UNIVERSITÉ DU QUÉBEC

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INGÉNIERIE

PAR

NICOLAS GAGNON B .Ing.

DÉVELOPPEMENT ET ÉTUDE D'UN SYSTÈME D'EXPLOITATION TOLÉRANT AUX DÉFAILLANCES POUR SYSTÈME UN MULTIPROCESSEUR

DÉCEMBRE 1997

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Remerciements

Je désire remercier tous ceux qui ont aidé de près ou de loin à la réalisation de ce mémoire. Je tiens à remercier tout particulièrement M. Daniel Audet, mon directeur de recherche pour le support qu'il m'a apporté tout au long de mon mémoire. Je tiens à remercier monsieur Dany Ouellet qui ma fourni les ressources matérielles nécessaires au développement du système d'exploitation ainsi qu'à l'accomplissement des tests expérimentaux. Je remercie également monsieur Nicolas Arel qui ma beaucoup aidé lors du développement du système d'exploitation. Je remercie Mme Chantale Dumas qui a été d'un grand support à la fin de mon mémoire. Finalement, je désire remercier ma conjointe Isabelle Jean qui ma supporté lors de la rédaction du mémoire.

Résumé

Ce projet fut développé dans le cadre d'un contrat de recherche pour le ministère de la défense nationale du Canada. Ce projet porte sur le développement d'un système d'exploitation tolérant aux défaillances pour un système multiprocesseur. Il comporte également une étude statistique sur le comportement du système d'exploitation ainsi que sur les systèmes multiprocesseurs lorsqu'ils sont soumis à des injections de pannes. Le développement de tels systèmes nécessite une connaissance approfondie des divers mécanismes utilisés pour la détection des pannes ainsi que ceux utilisés pour la récupération lorsqu'une panne est détectée.

Table des matières

Liste des figures

Liste des tableaux

CHAPITRE 1

INTRODUCTION

1. PRÉAMBULE

L'utilisation de plus en plus fréquente des ordinateurs pour exécuter des tâches complexes implique que ces derniers doivent être d'une très grande fiabilité afin d'éviter toute panne lors de l'exécution d'une application. Une seule panne dans le système peut devenir fatale pour des applications comme les systèmes d'aide à la navigation, les systèmes nécessaires aux traitements médicaux, les systèmes de télécommunication, les systèmes bancaires et bien d'autres encore. C'est pourquoi les systèmes informatiques d'aujourd'hui font appel à différents mécanismes permettant à ces derniers d'éviter que des pannes puissent affecter leur comportement de façon permanente.

1.1. Introduction sur le projet et ses objectifs.

Le présent projet fut développé dans le cadre d'un contrat pour le ministère de la défense nationale du Canada. Ce projet comprenait deux parties. La première partie consistait à développer deux prototypes d'ordinateurs multiprocesseurs [SAV93]. Elle fut développée par une équipe de recherche de l'École Polytechnique de Montréal. La seconde partie, celle qui concerne le présent document, consistait à développer un système d'exploitation nécessaire au bon fonctionnement des prototypes.

Le développement du système d'exploitation devait répondre à des spécifications précises. Il devait posséder plusieurs modes d'opération. Chacun de ces modes devait posséder des mécanismes de tolérance aux défaillances avec différents niveaux de robustesse. Le système d'exploitation devait également être facilement portable sur d'autres systèmes multiprocesseurs existants sans qu'on ait à effectuer des modifications au niveau matériel. Finalement, le

système d'exploitation devait également être peu coûteux et développé dans un court laps de temps.

Le développement du système d'exploitation a été divisé en deux parties. La première partie consistait à développer le système d'exploitation proprement dit. La seconde partie consistait à effectuer diverses expérimentations sur le comportement du système d'exploitation lorsque soumis à des injections de pannes.

Les expérimentations se sont déroulées en deux phases. La première phase faisait partie intégrante du développement du système d'exploitation. En effet, de nombreuses expérimentations ont été nécessaires avant d'obtenir une version correcte pour chacun des modes d'opération. La seconde phase consistait en une étude statistique préliminaire sur le comportement réel d'un système multiprocesseur lorsque soumis à des injections de pannes. De plus, on désirait déterminer les conditions qui permettaient de diminuer les risques qu'une panne introduite dans un système affecte ce dernier.

1.2. Les mécanismes de tolérance aux défaillances.

On peut parler de tolérance aux défaillances lorsqu'un système subissant des perturbations peut, à l'aide de différents mécanismes, corriger l'effet de ces perturbations sans affecter le résultat final. La tolérance aux défaillances s'effectue normalement en deux étapes. La première étape consiste à détecter les pannes qui ont pu s'introduire dans le système. La seconde étape consiste à entamer des procédures qui permettront la récupération des données affectées par toute panne détectée lors de la première étape.

Dans cette section, on désire donner un bref aperçu des différentes techniques utilisées tant pour la détection de pannes que pour la récupération de ces dernières. Le nombre de ces techniques étant important, on se limitera donc à celles qui ont davantage attiré notre attention. Les mécanismes de détection peuvent être introduits de façon matérielle ou encore de façon logicielle. Une approche matérielle sera privilégiée pour les mécanismes internes à un composant ou encore pour un ensemble restreint de composants étroitement liés. L'approche logicielle permet d'évaluer plus facilement de grands systèmes ainsi que les systèmes qui impliquent plusieurs échanges de données.

a) Mécanismes matériels de détection.

On parle de mécanisme matériel de détection lorsqu'on doit modifier ou ajouter un ou des composants afin de réaliser cette détection. Un exemple de tels mécanismes est l'ajout de bits aux unités de mémoire afin d'effectuer une vérification de la parité de ces derniers, ou encore l'utilisation d'unités de mémoire morte programmable (PROM) pour réaliser la détection de pannes dans les instructions du processeur [MAD90]. Ce denier exemple de détection fait déjà partie intégrante de la majeure partie des microprocesseurs actuellement sur le marché. D'autres mécanismes, qui sont eux aussi largement utilisés, permettent de déterminer si un processeur accède des adresses mémoire invalides ou de déterminer si un processeur tente d'exécuter des instructions provenant d'un segment de mémoire autre que celui réservé à cet effet.

b) Mécanismes logiciels de détection.

On parle de mécanisme logiciel de détection lorsque ce dernier peut être introduit dans un système existant sans qu'aucune modification matérielle n'y soit apportée. Une des caractéristiques principales de cette approche est qu'elle permet de détecter des pannes pouvant survenir dans des parties d'un système ne possédant aucun mécanisme de détection. On prend pour exemple l'utilisation du CRC (Cyclic Redundancy Check) pour la détection de pannes lors de l'envoi et la réception d'un message entre deux ordinateurs. Un autre moyen efficace pour détecter les pannes consiste à comparer les données provenant de deux composants d'un même système ou encore de systèmes différents.

Certains mécanismes utilisent des propriétés intrinsèques aux éléments pour établir une procédure de détection. En effet, l'utilisation d'un temporisateur de type "Watch Dog" permet de déterminer si une procédure à l'intérieur d'un processeur a outrepassé le temps alloué à son exécution. Ce genre de mécanisme est utilisé dans les automates programmables afin de s'assurer que le temps d'exécution du programme n'excédera pas le temps maximum prévu. C'est ce qui permet aux automates programmables d'être des outils puissants pour le contrôle en temps réel.

1.2.2. Mécanismes de récupération de pannes.

L'utilisation de mécanismes de récupération de pannes est la clef de la tolérance aux défaillances. En effet, sans l'utilisation de tels mécanismes, les mécanismes de détection de pannes serviraient uniquement pour des fins de diagnostic. Lorsqu'un système doit être d'une très grande fiabilité et qu'il ne peut subir la moindre interruption avant l'accomplissement de la

tâche qu'il a entreprise, il est important d'avoir un mécanisme qui permet le récupération des pannes détectées. Les paragraphes suivants donneront quelques techniques de récupération de pannes qui ont attirées notre attention.

a) Mécanisme à point d'évaluation et retour arrière.

Un des mécanismes de récupération qui suscite un intérêt marqué est le système à point d'évaluation et retour arrière décrit dans l'article intitulé "Processor- and Memory- Based Checkpoint and Rollback Recovery" [BOW93]. Dans cet article, on explique le principe utilisé pour s'assurer de la qualité du résultat obtenu par un processeur. Ce système peut être utilisé pour un ou plusieurs processeurs. Ce mécanisme fait appel à l'anté-mémoire des processeurs. Afin de bien saisir le fonctionnement de ce mécanisme, il est important de rappeler que l'écriture dans la mémoire principale d'un processeur qui possède une anté-mémoire s'effectue uniquement lorsqu'une requête du processeur ne peut être obtenue et que l'anté-mémoire est saturée. Il est également possible de forcer l'anté-mémoire à sauvegarder toutes les modifications qui y ont été effectuées par le processeur.

Le principe de fonctionnement est simple et surtout très efficace. En effet, lorsqu'une panne est détectée, un mécanisme de retour en arrière est exécuté. Ce dernier consiste à replacer le processeur dans l'état où il était à son dernier point d'évaluation. On doit également effacer tous les témoins qui indiquent que des modifications ont été effectuées. L'établissement d'un point d'évaluation consiste à sauvegarder l'état de tous les registres du processeur ainsi que toutes les modifications apportées par le processeur et contenues dans l'anté-mémoire. Un point d'évaluation est effectué lorsqu'il y a un échec lors de la recherche

d'un élément dans l'anté-mémoire et que cette dernière doit écrire dans la mémoire principale pour libérer de l'espace. Cet échec a pour effet de forcer le processeur à enregistrer l'état de chacun de ses registres dans une zone mémoire sécuritaire de la mémoire principale ainsi que de sauvegarder toutes les modifications qui ont été apportées au contenu de l'anté-mémoire dans la mémoire principale.

b) Mécanismes à éléments redondants.

L'utilisation d'éléments redondants augmente de façon significative la fiabilité des systèmes. Ce principe est couramment utilisé dans les applications qui ne laissent pas de place pour l'erreur.

Un exemple de système redondant est le projet Delta-4 [POW94] qui permet à un système distribué hétérogène d'être tolérant aux défaillances. Normalement ce type de système est considéré comme étant peu fiable puisqu'à tout moment l'une des stations qui le compose peut s'arrêter. En effet, plusieurs facteurs peuvent affecter les diverses composantes d'un tel réseau comme une panne dans le système d'exploitation, la fermeture d'une des stations par un usager et bien d'autres encore. Le principe de base de Delta-4 s'appuie sur la redondance d'information à travers le réseau, évitant ainsi qu'une seule station soit en charge d'une partie d'un programme donné. Cependant, le simple fait d'avoir une redondance de l'information n'était pas suffisant pour assurer une grande fiabilité du réseau. Afin d'augmenter sa fiabilité, les responsables du projet ont dû implanter certains mécanismes de protection supplémentaires, limitant ainsi les chances d'interruption d'une station. De plus, ils ont fait en sorte qu'un utilisateur ne puisse pas réinitialiser une station. Seul l'administrateur du réseau

peut effectuer ce genre d'opération. Finalement, ils ont créé une nouvelle interface d'accès réseau permettant d'éviter la saturation de ce dernier lorsqu'une panne survient dans l'une des stations.

