Détection d'erreurs transitoires survenant dans des architectures digitales par une approche logicielle : principes et résultats expérimentaux

B. Nicolescu

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|_|

THESE

Pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

préparée au laboratoire TIMA dans le cadre de l'Ecole Doctorale

« Electronique, Electrotechnique, Automatique, Télécommunications, Signal »

présentée et soutenue publiquement

par

Sergiu Bogdan NICOLESCU

le 24 Septembre 2002

Titre

Détection d'erreurs transitoires survenant dans des architectures digitales par une approche logicielle : principes et résultats expérimentaux

Directeur de thèse
Raoul Velazco

JURY

M. Pierre GENTIL	Président
M. Pascal FOUILLAT	Rapporteur
M. Fulvio CORNO	Rapporteur
M. Raoul VELAZCO	Directeur de thèse
M. Gary SWIFT	Examinateur
M. Matteo SONZA REORDA	Examinateur
M. Robert ECOFFET	Examinateur

Pentru Gabriela,
Pentru Familia mea

Credo quia absurdum est - Tertullian

Remerciements :

J'adresse mes remerciements à Monsieur Pierre Gentil, Responsable d'Ecole Doctorale, pour avoir accepté me faire l'honneur de présider le jury de cette thèse.

Je tiens à remercier Messieurs Fulvio Corno, Professeur à l'Institut Polytechnique de Torino, et Pascal Fouillat Professeur à l'Université de Bordeaux d'avoir accepté d'être les rapporteurs de cette thèse et pour l'intérêt qu'ils ont porté à mes travaux et leurs remarques judicieuses.

Je tiens de remercier Messieurs Matteo Sonza Reorda, Professeur à l'Institut Polytechnique de Torino et Garry Swift, Ingénieur au JPL-NASA pour leur participation en tant que examinateurs du jury.

J'exprime également ma reconnaissance à Monsieur Raoul Velazco, Directeur de Recherche au CNRS, pour m'avoir proposé ce sujet, pour l'encadrement de mon travail, ses conseils, ses critiques et ses encouragements, ainsi que pour sa disponibilité, sa bonne humeur et son optimisme.

Je suis très reconnaissant à Monsieur Matteo Sonza Reorda, Monsieur Massimo Violante et Monsieur Maurizio Rebaudengo, pour des discussions et l'échange des idées stimulantes qui ont constituées le point de départ de mes recherches présentées en cette thèse.

Un grand merci à mon adorable femme Gabriela, pour le soutien moral et ses conseils pendant tous ces trois ans de thèse.

Table des Matières

Liste des Figures

Liste des Tableaux

Introduction

La réduction continue de la taille et des paramètres électriques des circuits intégrés, due aux progrès technologiques dans le processus de fabrication microélectronique, a comme conséquence une sensibilité accrue aux effets de l'environnement (radiations, EMC, température). En particulier, les circuits intégrés fonctionnant dans l'espace sont sujets à différents phénomènes comme conséquence des radiations, dont les effets peuvent être permanents ou transitoires. Parmi ces conséquences, doivent être mentionnées :

- Les événements dits singuliers ou SEE (*Single Event Effect*) qui peuvent apparaître suite à l'interaction d'une unique particule chargée avec une zone sensible du circuit intégré considéré.
- Les effets de la dose cumulée résultante de l'accumulation de charges dans les oxydes.

Cette thèse s'intéresse particulièrement au phénomène « d'upset » qui résulte en l'inversion de bits d'un élément mémoire comme conséquence du passage d'une particule énergétique dans des zones sensible des circuits intégrés. Lorsque l'objet de l'étude est un circuit de type « processeur », capable d'exécuter un jeu d'instructions ou de commandes, les conséquences d'un upset peuvent être critiques, pouvant conduire dans des cas extrêmes à la perte du contrôle du système. Ceci entraîne la mise en œuvre de techniques de détection d'erreurs de type upset pour faire face aux effets des radiations.

Cette thèse a pour but d'explorer les possibilités et l'efficacité d'une technique purement logicielle de détection d'upsets qui s'adresse aux architectures numériques à base de processeurs. Une telle méthode fut initialement proposée par un groupe de recherche de l'Institut Polytechnique de Torino. Elle repose sur des transformations du code du programme d'application introduisant des redondances aussi bien au niveau des données que dans le code du programme. Bien que les expériences préliminaires d'injection des fautes ont montré un bonne capacité de détection, cette méthode entraîne une baisse considérable des performances du système qui est ralenti d'un facteur d'environ 5. Malgré cet inconvénient, les possibilités d'automatisation offertes par cette méthode et l'indépendance de ressources matérielles qu'elle entraîne, nous ont conduit au choix de cette approche comme point de départ de recherche effectuée au cours de cette thèse.

Cette thèse après une étude de l'état de l'art, propose un ensemble de règles permettant la transformation d'une application en une autre fonctionnellement équivalente, mais avec des capacités de détection d'erreurs. L'ensemble des nouvelles règles est issu d'une analyse approfondie de celles de la méthode mentionnée, ce dans le but d'améliorer la capacité de détection (réduction de nombre de fautes qui échappent au mécanisme de détection adopté) tout en minimisant le temps d'exécution du programme transformé. Une contribution significative de cette thèse est la validation de la qualité de détection par différentes approches d'injection de fautes et finalement par des essais sous radiation effectuées à l'aide plusieurs installations dédiées au tests d'upsets disponibles au sein d'accélérateurs de particules en Europe et Etats-Unis.

L'ensemble de ces travaux permet de conclure sur la validité et la généralité de l'approche compte tenu qu'elle a été appliquée à divers programmes et architectures incluant des modules d'une application destinée à être utilisée dans un projet satellite.

Cette thèse s'articule autour de quatre chapitres. Le premier chapitre est consacré à la présentation de l'état de l'art sur les méthodes existantes pour la détection d'erreurs de type upset, survenant dans des architectures à base de processeurs. Ce chapitre se termine par une description du contexte et des motivations encadrant notre travail : les effets des radiations sur des architectures numériques à base de processeurs destinées à fonctionner dans un environnement radiatif.

Dans le deuxième chapitre nous présentons l'extension d'une méthodologie existante pour la détection de fautes de type upset. Une première partie de ce chapitre décrit le principe de base de cette méthodologie, qui est basée sur un ensemble de règles permettant la transformation d'un programme cible en un nouveau programme, appelé programme « durci » possédant les mêmes fonctionnalités mais ayant la capacité de détecter des erreurs. Après avoir présenté et discuté cet ensemble de règles, un nouvel ensemble de règles est proposé. La deuxième partie de ce chapitre effectue une analyse sur les performances des programmes résultants suite à l'application de ce nouvel ensemble de règles. Enfin, dans la troisième partie nous présentons un flot de conception automatique des programmes ***durcis*** par l'application de la méthodologie proposée.

Dans le troisième chapitre est décrite l'application de la méthodologie pour l'obtention de programmes durcis, sur un ensemble de programmes destinés à être exécutés sur un microcontrôleur 8- bits : le 80C51 d'Intel. Ensuite seront présentés et discutés les résultats expérimentaux obtenus par deux types d'approche : sessions d'injection de fautes et essais sous radiations.

Le quatrième chapitre analysera l'efficacité de la méthodologie au cœur de cette thèse dans le cas d'une application industrielle destinée à être exécutée par un processeur complexe de traitement de signal numérique, le DSP32C, candidat à un projet satellite. Une technique novatrice d'injection de fautes a été mise en œuvre afin d'évaluer les résultats dans le cas de cette expérimentation.

Finalement, les conclusions feront la synthèse de ces recherches, mettant en évidence les avantages et les inconvénients de la technique proposée pour la de détection d'upsets et discutant son domaine d'application pour faire face aux conséquences des radiations dans différents environnements.

Chapitre 1

Méthodes pour la détection et/ou la correction d'erreurs transitoires dans des architectures digitales

- Approches matérielles
- Approches mixtes logicielles/matérielles
  - Contrôle en ligne du flot d'exécution
  - Le contrôle du flot d'exécution avec des expressions régulières
  - Le Bloc de recouvrement
  - L'approche multi-version basée sur l'utilisation de plusieurs versions du même programme
  - Tolérance aux fautes implantée par logiciel

I. Contexte

Les progrès technologiques réalisés ces dernières années dans les processus de fabrication des composants microélectroniques (taille du canal de transistors en constante réduction, actuellement de l'ordre de 0.09 μm), augmentation de la densité d'intégration (plusieurs dizaines de millions de transistors par puce) et baisse des tensions d'alimentation (1.8 V actuellement), a pour conséquence l'augmentation de la sensibilité à certain effets de l'environnent (ionisation du substrat due aux radiations, perturbation électromagnétique, effets thermiques) qui étaient considérés mineurs ou sans effets dans des technologies du passé.

Les circuits numériques fonctionnant en présence de radiations subissent des effets qui peuvent être permanents et transitoires [1]. Les premiers sont le résultat de l'accumulation de charges dans les oxydes et apparaissent après des longs périodes d'exposition à des radiations, produisant un effet dit de ***dose cumulée***. Ces charges sont dues au piégeage de paires électrons-trous lors des phénomènes d'ionisation. Pour les technologies MOS, il se produit une dérive des paramètres électriques [2] (tension de seuil par exemple). Ces charges sont cumulatives et peuvent conduire à une perte progressive, puis totale des fonctionnalités du composant.

Les effets transitoires, appelées SEE. (***Single Event Effects***), peuvent être provoqués par le passage d'une unique particule chargée à travers de zones sensibles des circuits. Deux types de SEE sont distingués [3] :

▪ Les **upsets** (*Single Event Upset* ou *SEU*) ayant pour conséquence le basculement intempestif du contenu de cellules mémoires. L'ionisation des matériaux, suite au passage d'une particule chargée dans un circuit, peut causer un changement intempestif de l'état d'un bistable. La particule produit des paires électrons-trous le long de sa trajectoire, ce qui fait apparaître une impulsion de courant laquelle, si l'amplitude et la durée sont suffisantes, peut produire le même effet que celui d'un signal extérieur appliqué au transistor. Les conséquences des SEUs sur le comportement d'un circuit intégré dépendent de la nature de l'information contenue dans les cellules perturbées et de l'instant d'occurrence, peuvent dans certaines cas avoir des conséquences critiques (i.e. arrêts du système suite à la perte de séquencement de son processeur) [4-8]. Pour des architectures complexes de processeurs, la sensibilité aux SEUs est fortement liée au nombre de cellules mémoires (registres, mémoire interne, etc.).

▪ Les **latchups** (*Single Event Latchup*) résultent de la mise en conduction d'un thyristor parasite (structure PNPN) présent dans tous les circuits intégrés CMOS bulk. Une impulsion transitoire de courant produite par l'impact d'une particule lourde peut amorcer la mise en conduction d'un tel thyristor parasite, qui en conditions normales est inactif. Le latchup crée un chemin direct entre la masse et l'alimentation et, par conséquent cause un échauffement important du composant, pouvant conduire à sa destruction. Pour désamorcer le thyristor, il est nécessaire de couper l'alimentation du circuit [9]. Dans la majorité des cas, les latchups entraînent une forte augmentation de la consommation du circuit, mais certains composants montrent seulement de faibles changements de consommation de courant. Ce dernier phénomène qui est appelé *micro-latchup*, pourrait être lié à la topologie même du circuit et être causé par des latchups survenant en des points précis du composant. De tels événements ont été observés sur certains processeurs [10] ou co-processeurs [11]. Les études réalisées dans ce domaine montrent qu'il est difficile de définir les causes exactes des micro-latchups et de prévenir ses conséquences.

L'objectif général de cette thèse est d'explorer les possibilités et l'efficacité d'une **technique de détection d'upsets** basée sur des mécanismes **exclusivement logiciels** et destinée à des applications digitales à base de processeurs. La réalisation d'expériences permettant l'obtention de données objectives sur l'efficacité de la technique au cœur de ces recherches a été une étape primordiale. Des expériences d'injection des fautes et des essais sous radiations ont été réalisées sur plusieurs programmes et processeurs. Le principe de base de la méthode de détection d'upsets consiste en la transformation du programme cible selon un ensemble de règles qui introduisent des redondances au niveau des données et au niveau du code. Cette transformation permettra d'obtenir une nouvelle version du programme ayant la même fonctionnalité que le programme initial, mais avec la capacité de détection de SEUs. Dans la suite de ce manuscrit, les programmes modifiés selon les règles étudiées seront, par abus de langage, appelés « programmes durcis ». En effet, bien que l'objectif de cette thèse se limitera à la détection d'erreurs de type upset, il est clair que des actions de correction des informations corrompues peuvent être enclenchées suite à la détection, rendant donc le programme robuste face à la cause provoquant les erreurs.

Une attention particulière a été prêtée aux aspects liés à l'automatisation de la méthodologie proposée. En effet, pour qu'elle soit d'utilité pour des applications scientifiques ou industrielles, elle doit servir comme support pour la génération automatique des programmes durcis à partir du code source du programme original.

II. Etat de l'Art : Méthodes de détection d'erreurs de type upset

Dans ce qui suit on s'intéressera seulement aux architectures digitales construites autour de processeurs. En effet, les processeurs et les mémoires au sein de ces architectures, de par le nombre très important de cellules mémoires qu'ils possèdent, sont parmi les circuits intégrés les plus sensibles face aux effets des upsets. Une analyse de la littérature spécialisée permet d'identifier deux grandes familles d'approches pour la détection et/ou la correction d'erreurs de type upset :

- Solutions *logicielles* : il s'agit de stratégies exclusivement basées sur des modifications des programmes cibles.
- Solutions *matérielles* : stratégies incluant des modifications de l'architecture matérielle cible, que ce soit au niveau de sa structure ou des circuits la constituant.

A. Approches matérielles

Des stratégies issues des domaines de la sûreté de fonctionnement (redondance massive de blocs matériaux, codage de l'information,…) peuvent être utilisées pour détecter/corriger les effets des upsets. Des telles solutions sont bien connues depuis les années 60 et ont fait l'objet de nombreuses recherches pour rendre robuste face aux effets des radiations, des architectures destinées à des applications spatiales.

De nos jours, l'évolution de la technologie permet la réalisation de circuits intégrant en une seule puce des architectures d'une complexité bien au delà de celle des ordinateurs complets d'il y a quelques dizaines d'années. Il est donc naturel de penser à utiliser des stratégies de sûreté de fonctionnement au niveau des circuits, dans le but de réaliser des circuits fiables en utilisant des processus de fabrication standard. Ainsi, dans la référence [12] des codes correcteurs tel les codes de Hamming et le code cyclique (CRC) sont intégrés dans un circuit pour protéger ses cellules mémoires contre les basculements des bits intempestifs. Des solutions basées sur le développement des bibliothèques des cellules durcies ont été largement explorées. A titre d'exemple, dans les références [13-15] sont étudiées des cellules mémoires capables de tolérer les upsets, alors que dans la référence [16] est présentée une famille de portes logiques permettant la conception de circuits combinatoires ou séquentiels robustes face à l'apparition de signaux incorrects intempestifs.

Une étude de synthèse très complète sur les approches matérielles peut être trouvée en [17], dans laquelle l'auteur conclut que ces solutions ne doivent plus être considérées comme viables compte tenu du coût très important de leur implémentation.

En effet, jusqu'à présent, aucune des solutions matérielles proposées dans la littérature spécialisée n'a été considérée comme étant capable d'atteindre un compromis acceptable entre le coût, la performance d'exécution et la fiabilité. La politique actuelle de la presque totalité des agences spatiales s'orientant vers l'utilisation de composants commerciaux (COTS : Circuits Off The Shelf), il est d'une haute importance la réalisation d'une étude approfondie sur les possibilités offertes par les solutions indépendantes du matériel. De telles solutions feront l'objet principal des recherches entreprises dans cette thèse.

B. Approches mixtes logicielles/matérielles

Les approches basées exclusivement sur des transformations logicielles devraient permettre le développement des systèmes tolérants aux fautes sans encourir les coûts élevés venant de la conception de circuits dédiés ou de l'utilisation de redondances matérielles. Des solutions représentatives peuvent être trouvées dans les références [18-20]. L'idée est d'atténuer les effets des SEUs dans les principales zones sensibles (la mémoire où le code du programme et les données sont stockés, les registres et les mémoires internes du processeur au cœur de l'architecture étudiée) par leur détection (et correction dans certains cas) au moyen de mécanismes logiciels appropriés.

B.1. Contrôle en ligne du flot d'exécution

Cette technique est basée sur l'analyse de signature et un mécanisme qui met en application un mécanisme d'analyse de la signature [21-22]. La vérification en ligne du flot d'exécution vise la détection de séquences d'instructions exécutées de manière incorrecte dans un programme [23-24].

Une signature représentant l'exécution correcte du flot d'exécution est calculée et stockée avant l'exécution du programme. La signature est recalculée de manière concurrente avec l'exécution du programme et comparée avec la signature de référence. La comparaison est réalisée à l'aide de mécanismes appropriés d'analyse de signature. Les approches d'analyse de la signature sont groupées en deux classes :

- Surveillance de signature (ESM - Embedded Signature Monitoring) : la signature est inclue dans le programme d'application [25-26];
- Surveillance autonome de signature (ASM - Autonomous Signature Monitoring) : la signature est stockée dans une mémoire consacrée au mécanisme d'analyse signature [27]

Cette méthode implique des modifications aussi bien matérielles qu'au niveau du système pour l'implantation du mécanisme d'analyse de signature. L'augmentation de la taille de la mémoire cache, mène à la diminution de performances de détection. Cela parce que le mécanisme d'analyse de signature ne peut pas accéder à la mémoire cache du processeur.

B.2. Le contrôle du flot d'exécution avec des expressions régulières

Cette approche est basée sur un formalisme de signature basée sur des expressions régulières. Un programme générique peut être représenté par un graphe de contrôle du flot (FCG - *Flow Control Graph*) dans lequel chaque nœud correspond à une instruction ou à un bloc d'instructions, tandis que les arcs représentent le flot d'exécution des instructions et des blocs. Les nœuds d'un graphe de contrôle du flot d'exécution peuvent être groupés en deux catégories :

- *Nœuds Séquentiels* : l'instruction associée ne modifie pas le flot d'exécution du programme ; le flot du programme est donc séquentiel,
- *Nœuds de contrôle* : l'instruction associée peut conduire à la modification du flot d'exécution du programme ; ils sont typiquement associés aux instructions de contrôle telles les branchements, les appels de fonctions ou de procédures etc.

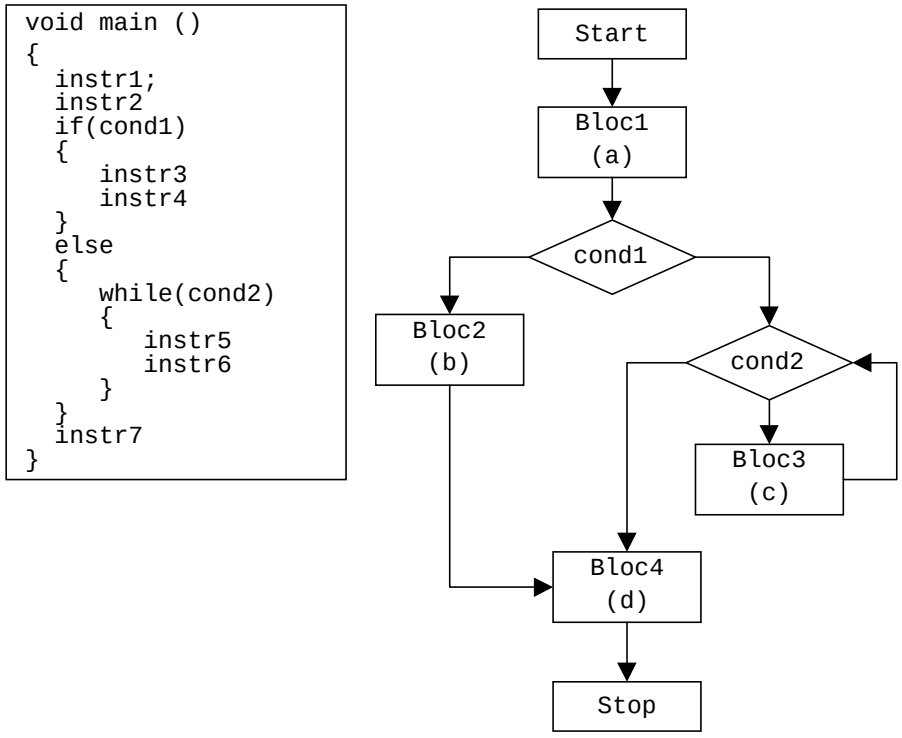


Figure 1.1 : Graphe de contrôle du flot d'un programme

A chaque bloc dans le graphe de contrôle du flot est associé un symbole unique appelé *symbole de bloc* (entre parenthèses dans la Figure 1.1). En utilisant cette notation les chemins corrects dans le graphe d'exécution génèrent un langage qui se compose de toutes les chaînes obtenues par la concaténation des tous les *symboles de bloc* composant le chemin correspondant dans le graphe de contrôle d'exécution [28].

Ce langage peut être représenté par :

$L = (A, R)$ ou :

- A : est l'alphabet d'entrée composé par tous les symboles de bloc ;
- R : est une expression régulière capable de générer toutes les chaînes de caractères nécessaires ;

Le langage L peut représenter une signature pour le programme d'application. L'exécution du programme est permise si le chemin correspondant dans le graphe de contrôle du flot d'exécution produit une chaîne de caractères appartenant au langage L. Dans le cas contraire une erreur dans le flot d'exécution est produite. Le langage *L* correspondant à l'application illustrée dans la Figure 1.1 est donc :

$L = (A, R)$ ou :

- A = (a, b, c, d)
- R = a(b | c*)d

Par exemple, si l'exécution du programme produit la séquence de symboles «*acccd*», nous décidons que la séquence des symboles appartient à l'espace d'exécutions correctes. Si le programme produit la séquence des symboles «*abdcd*», nous décidons qu'une erreur s'est produite dans le flot d'exécution du programme. Ceci parce que cette dernière séquence ne fait pas partie des séquences possibles dans le langage L.

L'avantage de cette méthode est le surcoût réduit en temps d'exécution et en espace mémoire. Cependant, son efficacité n'est pas garantie dans le cas où les fautes provoquent soit l'arrêt du système soit la perte de séquencement. En effet, la vérification du flot d'exécution étant réalisée à la fin du programme, après que la chaîne des symboles est complète, toute faute faisant que cette portion du programme ne s'exécute pas ne pourra pas être identifiée de manière précise. Les résultats d'expériences d'injection de fautes montrent que le taux de détection n'est pas élevé [29].

B.3. Le Bloc de recouvrement

Des techniques basées sur celle appelée *Bloc de Recouvrement* (Recovery Block), sont parmi les méthodes les plus populaires utilisées actuellement pour la détection et la correction de fautes. Le principe du *Bloc de Recouvrement* a été présenté par Horhing [30] et Randell [31].

Ses éléments de base sont illustrés dans la Figure 1.2, ils comprennent:

- Le module principal du programme élémentaire ;
- Un nombre n de modules alternatifs - modules qui devraient effectuer la même opération que le module primaire ;
- Un contrôleur de réception – module qui est en charge du test effectué sur la sortie du module primaire et des modules alternatifs pour confirmer que les résultats produits sont corrects ;

Afin de fournir la tolérance aux fautes et un fonctionnement continu en cas d'échec de fonctionnement, le *Bloc de Recouvrement* a besoin d'un ou plusieurs modules alternatifs. Les modules alternatifs ont tous la même fonctionnalité que le module primaire, mais sont physiquement différents du module primaire. En d'autres termes, les codes des programmes correspondants sont stockés dans des zones mémoires différentes.

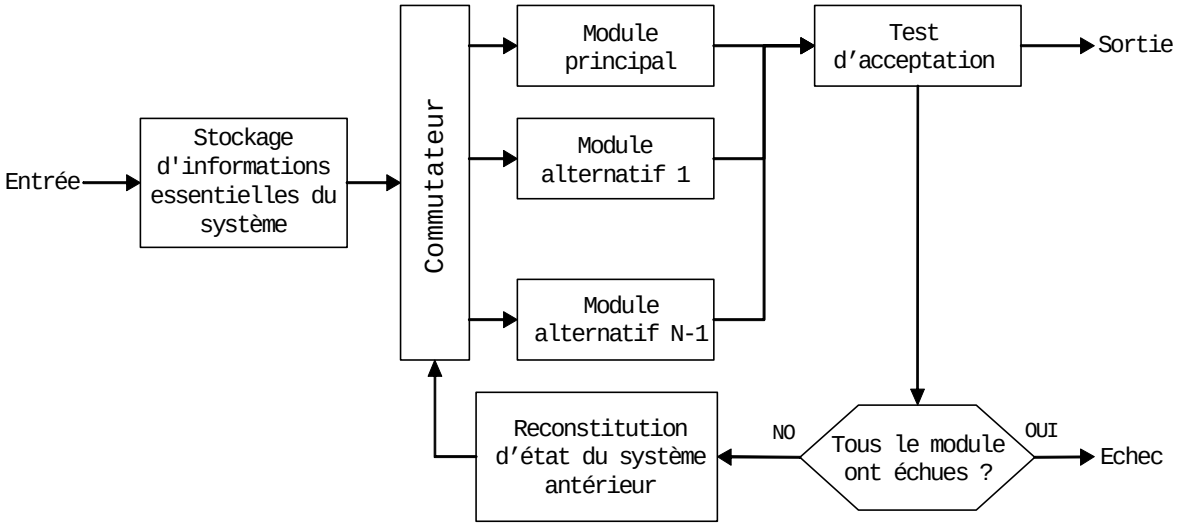


Figure 1.2 : Schéma bloc de Bloc de Recouvrement

Les différents modules composant le *Bloc de Recouvrement* peuvent eux-mêmes être hiérarchiques et incorporer des sous-modules internes de recouvrement. Un bloc réalisant le contrôle des sorties des modules, noté « *test d'acceptation* » (voire Figure 1.2) fournit une décision

sur la validité des réponses. Le *Bloc de Recouvrement* nécessite une *mémoire de recouvrement* et une structure de stockage des informations essentielles concernant l'état courant du système. Elles sont appelées *points de recouvrement* ou *points de contrôle.*

En entrée du bloc de recouvrement un point de contrôle est établi. Ce point de contrôle est stocké dans la *mémoire de recouvrement,* et contient toutes les données essentielles, décrivant l'état actuel du système. Quand le point de contrôle a été stocké le module primaire est exécuté. Après son exécution les résultats produits sont soumis au test d'acceptation.

Si les résultats sont considérés corrects, le bloc de recouvrement est terminé et les résultats sont passés à la sortie du bloc de recouvrement. Sinon, le test d'acceptation refuse les résultats d'un module ceci dans l'un des quatre cas suivants :

- Une erreur dans le fonctionnement d'un module, détectée par le test d'acceptation ;
- L'exécution d'un module ne se termine pas, ceci est détecté par un chien de garde ;
- Une erreur est détectée pendant l'exécution d'un module par un autre mécanisme de détection d'erreurs (par exemple des mécanismes d'exception tels la division par zéro, instruction illégale,…) ;
- L'exécution d'un bloc interne de recouvrement a échoué

Dans le cas où les résultats ne sont pas acceptés, une procédure de reprise est lancée. Cette procédure reconstituera l'état du système stocké dans la *mémoire de recouvrement*. Cette forme de recouvrement est appelée *retour à l'état antérieur (*backward recovery*)*. Quand le recouvrement est terminé, le premier module de remplacement est exécuté, et les résultats sont soumis à nouveau au contrôle de réception. Si le test est réussi, le bloc de recouvrement se termine. Cependant, dans le cas où l'exécution du module de remplacement échoue, l'état du système est de nouveau récupéré et le module de remplacement est remplacé à son tour par le module suivant. Les résultats seront vérifiés de nouveau par le contrôle de réception.

Cette suite d'événements s'arrêtera dans l'un des cas suivants : (a) un module produit les résultats corrects (qui passent le contrôle de réception) ou (b) tous les modules ont échoué et une erreur est signalée. Il est préférable d'utiliser le bloc de recouvrement avec un chien de garde pour vérifier que les systèmes en temps réel [32] respectent les contraintes de temps qui leur sont imposées.

