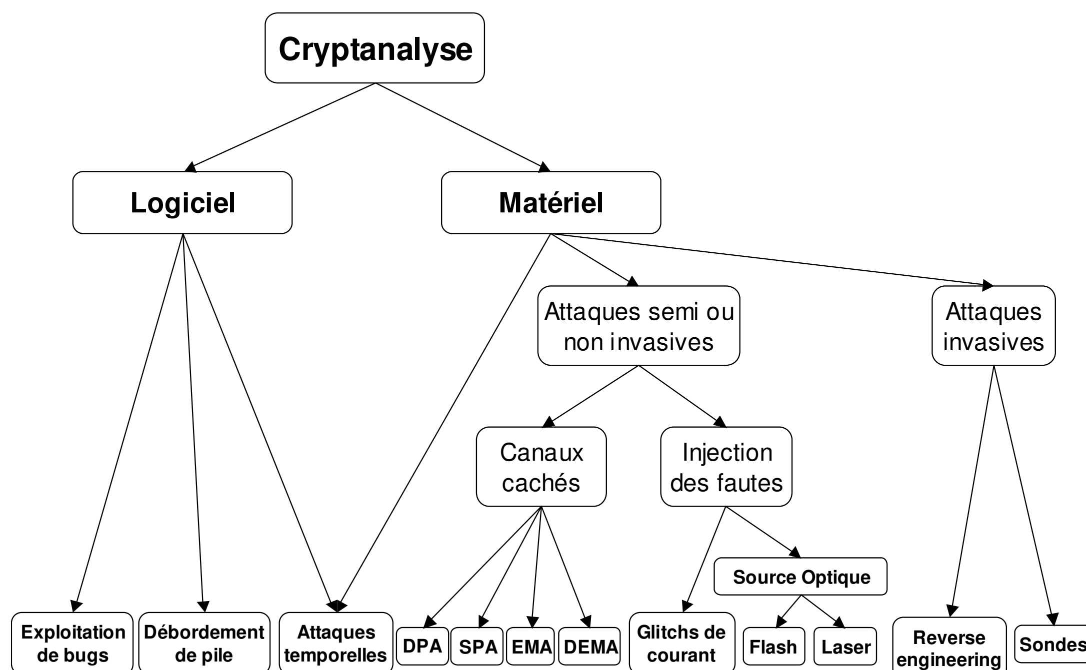


Figure 1-26 Récapitulatif des techniques utilisées en cryptanalyse

La méthode proposée sera appliquée sur un système significatif, représentatif d'un système embarqué monoprocesseur, qui sera présenté dans le chapitre suivant. Par la suite, peu de distinction sera faite entre fautes naturelles, dues par exemple à des particules, et fautes intentionnelles, dues par exemple à des attaques laser. Le modèle de fautes employé sera celui correspondant à des inversions de bits multiples (MBF), tel que présenté en section 1.4.2.

2 Système embarqué cible et approche globale de protection

Dans le Chapitre 1 on a donné un résumé des techniques de durcissement le plus complet possible, en ajoutant si possible des exemples représentatifs, dont notamment le système STAR. Cependant, ces systèmes ne sont pas disponibles et ne peuvent servir de base à notre étude, étant notamment couverts par des brevets. Il a donc été nécessaire pour notre projet de recherche de définir un système sur lequel travailler : c'est le but du paragraphe 2.1. Les paragraphes suivants, du 2.2 au 2.5, présenteront les techniques de protection proposées dans le cadre de cette thèse, leur mise en œuvre étant détaillée au chapitre 3.

2.1 Méthodologie et exemple représentatif

La complexité croissante des systèmes embarqués a considérablement augmenté les coûts de conception et de vérification. Par ailleurs, le temps de mise sur le marché, ou TTM, est souvent crucial pour la rentabilité d'un produit. Pour ces raisons, il y a de plus en plus d'intérêt pour les systèmes reconfigurables ou réutilisables qui permettent de diminuer énormément le temps de développement. Du point de vue matériel, cela signifie surtout l'utilisation de blocs réutilisables (IPs - Intellectual Properties), si possible configurables en fonction des besoins de l'application. Ces composants peuvent être décrits en utilisant des langages de description de matériel de haut niveau, ce qui leur permet d'être indépendants de la technologie cible. Du point de vue logiciel, l'attention est focalisée sur les logiciels libres ou Open Source qui permettent d'exploiter le savoir-faire d'une grande communauté d'usagers et donc d'obtenir des programmes de grande qualité avec des temps de développement/adaptation très limités. Suivant ces lignes guides, on a donc défini un exemple représentatif de système embarqué monoprocesseur :

- un processeur embarqué défini par une IP décrite à haut niveau (VHDL RTL) et Open Source : le Leon2 de Gaisler Research, présenté dans le paragraphe 2.1.1,
- un système d'exploitation embarqué, lui aussi Open Source : eCos de Red Hat, présenté dans le paragraphe 2.1.2,
- des IP pour réaliser des opérations spécifiques, présentées dans le paragraphe 2.1.3,
- des applications logicielles adaptées, présentées dans le paragraphe 2.1.4.

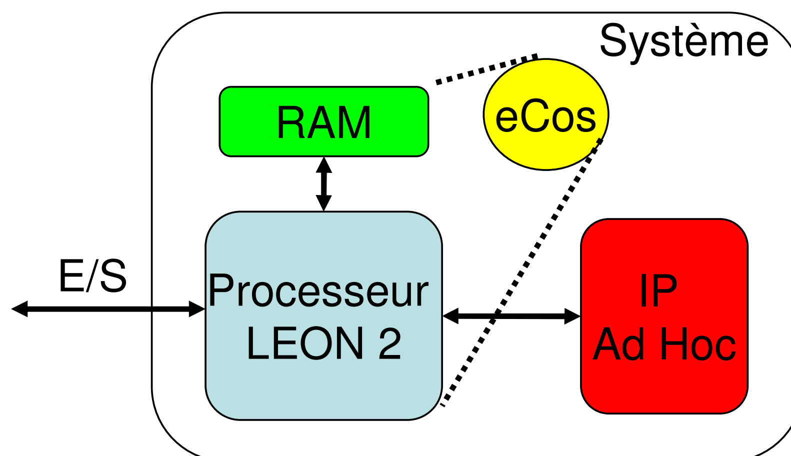


Figure 2-1 Schéma du système cible

Les approches considérées pour la tolérance aux fautes suivront les mêmes principe: certaines techniques introduites dans le Chapitre 1 seront généralisées pour obtenir des protections souples et facilement réutilisables. On concentrera notre attention sur des approches les plus générales possible, pour proposer des méthodes facilement adaptables à n'importe quel système semblable à celui illustré en Figure 2-1.

2.1.1 Le processeur Leon2

Le Leon2 est un processeur basé sur le jeu d'instructions SPARC V8 et qui a été conçu initialement pour des applications spatiales : il dérive des processeurs ERC32 et Leon2 de l'Agence Spatiale Européenne (ESA), pour laquelle le fondateur de Gaisler Research travaillait avant de créer sa propre entreprise [Gaisl06]. L'IP est disponible sous la forme d'un projet VHDL synthétisable protégé par la licence GNU LGPL qui permet d'avoir pleine connaissance du code source et pleine liberté de le modifier tant que ces modifications restent soumises elles aussi à la LGPL. En plus il existe une version tolérante aux fautes du Leon2, le Leon2-FT ([Gaisl02]), qui est payante mais qui offre une base de comparaison très intéressante. L'intérêt du Leon2 est témoigné par la production récente chez ATMEL d'un composant pour applications spatiales basé justement sur ce processeur ([Atmel06]). Il faut noter que l'IP Leon2 est utilisée dans un nombre croissant d'applications en dehors du domaine spatial, grâce à sa caractéristique d'être une IP d'utilisation libre. Par exemple dans [Hwang03] Leon2 est le cœur d'un système portable d'identification et dans [Stame06] il est utilisé pour gérer une application de communication sans fils.

Mises à part les questions de licence, plusieurs caractéristiques ont conduit au choix du Leon2, dont le schéma bloc est donné en Figure 2-2, notamment :

- un pipeline à 5 étages, bonne base pour étudier le durcissement d'un composant pipeline ;
- des mémoires caches instruction et données complètement paramétrables, composants clé dans les performances de tout processeur moderne ;
- une grande variété de configurations possibles ;
- un style de codage VHDL bien adapté aux approches décrites dans le paragraphe 2.3 ;
- un portage facile sur des cibles matérielles différentes ;
- la disponibilité de la suite développement logiciel standard GNU-Linux, dont notamment gcc, gdb et ddd ;
- la présence d'une unité de debug native, qui permet de contrôler aisément l'exécution des programmes sur un prototype ;
- le bus AMBA interne, qui permet d'ajouter facilement des périphériques comme ceux présentés dans le paragraphe 2.1.3.

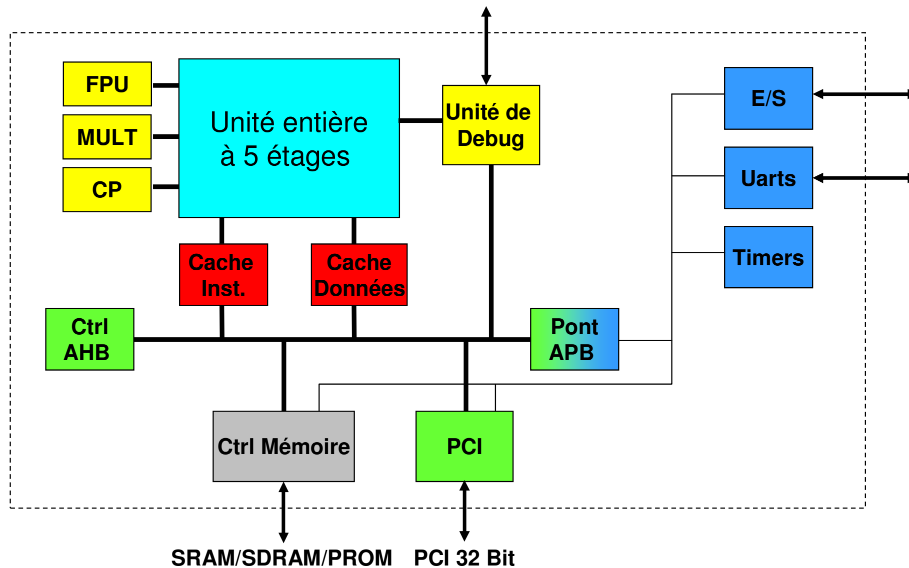


Figure 2-2 Schéma bloc du processeur Leon2

2.1.2 Le système d'exploitation eCos

En ce qui concerne le système d'exploitation, notre choix s'est porté sur eCos, un système d'exploitation embarqué proposé par Red Hat [Ecos06]. Il offre plusieurs caractéristiques intéressantes :

- sa configurabilité, qui s'accorde bien avec celle du processeur ;
- sa facilité de portage vers plusieurs plateformes grâce à une couche d'abstraction du matériel (HAL= Hardware Abstraction Level) ;
- sa documentation, très complète, précise et gérée de façon centralisée ;
- la présence d'une version complète et opérationnelle pour Leon2.

Ceux deux derniers points ont particulièrement pesé dans la comparaison avec deux concurrents très connus : Linux (pas de portage disponible en 2003) et µC-Linux (pas de documentation unifiée).

Il faut souligner que notre approche de protection, détaillée dans le paragraphe 2.2, fait référence à des fonctions de base présentes dans tout système d'exploitation, ce qui la rend très facilement applicable. eCos est parfait comme exemple représentatif, et même meilleur que Linux (dont le portage pour Leon2 est devenu disponible pendant le déroulement de la thèse) parce que plus configurable et plus facile à gérer. Toutefois une adaptation de la technique vers Linux ou un autre système cible serait très facile.

2.1.3 IPs ad hoc : AES et interface vidéo

Dans la plupart des applications réelles, un système embarqué est appelé à effectuer des opérations spécifiques, avec des contraintes de performances et/ou de temps réel qui peuvent devenir très importantes. Il est donc typique que le processeur soit connecté à des IPs « ad hoc » qui sont chargées d'accélérer des opérations ou de contrôler des périphériques. Un système qui veut être représentatif du domaine doit donc intégrer au moins un bloc de ce type.

Dans le cadre de notre travail, on en a sélectionné deux : un coprocesseur capable d'effectuer des opérations de cryptage/décryptage selon le standard AES ([Nist06]) et une interface d'acquisition et traitement vidéo.

Le coprocesseur AES répond à la nécessité croissante de sécurité dans les domaines embarqués qui sont de plus en plus portés à gérer des informations sensibles ou confidentielles. Les opérations de cryptographie sont très gourmandes en puissance de calcul et donc sont souvent accélérées par une implantation matérielle. AES est un exemple très important d'algorithme de cryptographie symétrique (voir paragraphe 1.5.1) et grâce à sa standardisation par le NIST il est très répandu et utilisé. Les données à crypter/décrypter sont organisées sur une matrice et l'algorithme est composé d'une série de calculs successifs (appelés « rondes ») qui sont basés sur des permutations entre lignes/colonnes et des transformations non linéaires. Pour simplifier ces dernières, on utilise en général des tableaux de correspondance qui contiennent tous les résultats possibles, appelés S-Box ou T-Box selon les choix d'implantation. Le décryptage est effectué à partir de la même clé que le cryptage, mais avec un autre ordre des rondes et des tableaux définissant la fonction inverse de la transformation (S-Box^{-1} et T-Box^{-1}). Le composant utilisé correspond à une implantation avec des S-Box, implanté comme ROM statiques ou mappées dans des BRAMs selon les paramètres, et fait partie de la bibliothèque d'IP de niveau RTL du groupe de recherche dans lequel j'ai préparé cette thèse. Dans le processeur Leon2 l'IP AES a été connecté sur le bus de périphériques, le APB (voir Figure 2-2 et [AMBA99]. Si ce choix s'avère trop limitante coté performances il est aussi possible de brancher l'IP sur le bus interne AHB, qui assure un débit de transfert plus important.

Du point de vue des approches de protection qui vont être proposées, utiliser une telle IP décrite au niveau RTL permet d'utiliser la même approche générale que pour le processeur Leon2. L'objectif est ici plus particulièrement de protéger ce bloc sensible contre des attaques par fautes (voir paragraphe 1.5.2). Les protections implantées seront détaillées dans le paragraphe 2.3.6.

L'interface vidéo de son coté représente bien ces IPs qui sont souvent ajoutées pour permettre au système embarqué de gérer des périphériques. Elle a été choisie pour permettre d'exécuter des applications de traitement d'images, qui sont très intéressantes car en jouant sur les paramètres de l'algorithme on peut facilement changer les contraintes de l'application : temps réel, charge de calcul, exploitation de la mémoire, etc.… Il faut noter que dans la suite du document cette IP ne fera pas l'objet d'un traitement particulier du point de vue des protections contre les fautes car cela n'apporterait pas de réelle information supplémentaire dans le cadre de notre travail.

2.1.4 Les applications logicielles

Les programmes d'application doivent permettre dans notre cas de démontrer certaines caractéristiques des approches de protection proposées et implantées sur la configuration matérielle résumée dans les sections précédentes. On a donc développé plusieurs types d'applications :

- jeux de test (benchmarks) permettant d'évaluer les performances du système. Dans cette catégorie, on privilégiera des programme standards, comme par exemple Dhrystone, mais aussi des tests plus pointus en fonction des besoins, par exemple pour évaluer les performances de certains composants du système d'exploitation ;
- applications de chiffrement, avec notamment un programme réalisant de façon complètement logicielle le cryptage AES. Cela permet d'avoir des éléments de comparaison directe avec l'IP matérielle introduite dans le paragraphe précédent. Il faut noter que l'on a utilisé ici un programme Open Source disponible en ligne [Devin04] et directement compilé sur Leon2 ;
- application de compression/décompression d'images fixes suivant le standard JPEG, basée elle aussi sur des bibliothèques Open Source [Lane98]. En choisissant le taux de compression et la fréquence de rafraîchissement on peut facilement changer la charge du processeur et les contraintes temporelles de l'application.

2.2 Protection Niveau Système d'Exploitation

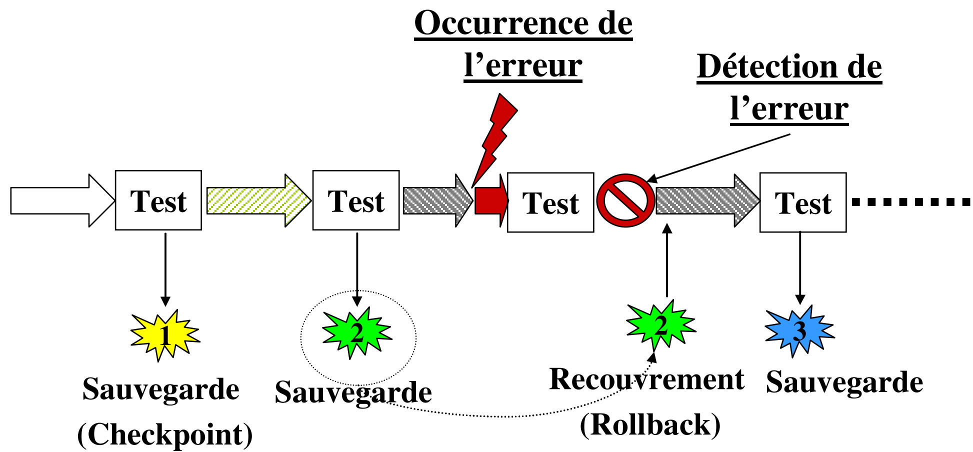
Dans un système embarqué le système d'exploitation est en général réduit au minimum pour ne pas peser trop sur les performances et sur les ressources à intégrer. Dans le paragraphe 1.4.4, on a mis en évidence comment dans ces conditions les systèmes d'exploitation offrent plutôt des protections pour les tâches qu'ils gèrent plutôt que de protéger leur propre exécution. Celle-ci ne correspond qu'à une faible partie du temps total d'exécution, et il est donc nettement plus probable qu'une perturbation ait un impact sur l'une des tâches fonctionnelles plutôt que sur l'exécution du système. Naturellement, dans un système devant avoir un très haut niveau de protection, il faudrait compléter ce qui va être présenté par une protection de la bonne exécution des appels systèmes et en général des services offerts par l'OS.

Afin de protéger l'exécution des tâches fonctionnelles, on a décidé dans notre cas d'implanter un schéma de type « Checkpoint & Rollback » qui a le point fort d'être très général et donc applicable virtuellement à toute tâche. Son principal point faible étant le coût, nous proposons une approche originale basée sur la ré-utilisation de fonctions existantes afin de réduire à la fois les ressources nécessaires et la perte de performances. L'approche proposée est présentée dans le paragraphe suivant.

2.2.1 Recouvrement à coût minimal : principes

Dans le paragraphe 1.4.5 on a introduit les principes du « Checkpoint & Rollback », résumés dans la Figure 1-22 qui est reproduite ici pour plus de clarté.

La technique se base sur une sauvegarde de l'état de la tâche à des instants bien précis (checkpoints), pour la restaurer (rollback) si une faute a été détectée par un test d'acceptation.



Reproduction de la Figure 1-22 : Principe du Checkpoint & Rollback

Le checkpoint de la tâche T_a sauvegarde l'état complet du système à l'instant « t_s » :

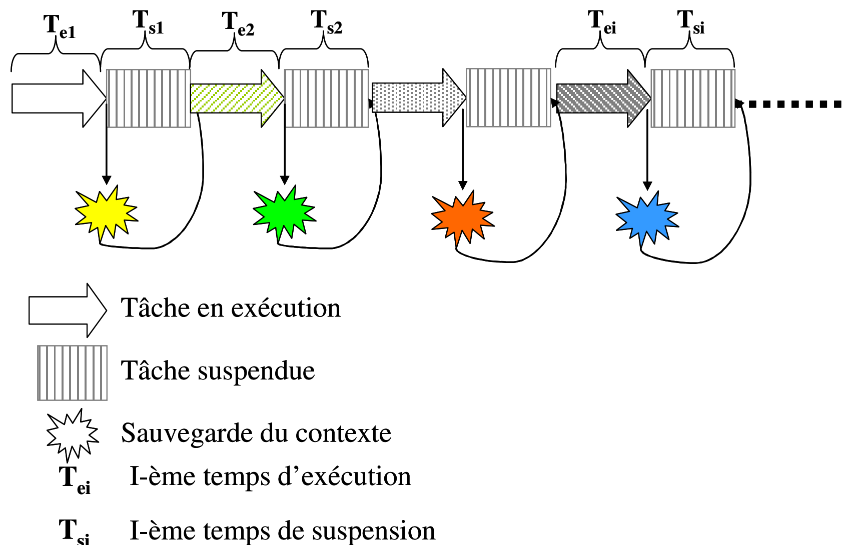
- son contexte, que l'on appellera $C_{Ta}(t_s)$, qui rassemble les valeurs des registres internes ;
- son espace mémoire, que l'on appellera $M_{Ta}(t_s)$.

Avec ces notations, un rollback à l'instant « t » (t étant postérieur à t_s, mais précédant l'instant de sauvegarde suivant) peut être exprimé par la formule suivante :

$$(1) \quad [C_{Ta}(t), M_{Ta}(t)] \rightarrow [C_{Ta}(t_s), M_{Ta}(t_s)]$$

L'état du système est donc complètement restauré aux valeurs de l'instant t_s.

Comme expliqué dans le premier chapitre, les opérations de checkpoint et de test sont souvent très onéreuses et responsables de la majeure partie du coût de réalisation. Il faut en plus mentionner la difficulté de trouver des tests d'acceptation bien adaptés à la tâche en cours d'exécution. Pour réaliser un schéma de C&R générique et à coût minimal, il faut résoudre ces problèmes. Pour cela, comparons la Figure 1-22 avec la Figure 1-7, qui montre le fonctionnement d'un changement de contexte. Elle aussi est reproduite ici pour permettre une comparaison immédiate.



Reproduction de la Figure 1-7 : Flot d'exécution d'une tâche dans un système multitâche

Le changement de contexte (CdC) sauvegarde seulement la valeur des registres, donc pour reprendre la nomenclature de la formule (1) un changement de contexte à l'instant « t » peut être exprimé comme suit :

(2) $[C_{Ta}(t), M_{Ta}(t)] \rightarrow [C_{Ta}(t_s), M_{Ta}(t)]$

L'état reconstruit est donc a priori incomplet : pour mieux comprendre les hypothèses de validité, on va faire référence à la modélisation d'un ordinateur comme une machine de Turing introduite dans le paragraphe 1.2. La tête de lecture/écriture avec son état interne représente le processeur et ses registres d'état (donc C_T), et la bande représente la mémoire de l'ordinateur, tous niveaux confondus (M_T). Chaque tâche à exécuter a son propre espace mémoire selon le principe du confinement d'erreur (voir 1.4.4), chaque espace étant représenté par une portion de la bande.

Selon ce modèle, un changement de contexte consiste en un déplacement de la tête de la machine lors de la mise à jour de son état interne. La Figure 2-3 montre cette modélisation.

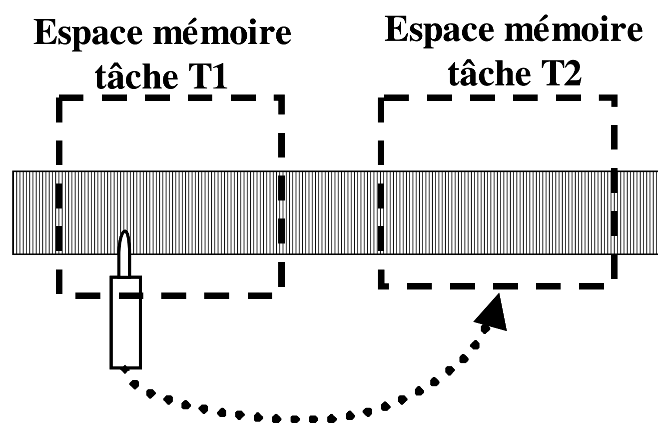


Figure 2-3 Changement de contexte dans une machine de Turing

La formule (2) est donc valide parce que l'espace mémoire M_{Ta} n'a pas été utilisé dans l'intervalle $[t_s, t]$ et donc $M_{Ta}(t)=M_{Ta}(t_s)$.

La similarité entre CdC et C&R est frappante : tous deux font des sauvegardes périodiques de l'état de la tâche pour la restaurer plus tard. Les différences principales sont :

- les données sauvegardées ;
- la procédure de restauration ;
- le déclencheur de la restauration.

Les formules (1) et (2) montrent les différences pour les deux premiers points : on comprend tout de suite qu'un C&R est beaucoup plus lourd que le CdC parce qu'il nécessite la sauvegarde/restauration de beaucoup de données et donc de nombreux accès mémoires et une taille de la mémoire de sauvegarde en conséquence.

Le **déclencheur** pour un C&R est le **test d'acceptation**, alors que pour le CdC il s'agit en général d'un compteur qui génère un **signal** quand le temps imparti à la tâche courante est épuisé.

L'idée proposée consiste à exploiter les similarités pour obtenir un schéma de C&R qui utilise le CdC. La stratégie résultante est illustrée dans la Figure 2-4.

La sauvegarde du checkpoint est remplacée par le dernier contexte enregistré pour la tâche et en cas de faute le système d'exploitation effectue un changement de contexte vers la

valeur sauvegardée. Cela permet à la tâche, sous certaines contraintes qui seront analysées dans le paragraphe suivant, de redémarrer l'exécution et donc de corriger l'exécution erronée si la faute était transitoire. Notons qu'il ne s'agit pas ici de garantir une reprise parfaite du fonctionnement dans tous les cas possibles, mais d'essayer d'atteindre un bon compromis entre le niveau de protection et les coûts induits.

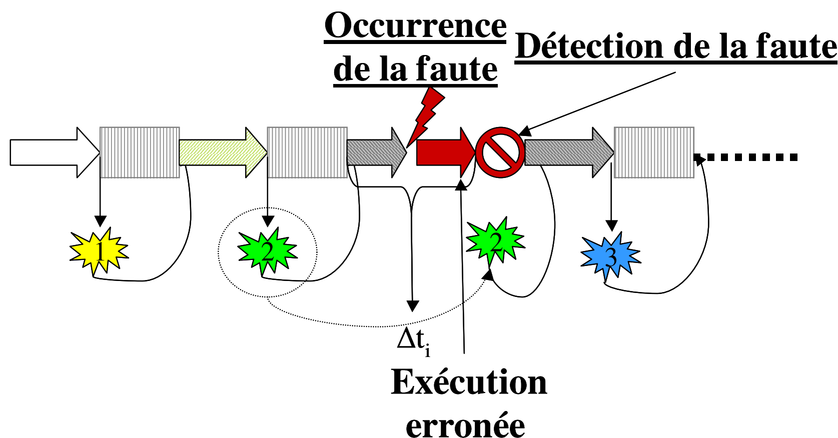


Figure 2-4 Checkpoint & Rollback basé sur le changement de contexte

Dans la Figure 2-4, il manque le test d'acceptation comme déclencheur : ce test est remplacé par les indications données par les protections implantées au niveau matériel et logiciel et décrites dans la suite du chapitre. Il n'y a donc pas besoin de développer des tests d'acceptation ad hoc, avec un grand gain en termes de généralité et de temps de développement. Par souci de lisibilité, toutes les grandeurs en jeu ne sont pas indiquées sur la figure. Elles sont résumées ci-dessous :

- on rappelle que t_s est l'instant du dernier changement de contexte ;
- le timeslice est le temps entre deux changements de contexte ;
- T_o indique le temps passé entre t_s et l'occurrence de la faute ;
- T_l indique la latence de détection de la faute ;
- Δt_i indique le temps passé entre t_s et la détection : $\Delta t_i = T_o + T_l$.

2.2.2 Contraintes et limitations

La technique proposée est une simplification du C&R et donc il faut définir les hypothèses sous lesquelles elle est valable.

Avant tout pour que ce schéma fonctionne, il faut absolument que la détection d'une faute survenue pendant l'exécution de la tâche T_i se fasse avant le changement de contexte suivant, sinon le contexte sauvegardé sera lui même erroné et on ne pourra plus faire une reprise correcte. On a donc besoin de détections très rapides à l'échelle de la durée d'exécution d'une tâche, c'est à dire qu'il faut T_l << timeslice. De ce point de vue, des protections matérielles sont très adaptées parce que très rapides. Si un timeslice peut être composé de

quelques centaines, voire milliers, d'instructions, une détection matérielle peut avoir au plus un retard de quelques cycles d'horloge. Dans le cas d'une détection réalisée dans le logiciel d'application, il faut s'assurer que les tests permettant la détection d'une erreur dans le calcul en cours sont effectués avant tout changement de contexte.

L'autre ensemble de contraintes vient du fait que, pendant son exécution, la tâche Ta est sensée faire des calculs et modifier son espace mémoire, donc a priori $M_{Ta}(t) \neq M_{Ta}(t_s)$. Il est cependant possible de définir des **critères d'application** pour lesquels l'approche proposée reste valable :

1) $\Delta t = t - t_s$ est suffisamment « petit » par rapport aux propriétés temporelles de la tâche, qui n'a donc pas pu « beaucoup » modifier $M_{Ta}(t_s)$;
2) La tâche n'est pas « très active » en mémoire pendant Δt ;
3) La variation ΔM_{Ta} de la mémoire ne « perturbe pas trop » le fonctionnement du système.

On peut remarquer que les trois critères permettent de définir des conditions pour lesquelles $M_{Ta}(t) \approx M_{Ta}(t_s)$, mais à partir de points de vue différents. Les sous paragraphes suivants analysent en détail ces critères, leurs particularités et les avantages de chacun.

2.2.2.1 Premier critère

Le premier critère est le plus simple et le plus immédiat : si la tâche n'a pas beaucoup modifié la mémoire (voire ne l'a pas modifié du tout) alors on peut la faire repartir du dernier changement de contexte en perdant peu d'information. La difficulté reste la quantification et l'identification précise des différences pouvant effectivement être tolérées par l'application (en lien avec le troisième critère). En effet, une modification unique dans une variable critique peut dans certains cas interdire une reprise correcte. Avoir peu de modifications n'est donc pas suffisant, il faut aussi que ces modifications correspondent à des données ayant peu de conséquences sur le déroulement de la tâche.

Une première façon de mettre ce critère en application serait de choisir des timeslices très petits pour limiter le nombre de modifications entre deux sauvegardes. Cependant cela est contraire aux règles d'optimisation du système d'exploitation : des timeslices trop petits impliquent beaucoup de changements de contexte, avec donc beaucoup de temps passé en opérations systèmes plutôt qu'en opérations utiles.

Il existe toutefois des tâches pour lesquelles ce critère est respecté naturellement : les tâches avec interaction humaine. La vitesse de réaction d'un être humain est beaucoup plus lente que celle des systèmes électroniques, donc les systèmes d'interface homme-machine passent beaucoup de temps en attente d'entrées. Des fautes intervenant pendant ces phases d'attente peuvent être tolérées à travers une reprise basée sur un CdC.

2.2.2.2 Deuxième critère

Le deuxième critère part du point de vue de la mémoire : la tâche a bien effectué ses opérations à partir des entrées, mais n'a pas effectué d'écritures en mémoire, ou elle n'en a pas fait beaucoup. $M_{Ta}(t_s)$ n'a donc pas été sensiblement modifiée et un retour à la valeur sauvegardée ne perdrait que peu de données.

Ce critère est très intéressant parce que l'on peut le renforcer en exploitant la hiérarchie mémoire des processeurs contemporains, présentée dans la Figure 1-5. Le processeur n'est pas directement lié à la mémoire centrale (MC), mais juste à une copie locale, la mémoire cache. Chaque fois que l'on a besoin de modifier une valeur en mémoire on modifie sa valeur en cache et cette modification est reportée en MC suivant la politique de cohérence utilisée pour le cache. On rappelle que les deux politiques typiques sont le « write-through » (écriture à travers : chaque modification en cache est directement reportée en MC) et le « write-back » (écriture différée : la MC est mise à jour à certains instants). Si on considère un cache en write-back, un « périmètre de modification » peut être défini entre deux mises à jour du cache, comme illustré en Figure 2-5.

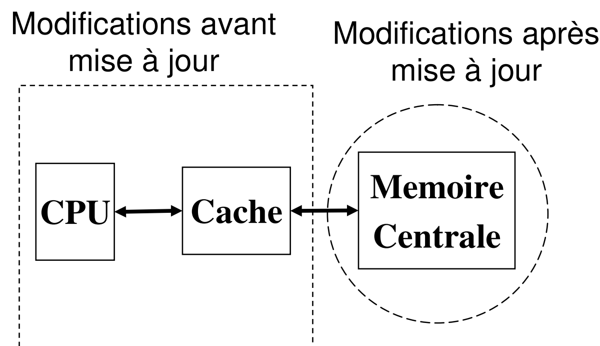


Figure 2-5 Périmètre de modification des données avec un cache "write-back"

Avec deux niveaux de mémoire, on peut appeler M_{Ta} la zone mémoire de la tâche en cache et MC_{Ta} celle en mémoire centrale. Entre deux write-back (WB) seulement M_{Ta} change, MC_{Ta} reste constante. On peut donc concevoir une approche mixte caches - système d'exploitation comme suit :

- les mises à jour de la MC sont interdites pendant l'exécution d'une tâche. Ceci suppose que le cache peut contenir tous les blocs mémoire pouvant être modifiés pendant chaque timeslice d'exécution de la tâche ou qu'en cas de défaut de cache nécessitant de libérer un emplacement modifié, le contrôleur de cache demande un CdC au système d'exploitation pour définir un nouveau point de reprise ;
- le système d'exploitation demande une sauvegarde des informations modifiées dans la cache à chaque changement de contexte ;
- en cas de rollback, la cache est marquée comme invalide et rechargée à partir de la MC.

Cette stratégie mixte est très intéressante car elle ne demande pas beaucoup de ressources ou de temps additionnel. L'opération la plus longue est la mise à jour de la cache en cas de rollback, surtout si elle doit être faite entièrement. Une optimisation importante pourrait venir d'une invalidation sélective qui ne concernerait que M_{Ta}.