D'autres méthodes bien connues comme le TMR (Triple Modular Redundancy) [AND81], le NMR (N Modular Redundancy) [AND81] et le " Roving spare " [SAV91] utilisent également la redondance pour assurer une très grand fiabilité aux systèmes. Ces méthodes ne seront pas abordées dans ce chapitre. Cependant, elles seront les fondations du système d'exploitation développé au cours de ce projet.

1.2.3. Mécanismes retenus.

Etant donné le temps de développement restreint ainsi que certaines spécifications du projet, il fut déterminé que le mécanisme de détection de pannes utiliserait une approche logicielle plutôt que matérielle. Une des raisons principales de ce choix est que le système d'exploitation devait être portable. En effet, toute utilisation de mécanismes matériels aurait eu pour effet de limiter ce système d'exploitation à un seul système.

Pour des raisons similaires, on a préféré une technique utilisant des processeurs redondants pour la récupération de pannes. Il est important de bien comprendre que seul un système redondant peut s'avérer pleinement efficace lorsqu'on doit avoir une confiance absolue sur les résultats qu'il génère ainsi qu'à sa capacité d'être disponible en tout temps. C'est d'abord et avant tout un moyen efficace d'éviter toute interruption d'un système. En effet, si un système n'est pas redondant et qu'une de ses composantes principales devient inopérante ou défectueuse, les chances que ce système donne le bon résultat ou encore les chances de le voir

compléter la tâche qu'il a entreprise sont minimes. Ceci aura pour effet de rendre le système inopérant jusqu'à sa prochaine remise à zéro ou lors du remplacement de la composante défectueuse. Les différents mécanismes de tolérance aux défaillances utilisés dans le développement du système d'exploitation seront détaillés dans le chapitre 2.

1.3. L'injection de pannes.

L'injection de pannes est une étape incontournable lorsqu'on implante des mécanismes de tolérance aux défaillances ou de simple mécanismes de détection de pannes. Elle est indispensable lorsqu'on désire étudier le comportement d'un simple composant où encore du système dans son ensemble. Afin de simplifier la nomenclature dans cette section, le terme système sera utilisé de façon générale pour représenter aussi bien une seule pièce que l'ensemble d'un système proprement dit.

Dans cette section, on désire donner un bref aperçu des différentes techniques d'injection de pannes existantes. En effet, il existe plusieurs méthodes qui ont été développées pour réaliser des injections de pannes dans les systèmes. L'article intitulé "Fault Injection A Method For Validating Computer-System Dependability" écrit par Clark et Pradhan [CLA95] donne une bonne description de plusieurs méthodes employées pour injecter des pannes.

On remarque principalement deux types d'approche pour l'injection de pannes, soit une approche matérielle qui consiste à effectuer des perturbations de toutes sortes à l'aide de ressources matérielles, ou encore une approche logicielle qui tend à simuler le comportement ou l'effet des pannes qu'on retrouve normalement dans un système donné. Les prochains

paragraphes de cette section mettront en lumière les différentes méthodes d'injection de pannes qui ont le plus attiré notre attention.

1.3.1. Injection matérielle de pannes.

On parlera premièrement des injections matérielles de pannes puisqu'elles semblent être les plus couramment utilisées. En effet, la majorité des articles écrits sur le sujet parlent de méthodes matérielles d'injection de pannes. On peut classer ces dernières en trois catégories : soit les injections de pannes qui sont introduites par perturbations externes aux systèmes, celles qui sont introduites à l'aide d'éléments perturbateurs au niveau des broches d'un circuit intégré "pin-level" et finalement celles qui sont introduites à l'aide de mécanismes faisant partie intégrante d'un système.

a) Injection par perturbation externe.

Dans les mécanismes d'injection par perturbations externes, on retrouve principalement deux types d'injection : soit l'injection par bombardement d'ions lourds ou l'injection par perturbation du bloc d'alimentation [KAR94].

L'injection de pannes par bombardement d'ions lourds consiste à exposer la surface d'un circuit intégré auquel on a préalablement retiré la surface du boîtier aux radiations d'une substance radioactive. Ce bombardement a pour effet de créer une inversion de bit dans la partie affectée par le passage d'un ion lourd. Cette technique permet un contrôle plus ou moins précis de l'endroit où on injecte les pannes. De plus, elle requiert des équipements très sophistiqués pour sa réalisation comme l'utilisation d'une chambre à vide miniature.

Cependant, elle permet de déterminer le comportement réel qu'aura un système lorsqu'il sera soumis à de tels perturbations. Ainsi on a pu déterminer que les inversions de bits se produisaient uniquement de l'état 1 à l'état 0 dans le registre utilisateur d'un processeur MC6809C de la compagnie Motorola.

L'injection de pannes à l'aide du bloc d'alimentation [KAR94] s'avère une approche plus économique que celle utilisant les ions lourds. Cependant, elle est moins utilisée puisque le contrôle du lieu des zone perturbées est impossible. En effet, l'injection de pannes par le bloc d'alimentation consiste à créer des perturbations dans ce dernier. Ces perturbations ont pour effet de créer des baises de tension et ainsi provoquer des inversions de bits dans tout le système. Le nombre de pannes introduites dans le système va dépendre de la durée de l'interruption et du temps que prend chaque cellule à perdre son état. Afin de contrôler le nombre approximatif de pannes qu'on désire introduire, on se doit d'utiliser une minuterie qui déterminera la durée de l'interruption d'alimentation en fonction du circuit qu'on désire utiliser. Cependant, les chances de recréer les mêmes conditions expérimentales sont pratiquement nulles.

b) Injection via les broches d'un circuit intégré.

Dans la catégorie des injections de pannes via les broches d'un circuit intégré, on retrouve MESSALINE [ARL89] qui se veut un outil automatisé de validation de systèmes tolérants aux défaillances. MESSALINE permet d'analyser plusieurs circuits intégrés simultanément. De plus, il se veut indépendant du circuit intégré utilisé. Ainsi, il est donc possible de l'adapter à différents circuits intégrés moyennant quelques informations sur ce dernier.

MESSALINE offre deux modes d'injection de pannes. Le premier utilise des sondes que l'on connecte aux différentes broches d'un circuit intégré. Cette technique s'effectue normalement lorsque ce dernier est sur un circuit imprimé. Ainsi, on peut étudier le comportement du circuit dans son milieu d'utilisation. Le second utilise une boite d'isolation qui permet, à l'aide de transistors, d'injecter des pannes dans un circuit intégré seulement. Ainsi, on évite d'affecter l'ensemble du système.

c) Injection à l'aide de mécanismes internes.

Le DEF.Injector [GER89] est un autre outil automatisé d'injection de pannes servant au développement de systèmes tolérants aux défaillances. Ce dernier fait appel à un module d'émulation de la mémoire du processeur. Ce module est accessible en tout temps par le microprocesseur visé et celui responsable des injections de pannes. Le type de panne injecté dépend de l'algorithme choisi par l'utilisateur du système.

1.3.2. Injection logicielle de pannes.

L'injection logicielle de pannes est un autre moyen très efficace d'accomplir cette tâche. Beaucoup moins onéreuse que l'approche matérielle, elle offre une très grande flexibilité et elle permet également d'être implantée plus rapidement sur les systèmes existants. Cependant, le temps nécessaire pour injecter une panne est de beaucoup supérieur à celui de l'approche matérielle et elle ne permet pas de déterminer les caractéristiques des différentes composantes d'un système sous l'influence d'un type de pannes bien précis. De plus, l'utilisation de cette technique influence les résultats lorsqu'on désire quantifier certains paramètres comme le temps de récupération des pannes et le temps moyen entre les pannes.

a) FERRARI.

FERRARI [KAN92] est un bon exemple d'outil d'injection de pannes utilisant une approche logicielle. En effet, FERRARI fut développé pour être exécuté dans un environnement X-window sur des stations de travail du type SPARC. Il permet de simuler jusqu'à huit types de panne différents. L'utilisateur peut également déterminer si les pannes injectées sont temporaires ou permanentes.

FERRARI fonctionne un peu sur le principe des outils de déverminage. En effet, après que l'utilisateur ait déterminé quel type de panne il veut simuler, le système analyse le jeu d'instructions du programme visé et insère un marqueur qui déterminera l'emplacement où devra être introduite la panne. Une fois le marqueur introduit, on procède à l'exécution du programme. Ce dernier s'arrêtera à l'endroit spécifié par le marqueur. Une substitution sera alors effectuée dans l'instruction qui est en cours d'exécution. Cette substitution durera un ou plusieurs cycles selon que cette panne est temporaire ou permanente. De plus, la portion de l'instruction qui sera affectée dépendra de la nature de la panne que l'utilisateur désire simuler.

1.3.3. L'approche d'injection retenue.

Etant donné les ressources limitées pour le développement du système d'exploitation, on a décidé d'utiliser une approche logicielle pour effectuer les injections d'erreurs. En effet, cette approche nécessite peu de ressources et peut être implantée rapidement dans n'importe quel système. La "qualité" des erreurs simulées dépend des algorithmes utilisées pour recréer celles-ci. Malgré le fait qu'elle ne permette pas d'évaluer avec exactitude certaines données statistiques, elle n'en est pas moins efficace lorsque vient le temps d'évaluer la viabilité d'un

système. Les détails sur les mécanismes d'injection d'erreurs adoptés seront abordés dans la section 5 du chapitre 3.

CHAPITRE 2

LE SYSTÈME D'EXPLOITATION FATMOS

2. PRÉAMBULE

Le système d'exploitation FATMOS "FAult Tolerant Multiprocessor Operating System" a été développé dans le cadre d'un projet avec le ministère de la défense nationale du Canada. Il se veut un outil permettant d'établir l'efficacité de certains mécanismes de détection et de tolérance aux pannes pouvant survenir dans un système multiprocesseur. De plus, il permet d'étudier le comportement des systèmes multiprocesseurs lorsque ces derniers sont soumis à différents types de pannes.

Le système FATMOS peut opérer sous cinq modes différents. De ces cinq modes, seulement quatre possèdent un mécanisme de détection de pannes. Ces quatre modes d'opération sont basés sur le même principe de détection des pannes, soit par la comparaison de certains résultats. Ils se distinguent cependant par la méthode employée pour réaliser cette comparaison. Ainsi, le jumelage d'un certain nombre de processeurs permet de détecter toute anomalie pouvant affecter le comportement normal d'une application. De par sa conception, FATMOS permet de simuler de façon logicielle trois types de pannes pouvant survenir dans un système multiprocesseur.

2.1.1. Raisons d'être des cinq modes d'opération.

Deux raisons principales motivent l'existence de cinq modes d'opération différents, tout d'abord, chacun des modes d'opération permet de répondre à un niveau de tolérance différent avec un coût plus ou moins important. De plus, l'utilisation des cinq modes permet de comparer leur efficacité.

2.1.2. L'utilisation du système HIBUS versus les autres systèmes.

Le système d'exploitation FATMOS est intimement lié à une classe d'architectures multiprocesseurs. L'architecture utilisée est celle développée dans le cadre du projet avec le ministère de la défense nationale. Cette architecture est basée sur l'utilisation de bus hiérarchiques. Elle se distingue surtout au niveau de son organisation générale et de ses possibilités d'intégration sur un circuit intégré à très grande échelle de la taille d'une gaufre de silicium. L'organisation structurelle du système HIBUS consiste en un processeur hôte auquel sont rattachés plusieurs processeurs d'exécution. Le processeur hôte communique avec les autres processeurs via un bus principal. Il peut exister plusieurs niveaux de hiérarchie dans la structure. Les processeurs sont toujours localisés sur les bus de plus bas niveau. L'architecture HIBUS est décrite en détail à la section 3.2.