Puisque le bloc de recouvrement utilise une *mémoire de recouvrement* et un mécanisme de rétablissement d'erreur, il nécessite un espace mémoire supplémentaire et un temps supplémentaire de rétablissement en cas d'erreurs. Le surcoût en terme d'espace mémoire est principalement dû à l'espace supplémentaire requis pour stocker le code des modules alternatifs et du contrôle de

réception, et à l'espace requis pour la *mémoire de recouvrement*. Le surcoût en terme de temps d'exécution dépend principalement du temps requis pour effectuer le contrôle de réception et de l'implémentation de la *mémoire de recouvrement*.

B.4. L'approche multi-version basée sur l'utilisation de plusieurs versions du même programme

La méthode dite Programmation à N-versions (*N-version programming*) [33] est une technique basée sur la réplication des modules du programme considéré. Les répliques d'un module sont conçues pour satisfaire les mêmes propriétés que le module du programme cible. La décision de la validité des résultats est réalisée par comparaison des résultats de toutes les versions des modules (le module de programme de base et ses copies).

Cette technique consiste en fait à générer indépendamment N versions des programmes. Ces N versions fonctionnent en parallèle produisant des résultats qui sont sujets à un mécanisme de décision, appelé *voteur*. Si la majorité des N versions présente un même résultat, ce résultat sera considéré comme étant le résultat correct. Dans le cas contraire, le système a échoué.

L'utilisation d'un algorithme générique de décision pour sélectionner la sortie correcte différencie cette approche de l'approche de blocs de recouvrement, ce dernier exigeant un contrôle de réception dépendant de l'application.

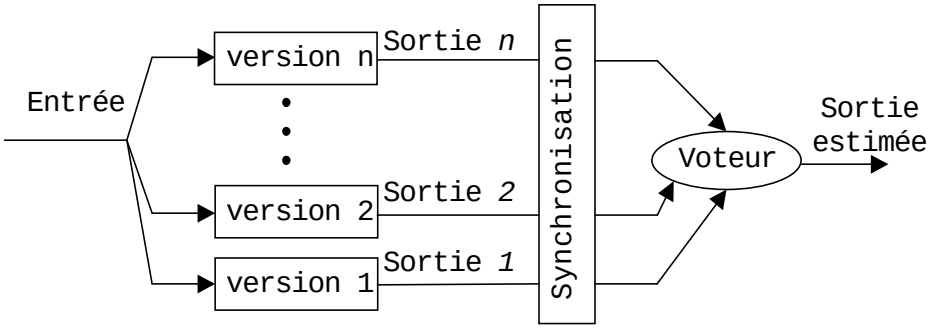


Figure 1.3 : Le schéma de N versions des programmes

La Figure 1.3 montre la structure à la base de cette technique qui comporte :

- Les N versions du programme : modules logiciels générés indépendamment à partir de la spécification initiale ;
- Un mécanisme de décision : le *voteur* qui décide le résultat final des calculs des N versions des programmes ;

§ Un programme de surveillance : c'est une structure logicielle utilisée pour synchroniser les N modules logiciels et le mécanisme de décision.

Les N versions des programmes sont générées indépendamment : chaque version est développée en utilisant différents algorithmes et, dans la mesure du possible, différents langages, compilateurs et systèmes d'exploitation.

Afin de permettre au mécanisme de décision d'atteindre son objectif, les sorties des N versions sont synchronisées. Le programme de surveillance contrôle l'interaction entre les N versions des programmes et le mécanisme de décision. Il manipule également la partie du mécanisme de synchronisation qui fixe les états des opérations pour chacune des N versions. Initialement, une version est dans un état inactif. Au moment où elle est appelée par le programme de surveillance, elle entre dans un état d'attente. La version attend un signal représentant une demande de service. Ce signal met la version du programme dans un état d'exécution. Si une condition d'interruption est signalée, l'exécution du programme est arrêtée et le programme entre dans l'état initial (inactif). Autrement, le programme finit son exécution et produit un résultat quand un point de synchronisation est atteint. Ensuite, le programme de surveillance est informé que le résultat est prêt. Ces états, ainsi que la transition entre les modules des programmes et le programme de surveillance, sont illustrés dans la Figure 1.4.

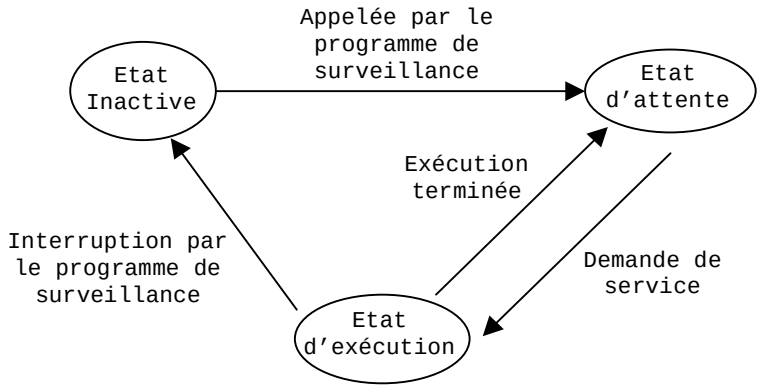


Figure 1.4 : Transitions d'état d'une version du programme

Cette méthode a des bonnes performances au sens de la capacité de détection ou correction des fautes, mais elle nécessite des ressources matérielles supplémentaires. Le surcoût de temps d'exécution est essentiellement pris par le mécanisme de synchronisation des modules logiciels et le mécanisme de décision.

B.5. Tolérance aux fautes implémentée par logiciel

L'ensemble des techniques de tolérance aux fautes basées exclusivement sur la modification du code d'un programme d'application est groupé sous l'appellation SIFT *(software implemented fault tolerance*). Parmi ces méthodes, qui ont l'ambition d'être mises en œuvre sans modifications matérielles du système considéré, celle proposée récemment par le groupe CAD du « Politecnico » di Torino [34] peut être considérée comme étant la plus représentative. Cette méthode introduit des redondances aussi bien au niveau des données que du code du programme en étude. Un ensemble de règles concrétise les différentes techniques de redondance utilisées pour détecter les erreurs affectant les données ou les instructions du programme.

Erreurs affectant les données

L'idée de base est la duplication de toutes les variables dans le programme et la vérification de la cohérence des valeurs des deux variables (la variable originale et sa copie) chaque fois qu'elles sont utilisées. Pour chaque opération de lecture sur une variable, l'opération est répétée pour la variable dupliquée et un contrôle de cohérence est effectué. S'il y a une différence entre la valeur de la variable originale et la valeur de la variable dupliquée, alors une erreur est signalée. Les erreurs affectant les variables après leur dernière utilisation ne sont pas détectables. Trois règles ont été définies pour réaliser ces modifications :

- Règle #1 : chaque variable **x** dans le programme est reproduite : alors **x1** et **x2** sont les noms des deux variables résultantes,
- Règle #2 : chaque opération sur la variable **x** est effectuée sur **x1** et **x**2,
- Règle #3 : après chaque opération de lecture sur **x**, les deux copies **x1** et **x2** sont examinées pour assurer la cohérence ; une procédure de détection des erreurs est lancée si une incohérence est détectée.

L'application de ces règles est illustrée dans la Figure 1.5 pour deux exemples élémentaires.

Code original	Code modifié

`a = b;`	`a1 = b1;`
	`a2 = b2;`
	`if(b1 != b2)`
	`  error();`

`a = b + c;`	`a1 = b1 + c1;`
	`a2 = b2 + c2;`
	`if((b1 != b2)||(c1 != c2))`
	`  error();`

Figure 1.5 : Exemple d'application de règles visant les données

Dans le cas d'une fonction, les paramètres passés ainsi que les valeurs retournées, sont considérés comme étant des «*variables*». Par conséquence, les règles définies ci-dessus peuvent être utilisées de la même manière que celles montrées dans les exemples précédents. La Figure 1.6 montre un exemple d'application de ces règles aux paramètres d'une fonction.

Code original	Code modifié

`int main()`
`{`
`  . . .`
`  res = execute(a);`
`  . . .`
`}`
`int execute(int p)`
`{`
`  int ret;`
`  . . .`
`  ret = p + 4;`
`  return ret;`
`}`
	`int main()`
	`{`
	`  . . .`
	`  execute(a1,a2,&res1,&res2);`
	`  . . .`
	`}`
	`void execute(int p1,int p2,int*res1,int*res2)`
	`{`
	`  int ret1,ret2;`
	`  . . .`
	`  ret1 = p1 + 4;`
	`  ret2 = p2 + 4;`
	`  if(p1 != p2) error();`
	`  *res1 = p1;`
	`  *res2 = p2;`
	`  return;`
	`}`

Figure 1.6 : Application des règles pour la détection d'erreurs affectant des paramètres d'une fonction

Erreurs affectant le code du programme

Les erreurs affectant le code du programme peuvent être divisées en deux types, selon la manière dont elles transforment les instructions :

- Type E : l'erreur change l'opération associée à l'instruction, sans changer le flot global d'exécution du programme (en changeant par exemple une opération d'addition en une opération de multiplication),
- Type E2 : l'erreur change le flot global d'exécution du programme (par exemple en transformant une opération d'addition en une instruction de branchement).

L'ensemble d'instructions d'un programme est divisé en deux catégories selon le type d'action déclenchée.

- Instructions d'affectation : instructions manipulant uniquement des données (ex : lecture/écriture de données, calculs arithmétiques ou logiques, etc.)
- Instructions de contrôle : instructions affectant le flot global d'exécution (ex : les instructions conditionnelles, les appels de fonctions et les retours de fonctions, les branchements).

Erreurs de type E1 affectant les instructions d'affectation

Dans ce cas les erreurs perturbent les instructions d'affectation des données sans modifier le flot d'exécution du programme. Ces erreurs sont détectées par les règles proposées pour la détection d'erreurs affectant les données.

Erreur de type E2 affectant les instructions d'affectation

Pour faire face à ce type d'erreurs, le code du programme est décomposé en blocs élémentaires. Un «*flag*» appelé *indicateur global d'exécution* que nous noterons par la suite ***fge*** (*flot global d'exécution*) est défini afin d'associer une identité à chaque bloc élémentaire du programme. Au début de chaque bloc élémentaire une valeur est assignée au ***fge***. Une vérification du ***fge*** est introduite également à la fin de chaque bloc élémentaire afin de détecter les erreurs affectant le flot d'exécution.

Les règles suivantes sont alors proposées, afin de vérifier si toutes les instructions d'un bloc élémentaire sont exécutées dans l'ordre correcte :

- Règle 4 : une valeur ***ki*** est associée à chaque bloc élémentaire ***i*** dans le programme,
- Règle 5 : une variable de contrôle d'exécution ***fge*** est définie ; une instruction assignant à ***fge*** la valeur du ***ki*** est ajoutée au début de chaque bloc élémentaire ***i*** ; une vérification sur la valeur de ***fge*** est réalisée à la fin de chaque bloc élémentaire.

La Figure 1.7 montre un exemple d'application des règles 4 et 5 visant les erreurs affectant les blocs élémentaires d'un programme. Un bloc élémentaire d'un programme est composé d'un ensemble d'instructions qui sont exécutées de manière séquentielle. Les blocs élémentaires ne contiennent donc pas d'instructions de branchements. La solution proposée est basée sur la vérification enligne du flot d'exécution, pour tenter de détecter les différences produites par des erreurs dans le flot d'exécution. Ces règles ont été définies afin de détecter des erreurs qui modifient les instructions d'affectation des données provoquant des sauts incorrects (par exemple en affectant l'opérande d'un saut existant, ou en transformant certaines instructions en instructions de saut) altérant le flot d'exécution. Cependant, les auteurs notent que les règles 4 et 5 ont des possibilités limitées de détection ; les erreurs produisant un saut à l'intérieur du même bloc élémentaire ou un saut au début d'un bloc élémentaire ne pourront pas être détectées.

Code original	Code modifié

/*Débout d'un bloc élémentaire*/
. . .
/*Fin d'un bloc élémentaire*/
	/*Débout d'un bloc élémentaire*/
	fge = 12;
	. . .
	if(fge != 12)
	error();
	/*Fin d'un bloc élémentaire*/

Figure 1.7 : Exemple d'application des règles 4 et 5

Erreurs affectant les instructions de contrôle

Dans ce cas les erreurs affectent les instructions de contrôle qui définissent l'ordre dans lequel doivent s'exécuter les instructions composant les blocs élémentaires du programme. Les effets des erreurs sur les instructions de contrôle sont l'évaluation incorrecte de la condition pour les instructions de branchement conditionnel ou l'affectation des appels et des retours de procédure. Les instructions de contrôle sont affectées par la modification d'une des données impliquées dans l'évaluation de la condition d'une instruction conditionnelle, par la modification de l'adresse cible de branchement ou par la modification du code de l'instruction de branchement.

Le principe est de vérifier l'évaluation correcte de la condition et le résultat du branchement. Si la valeur de la condition reste la même après la réévaluation, nous décidons qu'aucune erreur ne s'est produite. En cas de changement de la condition une procédure de traitement d'erreur est appelée.

Afin de détecter les erreurs affectant les instructions de contrôle, la règle #6 est définie.

- Règle #6 : la condition est réévaluée pour la clause FAUSSE au début du corps d'instruction correspondant à l'instruction de contrôle et pour la clause VRAI à la fin du corps de l'instruction de contrôle. Si la valeur de la condition n'est pas la même après la réévaluation, une erreur est signalée.

La Figure 1.8 fournit un exemple de l'application de la règle définie ci-dessus pour une instruction de contrôle.

Code original	Code modifié

`if(condition == VRAI)`
`{/* Bloc A */`
`. . .`
`}`
`else`
`{/* Bloc B */`
`. . .`
`}`
	`if(condition == VRAI)`
	`{/* Bloc A */`
	`if(condition == FAUSSE)`
	`error();`
	`. . .`
	`}`
	`else`
	`{/* Bloc B */`
	`if(condition == VRAI)`
	`error();`
	`. . .`
	`}`

Figure 1.8 : Transformation visant les instructions de contrôle

Une attention particulière doit être consacrée aux instructions d'appel et de retour de fonctions. Afin de détecter les erreurs qui affectent ce type d'instructions, les règles suivantes sont :

- Règle #7 : une valeur ***kj*** est associée à chaque fonction ***j*** dans le programme ;
- Règle #8 : avant le retour de chaque fonction, une valeur ***kj*** est assignée au ***fge*** ; une vérification de la cohérence sur la valeur du ***fge*** est introduite après tous les appels de fonction ;

Un exemple d'application des ces règles est illustré dans la Figure 1.9. Les transformations sur les paramètres de la fonction ne sont pas appliquées pour la clarté de l'exemple.

Code original	Code modifié

`void main()`
`{`
`. . .`
`a = ma_fonction(int p);`
`. . .`
`}`
`int ma_fonction(int p)`
`{`
`/*corps de fonction*/`
`. . .`
 	
`return 0x55;`
`}`
	`void main()`
	`{`
	`. . .`
	`a = ma_fonction(int p);`
	`if(fge != 12)`
	`error();`
	`. . .`
	`}`
	`int ma_fonction(int p)`
	`{`
	`/*corps de fonction*/`
	`. . .`
	
	`fge = 12;`
	`return 0x55;`
	`}`

Figure 1.9 : Transformation visant des erreurs affectant des appels et retours des fonctions

Les résultats expérimentaux de l'application de cette méthode présentés en [34] montrent une bonne efficacité de détection d'erreurs de type basculement de bit affectant les données et le code du programme. Dans la référence mentionnée des essais d'injections de fautes réalisées sur des programmes simples à l'aide d'un simulateur d'un processeur du commerce, révèlent une excellente capacité de détection : dans le pire des cas seulement deux parmi les 2000 fautes injectées, échappent aux mécanismes de détection implantés.

Cette méthode présente une mise en œuvre aisée pour des programmes décrits dans un langage de haut niveau. Cependant elle entraîne un ralentissement du système exécutant le programme qui peut atteindre un facteur de 5 pour des applications qui ne nécessitent pas un grand volume de calcul. Ceci peut être inacceptable dans un système où le temps d'exécution est une contrainte majeure. De plus, des essais sous radiations effectués sur des programmes exécutés sur une architecture à base d'un Transputer [35] ont mis en évidence une capacité de détection plutôt de l'ordre de 50%. De plus, des tests sous radiation effectués sur la même architecture et programmes, on montré que l'efficacité de détection de fautes réellement produites comme conséquence des effets de l'environnement radiatif, étaient bien plus faible (d'environ un ordre de grandeur) que celle calculée d'après des essais d'injection de fautes [36]. Ceci n'est pas étonnant compte tenu d'une part de la nature aléatoire des upsets aussi bien dans l'instant d'occurrence que dans la cible perturbée, conditions difficiles à être reproduites de manière fidèle par des méthodes d'injection de fautes couramment utilisées, d'autre part du fait que les upsets peuvent affecter des éléments de mémorisation inconnus du programmeur, et donc pas possibles à prendre en compte lors des simulation de fautes.

Ces résultats préliminaires, bien que soulevant des limitations intrinsèques de cette méthode SIFT, ont encouragé des recherches approfondies qui ont visé trois objectifs principaux :

- L'obtention de résultats expérimentaux de l'application de la méthode à divers circuits et programmes afin de récolter des évidences, même contradictoires, de son efficacité.
- L'analyse des fautes non détectées lors de ces expériences, dans le but de fournir des éléments permettant l'amélioration de l'ensemble de règles,
- La révision de l'ensemble des règles, cette fois dans l'optique d'obtenir une réduction substantielle du temps d'exécution des programmes durcis résultants.

La mise en œuvre de ces recherches ne peut être envisagée sans le développement et l'utilisation d'outils automatiques que ce soit pour l'obtention des programmes modifiés selon les règles étudiées, que pour la réalisation d'expériences d'injection de fautes ou d'essais sous radiation. Les chapitres suivants aborderont ces différents aspects, qui résulteront en une variante de la méthode SIFT, dont l'efficacité sera validée par des expériences appropriées.

Chapitre 2

Etude d'un nouvel ensemble de règles

- Introduction
- Exploration de la méthode
  - Caractéristiques d'un programme
    - Instructions élémentaires
    - Instructions de branchement
  - Groupe de règles visant les erreurs affectant les données
    - L'interaction directe avec les données
    - L'interaction indirecte avec les données
  - Groupe de règles visant les erreurs affectant les blocs élémentaires
  - Groupe de règles visant les erreurs affectant les instructions de contrôle
    - Erreurs affectant les instructions conditionnelles de contrôle
    - Erreurs affectant les instructions inconditionnelles de contrôle
- Evaluation de la méthode de détection
    - Le temps d'exécution du programme
    - L'espace mémoire d'un programme
  - Application de la transformation $\Psi|dd$
  - Application de la transformation $\Psi|fge$
  - Application de la transformation $\Psi|bd$
- Génération automatique des programmes durcis
  - Flot de génération des programmes durcis
  - Le translateur C2C
- Conclusions

I. Introduction

Dans ce chapitre nous décrirons la démarche suivie pour aboutir à une approche purement logicielle pour la détection d'erreurs de type upset. Comme présenté dans l'introduction de cette thèse, ce type d'erreur peut survenir dans des architectures numériques à base des processeurs comme conséquence des effets de l'environnent (radiations, EMC, etc.). La méthode est basée sur un nouvel ensemble de règles, issu d'une analyse approfondie d'une méthode existante [34]. Les nouvelles règles sont appliquées directement sur des programmes écrits en langage de haut niveau (par exemple le langage C). Ces transformations introduisent des redondances au niveau des données et au niveau du code du programme permettant d'obtenir un nouveau programme. Le programme obtenu a la même fonctionnalité que le programme original, mais possède en plus la capacité de détecter des basculements de bits affectant le contenu de la mémoire de données, le code du programme ou le processeur (les registres généraux, les registres de contrôle, la pile, la mémoire cache, etc.).

La méthode de détection proposée est dédiée aux erreurs transitoires affectant le contenu d'éléments de mémorisation inclus dans l'architecture sous étude. Il est important de rappeler, que ce type d'erreur n'est pas destructif : les cellules mémoires affectées peuvent être réécrites avec une nouvelle valeur. Cependant, cette méthode peut aussi détecter des erreurs permanentes dans le système telles que les bits collés résultant du cumul de dose suite à un fonctionnement prolongé en ambiance radiative. Les transformations que nous proposons étant appliquées sur le code du programme sont indépendantes du processeur qui exécute le programme, aussi bien que de l'architecture du système cible. Cependant, il doit être noté que les drapeaux d'optimisation du compilateur utilisés pour produire le code binaire avec capacité de détection doivent être désactivés afin de permettre de maintenir la redondance introduite sur les données et le code du programme.

Ce chapitre s'articule autour de cinq sections. Dans la deuxième section seront présentés les nouveaux concepts de base de la méthodologie de détection proposée. Dans la troisième une évaluation de la méthodologie de détection sera effectuée, dans la quatrième section nous proposons un flot de génération automatique des programmes durcis. La cinquième section donnera les conclusions et les perspectives.

II. Exploration de la méthode

Dans cette sous-section nous nous sommes focalisés sur les erreurs de type SEU affectant les données et le code d'un programme indépendamment de l'emplacement où elles sont stockées (mémoire principale, mémoire cache, registre d'instruction du processeur). Il doit être noté que ces transformations introduites complètent d'autres mécanismes de détection (EDM) qui peuvent exister dans le système capables d'identifier une partie de ces erreurs. Un exemple d'un tel mécanisme : les procédures de déclenchement d'exceptions de type instruction illégale ou de contrôle de logiciel qui mettent en application des EDMs additionnels non-systématiques introduites par le programmeur.

A. Caractéristiques d'un programme

Nous présentons dans cette section les concepts qui sont à la base de la définition de la méthodologie de détection proposée.

Un programme ***P*** est un ensemble spécifique d'opérations exécutées par un processeur. Il est caractérisé par un ensemble d'instructions et de données implanté selon un processus défini aboutissant à une solution. Les instructions et les données composant un programme seront définies et classifiées à la suite.

Suivant leur rôle dans le programme, les données du programme sont partagées en trois catégories de base :

- Données d'entrée : appliquées en entrée d'un programme
- Données intermédiaires : utilisées pour le calcul de résultat final d'un programme
- Données de sortie : résultat final de calcul d'un programme

L'ensemble de données désignant un programme est représenté dans la Figure 2.1.

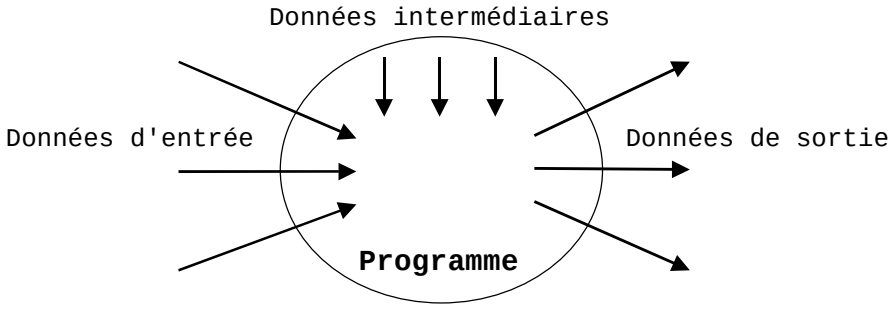


Figure 2.1 : Représentation de l'ensemble de données pour un programme

L'ensemble des instructions d'un programme est divisé en deux catégories selon le type d'action déclenchée.

- Instructions élémentaires : des instructions affectant que des données (ex : lecture/écriture de données, les calculs arithmétiques ou logiques, etc.)
- Instructions de contrôle : des instructions affectant le flot global d'exécution (ex : les boucles, les appels de fonctions et les retours de fonctions, des branchements)

A.1. Instructions élémentaires

Les instructions d'affectation permettent de réaliser des transferts de données entre les registres du processeur et la mémoire, ou d'effectuer des opérations entre les registres et une donnée puis stocker le résultat dans un registre ou dans la mémoire. Elles peuvent être classifiées comme suit :

- Instructions de transfert (par exemple : écriture d'un registre vers la mémoire ou lecture de la mémoire vers un registre)
- Instructions arithmétiques (par exemple : addition, soustraction)
- Instruction logiques (par exemple : ET, OU, XOR, décalage)

A.2. Instructions de branchement

Ce type d'instruction permet de sauter à une instruction non consécutive à l'instruction en cours. En l'absence de ce type d'instruction le processeur passe automatiquement à l'instruction suivante. Les instructions de branchement permettent donc de modifier le flot d'exécution et ainsi de choisir la prochaine instruction à exécuter.

Nous distinguons deux sortes de branchements:

- Les branchements conditionnels - en fonction de la valeur logique d'une condition, le processeur effectuera le branchement indiqué ou exécutera l'instruction suivante :
  - condition satisfaite : exécution de l'instruction située à l'adresse indiquée,
  - condition non satisfaite : exécution de l'instruction suivante.
- Les branchements inconditionnels - affectent la valeur définie de registre *compteur du programme*. Les branchements inconditionnels incluent :
  - des appels des fonctions,
  - des retours des fonctions,
  - des branchements.

B. Groupe de règles visant les erreurs affectant les données

L'une des zones sensibles aux erreurs de type SEUs est composée de l'ensemble des données d'un programme. Les erreurs affectant les données peuvent être divisées en deux types :

- Erreur affectant les données par interaction directe
- Erreur affectant les données par interaction indirecte

B.1. L'interaction directe avec les données

L'erreur affecte la zone mémoire dans laquelle les données sont stockées ou le contenu d'un registre de données. Il s'agit des mémoires RAM externes/internes ou bien de la mémoire cache de données (dans le cas des processeurs complexes). Dans la suite nous allons illustrer un exemple d'erreur affectant une donnée par la modification d'un registre interne du processeur.

Exemple : soit `var=var/5` l'instruction de haut niveau que nous nous proposons d'analyser, où `var` est une variable définie dans le programme. Au niveau du langage assembleur (pour le processeur Transputer T225), cette instruction donne lieu à quatre instructions illustrées dans le Tableau 2.1.

Tableau 2.1 : Représentation de l'instruction de haut niveau `var=var/5` en niveau assembleur

Code assembleur	Signification
LDL *var*	Charger dans l'accumulateur la valeur de la variable *var*
LDC 5	Transférer la valeur de l'accumulateur dans le registre B et charger dans l'accumulateur la constante 5
DIV00	Diviser la valeur du registre B avec la valeur de l'accumulateur, le résultat est stocké dans l'accumulateur
STL *var*	Ecrire le contenu de l'accumulateur dans la mémoire réservée à la variable *var*

Dans ce cas, le processeur utilise seulement deux registres (l'accumulateur et le registre B) pour le calcul de la variable `var`. L'apparition d'un basculement de bit dans un des deux registres utilisés mène à un changement du résultat final. Par exemple, le registre B contenant la valeur de la variable `var` est corrompu avant l'exécution de l'instruction `DIV`. Le résultat de la division entre la valeur du registre B et la valeur de l'accumulateur sera incorrecte si le renversement du bit se produit avant que la valeur soit consommée.

B.2. L'interaction indirecte avec les données

Les données sont affectées par le biais de la modification du code du programme (par la transformation d'une instruction en une autre sans changement du flot global d'exécution ou par l'apparition d'un saut intempestif dans le programme). Dans l'exemple suivant, nous mettrons en évidence l'effet du changement du code du programme sur les données.

Pour l'illustration de ce type d'erreur, nous prendrons le même exemple utilisé pour l'interaction directe avec les données. Il existe de nombreuses possibilités de fausser le résultat final de (*var = var/5*) en corrompant le code du programme. Chaque possibilité correspond à la corruption par un basculement d'un bit d'une instruction assembleur. Dans cet exemple, nous supposons que le code de la deuxième instruction (*DIV*) est modifié.

La Figure 2.2 montre l'état du code avant et après l'occurrence d'un basculement de bit. L'inversion d'un des bits de l'instruction *DIV* transforme cette dernière instruction en l'instruction *DIST,* dont l'effet est la ***désactivation du timer.*** La transformation introduite ne change pas le flot d'exécution et l'instruction de haut niveau (*var = var/5*) devient (*var = var*). Comme conséquence de cette modification, le processeur exécutera l'instruction résultante comme conséquence de basculement de bit, donc le résultat final sera incorrect.