Il faut quand même souligner que cette technique agit directement sur la politique de gestion des caches, ce qui peut avoir un effet important sur les performances statistiques du système. Un prototype est donc très important pour étudier ces variations dans en environnement réel.

Il faut souligner que cette stratégie de relaxation du critère n'est pas applicable au cas spécifique de Leon2 : le processeur a en effet une cache implantée en write-through (voir

paragraphe 1.2.2). Cela signifie qu'à tous moments le contenu de la cache données et de la mémoire centrale sont synchronisés. Pour ces types de processeur l'application de ce critère est donc soumise exclusivement aux caractéristiques de l'application logicielle.

Notons aussi que dans une architecture de style Harvard seule la mémoire cache des données est concernée puisque les instructions ne sont jamais modifiées par le processeur. Seule une détection d'erreur directement dans le cache des instructions peut nécessiter une invalidation à ce niveau.

2.2.2.3 Troisième critère

Le troisième critère est basé sur le point de vue de la fonctionnalité de la tâche exécutée : si la variation ΔM_{Ta} n'est pas trop importante sur le plan fonctionnel, la tâche peut quand même fournir un résultat acceptable pour l'utilisateur. Un système embarqué doit souvent acquérir des données, les élaborer et présenter les résultats en sortie, par exemple dans les applications de traitement de signaux. Si une faute arrive à perturber les valeurs des calculs le résultat va être une déviation de courte durée par rapport à la sortie de référence. Les applications de ce type ont souvent une tolérance intrinsèque à de telles déviations donc, sauf pour les cas les plus critiques, on peut espérer respecter les spécifications. Cela peut être le cas pour un pixel de couleur erronée dans une image haute définition, ou un signal de parole brièvement perturbé pendant une communication suite à des données erronées pendant la phase de reprise. La Figure 2-6 montre un exemple pour une telle application de traitement de signal.

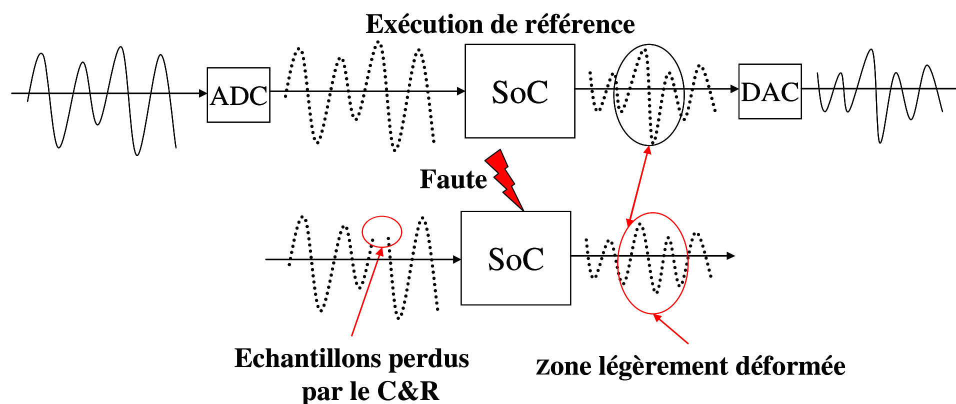


Figure 2-6 Exemple de déviation par rapport à la référence dans une application de traitement de signal

Notons aussi que les calculs sont souvent réalisés de façon itérative avec des cycles imbriqués, approche plus prédictive et moins gourmande en ressources que, par exemple, une approche récursive. Si la perte d'information en mémoire perturbe les variables de contrôle, le risque est de sauter, couper ou prolonger certains calculs : l'effet est de nouveau une déformation de la sortie et on retombe donc dans le cas précédent.

Dans la Figure 2-6 on peut remarquer une autre particularité de l'approche : la perte de valeurs d'entrée pendant le recouvrement se traduit aussi par une déformation de la sortie, au même titre que des données erronées dues à la sauvegarde partielle des informations de reprise. La déformation induite peut être estimée à partir des caractéristiques de l'algorithme de calcul. Il faut aussi souligner que la fréquence de travail des systèmes embarqués est

souvent sensiblement supérieure à la vitesse de modification de l'environnement, donc le retard dû à un recouvrement peut être négligeable.

L'application de traitement d'image proposée dans le paragraphe 2.1.3 est une excellente candidate pour tester ce critère.

2.2.3 Les communications inter - tâches

Si autrefois les systèmes embarqués étaient simples et chaque tâche s'exécutait indépendamment, la complexité des systèmes d'exploitation embarqués modernes permet d'imaginer un scénario composé de nombreuses tâches qui communiquent et se synchronisent entre elles.

Il y a 3 types principaux de communications entre tâches :

- sémaphores : ils servent à limiter l'accès à une certaine ressource à un nombre donné d'utilisateurs n. Une tâche voulant accéder à cette ressource demande la permission au sémaphore (opération « p »), et reste bloquée dans une file d'attente si le quota d'accès est atteint. Quand une tâche libère la ressource elle le communique au sémaphore (opération « v »), qui s'occupe de débloquer une éventuelle tâche en file d'attente.
- mutex : ils servent à assurer un accès exclusif à une ressource, et sont donc assimilables à des sémaphores avec n=1 ;
- boîtes aux lettres : elles servent à passer des messages entre tâches. T_e dépose un message dans la boite où T_r peut le récupérer. Si la boite est pleine, T_e est bloquée en attendant qu'une place se libère, et si elle est vide c'est T_r qui est bloquée en attente d'un message.

Toutes les structures citées peuvent être aussi utilisées pour synchroniser des tâches, en les faisant bloquer sur un sémaphore ou une boîte aux lettres en attendant que les conditions de synchronisation soient atteintes. Pour plus de détails sur ces structures, le lecteur peut se référer à [Tanne87].

Les communications inter - tâches sont un des points sensibles des schémas de C&R : si une tâche T_e a envoyé un message à une autre tâche T_r, T_r risque de recevoir le message en double (ou un message différent) après un rollback de T_e. Pire encore, T_r pourrait avoir reçu et « consommé » le message, qui ne serait donc plus disponible si c'est T_r qui doit exécuter une reprise. Le tout peut se traduire par un effet en chaîne si T_r contacte une autre tâche, qui fait passer le message et ainsi de suite. On appelle souvent cela un « effet domino » et si le temps entre deux checkpoints est important le problème risque de devenir très compliqué et a été analysé en détail, par exemple dans [Elnoz02].

Les sémaphores peuvent aussi être un problème : si T_a demande l'accès à un sémaphore et l'obtient, si elle est redémarrée elle va refaire la même demande. Le sémaphore va enregistrer deux accès et donc le compte du nombre d'utilisateurs est faussé. Dans le cas d'un mutex (1 seul utilisateur) la ressource serait directement perdue.

La méthode de C&R que l'on propose a un grand avantage : l'impossibilité d'un effet domino. En effet, en cas d'erreur, la tâche T_e est arrêtée avant son changement de contexte, et donc T_r ne s'exécute pas, coupant tout de suite la propagation.

Pour les autres problèmes, on propose une solution basée sur une **signature** : en cas de rollback, cette signature permettra de détecter un appel qui a été dupliqué. On va donc définir une variable appelée « timeslice_coms » qui est incrémentée pour chaque communication, couplée avec une variable booléenne « rollback_flag » qui est vraie si on est en train d'effectuer un rollback. Chaque fonction va garder en mémoire les « mem_max » dernières opérations.

L'algorithme va être le suivant :

- à chaque changement de contexte timeslice_coms=0 et rollback_flag=faux ;
- si on a un rollback rollback_flag=vrai (et restera donc vrai jusqu'au CdC suivant) ;
- Pour sémaphores ou mutexes :

```
        i=1
        Si (rollback_flag=vrai) et (timeslice_coms >=i)
            Ne fais rien ;          [il ne faut pas modifier le compteur]
            i=i+1
        Sinon
    Exécution normale
    timeslice_coms++
```

- Pour boîtes aux lettres avec « courant » messages:

```
        i=1
        Si (rollback_flag=vrai) et (timeslice_coms >=i)
            Si dépose d'un message
                Ne fais rien            [il est déjà là]
            Si réception d'un message
                Envoie le message (courant-i) [même réponse qu'avant le
    rollback]
            i=i+1
        Sinon
    Exécution normale
    timeslice_coms++
```

Ces algorithmes permettent de masquer les répliques d'accès et donc permettent à la tâche de recevoir les mêmes réponses qu'avant le rollback. Il faut quand faire attention qu'une boîte aux lettres efface les messages après leur réception, donc pour que les algorithmes marchent il faut les garder en mémoire : mem_max correspond donc aux nombre de messages gardé en mémoire après leur réception. Si on ne peut pas définir une limite, il est possible d'utiliser une implantation mémoire à base de listes chaînées, qui est toutefois assez lourde en termes d'espace mémoire et de performances.

2.2.4 Fonctionnalités avancées du système d'exploitation

Le mécanisme d'ordonnancement illustré dans la Figure 1-6 est à la base de tout système d'exploitation, mais bien sûr il a été décliné en beaucoup de versions suivant les besoins des applications. Une des modifications les plus importantes est dans l'introduction de **priorités** : chaque tâche a une certaine valeur de priorité associée, qui est exploitée par l'ordonnanceur pour décider quelle tâche est à exécuter. En plus beaucoup d'OS (par exemple eCos, Linux et Windows XP) permettent la **préemption** : si une tâche A est en cours d'exécution, le système d'exploitation peut à tout moment la suspendre pour exécuter la tâche B plus prioritaire. Cela permet, par exemple, d'éviter le paradoxe d'une tâche prioritaire suspendue pour laisser exécuter une tâche moins prioritaire. Une préemption peut se résumer dans un changement de contexte forcé par l'OS : le schéma de protection présenté dans les paragraphes précédents est donc compatible avec ce mécanisme.

La gestion des tâches est l'un des devoirs les plus importants d'un système d'exploitation, mais pas le seul. Une autre catégorie de services est représentée par les **appels système** : certaines opérations, par exemple les entrées/sorties, sont trop complexes ou trop bas niveau pour être traitées par les tâches. Le programmeur a donc la possibilité de demander au système d'exploitation de les effectuer de façon autonome et optimisée et se contenter de récupérer les résultats. Cela se traduit dans une exécution système, qui n'est pas couverte par le schéma de C&R présenté ici. Deux scénarios sont possibles :

1. faute pendant un appel système : un recouvrement peut risquer de déstabiliser le système. Il serait donc préférable de masquer les interruptions pendant l'appel et traiter la faute autrement ;
2. recouvrement qui fait ré-effectuer le même appel système : le cas peut être traité selon la stratégie des communications inter tâches du paragraphe 2.2.3.

Le choix de traiter une faute détectée en utilisant une interruption permet une grande souplesse. D'un côté on peut joueur sur le masquage/activation des interruptions pour décider quand le recouvrement est actif, comme par exemple dans le cas 1). De l'autre côté, on peut modifier le traitant d'interruption pour l'adapter à des cas particuliers si besoin.

2.3 Protection niveau matériel : approche RTL

La protection contre les fautes au niveau matériel est, comme indiqué dans le Chapitre 1, la plus ancienne et probablement la plus développée. Cependant il est peut-être réducteur et trompeur de parler de « niveau matériel » : il y a en effet beaucoup de niveaux d'abstraction possibles (niveau composants, niveau réseaux de transistors, niveau portes logiques, etc.…) et, bien sûr, les techniques de protection s'adaptent au niveau d'intervention choisi, comme expliqué dans le paragraphe 1.4.3.

Jusqu'à maintenant l'attention a été surtout pour les niveaux les plus bas (maîtrise des paramètres physiques du circuit, organisation des transistors, etc.…) ou les plus hauts (redondance de blocs, etc....) mais ces approches montrent leurs limites face aux besoins des systèmes embarqués. D'un coté les approches bas niveau tendent à être très efficaces, mais très lourdes à mettre en oeuvre sur des circuits de taille consistante, très difficiles à porter sur des technologies différentes et très rigides (il faut ré-appliquer complètement la protection pour chaque nouvelle version du circuit). De l'autre coté les approches de niveau trop élevé

sont souvent très coûteuses et difficiles à soutenir par les ressource des systèmes embarqués, souvent assez limitées.

Il y a donc un besoin de chercher des solutions intermédiaires, qui puissent atteindre un compromis différent entre les avantages et les inconvénients des différentes approches. Cette thèse applique les protections au niveau de la description comportementale du circuit (niveau RTL), qui offre plusieurs avantages :

- indépendance de la technologie cible ;
- compatibilité avec les architecture configurables, donc directement applicable à une grande gamme d'applications ;
- grandes possibilités d'optimisation car accès à la hiérarchie de conception complète et pas seulement à l'entité de plus haut niveau.

Il existe quand même des limitations, qui ont jusqu'à maintenant freiné le développement de techniques appliquées à ce niveau, liées notamment à la difficulté de les automatiser. Ces aspects on bien sûr été pris en compte et seront mieux abordés lors de la discussion des mises en œuvre dans le Chapitre 3.

Dans le Chapitre 1, on a introduit la modélisation d'effets singuliers, qui permet à la fois une analyse aisée et une grande généralité. On se focalise ici sur l'hypothèse de **fautes transitoires localisées** : on suppose qu'un événement unique va générer une seule faute, dont les effets vont rester locaux même si l'erreur peut concerner plusieurs bits. Les techniques des prochains paragraphes sont toutefois indépendantes, et peuvent bien sûr travailler de façon complémentaire et parallèle.

2.3.1 Approche micro-architecturale

Quand on parle d'une approche au niveau RTL, on a bien délimité le domaine d'intervention, mais on a encore une grande liberté pour ce qui concerne la philosophie d'intervention. La base sur laquelle travailler est composée par des fichiers sources écrits dans un langage de description matériel (VHDL pour nous) et on peut intervenir sur plusieurs aspects, depuis les éléments « basiques » (variables, signaux, etc.…) jusqu'aux niveaux supérieurs de la hiérarchie du circuit. Entre ces deux extrêmes il y a un niveau intermédiaire très intéressant : le niveau micro-architectural. On repère les blocs constituants dans le code source et, plutôt que d'intervenir à l'intérieur, on étudie leurs interconnexions et interactions.

L'approche micro-architecturale a le grand avantage d'être très générale, de permettre une bonne visualisation des modifications réalisées et de peu dépendre de l'analyse détaillée du code source, ce qui la rend beaucoup plus facile à employer. En effet chaque bloc est traité comme une « boite noire » et sa manipulation est donc simplifiée car on n'a pas besoin de se soucier de l'algorithme qu'il réalise. Cela est aussi très intéressant du point de vue de l'automatisation de la protection.

2.3.2 Redondance d'information : codage des données

Dans le chapitre 1 on a montré la variété des types de protections matérielles qui ont été développés au cours des années. Le fil conducteur de notre travail est la recherche d'un compromis optimal entre couverture des fautes, coût de l'intervention et généralité de l'approche : notre choix s'est donc porté sur la redondance d'informations.

Cette approche est très générale parce que, sauf de rares cas, elle ne dépend pas des données manipulées. Elle a un coût modéré pour une couverture des fautes donnée car il existe beaucoup de codes possibles, chacun avec ses caractéristiques bien définies, et il est donc possible de trouver la solution la mieux adaptée aux contraintes d'une application. Par rapport à la triplication/duplication, autres techniques à haut degré de généralité, la redondance d'information se révèle moins gourmande en ressources, raison pour laquelle on l'a préférée.

On a choisi dans nos implantations d'utiliser essentiellement le code de protection par parité avec décodeur en double rail (Figure 1-15): il s'agit d'une solution simple mais très représentative avec de bonnes caractéristiques de couverture et de performances. Cependant tout au long du travail on a gardé ouverte la possibilité de rajouter d'autres types de codages en faisant bien attention que le choix du code soit défini par des options de configuration.

Dans les deux sous paragraphes suivants on va détailler l'implantation pour les deux types de logiques dont tout circuit est composé : la logique séquentielle et la logique combinatoire.

2.3.3 Protection des éléments séquentiels

2.3.3.1 Protection des registres

Les éléments séquentiels sont fondamentaux pour assurer la transmission d'information entre les différents cycles d'horloge. Souvent plusieurs points de mémorisation sont regroupés ensemble dans un tableau de plusieurs bits qu'on appelle registre et dont la taille peut devenir considérable selon l'application.

Indépendamment de leur type (bascules, verrous, etc..) et de leur implantation physique, le principe de fonctionnement reste le même : à un certain instant un signal d'entrée est échantillonné et sa valeur est fournie en sortie pour être exploitée pendant une certaine durée. C'est justement dans cet intervalle de mémorisation qu'un SEU peut perturber la valeur stockée, comme on peut voir dans la Figure 2-7.

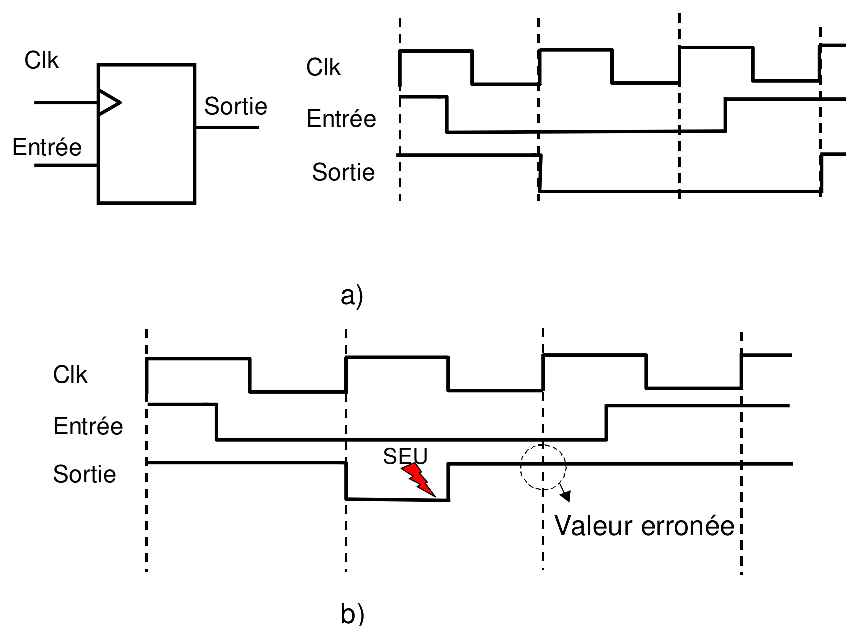


Figure 2-7 Bascule typique et son chronogramme en situation normale (a) et en présence d'une faute (b)

Pour protéger ces éléments contre les fautes transitoires à travers un codage des données il faut donc coder ces données avant stockage et vérifier le respect du code au moment de la lecture : si un SEU a eu lieu, le code n'est plus valide et il donc possible de détecter et/ou corriger l'erreur, suivant les caractéristiques du code choisi. La Figure 2-8 montre une implantation possible. Dans la figure, on a appelé R le registre à protéger, C les blocs de codage et B_{Cod} les éléments de mémorisation pour les bits de code supplémentaires. La mise en œuvre sur Leon2 et les résultats en termes de coût et de capacité de protection seront détaillés dans les Chapitres 3 et 4.

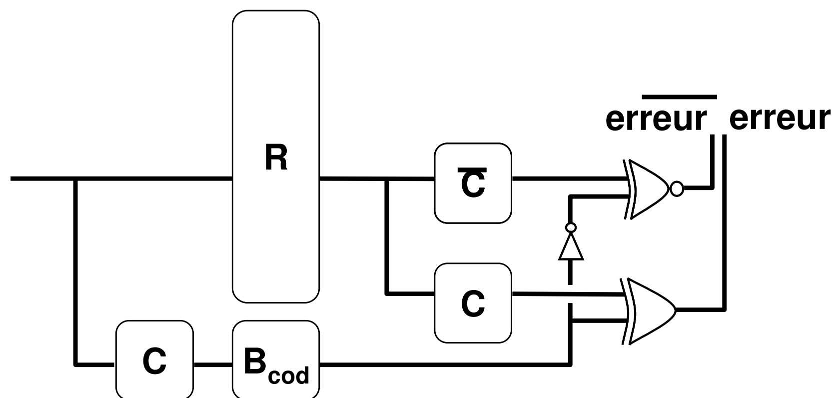


Figure 2-8 Protection du registre R à travers des blocs de codage C, avec contrôleur double rail

2.3.3.2 Protection des mémoires volatiles

L'évolution des techniques de fabrication des circuits permet désormais d'intégrer directement sur une puce des mémoires de taille conséquente (plusieurs Koctets) à accès très rapide (un ou deux cycles d'horloge). Cette innovation a eu un grand impact sur l'architecture des ordinateurs et des systèmes embarqués ainsi que sur les performances atteintes. Aujourd'hui, les mémoires embarquées rapides sont utilisées pour implanter les caches de premier niveau, mais pas seulement : quand il y a besoin d'un ensemble de points de mémorisation de grande taille, l'architecture est le plus souvent conçue de façon à pouvoir les implanter dans une RAM, éventuellement multiports.

Si on analyse cette situation du point de vue des fautes, on comprend bien que des grands bancs de mémorisation sont des éléments très sensibles aux fautes transitoires et qu'il faut donc les protéger avec des techniques adaptées. Le principe de base reste le même qu'avec les points de mémorisation classiques traités dans le paragraphe précédent, comme on peut le voir en comparant la Figure 2-9 avec la Figure 2-8.

Le coût de la protection est comme toujours représenté par les bits additionnels pour le codage, mais il peut être facilement modulé en jouant sur beaucoup de paramètres, par exemple : codage par octet, par mot de 16 ou 32 bits, par ligne, etc.… Une analyse complète de ces variations est faite pour le Leon2 dans le Chapitre 3.

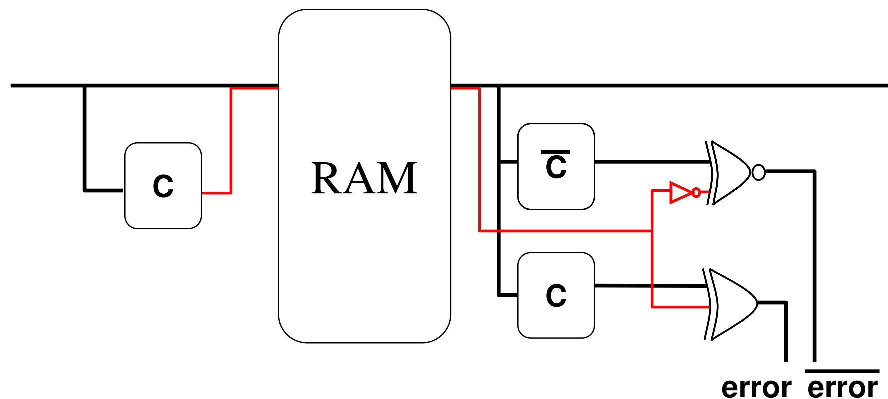


Figure 2-9 Protection d'une RAM embarquée à travers des blocs de codage C, avec contrôleur double rail

2.3.3.3 Correction des erreurs dans les caches

Les mémoires caches peuvent détecter des SEU (et même des MBF, selon le code choisi) en appliquant la technique décrite dans le paragraphe précédent. Le recouvrement peut de plus être obtenu en exploitant une caractéristique propre des caches : elles sont une copie de la mémoire centrale. Si un SEU a modifié une valeur en mémoire, il est donc possible de corriger la faute en allant chercher la bonne valeur dans la MC en forçant un défaut de cache. L'hypothèse de faute localisée assure que le même événement ne peut pas avoir corrompu les deux copies. On peut supposer en outre, dans le cas général, que la probabilité de deux évènements distincts ayant corrompu les deux copies dans l'intervalle de temps concerné est négligeable.

Pour éviter tout aléa de données il faut qu'à tout moment les données en cache et en MC soient identiques : la politique de write-through est alors obligatoire.

La version tolérante aux fautes de Leon2, le Leon2-FT ([Gaisl02]) applique cette technique, donc l'inclure dans notre prototype permettra de comparer les résultats.

2.3.4 Protection de la logique combinatoire

Les blocs logiques combinatoires sont responsables de l'exactitude des calculs et sont exposés eux aussi à des effets transitoires qui peuvent les perturber (SET). Il faut noter que les protections contre les SEUs décrites dans les paragraphes précédents ne peuvent pas suffire car le code à stocker serait calculé à partir du résultat erroné. Cependant quand on veut appliquer une protection par redondance d'information aux blocs combinatoires on se heurte à un problème important : la propagation du code à travers de la logique. Le concept sera plus clair en s'appuyant sur la Figure 2-10.

La façon la plus directe de protéger un bloc combinatoire comme celui du cas a) de la Figure 2-10 est bien sûr de coder ses entrées et de vérifier si ses sorties sont toujours des mots du code, comme indiqué dans le cas b). Cependant le gros problème est que les entrées codées ne peuvent généralement pas être traitées par le bloc d'origine, qui doit donc être transformé pour opérer sur des mots du code. Cette opération est très difficile voire impossible dans certains cas, même s'il existe des exceptions remarquables (codes arithmétiques, codes résidus, etc..).

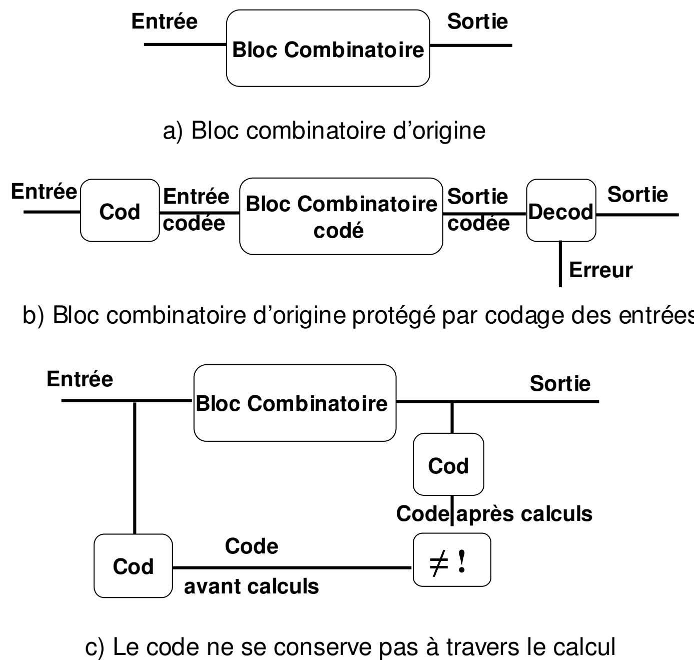


Figure 2-10 Exemple de protection d'un bloc combinatoire par redondance d'information

De plus les codes ne se conservent généralement pas à travers un calcul : comme indiqué dans le cas c) les codes avant et après calcul ne coïncident pas, sauf dans des cas très particuliers. Il est cependant possible d'implanter une **prédiction de code** : connaissant la fonction du bloc combinatoire, un composant spécifique prédit, à partir des valeurs d'entrée, la valeur des bits de code de sortie pour pouvoir faire une comparaison avec les bits calculés à partir de la vrai sortie. Le point clé de cette technique est justement le développement du bloc de prédiction de code, dont l'obtention des équations est une opération souvent longue et délicate. Il existe toutefois une possibilité d'automatisation, présentée dans [Ko04] pour la parité et bien adaptée à une intervention au niveau micro-architectural. Le principe est schématisé en Figure 2-11.

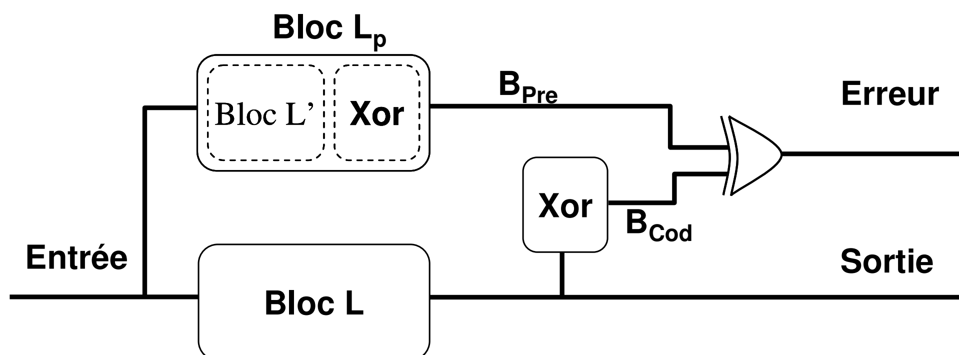


Figure 2-11 Protection par prédiction de parité automatisée

Le composant de prédiction de parité est construit en dupliquant le bloc original L et en ajoutant l'arbre de Xor qui calcule la parité. Le bloc L_p ainsi obtenu est ensuite simplifié par un outil de synthèse. Une seule sortie étant à générer (le bit de parité), les équations correspondant à l'ensemble des sorties du bloc L peuvent en général être notablement

minimisées même si certains blocs (comme par exemple les blocs arithmétiques) peuvent se prêter moins bien à une telle optimisation. [Ko04] démontre que cette optimisation va être en général très importante ; l'occupation en surface devrait donc être bien plus basse que pour une duplication simple, et la perte de vitesse va être aussi très limitée car L_p calcule en parallèle de L et seul l'arbre de vérification de parité peut affecter le chemin critique.

Pour développer une stratégie générale de protection contre les SETs par redondance d'information on s'est donc inspiré de cette technique de prédiction de parité, mais en la généralisant pour n'importe quel code et avec un contrôleur auto-testable implanté en double rail, comme illustré en Figure 2-12.

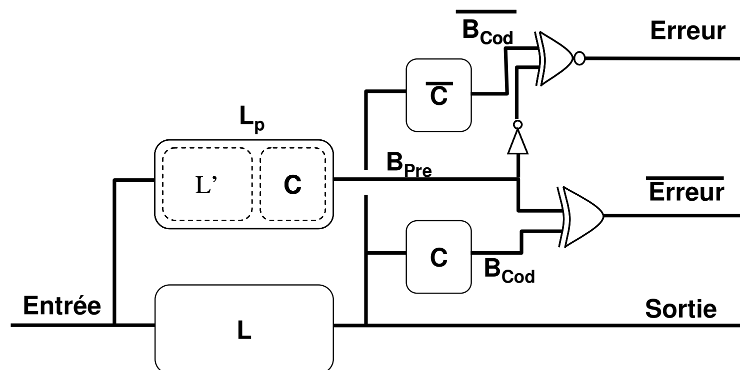


Figure 2-12 Protection par prédiction de code générique avec contrôleur double rail

A part le contrôleur, la différence la plus significative est la généralisation des arbres de Xor vers des blocs de codage génériques C. Même si au premier regard cela ne semble pas trop important cela a un effet très fort sur la stratégie d'implantation (détaillée dans le chapitre 3) et sur le plan théorique : ce principe n'est pas seulement applicable à la parité, mais à n'importe quel codage, y compris les codes non séparables. Une telle généralisation implique aussi l'impossibilité d'assurer un bon degré d'optimisation a priori dans tous les cas, mais le concepteur peut facilement vérifier l'efficacité pour un code donné en réalisant une synthèse et il est facile, si nécessaire, de modifier le bloc de codage pour mieux l'adapter aux contraintes de l'application.

2.3.5 Cas d'un pipeline : détection et recouvrement

Le pipeline occupe une place très importante dans les architectures de circuits. Le principe a déjà été introduit dans le paragraphe 1.2.1 pour les processeurs mais il est appliqué aussi pour tous les circuits devant faire des opérations complexes tout en assurant un débit moyen important. Par exemple, presque tous les composants effectuant des calculs à hautes performances en virgule flottante sont implantés avec un pipeline.

La caractéristique principale de ces composants est d'avoir la logique combinatoire partitionnée sur plusieurs cycles grâce à l'introduction de registres internes qui permettent de « couper » les opérations, avec un grand gain en termes de chemin critique. Le schéma présenté dans le paragraphe précédent est applicable directement à chaque étape du pipeline, et le résultat est montré dans la Figure 2-13.

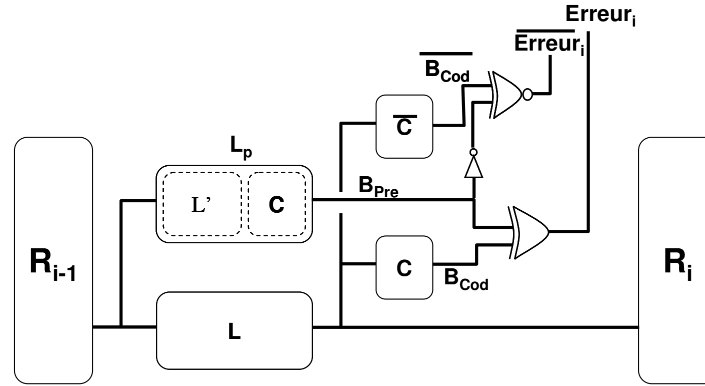


Figure 2-13 Protection par prédiction de code d'un étage i d'un pipeline

On obtient un signal d'erreur (plus son inverse) pour chaque étage du pipeline, ces signaux pouvant être traités séparément ou remontés dans la hiérarchie du circuit pour obtenir un signal d'erreur global pour le pipeline ou le composant entier.

La Figure 2-13 montre aussi très bien que les entrées d'un étage générique « i » proviennent d'un registre, et que ses sorties sont elles aussi stockées dans un autre registre. Il est donc possible de développer une stratégie de recouvrement qui joue sur la mise à jour des registres du pipeline : en cas d'erreur transitoire détectée, les registres maintiennent leur valeur au lieu de se charger normalement et au cycle suivant la logique va donc ré-exécuter le même calcul et corriger l'erreur. La Figure 2-14 montre ce schéma, appelé dans la littérature « pipeline freezing » (congélation ou gel du pipeline). Comme dans le cas des OS, il s'agit ici de ré-utiliser des mécanismes existant puisque le gel du pipeline est prévu dans la plupart des architectures afin de permettre par exemple la gestion d'aléas liés au parallélisme.