2.2. Définitions.

Pour des raisons de compréhension, il est important de bien définir les termes techniques utilisés dans ce chapitre.

Panne simple : Une panne est dite simple lorsqu'une seule panne est injectée dans le système.

Panne multiple : Une panne est dite multiple lorsque plusieurs pannes sont injectées dans le système.

Processeur hôte : Processeur responsable de l'interface usager du système multiprocesseur.

Processeur de surveillance : Processeur responsable de la gestion des pannes dans le système multiprocesseur.

Vecteur de syndrome : Chaîne de bits contenant l'information sur les résultats de la routine de comparaison des messages d'un processeur.

2.3. L'exécution en mode normal.

Le mode normal d'opération est comparable au mode d'opération que l'on retrouve dans les systèmes multiprocesseurs traditionnels. En effet, il ne comporte aucun mécanisme de tolérance aux pannes. Son implantation permet de vérifier le fonctionnement normal des applications avant d'effectuer des expériences en utilisant les autres modes d'opération. De plus, le mode d'opération normal permet d'étudier le comportement des pannes sur les systèmes multiprocesseurs traditionnels. Il permet également de quantifier le gain réel des différents modes de tolérance aux pannes. Puisque ce mode n'est pas notre champ d'intérêt principal, il ne sera pas approfondi davantage. Cependant, il permettra d'effectuer une étude comparative sur l'effet des pannes simples sur un système multiprocesseur.

2.4. Le triplet symétrique.

Deux objectifs ont été fixés pour le triplet symétrique. Le premier était d'obtenir un taux de succès près de 100% pour le récupération des pannes simples pouvant survenir dans un système multiprocesseur. On a fixé cet objectif en considérant le nombre de processeurs utilisés et le nombre de comparaisons qui sont effectuées entre les processeurs afin de détecter les pannes pouvant se produire dans le système. Étant donné l'absence de point critique de

défaillance dans ce mode d'opération, il était envisageable de fixer un taux de récupération approchant les 100%. Le second objectif était de construire un mode d'opération le plus robuste possible en vue de tolérer plusieurs types de pannes pouvant survenir dans le système. On peut atteindre cet objectif en combinant le mécanisme de vérification des processeurs à d'autres mécanismes de vérification des différentes composantes du système multiprocesseur, comme la gestion des communications lors de la réception de message, du système de gestion d'accès au bus de données via l'interface DMA ou les aiguilleurs.

2.4.1. Principe de fonctionnement.

Le triplet symétrique consiste à jumeler trois processeurs afin de former un groupe de processeurs exécutant le même programme avec les mêmes données. Afin de distinguer les triplets, on utilise un numéro obtenu à partir de l'adresse de chaque processeur dans le système. On détermine le numéro du triplet auquel un processeur appartient en divisant l'adresse réelle du processeur par quatre. La partie entière du résultat de cette division correspond à l'adresse logique du triplet. Ainsi, seulement quatre processeurs pourraient être associés à un même triplet. De ces quatre processeurs, seulement trois en feront partie. Le quatrième processeur n'est pas utilisé lorsqu'on utilise le système d'exploitation en mode triplet symétrique. Cependant, l'utilisation du système HIBUS permet de configurer l'adresse des processeurs par l'intermédiaire du processeur hôte. Chacun des trois processeurs formant un triplet occupe une fonction particulière. Cette fonction est régie par son adresse. Des processeurs appartenant à des triplets différents remplissent la même fonction si le résultat de l'opération modulo 4 sur leur adresse est identique. Le tableau suivant montre un exemple

d'attribution d'adresses logiques à l'intérieur d'un triplet ainsi que la fonction associée à chacun des processeurs.

Tableau 2.1 : Exemple d'attribution d'adresses logiques d'un triplet et de la fonction associée à chacun des processeurs.

Adresse réelle du processeur (hexadécimal)	**Adresse Logique (partie entière de l'adresse divisée par 4)**	**Résultat de modulo(adresse,4)**	**Fonction du processeur**
0x0000	0	0	Maître
0x0001	0	1	Adjoint
0x0002	0	2	Esclave
0x0003	0	3	Aucune
0x0004	1	0	Maître
0x0005	1	1	Adjoint
0x0006	1	2	Esclave
0x0007	1	3	Aucune

Lors de l'exécution d'un programme, les membres d'un même triplet exécutent les mêmes instructions sauf lors des opérations de comparaison requises pour la validation des traitements. Ces comparaisons peuvent être effectuées à tout moment. Il est cependant plus logique de les effectuer que lorsque les processeurs doivent transmettre de l'information à d'autres processeurs.

La routine de comparaison permet de détecter et de diagnostiquer les pannes pouvant survenir dans un des membres du triplet. Cette routine s'effectue en deux étapes. Chacune des étapes est essentielle au bon fonctionnement de la routine de comparaison. Chaque étape doit

être respectée rigoureusement puisqu'on fait appel a des temporisateurs pour détecter toutes défaillances pouvant survenir dans l'un des processeurs d'un triplet.

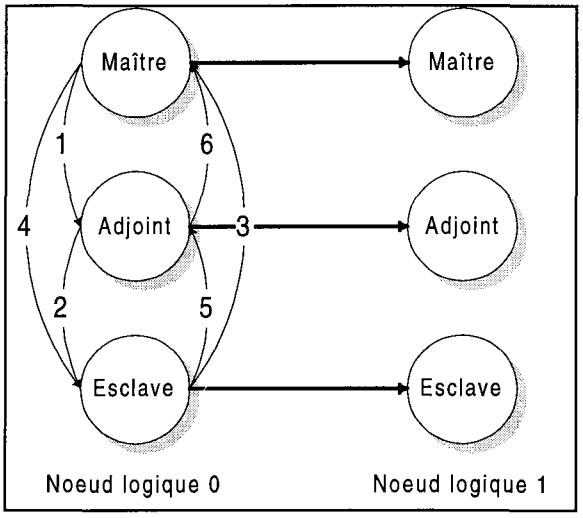

Figure 2.1 Échange de messages du triplet symétrique

La première étape consiste à échanger des messages entre les processeurs du triplet. Elle permet ainsi de détecter et de diagnostiquer les différentes pannes pouvant survenir dans les processeurs. Lors de la séquence d'échange des messages, chaque processeur membre du triplet envoie un message de comparaison aux autres membres du triplet. L'ordre dans lequel chacun des messages est échangé doit être respecté rigoureusement. La figure 2.1 montre l'ordre suivant lequel les messages sont échangés. Les flèches sont numérotées selon l'ordre d'envoi des messages.

Afin d'éviter que les membres d'un triplet n'attendent indéfiniment un message provenant d'un processeur en panne, on fait appel à un temporisateur. Des temporisateurs sont utilisés dans chacun des modes d'opération comportant des mécanismes de tolérance aux pannes. Leurs utilisations permettent d'éviter les points critiques de défaillance (« Single point failure»)

bien connu dans le domaine de la tolérance aux défaillances. Un point critique de défaillance dans le cas présent est défini comme étant une portion de programme où tous les processeurs d'un triplet dépendent d'un seul processeur pour poursuivre leurs tâches. Afin d'éviter les points critiques de défaillance, il faut s'assurer que le temporisateur donne une souplesse suffisante pour éviter d'interpréter une légère désynchronisation de la part d'un processeur comme étant une erreur. De plus, il est important qu'il dispose de mécanismes lui permettant de se modifier lorsqu'un message comporte plus d'une trame. Sans cette protection, la séquence d'échange de messages causerait des problèmes puisque les messages que l'on transmet n'ont pas toujours la même longueur et qu'il serait inefficace de déterminer un délai fixe qui répondrait à la pire des situations.

Une fois la séquence d'échange de messages terminée, on enregistre l'information tirée des comparaisons dans une chaîne de caractères que l'on appelle "vecteur de syndrome". L'information contenue dans le vecteur de syndrome sera expliquée dans la section suivante. Si aucun temporisateur n'est venu à expiration lors de la réception des messages de comparaison, on peut alors procéder à une dernière comparaison. Cette comparaison consiste à comparer les deux messages reçus des autres membres du triplet. Le résultat de cette comparaison permettra de compléter le vecteur de syndrome et de donner davantage d'information sur l'état d'un processeur suspect lorsque le processeur chargé de la surveillance du système aura à prendre une décision concernant l'un des membres du triplet.

L'étape suivante de la routine de comparaison se divise en deux parties, soit la gestion des erreurs et la synchronisation des processeurs suite à l'échange des messages. Cette étape est la plus complexe de la routine de comparaison car elle doit prendre en considération tous les cas

possibles pouvant survenir durant la procédure de comparaison. Dans la première partie, il y a deux cas à considérer, le premier étant lorsqu'aucune erreur ne survient pendant l'échange des messages et le second est lorsqu'une erreur survient pendant l'échange des messages. Par la suite, on procède à l'envoi des messages à la destination après synchronisation des membres du triplet.

Lorsqu'aucune erreur ne se produit pendant l'échange de messages, le processeur adjoint est responsable d'envoyer un message de synchronisation aux deux autres processeurs membres du triplet. Le choix d'utiliser le processeur adjoint pour envoyer le signal de synchronisation est justifié par le fait qu'il est le dernier à compléter la séquence d'échange de messages. La synchronisation est nécessaire afin d'éviter une mauvaise interprétation de l'état du triplet destination par le processeur de surveillance. En effet, tout retard dans l'envoi d'un message de la part d'un processeur du triplet source s'ajouterait aux délais de transfert de la routine de comparaison du triplet destination. On risquerait ainsi de voir expirer un temporisateur dans le triplet destination alors que la cause du retard proviendrait du triplet source. De plus, la synchronisation des processeurs devient capitale, afin d'éviter qu'un processeur envoie un message à son homologue du triplet destination avant qu'un autre processeur du triplet n'ait pu indiquer qu'il a détecté une erreur lors de l'échange de messages.

Puisqu'aucun processeur ne peut être responsable des deux autres processeurs (afin d'éviter les points critiques de défaillance) le processeur maître fait appel à un temporisateur afin d'éviter une étreinte fatale ("deadlock") du triplet lorsqu'une erreur survient dans le processeur adjoint responsable de la synchronisation. Il est cependant inutile que le processeur esclave utilise un temporisateur puisque suite à l'échec de deux processeurs sur trois dans un même

triplet, ce dernier devient non opérationnel et cause systématiquement une erreur fatale pour l'application.

Lorsqu'une erreur ou un comportement anormal est détecté pendant la séquence d'échange de messages, le premier processeur ayant détecté l'anomalie envoie un message via un signal d'interruption aux deux autres processeurs du triplet les obligeant à envoyer leurs vecteurs de syndrome au processeur de surveillance. Une fois les signaux d'interruption envoyés, le processeur qui a détecté l'anomalie peut envoyer son vecteur de syndrome via un signal d'interruption au processeur de surveillance. Lorsque le message est envoyé, chacun des processeurs attend la décision du processeur de surveillance afin de poursuivre l'exécution du programme.