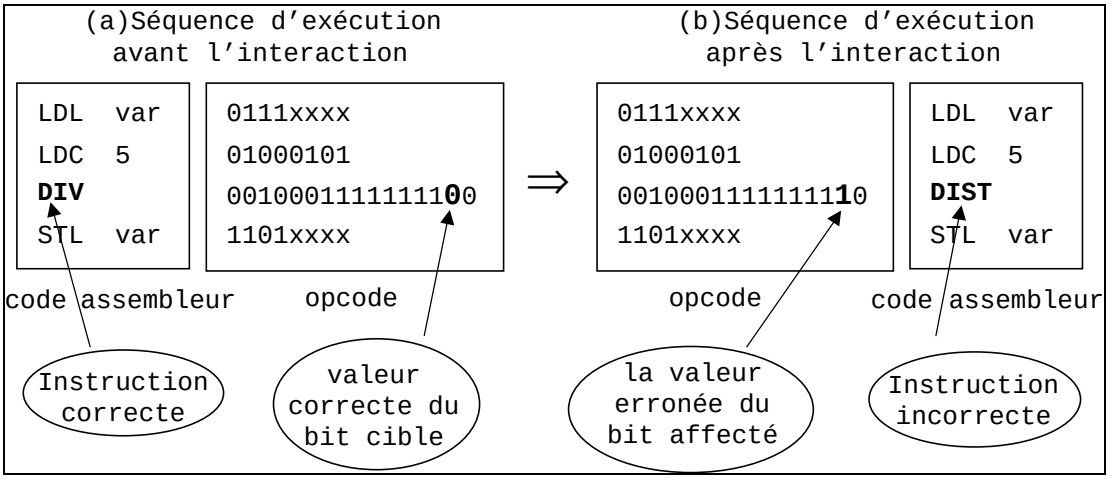


Figure 2.2 : Modification de code d'une instruction due à l'apparition d'un basculement d'un bit

Pour éviter les effets des basculements des bits affectant le segment de données, nous proposons un nouvel ensemble de règles que nous appellerons les règles de *Duplication des Données (DD).* Ce groupe de règles introduit des redondances pour les variables définies dans le programme, indépendamment de l'emplacement où les variables sont stockées (dans la mémoire principale, dans la mémoire cache, ou dans un registre du processeur). Nous nous référons

seulement au code de haut niveau. Les règles proposées complètent d'éventuels mécanismes de détection d'erreurs qui peuvent exister dans le système (par exemple ceux basés sur des bits de parité ou sur des codes de correction d'erreurs). Il est important de noter que les possibilités de détection des règles proposées dans le groupe *DD* sont plus élevées, puisqu'elles concernent les erreurs affectant les données, indépendamment du nombre de bits modifiés.

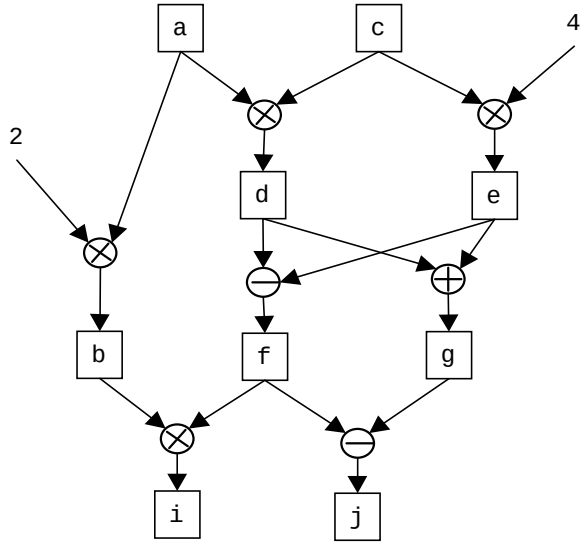
La première étape consiste à définir les relations d'interdépendance entre les variables du programme et les classifier en deux catégories selon leur mode d'utilisation :

- Variables *intermédiaires* : elles sont utilisées pour le calcul d'autres variables,
- Variables *finales* : elles ne participent pas dans de calcul d'autres variables.

Par exemple, dans la Figure 2.3.a nous considérons sept opérations arithmétiques sur les variables *a, b, c, d, e, f, g, i* et *j*; dans la Figure 2.3.b sont représentés les rapports d'interdépendance entre variables. Dans cet exemple, *a, b, c, d, e, f* et *g* sont des variables intermédiaires utilisées dans le calcul des variables *finales i* et *j* (aucune variable n'est dépendante de *i* ou *j*).

```
d = a * c;
e = 4 * c;
b = 2 * a;
f = e - d;
g = d + e;
i = f * b;
j = f - g
```

a)



b)

Figure 2.3 : Relations d'interdépendances entre les variables

Une fois que les rapports d'interdépendance sont établis, toutes les variables dans le programme sont dupliquées. Pour chaque opération effectuée sur une *variable originale*, l'opération est répétée pour sa réplique, que nous appellerons *variable dupliquée*, et donc les relations d'interdépendance entre les *variables dupliquées* sont les mêmes que celles entre les variables originales. Une vérification de l'égalité entre les valeurs des deux variables (*originale* et

dupliquée) est effectuée après chaque opération d'écriture sur les variables *finales*. S'il y a une différence entre la valeur de la variable *originale* et la valeur de la variable *dupliquée*, alors une erreur est signalée.

Les règles composant ce groupe de *Duplication de Données* peuvent être formulées comme suit :

- Règle #1 : chaque variable du programme est classifiée en variable *intermédiaire* ou variable *finale*,
- Règle #2 : chaque variable **x** dans le programme est dupliquée : soient **x1** et **x2** les variables résultantes,
- Règle #3 : chaque opération exécutée sur la variable **x** dans le programme original est exécutée sur les variables résultantes **x1** et **x2**,
- Règle #4 : après chaque opération d'écriture sur une variable **x** qui n'est pas opérande dans une opération sur d'autres variables, les deux copies **x1** et **x2** sont comparées pour assurer la cohérence; une procédure de détection d'erreurs est lancée si une différence est détectée.

La Figure 2.4 illustre l'application de ces règles à un exemple élémentaire. Dans la partie gauche de la figure, nous considérons le cas du programme original où sur la variable ***x*** nous avons une opération d'écriture. En appliquant les règles proposées pour la détection d'erreurs dans les données, nous avons obtenu un nouveau programme représenté dans la partie droite de la figure. L'opération appliquée pour ***x1*** est répétée pour ***x2***, puis, une vérification de la cohérence d'exécution (***x1 = x2***) est introduite.

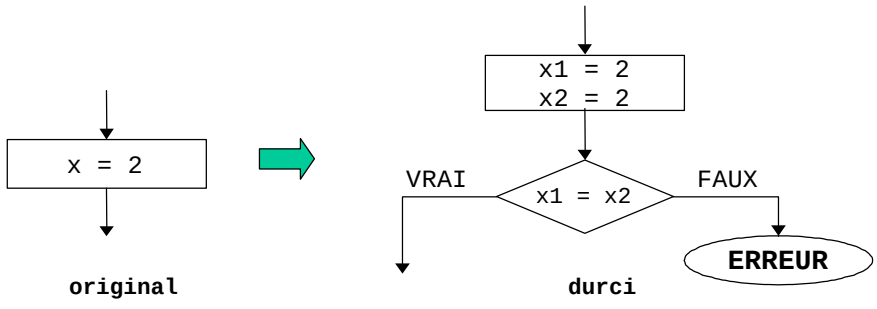


Figure 2.4 : Règles de transformation visant les erreurs affectant les données

Dans le cas des variables interdépendantes, la vérification de la valeur de la variable *originale* et de sa réplique n'est pas effectuée pour les variables qui sont des opérandes dans une opération d'écriture sur d'autres variables. Un exemple de l'application de ces règles dans ce cas est illustré dans la Figure 2.5, où la variable ***x*** intervient dans le calcul de la variable ***y***.

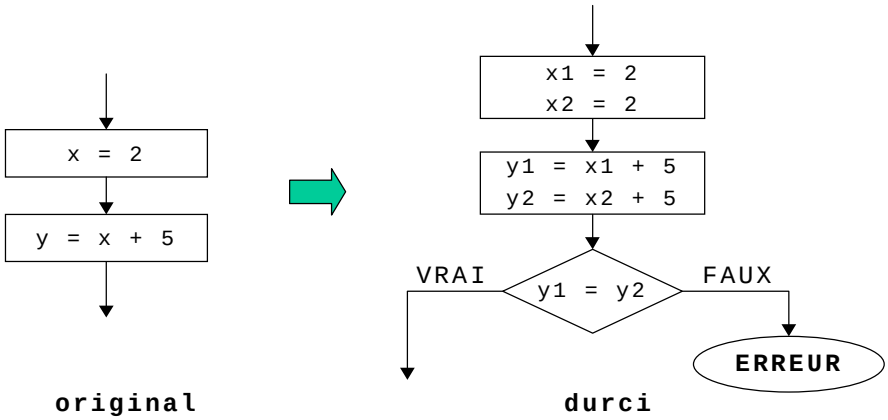


Figure 2.5 : Règles de transformation visant le basculement des bits des données (existence d'une relation d'interdépendance entre variables)

Les variables impliquées dans la condition de test dans le cas des instructions conditionnelles de contrôle, sont également le sujet des règles présentées (voire Figure 2.6).

Code original	Code modifié

`. . .`
`if(b == (a+2)*(c-4))`
`{`
`  /*corps de l'instruction if*/`
`}`
`. . .`
	`. . .`
	`bool1 = (b1 == (a1+2)*(c1-4));`
	`bool2 = (b2 == (a2+2)*(c2-4));`
	`if(bool1 != bool2)error();`
	`if(bool1 && bool2)`
	`{`
	`  /*corps de l'instruction if*/`
	`}`
	`. . .`

Figure 2.6 : Application des règles pour les variables utilisées dans la condition d'une instruction conditionnelle de contrôle

Dans le cas d'une fonction, les paramètres passés ainsi que les valeurs retournées, sont considérés comme étant des «*variables*». Par conséquent, les règles définies ci-dessus peuvent être utilisées de la même manière que celle montrée dans les exemples précédents. La Figure 2.7 montre un exemple de l'application des ces règles aux paramètres d'une fonction.

Code originale	Code modifié

`int main()`
`{`
`  . . .`
`  res = execute(a);`
`  . . .`
`}`
`int execute(int p)`
`{`
`  int ret;`
`  . . .`
`  ret = p + 4;`
`  return ret;`
`}`
	`int ret1; //définie comme variable globale`
	`int ret2; //définie comme variable globale`
	`int main()`
	`{`
	`  . . .`
	`  execute(a1,a2);`
	`  res1 = ret1;`
	`  res2 = ret2;`
	`  if(res1 != res2) error();`
	`  . . .`
	`}`
	`void execute(int p1, int p2)`
	`{`
	`  . . .`
	`  ret1 = p1 + 4;`
	`  ret2 = p2 + 4;`
	`  return;`
	`}`

Figure 2.7 : Application des règles pour la détection d'erreurs affectant des paramètres d'une fonction

Le code du programme en langage C illustré dans la Figure 2.7 est composé de deux fonctions : une fonction appelante ***main*** et une fonction appelée ***execute***. Dans le corps de la fonction ***main***, une valeur entière ***a*** est passée comme argument de la fonction appelée. La fonction ***execute*** retourne une valeur entière affectant la variable ***res*** définie dans la fonction ***main***. Suite à l'application du groupe de règles de ***Duplication de Données***, le paramètre ***p*** d'entrée et la variable de sortie de la fonction sont dupliqués. Comme la variable de sortie de la fonction ***execute*** a été dupliquée, la fonction ***execute*** doit retourner deux valeurs entières. Par conséquent, la définition de la fonction ***execute*** est modifiée (le type ***int*** devient type ***void)***. Afin de passer les deux valeurs de retour de la fonction appelée, nous définissons deux variables globales associées ces variables.

Le point faible de groupe initial de règles vient du fait que le contrôle de la cohérence entre les valeurs des deux variables (originales et reproduites) est effectué après chaque opération de ***lecture*** sur les variables. Dans ce cas les variables de sortie du programme ne sont pas vérifiées. Les erreurs affectant ces variables ne seront pas détectées. De plus, dans le cas des opérations complexes (volume grand de calcul), le groupe initial de règles introduit un surcoût considérable de temps d'exécution du programme. Ceci est dû au contrôle de la cohérence effectué sur les variables participantes dans le calcul des variables de sorties.

L'avantage majeur des nouvelles règles que nous proposons par rapport aux travaux existants réside dans le faite que le contrôle de la cohérence entre les valeurs de deux variables (la variable originale et sa réplique) est effectué après chaque ***écriture*** sur les ***variables finales***. Cela implique

une amélioration du temps d'exécution du programme durci et de l'efficacité de détection d'erreurs affectant les variables du programme.

C. Groupe de règles visant les erreurs affectant les blocs élémentaires

Dans ce cas le basculement de bit perturbe les instructions d'affectation des données au travers de la modification du contenu des registres internes du processeur ou suite à la modification de la mémoire cache d'instructions. Ces instructions ne changent pas le flot d'exécution du programme et elles peuvent être groupées en blocs élémentaires dans un programme. Un bloc élémentaire d'un programme est une séquence d'instructions qui sont exécutées d'une manière séquentielle. Les blocs élémentaires ne contiennent donc pas d'instructions de branchements. La solution proposée est basée sur la vérification en ligne du flot d'exécution, pour tenter de détecter des différences éventuelles dans le flot d'exécution.

Dans la Figure 2.8, nous avons représenté l'apparition d'un saut incorrect dans le code du programme à cause du changement de la valeur d'un seul bit du code d'une instruction. Dans cet exemple, nous avons *var=var+5*, où *var* est une variable définie dans le programme. Dans ce cas, l'instruction de haut niveau utilisée comme exemple correspond à trois instructions en langage assembleur, illustrées dans le Tableau 2.2.

Tableau 2.2 : Représentation de l'instruction `var=var+5` dans en langage d'assembleur

Code assembleur	Signification
LDL *var*	charger dans l'accumulateur la valeur de variable *var*
ADC 5	additionner la valeur de variable *var* avec la constante 5, le résultat est stocké dans l'accumulateur
STL *var*	écrire dans la mémoire réservée pour la variable *var*, le résultat de l'addition

Supposons, comme montré dans la Figure 2.8, que le code de la deuxième instruction (*ADC 5*) est corrompu. Le résultat de la modification est la transformation de l'instruction affectée en une autre instruction (*JMP @PC+5*) dont l'exécution provoque un saut incorrect de l'adresse courante (*@PC*) à l'adresse (*@PC+5*).

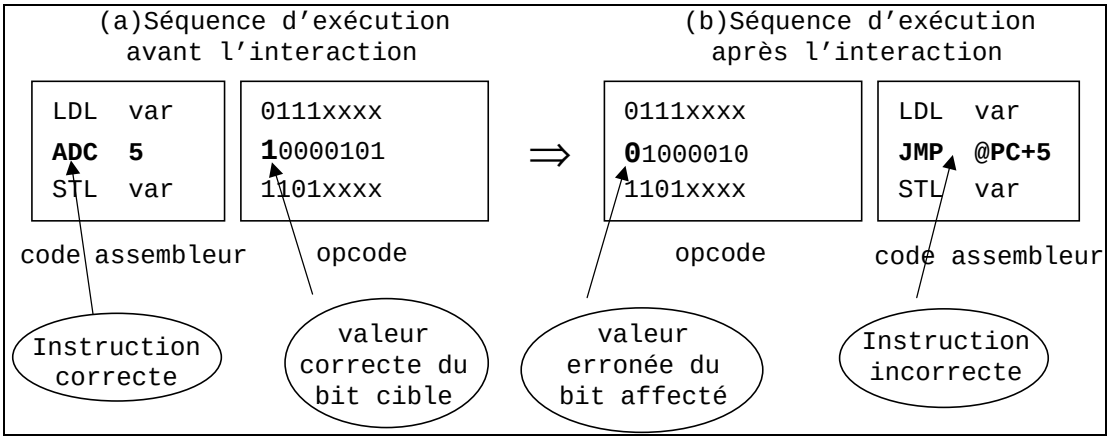


Figure 2.8 : Modification du code d'une instruction due au basculement d'un bit affectant le flot d'exécution

Pour faire face à ce type d'erreurs, les travaux proposés dans [34] apportent une solution qui consiste en la décomposition du code du programme dans des blocs élémentaires. Un «*flag*» ***fge*** (*flot global d'exécution*) est défini afin d'associer une identité à chaque bloc élémentaire du programme. Au début de chaque bloc élémentaire une valeur est assignée au ***fge***. Une vérification du ***fge*** est introduite également à la fin de chaque bloc élémentaire afin de détecter les erreurs affectant le flot d'exécution ; ces erreurs ont comme effet l'apparition des sauts incorrects altérant le flot d'exécution (i.e. : en affectant l'opérande d'un saut existant, ou en transformant certaines instructions en instructions de saut).

Cependant, cette solution a des possibilités limitées de détection dans l'hypothèse où l'erreur provoque un saut incorrect au début d'un autre bloc élémentaire ou bien dans l'hypothèse où la granularité de décomposition d'un programme en blocs élémentaires n'est pas assez fine. Ainsi ce type d'erreurs n'est pas détecté.

Afin de détecter ce type d'erreurs qui échappent aux règles proposées dans les travaux présentés précédemment, nous proposons une extension de deux nouvelles règles ayant comme objectif l'amélioration de la capacité de détection d'erreurs affectant les instructions élémentaires.

La première règle consiste en la définition d'un drapeau appelé *status_bloc*. Ce drapeau prend la valeur *1* si le bloc élémentaire est actif et *0* si le bloc est inactif. Au début et à la fin du chaque bloc élémentaire, la valeur du *status_bloc* est incrémentée modulo 2. Ainsi, la valeur du *status_block* est toujours *1* au début de chaque bloc élémentaire et *0* à la fin du bloc. Si une erreur provoque un branchement incorrect vers le début d'un autre bloc le *status_block* prendra la valeur *0* pour un bloc actif et *1* pour un bloc inactif. Cette situation anormale est détectée au premier contrôle effectué sur le ***fge***. L'application de cette nouvelle règle est illustrée dans la Figure 2.9.

Code original	Code modifié

/*début de bloc élémentaire ***i****/
. . .
/*fin de bloc élémentaire ***i****/
 	
/*début de bloc élémentaire ***j****/
. . .
/*fin de bloc élémentaire ***j****/
	/*début de bloc élémentaire ***i****/
	fge = i ^ (status_bloc ^= 1);
	. . .
	(status_bloc ^= 1);
	if(fge != i) error();
	/*fin de bloc élémentaire ***i****/
	
	/*début de bloc élémentaire ***j****/
	fge = j ^ (status_bloc ^= 1);
	. . .
	(status_bloc ^= 1);
	if(fge != j) error();
	/*fin de bloc élémentaire ***j****/

Figure 2.9 : Transformation de code visant les branchements incorrects au début d'un bloc élémentaire

La deuxième règle que nous proposons concerne les erreurs provocant des branchements incorrects dans l'intérieur du même bloc élémentaire. Une solution possible est de considérer chaque instruction (de haut niveau) du programme comme un bloc élémentaire. Mais, ceci implique un surcoût considérable de temps d'exécution introduit par l'application des règles présentées. Sachant que T_{cfge} (le temps supplémentaire nécessaire pour la vérification du flot d'exécution) est constant pour chaque bloc élémentaire identifié dans le programme, la décomposition du programme dans des blocs élémentaires est faite d'une telle manière que le temps d'exécution de chaque bloc doit être plus grand que T_{cfge}, cela pour réduire l'augmentation du temps global d'exécution du programme.

La solution que nous proposons consiste en deux épates :

- identification des blocs de taille maximale ; la structuration d'un programme en des blocs de taille maximale est donnée par la prisme des instructions de contrôle (voire Figure 2.11)
- analyse de toutes les instructions composant les blocs de taille maximale et décomposition du chaque bloc de taille maximale dans des sous-blocs en fonction du nombre et du type d'instructions qui composent le bloc respectif.

Dans la Figure 2.10 est illustrée une séquence d'instructions composant un bloc de taille maximale. Les premières quatre instructions et la septième sont des instructions d'assignation simples sur des variables. La cinquième et la sixième instruction impliquent un volume de calcul plus grand. Dans ce cas le bloc de taille maximale donné dans la Figure 2.10 sera décomposé en

trois blocs élémentaires comme : *bloc 1* – composé par les instructions 1, 2, 3 et 4 ; *bloc 2* - composé par l'instruction 5 et *bloc 3* – composé par les instructions 6 et 7.

```
instr. 1    i = 0;
instr. 2    j = 2;
instr. 3    k = 3;
instr. 4    m = 5;
instr. 5    n = (n + m * 6)/(k + 6);
instr. 6    p = p*5 - k*6;
instr. 7    a = 7 + i;
```

Figure 2.10 : Exemple d'un bloc élémentaire de taille maximale

Dans certains cas les blocs élémentaires de taille maximale ne peuvent pas être divisés dans des sous-blocs. Par exemple, dans la Figure 2.11 nous considérons un programme composé par 6 blocs de taille maximale. Les blocs *C* et *E* sont composés d'une seule instruction ; par conséquent elles sont indivisibles. La décomposition dans des blocs élémentaires sera effectuée seulement pour les blocs A, B, D et F.

```
void main (void)
{
     instruction 1;
     instruction 2;                Bloc A
     instruction 3;
     instruction 4;
     if (condition 1)
     {
         instruction 4;
         instruction 5;            Bloc B
         instruction 6;
     }
     else
     {
        instruction 7;             Bloc C
        while (condition 2)
        {
            instruction 8;
            instruction 9;         Bloc D
        }
        instruction 10;            Bloc E
     }
     instruction 11;
     instruction 12;               Bloc F
}
```

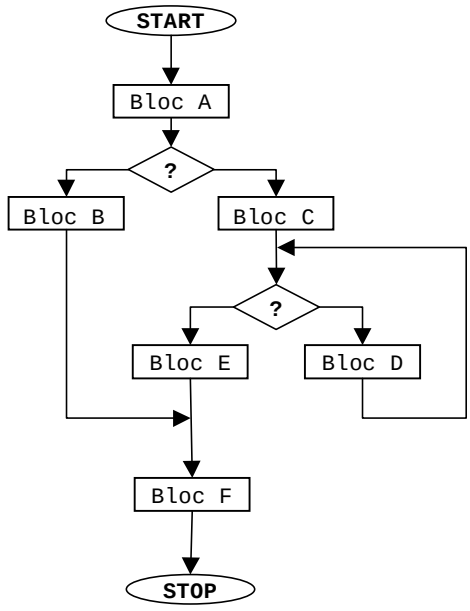


Figure 2.11 : Exemple d'un programme décomposé en blocs de taille maximale

En concordance avec ces modifications présentées, les nouvelles règles proposées pour la détection d'erreurs affectant des instructions élémentaires seront :

- Règle #5 : identification des blocs de taille maximale et décomposition de chaque bloc de taille maximale dans des blocs élémentaires en fonction du nombre d'instructions et du type d'instructions qui composent le bloc en question,
- Règle #6 : un indicateur booléen *status_bloc* est associé à chaque bloc élémentaire du programme ; cet indicateur prend la valeur *0* pour l'état inactif du bloc et *1* pour l'état actif,
- Règle #7 : une valeur ***ki*** est associée à chaque bloc élémentaire ***i*** dans le programme sous étude,
- Règle #8 : une variable de contrôle d'exécution ***fge*** est définie ; une instruction assignant à ***fge*** la valeur ***ki*** ^ ***(status_bloc*** ^= ***1)*** est ajoutée au début de chaque bloc élémentaire ***i*** ; une vérification sur la valeur de ***fge*** est réalisée à la fin de chaque bloc élémentaire.

La condition d'une instruction de contrôle conditionnelle est considérée comme un bloc élémentaire. La Figure 2.12 montre un exemple de l'application des ces règles à la condition d'une instruction conditionnelle de contrôle.

Code original	Code modifié

`. . .`
`if(b == (a+2)*(c-4))`
`{`
`  /*corps de l'instruction if*/`
`}`
`. . .`
	`. . .`
	`fge = 12 ^ (status_bloc ^= 1);`
	`bool = (b == (a+2)*(c-4));`
	`status_bloc ^= 1;`
	`if(fge != 12)error();`
	`if(bool)`
	`{`
	`  /*corps de l'instruction if*/`
	`}`
	`. . .`

Figure 2.12 : Application des règles pour la condition d'une instruction conditionnelle de contrôle

Un indicateur de flot d'exécution est donc associé à l'expression *b==(a+2)*(c-4)*. Le résultat de l'évaluation de la condition est transféré dans une variable de type booléen définie dans le programme. Le test de la condition est effectué seulement sur la variable booléenne.

D. Groupe de règles visant les erreurs affectant les instructions de contrôle

L'erreur affecte les instructions de contrôle qui définissent l'ordre dans lequel doivent s'exécuter les instructions composant les blocs élémentaires du programme. En général les effets des erreurs sur les instructions de contrôle sont l'évaluation incorrecte de la condition pour les instructions de branchement conditionnel ou l'affectation des appels et des retours d'une procédure. Les instructions de contrôle sont affectées par la modification d'une des données impliquées dans l'évaluation de la condition d'une instruction conditionnelle, par la modification de l'adresse cible de branchement ou par la modification du code de l'instruction de branchement.

D.1. Erreur affectant les instructions conditionnelles de contrôle

Le principe est de vérifier l'évaluation correcte de la condition et le résultat du branchement. Si la valeur de la condition reste la même après la réévaluation, nous décidons qu'aucune erreur ne s'est produite. En cas de changement de la condition une procédure de traitement d'erreur est appelée.

Afin de détecter les erreurs changeant les instructions de contrôle, nous proposons les règles suivantes :

- Règle #9 : la condition est réévaluée pour la clause FAUSSE au début du corps d'instruction correspondant à l'instruction de contrôle,
- Règle #10 : la condition est réévaluée pour la clause VRAI à la fin du corps de l'instruction de contrôle.

La Figure 2.13 fournit un exemple de l'application des règles définies ci-dessus pour une instruction de contrôle conditionnelle.

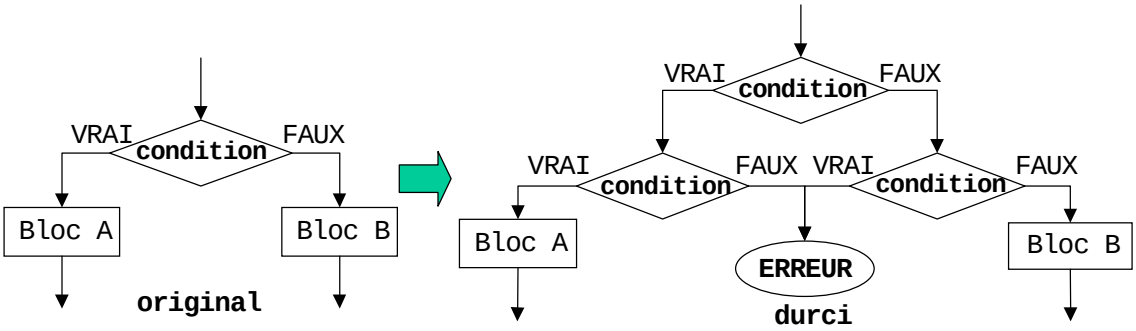


Figure 2.13 : Transformation visant la détection des erreurs affectant les instructions conditionnelles de contrôle

D.2. Erreurs affectant les instructions inconditionnelles de contrôle

L'ensemble d'instructions inconditionnelles de contrôle dans un programme de haut niveau que nous proposons de durcir s'adresse uniquement aux appels et retours de fonctions d'un programme. Dans une description de haut niveau d'un programme, les instructions de branchement direct sont utilisées rarement. Souvent, les appels et le retour des fonctions sont affectés par l'apparition d'un bit-flip par la modification de l'adresse de branchement, du registre de la pile où la pile elle même.