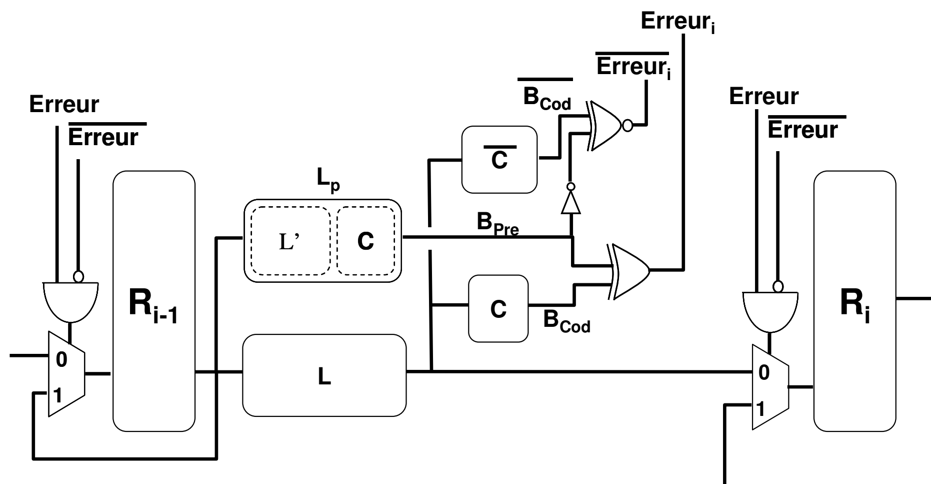


Figure 2-14 Recouvrement par prédiction de code et gel du pipeline

La technique s'applique directement pour tous les étages du pipeline, sauf pour le premier et le dernier, qui ne sont pas compris entre deux registres : il faudra ajouter des mémorisations en entrée et en sortie du composant pour compléter la symétrie. On va donc s'attendre à des coûts en surface supplémentaires, ainsi qu'à une dégradation de vitesse plus importante que pour la simple détection : le signal de validation de la mise à jour des registres dépend du signal d'erreur global, qui se trouve donc sur le chemin critique.

Naturellement, les registres internes du pipeline peuvent être protégés également contre les SEU comme expliqué dans le paragraphe sur les registres.

Une fois encore la mise en œuvre et les résultats sont détaillés dans les Chapitres 3 et 4.

Il faut souligner que les protections proposées dans ce paragraphe ne sont pas dépendantes de l'architecture du processeur : n'importe quelle architecture à base de pipeline peut être protégée de la même façon. Le gel du pipeline ("freezing") est compatible avec toutes les techniques de résolution d'aléas d'exécution parce qu'il se limite à répliquer un cycle, comme si le précédent n'avait pas eu lieu. Des processeurs avec une gestion d'aléas plus avancée que le Leon2 (par exemple avec l'insertion de bulles) peuvent soit implanter le gel du pipeline, soit obtenir le même effet avec des techniques plus adaptées à leur architecture spécifique.

2.3.6 Redondance temporelle : protection de l'IP AES

Dans le paragraphe 2.1.3 on a introduit l'IP de cryptographie AES qui est incluse dans le prototype, pour répondre aux besoins de confidentialité dont on a parlé dans le paragraphe 1.5.

En ce qui concerne la protection de sa logique séquentielle on a appliqué la même méthode que pour le Leon2, comme décrit dans le paragraphe 2.3.3.

En ce qui concerne la protection des parties combinatoires, les spécificités d'un algorithme comme l'AES permettent d'appliquer une technique de redondance temporelle. Dans le paragraphe 2.1.3, on a mis en évidence la symétrie de cet algorithme : pour chaque ronde de calcul il est possible de faire un calcul inverse pour retourner à la valeur originale. Il est donc possible d'exploiter cette caractéristique pour obtenir le schéma de redondance temporelle avec calcul inverse montré dans la Figure 1-18.

On peut s'attendre à une bonne capacité de protection contre les fautes simples et multiples avec un coût relativement faible en terme de surface et de fréquence maximale. Par contre le nombre de cycles nécessaire pour exécuter un calcul va être sensiblement doublé. Dans le Chapitre 4, nous analyserons les résultats obtenus, en choisissant cette approche pour l'AES afin d'avoir au moins un représentant de chaque type de méthode de protection.

2.4 Protection niveau logiciel

Dans le chapitre 1 on a montré que différentes techniques de tolérance aux fautes logicielles ont atteint un bon niveau de maturité. Dans le cadre de cette thèse, nous n'avons pas cherché à innover sur ce point : sans la possibilité d'y consacrer un temps suffisant, les résultats n'auraient pas pu être à la hauteur des techniques existantes. La décision a donc été de collaborer avec le groupe du professeur Sonza Reorda du Politecnico di Torino pour durcir l'une des applications logicielles développées pour le prototype, à savoir l'algorithme AES.

A partir du code Open Source préalamblement mentionné, l'équipe du Politecnico a pu nous fournir rapidement une version durcie selon les techniques décrites dans le paragraphe 1.4.5 de façon totalement automatique grâce à leur outil. Implantée dans le système, cette version nous permettra de comparer le comportement envers les fautes du même algorithme protégé grâce à des techniques complètement différentes et de démontrer la possibilité, pour la technique proposée au niveau OS, de prendre en compte des alarmes générées à la fois par le matériel ou le logiciel.

Elle sera aussi l'occasion de tester les capacités de couverture de cette technique de protection purement logicielle avec un modèle de fautes très différent de celui utilisé pour son développement et sa validation.

2.5 Coopération entre niveaux

Les protections présentées dans les paragraphes précédents peuvent être mises en collaboration pour combler les points faibles respectifs. Pour résumer :

- la protection du pipeline de l'unité entière traitée dans le paragraphe 2.3 offre de bonnes capacités de détection, mais le recouvrement par gel du pipeline est sensiblement plus coûteux et ne peut fonctionner que pour les SET dans la logique combinatoire, pas pour des SEU directement dans les registres si un code correcteur n'est pas employé ;
- le schéma de C&R utilisant le changement de contexte décrit dans le paragraphe 2.2 offre un bon degré de protection à coût sensiblement plus bas que le C&R classique, mais sa faisabilité est soumise à des critères qui peuvent être difficiles à vérifier pour certaines applications ;
- le recouvrement dans les caches est très performant et à coût très limité, mais est incompatible avec les critères d'application du C&R au niveau de l'OS.

Le C&R géré par le système d'exploitation est très rapide et léger parce qu'il exploite les informations déjà sauvegardées lors du changement de contexte. Sa principale limitation vient des valeurs modifiées en mémoire dans l'intervalle de temps compris entre le dernier changement de contexte et la détection de la faute. Il est a priori impossible de savoir quelles valeurs vont être modifiées, sauf si on demande une intervention directe du programmeur, ce que nous cherchons à éviter afin d'avoir une technique transparente du point de vue de l'application.

Toutefois, la connaissance des mécanismes mis en œuvre dans le matériel peut venir à notre aide. Avant d'arriver à la mémoire centrale, toute donnée modifiée passe par la mémoire cache de données. Il est donc possible de tracer tous les accès de façon dynamique, et de savoir quand et où la mémoire a été modifiée. Si on reprend la notation du paragraphe 2.2.1, pour que le recouvrement de la tâche T_a soit garanti correct il est nécessaire de reconstruire son état mémoire au moment du changement de contexte, $M_{Ta}(t_s)$. Si $M_{Ta}(t)$ est son état mémoire au moment de la détection de la faute il est possible en ayant observé les accès mémoire de connaître tous les emplacements $M_{Ta}[i]$ où il y a eu au moins une écriture dans l'intervalle $[t_s,t]$. Il n'y a pas besoin de faire $M_{Ta}(t) <= M_{Ta}(t_s)$ (restauration de tout l'état mémoire à l'instant t_s), il suffira de restaurer seulement les n emplacements effectivement modifiés : $M_{Ta}[i](t) <= M_{Ta}[i]\ (t_s)$, i=0,1,…,n-1. On notera que seules les valeurs à l'instant t_s sont significatives : si un emplacement est modifié plusieurs fois dans l'intervalle $[t_s,t]$ il n'est pas nécessaire de garder un historique complet, ce qui simplifie énormément le processus.

Dans le paragraphe 2.2.2.2 on a montré comment les mises à jour dues aux défauts de cache peuvent être parfois problématiques : dans le cadre de l'analyse qui vient d'être introduite, une telle action est équivalente à une écriture mémoire. Il suffira de sauvegarder et restaurer la valeur à l'instant t_s pour reconstruire $M_{Ta}[i]\ (t_s)$.

La proposition est donc de coupler la mémoire cache de données avec une mémoire cache fantôme, que l'on appellera « mémoire des victimes modifiées» (« Dirty Victim Cache », DVC) pour assonance avec la littérature [Zhang04], selon le schéma de la Figure 2-15.

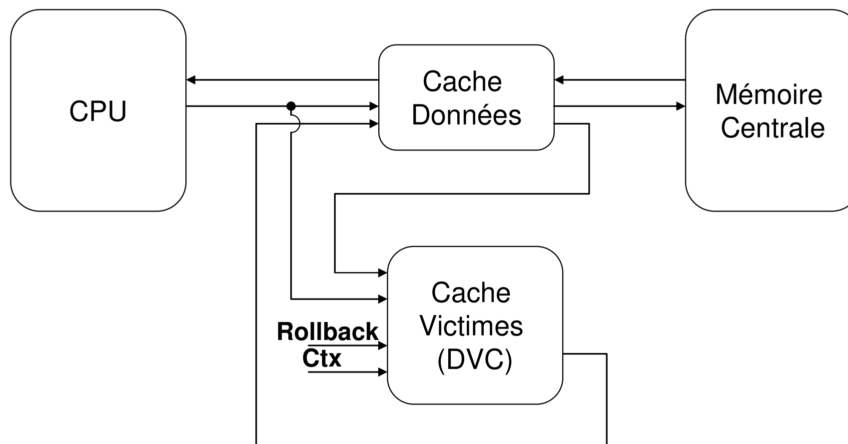


Figure 2-15 Schéma de base d'implantation d'un cache des victimes modifiées

La DVC est une mémoire cache équivalente à la cache de données CD (taille, associativité, etc.…) et analyse les transactions CPU-cache et cache-MC. La première écriture à l'emplacement CD[i] est facilement détectée en ajoutant au champ tag de DVC[i] un bit D (« dirty », modifié) qui est mis à 1 lors de l'écriture. L'algorithme final est donc le suivant :

- si écriture CPU-CD à l'emplacement CD[i] et DVC[i](D)=0 alors sauvegarder la valeur précédente en DVC[i] et mettre DVC[i](M)=1.
- si cache miss à CD[i] et CD[i] déjà occupé, alors sauvegarder la valeur précédente (ou le bloc entier, selon l'algorithme de remplacement) en DVC[i] et mettre DVC[i](M)=1.

Le signal « Ctx » sur la figure est le même signal que celui déclenchant le changement de contexte et son rôle est de remettre tous les bits D de DVC à 0. L'effet est celui du checkpoint, parce que cela définit l'état mémoire à cet instant là comme étant l'état sauvegardé.

Le signal « Rollback » indique le besoin d'effectuer un recouvrement : en cas de détection matérielle il peut être directement le signal d'erreur. Le recouvrement est obtenu en recopiant en cache de données toutes les cases modifiées dans DVC et donc en reconstituant complètement $M_{Ta}(t_s)$.

Cette technique permet au matériel de résoudre de façon optimale la faiblesse du C&R pae CdC modifié, en effectuant une sauvegarde mémoire dynamique. Théoriquement cette approche collaborative permet donc d'atteindre le niveau de sécurité d'un C&R classique : toute faute détectée peut être corrigée. Le seul problème vient de l'extérieur : comme tous les C&R, le schéma risque de ne pas respecter les contraintes temps réel si elles sont trop strictes. Cependant ce problème n'est pas trop contraignant parce qu'il est toujours possible d'appliquer des techniques d'ordonnancement adaptées pour le C&R classique ([Punne01] , [Sang05]) avec toujours un temps de recouvrement bien inférieur que pour des C&R classiques. Il faut aussi noter qu'un système temps réel très strict ne va pas utiliser un ordonnancement trop libre pour être sûr de respecter les contraintes, et va donc sortir du cadre d'analyse de ce travail de thèse. Il est aussi toujours possible d'appliquer le troisième critère de validité (paragraphe 2.2.2.3).

Un dernier point très intéressant de la technique proposée est son ouverture aux optimisations :

- la DVC peut être implantée comme une table de hachage pour réduire sa taille, éventuellement au prix d'un léger ralentissement ;
- à la place de recopier les valeurs modifiées lors d'un rollback on peut prévoir un multiplexage entre cache et DVC, qui déroute les accès vers DVC si besoin. Le seul problème serait que la mémoire centrale ne serait pas mise à jour.

2.6 Conclusions

Dans ce chapitre on a profité de l'analyse de l'état de l'art pour définir un exemple représentatif de système embarqué avec capacités cryptographiques. Le choix d'utiliser majoritairement des composants de base Open Source a été payant car il a permis d'obtenir un système complet d'un bon degré de complexité et qui pourrait très bien faire partie d'un produit réel. Si on avait décidé de tout développer de zéro, on aurait peut être eu plus de choix au début, mais on aurait été forcé à simplifier énormément le système, en se limitant par exemple à un processeur à fonctionnalités réduites et un système d'exploitation basique, sans parler du flot de conception logiciel (compilateur, debugger, éditeur de liens, etc....), qui aurait été encore plus difficile à développer. Grâce à ce choix on a au contraire un processeur 32 bits pipeline qui est à la base de plusieurs produits de pointe, équipé d'un système d'exploitation complet et très répandu, qui utilise le flot de conception logiciel le plus utilisé au monde.

Les licences Open Source nous permettent d'analyser et de modifier le système dans tous ses détails sans être gêné par des copyrights, en obtenant donc un outil très puissant pour nos expérimentations.

Nous avons ensuite proposé des techniques de protection à tous les niveaux du système. Certaines de ces techniques existaient et on fait l'objet de collaborations. Toutefois, nous avons proposé une technique novatrice au niveau OS et une nouvelle approche pour implanter des protections matérielles efficaces et particulièrement flexibles. La solution proposée dans le paragraphe 2.5 est particulièrement à remarquer, la collaboration entre niveaux permettant d'obtenir une technique puissante et originale, beaucoup trop complexe à réaliser seulement en logiciel ou en matériel.

Dans tous les cas, les blocs de base, les algorithmes et les techniques d'intervention proposées ont été choisis dans un souci de généralité et de réutilisation.

3 Implantation pratique des approches proposées

Dans le Chapitre 2 on a présenté les composants du système cible et les principes généraux des protections que nous proposons d'implanter.

Jusque là, la présentation a été plutôt concentrée sur les aspects théoriques des approches. Dans les paragraphes 3.2, 3.3et 3.5 nous allons au contraire donner des détails sur l'implantation possible dans un prototype réel, qui sera globalement caractérisé dans le chapitre 4. Dans le présent chapitre, nous indiquerons surtout, approche par approche, le surcoût obtenu en terme de complexité d'implantation, dont la stratégie de mesure est donc présentée au préalable dans le paragraphe 3.1. Nous indiquerons aussi les coûts en performance pouvant être estimés indépendamment d'une application particulière en cours d'exécution ou du contexte précis d'exécution. Ceci sera groupé sous l'appellation de "surcoûts statiques", par opposition aux surcoûts qui dépendent de l'instant d'occurrence d'une faute, de la tâche exécutée, voire des autres tâches en cours dans le système.

Le paragraphe 3.7 rassemble les conclusions générales sur les coûts statiques d'implantation des différentes approches proposées, considérées individuellement. L'analyse des caractéristiques globales lorsque ces approches sont combinées dans un système complet sera présentée dans le chapitre 4, ainsi que les analyses sur les surcoûts dynamiques.

3.1 Mesures de complexité

La complexité ou la « taille » d'un composant est fonction de l'ensemble des ressources dont il a besoin, et sa définition dépend du niveau considéré : matériel, système d'exploitation ou logiciel d'application.

Le **niveau matériel** est celui qui est le plus lié à l'implantation physique du système, donc les ressources utilisées sont celles fournies par la technologie cible dans laquelle le système va être réalisé. Des unités de mesure typiques sont le nombre de portes logiques ou le nombre de cm^2 de silicium nécessaires à l'ensemble du système. Dans notre cas, le système va être implanté notamment sur un circuit programmable du type FPGA de chez Xilinx. Sans rentrer dans les détails, on peut simplement constater qu'un FPGA type est composé d'une matrice de cellules de base, appelées CLB (« Configurable Logic Block », bloc logique configurable), elles-mêmes pouvant comporter plusieurs sous-blocs similaires (« slice »). Dans chaque CLB il y une ou plusieurs structures configurables permettant de réaliser des fonctions booléennes en implantant des tables de vérité de fonctions ayant un nombre maximum donné d'entrées (LUT, « Look-Up Table ») et un ou plusieurs points de mémorisation (FF pour « flip-flop », ou bascule en français). Ceci est illustré en Figure 3-1 pour la technologie que nous allons utiliser.

La configuration interne de chaque CLB et leurs interconnexions déterminent le comportement du FPGA. La capacité d'un FPGA à réaliser une application donnée dépend du nombre de CLB et de ressources de routage disponibles dans le composant, qui doit être supérieur à celui requis par l'application visée. On va donc pouvoir mesurer la taille des parties combinatoires et séquentielles d'une implantation sur la base respectivement de la quantité de **LUTs** et de **FFs** utilisées dans le FPGA. A ces quantités, on pourra, lorsque cela est pertinent, ajouter l'utilisation de ressources spécifiques offertes par le FPGA pour améliorer les performances, comme par exemple les bancs de **mémoire RAM embarquée**.

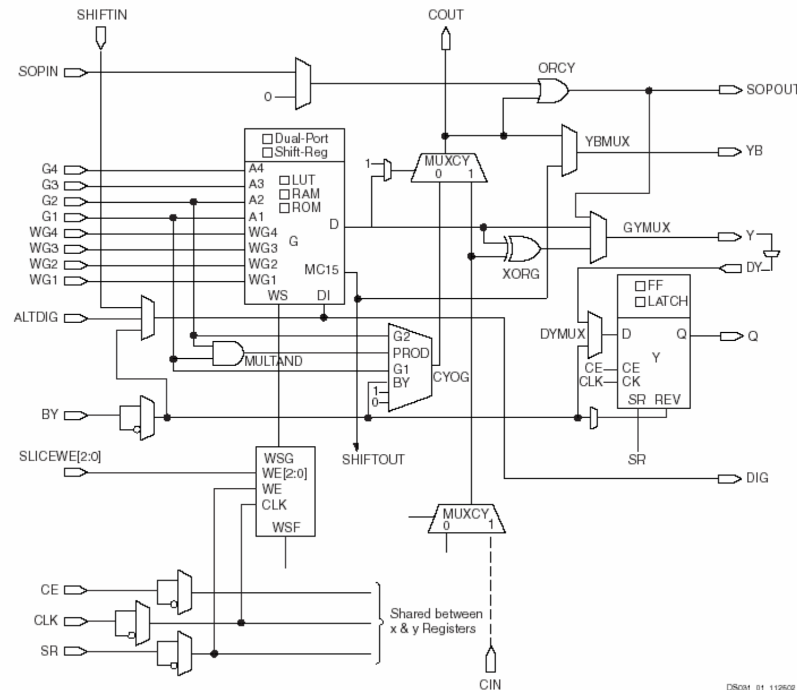


Figure 3-1 Schéma interne d'une cellule de base dans un FPGA Virtex-II Pro (pris de la documentation Xilinx)

Le système d'exploitation et le logiciel agissent tous les deux au même niveau d'abstraction, celui du processeur qui exécute des instructions traitant des données, le tout étant contenu en mémoire. La taille sera donc mesurée uniquement en termes d'**espace mémoire** occupé par le programme. L'impact dynamique lié au temps d'exécution par le processeur sera discuté dans le chapitre suivant.

3.2 Ajout du recouvrement dans le système d'exploitation eCos

3.2.1 Implantation du recouvrement

Dans le paragraphe 2.2 on a introduit une technique de Checkpoint & Rollback tirant profit de la modification de la fonction de changement de contexte. On va analyser ici la mise en œuvre pratique de cette approche dans le système d'exploitation choisi pour notre prototype, à savoir eCos. Le but n'est pas de détailler ligne par ligne les modifications du code, mais de donner en quelque sorte un « carnet de route » qui puisse être exploité pour répliquer la technique sur un autre système d'exploitation cible.

Les techniques de protection implantées en matériel fournissent un signal (ou plusieurs signaux) qui marque une faute détectée mais pas corrigée pendant l'exécution de la tâche Ta. Ce « signal de faute » SF peut être utilisé pour déclencher une **interruption**, mécanisme qui permet la réaction la plus rapide. C'est alors le traitant d'interruption qui appelle le processus de recouvrement. Il faut noter qu'une détection au niveau du logiciel d'application peut être traitée de la même manière, en générant une interruption logicielle. De même, les exceptions

générées classiquement dans les processeurs modernes, par exemple sur détection d'une instruction invalide, peuvent conduire au même type de recouvrement.

Après appel du traitant d'interruption, il n'est pas possible d'appeler directement la fonction de CdC, qu'on notera ctx_switch, pour demander un changement de Ta vers Ta : les fonctions ont toujours des contrôles pour éviter des situations de ce type. Du point de vue de l'ordonnanceur, ctx_switch(Ta,Ta) est une opération inutile, qui est donc supprimée. Modifier directement la fonction ctx_switch pour éviter ce problème peut être difficile et surtout pourrait changer les performances de base de l'OS de façon significative. La solution que l'on propose est donc d'exploiter les fonctions de bibliothèque C de **sauts non locaux**, setjmp et longjmp.

Setjmp permet de « marquer » un emplacement pendant l'exécution et **longjmp** permet d'y retourner. Du point de vue de l'utilisateur, ceci est donc très similaire à un saut inconditionnel, plus communément appelé « goto » du nom de l'instruction BASIC correspondante. Cependant, de tels sauts sont interdits dans les langages structurés comme le C. En réalité un setjmp est très similaire à une sauvegarde de contexte, et longjmp à un CdC car setjmp sauvegarde le contexte de la fonction. L'hypothèse de fonctionnement est dans le fonctionnement de la **pile** : à chaque appel de fonction on y stocke les informations sensibles (paramètres, adresses de retour, etc.…), qui sont enlevées au fur et à mesure que les fonctions imbriquées se terminent. Pour que longjmp puisse restaurer l'exécution il faut donc que la pile soit **au moins** au même point qu'à l'exécution de setjmp.

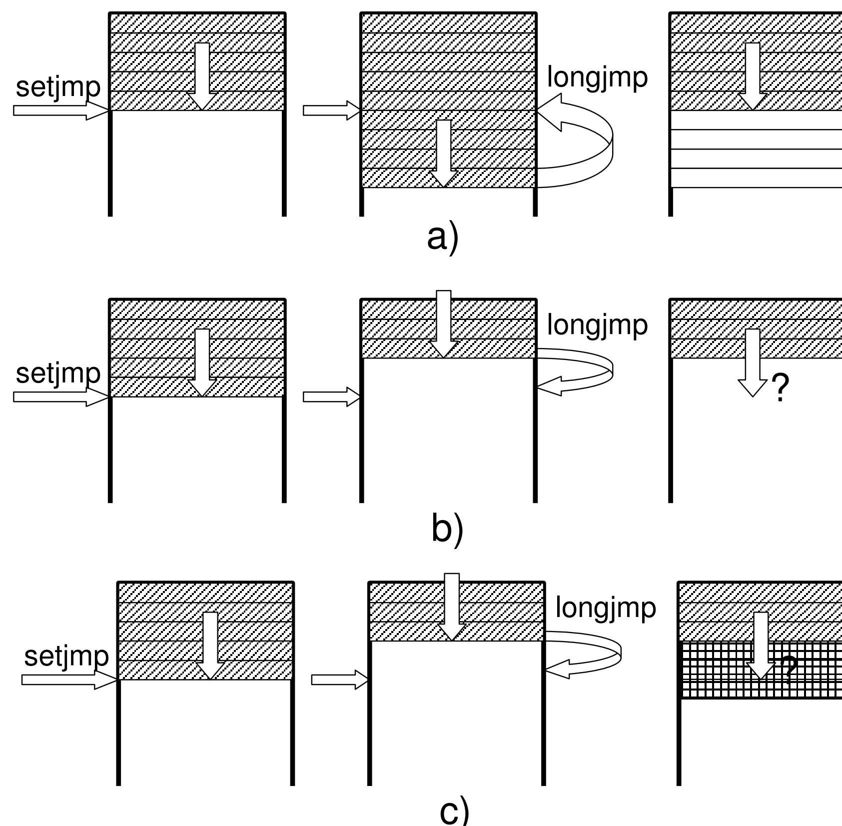


Figure 3-2 Exemple d'enchaînement setjmp-longjmp avec conservation (a), avec perte (b) et avec réécriture (c) de la pile

Pour simplifier, il ne faut pas que la fonction dans laquelle setjmp a été invoqué soit terminée. La Figure 3-2 montre ces deux cas. Dans le cas a) le longjmp amène l'exécution

vers une position valable, où la pile présente à l'instant du setjmp est conservée : les éléments déposés en plus pendant l'exécution vont être simplement ignorés. Dans le cas b) par contre l'exécution s'est déroulée de façon différente : avec le longjmp le programme se retrouve avec une pile qui n'est pas valable parce que des éléments ont déjà été dépilés. Le cas c) est plus sournois : la fonction dans laquelle setjmp a été appelé s'est terminée et a été lancée une autre fois : longjmp va donc retourner vers une pile qui n'est pas forcément valable, mais qui n'est pas non plus forcément incorrecte. Sous certaines hypothèses, propres au programme en cours d'exécution, le saut peut être valable.

Si le programmeur prévoit tous les cas possibles avec beaucoup de soin, longjmp peut être très puissant et utile.

Dans notre cas, on va l'appliquer à l'intérieur de la fonction de l'ordonnanceur, qui a une caractéristique très importante : elle ne se termine jamais. L'ordonnanceur est appelé à chaque changement de contexte, il choisit la tâche suivante à exécuter et il la lance avec la fonction ctx_switch, comme on peut voir dans la Figure 3-3, cas a). La boucle infinie se termine seulement quand il n'y a plus de tâche à exécuter et que le système s'arrête. Il existe plusieurs implantations et optimisations possibles, mais le schéma de base reste toujours celui-ci.

Dans le cas b) on montre donc le point d'insertion idéal pour le setjmp : juste avant l'appel de ctx_switch et après le choix de la nouvelle tâche à exécuter. En cas d'erreur pendant l'exécution de la tâche T_a, le longjmp va donc ramener l'exécution juste avant ctx_switch. $T_{vieille}$ et $T_{nouvelle}$ vont avoir gardé l'ancienne valeur, donc notamment $T_{nouvelle}=T_a$: la tâche va être redémarrée à partir de son dernier changement de contexte.

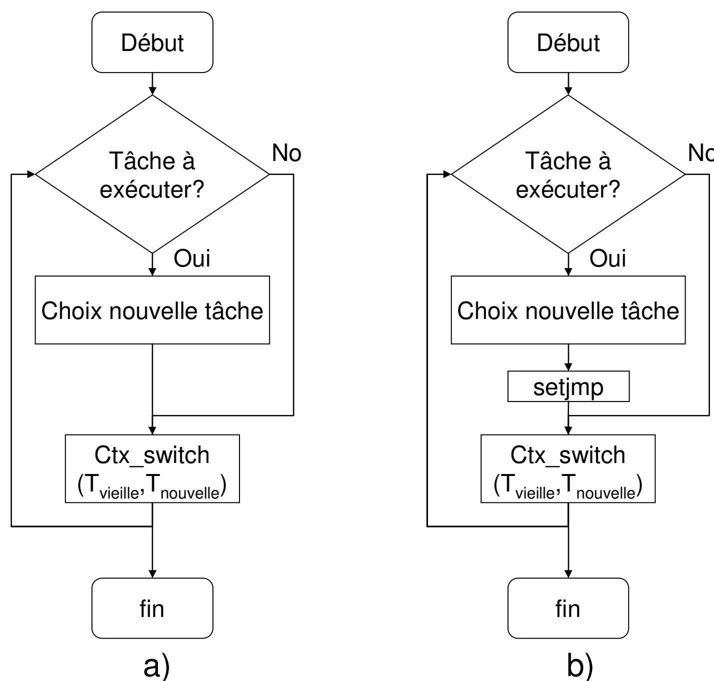


Figure 3-3 Algorithme de base de l'ordonnanceur (a) et insertion du setjmp (b)

Setjmp peut aussi donner une valeur de retour, décidée au moment de l'appel à longjmp. Cela va permettre de reconnaître des boucles infinies de recouvrement, causées par exemple par une faute permanente, et qui sont un des points critiques de toute technique de Checkpoint & Rollback, comme expliqué lors de la présentation de l'approche dans le paragraphe 1.4.5.

Cette implantation se prête très bien aux approches configurables comme dans eCos parce que les modifications des fonctions existantes sont très limitées et peuvent être soumises aux paramètres de configuration. Pour l'implantation en C, par exemple, il suffit de les inclure dans des clauses « ifdef » pour qu'elles disparaissent complètement lorsqu'elles ne sont pas sélectionnées, en restaurant l'état original du code.

Comme précédemment indiqué, du point de vue logiciel, la taille est représentée par l'espace mémoire occupé par le programme. Par ailleurs, en ce qui concerne l'OS, le coût en performances a deux composantes bien séparées. D'une part, les modifications introduites changent le temps nécessaire à l'exécution du CdC par l'OS. Cette modification est indépendante de l'application et surtout le surcoût temporel correspondant existe en l'absence de toute détection d'erreur. Une autre composante du coût, dynamique, est le temps nécessaire pour réaliser un recouvrement après une détection. Ce temps là correspond à l'activation du recouvrement, qui ne dépend que de la fonction implantée dans l'OS, puis au temps nécessaire pour revenir à l'état antérieur de la tâche, qui dépend de celle-ci et de l'instant d'occurrence de l'erreur. Nous ne parlerons ici que du temps nécessaire pour activer le recouvrement.

Tableau 3-1 Surcoûts statiques pour la protection du système d'exploitation

Implantation	Surcoût statique			Temps de recouvrement (µs)
	Exécution (%)	CdC (%)	Mémoire (octets)	
Appel direct du changement de contexte (estimé)	0	0	80	59,375
Avec sauts non locaux (estimé)	0,006	6,25	128	3,125
Avec sauts non locaux (mesuré)	0,02	20	128	10

Les pertes de performance peuvent être soit mesurées directement en temps d'exécution du programme soit estimées en comptant les instructions supplémentaires à exécuter. Le Tableau 3-1 montre les surcoûts estimés ou mesurés pour la protection de eCos avec la technique de C&R proposée. La colonne « CdC » indique la variation du temps d'exécution d'un changement de contexte, dont la durée initiale est approximativement de 50 µs. La colonne « Exécution » mesure la variation du temps d'exécution (temps écoulé entre deux CdC) : le timeslice pour une tâche étant de 50ms, les effets des variations sur le CdC sont très restreints.

L'implantation « sans sauts non locaux » estime le surcoût d'une réalisation qui appellerait directement la fonction de changement de contexte pour effectuer un recouvrement, en se souvenant des inconvénients précédemment discutés.

Les deux premières lignes du tableau sont obtenues à partir de l'estimation du nombre N d'instructions à ajouter. Pour obtenir la pénalité d'exécution on multiplie N par la

fréquence de travail dans le meilleur cas, c'est à dire pour une instruction terminée à chaque cycle. Pour l'occupation mémoire, on multiplie N par la taille d'une instruction (32bits) et on ajoute l'espace occupé par les éventuelles variables ajoutées. Ce sont seulement des estimations qui simplifient beaucoup la situation réelle, mais elles ont été très utiles pour choisir l'implantation du type « avec sauts non-locaux ». Outre les problèmes liés à la modification directe de la fonction ctx_switch, on peut remarquer que le temps de recouvrement estimé est beaucoup plus faible en utilisant des sauts non locaux, au prix d'un surcoût supplémentaire en mémoire et en temps d'exécution qui reste très acceptable.

La dernière ligne du tableau a été obtenue avec des mesures directes sur le prototype, et confirme bien les estimations. Il faut souligner que les estimations de vitesse étaient très optimistes, l'hypothèse « une instruction par cycle » n'étant pas respectée en général à cause des aléas d'exécution. Cela dit, l'ordre de grandeur mesuré reste le même que celui évalué, ce qui correspondait à l'objectif de nos évaluations.

3.2.2 Modification des sources eCos

Le fait que eCos soit un logiciel protégé par licence Open-Source, ainsi que sa forte configurabilité et portabilité, ont été les aspects qui ont conduit à son choix comme système d'exploitation de référence. Si on analyse les sources on se rend rapidement compte de la complexité du système d'exploitation : livrées dans une archive compressée de 16 Moctets, elles comptent une fois installées sur le disque 14477 fichiers (tous types confondus) organisés dans 5221 répertoires, pour une occupation totale de 478Moctets. Une complexité imposante qui demande de prendre des précautions avant toute intervention.

La stratégie d'approche a été ciblée vers la minimisation des modifications et la compatibilité avec les options préexistantes. Les nouvelles fonctions pour exécuter le recouvrement ont été regroupées dans un paquetage séparé, qui occupe 160Koctets une fois organisé selon les règles de eCos (5 fichiers dans 3 dossiers). L'implantation de ce paquetage dans eCos a demandé simplement de modifier 4 fichiers :

- « sched.cxx » qui réalise l'échéancier. 10 lignes ajoutées dans un fichier qui en fait 750 ;
- « sched.hxx » pour inclure le nouveau paquetage. 1 ligne à rajouter ;
- « pkgstart.cx » pour initialiser la méthode lors du lancement du système d'exploitation. 6 lignes dans un fichier de 100 lignes ;
- « ecos.db » qui est exploité par l'outil de configuration de eCos. 7 lignes dans un fichier de 4500 lignes.