Lorsque la séquence de synchronisation est terminée, les processeurs envoient leurs messages à leurs homologues du triplet destination ou au processeur hôte selon l'adresse du destinataire. Une fois que les processeurs ont envoyé leurs messages à leurs destinations, ces derniers poursuivent l'exécution de l'application.

2.4.2. Le mécanisme de tolérance aux pannes.

Le mécanisme de tolérance aux pannes fait appel à un processus décisionnel. Le processus décisionnel est assumé par le processeur de surveillance qui est responsable de déterminer si un processeur doit être mis hors service pour la durée de l'application. Le processus décisionnel implique plusieurs règles permettant de rendre un verdict aussi précis que possible tout en évitant d'éliminer un processeur inutilement. Cependant, il est important d'éviter que

les décisions rendues soient basées sur des vecteurs erronés. Afin d'éviter ces problèmes, on fait appel à plusieurs mécanismes permettant de contrer cette éventualité.

La détection d'une panne débute par l'envoi d'un message appelé « vecteur de syndrome » de la part d'un processeur qui a détecté une panne possible. Un vecteur de syndrome est un message contenant l'information obtenue lors des différentes comparaisons effectuées pendant la séquence d'échange de messages. L'information contenue dans un vecteur est structurée de la façon suivante :

Tableau 2.2 : Information contenue dans un vecteur de syndrome.

No de l'Octet	**Information contenue dans chaque octet**
0	Octet le plus significatif de l'adresse du processeur qui a envoyé le message.
1	Octet le moins significatif de l'adresse du processeur qui a envoyé le message.
2	Résultat de la première comparaison dans l'échange de messages.
3	Résultat de la deuxième comparaison dans l'échange de messages.
4	Résultat de la comparaison des deux messages reçu lors de l'échange de message.
5	Numéro du vecteur de syndrome.

Lorsqu'un vecteur de syndrome est reçu par le processeur de surveillance, ce dernier récupère l'information qu'il contient et l'enregistre dans une table.

L'utilisation d'une table contenant les vecteurs de syndrome permet de conserver ces derniers le temps qu'une décision soit rendue. En effet, les vecteurs de syndrome d'un triplet

donné n'arrivent jamais simultanément. De plus, l'utilisation de cette table permet au processeur de surveillance de traiter plusieurs erreurs provenant de triplets différents sans qu'il y ait confusion dans le processus décisionnel. C'est avec les informations contenues dans la table que le processeur de surveillance peut rendre son verdict sur un processeur suspect dans un triplet. Certaines informations sur un triplet sont enregistrées de façon permanente comme lorsqu'un des processeurs est reconnu fautif et qu'une reconfiguration du triplet devient alors nécessaire.

Le processus décisionnel du processeur de surveillance s'effectue en deux étapes. La première étape consiste à déterminer quel processeur est fautif et la seconde permet de reconfigurer les processeurs lorsqu'une décision est rendue contre l'un des membres d'un triplet.

Il est important de faire appel à plus d'un rapport pour rendre une décision concernant un des membres du triplet. Le but de ces deux rapports est d'éliminer l'existence de points critiques de défaillance au cas où une erreur survienne dans la génération des vecteurs de syndrome. On doit donc utiliser plusieurs processeurs se vérifiant entre eux. Ainsi, on ne fait jamais confiance à un seul processeur pour rendre un jugement sur l'ensemble des processeurs.

Dans le cas d'un triplet, seulement deux vecteurs de syndrome sont nécessaires pour rendre un verdict. En se basant sur le principe que l'on ne peut faire confiance à aucun des processeurs lorsque ces derniers envoient leur vecteur de syndrome, une décision est rendue lorsque deux processeurs, à l'exception de celui suspecté, démontrent qu'un processeur est

fautif. Par exemple, si le processeur adjoint envoie un vecteur de syndrome indiquant que les processeurs maître et esclave ont une erreur dans leur message, cela pourrait indiquer qu'effectivement les messages sont erronés ou, qu'en fait, c'est le processeur adjoint qui a effectué incorrectement la comparaison. Le tableau 2.3 indique les conditions qui permettent de mettre un processeur hors service suit à la réception de deux vecteurs de syndrome. Afin de bien comprendre le tableau 2.3, on doit comprend le contenu de l'information dans un vecteur de syndrome (tableau 2.2).

Tableau 2.3 : Conditions de mise hors service d'un processeur pour un triplet symétrique.

Processeur Fautif	**Vecteur du processeur**	**Vecteur de syndrome des processeurs**		
		Résultat de la 1er comparaison	**Résultat de la 2e comparaison**	**Résultat de la Comparaison entre les deux messages**
Maître	**Adjoint**	Erreur	Ignoré	Erreur
	Esclave	Erreur	Ignoré	Erreur
Adjoint	**Maître**	Erreur	Ignoré	Erreur
	Esclave	Ignoré	Erreur	Erreur
Esclave	**Maître**	Ignoré	Erreur	Erreur
	Adjoint	Ignoré	Erreur	Erreur

Une condition est satisfaite lorsque l'information contenue dans les vecteurs de syndrome des deux autres membres comportent des erreurs aux endroits spécifiés dans le tableau 2.3. Si aucune des conditions n'est respectée et qu'on a reçu les vecteurs de syndrome des trois processeurs du triplet, alors aucune mesure n'est prise contre le processeur suspect. Ceci permet d'éviter qu'un processeur soit mis hors service lors de la réception d'un mauvais

vecteur de syndrome d'un des processeurs du triplet. Cependant, il arrive parfois qu'un des processeurs du triplet soit réellement fautif et qu'il soit impossible de le diagnostiquer lors du premier envoi de vecteurs de syndrome du triplet. Par exemple, dans le cas où le processeur adjoint commence une boucle sans fin immédiatement après la routine de comparaison des messages, le temporisateur de resynchronisation du processeur maître viendra à échéance. Ce dernier forcera tous les processeurs du triplet à envoyer leurs vecteurs de syndrome. Une fois les vecteurs de syndrome reçu, le processeur de surveillance ne peut prendre une action contre le processeur adjoint puisque l'information contenue dans les vecteurs de syndrome n'indique aucune erreur de la part de ce dernier. Ainsi, l'erreur ne sera pas détectée avant la prochaine routine de comparaison de messages du triplet.

Dans le cas où les informations mènent à la mise hors service du processeur fautif, le processeur de surveillance envoie un ordre au processeur fautif lui indiquant de se placer hors service jusqu'à la fin du programme ou jusqu'à ce que tous les processeurs reçoivent un signal de réinitialisation du processeur hôte pour exécuter un nouveau programme. Ce dernier message est également envoyé via un signal d'interruption. Une fois le message envoyé, le processeur de surveillance envoie un message aux deux autres processeurs du triplet afin que ces derniers se reconfigurent et exécutent maintenant la routine de comparaison en mode duo. Cette routine sera expliquée à la section 2.7.

La procédure de reconfiguration permet aux deux autres membres du triplet d'exécuter normalement leurs tâches. De plus, ces derniers font en sorte que les messages qui devaient être transmis par le processeur mis hors service soient effectivement émis par un des membres du duo de manière à ce que les autres triplets ne perçoivent pas de changement dans leurs

opérations de communication. Lorsque le processeur de surveillance envoie le message de reconfiguration aux deux autres processeurs du triplet, il donne un rôle à chacun des processeurs selon la fonction qu'ils occuperont en mode duo. Le rôle de chacun des processeurs est déterminé par le tableau 2.4.

Tableau 2.4 : Rôle des processeurs lorsqu'ils opèrent en mode DUO.

Processeur Hors Service	**Processeur DUAL**	**Processeur COMPLEMENT**
Maître	Adjoint	Esclave
Adjoint	Esclave	Maître
Esclave	Maître	Adjoint

La fonction du processeur DUAL et celle du processeur COMPLEMENT seront détaillées à la section 2.7 portant sur le fonctionnement en mode duo symétrique. Les termes DUAL et COMPLEMENT sont arbitraires, ils permettent cependant d'identifier la fonction associée à chacun des processeurs lorsqu'ils opèrent en mode DUO.

Afin d'éviter l'effet d'un mauvais vecteur de syndrome, plusieurs protections ont été développées. Les premières protections se situent au niveau des triplets. On s'assure qu'un vecteur de syndrome ne peut être envoyé que pendant la routine de synchronisation des processeurs. Cette mesure assure que les autres membres du triplet ne soient pas désynchronisés suite à une requête du processeur de surveillance leur demandant leurs vecteurs de syndrome. Une deuxième protection consiste à initialiser les informations contenues dans les vecteurs de syndrome afin de présenter un état d'opération normal, ce qui

n'aura aucun effet sur le bon fonctionnement d'un triplet lors d'une transmission impromptue d'un vecteur de syndrome.

Les protections suivantes ont été ajoutées dans le programme de diagnostic du processeur de surveillance. Lorsqu'un vecteur de syndrome est envoyé au processeur de surveillance, un numéro lui est associé. Ce numéro permet au processeur de surveillance de déterminer si le vecteur de syndrome a déjà été traité ou si le numéro de rapport est le prochain à être traité. Dans le dernier cas, la portion de la table des vecteurs de syndrome associée à un triplet est réinitialisée lorsque le numéro de rapport change. De plus, il est essentiel que le processeur de surveillance ait reçu plus d'un message possédant le même numéro de syndrome avant de pouvoir réinitialisé la table des vecteurs.

2.4.3. Le choix de l'emplacement de la routine de comparaison.

Le choix d'implanter la routine de comparaison des processeurs à l'envoi ou à la réception d'un message dépend de l'approche du concepteur ainsi que de différents facteurs.

Dans le cadre de ce projet, on a déterminé que le positionnement de la routine de comparaison à l'envoi des messages s'avérait plus approprié que celle à la réception. Après une analyse rigoureuse de tous les facteurs pouvant influencer un tel choix, on est arrivé à la conclusion qu'il existait un facteur majeur pouvant influencer ce choix. En effet, les deux possibilités offrent des avantages et des inconvénients. On doit mentionner qu'il est possible d'implanter tous les mécanismes de protection désirés à l'envoi comme à la réception des messages. Cependant, seul le positionnement à l'envoi permet de circonscrire un problème immédiatement lorsqu'il survient contrairement au positionnement à la réception où la

vérification s'effectue après qu'il ait effectué son travail. Ainsi, lorsqu'un triplet transmet son message à son homologue, on est assuré de la validité des données ce qui n'est pas le cas lorsque la routine de vérification est exécutée à la réception. En effet, lorsqu'on effectue une vérification à la réception et que le triplet source est configuré en mode duo, il se peut qu'une erreur introduite dans l'un des deux processeurs soit propagée dans le système.

2.4.4. Communication.

Dans cette première version du système d'exploitation, on s'est attardé à développer la tolérance aux pannes au niveau des processeurs. Il n'existe aucun mécanisme permettant d'évaluer la validité des communications entre les processeurs. Cependant, certains mécanismes peuvent être utilisés afin de combler cette lacune.

L'ajout de mécanismes permettant d'évaluer les problèmes dans les communications permet d'augmenter la robustesse du système lorsque surviennent des pannes. Plusieurs solutions sont possibles pour valider les communications. Cependant, on se concentrera sur deux méthodes très efficaces qui permettent d'obtenir une vérification judicieuse de la qualité des communications.