Un tel exemple est illustré dans la Figure 2.14. En cet exemple, dans le corps d'instruction de la *fonction appelante*, nous avons un branchement inconditionnel (appel de fonction) vers la *fonction appelée*. Une erreur est produite et affecte l'instruction d'appel de la *fonction appelée*. Suite à cette modification, quelques instructions appartenants à la fonction appelée ne sont pas exécutées menant à un calcul incorrect des résultats finaux ou à une perte de séquencment.

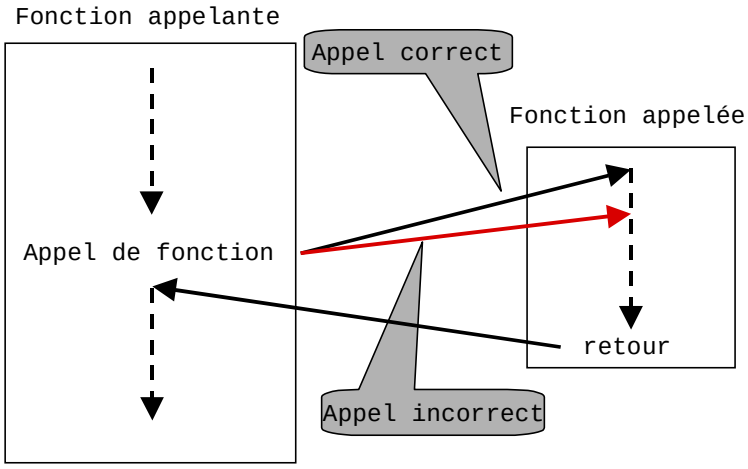


Figure 2.14 : Exemple d'un bit-flip affectant l'adresse de retour d'une fonction

Pour détecter ce type d'erreur, un indicateur ***carf*** (*contrôle d'appels et retours de fonction*) est défini, afin d'associer une identité à chaque fonction du programme. Au début de chaque corps de fonction une valeur est assignée au ***carf***. Avant chaque appel de fonction est introduite une vérification de la valeur de ***carf*** de la fonction appelante. Après l'instruction d'appel, une vérification de la valeur de ***carf*** de la fonction appelée est introduite afin de détecter les erreurs affectant ce type d'instructions.

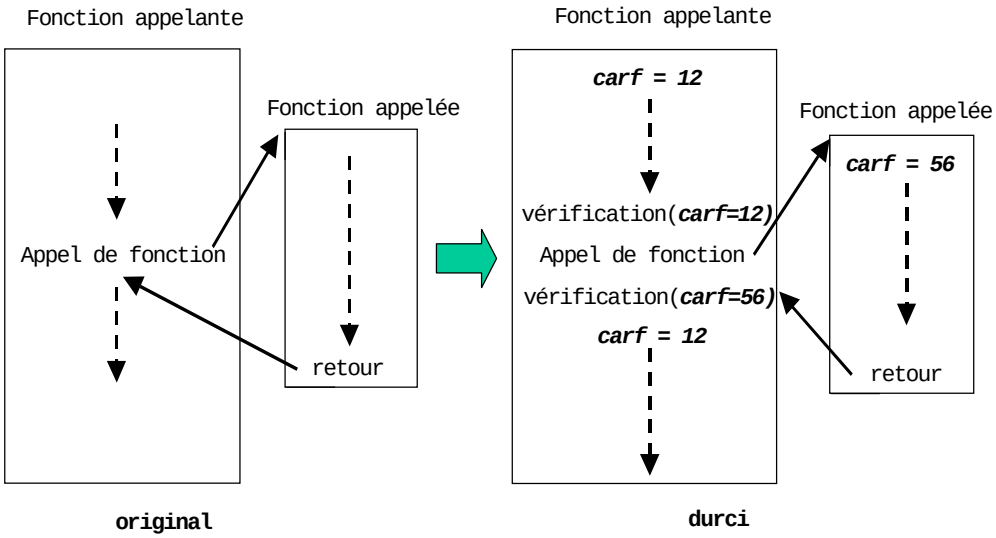


Figure 2.15 : Transformation visant des erreurs affectant des appels et retours des fonctions

Afin de détecter les erreurs qui affectent ce type d'instructions nous ajoutons les règles suivantes :

- Règle #11 : une valeur ***kj*** est associée à chaque fonction ***j*** dans le programme,
- Règle #12 : une variable globale ***carf*** est définie pour le contrôle des appels et des retours de fonction ; une instruction assignant à ***carf*** la valeur du ***kj*** est inclue au début de chaque fonction ***j***,
- Règle #13 : avant chaque appel de fonction une vérification de la valeur du ***carf*** de la fonction appelante est introduite, après chaque appel de fonction une vérification de la valeur du ***carf*** de la fonction appelée est introduite.

III. Evaluation de la méthode de détection

Dans le paragraphe précédent, nous avons présenté un ensemble de règles introduisant des mécanismes de détection d'erreurs de type basculement de bit. Dans cette section, nous proposons d'évaluer les performances du programme résultant par l'application séparée de chaque type de durcissement proposés dans le paragraphe précèdent.

Deux paramètres essentiels pour mesurer le coût d'un programme sont définis :

- Le temps d'exécution : la durée d'exécution du programme, exprimée en nombre de cycles de processeur ou en secondes,

§ L'espace mémoire requis : le nombre des mots nécessaire pour stocker le code du programme.

A.1. Le temps d'exécution du programme

L'exécution de chaque instruction du programme demande une durée de temps constante (deux instructions différentes peuvent prendre des temps différents). Chaque exécution de la $i^{ème}$ instruction prendra le temps t_i, où t_i est une constante.

Le temps d'exécution $\boldsymbol{T_p}$ du programme ***P*** peut être exprimé comme étant la somme des temps d'exécution t_i de chaque instruction exécutée. Une instruction qui s'exécute en un temps t_i et qui est exécutée k fois interviendra donc pour $k \times t_i$ dans le temps d'exécution total.

En conséquence, pour un programme ***P*** avec ***n*** instructions, le temps total d'exécution $\boldsymbol{T_P}$ est donné par la formule :

$$T_P = \sum_i^n t_i \times k_i \qquad (2.1)$$

où k_i représente le nombre de fois que l'instruction i a été exécutée.

Nous rappelons qu'un programme ***P*** est structuré en *blocs élémentaires* et *instructions de contrôle.* Le temps d'exécution de chaque bloc élémentaire représente la somme de la durée d'exécution de toutes les instructions composant le bloc. Nous avons noté T_l le temps d'exécution pour un bloc élémentaire et T_c le temps d'exécution pour une instruction de contrôle.

Une autre façon de mesurer le temps total d'exécution pour un programme ***P*** est de calculer la somme de tous les temps d'exécution nécessaires pour chaque bloc élémentaire T_l et la somme de tous les temps d'exécution nécessaires pour chaque instruction de contrôle T_c. Donc pour un programme qui a ***n*** blocs élémentaires et ***m*** instructions de contrôle, le temps total d'exécution sera donc :

$$T_P = \sum_i^n k_i \times T_{li} + \sum_i^m k_i \times T_{ci} \qquad (2.2)$$

A.2. L'espace mémoire d'un programme

L'espace mémoire (***M***) d'un programme ***P*** est la quantité de mémoire (mesurée en nombre de mots) nécessaire pour son exécution.

La place mémoire prise par un programme est une contrainte de moins en moins primordiale vu les possibilités techniques actuelles de stockage. Dans cette étude sur les performances du

programme résultant suite à l'application des règles proposées, nous nous focalisons donc sur le temps d'exécution.

Dans l'analyse suivante, nous définissons :

- **φ -** l'ensemble des groupes de règles définis correspondant à chaque groupe de règles,
- $\Psi|\varphi$ - *la fonction de durcissement* qui transforme un programme ***P*** dans un autre programme (*durci*) gardant la même fonctionnalité,
- **θ -** le facteur de pénalité, le rapport entre le temps d'exécution du programme résultant (suite à l'application du groupe de règles) et le programme original.

En appliquant la transformation Ψ sur un programme ***P***, nous obtenons donc un autre programme ***P'***. La fonction exécutée par le programme original et celle exécutée par le programme résultant est la même.

$$P'_{<T',M'>} = \Psi(P_{<T,M>})|\varphi \qquad \text{où} \qquad \varphi = \{dd \cup bd \cup fge\} \qquad (2.3)$$

B. Application de la transformation $\Psi|dd$

L'application de la transformation $\Psi|dd$ sur un programme interviendra seulement sur les opérations de lecture/écriture pour les variables définies dans le programme.

Nous rappelons que ce groupe de règles duplique toutes les variables et les instructions où les variables interviennent. Ensuite nous effectuons une vérification de la cohérence des valeurs des deux variables (la variable originale et sa réplique) pour toutes les variables qui ne sont pas des opérandes sources dans une opération sur d'autres variables (c.a.d. des variables qui ne participent pas au calcul d'une autre variable).

Nous représentons le temps d'exécution (noté T_o) du programme original comme la somme de la durée d'exécution de toutes les instructions de lecture/écriture sur les variables et un temps T_A correspondant aux durées d'exécution des autres instructions (par exemple les appels de fonctions et les branchement inconditionnels).

$$T_o = k_1 \cdot t_1 + k_2 \cdot t_2 + \ldots + k_n \cdot t_n + T_A \qquad \Rightarrow \qquad T_o = \sum_i^n k_i \cdot t_i + T_A \quad (2.4)$$

En général, dans un programme la contribution de T_A est négligeable par rapport au temps total d'exécution du programme. Dans la suite, nous négligerons donc le facteur T_A.

$$T_o = \sum_i^n k_i \cdot t_i \qquad (2.5)$$

En conformité avec les transformations introduites par ce groupe de règles (*Duplication des Données*), le temps d'exécution pour le programme durci est :

$$T_{dd} = k_1 \cdot t_1 + k'_1 \cdot t'_1 + k_1 \cdot t_{cv1} + k_2 \cdot t_2 + k'_2 \cdot t'_2 + k_2 \cdot t_{cv2} + \ldots + k_n \cdot t_n + k'_n \cdot t'_n + k_n \cdot t_{cvn} \quad (2.6)$$

Où :

- $k'_i \cdot t'_i$ est le temps introduit comme conséquence à la répétition de la $i^{ème}$ instruction
- t_{cvi} est le temps de la comparaison des deux valeurs de la variable originale et de la variable dupliquée

Le facteur $k_i \cdot t_i$ est étant égal à $k'_i \cdot t'_i$, l'équation 2.6 peut donc être écrite sous la forme :

$$T_{dd} = 2 \cdot k_1 \cdot t_1 + k_1 \cdot t_{cv1} + 2 \cdot k_2 \cdot t_2 + k_2 \cdot t_{cv2} + \ldots + 2 \cdot k_n \cdot t_n + k_n \cdot t_{cvn} \Rightarrow$$

$$T_{dd} = \sum_i^n 2 \cdot k_i \cdot t_i + \sum_i^n k_i \cdot t_{cvi} \quad \Rightarrow \quad T_{dd} = 2 \cdot T_o + \sum_i^n k_i \cdot t_{cvi} \qquad (2.7)$$

Le facteur de pénalité en temps d'exécution pour le programme durci est alors :

$$\frac{T_{dd}}{T_o} = 2 + \frac{\sum_i^n k_i \cdot t_{cvi}}{T_o} \quad (2.8) \quad \Rightarrow \quad \frac{T_{dd}}{T_o} = 2 + \frac{1}{\frac{T_o}{\sum_i^n k_i \cdot t_{cvi}}} \quad \Rightarrow$$

$$\frac{T_{dd}}{T_o} = 2 + \frac{1}{\mu} \qquad (2.9) \qquad \text{où} \qquad \mu = \frac{T_o}{\sum_i^n k_i \cdot t_{cvi}} \quad (2.10)$$

t_{cvi} est le temps pris pour comparer les valeurs des deux emplacements de la mémoire. C'est donc une constante qui ne dépend pas du type d'opération sur les variables. Dans l'équation 2.10, le facteur $k_i \cdot t_{cvi}$ est égal à zéro pour toutes les opérations affectant des variables intermédiaires dans des expressions arithmétiques.

Dans l'exemple donné en Figure 2.16, `i`, `U`, `C` et `y` sont des variables utilisées pour le calcul de `Eb`, elles sont donc, des variables intermédiaires. La variable `Eb` garde le résultat final et elle n'intervient pas dans le calcul d'une autre variable pendant toute l'exécution du programme, donc la vérification des valeurs des deux variables (la variable originale et sa copie) sera introduite uniquement pour cette variable.

```
        /* déclaration des variables */
         int i;
         float U[100],C[100]
         float y,Eb;

         . . .
t1       i = 0;                  // opération intermédiaire
t2       while(i < 2)
         {
t3           y = y + U[i]*C[i];  // opération intermédiaire
t4           i ++;               // opération intermédiaire
         }

t5       Eb = y*(1 - y*y);       // dans Eb le résultat
                                 // final est gardé
         . . .
```

Figure 2.16 : Exemple de programme

Le facteur de pénalité introduit par le groupe des règles *Duplication des Données* est calculé en utilisant l'équation 2.8.

$$\frac{T_{dd}}{T_o} = 2 + \frac{\sum_{i}^{n} k_i \cdot t_{cvi}}{T_o} \Rightarrow \frac{T_{dd}}{T_o} = 2 + \frac{t_{cvEb}}{t1 + 2 \cdot t2 + 2 \cdot t3 + 2 \cdot t4 + t5}$$

Tableau 2.3 : Facteur de pénalité pour divers processeurs

Processeur	T_o					t_{cvEb}	$\frac{T_{dd}}{T_o}$
	t1	t2	t3	t4	t5		
Intel 80c51	10	28	90	12	32	28	2.09
DSP 32C	12	42	115	37	57	50	2.10

Comme le Tableau 2.3 le montre, le facteur de pénalité introduit par ce groupe de règles peut être écrit comme suit :

$$\frac{T_{dd}}{T_o} = 2 + \varepsilon_{dd} \qquad (2.11) \qquad \text{où } \varepsilon_{dd} \in (0,1)$$

C. Application de la transformation *Ψ|fge*

Afin de contrôler le flot global d'exécution, la transformation ***Ψ|fge*** appliquée sur un programme ***P*** décompose en blocs élémentaires le programme et ajoute deux instructions à chaque

bloc élémentaire. Le temps supplémentaire introduit pour chaque bloc élémentaire est noté T_{cfge}. Ce temps est une constante qui dépend seulement du type de processeur utilisé. T_{cfge} a donc la même valeur pour tous les blocs élémentaires du programme.

En utilisant l'équation 2.2 qui donne le temps d'exécution pour un programme, nous représentons le temps d'exécution pour le programme résultant suite à l'application de $\Psi|fge$ comme :

$$T_{fge} = \sum_i^n k_i \cdot (T_{li} + T_{cfge}) + \sum_j^m k_j \cdot T_{cj} \qquad \Rightarrow \qquad T_{gef} = \sum_i^n k_i \cdot T_{li} + \sum_i^n ki \cdot T_{cfge} + \sum_j^m k_j \cdot T_{cj}$$

$$\Rightarrow$$

$$T_{fge} = T_o + \sum_i^n k_i \cdot T_{cfge} \qquad (2.12)$$

Le facteur de pénalité introduit par la transformation $\Psi|gef$ est :

$$\frac{T_{fge}}{T_o} = 1 + \frac{\sum_i^n k_i \cdot T_{cfge}}{T_o} \quad (2.13) \quad \Rightarrow \quad \frac{T_{fge}}{T_o} = 1 + \frac{\sum_i^n k_i \cdot T_{cfge}}{\sum_i^n k_i \cdot T_{li} + \sum_j^m k_j \cdot T_{cj}} \Rightarrow$$

$$\frac{T_{fge}}{T_o} = 1 + \frac{1}{\dfrac{\sum_i^n k_i \cdot T_{li} + \sum_j^m k_j \cdot T_{cj}}{\sum_i^n k_i \cdot T_{cfge}}} \quad (2.14) \quad \Rightarrow \quad \frac{T_{fge}}{T_o} = 1 + \frac{1}{T} \quad (2.15)$$

où $$T = \frac{\sum_i^n k_i \cdot T_{li}}{\sum_i^n k_i \cdot T_{cfge}} + \frac{\sum_j^m k_i \cdot T_{cj}}{\sum_i^n k_i \cdot T_{cfge}} \quad \Rightarrow \quad T = \frac{\sum_i^n k_i \cdot T_{li}}{\sum_i^n k_i \cdot T_{cfge}} + \delta \qquad (2.16)$$

Dans la Figure 2.17 nous représenterons la distribution statistique des instructions formant un bloc élémentaire en fonction du nombre des cycles d'horloge du processeur. Le temps d'exécution (T_{li}) nécessaire à un bloc élémentaire composé de plus de deux instructions est sensiblement plus grand que le temps de contrôle du flot global d'exécution T_{cfge}. En conclusion le facteur T défini dans l'équation 16 est beaucoup plus grand que 1 ($T >> 1$).

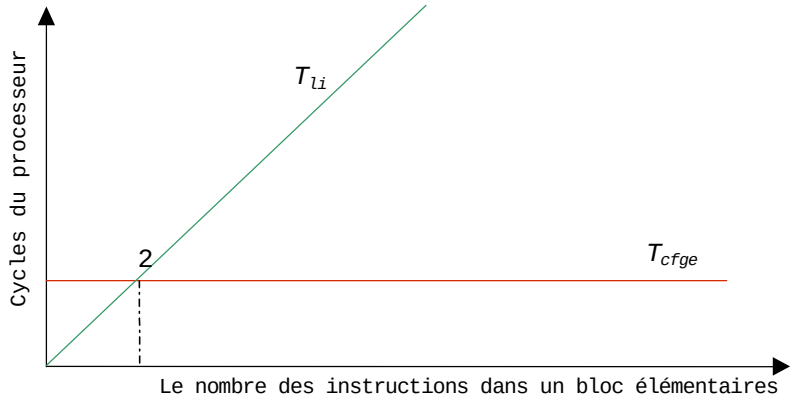


Figure 2.17 : Distribution statistique des instructions formant un bloc élémentaire en fonction du nombre des cycles d'horloge du processeur

Pour une estimation qualitative, nous calculons le facteur de pénalité introduit par ce groupe de règles pour l'exemple donné dans la Figure 2.18.

```
        /* déclaration des variables */
         int i;
         float U[100],C[100]
         float y,Eb;

         . . .
t1       i = 0;                  // opération intermédiaire
t2       while(i < 2)
         {
t3           y = y + U[i]*C[i];  // opération intermédiaire
t4           i ++;               // opération intermédiaire
         }

t5       Eb = y*(1 - y*y);       // dans Eb le résultat
                                 // final est gardé
         . . .
```

Figure 2.18 : Exemple de programme

Dans cet exemple nous avons identifié quatre blocs élémentaires de taille maximale illustrés dans le Tableau 2.4. L'analyse de chaque bloc de taille maximale montre que trois blocs sont composés d'une seule instruction (bloc 1,2 et 4). Le troisième bloc est composé d'une instruction complexe (le calcul de la variable *y*) et une instruction simple (incrémentation de la valeur de la variable *i*). En conséquence, nous décidons que les blocs de taille maximale sont indivisibles, donc ils sont élémentaires.

Tableau 2.4 : composition de blocs élémentaires

No	Instructions composant le bloc	Temps d'exécution
1	i = 0	t1
2	i < 2	t2
3	y = y + u[i]*c[i] i ++	t3+t4
4	Eb = y(1-y*y)	t5

Le facteur de pénalité introduit par le groupe des règles *FGE* est calculé en utilisant l'équation 2.13.

$$\frac{T_{fge}}{T_o} = 1 + \frac{\sum_i^n k_i \cdot T_{cfge}}{T_o} \Rightarrow \frac{T_{fge}}{T_o} = 1 + \frac{T_{cfge} \cdot (1+2+2+1)}{t1 + 2 \cdot t2 + 2 \cdot t3 + 2 \cdot t4 + t5}$$

Tableau 2.5 : Facteur de pénalité pour divers processeurs

Processeur	To					T_{cfge}	$\frac{T_{fge}}{T_o}$
	t1	t2	t3	t4	t5		
Intel 80c51	10	28	90	12	32	34	1.11
DSP 32C	12	42	115	37	57	62	1.13

Le facteur de pénalité introduit par ce groupe de règles est :

$$\frac{T_{fge}}{T_o} = 1 + \varepsilon_{fge} \qquad \text{où} \qquad \varepsilon_{fge} \in (0,1) \qquad (2.17)$$

D. Application de la transformation *Ψ|bd*

Cette transformation interviendra seulement sur les instructions de contrôle d'un programme. La transformation *Ψ|bd* introduit des instructions supplémentaires au niveau de chaque instruction de contrôle. Pour les instructions conditionnelles de contrôle, la condition est réévaluée, alors que pour chaque instruction inconditionnelle, plusieurs instructions sont ajoutées. Le temps d'exécution pour le programme résultant suite à l'application de cette transformation est évalué avec l'équation :

$$T_o = \sum_i^n k_i \cdot T_{li} + \sum_j^m k_j \cdot T_{ccj} + \sum_j^k k_j \cdot T_{cij} \qquad (2.18) \quad \text{où :}$$

§ $\sum_i^n T_{li} \cdot k_i$: représente le temps d'exécution nécessaire pour tous les blocs élémentaires,

§ $\sum_{j}^{m} T_{ccj} \cdot k_j$: représente le temps d'exécution pour toutes les instructions de contrôles conditionnelles,

§ $\sum_{j}^{k} T_{cij} \cdot k_j$: représente le temps d'exécution nécessaire pour toutes les instructions de contrôles inconditionnelles.

En utilisant l'équation 2.18, nous représentons le temps d'exécution pour le programme résultant suite à l'application de $\Psi|bd$ comme :

$$T_{bd} = \sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} 2 \cdot k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot (T_{cij} + T_{carf}) \Rightarrow$$

$$T_{bd} = \sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{cij} + \sum_{j}^{k} k_j \cdot T_{carf} \Rightarrow$$

$$T_{bd} = T_o + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{carf} \qquad (2.19)$$

Le facteur de pénalité est introduit par la transformation $\Psi|bd$ est :

$$\frac{T_{bd}}{T_o} = 1 + \frac{\sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{carf}}{T_o} \qquad (2.20)$$

$$\frac{T_{bd}}{T_o} = 1 + \frac{\sum_{j}^{m} k_j \cdot T_{ccj}}{\sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{cij}} + \frac{\sum_{j}^{k} k_j \cdot T_{carf}}{\sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{cij}} \Rightarrow$$

$$\frac{T_{bd}}{T_o} = 1 + \frac{1}{\frac{\sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{cij}}{\sum_{j}^{m} k_j \cdot T_{ccj}}} + \frac{1}{\frac{\sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{cij}}{\sum_{j}^{k} k_j \cdot T_{carf}}}$$

$$\frac{T_{bd}}{T_o} = 1 + \frac{1}{Bc} + \frac{1}{Bi} \qquad (2.21) \qquad \text{où :}$$

§ *Bc* : $$\frac{\sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{cij}}{\sum_{j}^{m} k_j \cdot T_{ccj}} = \frac{T_o}{\sum_{j}^{m} k_j \cdot T_{ccj}} \qquad (2.22)$$

§ *Bi* : $$\frac{\sum_{i}^{n} k_i \cdot T_{li} + \sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{cij}}{\sum_{j}^{k} k_j \cdot T_{carf}} = \frac{T_o}{\sum_{j}^{k} k_j \cdot T_{carf}} \qquad (2.23)$$

Dans l'exemple donné en Figure 2.19, nous avons deux instructions de contrôle. Une instruction de contrôle conditionnelle (`while(i < 2)`) et une instruction de contrôle inconditionnelle (l'appel de la fonction : `calculate`). Du côte gauche dans cette figure, sont représentés les temps d'exécution de chaque instruction du programme.

```
        calculate()
        {
             /* déclaration des variables */
             int i;
             float U[100],C[100]
             float y,Eb;

             . . .
t1           i = 0;                  // opération intermédiaire
t2           while(i < 2)
             {
t3               y = y + U[i]*C[i];  // opération intermédiaire
t4               i ++;               // opération intermédiaire
             }

t5           Eb = y*(1 - y*y);       // dans Eb le résultat
                                     // final est gardé
             . . .
        }
        main()
        {
t6           calculate();
        }
```

Figure 2.19 : Exemple de programme

Le facteur de pénalité introduit par le groupe des règles *Duplication de Branchement* est calculé en utilisant l'équation 2.20.

$$\frac{T_{bd}}{T_o} = 1 + \frac{\sum_{j}^{m} k_j \cdot T_{ccj} + \sum_{j}^{k} k_j \cdot T_{carf}}{T_o} \Rightarrow \frac{T_{bd}}{T_o} = 1 + \frac{2 \cdot t2 + T_{carf}}{t1 + 2 \cdot t2 + 2 \cdot t3 + 2 \cdot t4 + t5 + t6}$$

Tableau 2.6 : Facteur de pénalité pour divers processeurs

Processeur	To						T_{carf}	2*t2	$\frac{T_{bd}}{T_o}$
	t1	t2	t3	t4	t5	t6			
Intel 80c51	10	28	90	12	32	4	86	56	1.464
DSP 32C	12	42	115	37	57	18	136	84	1.463

Le facteur de pénalité introduit par ce groupe de règles est :

$$\frac{T_{bd}}{T_o} = 1 + \varepsilon_{bd} \qquad \text{où} \qquad \varepsilon_{bd} \in (0,1) \qquad (2.24)$$

IV. Génération automatique des programmes durcis

A. Flot de génération des programmes durcis

Dans cette section nous présentons le flot que nous proposons pour la génération automatique des programmes durcis. Ce flot prend en entrée le code source d'un programme décrit dans un langage de haut niveau (le langage C) et il produit en sortie le code source du programme durci en fonction du type de durcissement désiré par le concepteur. La Figure 2.20 illustre les principales étapes du flot de génération automatique des programmes durcis.

Le flot de génération des programmes durcis est composé principalement de deux étapes. Une première étape est l'analyse du programme à durcir et la deuxième étape est l'insertion du code durci approprié à chaque groupe de règles.