On souligne que dans tous les cas on s'est limité à ajouter les commandes nécessaires pour activer la technique de protection et dans aucun cas on a eu à modifier des commandes préexistantes, ce qui donne une complète compatibilité avec des éventuelles mises à jour des sources. L'insertion dans le cycle de configuration rend aussi l'approche complètement compatible avec toutes les options d'eCos, d'autant plus qu'en ayant travaillé au-delà de la couche d'abstraction matérielle (HAL) on n'a pas restreint l'application au processeur Leon2.

Comme prévu, l'intervention a été extrêmement légère en comparaison avec la taille du système d'exploitation. L'implantation dans un autre OS quelconque pourrait être faite de la même façon, avec un temps de développement très réduit.

3.2.3 Critères de validité dans l'application vidéo

L'application vidéo a été développée avec le but de donner un aperçu « visuel » de l'efficacité des techniques employées, surtout pour le C&R utilisant le changement de contexte. Une application de traitement d'images est à tous effets une application de traitement de signal, qui prend en entrée des échantillons (3 fois 8 bits, pour les trois couleurs primaires), les traite selon un algorithme précis et les redonne en sortie pour l'affichage. On va ici donner une introduction sur les principes de l'algorithme JPEG pour mieux comprendre la façon dont les critères du paragraphe 2.2.2 peuvent être appliqués. Le lecteur intéressé pourra trouver plus d'informations soit sur le site officiel du comité de standardisation [JPEG04], soit dans les innombrables tutoriaux en ligne, par exemple dans [Wikip06].

JPEG est un standard ouvert de compression à perte à taux variable qui a connu un grand succès grâce à ses capacités et sa maniabilité. Par exemple la plupart des images se trouvant sur internet sont compressées avec ce format. L'algorithme de base divise l'image source en carreaux de 8x8 pixels, qui sont traités séparément selon la boucle illustrée dans la Figure 3-4.

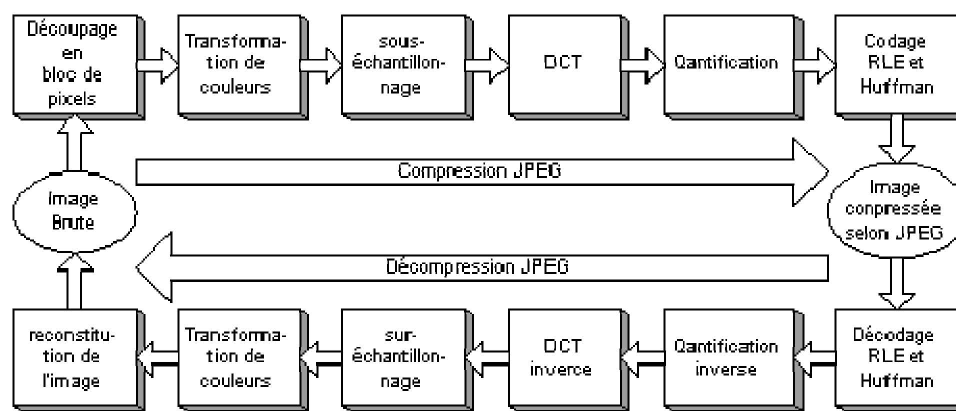


Figure 3-4 Boucle de compression/Décompression JPEG (image prise de [Wikip06])

Les différentes étapes sont :

- Transformation des couleurs : l'algorithme peut travailler sur une image exprimée dans n'importe quel format, mais les meilleurs taux de compression sont obtenus à partir du codage luminance/chrominance (YCbCr), donc on convertit toujours une image dans ce format. Dans le cas de notre application vidéo, cette étape n'est pas nécessaire car le signal d'entrée est déjà codé dans ce format ;
- Sous-échantillonnage : l'œil humain ne peut pas distinguer clairement tous les pixels à cause de sa faible résolution, donc la taille de l'image est réduite en considérant seulement un sous-ensemble d'échantillons ;
- DCT ou Transformée en Cosinus Discrète : similaire à la transformée de Fourier, cette fonction permet de coder dans la matrice 8x8 les fréquences spatiales et les variations d'amplitudes plutôt que les valeurs absolues des pixels. Aucune information n'est perdue parce qu'il suffit d'appliquer la DCT inverse pour récupérer les valeurs originales, mais ainsi codée l'image est plus pratique pour la suite de l'algorithme ;
- Quantification : c'est l'étape où il y a une perte de qualité. Chaque bloc est filtré pour éliminer les hautes fréquences auxquelles l'œil est très peu sensible. Cette étape est obtenue en multipliant la matrice représentant le bloc par une matrice de compression, dont les valeurs sont spécifiées par le standard.
- Le bloc ainsi transformé est codé avec un code à longueur variable de type Huffman pour réduire au maximum sa taille sans perdre aucune information supplémentaire.

- L'image ensuite obtenue peut être enregistrée dans un fichier, mais pour pouvoir l'afficher à l'écran il y a besoin d'effectuer un décodage en effectuant les étapes inverses des étapes de compression.

Ce n'est pas l'objectif de cette thèse de discuter de l'efficacité, des performances et des limitations de ce format. On a par contre remarqué que le codage/décodage d'une image est composé d'une suite d'opérations qui modifient de façon incrémentale des blocs de base : c'est un excellent candidat pour l'application du troisième critère de validité. Un recouvrement conduirait un ou plusieurs blocs à subir deux fois le traitement mais cela ne devrait pas trop perturber le résultat. Dans le cas d'un recouvrement réussi, on devrait simplement observer une dégradation de l'image finale.

Pour valider cette hypothèse on a effectué des tests sur l'algorithme JPEG : on a ajouté dans le code original des setjmp/longjump (voir paragraphe 3.2.1) qui nous ont permis de simuler des recouvrements. Dans cette phase, on s'est concentré sur les caractéristiques de l'algorithme, donc la plate-forme d'exécution n'est pas importante : pour pouvoir évaluer les résultats au mieux on a donc compilé la bibliothèque [Lane98] sur Linux et travaillé à partir de l'image la plus classique dans la littérature JPEG, Lena.

a) b)

Figure 3-5 Image originale "Lena" en bitmap (a) et compessée en JPEG avec une qualité de 20/100 (b)

Le standard JPEG décrit des algorithmes de compression et de décompression très similaires, comme illustré dans la Figure 3-4. On a donc choisi de se concentrer sur la décompression pour pouvoir évaluer directement les effets des recouvrements en visualisant l'image finale, mais vu la similitude entre algorithmes les considérations devraient être valables aussi pour la compression.

Le programme de test prend en entrée l'image compressée et génère une image bitmap non codée qu'on peut visualiser directement. On a choisi un taux de compression très élevé (qualité 20 sur 100) pour que la perte de qualité générée par le JPEG soit bien évidente et que les transformations parasites induites lors d'un recouvrement soient significatives. Pour simuler les recouvrements, on a suivi la stratégie suivante pour placer les points de saut :

1) Setjmp au début du calcul, longjump entre deux lignes de l'image;
2) Setjmp et longjump entre deux lignes;

3) Setjmp au début du calcul, longjump placé aléatoirement au milieu du code;
4) Setjmp et longjump placés aléatoirement au milieu du code;

Dans tous les cas, la décompression s'est effectuée jusqu'à la fin, avec seulement une petite déformation correspondant à un décalage d'une partie de l'image, comme on peut le voir dans la Figure 3-6. Dans le cas d'un flot de traitement temps réel, une telle déformation sur une image passerait inaperçue. Dans le cas d'une décompression d'une seule image, le décalage est visible mais n'est pas suffisant pour rendre l'image inexploitable dans de nombreuses applications.

Figure 3-6 Lena reconstituée à partir du fichier JPEG compressé sans (a) et avec simulation de recouvrement (b)

Le point où le décalage a lieu dépend exclusivement du moment du recouvrement, par contre on n'a aucune modification due à l'instant du checkpoint. Dans les cas 1) et 2) l'exécution se terminait toujours correctement, dans les cas 3) et 4) on a eu parfois des messages d'erreur (« codage JPEG non respecté ») et une fois le programme s'est bloqué, mais on a quand même toujours obtenu l'image en sortie. Ces erreurs sont sans doute causées par un décalage du point de lecture dans le fichier source, qui donc ne semble plus suivre le standard JPEG : c'est une erreur de synchronisation de l'entrée. Dans le cas d'une entrée temps réel, cela devrait se résoudre automatiquement grâce aux signaux de synchronisation générés pas les convertisseurs en entrée.

Pour pouvoir comparer ces résultats avec de vraies injections (voir chapitre suivant) on a développé une variante du programme de traitement d'image qui à la place de compresser une image prise de la camera prend comme entrée « Lena » qui a été au préalable mémorisée en mémoire. En la visualisant à l'écran comme illustré en Figure 3-7 on pourra avoir une comparaison directe avec les résultats de ce paragraphe.

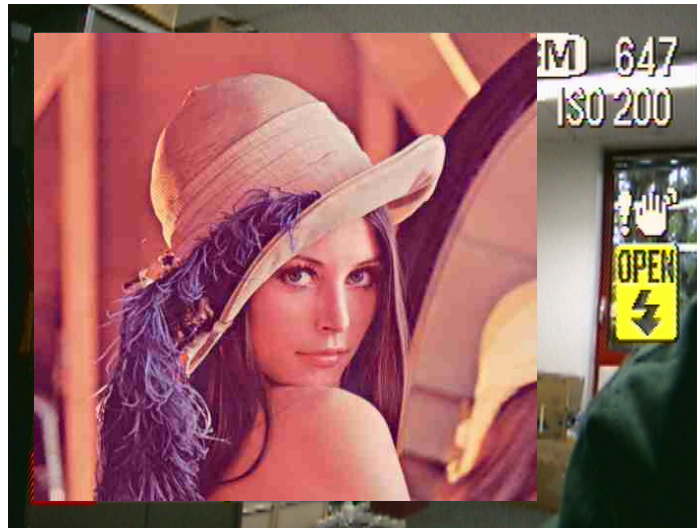


Figure 3-7 Lena comprimée en JPEG visualisée à l'écran par le prototype

3.3 Durcissement du processeur Leon2

3.3.1 Protections ajoutées et surcoûts

Le processeur Leon2 a été protégé de façon extensive avec les techniques décrites dans le paragraphe 2.3, en ayant soin de garder le plus haut degré de configurabilité possible. L'attention a été focalisée sur l'unité centrale, qui rassemble les éléments fondamentaux pour l'exécution des instructions : les mémoires caches, l'unité entière et les bancs des registres.

Dans la Figure 2-2, l'unité centrale n'avait pas été détaillée, on en donne donc la structure dans la Figure 3-8. On peut voir que tous les composants onté été protégés, mis à part l'interface AMBA qui est une liaison vers l'extérieur et qui n'intervient pas directement dans l'exécution des instructions.

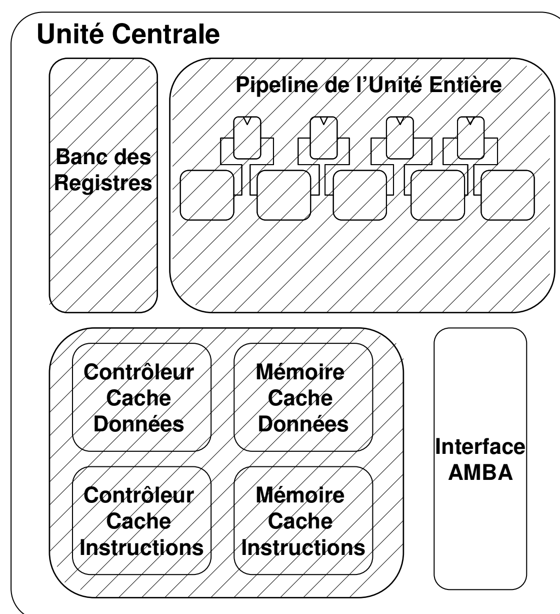


Figure 3-8 Structure de l'unité centrale du Leon 2. Les parties protégées sont hachurées

Le Tableau 3-2 résume les protections adoptées.

Tableau 3-2 Résumé des protections dans l'unité centrale du Leon 2

Composant	Logique combinatoire	Type de protection	Ressources séquentielles	Type de protection
Banc de registres	Néant		Registres implantés en SRAM	Détection par parité
Pipeline	Etages du pipeline	Détection par parité /Recouvrement	Registres inter étages du pipeline	Détection par parité
Contrôleurs des caches	Oui	Détection par parité	Registres internes	Détection par parité
Mémoires cache	Néant		Mémoires SRAM	Détection par parité, Recouvrement par défaut de cache

On peut remarquer que, exception faite de la logique combinatoire du pipeline et des mémoires cache, la protection est plutôt orientée vers la simple détection : son rôle est d'être le déclencheur des protections au niveau système d'exploitation décrites dans le paragraphe 3.2. Les détections sont basées sur un calcul de parité, mais tout autre code pourrait très facilement être ajouté.

Comme dans le cas de eCos, toutes les modifications sont soumises à des paramètres de configuration que l'utilisateur peut choisir librement, et sont complètement compatibles avec les paramètres originaux du Leon2.

Au moment de l'implantation réelle des techniques de prédiction de parité du paragraphe 2.3, on s'est heurté à un problème imprévu : les signaux indéfinis. Dans un cycle de calcul donné, une entité L peut ne pas utiliser toutes ses entrées, mais seulement une partie. Du point de vue du concepteur, la valeur des entrées non utilisées n'est pas importante : suivant le style de codage, ces signaux peuvent être laissés indéfinis ou recevoir une valeur quelconque. En VHDL, il est par exemple possible de leur donner la valeur phi-booléen (« don't care ») : au moment de la synthèse, l'outil va leur affecter la valeur qui permet d'obtenir la meilleure optimisation. Le problème est que pour calculer la prédiction de parité, il est nécessaire que toutes les entrées protégées soient définies en permanence : un seul signal non défini va se propager dans l'arbre de calcul et rendre indéfinie la sortie aussi, comme illustré dans la Figure 3-9. Il devient alors impossible de valider les protections au niveau RTL, avant synthèse.

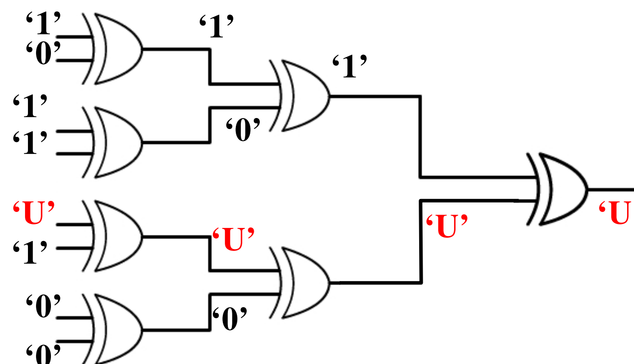


Figure 3-9 Propagation d'un signal indéfini dans un arbre de XOR

Pour résoudre ce problème, une première approche consiste à supprimer tous les signaux indéfinis en leur affectant une valeur par défaut. Toutefois, on réduit alors les possibilités d'optimisation lors de la synthèse. Une autre approche, utilisée dans notre cas, consiste à filtrer les signaux utilisés pour le calcul de parité. Trois stratégies sont possibles :

- statique minimale : seuls les signaux qui ne sont jamais indéfinis sont pris en compte ;
- dynamique : à chaque cycle, seuls les signaux définis à cet instant sont pris en compte;
- statique maximale : tous les signaux sont pris en compte, sans tenir compte de leur valeur.

La stratégie minimale est la plus directe et radicale, mais peu conduire à exclure un grand nombre de signaux et donc diminuer de façon importante le taux de couverture de fautes.

La stratégie dynamique s'adapte à la situation et donc offre une couverture optimale. Le prix est la nécessité de connaître à chaque cycle les signaux à prendre en compte, ce qui requiert la définition de conditions permettant d'identifier les instants où chaque signal est valide. Cette phase d'analyse peut s'avérer très longue si le composant à protéger est assez complexe ou si il a été développé par des tierces personnes. En plus, les conditions de validité peuvent être compliqués et donc avoir leur impact sur le chemin critique ou sur la taille si par exemple un prototypage du circuit est réalisé en maintenant cette option de filtrage.

Il faut ici souligner un point potentiellement important de la stratégie dynamique. Si l'option de filtrage mise en place pour permettre la validation au niveau RTL est maintenue lors de la synthèse, le circuit obtenu est capable d'identifier à chaque cycle les signaux qui sont importants du point de vue fonctionnel. En conséquence, une erreur survenant sur un signal à un instant où il est inutilisé peut ne pas générer de signal d'erreur. Ceci permet d'éviter l'activation d'une procédure de recouvrement alors que le calcul se serait déroulé normalement. Cette stratégie de filtrage dynamique peut donc avoir un réel intérêt au niveau des performances du système, bien au-delà son but initial. Naturellement, elle entraîne alors un coût plus important, mais pas forcément beaucoup plus important que la stratégie statique maximale, comme cela sera montré ci après.

La stratégie maximale repose sur l'observation qu'après synthèse tous les signaux vont avoir une valeur réelle, même ceux qui sont indéfinis dans la description RTL. Le résultat a la simplicité de l'approche minimale avec une bonne couverture de fautes, mais on perd la possibilité de valider complètement le système avant synthèse. Il faut aussi remarquer que si un signal « a » et sa réplique « a_{bis} » sont indéfinis, il n'est pas garanti que la valeur affectée lors de la synthèse soit la même pour les deux signaux. La réplique étant sensée être très optimisée, il est possible que lors de cette optimisation l'outil choisisse des valeurs différentes. Le choix de la stratégie maximale est donc soumis à une bonne confiance dans l'outil de synthèse, qui doit assurer la même valeur pour chaque signal indéfini et sa réplique.

Le banc de registres est implanté dans une mémoire embarquée à cause de sa taille. il a donc été directement protégé envers les SETs avec la technique décrite dans le paragraphe 2.3.3.1.

Les mémoires caches ont été protégées selon le même principe. Chaque ligne en cache est composée par deux champs, un champ « données » qui contient les données copiées de la mémoire centrale (données ou instructions) et un champ « tag » qui contient les informations

nécessaires au contrôleur pour gérer l'algorithme de cache. On a donc implanté un schéma illustré dans la Figure 3-10, avec un bit de parité par champ.

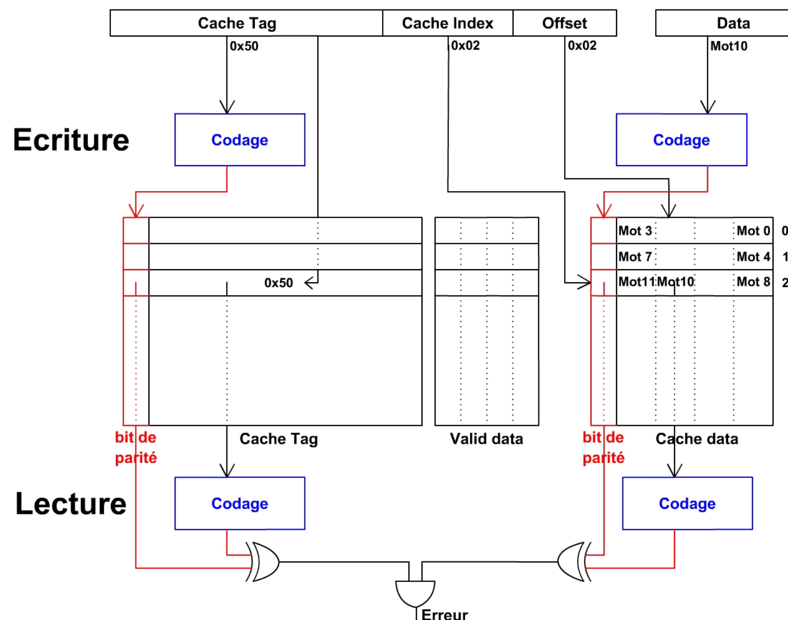


Figure 3-10 Protection des caches par parité

Les éléments de protection rajoutés causent une augmentation de taille du circuit et peuvent s'insérer sur le chemin critique et donc causer une diminution de la fréquence de travail. Le Tableau 3-3 et le Tableau 3-4 montrent les surcoûts de protection pour l'unité entière seule et avec le système de caches respectivement, en tenant compte des différentes stratégies de filtrage possibles pour l'unité entière. La synthèse a été effectuée avec la suite logicielle ISE 7.1 de Xilinx pour une cible Virtex II Pro. Les valeurs absolues ne sont pas très intéressantes pour notre analyse, qui est concentrée sur le surcoût des protections. Pour donner un ordre de grandeur, le système cible, synthétisé avec les options de base, peut fonctionner en moyenne autour de 45Mhz en occupant approximativement 40% d'un circuit VirtexII-Pro VP20. Ces valeurs peuvent être facilement améliorées en jouant avec les options de synthèse : avec une optimisation ciblée vitesse, par exemple, la fréquence maximale peut atteindre environ 80Mhz, au prix d'occuper 20% de ressources en plus.

Le détail des protections implantées dans les différents cas est le suivant :

- Détection de SETs dans l'Unité Entière : prédiction de parité dans les étages du pipeline ;
- Détection de SETs et SEUs dans l'Unité Entière : prédiction de parité dans les étages du pipeline et protection par parité des registres inter-étages ;
- Recouvrement de SETs dans l'Unité Entière : prédiction de parité dans les étages du pipeline avec validation de mise à jour des registres inter-étages ;
- Détection dans les caches : prédiction de parité dans les contrôleurs, protection par parité des registres des contrôleurs, protection par parité des champs données et tag de chaque cache ;
- Recouvrement dans les caches : protection par parité des champs données et tag de chaque cache avec génération de cache miss.

Tableau 3-3 Surcoût statique des protections pour l'Unité Entière du Leon2

Type de protection \ Filtrage des signaux		Dynamique	Statique Minimale	Statique Maximale
Détection de SETs dans l'Unité Entière	Slices	43,70%	13,97%	40,50%
	Bascules dans une Slice	2,86%	0,35%	2,35%
	LUTs	46,77%	12,53%	43,53%
	Fréquence Maximale	-12,69%	-12,55%	-12,69%
Détection de SETs et SEUs dans l'Unité Entière	Slices	44,39%	14,20%	43,09%
	Bascules dans une Slice	3,76%	1,30%	4,51%
	LUTs	47,48%	13,29%	46,12%
	Fréquence Maximale	-12,69%	-12,55%	-12,68%
Recouvrement de SETs dans l'Unité Entière	Slices	51,91%	22,12%	48,73%
	Bascules dans une Slice	32,77%	30,26%	32,26%
	LUTs	55,01%	20,78%	51,81%
	Fréquence Maximale	-70,18%	-48,51%	-64,38%

Tableau 3-4 Surcoût statique des protections pour l'Unité Entière et le système de caches du Leon2

		Par rapport au Leon2 original	Par rapport au Leon2 avec IU protégée
Détection dans les caches	Slices	0,19%	
	Bascules dans une Slice	0,00%	
	LUTs	0,12%	
	Fréquence Maximale	-1,33%	
Détection dans les caches et l'unité entière	Slices	43,92%	0,15%
	Bascules dans une Slice	2,86%	0,00%
	LUTs	46,90%	0,09%
	Fréquence Maximale	-12,69%	0,00%
Recouvrement dans les caches	Slices	0,52%	
	Bascules dans une Slice	0,00%	
	LUTs	0,79%	
	Fréquence Maximale	0,00%	
Recouvrement dans les caches et l'unité entière	Slices	52,48%	0,37%
	Bascules dans une Slice	32,77%	0,00%
	LUTs	55,85%	0,54%
	Fréquence Maximale	-70,18%	0,00%

En analysant le Tableau 3-3, on peut remarquer que l'hypothèse de simplification de l'unité de prédiction de parité est vérifiée : dans tous les cas, le surcoût combinatoire est bien plus bas que le 100% qu'on devrait attendre pour une duplication pure. Le parallélisme des schémas de détection est aussi très efficace : la perte de fréquence est seulement autour de 12%. Par contre la pénalité de performances pour l'approche de recouvrement dans l'unité entière est beaucoup plus élevée, autour de 70%. On rappelle que la mise à jour des registres est soumise à la validation du résultat dans tous les étages du pipeline (Figure 2-14), donc on s'attendait un résultat de ce type.

Comme précédemment mentionné, le Tableau 3-3 met également en évidence le surcoût très faible de la stratégie de filtrage dynamique, comparé à une vérification systématique de tous les signaux. Cette stratégie peut donc être valablement conservée jusque dans la version finale du circuit. Le coût beaucoup plus faible de la stratégie statique minimale doit quant à lui être mis en regard de la diminution, probablement notable, des capacités de détection.

Le Tableau 3-4 montre que le système de protection des caches est très léger et son surcoût est pratiquement inexistant à l'échelle du processeur complet. Cette protection, introduite dans le paragraphe 2.3.3.1 et détaillée dans le paragraphe 3.2, est basée sur l'ajout d'un bit de parité pour les champs de tag et de données des mémoires caches. Pour optimiser l'utilisation des ressources, les outils de synthèse permettent d'implanter les mémoires RAM dans les RAM embarquées (BRAM) du FPGA, selon le principe montré en Figure 3-11.

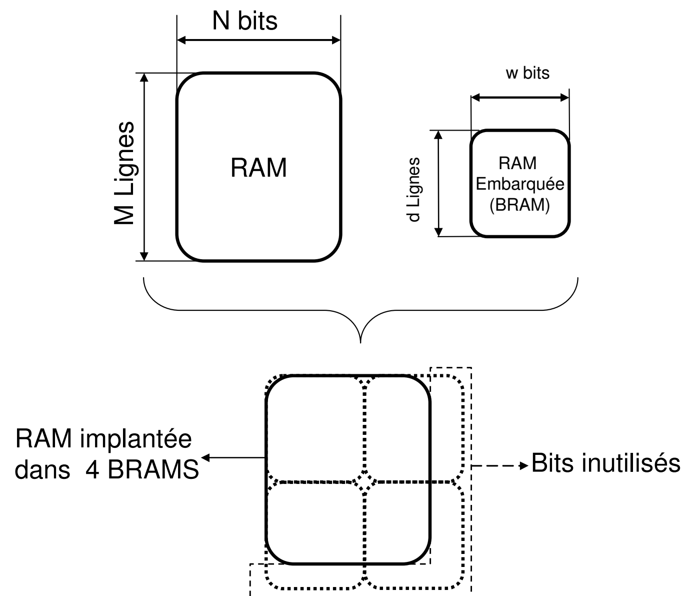


Figure 3-11 Implantation d'une mémoire RAM dans une FPGA

Comme les RAMs à implanter sont rarement des multiples exacts des blocs BRAM disponibles, il y a souvent des bits inutilisés, qui peuvent être exploités pour y stocker les bits de code. Dans notre cas, on a à implanter seulement le bit de parité, ce qui peut être fait sans besoin d'aucune mémoire supplémentaire et donc à coût zéro. Il faut quand même souligner que cet effet est dû aux caractéristiques des cibles FPGA : dans une implantation ASIC, par exemple, on pourrait avoir à considérer la surface occupée par les points mémoires supplémentaires. Le surcoût réel dépendrait alors des caractéristiques exactes du générateur de mémoire, et notamment de la possibilité d'avoir ou non exactement le nombre de bits par mot nécessaire sans l'ajout du code.

Pour donner un terme de comparaison, on a aussi effectué avec le logiciel Design Compiler (DC) de Synopsis la synthèse de Leon2 sur une cible ASIC dont les bibliothèques étaient disponibles au laboratoire (technologie AMS 3.60). Malheureusement, cette technologie n'est pas disponible dans la distribution de Leon2, donc on n'a pas eu la possibilité d'avoir, par exemple, les RAMs et ROMs implantées sur des cellules de bibliothèque. Sans cette aide, le logiciel de synthèse implantait toutes les mémoires sous la forme de bascules, ce qui donnait des surfaces énormes et irréalistes. Pour notre analyse, le plus important est l'occupation de surface due à la logique : porter Leon2 sur la nouvelle technologie n'aurait pas eu trop d'intérêt. On a donc décidé d'effectuer des synthèses en boîte

noire : les valeurs du Tableau 3-5 ne comptabilisent pas la surface occupée par les mémoires embarquées.

Tableau 3-5 Synthèse de Leon2 sans RAMs sur cellules AMS 3.60 à 80MHz

Leon2 original	Surface combinatoire (μm^2)	Surface non combinatoire (μm^2)	Surface Totale (μm^2)
	760385,50	480770,28	1695925,75

Type de protection \ Filtrage des signaux		Dynamique	Statique Minimal	Statique Maximal
Détection de SETs dans l'Unité Entière	Surface combinatoire	+85,35%	+22,26%	+86,51%
	Surface non combinatoire	+4,55%	+0,66%	+3,76%
	Surface Totale	+53,84%	+16,93%	+55,61%
Recouvrement de SETs dans l'Unité Entière	Surface combinatoire	+126,27%	+41,11%	+127,09%
	Surface non combinatoire	+32,83%	+28,55%	+32,56%
	Surface Totale	+87,85%	+38,08%	+89,52%

Une caractéristique de Design Compiler est que la fréquence de travail fait partie des contraintes : la synthèse est faite de façon à optimiser l'occupation de surface pour cette valeur. On a donc choisi de donner une fréquence sensiblement supérieure à celle obtenue pour la cible FPGA, 80MHz.

Si on compare le Tableau 3-5 au Tableau 3-3 on remarque qu'en détection le surcoût d'occupation totale (la seule valeur directement comparable) est du même ordre de grandeur que pour l'implantation FPGA : ~50% contre ~40%. En recouvrement, la taille augmente de façon significative (80%), mais l'implantation ASIC peut quand même travailler à 80MHz.

Les oscillations selon les paramètres sont beaucoup plus marquées : le surcoût de la solution de filtrage « statique minimale » est presque un quart de celui des autres approches, ce qui est probablement dû à la simplification de l'arbre de XOR qui calcule la parité. De façon intéressante, on remarque qu'avec cette technologie la stratégie de filtrage dynamique devient moins coûteuse que la stratégie statique maximale. L'utilisation des conditions de vérification a donc permis à l'outil de synthèse de simplifier la logique, conduisant à une implantation meilleure à la fois en termes de validation RTL, de surface et de robustesse vis-à-vis des erreurs sans effet fonctionnel.

3.3.2 Automatisation des protections

L'expérience acquise lors de la protection de l'unité entière de Leon2 a permis de développer un algorithme pour appliquer la détection par prédiction de parité à n'importe quel composant. Pour pouvoir appliquer directement le schéma de la Figure 2-11, il faut pouvoir départager la partie purement combinatoire L de la partie purement séquentielle R, ce qui n'est généralement pas immédiat.

De façon générique, on peut dire qu'une entité E, représentée dans la Figure 3-8, va être composée par :

- un ou plus processus P_i, qui sont sensés regrouper les opérations combinatoires ;
- des « signaux de registre » SR_j, qui regroupent les signaux représentant les points mémoire ;
- des « signaux de communication » SC_k, qui ont pour rôle de transmettre des informations entre les P_i, les SR_j et les entrées/sorties.

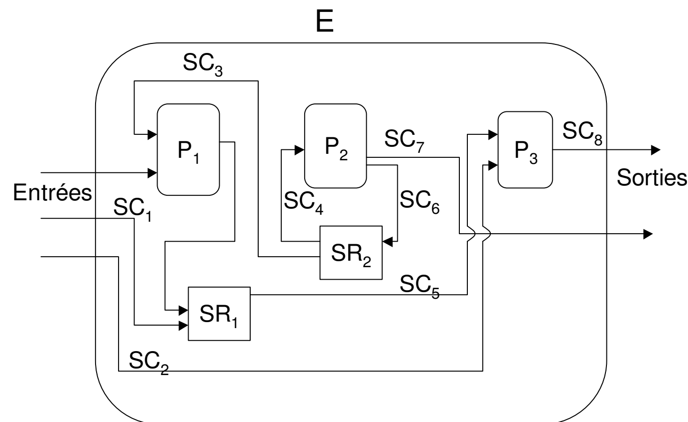


Figure 3-12 Modélisation d'une entité générique E

Répliquer un processus signifie faire une opération de « copier-coller », ce qui n'est pas très propre et qui pourrait donner des problèmes en phase de maintenance du code. Il est donc préférable d'encapsuler chaque processus P_i dans une entité P_E_i : les entrées sont les signaux SC_k dont la valeur est lue dans le processus, les sorties les signaux qui sont modifiés par le processus.

A partir de ce découpage, le schéma de la Figure 2-11 peut être directement appliqué, de façon complètement automatique, pour obtenir un schéma similaire à celui de la Figure 3-12. Pour des raisons de lisibilité, les décodeurs ne sont pas représentés ici en logique double rail, mais la même approche systématique peut bien sûr s'appliquer.

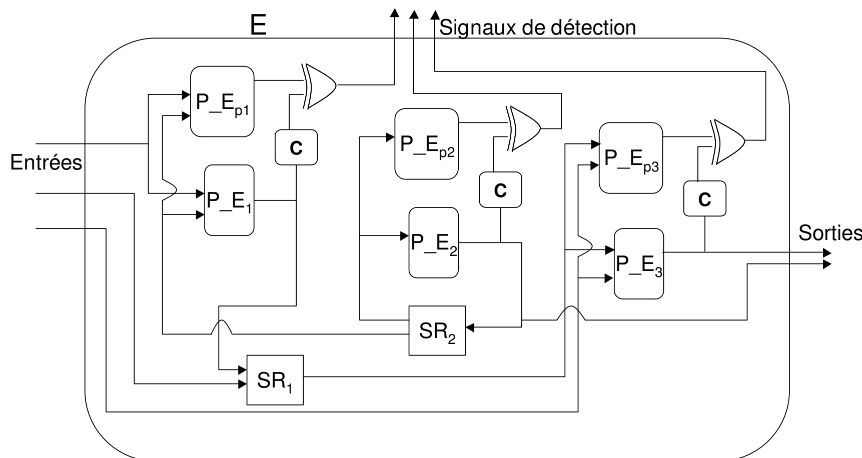


Figure 3-13 Entité de la Figure 3-12 protégée par réplication automatique

Il est aussi possible d'ajouter un niveau de hiérarchie supplémentaire en encapsulant chaque couple (P_i, P_E_i) dans une entité, de façon à garder l'entité de plus haut niveau, E, plus ordonnée et lisible.