La première consiste à effectuer une vérification de type CRC (Cyclic Redundancy Code) lorsqu'un processeur reçoit un message. Cette méthode exige l'exécution d'une série d'opérations à l'envoi du message ainsi qu'à la réception de ce dernier. L'utilisation du CRC pour vérifier les communications est très efficace. On peut ainsi s'assurer d'une transmission fiable entre deux processeurs. Cependant, il existe un problème lorsqu'on utilise cette technique pour vérifier les communications en mode d'opération triplet symétrique. En effet,

si un des processeurs devient défectueux après l'envoi du message de synchronisation et qu'aucun message n'est transmis à son processeur homologue du triplet destination, alors le processeur destination qui n'a pas reçu son message sera mis hors service puisque la routine de comparaison dans le triplet destination indiquera que ce processeur est défectueux.

La deuxième méthode consiste à forcer chaque processeur du triplet source à envoyer ses messages à tous les processeurs du triplet destination. Ainsi, chaque processeur du triplet destination pourra effectuer une comparaison des messages qu'il a reçus. Il suffit que deux messages provenant de processeurs différents soient identiques pour s'assurer de la validité des communications. Afin d'augmenter les chances de succès, les processeurs du triplet destination doivent attendre les trois messages pendant un certain laps de temps avant de poursuivre. Puisque cette technique ne nécessite que peu de modifications au système d'exploitation et permet d'éviter de mettre hors service un processeur inutilement, elle serait donc préférée à la technique utilisant le CRC. Notons que l'on peut réduire grandement le trafic dans les communications en utilisant une des fonctionnalités du système d'exploitation qui permet de faire en sorte qu'un seul message soit reçu par plusieurs destinations "multicast".

Malgré le peu de modifications qu'impose l'ajout de cette technique, elle n'a pas été implantée dans cette version du système d'exploitation. La raison provient du nombre réduit de mémoire tampon disponible pour la réception des messages dans chaque processeur. Cette limite est critique pour la taille et le nombre de messages que peut recevoir simultanément un processeur. La limite du nombre de tampons est liée à la faible taille de la mémoire des processeurs de type HPC utilisés dans le prototype expérimental HIBUS utilisé pour cette étude.

2.4.5. Forces et faiblesses du triplet symétrique.

Le triplet symétrique est a priori le plus robuste des modes d'opération. Cependant le prix de cette robustesse doit être établi afin que l'utilisateur potentiel puisse juger de la pertinence de ce mode d'opération.

Le principal avantage du triplet symétrique réside dans sa capacité à détecter une panne simple dans un des membres du triplet. Comme on l'a vu précédemment, il existe des circonstances qui ne peuvent être évitées lors de l'utilisation du mode triplet symétrique. Ce type d'erreurs ne peut être évité par aucun des cinq modes d'opération. Cette erreur se produit lorsqu'un processeur défectueux envoie continuellement le même message. Ceci a pour effet de congestionner les bus de données et crée une étreinte fatale dans le système. Cependant, il est possible d'éviter cette erreur. Pour ce faire, on devrait faire appel à des mécanismes de surveillance de l'utilisation des bus de données. Ces mécanismes seraient externes aux processeurs. On constate donc que l'on ne peut confier aux processeurs l'entière responsabilité de la gestion des erreurs afin d'obtenir des niveaux de tolérance aux pannes s'approchant des 100%.

Un autre avantage du triplet symétrique est qu'il peut éviter la propagation des erreurs à d'autres triplets lorsqu'il est combiné à des mécanismes de vérification des communications. Le fait d'utiliser plusieurs mécanismes de détection des différentes parties physiques du système multiprocesseur augmente les chances de récupération lorsque des pannes multiples surviennent dans une application.

Le triplet symétrique possède un inconvénient qui peut limiter son utilisation. Les nombreuses transmissions qu'il effectue lors de l'échange des messages augmentent le temps d'exécution de façon significative. Ce dernier se verrait augmenter davantage si l'on ajoutait une routine de vérification des messages à la réception dans un triplet.

2.5. L'exécution en mode triplet asymétrique.

Le triplet asymétrique est un mode de fonctionnement intermédiaire entre le triplet symétrique et le triplet mobile qui sera introduit dans la prochaine section. Sa capacité de tolérance aux défaillances est inférieure à celle du triplet symétrique. En effet, le triplet asymétrique possède un point critique de défaillance. Cependant l'étude du triplet asymétrique permet d'obtenir un point de référence pour le triplet symétrique. Ces comparaisons permettront d'évaluer le rapport gain de tolérance en fonction des coûts d'implantation et d'opération des différents modes d'opérations.

2.5.1. Principe de fonctionnement.

Le triplet asymétrique se veut une version allégée du triplet symétrique. Il existe deux différences importantes entre les deux modes d'opération. La première étant la routine de comparaison qui fait appel à une séquence d'échanges de messages moins importante que celle du triplet symétrique. La seconde est l'algorithme décisionnel qui permet au processeur de surveillance de rendre son verdict.

La principale modification dans la routine de comparaison se situe dans la séquence d'échange de messages. Cette modification permet d'accélérer la séquence d'échange. La figure

2.2 montre la séquence d'échange de messages et l'envoi du message final à un autre triplet. Les flèches numérotées de 1 à 4 représentent la séquence d'échange de messages. Les flèches sans numéro en gras représentent le message final qui est envoyé au triplet destination ou au

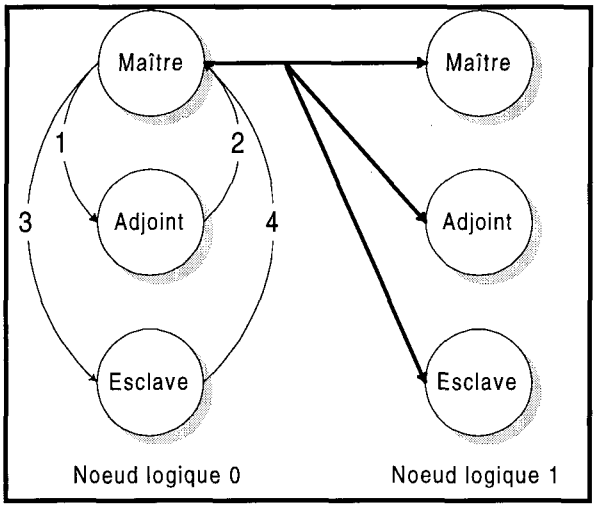

Figure 2.2 : Échange de messages du triplet asymétrique

processeur hôte après la resynchronisation des processeurs.

Les processeurs adjoint et esclave ne recevant qu'un seul message de comparaison, seul le processeur maître effectue une comparaison entre ces derniers. Ceci a pour effet de diminuer l'information contenue dans les vecteurs de syndrome de deux autres processeurs du triplet. Cette diminution d'information augmente la probabilité d'une interprétation incorrecte des vecteurs de la part du processeur de surveillance. Le dernier point à souligner, suite à la modification de la routine d'échange de messages, est la prise en charge du processeur maître pour la resynchronisation du triplet. Par conséquent, le processeur adjoint doit alors utiliser un temporisateur pour déterminer si le processeur maître a failli à la tâche.

Lorsque la resynchronisation des processeurs est terminée, seul le processeur maître envoie son message au triplet destination ou au processeur hôte. Dans le cas où la destination est un autre triplet, le processeur maître utilise un message à destination multiple. Ceci évite de congestionner les bus de données lorsque plusieurs triplets communiquent simultanément. Puisque le processeur maître est le seul à envoyer son message, on crée un point critique de défaillance dans ce mode d'opération. En effet, il y a point critique de défaillance puisque le bon fonctionnement du triplet dépend uniquement du processeur maître en un point précis de la routine de comparaison des messages. Ce problème peut occasionner une erreur fatale pour le système ou générer de mauvais résultats.

2.5.2. Le mécanisme de tolérance aux pannes.

Le triplet asymétrique fait appel aux mêmes mécanismes de reconfiguration que le triplet symétrique. Cependant, les conditions utilisées pour la mise hors service d'un processeur ne sont pas les mêmes. Les différences entre les deux modules décisionnels proviennent du fait que les vecteurs de syndrome des processeurs adjoint et esclave contiennent moins d'informations que celui du processeur maître. Le tableau 2.5 indique les conditions qui permettent au processeur de surveillance de mettre un processeur reconnu fautif hors service.

Tableau 2.5 : Conditions de mise hors service d'un processeur pour un triplet asymétrique.

Processeur Fautif	**Vecteur du processeur**	**Vecteur de syndrome**		
		1er message de comparaison	**2e message de comparaison**	**Comparaison entre les deux messages**
Maître	**Adjoint**	Erreur	Ignoré	Erreur
	Esclave	Erreur	Ignoré	Erreur
Adjoint	**Maître**	Erreur	Pas d'erreur	Erreur
	Esclave	Pas d'erreur	Pas d'erreur	Pas d'erreur
Esclave	**Maître**	Pas d'erreur	Erreur	Erreur
	Adjoint	Pas d'erreur	Pas d'erreur	Pas d'erreur

Comme le triplet symétrique, le triplet asymétrique utilise deux vecteurs de syndrome pour valider une décision concernant un processeur suspecté être fautif. Pour ce faire, on se base toujours sur le principe que l'on ne peut faire confiance à un seul processeur pour prendre une décision contre un autre processeur. Ceci assure qu'une décision ne peut être rendue qu'après l'envoi des vecteurs de syndrome de tous les membres du triplet.

2.5.3. Communication.

Les communications dans le triplet asymétrique sont moins nombreuses que dans le triplet symétrique. Cette diminution du nombre de messages est imputable au fait que le processeur maître est le seul à envoyer un message à la destination et que la routine d'échange de messages est moins complexe. De plus, lorsque le processeur maître envoie un message à un autre triplet, ce dernier utilise un message à destination multiple. Ceci permet d'utiliser les bus de données au minimum lors de la transmission d'un message.

Contrairement au triplet symétrique le nombre de méthodes permettant de valider les communications est limité. En effet, l'utilisation d'un système de vérification du type CRC s'avère la seule solution valable afin d'assurer une très grande fiabilité au niveau des communications. L'utilisation d'une routine de comparaison des messages à la réception comme dans le triplet symétrique s'avère inapplicable puisque le processeur maître est le seul à envoyer son message. Il existe d'autres routines de comparaison qui permettent une vérification des communications. Cependant, aucune ne donne un rendement supérieur à la méthode de vérification de type CRC.

2.5.4. Forces et faiblesses du triplet asymétrique.

Ce mode d'opération est moins robuste que le triplet symétrique, le processeur maître étant le talon d'Achille du triplet asymétrique. Cependant, il est moins lourd que le triplet symétrique de part son utilisation restreinte des communications. Ainsi on diminue les chances que ce dernier soit affecté par des erreurs aléatoires pouvant survenir dans le système lors de transmissions de messages.

2.6. L'exécution en mode triplet de vérification mobile.

Le mode triplet de vérification mobile "Roving Spare" est une variante du triplet asymétrique. Ses mécanismes de détection et de reconfiguration sont pratiquement identiques en tous points à l'exception qu'il n'existe qu'un seul triplet dans le système. Le triplet mobile est conçu pour vérifier le fonctionnement d'un seul processeur ou d'une des composantes qui lui est rattaché. Lorsqu'un comportement anormal survient dans un système multiprocesseur, il est important d'avoir des outils permettant de vérifier le fonctionnement du système.