Le but de l'étape d'analyse du programme d'entrée est de fournir des informations concernant l'architecture du programme donné. Ces informations sont nécessaires pour l'insertion des règles de durcissement. Dans cette étape, le programme à durcir est analysé en fonction du type de durcissement (options désiré par l'utilisateur). Le type de durcissent est en fait donné par les groupes de règles présentés antérieurement : nous avons ainsi le groupe de *Duplication de Données*, le groupe de *Flot Global d'Exécution* et le groupe de *Duplication de Branchement.*

Cette première partie est donc composée de trois blocs fonctionnels :

- Identification des données du programme
- Identification des instructions de contrôle
- Identification des blocs élémentaires du programme de taille maximale

La deuxième partie de ce flot est en charge de la génération du code associé à chaque groupe de règles proposé dans ce chapitre. L'utilisateur a la possibilité de choisir deux options d'optimisation :

- une pour les relations d'interdépendance entre les variables du programme donné
- une option concernant le facteur de granularité pour chaque bloc élémentaire dans le code du programme d'entrée

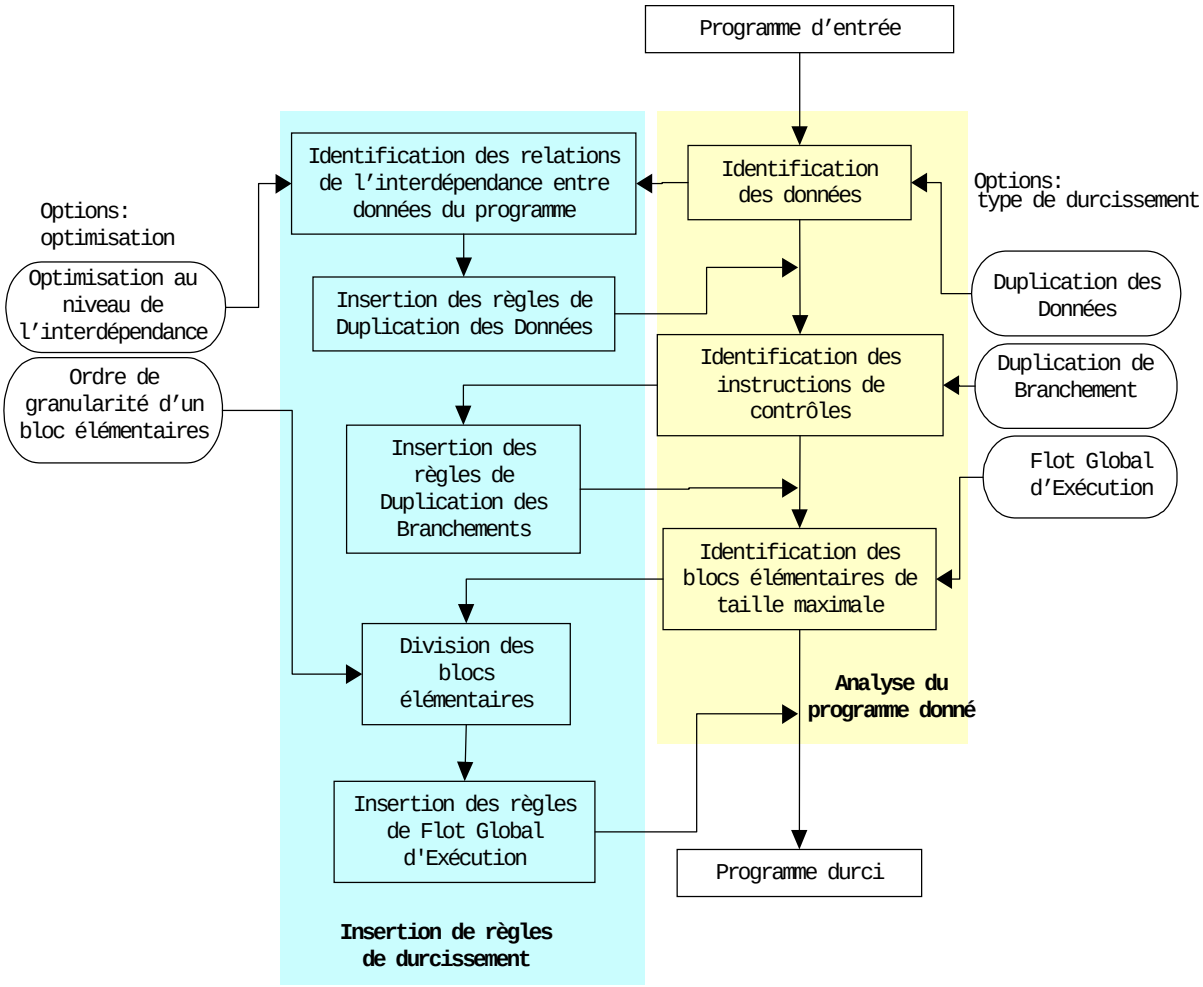


Figure 2.20 : Flot de génération des programmes durcis

Dans le cas de durcissement au niveau de données, nous chercherons les relations d'interdépendance entre les variables du programme et nous classifierons les variables du programme en *variables intermédiaires* et *variables finales*. Cela pour permettre l'application de la règle #4 appartenant au groupe *Duplication de Données*. Par l'intermédiaire d'une option d'optimisation, l'utilisateur peut décider d'appliquer la règle #4 pour toutes les variables du

programme (même pour les *variables intermédiaires*). Dans ce cas le facteur de pénalité est augmenté et il ne peut pas être estimé avec l'équation 2.9. Par défaut cette option est désactivée.

Une deuxième option d'optimisation est proposée afin de définir le facteur de granularité d'un bloc élémentaire i.e. combien d'instructions complexes peuvent composer un bloc élémentaire. Si l'utilisateur n'active pas cette option, le nombre d'instructions complexes composant un bloc élémentaire est fixé à deux.

B. Le translateur C2C

Afin de mettre en œuvre ce flot, nous avons développé un outil logiciel appelé *le translateur C2C*. Cet outil permet la génération automatique de programmes durcis à partir d'un programme donné en tenant compte des options de durcissement proposées. Les types de durcissement peuvent être appliqués indépendamment.

Les détails d'utilisation et différents exemples de génération de programmes sont données dans l'annexe A.

V. Conclusions

Dans ce chapitre, nous avons proposé un nouvel ensemble de règles pour la détection d'erreurs ainsi qu'une méthodologie pour la génération automatique de programmes durcis écrits en langage de haut niveau, avec la possibilité de détecter des erreurs affectant les données et le code. L'ensemble de règles proposées pour la détection d'erreurs est automatiquement mis en application à l'aide d'un outil de génération des programmes durcis comme une phase de pré-compilation, complètement transparente au programmeur. Ceci signifie la réduction du coût de développement des programmes durcis, et l'augmentation de la confiance au niveau du durcissement obtenu.

Nous avons également défini un cadre conceptuel pour l'estimation de l'impact de règles proposées, pour un programme donné.

D'autre part, l'application de la méthode de détection proposée augmente le nombre d'instructions d'un facteur moyen de 4, et un ralentissement d'exécution du programme d'un facteur **inférieur à 3**. Cependant, dans la plupart des systèmes qui nécessitent une sûreté de fonctionnement, l'espace mémoire et le ralentissement du programme ne sont pas des paramètres cruciaux pour le comportement correct du système global.

Chapitre 3

Validation de la méthode – Intel 80C51

- Introduction
- Terminologie
  - L'efficacité de détection
  - Taux d'erreur
  - Flux de particules
  - Fluence
  - Transfert Linéaire d'Energie
- Description du microcontrôleur Intel 80C51
- Injection des fautes
  - Les caractéristiques principales des programmes utilisés
  - Stratégie utilisée pour l'injection de fautes
  - Analyse de résultats obtenus
  - L'efficacité de chaque groupe de règles
  - Fautes échappantes au mécanisme de détection
- Essais sous radiations
- Conclusions

I. Introduction

Nous avons choisi le microcontrôleur Intel 80C51 comme véhicule préliminaire d'expérimentation pour valider l'efficacité de la méthode de détection d'erreurs en termes de capacité de détection d'erreurs de type basculement des bits ainsi que pour mettre en évidence ses limitations éventuelles et explorer les possibilités de généralisation et d'automatisation de l'approche proposée.

Après une brève description de l'architecture du microcontrôleur Intel 80C51, ce chapitre sera consacré à l'analyse des résultats obtenus suite à des expériences d'injection des fautes et des essais de radiations.

Les expérimentations concernant l'injection des fautes et les essais de radiations ont consisté en :

- Le test d'un ensemble de programmes que nous appelons par la suite les programmes originaux ;
- La génération des programmes durcis (à partir des programmes originaux suite à l'utilisation du translateur C2C) ;
- L'évaluation du comportement des programmes durcis et des programmes originaux face aux effets d'erreur de type basculement des bits.

Les effets des fautes injectées ont été classifiés dans les catégories suivantes :

- *Sans Effet* : la faute n'affecte pas le comportement du programme ;
- *Détection Logicielle* : le mécanisme de détection ajouté au programme original détecte la faute et donne une réponse indiquant le type d'erreur ;
- *Séquence Perdue* : la faute n'a pas été détectée et le programme testé n'atteint pas la fin d'exécution ;
- *Réponse Fausse* : la faute n'a pas été détectée et le résultat final du programme est différent de celui prévu.

Des essais réalisés avec un choix pseudo-aléatoire des paramètres d'upset (cible, instant d'occurrence) permettront l'estimation de l'efficacité de détection d'upsets pour les versions des programmes durcis.

II. Terminologie

Quelques définitions sont nécessaires avant la présentation des résultats expérimentaux.

A.1. L'efficacité de détection

L'*efficacité de détection* (ζ) du programme durci est le nombre de détections divisé par le nombre total de fautes affectant le comportement du programme testé. Elle est défini comme :

$$\xi = \frac{D}{D + E + P} \qquad (3.1)$$

où :

- D : le nombre de fautes détectées par le mécanisme de détection
- E : le nombre d'erreurs qui échappent au mécanisme de détection
- P : le nombre de pertes de séquencement

A.2. Taux d'erreur

Le *taux d'erreur* (τ) pour le programme testé est défini comme étant le nombre de toutes les fautes changeant le comportement du programme testé divisé par le nombre total de fautes injectées.

$$\tau = \frac{E + P}{\#\ fautes\ \ injectées} \qquad (3.2)$$

où :

- E : le nombre d'erreurs qui échappent au mécanisme de détection
- P : le nombre de pertes de séquencement

A.3. Flux de particules

Le *flux de particules* est défini comme le nombre de particules arrivant par unité de surface et de temps; il est donné en particules/cm^2.s.

$$\Phi = \frac{N}{S \cdot \Delta T} \left[\frac{\#\ particules}{cm^2 \cdot \sec} \right] \qquad (3.3)$$

A.4. Fluence

L'intégration dans le temps du flux de particules donne *la densité de particules* ou *fluence* (*F*) exprimée en particules/cm^2.

$$F = \frac{N}{S}\left[\frac{\#\ particules}{cm^2}\right] \qquad (3.4)$$

A.5. Transfert Linéaire d'Energie

Une particule interagissant avec la matière lui transfère son énergie. La quantité d'énergie déposée par la particule par unité de longueur de trajectoire est appelée pouvoir d'arrêt ou LET (*Linear Energy Transfer*). Elle est habituellement donnée en Me·V·cm^2/mg [37].

III. Description du microcontrôleur Intel 80C51

Les principales caractéristiques de la famille de contrôleurs MCS 51 sont les suivantes [38] :

- CPU de 8 bits
- Processeur booléen permettant le calcul sur un bit
- Mémoire interne de 128 octets (210 locations de mémoire interne adressable par bit)
- 64 k octets d'espace mémoire programme
- 64 k octets d'espace mémoire de données
- Communication sérielle

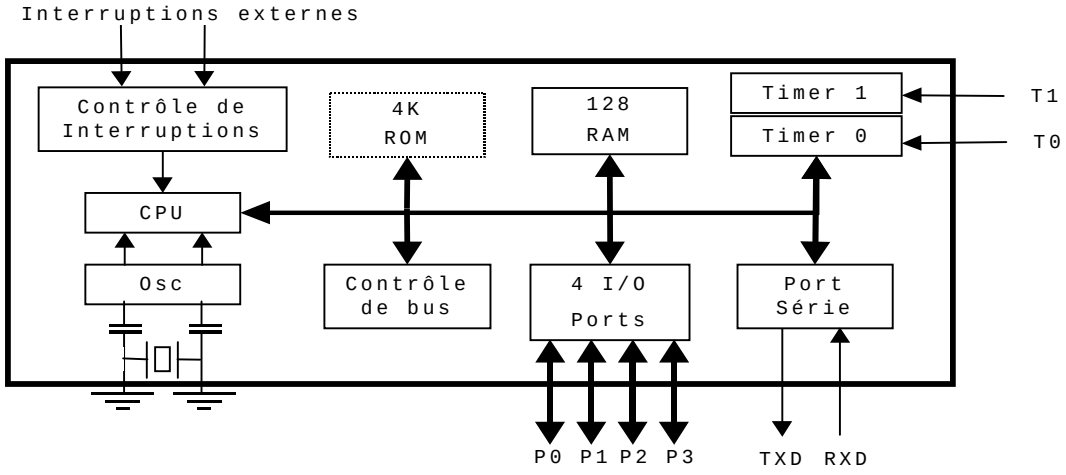


Figure 3.1 : Structure interne du microcontrôleur 80C51

Dans la Figure 3.1 sont donnés les principaux blocs de l'architecture interne du 80C51. Les zones mémoires directement accessibles par le jeu d'instructions sont les ports d'entrée/sortie, les accumulateurs A et B, les registres spéciaux (SFR), les registres généraux (R0-R7), le registre compteur de programme et la mémoire RAM interne (128 octets). Les registres généraux sont ciblés dans la mémoire interne dans la quelle ils sont accessibles par adressage direct.

Les zones mémoires qui ne pourront pas être perturbées par le jeu d'instructions, incluent les registres d'entrée de l'U.A.L., les latches, des registres d'adresse, etc. Une estimation de zones mémoires inaccessibles par le jeu d'instruction a été estimée dans [39] à 7% de toute la mémoire interne du microcontrôleur Intel 80C51. Dans la Figure 3.2 est indiquée la répartition de zones mémoires adressables par le jeu d'instructions.

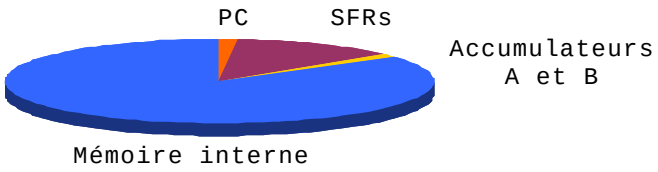


Figure 3.2 : Répartition des zones de mémoires adressable par le jeu d'instructions

IV. Injection des fautes

A. Les caractéristiques principales des programmes utilisés

En utilisant le translateur C2C, les règles proposées dans le chapitre II ont été appliqués sur trois applications:

- La multiplication de deux matrices de taille 8x8 ;
- Un programme de tri utilisant l'algorithme "bubble sort" ;
- Le protocole de communication "I2C" adapté pour 16 périphériques.

Dans le Tableau 3.1 nous illustrons le nombre de cycles et la taille du chacun des programmes utilisés.

Tableau 3.1 : Principales caractéristique des programmes originaux et durcis testés

Programme	Cycles exécutes	Taille du Code
Protocole I2C Original	11141	176
Protocole I2C Durci	35094 (x3.15)	880 (x5)
Multiplication de deux matrices Original	10110	160
Multiplication de deux matrices Durci	25275 (x2.5)	640 (x4)
Tri *Bubble Sort* Original	15096	204
Tri Bubble Sort Durci	40457 (x2.68)	918 (x4.5)

B. Stratégie utilisée pour l'injection de fautes

Des essais d'injection de fautes et de tests sous radiation ont été effectués en utilisant un système de test appelé THESIC [40] composé d'une carte mère construite autour d'un microcontrôleur Intel 80C51 et d'une carte fille mettant en application une architecture appropriée autour du processeur à tester. Les expériences présentées dans la suite ont été réalisées en utilisant une carte fille 80C51.

Dans la suite le mécanisme d'injection de fautes est brièvement décrit. Après que la carte mère soit activée une procédure est appelée pour initialiser toutes les ressources nécessaires pour l'injection de fautes. Elles sont : ***l'adresse*** (le ***registre***) à perturbé, la ***position du bit*** dans l'adresse cible et ***l'instant d'injection***. La prochaine étape exécutée par la carte mère est d'envoyer le signal ***DEMARRER*** à la carte fille et d'entrer dans une boucle d'attente jusqu'à ce que la condition de ***l'instant d'injection*** soit satisfaite. Après ceci, le signal d'***INJECTION*** est envoyé à la carte fille. En attendant la carte fille reçoit l'état de début et commence l'exécution du programme cible.

Quand le signal d'***INJECTION*** est reçu, le programme en cours d'exécution au sein de la carte fille est suspendu et le compteur de programme est placé au début de la routine d'injection. En fait, la carte fille s'auto-injecte des fautes, le rôle de la carte mère est limité à décider quand et quelle cible sera corrompue et à l'analyse des résultats envoyés par la carte fille par l'intermédiaire d'une mémoire partagée. Une fois que la routine d'injection de faute a été exécutée la tâche suspendue est reprise. Enfin la carte mère analyse les résultats après l'injection de la faute. Dans la Figure 3.3 est illustré le mécanisme d'injection de fautes adopté.

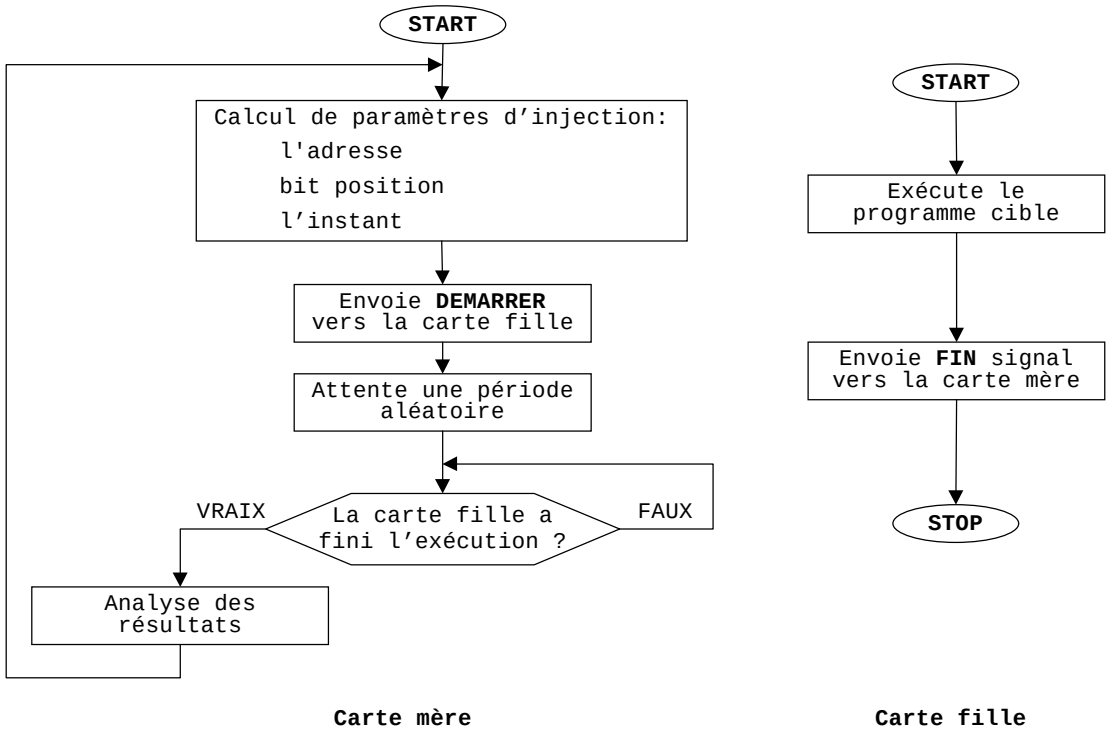


Figure 3.3 : Activité du THESIC pour une séance d'injection des fautes

Les fautes ont été injectées exclusivement dans des registres internes du microcontrôleur de la carte fille. L'occurrence aléatoire d'injection des fautes a été pris en considération au moyen d'un *LFSR*[1] [41] mis en application par une implantation logicielle. Le polynôme caractéristique du LFSR a été choisi pour garantir une bonne distribution pour les fautes injectées.

C. Analyse de résultats obtenus

Chaque séance d'injection de fautes effectuée a consisté en l'injection d'un seul bit-flip dans les registres internes du microcontrôleur 80C51 pendant chaque exécution des programmes sous étude. Comme le temps d'exécution pour les programmes durcis est trois fois plus grand que celui de programmes originaux, nous avons injecté trois fois plus de fautes dans le cas des programmes durcis. Ainsi la probabilité d'une faute affectant le microcontrôleur est augmentée proportionnellement.

C'est important de noter que les programmes pour l'essai d'injection des fautes utilisent seulement 15% de la mémoire interne du microcontrôleur 80C51 (dans la quelle les registres internes du microcontrôleur sont ciblés). De plus, plusieurs registres ont été utilisés rarement, ainsi

le nombre très grand de fautes *Sans Effet* est explicable. Les résultats obtenus suite à l'injection des fautes sont présentés dans le Tableau 3.2.

Tableau 3.2 : Résultats d'injection des fautes pour Intel 80C51

Programme	Type	#Fautes	#Sans Effet	#Détection	#Réponse Fausse	#Séquence Perdue
Protocole I2C	Original	8192	7925	-	229	38
	Durci	24576	23622	929	13	12
Multiplication des Matrices	Original	8192	7926	-	225	41
	Durci	24576	23740	828	5	3
Tri Bubble Sort	Original	8192	7936	-	220	36
	Durci	24576	23673	890	6	7

Les résultats du Tableau 3.2 montrent que le nombre de fautes résultant en ***Réponse Fausse*** échappant au mécanisme de détection mis en application dans les versions durcies est réduit considérablement par rapport aux programmes originaux. En ce qui concerne le nombre d'erreurs de type ***Séquence Perdue***, il a été réduit d'environ trois fois pour le programme de protocole I2C et plus de cinq fois pour les programmes de multiplication des matrices et l'algorithme du tri ***bubble sort.***

Dans le Tableau 3.3, il est montré le taux d'erreurs pour chacun des programmes étudiés dans ces deux versions (la version originale et la version durcie). Le taux d'erreur a été calculé avec la formule *3.2*. Pour toutes les versions des programmes durcis, le taux d'erreur est réduit énormément par rapport aux celui de versions originaux de plus d'un ordre de grandeur dans le cas du protocole I2C et de deux ordres de grandeur pour les deux autres programmes (la multiplication des matrices et le programme du tri).

Tableau 3.3 : Taux d'erreur obtenu suite à l'injection des fautes

Programme	Type	$\tau_{\text{Injection des Fautes}}$
Protocole I2C	Original	3.25 %
	Durci	0.10 %
Multiplication des Matrices	Original	3.24 %
	Durci	0.03 %
Tri Bubble Sort	Original	3.12 %
	Durci	0.05 %

1 LFSR : Linear Feedback Shift Register

Le Tableau 3.4 illustre la valeur de l'efficacité de détection calculée selon l'équation *3.1* pour chacun des programmes durcis étudiés. Ces résultats prouvent que la méthode de détection proposée peut aboutir à produire une valeur très élevée d'efficacité de détection.

Tableau 3.4 : Efficacité de la détection obtenue suite à l'injection des fautes

Programme	$\xi_{Injection\ des\ Fautes}$
Protocole I2C	97.38 %
Multiplication des Matrices	99.04 %
Tri Bubble Sort	98.56 %

D. L'efficacité de chaque groupe de règles

L'efficacité des différents groupes de règles ajoutées en utilisant le translateur C2C est classifiée selon la règle qui a été déclenchée. La Figure 3.4 donnent le pourcentage des fautes détectées par chaque groupe de règles pour tous les programmes durcis.

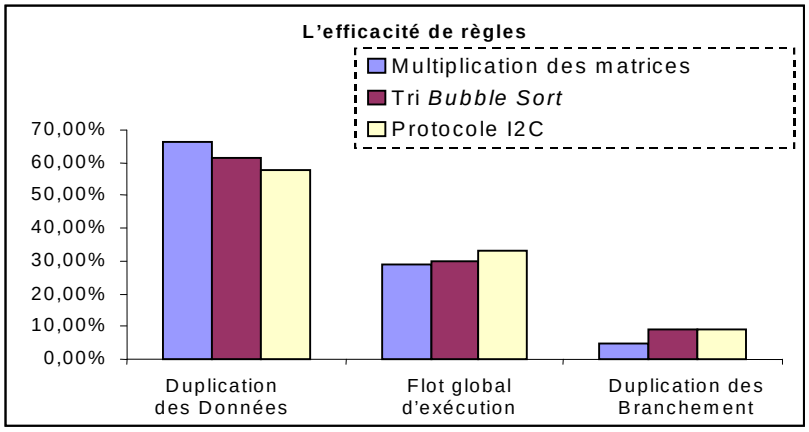


Figure 3.4 : Efficacité de règles de détection

Les résultats illustrés dans la Figure 3.4 montrent :

- La duplication de données est responsable de la majorité de détections d'erreurs (plus de 60% de basculements des bits détectés affectant les registres internes du microcontrôleur Intel 80C51) ;
- Un pourcentage significatif des basculements de bits (environ 30%) affectant le flot global d'exécution sont détectés par les règles appartenant au *FGE ;*

§ L'efficacité des règles visant le durcissement des branchements est faible par rapport aux deux autres groupes de règles (moins de 10%); cette efficacité dépend fortement de la structure (abondance des instructions de contrôle) du programme considéré.

E. Fautes échappant au mécanisme de détection

Nous avons fait une analyse détaillée et exhaustive dans les cas où les fautes injectées n'ont pas été détectées. La conclusion est que ces fautes sont celles provoquant un branchement erroné dans le programme.

Pour les fautes provoquant des résultats erronés (*Réponse Fausse*), deux cas se distinguent :

§ Un premier cas représente toutes les fautes ayant pour conséquence un branchement à une adresse près de la fin du programme, où l'indicateur du flot global d'exécution est dépassé.

§ Un deuxième cas concerne toutes les fautes modifiant le contenu d'une variable ainsi que le contenu de sa réplique. Cela peut se produire par un court branchement en arrière ce qui implique une re-exécution erronée des certaines instructions.

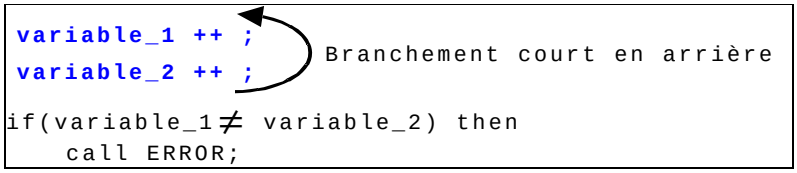


Figure 3.5 : Exemple d'un basculement de bit non détecté introduisant une erreur double dans le segment de données

La Figure 3.5 illustre un exemple d'une telle situation. Dans ce cas, l'opération d'écriture est effectuée deux fois sur la variable *variable_1* ainsi que sur sa copie *variable_2* (introduite en concordance avec le groupe de règles *Duplication des Données*). Le branchement en arrière modifie les valeurs correctes des variables. L'erreur introduite est alors propagée chaque fois que les deux variables sont impliquées dans une opération d'écriture sur d'autres variables. Cela sans changer le flot d'exécution du programme.

Dans le cas où des basculements des bits échappent au mécanisme de détection et qui ont pour résultat la *perte de séquencement*, le programme testé entre dans une boucle infinie ou il exécute un branchement vers une zone de mémoire non adressable. Les deux situations peuvent apparaître en un des cas présentés à la suite :

§ Certaines instructions sont modifiées et elles deviennent des branchements ;

§ Le déplacement d'un branchement existant est remplacé par une valeur fausse ;

§ L'adresse de retour de fonction est changée.

La Figure 3.6.a illustre un cas concret de perte de séquencement dû à l'apparition d'une boucle infinie. Généralement, cela est l'effet de la perturbation du registre de l'indicateur de pile ou de la pile elle même contenant l'adresse de retour d'une fonction. Dans cette dernière situation la pile est modifiée avant l'exécution de l'instruction de retour.

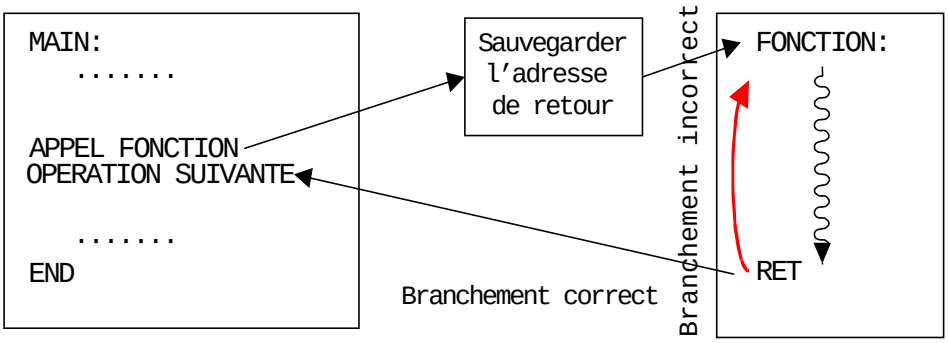


Figure 3.6.a : Exemple de faute échappant au mécanisme de détection – apparition d'une boucle infinie

La Figure 3.6.b montre un branchement incorrect vers une zone de mémoire inutilisée. Pour faire face à un tell fait, une technique généralement inutilisée consiste en remplir d'instructions de *nop* (aucune opération) la zone de mémoire inutilisée. Ainsi le processeur exécute seulement des instructions *nop*, jusqu'à la fin de la zone mémoire existante.

Evidemment, après l'exécution du dernier *nop*, le *compteur de programme* indique une adresse qui ne correspond pas à une cellule de mémoire physique.