Cette technique a été utilisée pour la protection des contrôleurs de caches, opération qui s'est avérée très facile et mécanique. Un parseur VHDL (ou un programme équivalent) pourrait facilement effectuer cette opération, avec la limitation que le code source doit être écrit de façon à bien séparer les parties combinatoires et séquentielles. Par exemple, une machine à états spécifiée par un seul processus ou ayant des éléments de mémorisation parasites (dus par exemple à des sorties non assignées) nécessiterait une transformation avant de pouvoir être traitée.

3.4 Implantation collaborative entre mémoire cache et OS

La technique de collaboration matériel - système d'exploitation décrite dans le paragraphe 2.5 a été élaborée dans les phases finales de la thèse et n'a pas encore été implantée complètement au moment de rédiger ce document. Cependant, un gros effort à été apporté à sa spécification, ce qui va rendre très facile son implantation.

L'architecture de principe proposée dans le paragraphe 2.5 est bien adaptée à une implantation dans le système réel, des modifications restant possibles pour améliorer les performances. Une solution est illustrée dans la Figure 3-10 :

- la cache données CD est implantée dans une mémoire double port pour permettre à la DVC d'y accéder en même temps que l'unité entière IU ;
- un bloc de multiplexage Mux est implanté entre l'IU et la CD, ce qui permet en cas de recouvrement d'accéder directement à la DVC, sans besoin de mettre à jour la CD ;
- les bits de modification (D) sont sauvegardés directement dans le bloc Mux, ce qui permet à la DVC d'être la réplique exacte de DC, avec un bon gain de simplicité.

DVC est donc une mémoire simple, et ce sera au bloc Mux de gérer l'algorithme de collaboration avec le système d'exploitation. On rappelle que :

- CD[i] et DVC[i] indiquent l'adresse de l'i-ème emplacement dans le cache données et dans la cache des victimes modifiées respectivement ;
- D(i) est son bit de modification (« dirty », à 1 si modifié) ;
- « mot lu » est le mot reçu par l'unité entière ;
- « mot écrit » est le mot envoyé par l'unité entière ;
- « rollback » est le signal qui indique un recouvrement ;
- « ctx » est le signal de changement de contexte. .

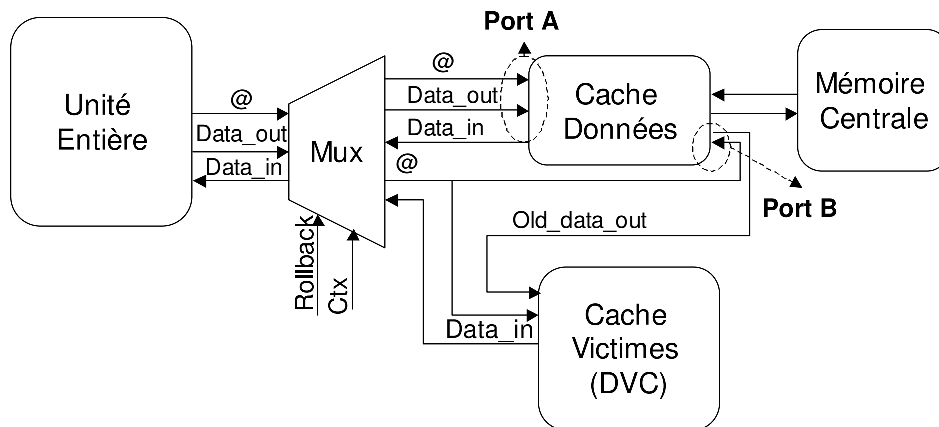


Figure 3-14 Implantation de la mémoire de victimes modifiées

La Figure 3-15 montre l'algorithme complet. Il faut remarquer qu'après un recouvrement on perd toute information sur les écritures en cache données, puisqu'au fur et à mesure tous les D(i) vont être remis à 0. Cela n'est pas forcément gênant parce que dans un schéma de C&R on n'est pas sensé enchaîner plusieurs recouvrements, situation qui généralement est détectée comme une boucle infinie potentielle et donc assimilée à une erreur non récupérable (voir paragraphe 1.4.5). Si on veut quand même garder le schéma valable dans le cas de plusieurs recouvrements successifs, il suffit de définir D comme une variable à trois valeurs : E (« empty », vide), D (« dirty », modifié) et D^2 (modifié deux fois). Dans le

cas d'une écriture pendant un recouvrement dans une case i « dirty » la valeur DVC[i] n'est pas modifiée, mais son état devient « doublement modifié », donc D^2. Le prix à payer est qu'il faudra coder D sur deux bits, donc on aura besoin de deux fois plus de registres.

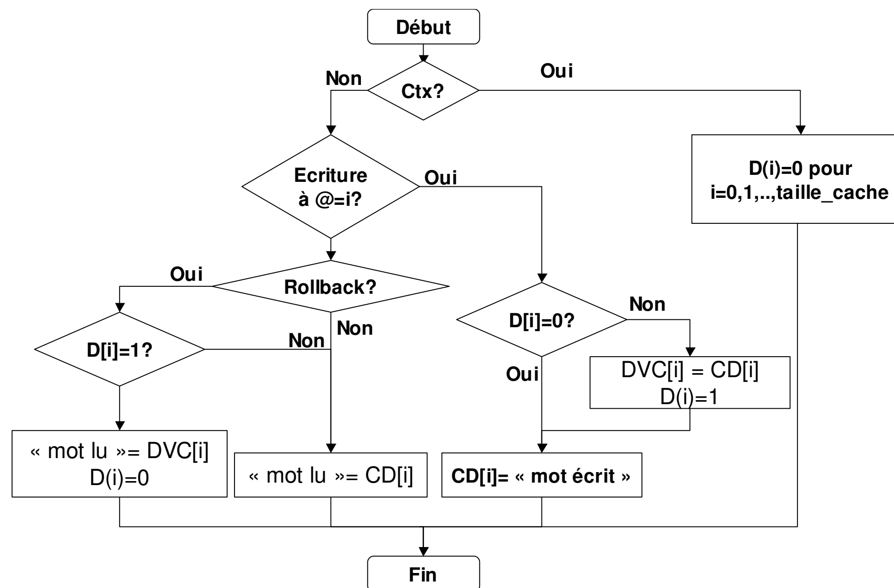


Figure 3-15 Algorithme du bloc de multiplexage entre IU, CD et DVC

Si la taille de la cache données est importante, il y a un risque que le registre D devienne très grand : une solution possible est de l'implanter dans une mémoire RAM embarquée.

3.5 Sécurisation du coprocesseur AES

L'IP AES possède plusieurs paramètres qui modifient ses performances et sa taille : le Tableau 3-6 montre les résultats de synthèse avec Leonardo de Mentor Graphics. On peut remarquer que la version avec les S_boxes en ROM (c'est-à-dire, en pratique, implantées sous forme de logique combinatoire) est beaucoup plus rapide mais sa taille est conséquente aussi. La version avec les S_boxes en BRAM est ralentie par les accès aux mémoires, mais est moins gourmande en ressources. Les lignes "All" indiquent la version de l'IP permettant de choisir la taille de clé de façon dynamique.

La protection du coprocesseur AES suivant les indications du paragraphe 2.3.5 s'est déroulée sans soucis particuliers. Les registres sensibles sont protégés avec un bit de parité par octet : l'importance des données qu'ils contiennent justifie le surcoût. Les mémoires RAM ou ROM utilisées pour implanter les S-boxes [Nist06] ont été aussi protégées de la même façon vue leur importance capitale pour la validité des résultats. La redondance temporelle a été implantée en modifiant les machines à états de contrôle. Leur complexité étant très importante (certaines comptent plusieurs dizaines d'états) on a préféré développer des versions dédiées qui sont instanciées suivant les options de configuration. Ce choix permet aussi de réduire au maximum la taille des machines à états, en obtenant donc des solutions plus légères et plus rapides.

Tableau 3-6 Résultats de synthèse par Leonardo pour le coprocesseur AES sur un VirtexII-Pro VP20

Taille de la clé (bits)	S_boxes en BRAM	Cycles pour codage	Cycles pour décodage	Fréquence maximale (MHz)	Occupation FPGA	
					LUTs	BRAMs
128	Non	70	130	44,29	8160	0
192	Non	82	122	45,85	9460	0
256	Non	104	142	42,86	9836	0
All	Non	***	***	45,23	11692	0
128	Oui	580	642	56,66	4454	16
192	Oui	584	624	47,98	5752	16
256	Oui	610	654	56,21	6124	16
All	Oui	***	***	47,28	7610	16
***: dépend de la taille de la clé						

Tableau 3-7 Surcoût des protections pour le coprocesseur AES

Taille de la clé (bits)	S_boxes en BRAM	Cycles pour codage	Cycles pour décodage	Fréquence maximale	Occupation FPGA	
					LUTs	BRAMs
128	Non	+100%	+80.7%	-14.5%	+36.5%	0
192	Non	+90.2%	+45.9%	-12.2%	+36.5%	0
256	Non	+82.6%	+66.2%	-10.9%	+39.9%	0
All	Non	***	***	-12.4%	+34.0%	0
128	Oui	+9.6%	+8.0%	-24.2%	+54.3%	16
192	Oui	+11.6%	+10.3%	-25.1%	+48.5%	16
256	Oui	+12.8%	+14.4%	-22.1%	+54.7%	16
All	Oui	***	***	-19.6%	+48.8%	16
***: dépend de la taille de la clé						

Le Tableau 3-7 résume les surcoûts des protections par rapport aux valeurs du Tableau 3-6.

On peut remarquer que dans la version avec les S_Boxes implantées en ROM le nombre de cycles nécessaires à une opération est presque doublé, bien en ligne avec la théorie de la redondance temporelle. Par contre la version avec S_Boxes en RAM est beaucoup moins touchée : il faut se souvenir que déjà en version de base elle est ralentie (~600cycles contre ~100) par les accès en RAM. Évidemment, cet effet est prépondérant pour déterminer le temps total d'exécution et masque l'effet de la redondance temporelle. De façon similaire, le surcoût d'occupation des ressources du FPGA est plus faible en pourcentage pour la version avec ROM, parce qu'une bonne partie de l'occupation est due justement aux S_boxes.

3.6 Sécurisation de la version logicielle de l'AES

La sécurisation de l'implantation logicielle de l'algorithme AES a été faite de façon automatique par le Politecnico di Torino avec leur outil. On n'a donc aucune connaissance détaillée des modifications et pour évaluer le surcoût des protections on va faire une analyse « en boîte noire » en comparant les exécutables. Le Tableau 3-8 présente les caractéristiques principales.

Tableau 3-8 Comparaison de la taille en octets de différentes implantations de l'algorithme AES

Section / Version	Text	Data	Bss	Taille totale
Originale	15544	104	14376	30024
Protégée	79256	24	28752	208032
Matérielle	2116	40	484	2640

La version « originale » indique le programme de base présent dans la bibliothèque Open-Source ([Devin04]). La version « Protégée » est le programme durci. La version « Matérielle » est le programme qui commande l'IP AES connectée sur le bus APB du processeur Leon2.

Le tableau, obtenu à partir des exécutables avec la commande « size », montre une répartition différente dans les sections qui composent l'exécutable ([ELF01]), certainement due à l'algorithme de protection, mais peu importantes dans une analyse « boîte noire ». Si on compare les tailles totales on voit que pour la protection purement logicielle la taille est presque 7 fois plus grande.

En ce qui concerne l'implantation matérielle, le logiciel pour la contrôler est très simple (il doit simplement écrire des mots sur le bus), et occupe 11 fois moins de place que l'original et 78 fois moins que la version protégée (notons que le logiciel de contrôle est le même pour l'IP de base et l'IP durcie). Ce gain en mémoire est à mettre en regard du coût d'implantation de l'IP matérielle, et s'ajoute à l'intérêt de l'accélération matérielle du point de vue des performances de traitement.

3.7 Conclusions

Dans ce chapitre, on a détaillé les interventions qui ont permis de durcir le système présenté dans le Chapitre 2. Dans certains cas, comme pour le coprocesseur AES, cela a été une simple application des techniques théoriques. Dans d'autres cas, comme la protection par parité des mémoires cache, la situation réelle s'est avérée légèrement plus complexe et donc on a dû raffiner l'approche. Dans le cas du pipeline de l'unité entière, il a également fallu adapter l'approche en mettant en place des stratégies de filtrage de signaux. Il faut rappeler que ceci a permis de démontrer le faible coût d'implantation d'un filtrage dynamique, qui peut être par ailleurs réalisé assez simplement s'il est prévu dès la conception initiale du bloc matériel. Un tel filtrage, réduisant le nombre potentiel de recouvrements, peut être intéressant à conserver dans la version finale du circuit. Enfin, dans le cas du système d'exploitation, l'application réelle a conduit à une solution plus souple et optimisée que l'application littérale du principe théorique. Dans le chapitre 4, on analysera en détail les effets de ces protections sur le système complet et le degré de sûreté ainsi atteint.

4 Prototype global : évaluation des coûts et de l'efficacité

Ce chapitre est dédié à l'évaluation des résultats obtenus en implantant, selon les méthodes décrites dans le Chapitre 3, dans un même prototype, les différentes techniques de protection proposées dans le Chapitre 2. Des analyses de surcoût ont déjà été présentées dans le chapitre 3, pour chaque méthode et chaque niveau d'intervention pris indépendamment des autres. Les résultats complémentaires donnés dans ce chapitre montrent tout d'abord les surcoûts obtenus dans le contexte du système global, à savoir une application, ordonnancée par l'OS, et exécutée sur le matériel défini au début du Chapitre 2. Ils montrent ensuite le niveau de couverture de fautes atteint sur ce prototype.

Le chapitre commence en détaillant dans le paragraphe 4.1 le flot de conception du prototype qui est à la base des évaluations et mesures des paragraphes suivants. Une méthode d'évaluation, cherchant à comparer de façon précise les surcoûts réels de l'implantation du point de vue des performances, est ensuite introduite dans le paragraphe 4.2. La méthode d'évaluation de l'efficacité en termes de couverture de fautes sera également présentée. Le paragraphe 4.3 sera ensuite consacré à l'analyse statique des surcoûts au niveau du système global. Un paragraphe à part, le 4.4, est dédié à l'analyse de l'algorithme AES dans ses deux implantations, logicielle et matérielle. Les limites d'une analyse statique de la couverture de fautes seront brièvement commentées dans le paragraphe 4.5 . L'analyse dynamique sera ensuite détaillée dans le paragraphe 4.6, aussi bien du point de vue des performances que de la couverture de fautes. Ce paragraphe décrira donc aussi brièvement l'environnement d'injection de fautes utilisé pour créer les conditions nécessaires à ces évaluations, ainsi que la spécification d'un environnement automatisé qui permettra dans le futur proche de continuer l'analyse et d'exploiter toutes les capacités du prototype.

4.1 Flot de conception global du prototype

4.1.1 Plateformes cible

Le système décrit dans le chapitre précédent a été implanté sur deux plateformes cibles, chacune adaptée à certaines applications. La première implantation a été faite sur une carte de la société Avnet, équipée d'un FPGA Xilinx Virtex II-Pro VP20. L'intérêt principal de cette carte est sa souplesse : la logique programmable est connectée à un bon ensemble de ressources (ports série, mémoires SRAM et SDRAM, Ethernet, connecteurs pour carte d'expansion, etc…), qui peuvent être exploitées avec facilité. Cela a été un atout dans les phases initiales du travail de doctorat puisque cela a permis d'avoir une grande liberté dans l'exploration des implantations possibles.

Après une exploration des différentes possibilités de la carte, on a défini un système opérationnel, illustré dans la Figure 4-1 : le processeur Leon2 est connecté au banc de SRAM pour exécuter l'application et est programmé à partir de son unité de debug intégrée (DSU) grâce à une connexion série. Le deuxième port série de la carte est exploité pour véhiculer les entrées/sorties standard qui donc ne risquent pas d'interférer avec les informations de debug. Si besoin, il est aussi possible d'exploiter la mémoire SDRAM (32MB), qui est connectée sur le même bus mémoire que la SRAM. La programmation de la carte FPGA est faite soit à partir de l'ordinateur hôte grâce à une connexion JTAG (non indiquée sur la figure pour plus de simplicité), soit à partir du contrôleur Flash dont la carte est équipée. La première solution est très pratique en phase d'expérimentation parce qu'elle permet de reprogrammer à la volée le FPGA avec les configurations souhaitées. La deuxième est plus adaptée, par exemple, à des démonstrations où l'on a besoin d'une exécution autonome.

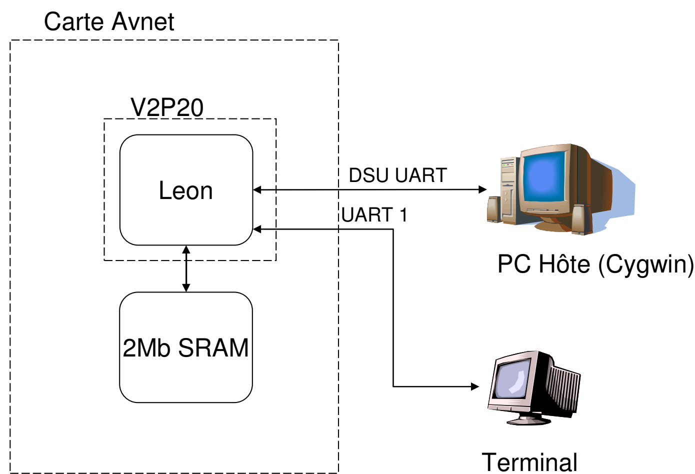


Figure 4-1 Implantation du prototype sur la carte Avnet

Cette configuration a permis d'effectuer des tests de non régression pour les modifications matérielles illustrées dans le paragraphe 3.3, en montrant que le processeur Leon2 modifié était parfaitement opérationnel. La connexion avec la DSU permet de déverminer, à partir du PC hôte, les programmes exécutés sur la carte grâce à la suite classique GNU dont le flot sera détaillé dans le paragraphe suivant.

Le prototype ainsi implanté a permis le développement et la validation de la protection dans le système d'exploitation (paragraphe 3.2). En particulier, on a pu valider le recouvrement suite à la génération d'une interruption, en la générant grâce à un bouton poussoir. Avec cette configuration, on a pu mesurer le temps nécessaire au recouvrement (Tableau 3-1), et valider les hypothèses de fonctionnement des sauts non locaux.

L'étape suivante à été le développement d'une démonstration temps réel, implantée dans un première temps sur la carte Avnet grâce à une carte d'extension équipée de convertisseurs A/N/A vidéo. Pour pouvoir paramétrer les contraints temps réel, on a décidé de ne pas connecter directement le Leon2 avec les convertisseurs, mais d'interposer un tampon mémoire, comme illustré dans la Figure 4-2.

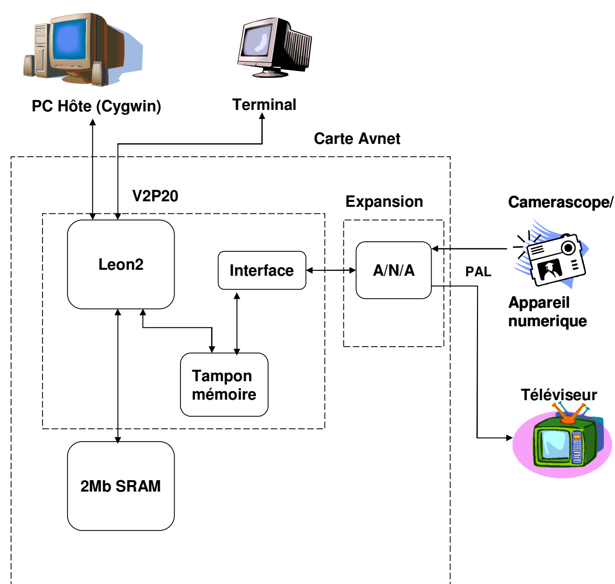


Figure 4-2 Prototype Vidéo sur la carte Avnet

En jouant sur le temps de rafraîchissement de l'image dans le tampon, on peut facilement changer les contraintes temps réel. Toutefois, on commence ici à atteindre les limitations de la carte Avnet, qui n'est pas conçue pour ce type d'application : le tampon mémoire a dû être implanté dans la logique programmable parce que toute la mémoire de la carte est connectée sur un seul bus, qui est déjà exploité par le processeur Leon2. Cela contraint de façon importante la taille du tampon et donc la taille de l'image traitée, qui est limitée à une centaine de pixels de largeur/hauteur, et doit être rigoureusement en noir et blanc.

On a donc choisi d'acquérir une autre carte spécifique pour les applications multimédia, la Xilinx Multimedia Board. Cette carte exploite un FPGA de type Virtex II XC2V2000 : la technologie programmable est la même que pour la version Pro, mais il n'y a pas de processeurs Power PC embarqués. Cela n'a pas de conséquence directe sur le prototype, qui ne les exploite pas, mais va avoir des conséquences en terme de stratégie d'injection de fautes (voir paragraphe 4.7). Les deux différences principales par rapport à la carte Avnet sont la taille de la logique programmable (presque doublée) et la mémoire disponible : la carte Multimedia est équipée de 5 bancs de SRAM de 5Mb chacun, accessibles de façon complètement indépendante. Le schéma de la Figure 4-2 peut donc être modifié en exploitant un de ces bancs en tant que tampon, pour obtenir la Figure 4-3. L'accès au tampon est réglementé par un registre directement inséré dans la plage d'adressage mémoire du Leon2. Ce registre peut aussi commander l'interface vidéo pour demander le rafraîchissement de l'image capturée.

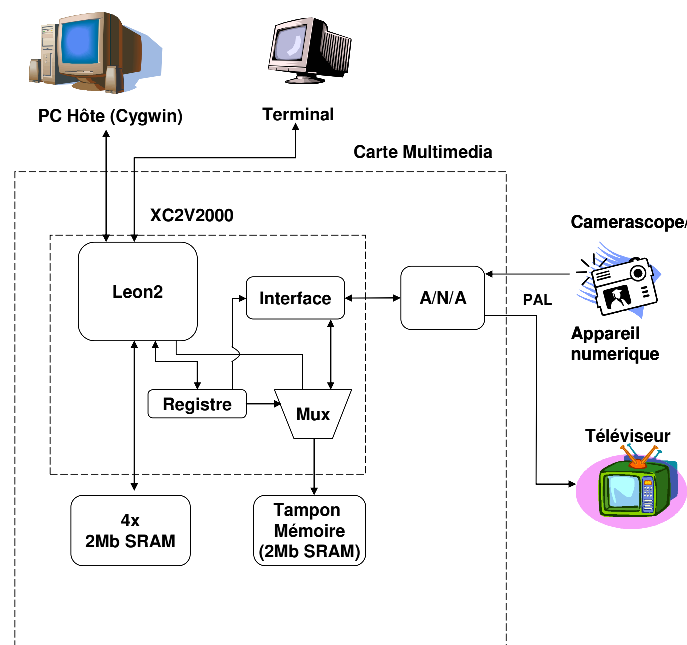


Figure 4-3 Prototype Vidéo sur la carte Multimedia

Le portage du prototype vers cette nouvelle carte a été très facile grâce à la similitude entre les technologies et au fait qu'aucune intervention n'est directement dépendante du système cible. Le seul contretemps a été avec les convertisseurs : ceux de la carte Multimedia étant complètement différents de ceux implantés sur la carte Avnet, il a fallu adapter la logique « ad hoc » d'interface. Les performances multimédia de cette configuration sont très intéressantes : la taille du tampon permet de sauvegarder l'image entière en couleur, même si souvent on se limite à un sous-ensemble carré. Par exemple, la capture d'écran de la Figure 3-7 a été obtenue à partir de ce prototype.

4.1.2 Flot de conception matériel

Le flot de conception matériel part de la description VHDL du processeur Leon2 et des autres composants du prototype (interface vidéo, AES, etc.). Tous les composants sont configurables : par cohérence, on a adopté le style de configuration de Leon2 pour toutes les modifications et les composants ajoutés. Cela nous a permis aussi d'intégrer les nouveaux paramètres dans l'interface graphique de configuration de Leon2. Tous les paramètres liés à la robustesse sont donc complètement intégrés dans le flot de configuration, comme ont peut le voir dans la Figure 4-4.

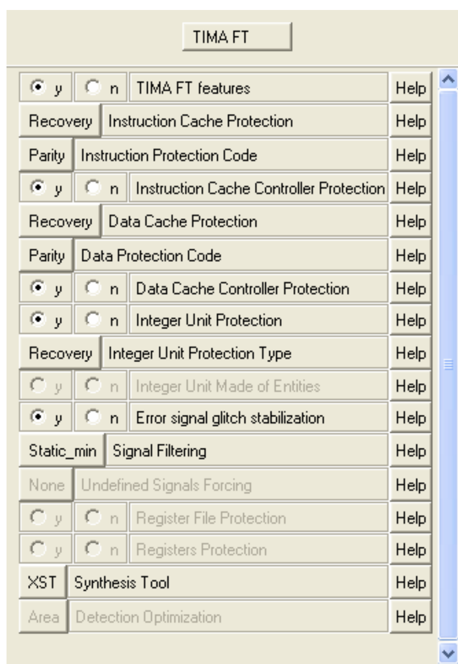


Figure 4-4 Interface graphique de configuration des protections matérielles

Une fois configuré, le VHDL peut être compilé et simulé avec les outils classiques, par exemple avec Modelsim. Cela permet, entre autres, d'exploiter les environnements de validation (testbench) de Leon2 comme tests de non-régression. Le style de codage VHDL ne présente aucune spécificité et les sources peuvent donc être synthétisées avec tous les outils principaux : dans le cadre de ce travail de doctorat, on a effectué des synthèses avec Leonardo, Design Vision et XST.

Si on veut arriver jusqu'à la programmation du FPGA, la synthèse ne suffit pas : Xilinx fournit la suite logicielle ISE qui se charge de toutes les étapes à partir de la synthèse avec XST jusqu'à la génération du fichier de programmation, le bitfile. Si nécessaire, il est toujours possible d'intégrer des netlists Edif obtenues à partir d'outils de synthèse tiers, comme Leonardo par exemple.

Ce flot de développement matériel, résumé dans la Figure 4-5, est complet et parfaitement opérationnel, même si la complexité du système source tend à le pousser à ses limites, le rendant parfois un peu instable et très dépendant des options de configuration fines. Par exemple, le même projet qui génère des bitfiles parfaitement opérationnels avec une synthèse « à plat » peut générer des bitfiles erronés si on demande le maintien de la hiérarchie. Cependant, une fois l'outil correctement configuré pour une version donnée du projet, il reste compatible avec toutes les modifications du projet.

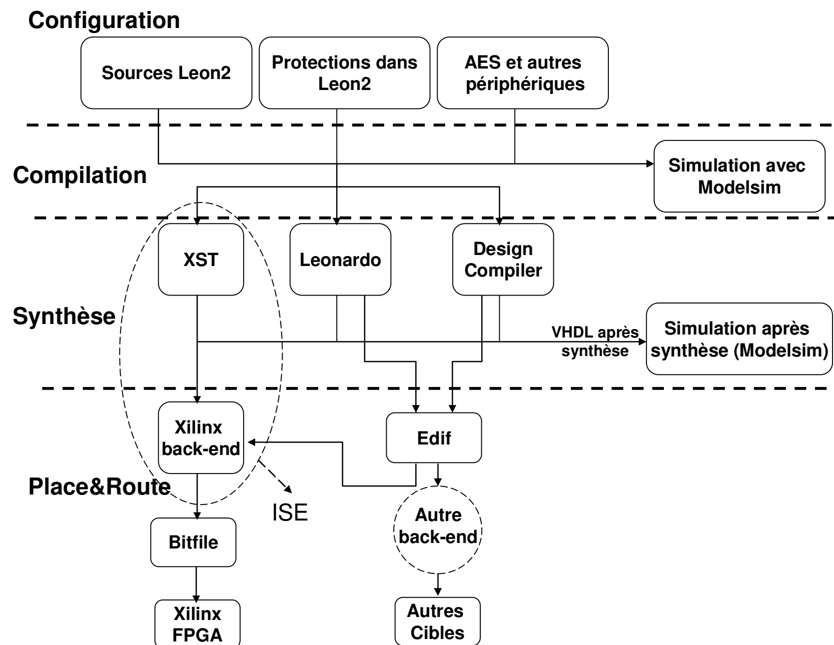


Figure 4-5 Flot de développement matériel

Parmi toutes les options des outils composant le flot, l'une d'entre elles est très importante : le partage de ressources. Généralement, les outils de synthèse cherchent à identifier toutes les redondances dans le code source pour les simplifier et obtenir un composant final plus petit et plus performant. Mais l'effet est aussi que les redondances introduites volontairement pour durcir le système sont éliminées, faussant les résultats. XST en particulier est très agressif en ce qui concerne la recherche d'optimisations : lors des premiers essais de synthèse de l'unité entière protégée (paragraphes 2.3 et 3.3), on a obtenu des surcoûts extrêmement faibles, de l'ordre de quelques pourcents à peine. Dans tous les outils (XST, Leonardo, etc.) il est toutefois possible de désactiver complètement le partage de ressources, mais cela signifie en général un gros surcoût en terme d'occupation de surface. En plus, pour ISE, cela correspondait à l'un des cas d'instabilité, générant des bitfiles inutilisables. Heureusement, il est possible d'indiquer de façon plus précise à XST les composants pouvant être optimisés et ceux que l'on veut garder pendant la synthèse même s'ils sont redondants. Avec ce type de configuration de la synthèse, il est possible de garantir que les protections implantées au niveau RTL sont bien toujours présentes dans la description niveau portes ; les résultats de synthèse des tableaux 3-3 et 3-4 on été obtenus avec ces options, et le gain de surface entre une duplication totale (100%) et les résultats affichés est bien complètement dû à la simplification du composant de prédiction de parité (paragraphe 2.3.4 et [Ko04]), pas à une suppression des protections.

4.1.3 Flot de conception logiciel

Le choix de Leon2 comme processeur cible permet d'exploiter une suite logicielle complète, développée par Gaisler Research et compatible avec le flot standard GNU. Cet aspect est souvent sous-estimé par les architectes de systèmes, mais la spécification et le développement d'un compilateur efficace pour un nouveau processeur sont loin d'être triviaux. La suite de développement existante, appelée « sparc-elf », contient tous les outils classiques de développement logiciel : compilateur, éditeur de liens, debugger, etc. Son environnement naturel est Linux, mais comme la plupart des outils du flot matériel (voir

paragraphe précédent) sont sous Windows on a préféré faire un choix de compatibilité. On a donc installé Cygwin, qui permet d'émuler un système « Linux-like » sous Windows sans avoir besoin de redémarrer la machine à chaque fois.

Le système d'exploitation eCos est librement disponible sur Internet soit dans des archives compressées soit en forme collaborative CVS (un système qui permet à plusieurs utilisateurs distants de travailler sur les mêmes sources et de partager les résultats). Une fois copié dans le répertoire Cygwin, on peut le configurer grâce à une interface graphique : si les modifications et les nouveaux fichiers (voir paragraphe 3.2.2) sont implantés selon la sémantique de eCos, les fonctions de protection apparaissent directement dans l'interface graphique, comme on peut le voir en Figure 4-6.

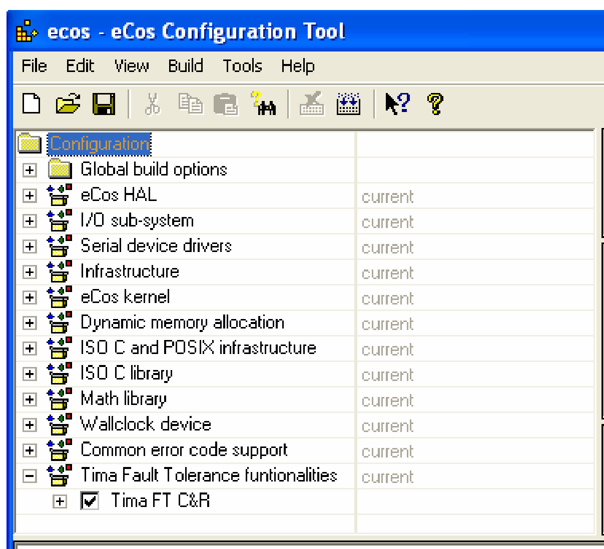


Figure 4-6 Outil de configuration de eCos, avec les options de protection

Une fois le système d'exploitation configuré, on peut le compiler avec la suite « sparc-elf » et les makefiles générés automatiquement par l'outil de configuration et obtenir des bibliothèques dynamiques, ce qui termine le flot OS.

Les applications logicielles peuvent être développées à part : la connexion avec les bibliothèque dynamiques est faite par l'éditeur de liens au moment de la compilation.

Une fois le fichier exécutable obtenu, son exécution peut être simulée grâce à GRMON, un simulateur de Leon2 fourni par Gaisler et qui peut être connecté à gdb pour un debug complet. GRMON peut aussi être connecté à la DSU de Leon2 à travers une connexion série, comme indiqué dans les Figure 4-2 et 4-3, ce qui permet d'exécuter et déboguer le programme directement sur le processeur réel.

La Figure 4-7 résume le flot logiciel.