L'utilisation de modes d'opération complexes comme le triplet symétrique ne sont pas nécessaires lorsque le but de l'opération est la vérification d'un processeur ou de ses composantes et non l'ensemble du système.

2.6.1. Principe de fonctionnement.

La différence de fonctionnement du triplet de vérification mobile versus le triplet asymétrique se situe dans la façon dont sont regroupés les processeurs pour former un triplet. Dans le mode triplet asymétrique, tous les processeurs du système forment des triplets, ce qui n'est pas le cas pour le triplet mobile. Dans le triplet mobile, il n'existe qu'un seul triplet. Les membres de ce triplet sont déterminés par l'utilisateur ce qui donne une grande souplesse à ce mode d'opération. L'efficacité du triplet mobile pour la vérification d'un processeur suspect dépend du choix de la fonction que l'on associe à ce dernier. En effet, la fonction que doit associer l'utilisateur à un processeur suspect est celle du processeur adjoint ou du processeur esclave. Ce choix est justifié par le fait que le processeur maître est un point critique de défaillance ce qui n'est pas le cas pour les processeurs adjoint et esclave. De plus, l'utilisation de la fonction du processeur maître le rendrait plus sensible à un vecteur de syndrome corrompu ou incomplet. Les autres processeurs du système qui n'appartiennent pas au triplet opèrent sans redondance. La figure 2.3 montre la séquence d'échange de messages et l'envoi du message final à un autre noeud.

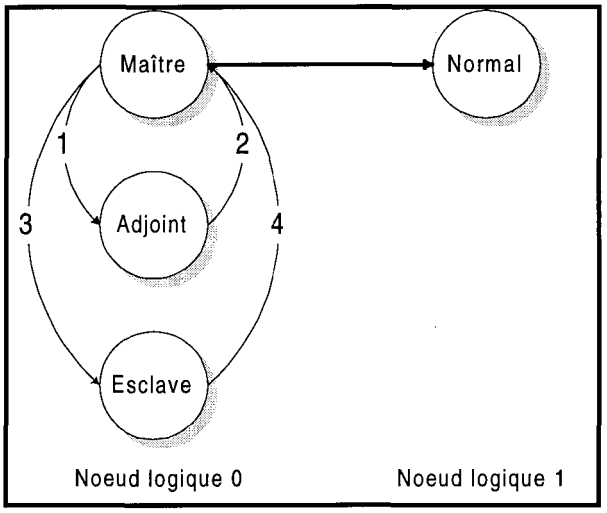

Figure 2.3 : Échange de messages du triplet mobile

On notera qu'il serait très facile d'utiliser le protocole d'échange de messages du triplet symétrique pour réaliser le triplet mobile. Cependant, il a été convenu de mettre en évidence certaines faiblesses du mode asymétrique en assignant la tâche de l'adjoint au processeur étudié. Ce sujet sera abordé au chapitre 4. Une des difficultés de gestion du triplet de vérification mobile réside dans l'attribution des adresses logiques de chacun des processeurs et les différentes conversions à effectuer lors de l'envoi de messages. Les adresses logiques des processeurs d'exécution sont déterminées par leurs adresses réelles, sauf pour les processeurs adjoint et esclave. Les adresses logiques se doivent d'être consécutives pour éviter toutes interventions de la part de l'utilisateur. On attribue les adresses logiques par ordre croissant des adresses physiques des processeurs, en excluant de la distribution les processeurs adjoint et esclave. Ces derniers se voient attribuer l'adresse logique du processeur maître. Ceci donne N-2 adresses logiques où N représente le nombre total de processeurs dans le système. Le tableau 2.6 donne un exemple d'attribution d'adresses logiques où le triplet mobile est assigné à l'adresse 0x0003.

Tableau 2.6 : Exemple d'attribution des adresses logiques pour le triplet mobile.

Adresse Physique	Adresse Logique	Fonction du processeur
0x0000	0x0000	Normal
0x0001	0x0001	Normal
0x0002	0x0003	Esclave
0x0003	0x0002	Normal
0x0004	0x0003	Maître
0x0005	0x0003	Adjoint
0x0006	0x0004	Normal
•	•	•
•	•	•
•	•	•
0xN	0xN-2	Normal

Dans le tableau 2.6, on peut constater que les adresses logiques et réelles du processeur 0x0000 sont identiques ainsi que celles du processeur 0x0001. Les autres processeurs ont des adresses logiques différentes de leurs adresses réelles. Les processeurs adjoint et esclave utilisent la même adresse logique que le processeur maître.

Une fois les adresses logiques déterminées, le triplet mobile est comparable à un hybride entre les modes d'opération normal et celui du triplet asymétrique. En effet, tous les processeurs qui n'ont pas de fonction particulière exécutent les mêmes instructions que celles normalement exécutées sans mécanisme de tolérance aux défaillances sauf lors de l'envoi d'un message où l'on doit distinguer la destination du message. Si la destination correspond à l'adresse logique du processeur maître, alors un message est envoyé à chacun des processeurs

du triplet mobile (en général des messages à destination simple). Dans le cas contraire, le message est envoyé normalement. Lorsqu'un processeur se voit attribuer une des trois fonctions du triplet mobile, alors il opère exactement comme un processeur dans le triplet asymétrique à la seule différence que le processeur maître envoie un message à une destination unique contrairement à un message à destination multiple pour le triplet asymétrique lorsque la destination est un triplet.

a) Le mécanisme de tolérance aux pannes.

Le triplet mobile fait appel aux mêmes mécanismes que le triplet asymétrique pour la reconfiguration en cas de pannes. Cependant, il existe quelques différences qui ne sont pas liées au mécanisme décisionnel mais plutôt à la gestion de la mémoire. En effet, contrairement aux triplets symétrique et asymétrique qui allouent de la mémoire pour chacun des processeurs du système afin de conserver l'état de ces derniers ainsi que leurs vecteurs de syndrome, le triplet mobile alloue de la mémoire uniquement pour les trois processeurs membres du triplet. Ceci a pour effet de libérer beaucoup d'espace-mémoire dans le processeur de surveillance.

Une autre différence touche l'envoi des messages de resynchronisation des processeurs. Étant donné que les adresses mémoire des trois processeurs formant le triplet ne sont pas nécessairement consécutives, il n'est pas toujours possible d'utiliser un message à destination multiple pour les réactiver. Il faut donc dans ce cas utiliser des messages à destination simple pour chacun des processeurs membres du triplet.

2.6.2. Communication.

L'utilisation des communications est minimale pour ce mécanisme de tolérance aux défaillances. En effet, il n'y a que trois processeurs au total qui effectuent une routine de comparaison de messages, ce qui est peu comparativement aux autres modes de détection de pannes.

2.6.3. Forces et faiblesses du triplet de vérification mobile.

Le triplet de vérification mobile est un mode d'opération très efficace pour la vérification d'un processeur ou d'une composante qui lui est associée. Il permet de vérifier un seul processeur comparativement aux autres modes. L'augmentation des coûts de communication imputables aux échanges pour fin de comparaison est minime. Il faut noter cependant que la vitesse d'exécution du triplet mobile est moindre que celle des autres processeurs et le triplet peut donc ralentir le temps de traitement.

Un problème important est associé à l'utilisation du triplet mobile. Ce problème réside dans le choix de la fonction associée au processeur suspect. Si la fonction choisie par l'utilisateur correspond à celle du processeur maître, alors le système hérite des problèmes du mode triplet asymétrique. Comme il sera démontré au chapitre 4, si la fonction choisie est différente de celle du processeur maître, alors la probabilité qu'une erreur dans un processeur suspect affecte le fonctionnement du système est grandement diminuée.

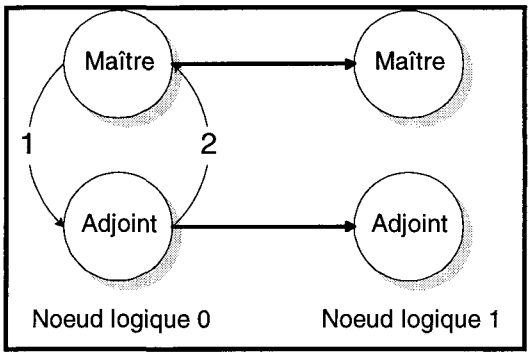

Figure 2.4 : Échange de messages du triplet mobile

2.7. L'exécution en mode duo symétrique.

Le duo symétrique est un mode d'opération qui ne permet que la détection d'erreur. Il ne permet pas de reconfiguration lorsqu'une erreur est détectée dans le système, puisqu'il est impossible de déterminer le processeur fautif. Cependant, il permet de valider un résultat puisqu'il ne possède aucun point critique de défaillance. Ceci assure l'intégrité des différents éléments du système. L'utilisation d'un tel mode lors de la conception de nouveaux systèmes peut s'avérer très intéressante.

2.7.1. Principe de fonctionnement.

Le duo symétrique, comme son nom l'indique, groupe les processeurs par paires. L'obtention des adresses logiques s'effectue de façon similaire aux triplets symétrique et asymétrique. Cependant, on utilise la division par deux pour obtenir l'adresse logique d'un duo comparativement à une division par quatre pour un triplet.

Dans le duo symétrique, la routine de comparaison permet uniquement de détecter les erreurs. Comme pour les triplets, cette dernière s'exécute en deux étapes. La routine d'échange de messages est restreinte au minimum comme on peut le voir dans la figure 2.4.

Le processeur maître initie la séquence. Une fois l'échange de messages terminé, les deux processeurs d'un duo effectuent une comparaison des données transmises. Si aucune erreur n'est détectée, le processeur maître devient responsable d'envoyer un message de synchronisation au processeur adjoint. Pendant ce temps, le processeur adjoint fait appel à un temporisateur afin de déterminer si le processeur maître est en panne. Ceci permet d'éviter les points critiques de défaillance.

Lorsqu'une erreur ou un comportement anormal est détecté pendant la séquence d'échange de messages, le premier processeur ayant détecté l'anomalie envoie un signal à l'autre processeur du duo lui indiquant qu'il doit envoyer son vecteur de syndrome au processeur de surveillance. Une fois ce signal expédié, le processeur qui a détecté l'anomalie peut envoyer son vecteur de syndrome. Après cet envoi, les deux processeurs attendent la décision du processeur de surveillance afin de poursuivre l'exécution de l'application.

Après la séquence de synchronisation, les deux processeurs du duo envoient leur message à leur homologue du duo destination ou au processeur hôte selon l'adresse du destinataire. Une fois les messages envoyés, les processeurs poursuivent l'exécution de l'application.

2.7.2. Le mécanisme de tolérance aux pannes.

Malgré le fait que le duo symétrique ne possède pas de mécanisme permettant la tolérance aux pannes, on fait appel au processeur de surveillance pour déterminer si un duo doit être mis hors service. Lorsqu'un vecteur de syndrome est reçu, le processeur de surveillance détermine si l'information qu'il contient permet de déterminer si l'un des deux processeurs du duo est fautif. Puisqu'il n'y a que deux processeurs, la réception d'un vecteur de syndrome

implique généralement que le duo sera mis hors service et que l'application se terminera de façon prématurée. En effet, puisqu'on a seulement deux processeurs et qu'il est impossible de déterminer lequel des deux processeurs est correct. Il est normal que les deux soient mis hors service lorsqu'un tel vecteur de syndrome est reçu par le processeur de surveillance.