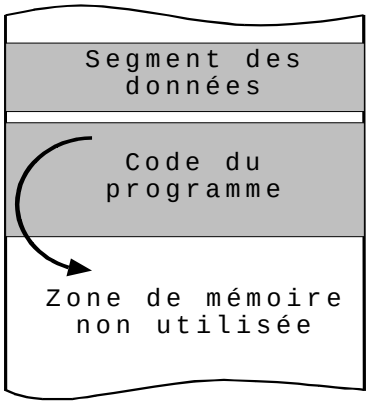


Figure 3.6.b : Exemple de faute échappant au mécanisme de détection – branchement vers une zone de mémoire non utilisée

Dans le Tableau 3.5 est donnée la classification des fautes non-détectées en utilisant comme critère de base la cause de non-détection.

Tableau 3.5 : Classification des fautes non-détectées

La cause provocant des erreurs non détectées	Type de fautes non détectée	
	Réponse fausse	Perte de séquencement
Branchement vers une adresse près de la fin du programme	40 %	-
Branchement court en arrière (affectant le segment des données)	60 %	-
Branchement incorrect vers une zone de mémoire inutilisée	-	20.8 %
Perturbation des registres critiques (la pile, le registre de pile)	-	78.2 %

V. Essais sous radiations

Pour confirmer les résultats obtenus à partir des expériences d'injection de fautes nous avons effectué une campagne d'essais sous radiations au sein du dispositif de cyclotron *CYCLONE* à *Louvain-La-Neuve* (Belgique).

Pendant les expériences le microcontrôleur 80C51 a été exposé à des faisceaux de trois types différents de particules (l'argon, krypton et xénon). Chaque programme a été exécuté sous radiations jusqu'à ce que le nombre total de particules (*la fluence*) affectant le microcontrôleur 80C51 soit de 10^6. Le Tableau 3.6 montre les caractéristiques des faisceaux utilisés.

Tableau 3.6 : Particules utilisées pour l'expérimentés de radiation

Particule	Flux (#particules/cm^2s)	LET(MeV cm^2/mg)	Temps d'exposition (sec)
Argon	5000	14.1	200
Krypton	6000	34.0	167
Xénon	5.600	55.9	179

Dans les tableaux 3.7, 3.8 et 3.9 les résultats obtenus dans l'expérience de radiations sont présentés pour chaque programme séparément.

Tableau 3.7 : Protocole I2C – résultats obtenus sur radiation

	Programme	Type	LET	#Upsets Observés	#Détection Logicielle	#Réponse Fausse	#Séquence Perdue
Protocole I^2C	Original	Argon	14.1	53	-	47	6
	Durci			68	60	5	3
	Original	Krypton	34.0	114	–	95	19
	Durci			169	155	4	10
	Original	Xénon	55.9	167	-	153	14
	Durci			235	208	18	9

Tableau 3.8 : Multiplication des matrices – résultats obtenus sur radiation

	Programme	Type	LET	#Upsets Observés	#Détection Logicielle	#Réponse Fausse	#Séquence Perdue
Multiplication des Matrices	Original	Argon	14.1	62	-	52	10
	Durci			60	57	2	1
	Original	Krypton	34.0	125	-	109	16
	Durci			178	161	9	8
	Original	Xénon	55.9	208	-	200	8
	Durci			255	234	14	7

Tableau 3.9 : Programme du tri *Bubble Sort* – résultats obtenus sur radiation

	Programme	Type	LET	#Upsets Observés	#Détection Logicielle	#Réponse Fausse	#Séquence Perdue
Tri Bubble Sort	Original	Argon	14.1	58	-	52	6
	Durci			68	63	2	3
	Original	Krypton	34.0	116	–	101	15
	Durci			162	139	7	16
	Original	Xénon	55.9	197	–	180	17
	Durci			202	179	13	10

Comme nous pouvons remarquer dans le Tableau 3.7, le Tableau 3.8 et le Tableau 3.9, les résultats obtenus pendant la campagne de radiations confirment les résultats venant dans des expériences d'injection des fautes. En effet, une caractéristique générale pour tous les programmes durcis est que l'efficacité de détection est plus grande que 85% (pour chacune des application durcies, l'efficacité de détection a été calculée selon le type de particules, c.f. *3.1*).

Le Tableau 3.10 récapitule les expériences de radiations; les résultats sont présentés comme la moyenne (de toutes les particules) pour chaque programme.

Tableau 3.10 : Résultats obtenu sous radiation (moyenne sur les trois faisceaux utilisés)

	Type	#Upset observés	#Détection	#Réponse Fausse	#Séquence Perdue
Protocole I2C	Original	111	-	98	13
	Durci	157	141	9	7
Multiplication des matrices	Original	132	-	120	11
	Durci	164	151	8	5
Tri Bubble Sort	Original	124	-	111	13
	Durci	144	127	7	10

Le Tableau 3.11 illustre l'efficacité de détection calculée suite aux essais d'injection des fautes et aux campagnes de test sous radiations pour tous les programmes durcis utilisés. Les résultats donnés dans ce tableau prouvent que dans le cas de radiations (les fautes produites sont les conséquences des effets des faisceaux d'ions lourds) l'efficacité de détection est approximativement de 10% moins élevée que dans le cas d'injection des fautes. Ceci peut être expliqué par le fait que pendant la réalisation des expériences d'injection des fautes, un simple bit-flip a été injecté par chaque exécution de programme. Evidemment, ce n'est pas le cas pour les essais des radiations où des renversements multiples des bits peuvent être produits.

Tableau 3.11 : Efficacité de la détection obtenue suite à l'injection des fautes et radiation

Programme	$\xi_{Injection\ des\ Fautes}$	$\xi_{Radiation}$
Protocole I2C	97.38 %	89.80 %
Multiplication des Matrices	99.04 %	92.07 %
Tri Bubble Sort	98.56 %	88.19 %

VI. Conclusions

Nous avons présenté une application de la méthodologie de détection de erreurs de type basculement de bits dans le cas d'une architecture digitale bâtie autour du microcontrôleur Intel 80C51. Les basculements des bits dans des zones accessibles sont injectés simultanément avec l'exécution du programme testé par une interruption asynchrone afin de simuler le comportement du programme face aux effets des radiations. Les performances des programmes durcis en terme de détection d'erreurs de type basculement des bits, ont été évaluées pour trois programmes durcis obtenus d'une manière automatique suite à l'utilisation d'un outil logiciel développé dans le cadre de cette thèse : le translateur C2C.

Le chapitre suivant consistera en l'extension de l'application de cette technique à un processeur numérique de traitement de signal comme le processeur DSP32C Lucent. L'objectif est la conception des logiciels capables de s'exécuter dans des environnements radioactifs. Ces logiciels pourront être utilisés à bord des véhicules spatiaux.

Chapitre 4

Evaluation de la méthode sur une application satellite

- Introduction
- Architecture du processeur DSP32C
  - Unité arithmétique de données (DAU)
  - Unité arithmétique de contrôle (CAU)
  - Organisation d'espace mémoire
- Caractéristiques générales de l'application testée
- Injection de fautes
  - Le simulateur D3SIM
  - La stratégie d'injection d'upset
- Résultats expérimentaux
  - Injection des fautes dans les données du programme
  - Injection des fautes dans la pile du programme
  - Injection des fautes dans le code du programme
  - Injection des fautes dans les registres du processeur
- Essais sous radiations
- Conclusions

I. Introduction

Pour évaluer l'efficacité de la méthode de détection d'erreurs proposée et pour obtenir une idée de sa généralité et de son applicabilité pour différentes architectures, nous avons effectué des campagnes d'injection de fautes et des tests sous radiations sur une architecture à base d'un processeur complexe. Il s'agit du processeur DSP32C de LUCENT technologie. Le DSP32C est un processeur numérique de traitement de signal, candidat au projet spatial argentin µARGO (Argentinian Gamma-ray Observatory). Ce projet a la mission scientifique d'explorer les rayons gamma avec une grande précision à l'aide d'un télescope puissant embarqué.

Nous utilisons une application industrielle complexe utilisée dans le domaine de la communication spatiale : un égaliseur CMA (Constant Modules Algorithme).

Une première partie de ce chapitre sera consacrée à l'architecture générale du processeur DSP32C. Une deuxième partie présentera les caractéristiques générales de l'application que nous avons utilisée tout au long de nos expériences : l'application CMA. La troisième partie de ce chapitre expliquera la stratégie d'injection des fautes utilisée ainsi que l'outil mettant en ouvre cette stratégie. La quatrième partie illustrera les résultats expérimentaux d'injection des fautes obtenus suite à l'application de la méthodologie de détection d'erreurs dans le cas de l'application CMA. La cinquième partie donnera les résultats expérimentaux des tests sous radiations. La dernière partie de ce chapitre présente les conclusions.

II. Architecture du processeur DSP32C

Les principaux blocs fonctionnels de l'architecture du DSP32C sont illustrés dans la Figure 4.1. Elles sont :

- RAM0 de 2KO
- RAM1 de 2KO
- RAM2 de 2KO
- La CAU (unité arithmétique de contrôle)
- La DAU (unité arithmétique de données)
- Un port parallèle
- Un port série

- Bus des données de 32 bits
- Bus d'adresses de 24 bits
- Une unité de contrôle de pipeline

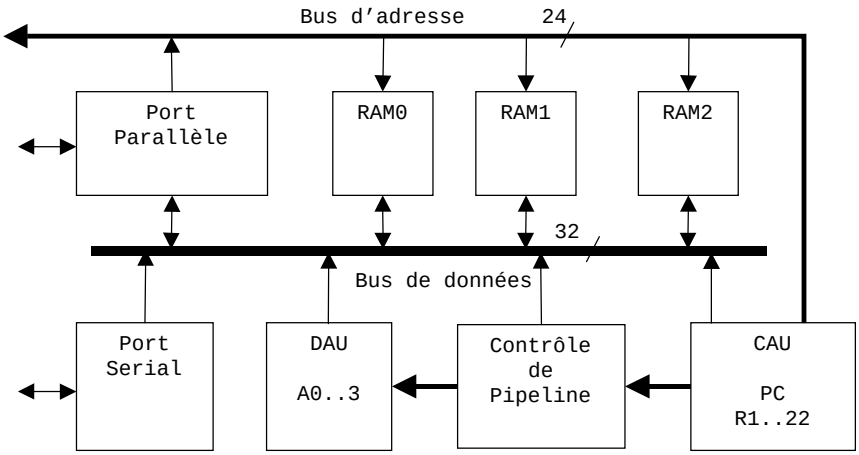


Figure 4.1 : Structure interne de DSP32C

L'architecture du processeur DSP32C possède deux unités d'exécution : l'unité arithmétique de contrôle (CAU) et l'unité arithmétique de données (DAU). Chaque unité d'exécution a son propre groupe d'instructions. La CAU effectue des opérations arithmétiques sur des données en virgule fixe sur 16 ou 24 bits pour les fonctions logiques ou de contrôle, tandis que la DAU effectue des opérations sur des données en virgule flottante sur 32 bits et des opérations de conversion des données. La CAU peut fonctionner dans un mode autonome et elle est en charge du transfert de données, du contrôle des branchements et des opérations arithmétiques ou logiques en virgule fixe. Cela, en parallèle avec l'exécution des opérations en virgule flottante par la DAU. En plus la CAU génère les adresses pour les opérandes de la DAU.

L'exécution d'une instruction par la CAU est finie avant le commencement de l'instruction suivante ; l'instruction de chargement d'un registre représente une exception. Cela simplifie la CAU pour des opérations logiques et de contrôle. La DAU utilise un pipeline en quatre étapes ce qui permet l'exécution de 25 millions de calculs en virgule flottante par seconde. Ainsi, le groupe d'instructions de la CAU présente un nombre réduit de latences.

Le port d'entrées/sorties parallèles (PIO) fournit une interface parallèle pour la communication entre le DSP32C et les composants externes. Ce port peut être configuré comme un port de 8 bits (compatibilité avec le DSP32C) ou comme un port de 16 bits. Le port d'entrées/sorties sérielles (SIO) fournie la communication sérielle et la synchronisation avec les composants situés à l'extérieur du DSP32C.

A.1. Unité arithmétique de données (DAU)

La DAU est l'unité principale d'exécution pour les algorithmes de traitement de signal. Cette unité contient un multiplicateur sur 32 bits en virgule flottante, un additionneur sur 40 bits en virgule flottante, quatre accumulateurs sur 40 bits et un registre de contrôle de l'unité arithmétique de données (DAUC). Le multiplicateur et l'additionneur fonctionnent pour effectuer 12.5 millions de calculs par seconde de forme ($a=b+c*d$). La Figure 4.2 illustre le schéma bloc de la DAU.

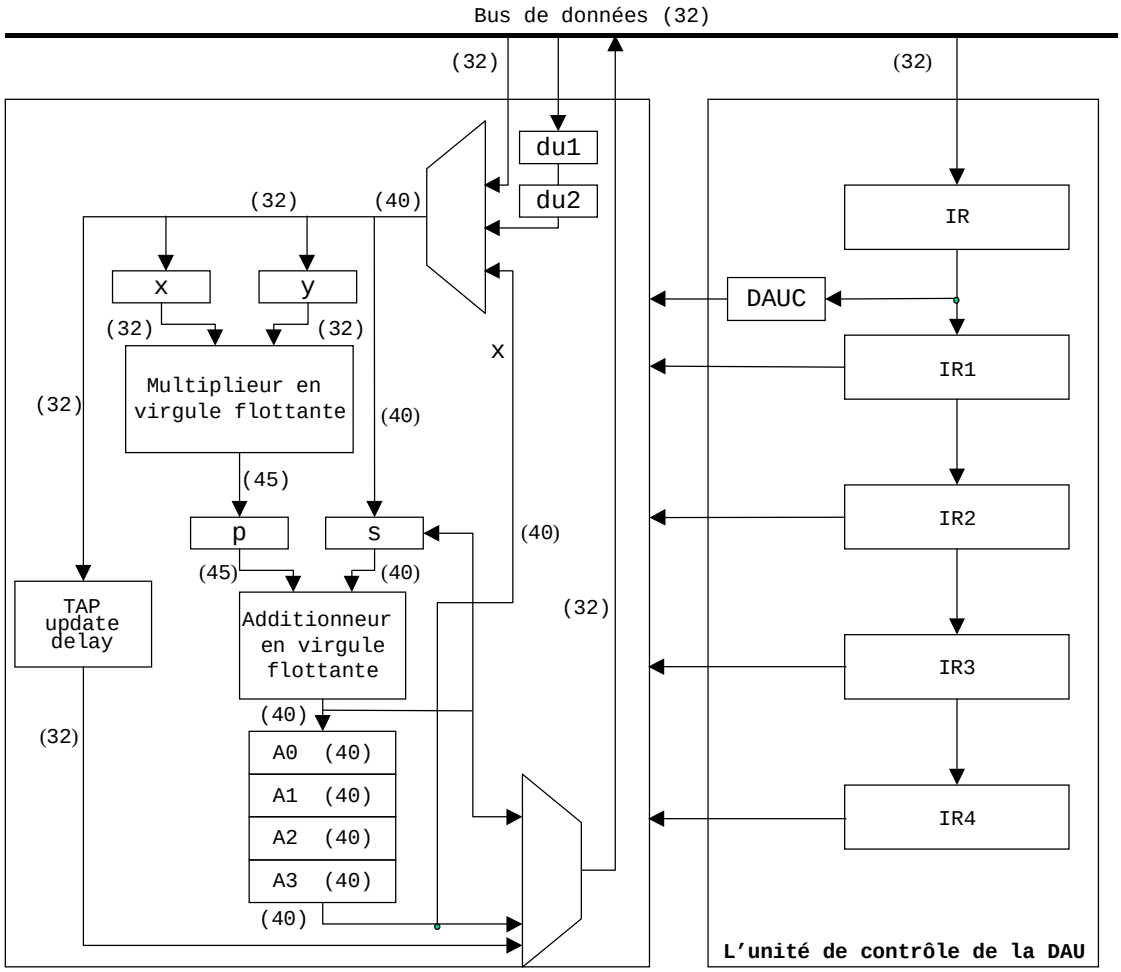


Figure 4.2 : Structure du DAU

- A0-A3 : Accumulateurs
- DAUC : Registre de contrôle de DAU
- du1, du2 : registres de délai
- IR : Registre d'instruction
- IR1, IR2, IR3, IR4 : Registre d'instruction DAU (pour chaque cycle)

- p : Registre de multiplication (il garde le résultat de la multiplication entre les registres x et y)
- s : Registre de fonction spéciale
- x : Registre de multiplication (il est opérande)
- y : Registre de multiplication (il est opérande)

Le bus interne de communication assure la communication de données entre la DAU et d'autres blocs. La structure de la DAU est divisée en une partie de contrôle et une partie de traitement de données. Le traitement de données est composé d'un multiplieur en virgule flottante, un additionneur en virgule flottante, des registres des bus et des connecteurs. Le multiplieur et l'additionneur fonctionnent en parallèle, chacun produisant un résultat par cycle d'instruction. Les quatre registres accumulateurs peuvent être accédés en lecture ou en écriture par le programme. L'unité de contrôle de la DAU décode les instructions dans des signaux de commande du chemin de données. Le chemin de données ainsi que l'unité de contrôle contient un pipeline en quatre étapes. Ainsi, pendant chaque cycle d'instruction, la DAU peut traiter quatre instructions différentes ; les instructions peuvent se trouver dans des étapes d'exécution différentes.

La DAU contient des registres de 32 bits ainsi que des registres de 40 bits et des bus afin de pouvoir supporter deux formats de données en virgule mobile : des données en virgule mobile de précision 32 bits et 40 bits. Le format sur 40 bits fourni une mantisse de 40 bits utilisée par la DAU pendant les opérations d'accumulation.

A.2. Unité arithmétique de contrôle (CAU)

La CAU réalise le calcul des adresses, le contrôle de branchement et les opérations logiques et arithmétiques sur 16 bits or 24 bits. La CAU est composée de :

- une unité arithmétique logique (ALU) sur 24 bits qui effectue les opérations logiques ou les opérations arithmétiques des entiers
- un registre compteur de programme (PC) sur 24 bits
- 22 registres d'usage général sur 24 bits

La CAU présente deux modes d'opération :

- un mode qui exécute les instructions de contrôle arithmétique (CA)
- un mode qui génère les adresses pour les instructions arithmétiques de données (DA).

Les instructions DA effectuent les mouvements des données, les branchements de contrôle et les opération arithmétiques ou logiques sur des entiers de 16 ou 24 bits. Les instructions DA peuvent présenter plus que quatre accès mémoire par instruction et la CAU est responsable de la

génération de ces adresses en utilisant le mode d'adressage *post-modifié* par registre indirect – une adresse pendant chacune des quatre étapes d'un cycle instruction. La Figure 4.3 illustre le schéma global de la CAU.

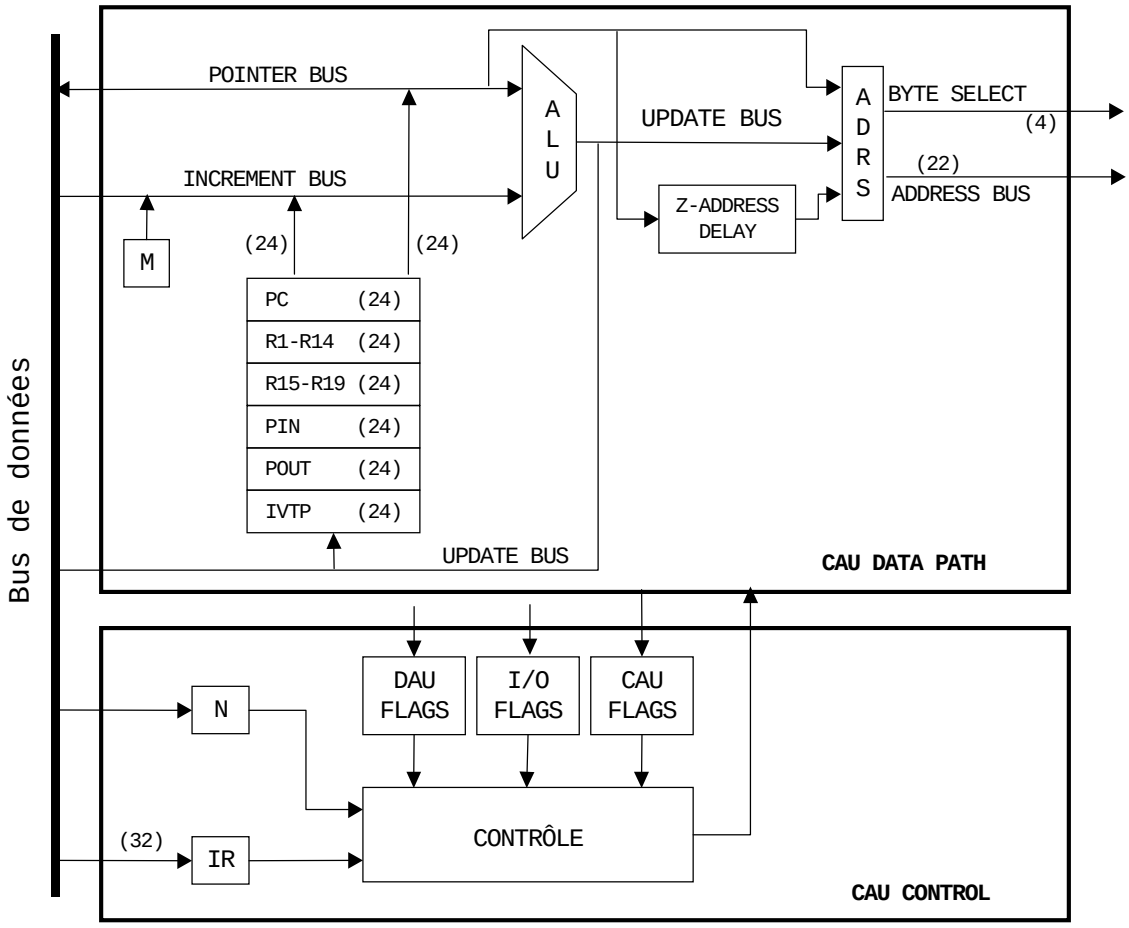


Figure 4.3 : Structure du CAU

- IR : Instruction registre
- PC : Compteur de programme
- R1-R14 : Registres d'usage général de CAU ; pointeur de registre de DAU
- R15-R19 : Registres d'usage général de CAU ; registre d'incrémentation de DAU
- PIN (R20) : Registre d'usage général de CAU ; registre d'entrée série pour DMA
- POUT (R21) : Registre d'usage général de CAU ; registre de sortie série pour DMA
- IVTP (R22) : Registre d'usage général de CAU ; Pointeur vers le tableau d'interruptions
- M : Compteur de nombre d'instructions dans une boucle
- N : Compteur de nombre d'itérations pour une boucle

Les registres R1-R4 sont utilisés comme des registres d'usage général pour les instructions de contrôle arithmétique (CA) et comme des pointeurs vers la mémoire pour les instructions arithmétiques de données (DA). Dans le cas ou ils sont utilisés comme pointeurs vers la mémoire les registres R1-R4 stockent des adresses de 24 bits. Les registres R15-R19 sont utilisés comme des registres d'usage général pour les instructions de contrôle arithmétique (CA) et comme registres d'incrément (RI) pour les instructions arithmétiques de données (DA). Dans le cas ou ils sont utilisés comme des registres d'incrémentation, les registres R15-R19 stockent des valeurs sur 24 bits, valeurs qui peuvent modifier les adresses dans les pointeurs vers la mémoire. Le registre R20 dit Pin (*pointer in*) est utilisé comme le pointer d'entrée sérielle (SIO) pour l'accès direct à la mémoire (DMA). Le registre R21 dit POU (pointeur out) est utilisé comme le pointeur de sortie vers le SIO de la DMA. Dans le cas où les registres R20 et R21 sont utilisés comme des registres d'usage général, leur effet sur la DMA est pris en compte si la DMA sérielle est activée. Le registre R22 dit aussi IVTP (de l'anglais *interrupt vector table pointer*) stocke l'adresse de base du pointeur de la table d'interruptions. Dans le cas où le registre R22 est utilisé comme un registre d'usage général, son effet sur les interruptions doit être pris en compte si les interruptions sont activées.

A.3. Organisation d'espace mémoire

Le processeur DSP32 possède la mémoire interne et une interface pour la mémoire externe afin de permettre l'extension mémoire en extérieur de la puce et *memory mapped peripherals.* Deux options sont possibles pour la mémoire sur puce. Dans le cas de la première option la mémoire interne est organisée en trois RAM de 512 mots. La deuxième option offre la possibilité de remplacer une RAM de 512 mots par une ROM de 4K mots. L'adressage sur 24 bits augmente la capacité de la mémoire externe à 16 MOctets. Les programmes peuvent traiter la mémoire comme une ressource commune, les données et les instructions résidant aléatoirement dans la RAM interne, dans la ROM interne ou dans la mémoire externe.

La Figure 4.4 illustre la configuration mémoire utilisée dans le cas de nos expériences. Le processeur DSP32C présente trois mémoires RAM internes et deux mémoires RAM externes. La mémoire RAM externe A stocke le code du programme. La mémoire RAM externe B est utilisée comme une mémoire partagée ; elle stocke aussi les données d'entrée/sortie. La mémoire interne RAM 0 est utilisée pour stocker les données du programme, la mémoire interne RAM 2 stocke la pile du programme et la mémoire RAM 1 est inutilisée.

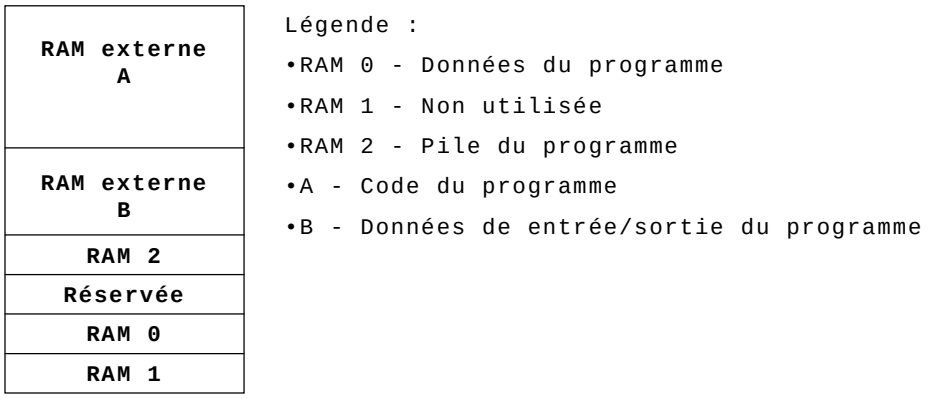


Figure 4.4 : Organisation d'espace mémoire

III. Caractéristiques générales de l'application testée

L'application que nous avons utilisée pour l'expérimentation est un programme utilisé dans les communications spatiales : il s'agit d'un égaliseur CMA, basé sur l'algorithme d'égalisation Bussgang [42].

En utilisant le translateur C2C, les règles proposées dans le chapitre II ont été appliquées à l'égaliseur CMA. A l'aide du translateur nous avons généré automatiquement la version durcie de cette application.

Tableau 4.1 : Caractéristiques générales de programmes testés

Caractéristiques du programme	CMA original	CMA durci	
Temps d'exécution (cycles)	1,231,162	3,251,254	2.64
Code du programme (octets)	1,104	4,100	3.71
Mémoire de données (octets)	1,996	4,032	2.02
Pile (octets)	44	72	1.60

Le Tableau 4.1 donne des éléments de comparaison entre la version originale du programme utilisé et sa version durcie. Les critères considérés sont : le temps d'exécution, la taille du programme, l'espace mémoire des données et la taille de la pile. Comme le tableau le montre, le facteur de pénalité en temps d'exécution du programme durci (voir le chapitre II de ce mémoire) est 2.64. L'espace mémoire occupé par le code du programme durci est environ 3.7 fois plus grand que l'espace mémoire nécessaire pour la version originale. En ce qui concerne l'espace mémoire utilisé pour les données du programme la version durcie requiert un espace mémoire double. La taille de la pile augmente 1,6 fois dans le cas du programme durci.

IV. Injection de fautes

La comparaison du comportement du programme CMA durci avec le programme CMA original et l'évaluation des performances en terme d'efficacité de détection pour la version durcie nécessite un mécanisme d'injection des fautes qui sera présenté dans ce paragraphe. Pour cela, dans le cas de nos expériences nous avons défini un tel mécanisme pour le simulateur D3SIM. La compréhension du mécanisme d'injection de fautes nécessite une vue d'ensemble du simulateur. Cette vue d'ensemble est présentée dans une première section de ce paragraphe. Ensuite nous présenterons la stratégie d'injection.