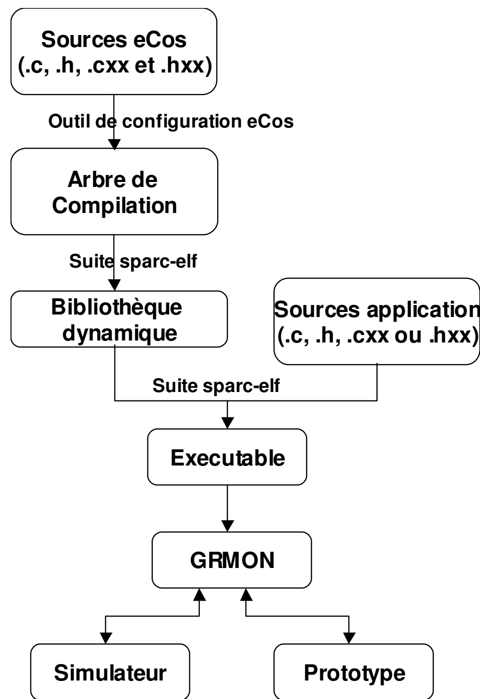


Figure 4-7 Flot de développement logiciel

4.1.4 Versions du prototype

Grâce à ses nombreux paramètres, le prototype peut assumer un grand nombre de configurations possibles. On va ici résumer l'ensemble des options possibles : à noter que toute combinaison des paramètres est possible.

- Options de Leon2 : toutes celles du processeur original ;
- Options de protection matérielle :
    - Détection ou recouvrement dans la mémoire du cache instructions ;
    - Détection ou recouvrement dans la mémoire du cache données ;
    - Détection dans le contrôleur du cache instruction (SET et SEU) ;
    - Détection dans le contrôleur du cache données (SET et SEU) ;
    - Détection dans l'unité entière
        - Filtrage des signaux : dynamique, statique minimal ou maximale ;
    - Recouvrement dans l'unité entière
        - Filtrage des signaux : dynamique, statique minimal ou maximale ;
    - Forçage des signaux non définis (pour simulation avant synthèse) ;
    - Injection de fautes dans l'unité entière : SET (paragraphe 4.7.2) ou forçage par bouton poussoir ;
    - Détection dans les registres internes de l'unité entière ;
    - Détection dans le banc de registres ;
    - Unité entière optimisée pour XST ou Léonardo ;
    - Unité entière optimisée pour surface ou vitesse ;
- Options de protection logicielle :
    - Recouvrement sur la base du changement du contexte (demande au moins une option active de détection matérielle ou logicielle) ;
    - Options de debug (traces d'exécution).

4.2 Stratégie de mesure

Quand on veut analyser un système, plusieurs aspects doivent être pris en compte en compte. Premièrement, il est important de connaître sa taille, c'est à dire la quantité de ressources dont il a besoin pour fonctionner correctement. Cet aspect est encore plus important dans les milieux embarqués où les ressources disponibles sont limitées. L'autre aspect fondamental est la capacité du système à effectuer dans un temps donné les tâches qui lui sont assignées, ce qu'on appelle généralement ses performances. Plus un système est performant, plus il sera capable d'exécuter des tâches complexes et donc mieux il pourra répondre aux contraintes de son environnement. Dans le cadre qui nous intéresse, un troisième aspect fondamental est la couverture de fautes, c'est à dire le pourcentage des fautes possibles qui vont pouvoir être détectées et correctement traitées par le système.

Le système cible de toute cette thèse est un système embarqué complexe, composé par un processeur, un système d'exploitation, une application logicielle et des IP dédiées. Chacun de ces éléments a sa taille et ses performances souvent exprimées de façon très différente. Il est donc très important de définir un cadre d'analyse unifié pour pouvoir comparer les caractéristiques de chacun et trouver la solution optimale. En particulier, l'évaluation des surcoûts en performances doit permettre d'identifier la contribution des différentes interventions réalisées à des niveaux très différents ; c'est l'objectif de la méthodologie proposée dans la section suivante.

4.2.1 Mesures de performances

Le concept de performances est intuitivement très facile, mais assez difficile à cerner de façon mesurable et donc scientifique, même si on se limitera aux performances au sens puissance de calcul, sans faire intervenir d'autres caractéristiques comme par exemple la consommation. L'analyse est encore plus difficile dans les cas comme le notre où on travaille sur plusieurs niveaux d'abstractions. Dans le cadre de ce travail de thèse, on a donc décidé de s'appuyer sur la définition intuitive de « capacité du système à effectuer des tâches » pour ensuite la décliner selon les besoins de chaque contexte d'analyse.

Si on considère le **niveau matériel, l'architecture du processeur et du système étant fixées (le niveau de parallélisme étant donc constant),** la capacité d'un circuit à effectuer une tâche en un temps donné est directement liée à sa **fréquence de travail maximale**. Plus l'horloge est rapide, plus le nombre d'opérations terminées dans un intervalle de temps donné sera grand.

Le discours est similaire pour le **niveau logiciel** : une application est plus performante si elle arrive à effectuer la même tâche en moins de temps. Une augmentation de la fréquence de travail du processeur va bien sûr avoir un impact. Toutefois, une fois les performances du matériel fixées, celles du logiciel vont essentiellement dépendre du **nombre d'instructions exécutées et de leur type**. Si l'application est trop complexe pour estimer a priori le flot d'exécution complet (et donc la suite d'instructions exécutées) il faut alors mesurer directement le **temps d'exécution** de la tâche. L'avantage d'une mesure réelle sur un prototype est la possibilité de considérer aussi les aléas d'exécution (dépendances entre instructions, accès mémoire, etc…) qui peuvent empêcher le pipeline de terminer une instruction par cycle d'horloge.

Les performances d'un **système d'exploitation** sont beaucoup plus difficiles à mesurer parce que son comportement n'est pas sensé être prédictible à l'avance. Son rôle est

d'exécuter plusieurs tâches en parallélisme simulé et de fournir des fonctions spécifiques (synchronisations, partage de ressources, etc.…). Une **façon indirecte** d'évaluer ses performances est de mesurer la **pénalité d'exécution d'une tâche** : si on compare le temps d'exécution d'une tâche seule et d'une tache insérée dans le système d'exploitation on peut se rendre compte du surcoût généré par ce dernier. Si par contre on est intéressé par une **estimation directe** des capacités d'un système d'exploitation on peut mesurer **le temps d'exécution de ses fonctions de base** (changement de contexte, gestion des sémaphores, alarmes, etc.…). Un système d'exploitation sera d'autant plus performant qu'il est rapide pour effectuer les opérations qu'on lui demande. On va privilégier cette deuxième stratégie de mesure parce qu'elle permet de faire abstraction de la tâche exécutée.

4.2.2 Couverture de fautes

Estimer les capacités d'une technique de protection envers les fautes signifie savoir la probabilité qu'une faute survenant dans le système soit détectée/corrigée avant de se propager jusqu'à la sortie et de provoquer une défaillance. Dans notre cas, la correction d'une erreur est basée sur une réaction engagée après détection ; la couverture de fautes est donc essentiellement liée à la capacité de détection du système. Si on considère le comportement d'un système soumis à une faute et si on trace le couple (résultat, détection) on peut construire le graphe de la Figure 4-8.

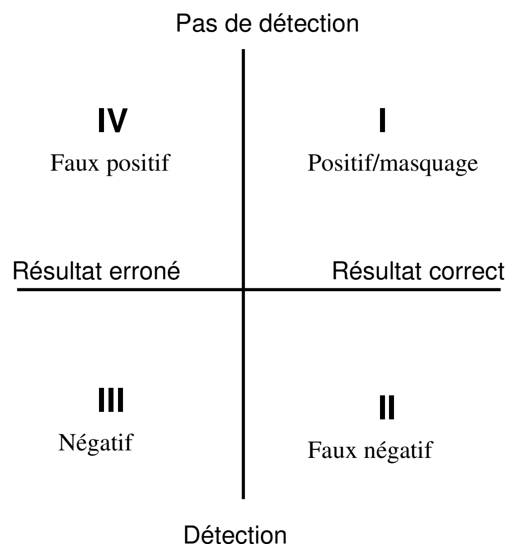


Figure 4-8 Classification des fautes

- Le quadrant I montre la situation normale : il n'y a pas de détection et le résultat est correct. On appelle ce cas « positif » parce que c'est le comportement nominal du système en absence de fautes. Il est aussi appelé « masquage » car c'est le comportement observé si des techniques de tolérance par masquage sont employées (par exemple, un TMR sans génération de signal de détection) et si les fautes survenant dans le système ne dépassent pas les capacités de tolérance.
- Le quadrant III montre la situation typique de détection : le résultat est erroné et le signal de détection est actif. On appelle ce cas « négatif » parce que le résultat du système est incorrect ;
- Le quadrant II montre une situation de fausse détection : le signal de détection est activé alors que le résultat est finalement correct. C'est un « faux négatif » parce qu'on va engager une action de correction qui n'était pas nécessaire.

- Le quadrant IV montre au contraire un échec de la protection : il n'y a pas eu de détection alors que le résultat est erroné. C'est un « faux positif » parce qu'on va accepter comme bon un résultat incorrect.

La classification des différentes fautes du modèle selon ces quatre quadrants peut être faite dans certains cas a priori, sur la base d'une analyse des propriétés des protections implantées et en fonction du modèle de fautes considéré. Une telle analyse de notre système sera brièvement discutée dans le paragraphe 4.5. Toutefois, dans différents cas, il est très difficile de prévoir exactement la classification de toutes les fautes de manière purement analytique. Ceci est d'autant plus vrai lorsque des erreurs multiples (modèle MBF présenté au chapitre 1) sont prises en compte. Il est alors nécessaire de réaliser des expérimentations permettant d'observer le comportement du système en présence de fautes. Une telle campagne de mesures, dite campagne d'injection de fautes, sera le sujet du paragraphe 4.6.

4.3 Performances : analyse statique

Les coûts statiques des protections peuvent être mesurés assez facilement en termes de perte de vitesse et d'augmentation de la taille. Cette dernière est notamment très facile à évaluer pour les protections matérielles : les éléments ajoutés augmentent les ressources nécessaires et donc la taille, ce qui correspond aux mesures données dans le Chapitre 3.

Si on cherche à analyser les performances, le travail est plus difficile dans un système complexe où plusieurs éléments développés à des niveaux d'abstraction différents interagissent pour déterminer le comportement final. Les mesures faites sur chaque composant pris individuellement peuvent ne pas être suffisantes pour déterminer les effets globaux dans une application réelle. Par exemple, dans le cas de modifications au niveau du système d'exploitation, les mesures du Tableau 3-1 sont importantes mais ne montrent pas le cadre complet. Il est impossible de déterminer complètement a priori l'évolution de l'échéancier, et donc de savoir combien de fois les instructions ajoutées vont être exécutées.

On va donc introduire la stratégie d'analyse globale de performances proposée dans le cadre de cette thèse, et qui a été le sujet d'une publication lors de la conférence internationale IOLTS 2005.

4.3.1 Point communs

Pour comparer les performances de deux versions d'un système, notamment la version initiale et une version protégée, il faut définir des points communs qui peuvent être retrouvés dans les deux situations. Il faut surtout que la **protection** soit la seule vraie **différence** entre les deux systèmes.

Le premier point fixe doit être la **charge de travail** ou **testbench** : les deux systèmes doivent effectuer exactement la même tâche, sinon il sera impossible savoir si les différences sont dues aux protections ou au changement de la tâche. Cette charge peut être l'application finale (on l'appelle alors une « application réelle »), ou un programme développé expressément dans le but de mesurer certaines capacités du système (c'est le cas d'un « testbench synthétique » ou « benchmark »). On ne veut pas rentrer ici dans le débat sur le choix d'un testbench optimal ([Gwenn94] ou [Kewel04]) : notre intérêt est dans la **variation** des performances, pas dans leur valeur absolue. Si un testbench est biaisé par quelque effet particulier dans le système d'origine, il le sera aussi dans le système protégé. C'est une

question d'honneur intellectuel du développeur de choisir un testbench réellement représentatif.

Dans les applications réelles, un circuit ne travaille souvent pas à sa fréquence maximale f_{max}, mais plutôt à une fréquence $f_{tra} < f_{max}$. Les raisons sont multiples : l'utilisation d'une marge de sécurité, les oscillateurs disponibles, des soucis de synchronisation avec d'autres composants, etc. Dans nos prototypes, par exemple, le Leon2 peut être connecté à un oscillateur à 33Mhz, mais pour les applications vidéo il est préférable de le connecter à l'horloge à 27Mhz extraite du signal d'entrée pour mieux le synchroniser avec les IPs d'acquisition.

Dénotons par f_{pro} la fréquence maximale de l'IP protégée, et par So(f) et Sp(f) le système original et le système protégé respectivement, travaillant à fréquence f. Deux cas sont possibles :

1) $f_{tra} < f_{pro} < f_{max}$ le système va fonctionner avec la même vitesse que sa configuration originale.
2) Si par contre $f_{pro} < f_{tra} < f_{max}$ alors le système va fonctionner à vitesse réduite, $f_{tra2} \leq f_{pro}$.

Le cas **1)** est le plus simple : on peut directement exécuter le testbench sur $So(f_{tra})$ et $Sp(f_{tra})$ et comparer les résultats pour connaître la **perte de performance intrinsèque**. La fréquence de travail étant la même, seuls les effets statistiques et d'interaction entre techniques peuvent causer cette perte.

Le cas **2)** est plus complexe parce qu'il y a deux effets qui se superposent : la **perte de performance absolue** et la **perte intrinsèque**. La perte absolue indique la différence de performances avec le système original travaillant à fréquence f_{tra}, et regroupe donc le ralentissement dû à la différence de fréquence (f_{pro}-f_{tra2}/f_{pro}) avec les effets intrinsèques du cas 1). La perte absolue va pouvoir être mesurée en comparant la durée d'exécution du testbench sur $So(f_{tra})$ et sur $Sp(f_{tra2})$. La perte intrinsèque va pouvoir être mesurée en mettant les deux systèmes dans les mêmes conditions, donc en comparant la durée d'exécution du testbench pour $So(f_{tra2})$ et $Sp(f_{tra2})$.

4.3.2 Application au prototype : performances de eCos

Pour avoir une idée de l'impact des modifications dans l'OS, on a décidé d'exécuter une charge de travail composée par une batterie de tests qui mesurent des grandeurs typiques d'un système d'exploitation, en le compilant dans la version originale de l'OS et dans la version avec le C&R. Les tests sont répartis en 6 groupes :

- échéancier : on mesure les grandeurs typiques de l'échéancier, comme naturellement le changement de contexte, mais aussi la création et destruction des threads, etc. ;
- mutex : création, destruction, accès, etc. ;
- boîtes à lettres : création, destruction, dépose et réception de messages, etc. ;
- sémaphores : création, destruction, p, v, etc. ;
- compteurs et flags : création, destruction, délai de notification, remise à zéro, etc. ;
- alarmes : création, destruction, notifications, attentes variables, etc.

Les graphes qui vont suivre montrent le temps moyen (en ordonnées) pour effectuer chaque test (en abscisses). Le fait de relier chaque échantillon dans une ligne plutôt que de laisser des points séparés est simplement un choix de visibilité, chaque test étant indépendant.

Si on fait référence à la méthode introduite dans le paragraphe précédent, on est dans le cas 1) : la modification du système d'exploitation ne cause aucune diminution de fréquence de travail. On va donc effectuer deux exécutions du testbench à la même fréquence (33MHz).

Dans la Figure 4-9 on peut voir que toutes les opérations sont perturbées, même si en moyenne la différence (ligne rouge) est très petite, de l'ordre de la milliseconde. En modifiant le changement de contexte, on a donc introduit une petite **perte intrinsèque** dans le **comportement statistique** du système d'exploitation, qui n'était pas estimable a priori en se basant juste sur la théorie.

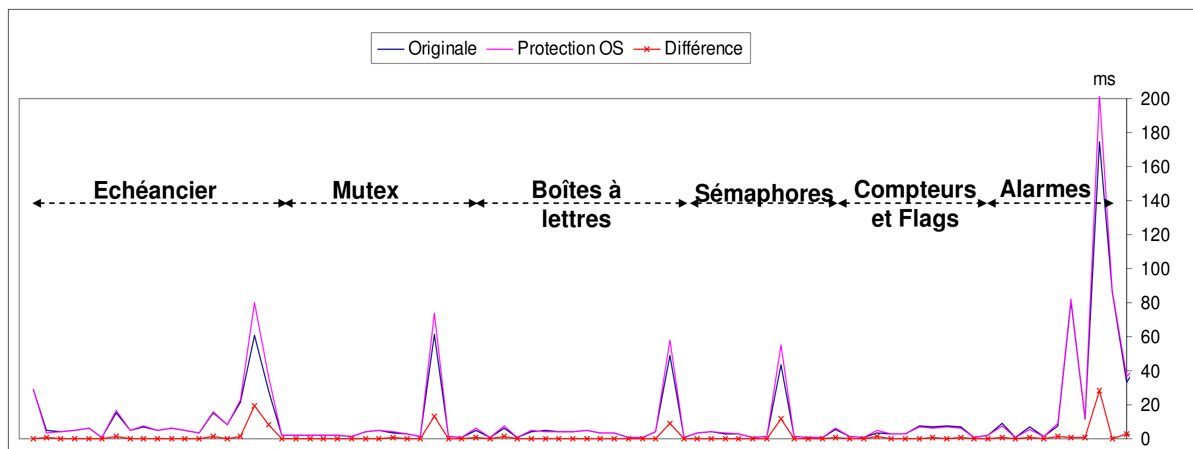


Figure 4-9 Comparaison des grandeurs de base de eCos en configuration originale et avec C&R

Si on applique au système la détection matérielle, on reste toujours dans le cas 1) parce que $f_{tra} < f_{pro} < f_{max}$. La Figure 4-10 montre les résultats obtenus en exécutant la même batterie de tests de l'OS.

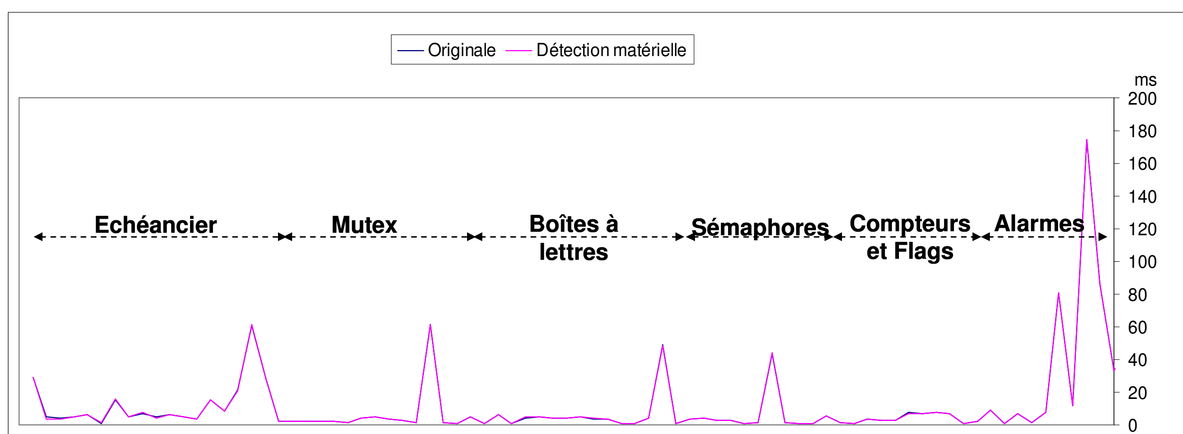


Figure 4-10 Comparaison des grandeurs de base de eCos en configuration originale et avec détection matérielle

Les deux graphes sont pratiquement complètement superposés : la protection matérielle n'a donc aucune influence sur le comportement statistique du système d'exploitation. On peut donc conclure que dans les conditions de travail de notre prototype, et en l'absence de fautes, la détection matérielle n'interagit pas avec le système d'exploitation et n'entraîne donc **aucune perte absolue** de performances.

Si par contre on observe le cas du recouvrement matériel, on a $f_{pro} < f_{tra} < f_{max}$, et le système va devoir être connecté à l'horloge à 27Mhz ce qui correspond au cas 2). La Figure

4-11 montre les résultats. On peut observer que les profils des deux courbes sont équivalents, même si en valeur absolue la configuration avec recouvrement est plus lente. Cela conduit à supposer que la perte absolue de performances est complètement due à la diminution de la fréquence de travail, et pas à des raisons intrinsèques.

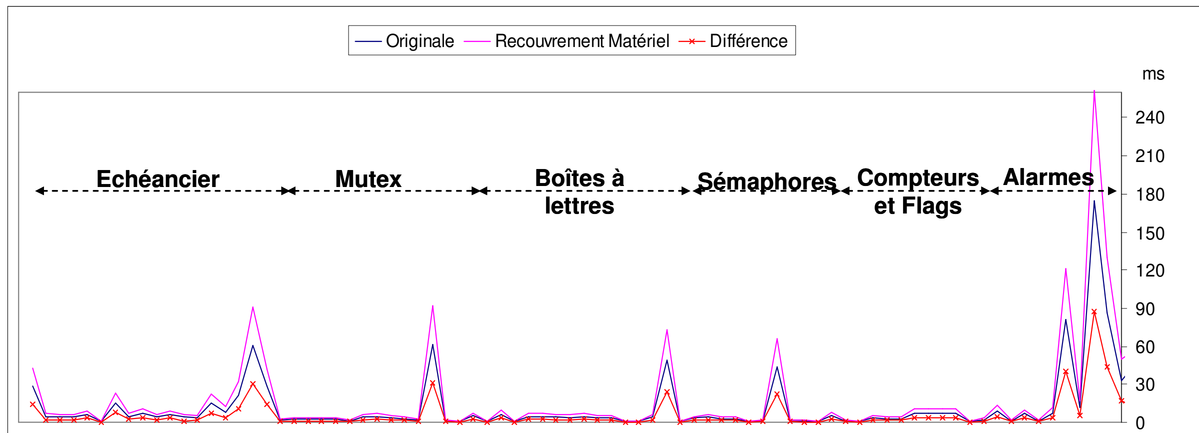


Figure 4-11 Comparaison des grandeurs de base de eCos en configuration originale et avec recouvrement matériel

Cette hypothèse est vérifiée par la comparaison des temps d'exécution du testbench exécuté sur So(27Mhz) et Sp(27Mhz), dont les résultats sont tracés dans la Figure 4-12.

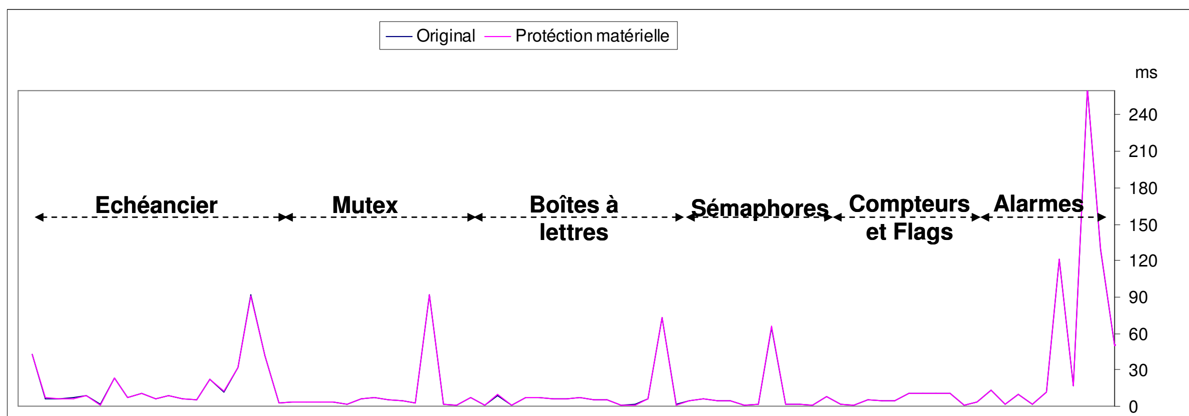


Figure 4-12 Comparaison des grandeurs de base de eCos en configuration originale et avec recouvrement matériel à la même fréquence de travail

Les deux courbes sont parfaitement superposées comme dans la Figure 4-10, donc la perte intrinsèque est nulle. Si jamais on arrivait à optimiser assez les protections pour obtenir $f_{pro} > f_{tra}$ on pourrait neutraliser complètement les pertes de performances dans nos conditions de travail, au moins pour ce qui concerne les coûts statistiques. On s'attend par contre à voir des fluctuations statistiques au moment de l'injection des fautes, quand les techniques de recouvrement vont modifier le comportement du pipeline et des mémoires caches (paragraphes 2.3.5 et 2.3.3.2 respectivement).

4.4 Cas particulier : comparaison des implantations logicielle et matérielle du chiffrement AES

Dans le cas de l'algorithme AES, on a la possibilité de faire des comparaisons entre une implantation purement logicielle et sa contrepartie accélérée en matériel. Les deux versions ont été protégées suivant des techniques adaptées (paragraphes 2.4 et 2.3.6 respectivement), donc en première analyse on peut comparer les 4 configurations. Dans le Tableau 4-1, on résume la comparaison de certains temps d'exécution réels des différentes versions, mesurés sur le prototype avec une horloge de 27 MHz.

Tableau 4-1 Temps d'exécution de l'algorithme AES à 27Mhz avec une clé de 128 bits

Version	Codage (µs)	Décodage (µs)
Logicielle originale	119,13	114,32
Logicielle protégée	1116,94	1049,56
Matérielle	28,70	21,56
Logiciel de commande de l'IP	27,32	20,12

Les trois premières lignes du Tableau 4-1 font référence aux mêmes configurations que dans le Tableau 3-6, la dernière ligne indiquant le temps d'exécution pour le logiciel de commande de l'IP, mais sans attendre le résultat fourni par celle-ci. On peut voir que cette partie est largement prépondérante dans le temps de calcul total. Cela ne doit pas surprendre : les 70 cycles d'un codage avec une clé de 128 bits (Tableau 2-1), par exemple, à 27Mhz correspondent à 2,6µs. Dans le pire cas (clé de 256 bits, implantation en RAM et IP durci, donc 654 cycles) l'IP va demander 24,2 µs pour exécuter un calcul. Le pilote logiciel de son coté va devoir écrire à travers le bus APB les données à chiffrer puis lire le résultat (128 bits chaque fois), ce qui va prendre un certain temps sur le bus de périphériques APB, simple mais à faibles performances. Dans une implantation orientée performances, il serait donc intéressant de connecter l'IP AES au bus à hautes performances AHB ([AMBA99]) pour diminuer les temps de communication.

Si on compare les résultats on voit qu'avec la protection purement logicielle le temps d'exécution est multiplié par 9, ce qui implique une perte de performances considérable. Dans [Nicol04] et [Berna06] cela est justifié par l'excellent niveau de sécurité atteint : il sera intéressant de voir si les techniques présentées dans cette thèse permettent de définir un compromis moins pénalisant mais avec une sécurité comparable.

4.5 Couverture de fautes : analyse statique

La capacité de couverture de fautes est un facteur fondamental pour évaluer l'efficacité d'une protection, comme dit dans le paragraphe 4.2.2. Une estimation statique est en partie possible en analysant les propriétés des techniques choisies et de leur implantation.

Il est par exemple facile de savoir qu'un registre protégé par code de parité pourra détecter toute faute causant un nombre impair d'inversions de bits, mais sera incapable de détecter les erreurs de multiplicités paires (cas du type "faux positif"). Un faux négatif peut être généré par une inversion du seul bit de parité ou par un SET survenant dans la logique de décodage. Cela s'applique à tous les composants séquentiels protégés : registres internes du pipeline, banc de registres, registres des contrôleurs des caches, mémoires caches, registres de

l'IP AES. Pour un autre code séparable, l'analyse serait la même : les faux négatifs proviennent de modifications restreintes aux bits de vérification du code ou à la logique de décodage. Il faut noter que dans le cas de l'utilisation d'un code non séparable, l'analyse serait beaucoup plus complexe.

Considérons maintenant la protection de la logique combinatoire par prédiction de code (prédiction de parité dans notre cas, cf. Figure 2-11, reproduite en Figure 4-13 pour plus de clarté). Dans ce cas, une analyse théorique n'est pas suffisante. En effet, si on considère un évènement singulier causant l'inversion d'un signal S dans le bloc L, on ne peut pas savoir a priori combien de signaux de sortie vont être modifiés par l'erreur sur S, qui plus est au moment où ces signaux de sortie sont échantillonnés. Pour déterminer cela, il est nécessaire de connaître la structure exacte au niveau portes, et les temps de propagation exacts dans le réseau. Ceci ne peut être obtenu que très tard dans le flot de conception. Une approche peut consister à intervenir sur le processus de synthèse de manière à garantir que seules des erreurs de multiplicités impaires sont possibles [Touba97]. En l'absence d'une telle intervention, le taux de couverture des fautes de type SET ne peut pas être évalué. Une fois le circuit placé et routé, voire implanté sur un FPGA, il est possible d'obtenir une évaluation par exemple en réalisant une campagne d'injection de fautes ou un outil comme Roban [Alexa02] .

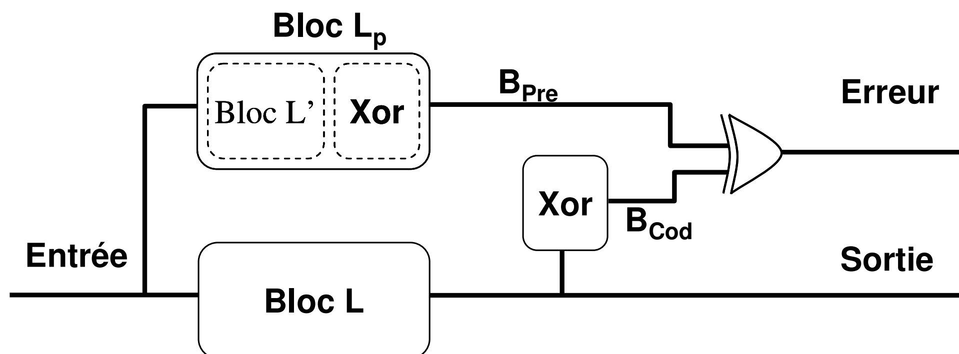


Figure 4-13 Reproduction de la figure 2-11 "Protection par prédiction de parité automatisée"

Indépendamment des taux de couverture atteints, il est possible de donner une estimation de la différence de couverture entre les 3 stratégies de filtrage de signaux (paragraphe 3.3), sur la base du nombre de signaux pris en compte par les protections. Le Tableau 4-2 montre, pour les 5 étages du pipeline de l'unité entière, le pourcentage de signaux étant au moins une fois indéfinis et donc n'étant pas pris en compte lors du test en ligne avec un filtrage statique minimal. Hormis l'étage de recherche d'instruction, ce pourcentage varie entre 53% et 88%. Ces chiffrent peuvent être interprétés de deux façons. Ils donnent tout d'abord un ordre de grandeur de la réduction du taux de couverture (donc de l'augmentation du nombre de faux positifs) si un filtrage statique minimal est maintenu dans le circuit final. Les coûts d'implantation très réduits présentés au chapitre 3 pour ce type de filtrage sont à mettre en regard de ces valeurs. Les mêmes chiffrent donnent également une idée de l'augmentation de la probabilité de faux négatifs si un filtrage statique maximal est maintenu dans le circuit final. Pour obtenir un meilleur ordre de grandeur de cette probabilité, il faudrait cependant analyser finement le nombre de cas dans lesquels chacun des signaux est indéfini.

Tableau 4-2 Estimation des bits pris en compte selon la stratégie de filtrage

Etage \ Filtrage	Dynamique et Statique Maximal	Statique Minimal	Différence
Fetch	151	151	0
Decode	310	147	163 (53%)
Execute	510	107	403 (79%)
Memory	267	99	168 (63%)
Write	138	17	121 (88%)
Totale	1376	521	885 (62%)

L'objectif des campagnes d'injection de fautes qui seront présentées dans la suite de ce chapitre sera notamment d'évaluer la robustesse atteinte dans les cas où le raisonnement analytique ne permet pas de conclure.

4.6 Efficacité de la collaboration OS-matériel : analyse dynamique

L'analyse des coûts de protection dynamiques nécessite l'activation des protections. Attendre simplement qu'une faute se déclenche naturellement n'est pas une solution possible, cela prendrait trop de temps et, surtout, ne serait ni contrôlable ni répétable.

Une solution classique et efficace consiste à accélérer le processus naturel : le circuit sous test ou CUT (Circuit Under Test) est soumis à des radiations ou faisceaux de particules, ce qui permet d'observer un nombre important d'événements. Ce type de test a été, par exemple, exploité dans [Cheyn00] pour évaluer la méthode de protection logicielle proposée. Cette méthodologie permet de contrôler avec précision l'intensité des radiations et le type de particules projetées, ainsi que la focalisation du faisceau sur la puce (mais pas la localisation exacte des erreurs induites). Sa principale faiblesse est le coût de réalisation : il faut avoir une source radioactive, ce qui implique d'obtenir un créneau dans une installation spécialisée, ce qui est en général très cher, implique une durée des tests très limitée et une difficulté de répétition de l'expérience. La maquette de test doit être aussi spécialement conçue pour que les composants de mesure ne soient pas eux aussi affectés par les radiations. Une autre limitation très importante est que pour pouvoir réaliser les tests il faut avoir à disposition le circuit final, implanté dans la technologie cible. Ces tests se placent donc tout à la fin du cycle de développement, en phase de qualification : s'ils identifient des problèmes, la correction peut être très coûteuse. Par ailleurs, les résultats ne sont significatifs que pour la technologie employée. Un circuit devant être fabriqué dans une technologie précaractérisée ne peut donc pas être qualifié avant sa fabrication dans cette technologie, par exemple en utilisant un prototypage sur réseau programmable. Chaque technologie réagit de façon différente aux radiations, donc les résultats obtenus avec une technologie ne sont pas forcément valables pour une autre.