2.7.3. Communication.

Le duo symétrique exige peu de communications pour la gestion des erreurs comparativement à un triplet. En effet, il utilise seulement deux messages pendant la routine d'échange. Cependant, malgré le nombre limité de processeurs, il est possible de vérifier la validité des communications selon le même principe que le triplet symétrique. En effet, puisque chacun des processeurs envoie son message à la destination, on peut de façon analogue au triplet symétrique, obliger les processeurs du duo source à envoyer des messages à chacun des processeurs de la destination. De leur côté, les processeurs destinations devront effectuer une comparaison des messages reçus. Cependant, cette approche demande quelques modifications dans la routine de gestion des erreurs à la réception des messages puisque le duo symétrique ne possède que deux processeurs comparativement à trois pour le triplet symétrique. En effet, puisque les processeurs ne reçoivent que deux messages, il devient donc impossible de déterminer lequel de ces deux messages est fautif lorsque la situation se présente. Il faut donc créer une procédure qui permet au duo destination d'effectuer une requête au duo source pour que ce dernier envoie de nouveau ses messages.

2.7.4. Forces et faiblesses du duo symétrique.

Le duo symétrique est un mode d'opération des plus viables pour le développement d'un nouveau système. En effet, il permet de déterminer l'intégrité des différentes composantes d'un système. L'exécution en mode duo symétrique est très rapide puisque les processeurs échangent seulement deux messages pendant la routine d'échange de messages. De plus, la gestion des temporisateurs peut se réaliser facilement, puisqu'il n'y a que deux processeurs et que ces derniers sont toujours impliqués dans le message qui est en traitement. Lorsque le duo symétrique termine correctement une application, on est certain du résultat obtenu puisqu'il ne possède aucun point critique de défaillance pouvant produire des résultats erronés.

Cependant, le duo symétrique ne peut tolérer une erreur dans un des duos puisqu'il ne possède pas suffisamment d'information pour déterminer avec exactitude la source du problème. Même si le processeur de surveillance pouvait déterminer avec exactitude quel processeur du duo est fautif, il n'est pas souhaitable de mettre hors service ce dernier pour utiliser l'autre processeur en solo. En effet, le fait d'avoir un processeur unique dans un duo introduit un point critique de défaillance. Ceci n'est pas souhaitable si on désire obtenir avec exactitude un résultat. C'est pourquoi la très grande majorité des erreurs détectées lors de l'exécution d'une application dans ce mode sont fatales pour le système.

2.8. L'exécution en mode duo d'un triplet reconfiguré.

Lorsque le processeur de surveillance met hors service un processeur dans un triplet, le processeur de surveillance envoie un message aux deux autres processeurs du triplet afin de les

reconfigurer en mode duo. Cette nouvelle configuration permet au système de continuer l'exécution de l'application sans causer d'erreur fatale au système.

2.8.1. Principe de fonctionnement.

L'exécution en mode duo d'un triplet reconfiguré est basé sur les principes d'opération du mode duo symétrique. Il se voit régi par les mêmes règles en ce qui concerne la gestion des erreurs. Cependant, il existe une différence lors de la transmission du message final à un autre triplet. En effet, on ne peut utiliser le mécanisme du duo symétrique pour l'envoi d'un message à un destinataire puisque la destination n'est pas un duo mais plutôt un triplet. C'est pourquoi on a développé des mécanismes particuliers pour chacun des modes d'opération utilisant le triplet pour la gestion des erreurs. La figure 2.5 montre la gestion des communications pour les modes d'opération basés sur un triplet.

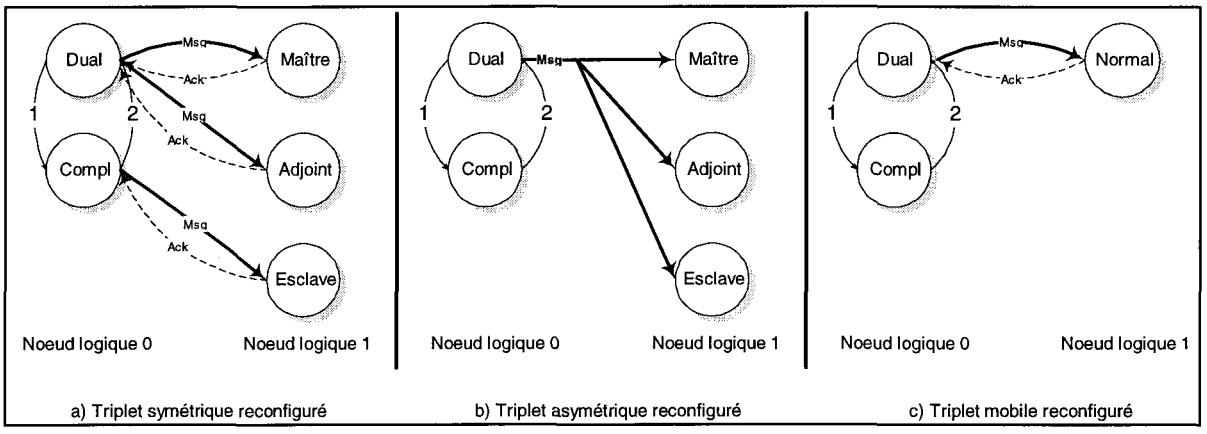

Figure 2.5 : Échange de messages des triplets en mode duo

On remarque que le processeur qui possède la fonction DUAL dans le triplet symétrique est responsable d'envoyer un message au processeur maître et au processeur adjoint afin de combler l'absence d'un des processeurs du triplet. Contrairement au triplet symétrique, le

triplet asymétrique et le triplet mobile font toujours appel à un seul processeur pour envoyer un message à la destination. On notera aussi que le processeur DUAL est responsable de transmettre les accusés réception lors de transferts d'information provenant d'autres triplets. La figure 2.6 montre les stratégies adoptées.

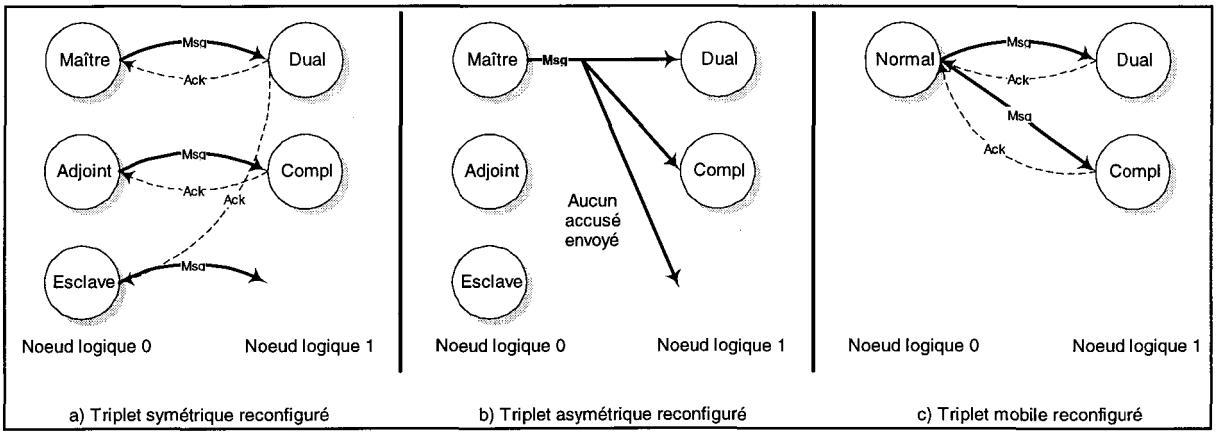

Figure 2.6 : Gestion des accusés de réception des triplets en mode duo

En regardant attentivement la figure 2.6a on s'aperçoit qu'il y a une chance qu'un processeur reçoive un accusé de réception (ACK) avant d'avoir envoyé son message. Cependant, cela n'affectera pas ce dernier, puisqu'il ne regardera pas pour un accusé de réception avant d'avoir envoyé son message. De plus, cette possibilité est très mince, puisque les processeurs sont synchronisés, lorsque viens le temps d'envoyé leurs messages.

Le fait d'utiliser trois stratégies différentes pour la gestion des communications d'un triplet reconfiguré en mode duo est justifié. En effet, le triplet symétrique permet de vérifier la validité des données transférées lors de communications en comparant plus d'un message lors de la réception de ce dernier. Pour ce faire, on doit avoir recours à plus d'un processeur afin d'éviter les points critiques de défaillance. Ceci n'est pas le cas pour le triplet asymétrique et le triplet mobile puisque l'on ne peut effectuer la gestion de communication de cette façon. Il est

donc compréhensible d'utiliser plus d'un mécanisme pour les différents modes d'opération utilisés.

CHAPITRE 3

EXPÉRIMENTATIONS ET OUTILS DE DÉVELOPPEMENT.

3. PRÉAMBULE

L'expérimentation constitue une partie importante d'un travail de recherche. Le choix et la qualité des expériences qui sont effectuées influencent grandement la nature des résultats obtenus. Des expérimentations mal définies auront pour effet d'enlever de la valeur aux résultats et à l'analyse de ces derniers. Dans le présent projet, la partie expérimentale s'est avérée d'autant plus importante qu'elle a permis de vérifier et d'améliorer la robustesse des différents modes d'opération du système d'exploitation. De plus, elle a permis d'étudier le comportement du système multiprocesseur lorsqu'il est soumis à des injections de pannes contrôlées. En effet, à l'aide des expériences, on a pu déterminer les performances réelles du système d'exploitation FATMOS. Ceci a permis de démontrer les forces et les faiblesses de l'approche utilisée pour la détection et la récupération de pannes.

3.1. L'architecture utilisée.

Le but de ce mémoire n'étant pas de justifier l'architecture utilisée mais plutôt le système d'exploitation qui lui est associé, on ne donnera qu'un bref aperçu des caractéristiques de cette première. Le simulateur sur lequel a été effectué les expérimentions est basé sur l'architecture HIBUS, afin de refléter le plus fidèlement possible le système multiprocesseur développé dans le cadre du contrat de recherche avec le ministère de la défense nationale du Canada[ARE91][ARE94]. Ce dernier fut développé à l'École Polytechnique de Montréal dans le cadre du présent projet. L'architecture HIBUS est basée sur une hiérarchie de bus pour permettre la communication entre les différents éléments du système. Les éléments qui la composent donnent une très grande flexibilité à cette dernière. En effet, l'architecture HIBUS

permet d'optimiser l'utilisation des processeurs du système en fonction du mode d'opération utilisé. Par exemple, lorsque le système opère en mode triplet symétrique ou triplet asymétrique, on remarque qu'un certain nombre de processeurs ne seraient jamais utilisés dans certaines architectures conventionnelles puisque les adresses physiques sont fixes pour chacun des processeurs de ces systèmes. Cependant, dans le système HIBUS, les adresses physiques des différents processeurs sont gérées par les aiguilleurs qui se les voient attribuées à l'initialisation du système par le processeur hôte. Puisque les adresses physiques dont le résultat de l'opération modulo quatre est égal à 3 ne sont pas utilisées dans les modes triplet symétrique et triplet asymétrique, ces dernières ne seront tout simplement pas distribuées lors de l'initialisation du système. Ceci permet de former un plus grand nombre de triplets avec les processeurs qui, normalement, ne sont pas utilisés, en leur donnant d'autres adresses qui permettront de former de nouveaux triplets.

a) L'architecture HIBUS modifiée pour les simulations.