A. Le simulateur D3SIM

Pour injecter des bit-flips pendant l'exécution du programme CMA, nous avons utilisé un simulateur disponible dans le commerce appelé le D3SIM [43]. Le fonctionnement de cet outil est basé sur l'utilisation d'un fichier spécifique (noté *«.ex»*) que nous allons appeler par la suite *le fichier de commandes.*

Pour étudier les effets d'un bit-flip, nous utilisons un tel fichier ".ex". Celui-ci permet le contrôle de l'insertion d'un bit-flip à un instant spécifique de l'exécution. La Figure 4.5 montre quatre commandes d'un tel fichier pour l'injection d'un bit-flip et puis pour analyser des résultats.

```
(1) when t > TOTAL_CYCLES{"Perte de séquencement" ; quit}
(2) when t >= INSTANT {TARG=TARG ^ XOR_VALUE; continue}
(3) b  end_program {output data > "res.dat"; quit}
(4) run
```

Figure 4.5 : Exemple de fichier de commande

La commande (2) indique au simulateur D3SIM que l'injection de faute sera effectuée à l'instant INSTANT. Quand cette condition est satisfaite, l'exécution du programme est suspendue et l'opération XOR entre la cible choisie TARG et un masque binaire approprié XOR_VALUE est réalisée. Ultérieurement, l'exécution du programme testé est reprise. TARG symbolise une adresse de mémoire ou un registre et XOR_VALUE indique la position du bit affecté.

La commande (3) place un point d'arrêt à la fin du programme et les données de sortie sont stockées dans un fichier spécifique. La dernière commande (4) est nécessaire pour lancer le programme.

Puisque les bit-flips peuvent perturber le flot d'exécution provoquant le dysfonctionnement critique comme *perte de séquencement* (les boucles infinies ou l'exécution des instructions invalides), une commande permettant la mise en œuvre d'un chien de garde pour détecter des tels événements doit être introduite dans le fichier de commandes. La commande (1) de la Figure 4.5 est un exemple d'une telle commande. TOTAL_CYCLES représente le nombre total de cycles nécessaire pour l'exécution du programme testé.

B. La stratégie d'injection d'upsets

Afin d'automatiser l'injection de fautes nous avons développé un outil logiciel qui présente deux entrées :

- Le programme de test
- La cible injectée avec une faute (la RAM interne, le code du programme ou les registres du processeur)

Le principe de base du mécanisme d'injection proposé est illustré dans la Figure 4.6.

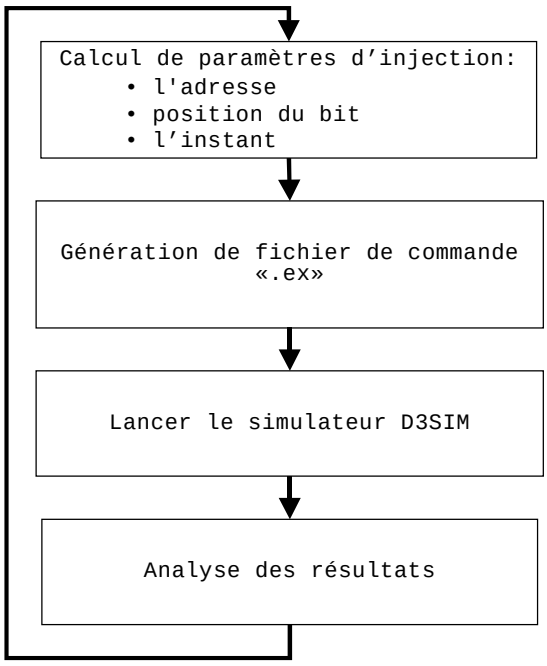


Figure 4.6 : Mécanisme d'injection des bit-flips

Dans une première étape, les paramètres d'injection sont calculés à l'aide d'un algorithme de génération de nombres aléatoires [44]. Dès que l'INSTANT et la CIBLE ont été calculés, un fichier

de commandes est généré afin de lancer le simulateur D3SIM pour l'injection des fautes. La dernière étape est l'analyse et la classification des résultats obtenus. Plus spécifiquement, cette analyse consiste en la classification des effets des upsets selon les critères présentés dans le Chapitre III. Une statistique des upsets est aussi effectuée.

V. Résultats expérimentaux

Le mécanisme d'injection de fautes présenté précédemment a été utilisé pour évaluer le comportement du programme CMA durci par rapport au comportement du programme CMA original. Nous avons effectué quatre séances d'injection : au niveau de données du programme, au niveau de la pile, au niveau du code du programme et au niveau des registres internes du processeur. Les résultats obtenus pendant ces quatre séances seront présentés par la suite.

A. Injection des fautes dans les données du programme

Le Tableau 4.2 résume les résultats obtenus dans le cas d'injection des fautes dans le segment de données du programme. La classification des effets des fautes est la même avec celle présenté dans le chapitre précèdent (voir Chapitre III).

Dans une première expérience pour chaque exécution du programme nous avons injecté un seul bit-flip dans la zone mémoire réservée aux données du programme. Dans le cas du programme durci, le nombre de bit-flip injectés est doublé en concordance avec le facteur d'augmentation de l'espace mémoire occupée par les données du programme.

Tableau 4.2 : Résultats d'injection des fautes dans les données du programme (RAM 0)

Version Programme	#Fautes injectées	Classification des effets des fautes injectées				
			Détectées		Non-Détectées	
		#Sans Effet	#Détection Logicielle	#Détection Matérielle	#Réponse Fausse	#Séquence Perdue
CMA Original	5000	4072 (81.44 %)	_	_	928 (18.56 %)	_
CMA Durci	10000	6021 (60.21 %)	3979 (39.79 %)	_	_	_

Dans le cas du programme durci nous avons détecté 39,79% du total des fautes injectées. Sachant que 60,21% du nombre de fautes sont des fautes ***sans effet***, aucune faute n'échappe au mécanisme de détection implanté. Cependant, dans le programme original 18,56% des fautes

injectées mènent à des *réponses fausses*. Le reste de 81,44% de fautes injectées ont été des fautes sans effet.

Dans le Tableau 4.3, nous présentons le taux d'erreurs pour chacun des programmes étudiés (la version originale et la version durcie). Ce taux a été calculé en utilisant l'équation 3.2 du chapitre précédent. Comme le tableau le montre, le taux d'erreur pour le programme durci est zéro. Avec l'équation 3.1 du chapitre précédent, qui décrit l'efficacité de détection, nous obtenons une efficacité de détection de 100%.

Tableau 4.3 : Taux d'erreur calculé – injection de fautes dans les données du programme

Programme	$\tau_{\text{Injection des Fautes}}$
CMA Original	18.56 %
CMA Durci	0 %

Efficacité de détection de chaque groupe de règles

Le Tableau 4.4 illustre l'efficacité de chaque groupe de règles. L'efficacité est exprimée en termes de pourcentage du nombre total de fautes détectées.

Tableau 4.4 : Efficacité de règles de détection - fautes injectées dans les données du programme

Groupes de règles		
Duplication de Données	Flot Global d'Exécution	Duplication de Branchement
82.74 %	0.00 %	17.26 %

En concordance avec nos attentes, la plupart des fautes injectées dans le segment de données, ont été détectées en utilisant le groupe de règles *Duplication de Données*. Une petite partie des fautes qui ont été détectée sont dues à l'utilisation du groupe de règles *Duplication de Branchement* ; les fautes ont affecté les variables impliquées dans les conditions de test pour les instructions conditionnelles de branchement.

B. Injection des fautes dans la pile du programme

Nous utilisons le même principe de choix du nombre de fautes injectées que celui présenté dans la section précédente. Le Tableau 4.5 résume les résultats obtenus dans le cas d'injection des fautes dans la pile du programme. L'expérience acquise dans le cas du processeur Intel 8051 nous a poussé à traiter séparément le cas d'injection des fautes dans la pile du cas d'injection des fautes dans les données du programme.

Tableau 4.5 : Résultats d'injection des fautes dans la pile du programme (RAM 2)

Version Programme	#Fautes injectées	Classification des effets des fautes injectées				
			Détectées		Non-Détectées	
		#Sans Effet	#Détection Logicielle	#Détection Matérielle	#Réponse Fausse	#Séquence Perdue
CMA Original	88	70 (79.54 %)	_	_	14 (15.90 %)	4 (4.56 %)
CMA Durci	144	120 (83.34 %)	20 (13.89 %)	_	3 (2.08 %)	1 (0.69 %)

Suite aux expérimentations, 4 fautes ont échappé au mécanisme de détection (1 faute déclenchant une perte de séquencement et 3 fautes déclenchant une réponse fausse), tandis que dans la version originale 18 fautes ont provoqué une déviation du comportement du programme (réponse fausse et séquence perdue).

Tableau 4.6 : Taux d'erreur calculé – injection de fautes dans la pile du programme

Programme	$\tau_{\text{Injection des Fautes}}$
CMA Original	20.45 %
CMA Durci	2.77 %

Le taux d'erreur calculé dans le cas d'injection de fautes dans la pile est illustré dans le Tableau 4.6. Ce taux est considérablement réduit (10 fois) pour la version durcie du programme. L'efficacité de détection pour le programme durci est 83.33%.

Efficacité de chaque groupe de règles

Dans le Tableau 4.7 nous donnons le pourcentage du nombre de fautes détectées par chaque groupe de règles.

Tableau 4.7 : Efficacité de règles de détection - fautes injectées dans la pile du programme

Groupes de règles		
Duplication de Données	Flot Global d'Exécution	Duplication de Branchement
57.15 %	9.52 %	33.33 %

Nous observons que le groupe de règles *Duplication de Données* a la plus grande contribution avec approximativement 60% du nombre total des fautes détectées. Seulement 9.52 % du nombre total de fautes détectées sont dues à l'utilisation du groupe de règles *Flot Global d'Exécution*, et 33.33 % sont détectées par le groupe *Duplication de Branchement*.

C. Injection des fautes dans le code du programme

Dans ce cas d'injection des fautes, le nombre de fautes injectées correspond au nombre d'octets nécessaires pour stocker les programmes, multiplié par 8. Cela pour permettre l'injection des fautes dans tous les bits de la mémoire stockant le code du programme. Le Tableau 4.8 montre les résultats obtenus dans le cas d'injection des fautes dans le code du programme.

Tableau 4.8 : Résultats d'injection des fautes dans le code du programme

Version Programme	#Fautes injectées	Classification des effets des fautes injectées				
		#Sans Effet	Détectées		Non-Détectées	
			#Détection Logicielle	#Détection Matérielle	#Réponse Fausse	#Séquence Perdue
CMA Original	8832	5951 (67.38 %)	_	1416 (16.03 %)	753 (8.53 %)	712 (8.06 %)
CMA Durci	32800	20328 (61.97 %)	7137 (21.76 %)	5015 (15.29 %)	160 (0.49 %)	160 (0.49 %)

Les résultats du Tableau 4.8 montrent que le nombre de fautes ayant comme effet une instruction illégale (***Détection Matérielle***) est approximatif de 15 % dans les deux programmes (le CMA original et le CMA durci). Dans le cas du programme CMA durci, les fautes échappant au mécanisme de détection mis en application est réduit avec un facteur de 4.5 (ce facteur a été calculé par la division du nombre total des fautes non-détectées dans le programme original par le nombre des fautes non-détectées par le programme durci).

Dans le Tableau 4.9, nous présentons le taux d'erreur pour chacun des programmes étudiés (la version originale et la version durcie).

Tableau 4.9 : Taux d'erreur calculé – injection de fautes dans le code du programme

Programme	$\tau_{\text{Injection des Fautes}}$
CMA Original	16.58 %
CMA Durci	0.97 %

Le taux d'erreur a été réduit considérablement avec plus d'un ordre de grandeur. L'efficacité de détection est de 95.70 %.

Efficacité de chaque groupe de règles

Le Tableau 4.10 donne le pourcentage des fautes détectées en utilisant chaque groupe de règles. Comme le tableau le montre, chaque groupe de règles a sa contribution pour la détection des fautes injectées. Le groupe *FGE* est responsable de la détection de la plupart des fautes injectées.

Tableau 4.10 : Efficacité de règles de détection - fautes injectées dans la pile du programme

Groupes de règles		
Duplication de Données	Flot Global d'Exécution	Duplication de Branchement
38.17 %	40.45 %	21.38 %

D. Injection des fautes dans les registres du processeur

L'injection des fautes dans les registres internes du processeur DSP32C a consisté en la modification du contenu de la plupart des registres accessibles par le simulateur. Malheureusement, 11 sur 55 registres (les registre de pipeline (x, y, s, p), les registres d'instructions DAU (IR_i), le registre *compteur de nombre d'instructions* dans une boucle (M) et le registre *compteur de nombre* d'itérations pour une boucle (N)) ne sont pas accessibles par le simulateur. Les registres dans lesquels des fautes ont été injectées, à l'aide de simulateur D3SIM sont :

- A0-A3 : Accumulateurs
- DAUC : Registre de contrôle de DAU
- PC : Compteur de programme
- PCSH : Copie du PC
- R1-R14 : Registres d'usage général de CAU ; pointeur de registre de DAU
- R15-R19 : Registres d'usage général de CAU ; registre d'incrémentation de DAU
- PIN (R20) : Registre d'usage général de CAU ; registre d'entrée série pour DMA
- POUT (R21) : Registre d'usage général de CAU ; registre de sortie série pour DMA
- IVTP (R22) : Registre d'usage général de CAU ; Pointeur vers le tableau d'interruptions
- IBUF, OBUF, IOC, PAR, PARE, DPR, DPR2, PIR, PCR, EMR, ESR, PCW, PIOP

Le Tableau 4.11 présente les résultats obtenus dans le cas d'injection des fautes dans les registres du processeur DSP32C. Afin d'obtenir des résultats réalistes, le nombre de fautes injectées est choisi en fonction du nombre de cycles d'horloge pendant lesquels le programme est exécuté : les fautes injectées dans le processeur peuvent affecter le programme uniquement pendant son exécution.

Par exemple, si le programme original s'exécute dans 10^6 cycles d'horloge, il faudra injecter un million de fautes. Malheureusement, le temps pris pour la simulation d'une seule faute de type bit-flip est de 6 secondes dans le cas de programme CMA original. Donc, nous avons besoin de 70 jours pour injecter un million de bit-flips. C'est pour cette raison, que dans le cas de notre expérimentation nous avons choisi un nombre de 20000 fautes injectées dans le CMA original et 52800 fautes injectées dans le CMA durci (selon le facteur d'augmentation de temps d'exécution).

Nous pouvons remarquer une forte réduction du nombre de fautes qui mènent à une perte de séquencement dans le programme CMA durci – facteur de 53. Concernant le nombre de fautes déclenchant une réponse fausse, il est réduit avec un facteur de 4.5 dans la version durcie du programme.

Dans le cas de deux versions des programmes, 1.11% du nombre total de fautes sont détectées par le mécanisme matériel de détection, mécanisme intégré dans le processeur DSP32C.

Tableau 4.11 : Résultats d'injection des fautes dans les registres du DSP32C

Version Programme	#Fautes injectées	Classification des effets des fautes injectées				
			Détectées		Non-Détectées	
		#Sans Effet	#Détection Logicielle	#Détection Matérielle	#Réponse Fausse	#Séquence Perdue
CMA Original	20000	18810 (94.05 %)	_	223 (1.11 %)	490 (2.45 %)	477 (2.39 %)
CMA Durci	52800	48720 (92.27 %)	3371 (6.38 %)	591 (1.12 %)	109 (0.21 %)	9 (0.02 %)

Le Tableau 4.12 montre les taux d'erreur calculés pour les programmes testés. Dans le cas du programme durci le taux d'erreur a été réduit avec un facteur plus grand que 20 par rapport au taux d'erreur du programme original. L'efficacité de détection est de 96%.

Tableau 4.12 : Taux d'erreur calculé – injection de fautes dans les registres du programme

Programme	$\tau_{\text{Injection des Fautes}}$
CMA Original	4.83 %
CMA Durci	0.22 %

L'efficacité de chaque groupe de règles

Le Tableau 4.13 donne le pourcentage du nombre de fautes détectées par chaque groupe de règles. Comme le tableau le montre, le groupe de règles *FGE* présente l'efficacité la plus élevée.

Ainsi, la moitié du nombre total de fautes détectées est le résultat de l'utilisation de ce groupe de règles.

Tableau 4.13 : Efficacité de règles de détection - fautes injectées dans la pile du programme

Groupes de règles		
Duplication de Données	Flot Global d'Exécution	Duplication de Branchement
35.40 %	50.00 %	14.60 %

VI. Essais sous radiations

Pour confirmer les résultats obtenus à partir des expériences d'injection de fautes nous avons effectué une campagne de test sous radiations.

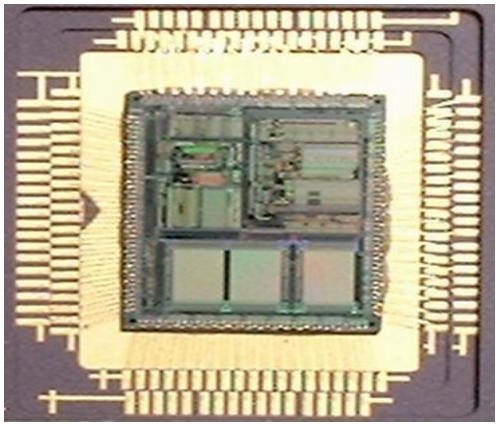


Figure 4.7 : La puce DSP32C (décapoté)

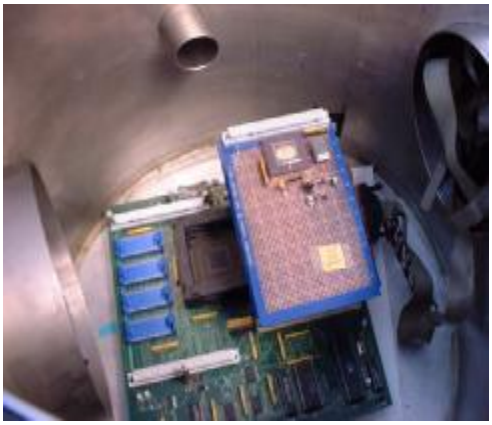


Figure 4.8 : DSP32C à l'intérieur de la chambre sous vide

Les deux programmes étudiés (CMA Original et CMA Durci) ont été exposés sous un faisceau de particules (californium) avec un flux de 285 particules par secondes. En concordance avec le temps d'exécution de chaque programme et le flux de particules nous pouvons estimer le nombre de particules pour chaque exécution du programme.

Ceci sera calculé avec la formule :

$N = \Phi \times T_a$ où :

- Φ : représente le flux de particules
- T_a : représente le temps d'exécution du programme

Dans le Tableau 4.14 sont illustrés les principaux éléments pour chaque programme testé sous radiation. Nous rappelons que les définitions du flux, fluence et du temps d'exposition ont été présentés dans le chapitre III, section 2.

Tableau 4.14 : Fluence, Flux, Temps d'exposition et Nombre de particules per exécution du programme

Programme	Fluence	Flux	$N = \Phi \times T_a$	Temps d'exposition
CMA Original	149,625	285	350	525 (8.75 min.)
CMA Durci	188,333	285	926	660 (11 min.)

Comme le Tableau 4.14 le montre, le nombre de particules (*N*) affectant le processeur DSP32C pendant une seule exécution du programme, a été plus de deux fois plus élevé dans le cas du programme durci. Les résultats obtenus dans cette campagne de radiation sont présentés dans le Tableau 4.15.

Tableau 4.15 : Résultats des radiations

Version Programme	Upsets observés	Classification des effets des fautes injectées			
		Détectées		Non-Détectées	
		#Détection Logicielle	#Détection Matérielle	#Réponse Fausse	#Séquence Perdue
CMA Original	48	_	_	47 (97.91 %)	1 (2.09 %)
CMA Durci	99	84 (84.85 %)	_	15 (15.15 %)	_

L'efficacité de détection obtenue dans le cas du programme durci est de 84.85%, confirmant donc l'efficacité de détection dérivée de l'injection de fautes dans des registres du processeur DSP32C. Les erreurs non-détectées, ont été réduites d'un facteur de 3, bien que le programme durci ait été exposé sous radiations une période de temps plus longue.

VII. Conclusions

Dans ce chapitre nous avons présenté l'application de la méthode de détection proposée, pour un processeur numérique de traitement de signal comme le DSP32C. Le programme étudié est une application spatiale, implantant l'algorithme de communication CMA. Dans le but d'évaluer cette approche, une étape préliminaire d'injection des fautes a été nécessaire pour le processeur DSP32C.

La comparaison des performances du programme de test original à celles du programme durci montre le gain apporté par la méthodologie proposée : le taux d'erreur obtenu par le durcissement du programme a été réduit d'un facteur supérieur à 20. Cependant, les performances du système ont été dégradées d'un facteur de 2.64 en terme du temps d'exécution du programme et 3.71 en espace mémoire utilisé pour stocker le code du programme.

Afin de valider la méthode de détection dans des environnements radiatifs, nous avons effectué des tests sous radiations avec un dispositif dédié à ce type d'expériences. Les résultats expérimentaux issus dans des essais de radiations mettent en évidence l'efficacité cette approche.

Conclusions et perspectives

Dans cette thèse, nous avons présenté une méthodologie pour la détection d'erreurs induites par des basculements des bits survenant en environnement radiatif sur des architectures digitales à base de processeurs. La méthodologie repose sur des transformations du programme d'application introduisant des redondances aussi bien au niveau des données que dans le code du programme. Cette méthode se base sur un ensemble de règles permettant la transformation automatique d'une application logicielle en une nouvelle possédant des capacités de détection d'erreurs, tout en ayant la même fonctionnalité que l'application originale.

L'ensemble de règles est issu d'une analyse approfondie de celles d'une méthode existante, ceci dans le but d'améliorer la capacité de détection (réduction de nombre de fautes qui échappent au mécanisme de détection adopté) tout en minimisant le temps d'exécution et l'occupation mémoire du programme transformé.

L'ensemble de règles proposées pour la détection d'erreurs est automatiquement mis en application à l'aide d'un outil comme une phase de pré-compilation, devenant complètement transparente au programmeur. Ceci entraîne la réduction du coût de développement des programmes durcis.

Une contribution significative des recherches entreprises dans cette thèse s'est traduite par la validation de la qualité de détection fournie par notre méthode dans le cadre de différentes séances d'injection de fautes et finalement par des essais sous radiation effectués au sein de plusieurs installations dédiées au tests d'upsets disponibles au sein d'accélérateurs de particules en Europe et aux Etats-Unis. Pour ce faire, nous avons appliquée la méthode proposée à des programmes développés pour processeurs différents : le microcontrôleur 80C51 d'Intel et le processeur numérique de traitement de signal DSP32C, ceci en nous servant du testeur THESIC développé à TIMA en collaboration avec le CNES.

Une autre contribution de cette thèse a consisté en la définition d'un cadre conceptuel pour l'estimation du coût de temps d'exécution introduit par le durcissement de l'application.

Les résultats présentés dans cette thèse sont très encourageants, et ouvrent une nouvelle voie dans le domaine de la détection d'erreurs de type upset, survenant dans des architectures digitales destinées à des projets spatiaux. La technique de détection proposée, a été appliquée à différents processeurs. La corrélation entre l'efficacité de détection estimée par injection de fautes et l'efficacité mesurée pour les programmes durcis a été réalisée au cours de plusieurs campagnes d'essais sous radiations.

Les travaux futurs à court terme consisteront en l'extension de l'application de cette technique à d'autres processeurs complexes. L'objectif est la conception de logiciels capables d'être exécutés de manière fiable dans des environnements radioactifs. Ces logiciels pourront être utilisés à bord des véhicules spatiaux. Les travaux futurs à long terme consisteront en l'étude d'une méthodologie mixte logicielle/matérielle pour la détection d'erreurs. En se basant sur la recherche effectuée dans cette thèse, ces travaux se proposent d'exploiter les points forts des approches logicielles et matérielles ainsi que d'améliorer leurs points faibles par leur assemblage dans une même méthodologie.

Références

[1]. E. Normand, "*Single Event in Avionics*", IEEE Trans. On Nuclear Science, vol. 43, n°2, pp. 461-474, April 1996

[2]. A. Moran, K. LaBel, M. Gates, C. Seidleck, R. McGraw, M. Broida, J. Firer, S. Sprehn, "*Singe Event Effect Testing of the Intel 80368 Family and 80486 Microprocessor*", Proceeding of Second European Conference on Radiation and its Effects on Component and Systems (RADECS'95), Arcachon, France, pp. 263-269, 1995

[3]. T. P. Ma, P. Dussendorfer, "Ionizing Radiation Effects in MOS Devices and Circuits", Wiley, New-York, 1989

[4]. E. L. Peterson, "*Single event upsets in space: basic concepts*", Tutorial short course, IEEE Nuclear and Space Radiation Effects Conference, Jul. 1983.

[5]. R. Koga, et al., "Heavy ion induced single event upsets of microcircuits; a summary of aerospace corporation test data", IEEE Trans. Nuclear. Science, Vol. 31, No. 6, p.1 190, Dec. 1984.

[6]. J. C. Pickel, "*Single event upset mechanisms and predictions, Tutorial short course*", IEEE Nuclear and Space Radiation Effects Conference, Jul. 1983.

[7]. E. L. Petersen, "*Single event upsets in space: basic concepts*", Tutorial short course, IEEE Nuclear and Space Radiation Effects Conference, July 1983

[8]. D. K. Nichols, et al., "*A summary of JPL single event upset test data from 1982, through 1984*", IEEE Trans. Nuclear Science., Vol. 32, N'6, p. 1186, Dec. 1985.

[9]. S. Buchner, M. Olmos, R. Velazco, Ph. Cheynet, J. Mellinger, "*Pulsed LASER Validation of Recovery Mechanism of Critical SEE's in Artificial Network System*", Proceeding. of Fourth European Conference on Radiation and its Effects on Component and Systems (RADECS'97), Cannes, France, pp. 110-111, 1997

[10]. A. Moran, K. LaBel, M. Gates, C. Seidleck, R. McGraw, M. Broida, J. Firer, S. Sprehn, "*Single Event Effect Testing of the Intel 80368 Family and the 80468 Microprocessor*", Proceeding. of Second Fourth European Conference on Radiation and its Effects on Component and Systems (RADECS'95), Arcachon, France, pp. 263-269, 1995

[11]. D. Bessot, "Radiaion Hardening Technics Facing Total Dose, SEU and SEL in the space Environment", rapport technique No. B465, Oxon, UK, Harwell laboratory, 1993

[12]. B. Nicolescu, R. Velazco, M. Sonza Reorda, "Effectiveness and limitations of various software techniques for "soft error" detection: A comparative study", 7th IEEE International On-Line Testing Workshop (IOLTW 2001), Taormina, Italy, July 9-11, 2001

[13]. R. Velazco, D. Bessot, R. Eccofet, S. Duzellier, "Two CMOS memory cells suitable for the design of SEU tolerant VLSI circuits", IEEE Transactions on Nuclear Science, Vol. 6, n° 41, 1994, pp. 2229-2234.