En conséquence, de nombreuses méthodes ont été développées ces dernières années pour réaliser des injections de fautes pendant le cycle de développement, avant la fabrication du circuit. Ces techniques peuvent être basées sur des simulations ou de l'émulation sur réseau programmable, et peuvent être appliquées à différents niveaux d'abstraction (RTL, portes, transistors, etc.). Ces techniques ne remplacent pas la qualification finale sous faisceaux, mais permettent d'avoir une première évaluation du niveau de robustesse avant toute fabrication. Dans notre cas, la méthode considérée a été choisie pour être cohérente avec l'ensemble des développements réalisés ; elle s'applique donc sur des descriptions de niveau RTL. Par

ailleurs, cette technique utilise l'émulation sur FPGA pour réduire les durées d'expérimentation par rapport à des simulations. Il faut noter que ces expérimentations viennent en complément des validations faites par simulation, en perturbant le système grâce à un forçage des signaux. De telles validations permettent de vérifier la bonne activation des dispositifs de détection implantés ; elles ne permettent pas en revanche d'évaluer la réaction globale du système complet sans induire des temps d'expérimentation inacceptables. Ainsi, les techniques de détection matérielle présentées dans les Chapitres 2 et 3 on été validées par injection en simulation RTL avec Modelsim. Par ailleurs, la fonctionnalité du recouvrement au niveau OS en utilisant le changement de contexte a été validée sur le prototype en déclenchant le recouvrement grâce à une injection externe par bouton poussoir. Les expériences qui vont être présentées dans la suite ont pour but de valider la globalité des protections au niveau matériel et logiciel.

Le premier objectif pour compléter les validations individuelles des différents mécanismes de base de valider les capacités de recouvrement d’une erreur détectée par le matériel. Observer par simulation le comportement d'un système d’exploitation exécuté par un processeur, même décrit au niveau comportemental, est beaucoup trop lourd et difficile. On a donc décidé d’exploiter des injections ponctuelles pour valider cette partie.

Pour injecter des fautes, on peut travailler directement sur le niveau de description RTL. A ce niveau d’abstraction, une faute correspond à une inversion de valeur dans un signal : on aura donc un SET si c’est un signal combinatoire et un SEU si c’est un signal mémorisé (registre). Si on a une porte « Ou exclusif » (XOR) avec deux entrées C (contrôle) et S_E (signal entrant) on a la table de vérité suivante pour sa sortie S_S (signal sortant):

- si C=0 → $S_S = S_E$
- si C=1 → $S_S = \text{not } S_E$

Cela permet donc d’injecter une faute sur le signal S en commandant le signal C, tout en évitant de perturber le système quand C est désactivé. Il est possible d’insérer des portes XOR dans la description RTL, commandées par un signal d’injection extérieur, comme on peut voir dans la Figure 4-14.

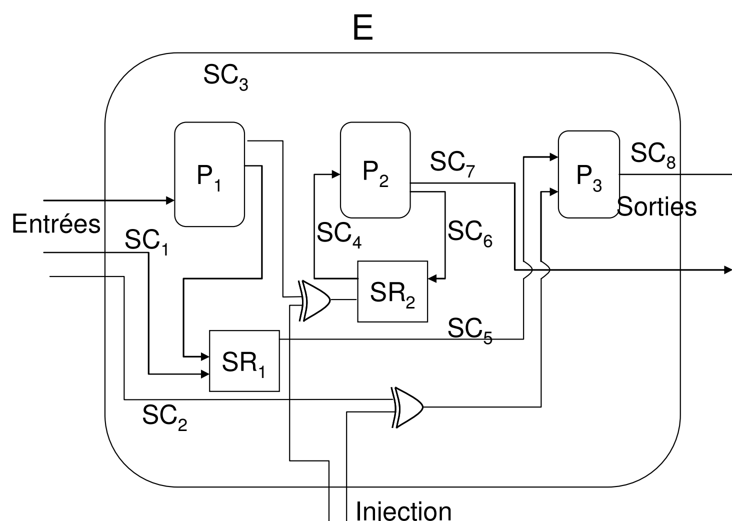


Figure 4-14 Exemple d'injection de fautes à l'aide de portes XOR

Cette technique a l’avantage d’être très précise dans le choix des cibles, ainsi que très mécanique. Le flot d’insertion d’un point d’injection est le suivant :

- identification du signal S sur lequel injecter la faute ;

- insertion d'un XOR sur le chemin du signal ;
- connexion sur la deuxième entrée du XOR du signal d'injection SI_i ;
- routage de SI_i à travers la hiérarchie jusqu'à l'entité de niveau le plus élevé.

La procédure est fortement automatisable, mais aucun parseur VHDL assez évolué pour gérer la complexité du Leon2 n'était disponible. Donc l'insertion des portes XOR a été réalisée en ajoutant manuellement un paramètre de configuration à l'IP. En plus, le nombre d'entrées SI nécessaire pour contrôler les injections croît de façon linéaire avec le nombre de cibles et devient rapidement ingérable au niveau du prototype : les entrées/sorties d'une carte de prototypage sont toujours très limitées. Par exemple, la carte Multimedia a seulement 4 interrupteurs directement utilisables : la plupart des plot d'entrées/sorties du FPGA sont exploités par les 5 bancs de mémoire.

Cette technique est cependant bien adaptée pour obtenir des injections ponctuelles sur un nombre limité de cibles avec un temps de mise en place très limité : elle est donc une bonne candidate pour les premières validations du prototype complet. On a choisi comme cibles des fautes qui ne sont pas corrigeables directement par les protections matérielles implantées, à savoir des SEUs dans les registres internes du pipeline. Plus précisément, on a injecté des fautes sur les cinq signaux suivants :

- Compteur ordinal dans l'étage Fetch ;
- Compteur ordinal dans l'étage Decode ;
- Signal d'erreur d'alignement mémoire dans l'étage Memory ;
- Opération arithmétique à effectuer dans l'étage Execute ;
- Résultat de l'instruction dans l'étage Write.

Le choix n'a pas été aléatoire : les trois premiers cas représentent des erreurs potentiellement critiques qui, si elles ne sont pas traitées, vont conduire soit à l'arrêt du système, soit à sa perte de contrôle. Les deux derniers cas sont potentiellement des erreurs de corruption de données, qui risquent plus probablement de fausser le résultat plutôt que d'arrêter le système.

La décompression d'une image JPEG a été utilisée comme workload, pour avoir des résultats comparables aux analyses préliminaires du paragraphe 3.2.3. Pour que la technique de recouvrement basée sur un changement de contexte puisse fonctionner, il faut qu'il y ait au moins une deuxième application qui tourne en parallèle et qui demande régulièrement du temps d'exécution : on a donc choisi de lancer un thread avec la version logicielle de l'AES. La complexité de cet algorithme garantit qu'il ne va pas se terminer avec le JPEG et qu'il va demander en continu des changements de contexte.

Dans l'implantation sur la carte, les 5 signaux de commande d'injection sont liés à un buffer circulaire commandé par un interrupteur, ce qui permet d'injecter une faute à la fois sur toutes les cibles tout en réduisant les entrées nécessaires. Il faut aussi rappeler que l'objectif principal de cette expérience n'est pas d'établir des statistiques de couverture, mais plutôt de valider le fonctionnement de la technique de recouvrement collaborative OS-Matériel. Ces premiers résultats permettront de mieux définir les expériences ultérieures.

Le setup, basé sur l'implantation de la Figure 4-3, est donc le suivant : la carte Multimedia est chargée avec une version du Leon2 (protégé ou pas), ensuite le programme de test est chargé grâce à la connexion série de l'unité de debug. Une fois l'exécution du programme lancée, on peut suivre son déroulement grâce aux sorties de debug sur le terminal. La fin du programme JPEG est annoncée par l'image « Lena » qui s'affiche à l'écran, comme dans la Figure 3-7.

Une série d'injections sur le Leon2 non protégé a montré différents cas, listés en ordre d'apparition décroissante :

1. Erreur non récupérable : arrêt de l'exécution
2. Jpeg en boucle infinie
3. Pas de conséquence
4. Blocage total du processeur

Le nombre limité de cibles et d'injections ne nous permet pas d'obtenir des pourcentages statistiquement significatifs, mais on peut quand même déjà observer des tendances.

Le plus souvent, la faute a amené le processeur vers un état erroné, qui interrompt l'exécution du programme. C'est le comportement que l'on attendait à la suite de l'injection des fautes critiques dans 3 étages.

Juste après, on observe le cas 2 : le programme ne se bloque pas mais il ne se termine pas non plus et continue à boucler sans donner de résultat à l'écran. La cause la plus probable vient de la corruption des variables liées à une ou plusieurs boucles, ce qui ne permet pas au programme de se terminer.

Dans le cas 3 le programme se termine correctement, comme si aucune faute n'avait été injectée. Cela vient probablement d'une injection sur un signal qui n'a pas été exploité pour un calcul et donc la faute ne s'est pas propagée. Il faut aussi souligner que dans la configuration utilisée il n'est pas possible de savoir exactement dans quelle tâche on injecte, sauf quand il y a une erreur grâce aux traces de debug. Le cas 3 peut donc aussi correspondre à des injections faites lors de l'exécution de la tâche AES.

Enfin il y a un petit nombre de cas où suite à l'injection des fautes le processeur Leon2 s'est complètement bloqué, sans permettre aucune connexion avec l'extérieur même après des réinitialisations. C'est le pire cas qu'on puisse attendre, parce que le système est irrécupérable sans un arrêt total.

Si on exécute le même nombre d'injections sur le processeur protégé en détection et avec le recouvrement OS la tendance est différente :

1. Pas de conséquence
2. Jpeg en boucle infinie
3. Erreur non récupérable : arrêt de l'exécution

La première remarque est que le cas de blocage total disparaît complètement, ce qui permet donc d'avoir toujours le système opérationnel.

Ensuite on peut remarquer comment la tendance change : dans la plupart des cas, le programme s'exécute correctement, avec en plus l'image affichée correctement sans le décalage observé lors des analyse du paragraphe 3.2.3 et illustré dans la Figure 3-6. Cela est probablement dû au fait que le temps Δt_i passé entre le dernier changement de contexte et la détection de la faute est bien inférieure au temps entre deux sauts simulés dans le paragraphe 3.2.3. Il faut aussi souligner aussi que grâce aux informations de trace ajoutées par le C&R il est possible de savoir exactement dans quelle tâche la faute a été injectée et donc les injections dans la tâche AES peuvent être éliminées de l'analyse.

Le cas d'une boucle infinie arrive en deuxième position, l'arrêt de l'exécution étant l'évènement le plus rare.

En conclusion, le C&R basé sur le changement de contexte apporte bien une amélioration de sûreté. Pour pouvoir quantifier cette amélioration et mesurer la couverture de fautes, il est nécessaire de réaliser un grand nombre d' injections automatisées. Il sera aussi intéressant de développer une stratégie de synchronisation pour pouvoir savoir quelle tâche est exécutée au moment de l’injection et donc obtenir des mesures plus précises ; cela devrait être possible en observant le compteur ordinal. Une approche pouvant être employée et les expériences à réaliser sont présentées dans les sections suivantes. Ces expériences pourront être menées sur la base du prototype existant, validé dans le cadre de la préparation de cette thèse. Elles n'ont pu être menées à bien immédiatement à cause de problèmes liés au développement de la plateforme d'injection utilisée, en cours dans le cadre d'un autre travail de thèse.

4.7 Performances et couverture de fautes : analyse dynamique automatisée

4.7.1 Environnement d’injection de fautes

Le développement des circuits programmables permet aujourd’hui d’obtenir des prototypes très tôt dans le cycle de conception et à très bas coûts. En exploitant leurs potentialités il est possible d’obtenir des outils d’injection de fautes très puissants et avec des prix très réduits.

Une approche possible pour les injections se base sur les propriétés de configuration des FPGA SRAM : le comportement d’un tel composant est spécifié par un fichier, appelé généralement « bitstream », qui est généré par les outils de placement/routage. En modifiant ce fichier, il est possible d’insérer des fautes dans le circuit. En plus, la plupart des FPGA modernes permettent de relire et de modifier « à la volée » une configuration, ce qui permet d’injecter des fautes transitoires, de décider du moment d’injection de la faute et d’observer l’évolution du circuit après un intervalle de temps donné. Cette approche est particulièrement bien adaptée pour injecter des fautes de configuration, ce qui sort du cadre de cette thèse. Elle peut cependant aussi être employée pour modifier la valeur de signaux ou le contenu de bascules correspondant aux données traitées par le circuit [Anton03]. Un outil de ce type est en cours de développement au sein du groupe QLF : la carte Multimedia (voir paragraphe 4.1.1) a été choisie aussi parce que son FPGA, le Virtex 2, est compatible avec cet outil. Malheureusement il n’a pas encore atteint une maturité suffisante pour être employé sur un exemple aussi complexe que le système cible de cette thèse.

Une autre approche possible exploite la disponibilité d'une description comportementale synthétisable du système cible, que l’on peut donc modifier pour permettre des injections de fautes. Le groupe QLF a une grande expérience dans la modification des descriptions de haut niveau pour l’injection de fautes. Elle est démontrée par exemple dans le travail de thèse de Karim Hadjiat [Hadij05]. Ces techniques sont très adaptées aux approches à base de simulation, mais s’avèrent souvent très lourdes pour des implantations en FPGA. Dans le cas d'une émulation, l'instrumentation du circuit pour permettre les injections peut être réalisée au niveau portes logiques, après synthèse. Cette approche offre alors la même souplesse que les approches de haut niveau mais permet de simplifier les modifications à réaliser et d'être indépendant du langage de description de matériel employé pour spécifier le circuit. La thèse de Pierre Vanhauwaert, en cours dans le groupe QLF, vise à développer une approche de ce type et celle-ci est désormais assez mature pour être appliquée à notre système [Vanha06].

L'environnement d'injection est basé autour du processeur PowerPc 405 (PPC) embarqué dans les FPGA de la famille Virtex 2 Pro de Xilinx. Les PPC embarqués dans la Virtex II-Pro qui équipe la carte Avnet (paragraphe 4.1.1) peuvent être exploités pour appliquer cette technique.

Le circuit sous test (CUT) est connecté à une interface matérielle, elle-même connectée au bus du processeur. L'interface fournit au CUT toutes ses entrées (horloge comprise) et recueille toutes les sorties à analyser, comme on peut le voir sur la Figure 4-15. Le PPC est connecté à un PC hôte externe qui contrôle la campagne d'injection de fautes et recueille les résultats.

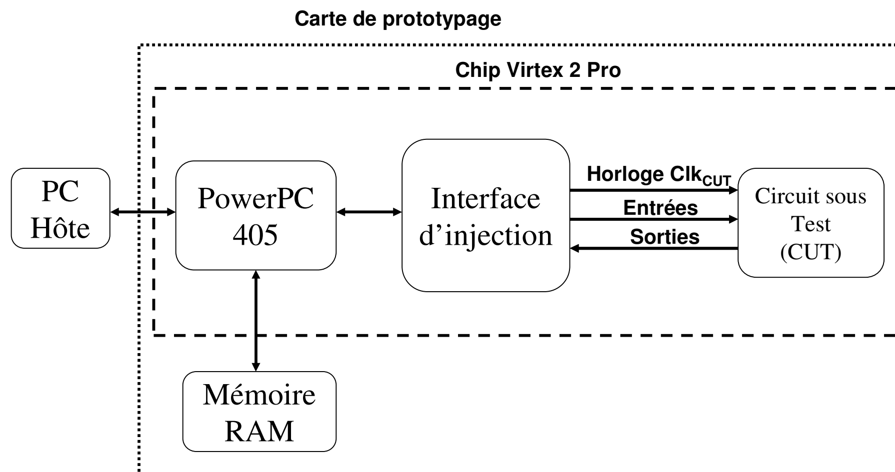


Figure 4-15 Schéma de la plate-forme d'injection de fautes développée par Pierre Vanhauwaert

L'interface est connectée à la même horloge que le processeur, Clk_{PPC}, et une de ses tâches est de générer l'horloge pour le CUT ; Clk_{CUT}. L'algorithme de contrôle de l'interface est résumé dans la partie a) de la Figure 4-16. Ce choix permet de présenter de nouvelles entrées au CUT à chaque front montant de Clk_{CUT}, même si le PPC doit aller les chercher en mémoire et les transférer dans des registres d'entrée, opération qui peut prendre plusieurs cycles de Clk_{PPC}.

Pour effectuer des injections de fautes, un programme spécifique travaille sur la description niveau portes du CUT en format EDIF, qui peut être obtenue à partir de tous les principaux outils de synthèse. Ce programme analyse le fichier et compile la liste de tous les registres internes à partir de laquelle on peut choisir les cibles d'injection. Il modifie ensuite la netlist EDIF pour permettre l'injection de fautes dans les cibles sélectionnées, grâce à des signaux de commande correspondant à de nouvelles entrées primaires du CUT. La netlist instrumentée est implantée dans la logique programmable et les entrées de commande d'injection sont gérées par l'interface. L'approche permet d'injecter des bits erronés dans tous les registres sélectionnés comme cibles, à n'importe quel cycle pendant l'exécution d'une application. L'erreur injectée à un cycle donnée peut ne modifier qu'un bit (modèle SEU) ou un nombre quelconque de bits (modèle MBF) dont la localisation peut être contrôlée selon les besoins.

Une fois l'environnement configuré et les cibles et les instants d'injection définis, on peut lancer la campagne : une première exécution de l'application est faite suivant l'algorithme a) de la Figure 4-16, ce qui permet d'obtenir des résultats de référence (« golden run »). De nouvelles exécutions sont faites ensuite et, pour chaque exécution, les injections sont effectuées quand on atteint les instants choisis, comme indiqué dans la partie b) de la Figure 4-16.

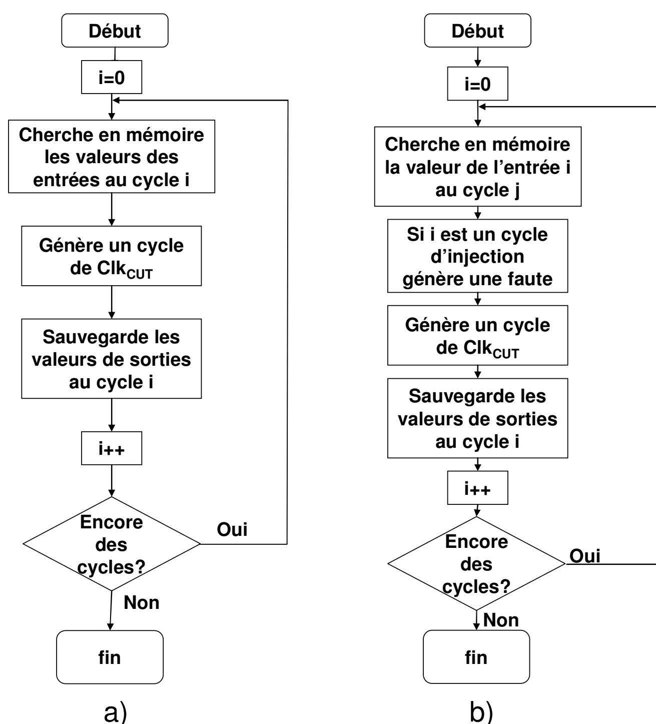


Figure 4-16 Algorithme de l'interface PPC-CUT en modalité prototypage (a) et injection de fautes (b)

4.7.2 Implantation de Leon2 dans l'environnement

L'environnement décrit dans le paragraphe précédent a été développé avec comme circuits ciblés des IPs classiques, qui à chaque cycle d'horloge prennent des signaux en entrée (fournis par l'interface ou stockés en mémoire ROM émulée) et donnent des signaux de sortie. Cependant, Leon2 est un processeur qui prend ses entrées à partir d'une mémoire RAM : ses seules entrées sont les données lues de la mémoire (data_in), dont la valeur dépend de l'adresse demandée et des signaux de contrôle (sélection, lecture/écriture, etc.). Une fois inséré dans l'environnement d'injection, Leon2 ne va pas pouvoir accéder à une mémoire indépendante : les cartes FPGAs disponibles sur le marché ont typiquement toute la mémoire connectée à un seul bus pour économiser les connexions. Si le PPC l'utilise pour l'exécution de son propre programme, Leon2 ne pourra pas l'exploiter. Sans compter que connecter le processeur à une mémoire externe enlèverait à l'outil d'injection la capacité de contrôler ses entrées/sorties. La solution est d'implanter la mémoire du Leon2 dans la mémoire du PPC : l'adaptation de l'outil du paragraphe 4.7.1 est très rapide parce qu'il suffit d'ajouter un module logiciel, sans nécessité de modifier les parties matérielles. A chaque cycle (voir Figure 4-16) l'interface lit les signaux de contrôle, les données sortantes et l'adresse demandée par Leon2. Le PowerPC récupère alors ces données et agit en conséquence, soit en allant chercher les données à lire pour les présenter en entrée du Leon2, soit en mettant à jour l'image de la RAM.

Une autre conséquence de la gestion de l'horloge par l'interface d'injection est la perte de la notion de temps réel, et donc l'impossibilité de s'interfacer avec l'extérieur : l'interface vidéo n'est pas directement utilisable, mais elle aussi peut être facilement simulée en RAM. Suivant le même principe, on peut résoudre l'autre problème lié à l'absence de temps réel : le chargement en mémoire du programme à exécuter. Normalement, cela se fait de façon très simple avec l'unité de debug intégrée, mais cela devient inutilisable dans l'environnement

d'injection. Il faut donc l'implanter lui aussi en mémoire, en faisant attention que les exécutables obtenus avec gcc ne peuvent plus être directement exploités mais doivent être traités au préalable pour résoudre les entrées relogeables du format ELF et obtenir un fichier binaire statique [ELF01].

L'outil d'injection est capable de modifier la valeur des registres, ce qui permet d'injecter des SEUs et des MBFs, mais n'est pas prévu pour injecter des SETs dans les parties combinatoires. Dans le cas des protections matérielles implantées dans le système (paragraphes 2.3 et 3.3), la logique séquentielle est protégée par codage de parité : son comportement vis à vis des fautes est complètement connu par la théorie en se basant sur le modèle de fautes (simples ou multiples). Faire des campagnes d'injection de SEUs n'est donc pas très utile pour évaluer la robustesse au sens de la couverture de fautes, mais peut permettre de valider les dispositifs implantés à différents niveaux pour réaliser des recouvrements d'erreurs. De telles campagnes peuvent aussi donner des informations sur les surcoûts dynamiques en cas de détection et de recouvrement. Afin d'évaluer la couverture de l'ensemble des dispositifs implantés, on a également modifié le Leon2 de façon à pouvoir simuler l'injection de SETs avec un outil capable de générer des SEUs. Selon la stratégie de la Figure 4-17, chaque registre interne R_i du pipeline est dupliqué pour les expériences d'injection et les sorties de la réplique R'_i sont utilisées comme entrées pour le bloc de prédiction de parité L_p. Si on injecte un SEU dans le signal S sauvegardé dans R_i, L et L_p vont avoir des entrées différentes, ce qui est équivalent à un SET sur le signal S. Bien sûr, ceci ne représente qu'imparfaitement les SET possibles dans le système. Toutefois, cela permet un premier niveau d'injection de telles fautes.

Il faut aussi remarquer que le premier étage du pipeline (fetch) n'a pas de registre d'entrée et que la technique proposée ne peut donc pas s'appliquer directement. Une solution possible est d'ajouter un couple de registres R_0 et R'_0 devant le premier étage : on aura une pénalité d'un cycle pour l'exécution de chaque instruction pendant la campagne d'injection, mais il sera alors possible d'évaluer aussi la couverture sur le premier étage.

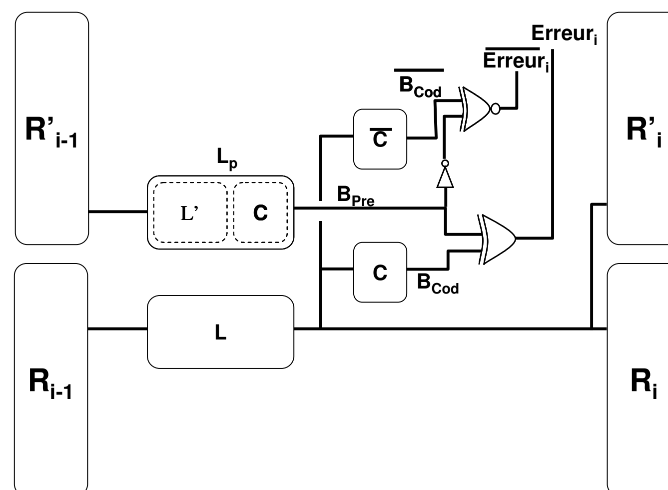


Figure 4-17 Duplication des registres de l'unité entière pour simuler l'injection des SETs

Cette technique va notamment nous permettre une première évaluation des caractéristiques de couverture des fautes de la prédiction de parité, qu'on ne peut pas obtenir analytiquement à partir de la description de la micro-architecture (fin du paragraphe 4.2.2). Du point de vue de la propagation d'une faute dans le circuit, cette technique nous place dans un cas devant conduire à une relativement forte proportion d'erreurs multiples en sorties,

parce que la faute est injectée juste à l'entrée de L ou de L_p. L'effet singulier a donc une plus grande probabilité de perturber le calcul d'un grand nombre de sorties que s'il survient dans les éléments logiques proches de ces sorties.

Il sera aussi intéressant de vérifier le comportement de la détection dans les trois stratégies de filtrage (paragraphe 3.2), en comparant les capacités de couverture respectives.

4.7.3 Spécification des campagnes d'injection de fautes

Une fois la mise en place de l'environnement d'injection terminée, il faut spécifier la campagne (ou les campagnes) d'injection de fautes à mener. Pour cela il faut déterminer :

- le modèle de fautes à injecter ;
- les grandeurs à mesurer ;
- les configurations à tester ;
- les cibles des injections pour chaque configuration.

En ce qui concerne le **modèle de fautes** on va bien sûr se concentrer sur les **effets singuliers**, vu leur importance mise en évidence dans le paragraphe 1.4.2. Dans la campagne d'injection on cherchera à étudier l'effet de **SET**s et aussi de **MBF**s (avec le cas particulier des SEUs), qui peuvent être générés à partir des effets singuliers ou modéliser des attaques ciblées par exemple par laser (paragraphe 1.5.2).

La grandeur la plus importante à mesurer est la **couverture des fautes** : pour comprendre l'efficacité des protections intégrées et de leurs interactions, il faut corréler les fautes injectées avec la réaction du système. Typiquement, on va cataloguer les fautes selon le schéma de la Figure 4-8. Les autres grandeurs très importantes à mesurer sont la **perte de performances intrinsèque** (paragraphe 4.3.1), les **délais de détection** et, si applicable, le **temps de recouvrement** lors de l'activation des protections. On rappelle que certaines des techniques proposées peuvent modifier le comportement statistique du système (notamment le recouvrement dans le pipeline et dans les caches). La campagne d'injection doit nous permettre de mesurer avec précision ces valeurs (au moins en termes de nombre de cycles d'horloge, lorsque le prototype ne fonctionne pas en temps réel pendant les injections).

Les configurations à tester doivent former un **ensemble représentatif** des protections implantées pour permettre leur évaluation. Le Tableau 4-3 et le Tableau 4-4 rappellent l'ensemble des protections implantées.

Tableau 4-3 Résumé des protections matérielles dans Leon2 et l'IP AES

<table>
<tr><th>Type de protection</th><th colspan="3">Composant Protégé</th><th>Type de logique</th><th>Fautes visées</th></tr>
<tr><td rowspan="8">Détection</td><td colspan="3">Etages du pipeline</td><td rowspan="4">Combinatoire</td><td rowspan="4">SET, MBF</td></tr>
<tr><td>Dynamique</td><td>Statique maximale</td><td>Statique minimale</td></tr>
<tr><td colspan="3">Contrôleurs des caches</td></tr>
<tr><td colspan="3">Redondance temporelle IP AES</td></tr>
<tr><td>Registres d'étages</td><td colspan="2">Banc de registres</td><td rowspan="3">Séquentielle (registres)</td><td rowspan="4">SEU, MBF</td></tr>
<tr><td colspan="3">Registres des contrôleurs</td></tr>
<tr><td colspan="3">Registres de l'IP AES</td></tr>
<tr><td colspan="3">Mémoires cache</td><td>Séquentielle (mémoire)</td></tr>
<tr><td rowspan="3">Recouvrement</td><td colspan="3">Etages du pipeline</td><td rowspan="2">Combinatoire</td><td rowspan="2">SET, MBF</td></tr>
<tr><td>Dynamique</td><td>Statique maximale</td><td>Statique minimale</td></tr>
<tr><td colspan="3">Défaut de cache</td><td>Séquentielle (mémoire)</td><td>SEU, MBF</td></tr>
</table>

Tableau 4-4 Résumé des protections logicielles et OS

Type de protection	Composant Protégé	Fautes visées
Détection	AES - Protection logicielle	MBF en mémoire ou en exécution
Recouvrement	OS - C&R par changement de contexte	Fautes détectées à d'autres niveaux

Le système cible est construit autour d'un processeur, donc dans la configuration il faudra aussi considérer le **programme à exécuter** ou **workload**. Le Tableau 4-5 résume les applications présentées jusque là.

Tableau 4-5 Résumé des applications disponibles

Application	Type d'algorithme	Particularités
Codec JPEG	Traitement d'images	Données issues de l'environnement, Contraintes temps réel, Boucle itérative de traitement
AES	Chiffrement	Puissance de calcul requise, Beaucoup d'accès mémoire, Criticité du résultat
Testbench synthétiques	Evaluation des performances	Opérations standard, résultats mesurables

4.7.3.1 Validation de la prédiction de parité

Dans l'analyse des approches de détection matérielle implantées, on a noté l'impossibilité de calculer a priori le taux de couverture pour la technique de prédiction de parité (paragraphe 4.5). Suivre la propagation d'un SET ou MBF à travers de la logique combinatoire, et donc comprendre si ses effets vont être couverts par la techniques de codage choisie (parité dans notre cas), est une opération très difficile et qui ne peut de toute façon pas être réalisée à partir d'une description comportementale. Le problème des signaux indéfinis (paragraphe 3.3.1) vient se superposer pour rendre le cas encore plus compliqué. Le premier objectif des injections de faute va donc être une mesure de taux de couverture de la prédiction de parité dans les trois stratégies de filtrage proposées (dynamique, statique maximal ou statique minimal), et leur comparaison avec les chiffres du Tableau 4-2, pour l'exemple d'implantation correspondant au prototype. L'unité entière est le composant où la prédiction de parité a été implantée de la façon la plus complète et va donc naturellement être le centre de cette analyse. Ce composant est composé de 5 étages, chacun d'eux ayant été protégé individuellement. La technique proposée dans le paragraphe 4.7.2 permet de simuler l'injection de SET à travers la duplication des registres d'entrée de chaque étage.

Du Tableau 4-2 on peut lire qu'en tout 1376 bits sont exploités pour le calcul de parité, et sont donc autant de cibles d'injection. Une étude exhaustive d'injection de MBFs demanderait donc d'injecter 2^{1376} configurations d'erreur différentes, ce qui n'est pas réalisable. D'autant plus que pour être vraiment exhaustive et tester le processeur dans toutes les conditions, il faudrait que chaque injection soit répétée pour chaque instruction dans le jeu du processeur (voire chaque combinaison d'instructions présentes dans le pipeline dans les cas de dépendances). Il est donc nécessaire de définir un sous-ensemble d'injections pouvant

réellement être effectuées : si d'un côté on peut décider de valider l'effet pour au moins toutes les fautes singulières (1376), pour les MBF il faudra se limiter à un ensemble aléatoire, de même que pour l'état initial des registres au moment de l'injection.

La campagne va s'organiser autour de quatre configurations sur lesquelles seront injectées le même ensemble de fautes :

1) Sans protection ;
2) Détection dans l'unité entière, filtrage dynamique ;
3) Détection dans l'unité entière, filtrage statique maximal ;
4) Détection dans l'unité entière, filtrage statique minimal;

Les fautes vont être cataloguées selon la nomenclature du paragraphe 4.2.2, de façon à pouvoir extraire les taux de couverture respectifs. De la configuration 1, on va pouvoir évaluer les fautes masquées par la redondance intrinsèque du processeur (« positif »). Les configurations 2 et 3 vont permettre d'évaluer entre autres les « faux négatifs » insérés par l'approche statique maximale, et la configuration 4 la perte de couverture effective ou « faux négatifs » dus à la diminution du nombre de signaux pris en compte.

4.7.3.2 Validation du recouvrement matériel

Le recouvrement dans l'unité entière a déjà été validé par des injections en simulation, mais pour un nombre très limité de cibles. La mise en place expérimentale de la première campagne d'injection peut être directement exploitée pour effecteur une validation plus complète du recouvrement. Les configurations à étudier sont :

1) Recouvrement dans l'unité entière, filtrage dynamique ;
2) Recouvrement dans l'unité entière, filtrage statique maximal ;
3) Recouvrement dans l'unité entière, filtrage statique minimal;

En injectant le même ensemble de fautes que pour les configurations de détection, il est possible d'évaluer directement l'efficacité des corrections : les cas « négatifs » en mode détection devraient devenir des cas « positifs » si toutes les fautes détectées sont corrigées.

4.7.3.3 Recouvrement par le système d'exploitation

La mise en place d'une campagne d'injections pour tester le Checkpoint & Rollback basé sur le changement de contexte est plus difficile que les précédentes : la perte du fonctionnement en temps réel implique une modification dans l'application JPEG qui a été développée pour tester les critères d'application. Il faut simuler en mémoire l'acquisition et la visualisation d'une image. Il faut aussi prévoir la mise en place du système d'exploitation eCos pour que le recouvrement puisse s'effectuer dans des conditions optimales : si on veut des changements de contexte, il faut qu'il y ait au moins deux tâches en exécution. Une solution consiste à faire exécuter deux threads JPEG, travaillant sur deux images différentes.

L'environnent d'injection a été testé jusqu'à maintenant avec des circuits beaucoup plus simples et pour le moment les injections peuvent simplement être temporisées en termes de cycles exécutés après le reset. Dans le cadre d'une campagne centrée sur l'OS et le logiciel il serait beaucoup plus utile de pouvoir se synchroniser sur des événements (registres qui prennent une valeur donnée, accès à une certaine adresse mémoire, etc.), mais cela n'est pas possible pour le moment. On va donc se limiter à des injections en aveugle, sans savoir avec précision l'état d'exécution du programme, ce qui modélise, par exemple, l'exposition à un faisceau de particules.