[14]. L. Rockett, "An SEU Hardened CMOS Data Latch Design", IEEE Trans. On Nuclear Science, Vol. 35, No.6, Dec. 1988

[15]. M. N. Liu and S Witaker, "*Low Power SEU Immune CMOS Memory Circuits*", IEEE Trans. On Nuclear Science, Vol. 39, No.6, Dec. 1992, pp. 1679 – 1684

[16]. M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies", VTS'99: IEEE VLSI Test Symposium, 1999, pp. 86-94

[17]. Martin Hiller, "Software Fault Tolerant Techniques from a Real-Time System Point of View", November 1998

[18]. K. H. Huang, J. A. Abraham, "*Algorithm-Based Fault Tolerance for Matrix Operations*", IEEE Trans. on Computers, vol. 33, December 1984, pp. 518-528

[19]. Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "*Design and Evaluation of System-level Checks for On-line Control Flow Error Detection*", IEEE Trans. On Parallel and Distributed Systems, Vol. 10, No. 6, Jun. 1999, pp. 627-641

[20]. A. Mahmood, E. J. McCluskey, "*Concurent Error Detection Using Watchdog processor –A Survay*", IEEE Transaction on Computer, Vol. 37, No. 2, pp. 160-174, 1998

[21]. Shambhu Upaddhyaya, Bina Ramaniurthy, "*Concurrent Process Monitoring with No Reference Signatures*", IEEE Transaction on Computer, Vol. 43 no. 4, pp. 475-480, April 1994

[22]. G. Miremadi, J. OhIsson, M. Rimen, J. Karlsson, *"Use of Tune and Address Signatures for Control Flow Checking",* 5th IFIP Working Conference on Dependable Computing for Critical Appleation (DCCA-5), pp. 113-124, 1995

[23]. S. S. Yau, F. Ch. Chen, *"An Approach to Concurrent Control Flow Checking"*, IEEE Transaction on Software Engineering, Vol. SE-6, No. 2, pp. 126-137, 1980

[24]. R. Leveugle, T Michel, G. Saucier, "*Design of Microprocessors with Built-In On-Line test*", 210th International Symposium on Fault-Tolerant Computing (FrCS-20), pp. 450-456, 1990

[25]. M. A. Schutte, J. P. Shen, D. P. Siewiorek, Y. X. Zbu, "*Experimental Evaluation of Two Concurrent Error Detection Schemes*", 16th International Symposium on Fault Tolerant Computing (FTCS- 16), pp. 138-143, 1986

[26]. K. Wilken, J. P. Shen. "*Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Errors*", IEEE Transaction on Computer Aided Design and Systems, Vol. 9, Issue 6, pp. 629-641, June 1990.

[27]. T. Michel, R. Leveugle, G. Saucier, "*A New Approach to Control Flow Checking without Program Modification*", 21st International Symposium on Fault Tolerant Computing (FTCS-21), pp. 334-341, 1991

[28]. A. V. Aho, R. Sethi, J. D. Ullman, *"Compilers: Principles, Techniques and Tools"*, Addison-Wesley, 1986

[29]. A. Benso, S. Di Carlo, .G. Di Natale, P. Prineto, L. Tagliaferri, "*Control-Flow Checking Via regular Expressions*", IOLTW 02,

[30]. J. J. Horhing, H. C. Lauer, P. M. Melliar-Smith, B. Randell, "*A Program Structure for Error Detection and Recovery*", Lecture Notes in Computer Science, Vol. 16, pp. 172-187, 1974

[31]. B. Randell, "*System Structure for Software Fault Tolerant*", IEEE Trans. On Software Engineering, Vol. 1, No. 2, Jun. 1975, pp. 220-232

[32]. M. Hech, "*Folt Tolerant Software for Real-Time Applications*", ACM Computing Surveys, Vol. 8, No. 4, pp. 391-407, December 1967

[33]. A. Avizienis, L. Chen, "*On the Implementation Of N-Version Programming for Software Fault Tolerant During Program Execution*", Proceedings of 1977 International Conference on Computer Software and Application, pp. 149-155, 1977

[34]. M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, *"Soft-error Detection through Software Fault-Tolerance techniques*", DFT'99: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Austin (USA), November 1999, pp. 210-218

[35]. P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, "*Hardening the software with respect to transient errors: a method and experimental results*", 1st IEEE Latin-American Test Workshop (LATW 2000), Rio de Janeiro, Brazil, March 13-15, 2000

[36]. M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, R. Velazco, "*System safety through automatic high-level code transformations: an experimental evaluation*", Design Automation and Testing in Europe (DATE 2001), Munich, Germany, March 13-16, 2001

[37]. A. Hilmes-Siedlea and L. Adams, "*Handbook of Radiation Effects*", Oxford University Press 1993

[38]. The 8051 Family of Microcontroller, Richard H. Barned, Prentice-Hall, 1995

[39]. S. Rezgui, "Prediction du Taux d'erreurs d'Architectures Digitale: Une Méthode et Résultats experimentaux", January 2000

[40]. Velazco R., Cheynet Ph., Bofill A., Ecoffet R., "*THESIC: A testbed suitable for the qualification of integrated circuits devoted to operate in harsh environment*", IEEE European Test Workshop (ETW'98), Spain, 27-29 May 1998 , IEEE , 1998

[41]. LFSR, http://www-s.ti.com/sc/psheets/scta036a/scta036a.pdf

[42]. S. Haykin, et al., "*Unsupervised Adaptive Filtering*", vols. 1-2; John Wiley and Sons Inc, USA, 2000

[43]. WE DSP32C Digital Signal Processor. Information Manual

[44]. M. Matsumoto and T. Nishimura, Mersenne Twister, "*A 623-dimensionally equidistributed uniform pseudo-random number generator*", ACM Transaction on Modeling and Computer Simulation Vol. 8, No. 1, January pp. 3-30, 1998

Annexe A

Outil de génération automatique des programmes durcis : le translateur C2C

A. Vu d'ensemble

Le translateur C2C est un outil de logiciel développé pour fournir de détection d'erreurs dans des applications fonctionnant dans un environnement sévère. L'objectif principal est de transformer un programme dans un autre ayant la même fonctionnalité que l'original. Le programme obtenu a la même fonctionnalité que celle du programme original et en plus il est capable de détecter des basculement de bits affectant le contenu de la mémoire de données, le code du programme ou le processeur (les registres généraux, les registres de contrôle, la pile, la mémoire cache, etc.). Le translateur C2C accepte en entrée un programme décrit dans le langage C et en sortie donne le code de source C correspondant à un programme avec des possibilités de détection d'erreurs. La Figure A.1 montre flot de conception du translateur C2C.

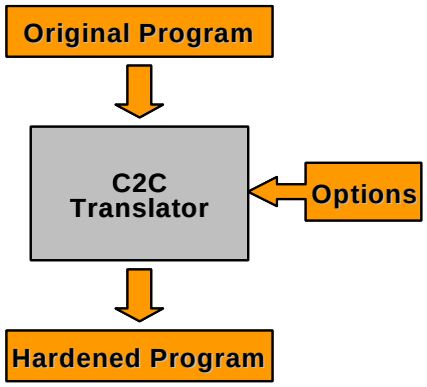


Figure A.1 : Structure du translateur C2C

B. Utilisation du translateur C2C

Utilisant le translateur C2C, l'utilisateur a huit options tenant compte des combinaisons des groupes des règles de détection d'erreurs en incluant toutes les sous-ensemble des règles proposées.

La commande *`C2C inputfile [-options]`* appelle l'exécution du translateur C2C où :

- `inputfile` : représente le code source d'entrée (le programme non durci)

§ `options` : représente les options de durcissement proposé

Les options sont :

§ -o : établi le nome du programme durci désiré par l'utilisateur

§ -d : introduit durcissement au niveau de données

§ -i: permet l'application de la règle #4 pour toutes les variables

§ -c : introduit durcissement au niveau du flot global d'exécution

§ -n : établi le nombre d'instructions de données composant un bloc élémentaire

§ -t : introduit durcissement au niveau de branchement

§ -a : tous les règles de durcissement sont insérées dans le programme de sortie

L'utilisateur peut appeler la commande d'aide en écrivant la commande de «*help*». La Figure A.2 montre une fenêtre d'aide du translateur C2C.

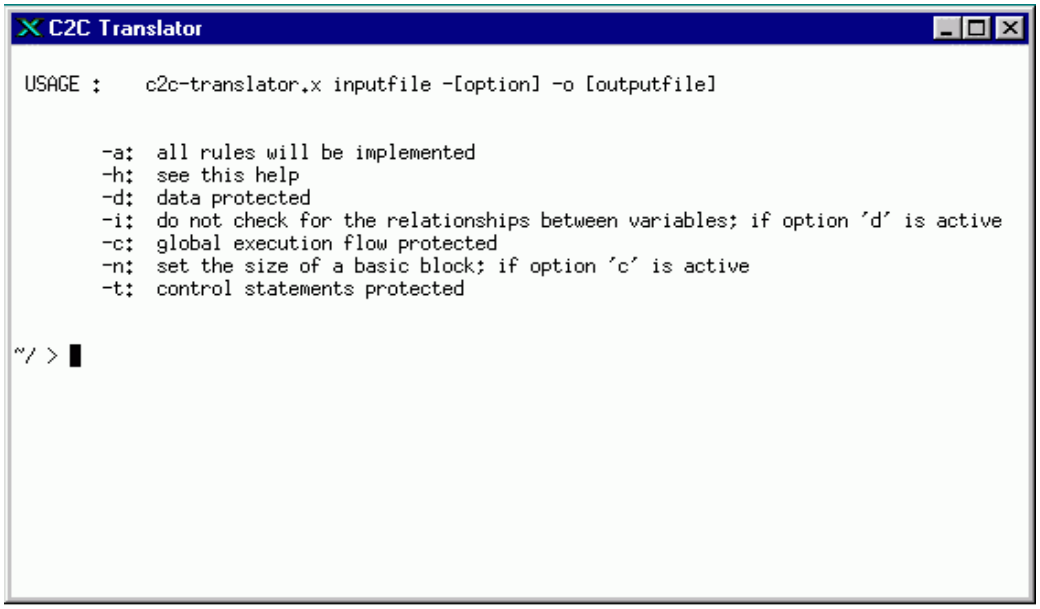


Figure A.2 : Commande de «*help*» du translateur C2C

C. Création d'un exemple

C.1. Description du programme ciblé

Dans cette section nous illustrons l'application de chaque groupe de règles sur un programme industriel utilisé dans le domaine de communication spatiale. Ce programme est analysé en détailles pour donné une vue d'ensemble sur sa fonctionnalité, il est donné dans la Figure A.3.

Il est important de noter que la fonction **`error()`** n'est pas générée par le translateur C2C, elle doit être crée par l'utilisateur. Cela pour permette au l'utilisateur de géré en fonction de ses besoins le traitement d'erreur.

La structure du programme est composée des deux fonctions : la fonction principale ***main*** et la fonction *cma_equalizer* qui prend en charge le calcul des paramètres utilisés par un égalisateur de signal numérique.

Chaque fonction du programme est composée par trois instructions contrôle (deux sont instruction conditionnelles de contrôle et une instruction inconditionnelle de contrôle) et 6 blocs élémentaires (de taille maximale).

L'analyse de l'interdépendance entre les variables du programme permette de classifier les variables en deux catégories :

- Variables intermédiaires :
  - *input_data*, *CC* (variables globales)
  - *UU*, *i*, *j* (variables locales définis dans la fonction ***main***)
  - *y*, *Eb*, *i* (variables locales définis dans la fonction *cma_equalizer*)
- Variables *finales* :
  - *YY* (variable locale défini dans la fonction ***main***)

Nous rappelons que une variable *finale* n'intervienne pas dans le calcul d'une autre variable du programme.

```
/* déclaration variables globale */

#define      FILTERSIZE         11
#define      No_SAMPLES         133
#define      R2                 1           /* CMA constant */
#define      step_adapt         0.0001      /* CMA equalizer step size */

float input_data[No_SAMPLES];                /* input data */
float CC[FILTERSIZE]={0,0,0,0,0,1,0,0,0,0,0};/* CMA equalizer coefficients */

float cma_equalizer(float *U);
```

```
/* fonction principale main */

void main(void)
{
	float UU[FILTERSIZE];			/* CMA equalizer input */
	float YY;				/* CMA equalizer output */
	int i;
	int j;

	j = 0;
	i = 0;
	while(j < No_SAMPLES)
	{
		i = 0;
		while(i < FILTERSIZE)
		{
			UU [i] = input_data [i + j];
			i ++ ;
		}
		j ++;
		YY = cma_equalizer(UU);
	}
}
```

```
/* fonction cma_equalizer */

float cma_equalizer(float *U)
{
	float y;				/* CMA output equalizer */
	float Eb;				/* CMA output equalizer divergence */
	int i;

	i = 0;
	while(i < FILTERSIZE)
	{
		y = y + U [i] * CC [i];
		i ++;
	}
	Eb = y *(R2 – y * y);
	i = 0;
	while(i < FILTERSIZE)
	{
		CC[i] = CC [i] + step_adapt * Eb * U [i];
		i ++;
	}
	return y;
}
```

Figure A.3 : Programme non durci

C.2. Application du groupe de règles « Duplication de Données »

L'application de ce groupe de règles duplique toutes les variables définies dans le programme et toutes les opérations effectuées sur les variables. Ensuite, une vérification d'égalité entre les valeurs des deux variables (***originale*** et ***dupliquée***) est effectuée après chaque opération d'écriture sur les variables ***finales.*** Le code résultant suite à l'application de ces règles est montré dans la Figure A.4.

Les modifications apparues dans le programme sont :

- Duplication des toutes les variables et des toutes les opérations effectuées sur les variables ;
- Le prototype de la fonction ***cma_equalizer***, a été changé ; il devient de type ***void*** alors que initialement a été de type ***float*** ;
- Les variables de retour de la fonction `cma_equalizer` sont déclarées comme des variables globales puisque le prototype de la fonction `cma_equalizer` a été modifié
- Une vérification de la cohérence d'égalité est introduite antre *YY_1* et *YY_2* (les deux variables ***finales)***

```
/* déclaration variables globale */

#define      FILTERSIZE          11
#define      No_SAMPLES          133
#define      R2                  1              /* CMA constant */
#define      step_adapt          0.0001         /* CMA equalizer step size */

float input_data_1[No_SAMPLES];                 /* input data */
float input_data_2[No_SAMPLES];                 /* input data */

float CC_1 [FILTERSIZE]={0,0,0,0,0,1,0,0,0,0,0};/* CMA equalizer coefficients */
float CC_2 [FILTERSIZE]={0,0,0,0,0,1,0,0,0,0,0};/* CMA equalizer coefficients */

float y_1;
float y_2;

void cma_equalizer(float *U_1, float *U_2);
```

```
/* fonction principale main */

void main(void)
{
	float UU_1[FILTERSIZE];			/* CMA equalizer input */
	float UU_2[FILTERSIZE];			/* CMA equalizer input */

	float YY_1;				/* CMA equalizer output */
	float YY_2;				/* CMA equalizer output */
	int i_1;
	int i_2;

	int j_1;
	int j_2;

	j_1 = 0;
	j_2 = 0;

	i_1 = 0;
	i_2 = 0;

	while(j_1 < No_SAMPLES && j_2 < No_SAMPLES)
	{
		i_1 = 0;
		i_2 = 0;

		while(i_1 < FILTERSIZE && i_2 < FILTERSIZE)
		{
			UU_1 [i_1] = input_data_1 [i_1 + j_1];
			UU_2 [i_2] = input_data_2 [i_2 + j_2];

			i_1 ++;
			i_2 ++;
		}
		j_1 ++;
		j_2 ++;

		cma_equalizer(UU_1,UU_2);

		YY_1 = y_1;
		YY_2 = y_2;
		if(YY_1 != YY2)
			error();
	}
}
```

```
/* fonction cma_equalizer */

void cma_equalizer(float *U_1, float *U_2)
{
	float Eb_1;			/* CMA output equalizer divergence */
	float Eb_2;			/* CMA output equalizer divergence */
	int i_1;
	int i_2;

	i_1 = 0;
	i_2 = 0;

	while(i_2 < FILTERSIZE && i_2 < FILTERSIZE)
	{
		y_1 = y_1 + U_1 [i_1] * CC_1 [i_1];
		y_2 = y_2 + U_2 [i_2] * CC_2 [i_2];

		i_1 ++;
		i_2 ++;
	}

	Eb_1 = y_1 *(R2 - y_1 * y_1);
	Eb_2 = y_2 *(R2 - y_2 * y_2);

	i_1 = 0;
	i_2 = 0;

	while(i_1 < FILTERSIZE && i_2 < FILTERSIZE)
	{
		CC_1 [i_1] = CC_1 [i_1] + step_adapt * Eb_1 * U_1 [i_1];
		CC_2 [i_2] = CC_2 [i_2] + step_adapt * Eb_2 * U_2 [i_2];

		i_1 ++;
		i_2 ++;
	}

	return;
}
```

Figure A.4 : Application du groupe de règles Duplication de Données

C.3. Application du groupe de règles « Flot Global d'Exécution »

Ce groupe de règles interviendra sur les instructions élémentaires d'affectation des données. Elles composent les blocs élémentaires du programme. Le code du programme est décomposé en blocs de taille maximale. Ensuite, les instructions composent les blocs de taille maximale sont analysées et les blocs de taille maximales sont structurés dans des blocs élémentaires en fonction de nombre et le type d'instructions qui composent le bloc maximale. Deux variables globales sont définies afin de permettre le contrôle du flot d'exécution du programme. Elles sont :

- fge : définie pour associé à chaque bloc élémentaire une identification
- status_bloc : définie pour décrire l'état du chaque bloc

L'application de ce groupe de règles est illustrée dans la Figure A.5.

```
/* déclaration variables globale */

#define      FILTERSIZE          11
#define      No_SAMPLES          133
#define      R2                  1            /* CMA constant */
#define      step_adapt          0.0001       /* CMA equalizer step size */

float input_data[No_SAMPLES];                 /* input data */
float CC[FILTERSIZE]={0,0,0,0,0,1,0,0,0,0,0}; /* CMA equalizer coefficients */

float cma_equalizer(float *U) ;

int fge;
int status_bloc = 1;
```

```
/* fonction main */

void main(void)
{
	float UU[FILTERSIZE];		/* CMA equalizer input */
	float YY;			/* CMA equalizer output */
	int i;
	int j;

	fge = 1 ^(status_bloc ^= 1);
	j = 0;
	i = 0;
	status_bloc ^= 1;
	if(fge != 1) error();

	while(1)
	{
		fge = 2 ^(status_bloc ^= 1);
		if(!(j < No_SAMPLES))
		{
			status_bloc ^= 1;
			if(fge != 2) error();
			break;
		}
		status_bloc ^= 1;
		if(fge != 2) error();

		fge = 3 ^(status_bloc ^= 1);
		i = 0;
		status_bloc ^= 1;
		if(fge != 3) error();

		while(1)
		{
			fge = 4 ^(status_bloc ^= 1);
			if(i < FILTERSIZE)
			{
				status_bloc ^= 1;
				if(fge != 4) error();
				break;
			}
			status_bloc ^= 1;
			if(fge != 4) error();

			fge = 5 ^(status_bloc ^= 1);
			UU [i] = input_data [i + j];
			i++ ;
			status_bloc ^= 1;
			if(fge != 5) error();

		}

		fge = 6 ^(status_bloc ^= 1);
		j ++;
		status_bloc ^= 1;
		if(fge != 6) error();

		YY = cma_equalizer(UU);
	}
}
```

```
/* fonction cma_equalizer */

float cma_equalizer(float *U)
{
	float y;			/* CMA output equalizer */
	float Eb;			/* CMA output equalizer divergence */
	int i;

	fge = 7 ^(status_bloc ^= 1);
	i = 0;
	status_bloc ^= 1;
	if(fge != 7) error();

	while(1)
	{
		fge = 8 ^(status_bloc ^= 1);
		if(i < FILTERSIZE)
		{
			status_bloc ^= 1;
			if(fge != 8) error();
			break;
		}
		status_bloc ^= 1;
		if(fge != 8) error();

		fge = 9 ^(status_bloc ^= 1);
		y = y + U [i] * CC [i];
		i ++;
		status_bloc ^= 1;
		if(fge != 9) error();
	}

	fge = 10 ^(status_bloc ^= 1);
	Eb = y *(R2 – y * y);
	i = 0;
	status_bloc ^= 1;
	if(fge != 10) error();

	while(1)
	{
		fge = 11 ^(status_bloc ^= 1);
		if(i < FILTERSIZE)
		{
			status_bloc ^= 1;
			if(fge != 11) error();
			break;
		}
		status_bloc ^= 1;
		if(fge != 11) error();

		fge = 12 ^(status_bloc ^= 1);
		CC [i] = CC [i] + step_adapt *Eb * U [i];
		i ++;
		status_bloc ^= 1;
		if(fge != 12) error();
	}

	return y;
}
```

Figure A.5 : Application du groupe de règles Flot Global d'Exécution

C.4. Application du groupe de règles « Duplication de Branchement »

Ce groupe de règles intervient sur les instructions de contrôle. L'application de ces règles est illustre dans la Figure A.6.

```
/* déclaration variables globale */

#define      FILTERSIZE          11
#define      No_SAMPLES          133
#define      R2                  1            /* CMA constant */
#define      step_adapt          0.0001       /* CMA equalizer step size */

float input_data[No_SAMPLES];                 /* input data */
float CC[FILTERSIZE]={0,0,0,0,0,1,0,0,0,0,0}; /* CMA equalizer coefficients */

float cma_equalizer(float *U) ;

int carf;
```

```
/* fonction main */
void main(void)
{
      float UU[FILTERSIZE];           /* CMA equalizer input */
      float YY;                       /* CMA equalizer output */
      int i;
      int j;

      carf = -1;
      j = 0;
      i = 0;
      while(j < No_SAMPLES)
      {
            if(!(j < No_SAMPLES))error();

            i = 0;
            while(i < FILTERSIZE)
            {
                  if(!(i < FILTERSIZE))error();

                  UU [i] = input_data [i + j];
                  i ++ ;
            }
            j ++;

            if(carf != -1)error();
            YY = cma_equalizer(UU);
            if(carf != 1) error();
            carf = -1;
      }
}
```

```
/* fonction cma_equalizer */

float cma_equalizer(float *U)
{
       float y;                          /* CMA output equalizer */
       float Eb;                         /* CMA output equalizer divergence */
       int i;

       carf = 1;
       i = 0;
       while(i < FILTERSIZE)
       {
              if(!(j < No_SAMPLES))error();

              y = y + U [i] * CC [i];
              i ++;
       }
       Eb = y *(R2 – y * y);
       i = 0;
       while(i < FILTERSIZE)
       {
              if(!(j < No_SAMPLES))error();

              CC[i] = CC [i] + step_adapt * Eb * U [i];
              i ++;
       }
       return y;
}
```

Figure A.6 : Application du groupe de règles Duplication de Branchements

Liste des publications

[1]. B. Nicolescu, A. Corominas, R. Velazco, "An automated technique to provide software applications with SEU detection capabilities: basic principles and preliminary results", Radiation and its effects on Components and Systems (RADECS 2002), Padova, Italy, September 19-20, 2002

[2]. B. Nicolescu, R. Velazco, M. Sonza Reorda, M. Rebaudengo, M. Violante, "*A Software Fault Tolerance Method for Safety-Critical Systems: Effectiveness and Drawbacks*", 15th Symposium On Integrated Circuits And System Design (SBCCI 2002), Porto Alegre, RS, Brazil, September 9-14, 2002

[3]. M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, R. Velazco, "*Error detection and correction by software means: a new method and preliminary experimental results*", Radiation and its effects on Components and Systems (RADECS 2002), Padova, Italy, September 19-20, 2002

[4]. S. Duzellier, R. Velazco, B. Nicolescu, S. Bourdarie, R. Ecoffet, "*SEE In-Flight Data for two Static 32KB Memories on High Earth Orbit*", IEEE Nuclear and Space Radiation Effects Conference (NSREC 2002), Phoenix, Arizona , July 15-19, 2002

[5]. M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, R. Velazco, "*Coping with SEUs/SETs in Microprocessors by means of Low-Cost Solutions: A comparative study and Experimental results*", IEEE Transaction On Nuclear Science, Vol. 49, N° 3, June 2002

[6]. B. Nicolescu, R. Velazco, M. Sonza Reorda, "*Effectiveness and limitations of various software techniques for "soft error" detection: A comparative study*", 7th IEEE International On-Line Testing Workshop (IOLTW 2001), Taormina, Italy, July 9-11, 2001

[7]. M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, R. Velazco, "*System safety through automatic high-level code transformations: an experimental evaluation*", Design Automation and Testing in Europe (DATE 2001), Munich, Germany, March 13-16, 2001

[8]. P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, "*Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors*", IEEE Transactions On Nuclear Science, Vol. 47, No.6, December 2000

[9]. C. Godin, M. Gordon, B. Nicolescu, P. Cheynet, R. Velazco, "*Robustness of Neural Network in Radioactive Environment*", 8th International Conference on Microelectronics for Neural, Fuzzy and Bio-inspired Systems Grenoble, France, September 25-27, 2000

[10]. M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Cheynet, B. Nicolescu, R. Velazco, "*Evaluating the effectiveness of a software fault-tolerance technique on RISC and CISC-based architectures*", 6th IEEE International On-Line Testing Workshop (IOLTW 2000), Mallorca, Spain, July 3-5, 2000

[11]. P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, "*Hardening the software with respect to transient errors: a method and experimental results*", 1st IEEE Latin-American Test Workshop (LATW 2000), Rio de Janeiro, Brazil, March 13-15, 2000

RESUME

Cette thèse est consacrée à l'étude d'une méthodologie logicielle pour la détection d'erreurs induites par l'environnement radiatif : le phénomène dit SEU ou upset qui se traduit par le basculement intempestif du contenu d'un élément mémoire comme conséquence de l'ionisation produite par le passage d'une particule chargée avec le matériel. Les conséquences de ce phénomène dépendent de l'instant d'occurrence et de l'élément mémoire affecté et peuvent aller de la une simple erreur de résultat à la perte de contrôle d'un engin spatial.

La méthodologie repose sur des transformations du programme d'application introduisant des redondances aussi bien au niveau des données que dans le code du programme. Cette méthodologie est basée sur un ensemble de règles permettant la transformation automatique d'une application logicielle en une nouvelle possédant des capacités de détection d'erreurs de type SEU, tout en ayant la même fonctionnalité que l'application originale. L'ensemble de règles est issu d'une analyse approfondie dans le but d'améliorer la capacité de détection (réduction de nombre de fautes qui échappent au mécanisme de détection adopté) tout en minimisant le temps d'exécution et l'occupation mémoire du programme transformé. Cette méthodologie a constitué le cadre conceptuel pour la construction d'un outil de génération automatique des programmes tolérants aux erreurs induits par l'environnement radiatif.

L'évaluation de cette méthodologie a été effectuée par des expériences d'injection de fautes et des essais de radiations sur plusieurs processeurs. Ces expérimentations ont confirmé nos attentes : la version tolérante aux fautes d'une application, permet la détection en moyenne de 88% des erreurs survenues.

En terme de perspectives, ce travail de recherche constituera la base pour la définition d'une méthodologie logicielle/matérielle pour la détection d'erreurs.

TITRE EN ANGLAIS:

A SOFTWARE APPROACH FOR DETECTION OF TRANSIENT ERRORS OCCURRING IN PROCESSOR-BASED DIGITAL ARCHITECTURES: PRINCIPLES AND EXPERIMENTAL RESULTS

ABSTRACT

This thesis is devoted to the study of a software methodology for detection of the errors induced by the radioactive environment: the SEU phenomenon, also called upset - which may modify the content of memory elements as the result of the silicon ionization resulting from the impact of a charged particles. The consequences of the upsets for a given application depend on both the occurrence instant and the perturbed memory element, and can go from erroneous results to system crashes which may provoke the loose of the control of a space vehicle.

The proposed software approach is based on the transformation of the programs, written in any high-level language in such a way they have capabilities to detect transient errors affecting data and code. The software modifications are achieved through the application of a set of transformation rules derived from a through analysis of an existing set of rules formerly proposed in the specialized literature. The new set of rules improved the system performances in terms of reducing the number of errors escaping the detection mechanism and the program execution time.

The evaluation of this methodology was carried out by fault injection experiments and the radiation testing campaigns for several processors. These experiments confirmed our expectations: the hardened version of the application allows a high error detection rate (average of 88% of the errors which have occurred in the system).

Future works will constitute the base for the definition of a mixed technique software/hardware for the error detection.

Spécialité: MICROELECTRONIQUE

MOTS CLES : Environnement spatial, Single Event Upset, bit-flip, basculement des bits, efficacité de détection, programme durci, injection de fautes, architectures digitales.

Laboratoire **TIMA**, Techniques de l'Informatique et de la Micro-électronique pour l'Architecture des ordinateurs, 46 Avenue Félix Viallet, 38031 Grenoble

ISBN 2-84813-004-0 Broché
ISBN 2-84813-005-9 Format Electronique