Le système matériel doit pouvoir détecter les fautes pour déclencher le recouvrement du système d'exploitation. Pour la campagne on peut donc exploiter 4 configurations :

1) Détection de SET dans l'unité entière, filtrage dynamique ;
2) Détection de SET dans l'unité entière, filtrage statique maximal ;
3) Détection de SET dans l'unité entière, filtrage statique minimal;
4) Détection de SEU dans l'unité entière et le banc de registres.

L'objectif ici étant de valider le recouvrement par le système d'exploitation et d'évaluer l'impact sur les performances du système, il n'est cependant pas nécessaire de réaliser des injections sur ces 4 configurations. Les trois premières étant déjà étudiées lors de la première campagne proposée, il est suffisant ici de réaliser une campagne sur la base de la dernière configuration, ce qui permet aussi de valider les dispositifs de détection dans les registres inter-étages et le banc de registres. Cette étude permettra de savoir si le C&R niveau OS est une alternative efficace aux techniques de recouvrement de SEU au niveau matériel, en général assez gourmandes en ressources. En comparant les images en RAM à la fin de chaque exécution, on pourra aussi estimer de façon précise la déformation des données en sortie causée par le C&R (voir paragraphe 2.2.2.3).

L'exploitation des trois premières configurations peut avoir un autre but. Si on exploite le même ensemble de fautes que dans les campagnes précédentes, on peut directement comparer les résultats et donc comparer les taux de recouvrement par OS avec les taux de recouvrement au niveau matériel dans le pipeline (paragraphe 4.7.3.2).

Il faut quand même souligner qu'en injectant en aveugle on ne peut pas distinguer les échecs de recouvrement causés par le non-respect des critères de validité du C&R et ceux causés par des injections pendant l'exécution du système d'exploitation. Le choix d'avoir deux applications JPEG en tant que threads permet de ne pas devoir distinguer les fautes injectées pendant l'exécution d'une tâche ou de l'autre, mais les taux de couverture seront pessimistes par rapport à ceux que l'on pourrait attendre en ciblant avec précision l'exécution de la tâche.

Une autre approche peut être envisagée, basée sur une modification de structure ciblée du processeur (option de configuration adaptée à l'injection), qui permettrait d'être plus précis sur les instants d'injection. En revanche, l'approche étant moins générale, il faudrait limiter le nombre de cibles.

4.7.3.4 Autres campagnes

Le nombre de dispositifs implantés dans le prototype permettrait de réaliser un grand nombre d'autres campagnes d'injections. En particulier, on peut citer des campagnes réalisées dans l'IP AES, notamment pour valider l'efficacité de la détection par la technique de redondance temporelle. On peut également citer la possibilité de comparer l'efficacité du durcissement logiciel de l'AES et celle du durcissement matériel. Le temps nécessaire à la mise en œuvre de l'ensemble des campagnes possibles sur le prototype est considérable. Nous nous sommes donc limités, dans le cadre de la thèse, aux campagnes les plus importantes vis-à-vis du cœur du travail réalisé. Des campagnes complémentaires exploitant le prototype sont planifiées dans les mois qui viennent et font donc partie des perspectives à court terme du travail présenté dans ce document.

4.8 Conclusion

Dans ce chapitre, on a analysé les résultats obtenus en appliquant les techniques identifiées, spécifiées et développées au fil des chapitres précédents.

Les techniques de protection matérielles ont montré une bonne capacité de couverture avec des coûts très limités. Le Checkpoint & Rollback basé sur le changement de contexte modifié est aussi une technique qui s'est avérée très légère, facile à implanter et très prometteuse, surtout en collaboration avec les approches matérielles.

Le développement du prototype a permis de mesurer et de comparer directement les performances des différentes approches sur des bases communes, ce qui n'est pas évident pour des interventions à des niveaux d'abstraction si différents. La mise en place de l'environnement d'injection de fautes permettra aussi une mesure précise des surcoûts dynamiques en tenant compte des interactions entre techniques, ainsi que des capacités de couverture de fautes et des nombreux compromis.

A moyen terme, il serait aussi intéressant d'exploiter l'environnement d'injection de fautes pour comparer les caractéristiques de sécurité de l'algorithme AES dans ses différentes implantations, même si ce n'est pas le sujet central de cette thèse.

5 Conclusions

Ce manuscrit résume le travail réalisé au cours des trois années de préparation du Doctorat. L'analyse de l'état de l'art a conduit à définir une stratégie de protection basée sur des interventions à la fois aux niveaux matériel, système d'exploitation et logiciel. Le choix d'utiliser des approches à haut niveau a permis de proposer des techniques très générales qui font de la configurabilité leur point clé. L'attention envers les relations entre les différents niveaux a notamment conduit à définir une technique où système d'exploitation et matériel collaborent étroitement pour combler les lacunes respectives des interventions possibles à chacun de ces niveaux et obtenir des résultats impossibles si seul le matériel ou le logiciel est pris en compte. La définition et l'implantation d'un prototype représentatif a permis d'affiner l'implantation de techniques théoriques sur un système réel, enrichissant les résultats avec des données difficiles à estimer dans un cadre purement théorique.

L'analyse des résultats obtenus montre que les techniques employées permettent d'obtenir un système très souple, paramétrable à souhait pour obtenir le compromis visé entre performances, taille et robustesse, en fonction des contraintes de l'application réelle. La mise en oeuvre d'un premier environnement d'injection de fautes permet de montrer comment la collaboration entre niveaux permet d'augmenter la robustesse globale du système avec des coûts très limités. Une telle approche permet aussi d'analyser l'effet des protections sur les performances du système dans son ensemble, et pas seulement sur chaque élément extrait de son contexte.

La spécification et la mise en œuvre d'un environnement d'injection automatisé ont ensuite été réalisées, en collaboration avec le travail de thèse de Pierre Vanhauwaert. Cela permettra dans le futur proche de réaliser des campagnes très approfondies, dont la spécification a également été présentée. Ces campagnes d'injection intensives permettront, entre autres, d'analyser en détail les interactions entre les différentes interventions, leur niveau de collaboration et, potentiellement, leurs incompatibilités.

Sur la base des résultats obtenus dans les campagnes d'injection de fautes, différentes perspectives pourront être dégagées. Dans un avenir proche, on prévoit notamment d'implanter d'autres types de codages pour pouvoir comparer les résultats en termes de taille, performances et couvertures avec ceux obtenus avec la parité. La description comportementale du système est déjà prête à les recevoir une fois que les unités de codage/décodage seront spécifiées : les premiers candidats vont être le code de Hamming pour la logique séquentielle et le code résidu pour les blocs combinatoires de calcul arithmétique. L'approche proposée au niveau système d'exploitation pourra aussi bénéficier de mises au point grâce aux résultats d'injection et aux expérimentations qui vont être facilitées par l'existence du prototype complet.

Dans le domaine du chiffrement, il sera prochainement possible de comparer différentes approches en termes de couverture de fautes et de sécurité. En particulier, un bloc IP pour l'algorithme asymétrique RSA est développé au sein de l'équipe et pourra être inséré dans le système de la même façon que l'AES. Des expériences de tir au laser sont en train d'être organisées en collaboration avec le laboratoire IXL de Bordeaux, et il sera possible à moyen terme d'évaluer la robustesse du prototype développé lorsqu'il est soumis à ce type de stress.

La recherche d'un compromis idéal est aussi le moteur des techniques d'exploration d'architecture qui cherchent à obtenir de façon automatique (ou guidée) la meilleure architecture pour un système dont on a seulement des spécifications algorithmiques. Des

approches comme celles qui ont été développées ici permettraient de prendre en compte aussi la sûreté et sécurité dans les paramètres d'exploration. Une extension des travaux réalisés est donc d'augmenter encore le niveau d'abstraction, en travaillant sur des spécifications de niveau "système", écrites dans des langages comme SystemC.

A plus long terme, ce travail de Doctorat peut conduire à des recherches dans un domaine plus large. Les systèmes multiprocesseurs sont désormais de plus en plus répandus, et les systèmes intégrés incluent de plus en plus souvent des modes de communication s'apparentant aux protocoles utilisés dans le domaine des réseaux (Réseaux sur Puce - Network on Chip, NoC), très prometteurs sous beaucoup d'aspects. Les techniques étudiées pourraient bien s'adapter dans ces contextes, grâce à leurs caractéristiques de souplesse et de configurabilité. Encore plus intéressant, la philosophie qui a guidé ce travail sur les systèmes monoprocesseurs pourrait être à la base du développement de techniques innovantes pour ces nouveaux paradigmes d'architecture.

6 Bibliographie

[Alexa02] Alexandrescu D., Anghel L. et Nicolaidis M., «*New methods for evaluating the impact of single event transients in VDSM ICs*», The IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Vancouver, Canada, November 6-8, 2002, IEEE Computer Society Press, Los Alamitos, California, 2002, pp. 99-107

[Aloma01] Al-Omari R., Somani A.K., Manimaran G., «*A New Fault-Tolerant Technique for Improving Schedulability in Multiprocessor Real-time Systems* », Internal Proceedings International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, USA, Avril 2001

[AMBA99] Spécifications du standard AMBA 2, disponibles gratuitement sur http://www.arm.com/products/solutions/AMBAHomePage.html

[Anghe00] Anghel L., *« Les Limites Technologiques du Silicium et Tolérance aux Fautes »*, Thèse de Doctorat, Laboratoire TIMA, INPG Grenoble, 15 Décembre 2000

[Anghe05] Anghel L., Leveugle R. et Vanhauwaert P, «*Evaluation of SET and SEU Effects at Multiple Abstraction Levels* », 11th International On-Line Testing Symposium (IOLTS05), Saint-Raphael, France, 6-8 Juillet 2005

[Anton03] Antoni L., Leveugle R. et Fehér B., «*Using run-time reconfiguration for fault injection applications* », IEEE Transactions on Instrumentation and Measurement, vol. 52, no. 5, October 2003, pp. 1468-1473

[Atmel06] AT697E Datasheet, disponible à partir du site Atmel, http://www.atmel. com/, dernière mise à jour : Fevrier 2006

[Berna06] Bernardi P., Veiras Bolzani L. M., Rebaudengo M., Sonza Reorda M., Vargas F. L. et Violante M., *«A New Hybrid Fault Detection Technique for Systems-on-a-Chip»*, IEEE Transactions on Computers, Vol. 55 No. 2, February 2006.

[Binde75] Binder D., Smith E.C., Holman A.B., «*Satellite anomalies from galactic cosmic rays* », IEEE Trans. on Nuclear Science, vol. NS-22, no. 6, pp. 2675-2680, Décembre 1975.

[Burns91] Burns A. "*Scheduling hard real-time systems: a review*" Software Engineering Journal Volume 6, Issue 3, pp. 116-128, 1991

[Chen84] Chen C. L. et Hsiao M. Y., *« Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review»*, IBM J Res Development, pp. 124-132, March 1984

[Cheyn00] Cheynet P., Nicolescu B., Velazco R., Rebaudengo M., Sonza Reorda M. et Violante M., «*Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors»*, IEEE Transactions on Nuclear Science Vol. 47 No. 6, Decembre 2000

[Denni76] Denning P**.,** «*Fault-tolerant operating systems*», ACM Computing Surveys Volume 8 Issue 4 pp 359-389, Dec. 1976.

[Devin04] Devine C., *« FIPS-197 compliant AES implementation »*, disponible librement sur Internet sous licence GNU GPL, 2004

[Ecos06] Homepage de eCos , http://sources.redhat. com/ecos/, mise à jour de façon régulière

[ELF01] Executable and Linkable Format (ELF). Tools Interface Standard (TIS). Portable Formats Specification, Ver 1.1. http://www.skyfree.org/linux/references/ references.html

[Elnoz02] Elnozahy M., Alvisi L., Wang Y., Johnson D. B., *«A Survey of Rollback-Recovery Protocols in Message-Passing Systems»*, ACM Computing Surveys (CSUR) Volume 34 , Issue 3, pp: 375 – 408, Septembre 2002

[Gaisl02] Gaisler J. , *"A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture"*, Proceedings of the 2002 International Conference on Dependable Systems and Networks(DSN 2002) pp 409-415, 2002

[Gaisl06] Gaisler Research Homepage, www.gaisler.com, mise à jour de façon régulière

[Gwenn94] Gwennap L., *"Comparing RISC Microprocessors"*, Proceeding of the Microprocessors Forum, October 1994

[Hadij05] Hadjiat K., *« Evaluation Prédictive de la sûreté de fonctionnement d'un circuit intégré numérique »*, Thèse de Doctorat, Laboratoire TIMA, INPG Grenoble, 10 Juin 2005

[Hwang03] Hwang D., Schaumont P., Fan Y., Hodjat A., Lai B., Sakiyama K., Yang S., Verbauwhede I., *« Design Flow for HW/SW Accelleration Transparency in the Thumbpod Secure Embedded System »*, 40th Design Automation Conference (DAC 2003), pp 60-65, Anahiem, California, 2-6 June 2003

[Hager03] Hager C.T., Midkiff S.F., *« An analysis of Bluetooth security vulnerabilities»*, Wireless Communications and Networking, 2003 (WCNC 2003), vol.3 pp1825-1831, 16-20 March 2003

[Henne96] Hennessy J. L. et Patterson D. A., *« Computer Architecture, A Quantitative Approach, 2nd Edition »*, Morgan Kaufmann Publishers Inc., San Francisco California, 1996

[Kanda99] Kandasamy N, Hayes J.P. and Murray B.T *« Scheduling Algorithms for Fault Tolerance in Real-Time Embedded Systems »* Dependable Network Computing, D. Avresky (Ed.), Kluwer Academic Publishers, Boston 1999

[Kewel04] Kewell K ., *"Can Benchmarking Be Rational?"*, Processor Watch, issue #182, 09/20/2004, September 2004

[Ko04] S.-B. Ko and J. –C. Lo, "Efficient Realization of Parity Prediction Functions in FPGAs", Journal of Electronic Testing Theory and Applications (JETTA), Volume 20 Number 5 pp 489-499, October 2004

[Krish86] Krishna C. et Shin K., *« On scheduling tasks with a quick recovery from failure »* IEEE Transactions on Computers, Vol 35, No. 5, pp 448-455, 1986

[JPEG04] Site officiel du «Joint Photographic Experts Group » http://www.jpeg.org/

[Lane98] Tom Lane, Philip Gladstone, Luis Ortiz, Jim Boucher, Lee Crocker, Julian Minguillon, George Phillips, Davide Rossi, and Ge' Weijers, *« The independent*

jpeg group's jpeg software release 6b », disponible librement sur Internet sous licence GNU GPL, 1998

[Lapri88] Laprie J-C., « *Sûreté de Fonctionnement et Tolérance aux Fautes : Concepts de Base* », Rapport LAAS n° 88.287, Novembre 1988

[Lenst96] Lenstra A. K., « Memo on RSA Signature Generation In The Presence of Faults », communication privée (disponible chez l'auteur), 28 Septembre 1996

[Masse98a] Massey J. L., « *Applied Digital Information Theory I* », disponible librement sur http://www.isi.ee.ethz.ch/education/public/free_docs.en.html, Swiss Federal Institute of Technology Zurich, 1998

[Masse98b] Massey J. L., « *Applied Digital Information Theory II* », disponible librement sur http://www.isi.ee.ethz.ch/education/public/free_docs.en.html, Swiss Federal Institute of Technology Zurich, 1998

[May79] May T.C., Woods M.H., « *Alpha-particle-induced soft errors in dynamic memories* » IEEE Trans. on Electron Devices, vol. ED-26, no. 1, pp. 2-9, Janvier 1979.

[Mejia00] Mejia-Alvarez P., Aydin H. , Mossé D. , and Melhem R. "*Scheduling Optional Computation in Fault-Tolerant Real-Time Systems*", Conference on Real-Time Computing Systems and Applications (RTCSA) Korea, p 323, 2000

[Micro05] Embedded Linux/Microcontroller Project, http://www.uclinux.org/

[Moore56]Moore E.F. et Shannon C.E. , «*Reliable Circuits Using Less Reliable Relays*», J. Franklin Institute, vol. 262, pp. 191-208 and 281-297, 1956

[Neuma52] Von Neumann J., « *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components* », Automata Studies, Annals of Mathematics, No.34, pp. 43-98, Princeton, 1956

[Nicol04] Nicolescu B., Savaria Y., Velazco R., « Software Detection Mechanisms Providing Full Coverage Against Single Bit-flip Faults », IEEE Transaction on Nuclear Science, Vol. 51, No. 6, December 2004, pp 3510-3518

[Niels00] Nielsen M. et Chuang I.L, « *Quantum Computation and Quantum Information* », Cambridge University Press, Octobre 2000

[Nist06] National Institute of Standards and Technology homegae, http://www.nist.gov/, mis à jour de façon régulière

[Parul05] Parulkar I. and Cypher R., « T*rends and Trade-Offs in Designing Highly Robust Throughput Computing Oriented Chips and Systems*», 11th IEEE international On-Line Testing Symposium (IOLTS05), pp 74-77, Saint-Raphael, France, 6-8 Julliet 2005

[Pinke95] Pinker S., «*The Language Instinct*», Penguin Books Ltd, 1995

[Pradh96] Pradhan, D., "*Fault-Tolerant Computer Systems Design*". Prentice Hall PTR, 1996.

[Punne01] Punnekkat S, Burns A. and Davis R. "*Analysis of Checkpointing for Real-Time Systems*", The International Journal of Time-Critical Computing Systems Vol 20, pp 83-102, 2001

[Rebau00] Rebaudengo M., Sonza-Reorda M., Violante M., Cheynet P., Nicolescu B., Velazco R., "*Evaluating the effectiveness of a software fault-tolerance technique on RISC- and CISC-based architectures*", 6th IEEE International On-Line Testing workshop, Palma de Mallorca, Spain, July 3-5, pp. 17-21, 2000

[Renau04] Renaudin M., Bouesse F., Proust Ph., Tual J.P., Sourgen L., Germain F., *« High Security Smartcards»*, Design, Automation and Test in Europe (DATE) Conference, Paris, Février 2004

[Rives78] Rivest R. L., Shamir A. et Adleman L., *« A Method for Obtaining Digital Signatures and Public Key Cryptosystems »*, Communications of the ACM, Volume 21 Number 2, Février 1978

[Sang05] Sang-Moon R. et Dong-Jo P., «Checkpointing for the Reliability of Real-Time Systems with On-Line Fault Detection», Embedded and Ubiquitous Computing International Conference (EUC05), Nagasaki Japan, pp 194-202, December 2005

[Siewi82] Siewiorek D; P., Swartw R. S. *«The Theory and Practice of Reliable System Design»*, Digital Press, 1982

[Siewi91] Siewiorek D. P., *"Architecture of Fault-Tolerant Computers: An Historical Perspective"*, Proceedings of the IEEE, Vol. 79 no. 12, December 1991

[Snapg04] SnapGear Embedded Linux Distribution homepage, www.snapgear.com

[Stame06] Stamenković Z., Wolf C., Schoof G.and Gaisler J., *"LEON-2: General Purpose Processor for a Wireless Engine"*, 2006 IEEE Design and Diagnostics of Electronic Circuits and systems Page(s):48 – 51, April 18-21, 2006

[Stins96] Stinson D. « *Cryptographie : Théorie et Pratique* » (traduction Serge Vaudenay) International Thomson Publishing France, 1996

[Tamir90] Tamir Y. et Tremblay M., « *High-performance fault-tolerant VLSI systems using micro rollback »,* IEEE transactions on Computers, vol. 39, no. 4, April 1990, pp. 548-554

[Tande90] Tandem Computers Incorporated, « Guardian 90 OS », 1990

[Tande91] Tandem Computers Incorporated, « Non-Stop Cyclone/R Systems », 1991

[Tanne87] Tannenbaum A. S., «*Operating Systems: Design and Implementation* », Prentice Hall, January 1, 1987

[Touba97] Touba N.A. et McCluskey E.J., *« Logic Synthesis of Multilevel Circuits with Concurrent Error Detection »*, IEEE Transactions on Computer-Aided Design, Vol. 16, No. 7, pp. 783-789, Jul. 1997

[Vanha06] Vanhauwaert P., Leveugle R. et Roche P. , «*A flexible SoPC-based fault injection environment* », 9th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Prague, Czech Republic, April 18-21, 2006, pp. 192-197

[Wallm62] Wallmark J.T., Marcus S.M., « *Minimum size and maximum packaging density of non-redundant semiconductor devices* », *Proceedings IRE*, vol. 50, pp. 286-298, Mars 1962.

[Wikip06] « Wikipedia, l'encyclopédie libre », http://fr.wikipedia.org/wiki/Accueil, mise à jour de façon régulière

[Zhang04] Zhang W., « Replica Victim Caching to Improve Reliability of In-Cache Replication", ACSAC 2004, Lecture Notes In Computer Science 3189, pp 2-15, 2004

7 Publications obtenues pendant la thèse

Revues Internationales

[R.i.1] Portolan M., Leveugle R., « A Highly Flexible Hardened RTL Processor Core Based on LEON » – IEEE Transactions on Nuclear Science, Volume 53, Issue 4, Part 1, Aug. 2006 Page(s):2069 - 2075

Conférences Internationales

[C.i.1] Portolan M., Leveugle R., « Operating systems function Reuse to achieve Low-Cost Fault-Tolerance » – Proceedings of the 10th International On-Line Testing Symposyum (IOLTS04) – 2004

[C.i.2] Portolan M., Leveugle R., « A Context-Switch Based checkpoint And Rollback Scheme » – Proceedings of the XIX Conference on Design of Circuits and Integrated Systems (DCIS 04) – 2004

[C.i.3] Portolan M., Leveugle R., « On The Need for Common Evaluation Methods for Fault Tolerance Costs in Microprocessors » Proceedings of the 11th International On-Line Testing Symposyum (IOLTS05) – 2005

[C.i.4] Portolan M., Leveugle R., « Towards a Secure and Reliable System » – Proceedings of the 2005 IFIP International Conference on Embedded and Ubiquitous Computing (EUC'2005) – 2005

[C.i.5] Portolan M., Leveugle R., « A Highly Flexible Hardened RTL Processor Core Based on LEON » – Proceedings of the 8th European Conference on Radiation and Its Effects on Components and Systems (RADECS 05) – 2005

Conférences Nationales

[C.n.1] Portolan M., Leveugle R., « Réalisation d'une Tolérance aux Fautes à Bas Coût dans les SoCs en Utilisant le Système d'Exploitation » – Actes des Journées Nationales du Réseau Doctoral de Microélectronique - 2004

8 Glossaire

Mot	Description	Première Occurrence
Activation	Une faute est « activée » quand elle affecte l'état du composant	Page 11
ASIC	Application-Specific Integrated Circuit : un circuit intégré développé pour effectuer une tâche spécifique	Page 4
Auto-contrôlable, circuit	Un circuit est dit « auto-contrôlable » quand il capable de détecter ses propres erreurs	Page 19
Availability	Anglais pour « disponibilité »	Page 12
Bug	Une imperfection dans un logiciel qui peut causer des erreurs erratiques ou systématiques	Page 26
CdC	Acronyme pour Changement de Contexte	Page 43
C&R	Acronyme pour Checkpoint and Rollback	Page 26
Cache Miss	Echec de mémoire cache, généré quand la donnée recherchée n'est pas en mémoire.	Page 8
Checkpoint	Le moment où l'état du système est sauvegardé dans un schéma de C&R	Page 26
Contamination	On dit que un erreur « contamine » un système quand il arrive à affecter la sortie et donc causer une défaillance	Page 11
Contexte, changement de	Opération fondamentale d'un système d'exploitation multitâche qui permet de changer la tâche en cours d'exécution	Page 9
COTS	« Commercial Off-The-Shelf », terme utilisé pour indiquer un composant standard (« sortie des étagères ») qui est adapté pour travailler dans un autre milieu.	Page 18
CUT	Circuit Under Test, ou circuit sous test. Dans les processus de test indique l'entité soumise à analyse.	Page 105
Défaillance	Un système est défaillant quand sa sortie est erronée et donc il ne peut pas effectuer la tâche qu'il lui est confiée.	Page 11
Défaillance sûre, système à	Un système est à défaillance sûre si une défaillance éventuelle ne risque jamais de causer des événements catastrophiques	Page 12
Dependability	Anglais pour « sûreté »	Page 12

DFA	Differential Fault Analysis, ou Analyse Différentielle de Fautes. Technique de cryptanalyse basée sur l'observation d'un système cryptographique soumis à plusieurs fautes.	Page 33
Disponibilité	Un système est disponible s'il est prêt à répondre à une requête à n'import quel moment	Page 12
Dual Rail, codage	Cas particulier de la duplication où le dupliqué est implanté en logique inverse par rapport à l'original. Il offre une meilleure résistance aux fautes réelles par rapport à la duplication simple.	Page 19
EOS	Embedded Operating System, ou Système d'Exploitation Embarqué : un système d'exploitation expressément conçu pour es applications embarquées où ressources et temps sont facteurs clés.	Page 8
Erreur	Une erreur est une faute qui a été activée et a donc modifié l'état du composant	Page 11
Fail-safe, system	Anglais pour « système à défaillance sûre »	Page 12
Failure	Anglais pour « défaillance »	Page 11
Fault	Anglais pour « faute »	Page 11
Faute	Une faute est une valeur erronée dans un composant qui pourrait causer une erreur	Page 11
Fetch	Extraction : indique la phase de lecture à partir de la mémoire d'une instruction à exécuter	Page 5
Fiabilité	Un système est fiable quand il peut assurer une continuité de fonctionnement	Page 12
FIT	Failures In Time : unité de mesure que indique le nombre moyen des fautes attendues dans un circuit pendant un temps de 10^9 heures.	Page 15
Glitch	Oscillation, le plus souvent de tension, courante ou d'alimentation.	Page 14
GPL	General Public Licence : licence développé par la GNU Foundation qui protége les logiciels/IP libres. Les sources d'un élément protégé par cette licence doivent être complètement publiques et librement modifiables. Tous produits contenant des éléments GPL sont soumis à la GPL eux aussi (à différence que pour la LGPL).	Page 122

HAL	Hardware Abstraction Level, couche	Page 39

	d'abstraction su matériel qui permet au système d'exploitation eCos d'être facilement portable entre plateformes différentes : une fois cette couche est adaptée tous les niveaux supérieurs seront immédiatement opératifs	
Harvard, architecture	Un ordinateur suive une architecture de style Harvard si il a données et instructions stockées dans mémoires séparées.	Page 8
Latente, faute	Une faute est dite « latente » quand elle est apparue mais elle n'a pas encore affecté l'état du composant	Page 11
LGPL	Lesser General Public Licence : license développé par la GNU Foundation qui protége les logiciels/IP libres. La différence par rapport à la GPL est qu'elle permet, sous des conditions très précises, de intégrer des éléments LGPL dans des produits commerciaux.	Page 38
Longue vie	Un système est à « longue vie » s'il peut assurer une bonne probabilité d'être opérationnel après un long période de temps	Page 12
MBF	Multiple Bit Fault : un effet singulier qui cause la transition de plusieurs bits (logique et/ou combinatoires) au même temps	Page 14
MBU	Multiple Bit Upset : un effet singulier qui cause la transition de plusieurs point mémoire au même temps	Page 14
MC	Mémoire Centrale : indique la mémoire principale d'un processeur, en générale caractérisée par une grande taille et des longs temps d'accès.	Page 7
MMU	Memory Management Unit : unité qui permette une gestion avancé de la mémoire, avec notamment la pagination, indispensable pour la plupart des OS modernes comme pour exemple Linux	Page 9
MTBF	Mean Time Between Failures : quantité qui mesure l'intervalle de temps moyen entre deux fautes successives	Page 15
Multitâche, système	Un système multitâche permet l'exécution de plusieurs tâches sur le même matériel en parallélisme simulé	Page 9
Ordonnanceur	Composant d'un système d'exploitation multitâche qui décide la tâche qui va être exécutée lors d'un changement de contexte	Page 10
OS	Operating System, système d'exploitation	Page 8

Pipeline	Architecture qui découpe l'exécution d'une tache en plusieurs étages qui sont opéré en parallèle, chacun sur une instruction différente.	Page 6
Processus	« Coquille » de ressources affectées en exclusivité à une tâche pour lui permettre de s'exécuter en autonomie	Page 9
RAM	Random Acces Mémory : une mémoire volatile qui peut être accédée en lecture/écriture à n'importe quel adresse	Page 5
Reliability	Anglais pour « fiabilité »	Page 12
Rollback	Opération de rétablissement d'un état sauvegardé en précédence lors d'un checkpoint pour rétablir l'exécution d'une tâche qui a été affectée par une erreur	Page 26
ROM	Read-Only Memory ou mémoire morte. Une mémoire qui peut être seulement lue mais pas écrite.	Page 5
RTL	Register Transer Level : niveau de description du matériel qui permet de mettre en évidence les échange d'informations entre composants	Page 14
Safety	Anglais pour « sûreté »	Page 12
Scheduler	Anglais pour « ordonnanceur »	Page 10
SEC/DED, codes	Single Error Correction/Double Error Detection : codes qui permettent de corriger une erreur simple et de détecter une erreur double. L'exemple plus typique est un code de Hamming de distance 3 ou 4	Page 23
Sécurité, de fonctionnement	Condition où l'occurrence des défaillances catastrophiques est impossible.	Page 12
Sécurité, des données	Condition où la confidentialité des données est toujours assurée.	Page 12
SEE	Single Event Effect : événement singulier, typiquement causé par un impact de particule.	Page 14
Self-Checking circuit	Anglais pour « Circuit auto-contrôlable »	Page 19
SET	Single Event Transient : une SEE qui cause l'inversion d'un bit dans un calcul combinatoire	Page 14
SEU	Single Event Upset : un SEE qui cause l'inversion de un point mémoire	Page 14
SoC	System on Chip, Système sur Puce	Page 4
Thread	Un thread est un processus « léger », où certaines contraintes de confinement des tâches sont	Page 25

	relâchées pour améliorer les performances	
Timeslice	Temps que une tâche a à sa disposition pour s'exécuter entre deux changements de contexte	Page 9
TMR	Triple Modular Redundancy : technique de masquage d'erreur par triplications des modules	Page 16
TTM	Time To Market ou temps de mise en marché: indique le délai qui passe entre le début d'un projet et la mise sur le marché du produit final.	Page 37
Von Neumann, architecture	Un ordinateur suive une architecture de style Von Neumann si il a une seule mémoire où données et instructions sont stocké ensemble.	Page 8
WB	Acronyme pour Write Back	Page 46
Write-Back	Stratégie de mise à jour des caches données où les modifications sont reportées en MC* seulement au moment de vider la mémoire cache	Page 8
Write-Through	Stratégie de mise à jour des caches où chaque modification est directement reportée dans la MC* aussi	Page 8

1.1.1 RESUME

Cette thèse s'attache à définir une méthodologie globale permettant d'augmenter le niveau de sûreté et de sécurité face à des fautes logiques transitoires (naturelles ou intentionnelles) survenant dans un système intégré matériel/logiciel, par exemple de type carte à puce. Les résultats peuvent être appliqués à tout circuit construit autour d'un cœur de microprocesseur synthétisable et d'un ensemble de périphériques spécialisés. Les méthodes de protection portent simultanément, de manière complémentaire, sur le matériel, le logiciel d'application et les couches d'interface (en particulier, le système d'exploitation). Les modifications sur des descriptions de haut niveau on été privilégiées pour leurs avantages en terme de généralité, configurabilité, portabilité et pérennité.

L'approche proposée vise un bon compromis entre le niveau de robustesse atteint et les coûts induits, aussi bien au niveau matériel qu'au niveau performances. Elle est appliquée et validée sur un système significatif, représentatif d'un système embarqué monoprocesseur, basé sur le processeur 32 bits Leon2 et le système d'exploitation eCoS. Le processeur a été modifié pour augmenter sa capacité à réagir face à des perturbations, en modifiant ses mémoire caches, leurs contrôleurs et l'unité entière pipeline. Le système d'exploitation a également été modifié pour implanter une technique de recouvrement à bas coût basée sur la réutilisation de fonctions préexistantes, notamment les fonctions de changement de contexte, en collaboration avec les techniques de détection implantées au niveau matériel.

Un démonstrateur exécutant des applications représentatives de chiffrement et de traitement d'images a été développé et a servi de cible à des expériences pratiques d'injection de fautes.

1.1.2 MOTS-CLES

Systèmes embarqués, systèmes intégrés, sûreté, sécurité, systèmes d'exploitation, LEON2

1.1.3 TITLE

DEPENDABLE AND SECURE DESIGN OF AN EMBEDDED SYSTEM

1.1.4 ABSTRACT

This PhD researches a global methodology enabling to improve the dependability and security level against transient logic faults (natural or provoked) appearing inside a hardware/software integrated system, like for instance a smart card. Results can be applied to all systems built around a synthesisable microprocessor core and a set of specialised peripherals. The protection methods operate simultaneously and in complementary manner on hardware, application software and interface layers (most noticeably, the operating system). High level modifications have been favoured for their advantages in terms of generality, configurability, portability and perpetuity.

The proposed approach aims at achieving a good trade-off between robustness and overheads, from both hardware and performance point of views. It is applied on a significant system example, representative of an embedded monoprocessor system, based on the 32-bit Leon2 processor and the eCos operating system. The processor has been modified to improve his ability to react against perturbations, by modifying its cache memories, their controllers and the integer unit pipeline. The operating system has been modified as well to implement a recovery technique based on the reutilisation of existing functions, noticeably the context switch function, in collaboration with the detection techniques applied at hardware level.

A demonstrator executing representative cryptographic and image processing applications has been developed and targeted by fault injection experiments.

1.1.5 KEY WORDS

Embedded systems, integrated systems, dependability, security, operating systems, LEON2

1.1.6 INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 978-2-84813-089-9

ISBNE : 9782848130989