Lors des expérimentations sur le simulateur, on a eu recours à une configuration particulière de l'architecture HIBUS. Les raisons qui ont motivé ces configurations sont les suivantes : 1) l'espace mémoire disponible pour simuler le système est proportionnel au nombre d'éléments qui le composent. 2) le temps de simulation augmente proportionnellement avec le nombre de processeurs. 3) toute modification à la configuration rend difficile la comparaison entre les différents modes d'opération. La figure 3.1 montre le schéma de l'architecture pour les tests.

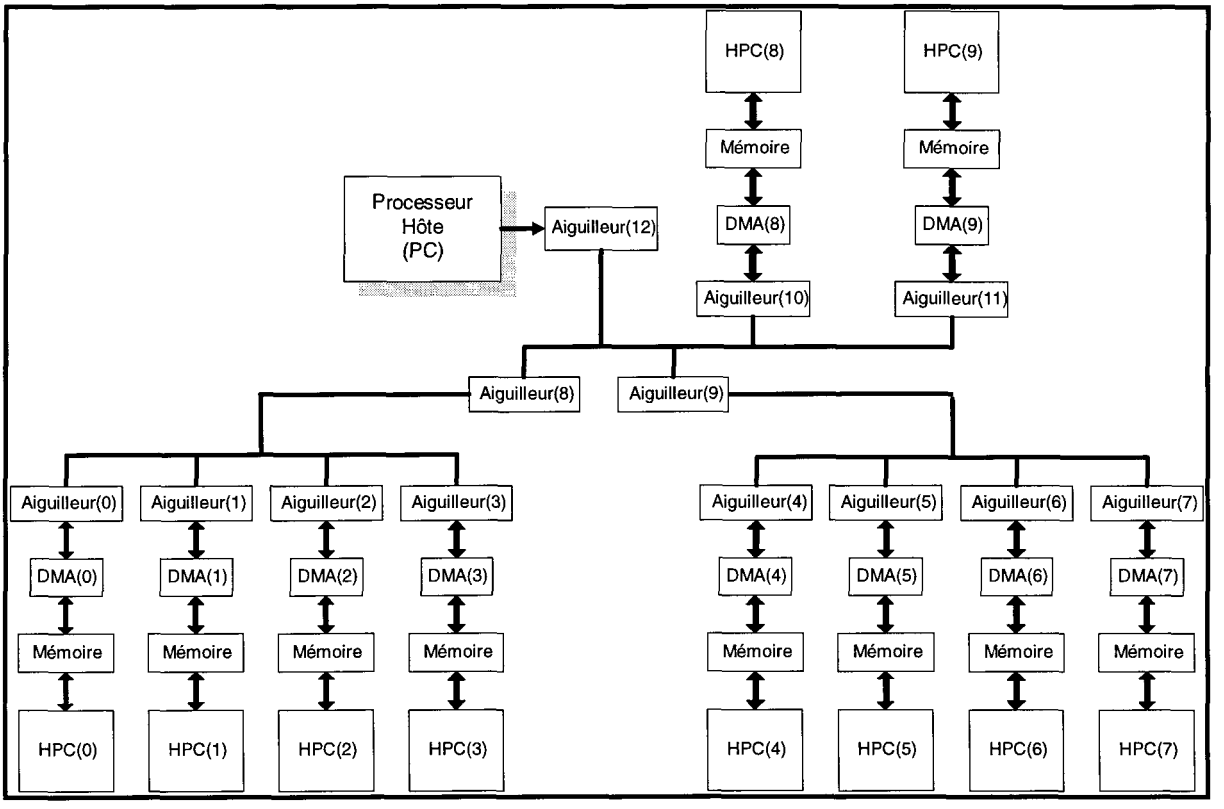

Figure 3.1 : Architecture expérimentale

Il existe une différence importante entre la configuration expérimentale et la configuration réelle du prototype physique. Contrairement à la configuration expérimentale utilisée par le simulateur où l'on retrouve quatre processeurs par bus local, la configuration du prototype physique regroupe les processeurs par groupe de trois sur un même bus local et le nombre de groupe de processeurs est supérieur à celui utilisé pour les simulations.

3.1.2. Les composantes et leurs relations.

Cette section permettra de mettre en lumière les relations entres les différentes composantes du système ainsi que les fonctions qui leurs sont associées pour les expérimentations effectuées.

Le rôle du processeur hôte est assumé par un ordinateur de type compatible IBM qui est relié au système via une interface hôte. Cette dernière n'est pas illustrée. Le processeur hôte constitue l'interface entre l'utilisateur et le système. C'est à l'aide de ce dernier que l'on charge les différentes applications dans la mémoire des processeurs de type HPC. Il a la responsabilité de configurer l'adresse physique des processeurs selon le mode d'opération choisi. De plus, il est responsable des entrées/sorties et de la plupart des ressources du système.

Les processeurs utilisés pour l'exécution d'une application sont du type HPC 16083 de la compagnie National Semiconductor. Les processeurs numérotés de 0 à 7 sont utilisés pour l'exécution des applications. Le processeur #8 exécute la fonction de processeur de surveillance. On lui attribue l'adresse physique 100 afin de le différencier des autres processeurs et ainsi éviter d'avoir à modifier son adresse constamment. Le processeur #9 est utilisé pour l'injection de pannes et l'adresse qui lui est attribuée est 99 pour les mêmes raisons que le processeur de surveillance. Afin d'obtenir des injections de pannes contrôlées, on a recours à un temporisateur qui permet de déterminer les temps d'injections des pannes.

La communication entre les processeurs fait appel à deux composantes bien distinctes. La première, directement liée aux processeurs est l'interface d'accès direct à la mémoire (interface DMA). Lors de la réception d'un message, l'interface DMA écrit le message reçu dans une zone de mémoire (tampon) libre. Cette zone tampon fait partie de la mémoire du processeur. Chaque processeur possède 8 zones pouvant contenir chacune une trame de message ayant une longueur maximale de 32 caractères. Une fois l'écriture de la zone tampon complétée, l'interface envoie un signal d'interruption au processeur lui indiquant qu'un message a été reçu. Une fois le signal d'interruption reçu par le processeur, ce dernier exécute une routine qui lui

permet d'interpréter le message. Dans le cas où un processeur envoie un message, celui-ci écrit dans une zone tampon libre. Par la suite, il envoie un message d'interruption à l'interface lui indiquant qu'il y a un message à envoyer. L'interface prend alors la responsabilité de transmettre ce message.

La deuxième composante nécessaire à la transmission des messages est l'aiguilleur. Il permet de relier le bus de données à l'interface DMA d'un processeur. C'est ce dernier qui détermine si un message provenant de l'extérieur (bus) lui est destiné à partir de l'adresse physique du processeur. De plus, on utilise ces mêmes aiguilleurs pour relier les différents bus. Ceci permet aux processeurs de recevoir et d'envoyer des messages à n'importe quel processeur du système peu importe son emplacement dans la hiérarchie de bus.

3.2. Les conditions expérimentales.

Les expérimentations ont été réalisées à l'aide d'un simulateur développé dans le cadre du projet de recherche avec le ministère de la défense nationale[ARE91][ARE94]. Le simulateur fonctionne sous le système d'exploitation MS-DOS et sur des ordinateurs compatibles IBM (PC). L'utilisation du simulateur demande des processeurs très performants si l'on désire obtenir les résultats dans un laps de temps raisonnable. Les PCs utilisés pendant la présente série d'expérimentations étaient dotés de processeur Pentium opérant à une fréquence d'horloge de 90MHz. Le temps nécessaire pour effectuer une série d'expérimentations complète est d'environ 48 heures. L'architecture utilisée dans l'ensemble des expérimentations a été présentée à la section 3.1.

Puisque le temps était un facteur important lors des expérimentations, on a donc décidé d'utiliser seulement deux programmes pour l'ensemble des expérimentations. Ces derniers sont détaillés dans la section suivante. Les programmes sont de petite taille puisque les processeurs utilisés ne possèdent qu'un espace mémoire total de 64k. La plage sur laquelle ont porté les injections de pannes débute à l'adresse 0x8000 et elle se termine à l'adresse 0xFFFF. Cette plage mémoire contient la majeure partie des programmes puisque ces derniers ont été compilés avec des options les contraignant à être localisés dans cette plage. De plus, lors de la compilation des programmes, on s'est assuré que les différentes zones de codes (instructions, données, ...) ne serait pas entrelacées afin d'améliorer la compréhension du comportement du système. Lors des expérimentations, une partie de la mémoire contenant une zone d'instructions n'a pas fait l'objet d'injections de pannes. Pour des raisons de temps, on n'a pas inclus cette partie de mémoire dans notre étude puisqu'elle est de taille peu significative comparativement à la plage mémoire étudiée. Cependant, elle pourra faire l'objet d'études plus approfondies ultérieurement.

La plage mémoire sur laquelle des pannes ont été injectées était divisée en 8 sections. Ces sections étaient de taille identique. Les prochaines sections donneront des détails sur les programmes qui ont été choisis, sur les différents mécanismes d'injection de pannes qui ont été développés ainsi que sur le mécanisme d'analyse automatique des résultats.

3.3. Les programmes ayant servi aux expériences.

Le but des expérimentations était de caractériser le comportement d'un système multiprocesseur lorsque ce dernier est soumis à une injection de pannes. De plus, elles

devaient permettre d'observer le fonctionnement des différents modes d'opération. Afin d'obtenir des résultats qui reflètent la réalité, on devait utiliser différents programmes.

Pour les raisons mentionnées dans la section précédente, seulement deux programmes ont été utilisés pour les expérimentations. Bien que ces deux programmes ne permettent pas d'étudier en détails tous les aspects du système, ils ont permis d'effectuer une bonne évaluation du fonctionnement des différents modes d'opération du système d'exploitation.

3.3.1. Le calcul de la constante π.

Le premier programme qui a été développé permet de déterminer la valeur de la constante π. La valeur de la constante π peut être obtenue à l'aide de l'équation 3.1. Afin de permettre le calcul de l'intégrale par ordinateur, on fait appel à une méthode d'analyse numérique. La méthode choisie est celle des rectangles. L'équation 3-2 est une expression simplifiée pour le calcul de l'intégrale par la méthode des rectangles.

$$\pi = \int_0^1 \frac{4}{1+x^2} \tag{3.1}$$

$$\pi = \sum_{i=0}^{n} \left(\frac{4}{1+\left(\frac{2i+1}{2n}\right)^2} \right) * \frac{1}{n} \tag{3.2}$$

On peut remarquer que la précision du résultat de l'équation 3-2 dépend de la valeur de n. En effet, plus n est élevé et plus le résultat obtenu sera précis. Cependant, l'équation 3.2 n'est pas adaptée au calcul multiprocesseur. Lorsque ce calcul est réalisé à l'aide d'un système multiprocesseur, il est important que les différents processeurs impliqués dans le calcul procèdent à l'évaluation d'une partie du calcul de l'intégrale. Lorsque le nombre de processeurs impliqués dans le calcul est faible, il peut être tentant d'écrire un programme différent pour chacun des noeuds. Cependant, l'utilisation d'une telle pratique augmente le risque d'erreurs causées par de nombreuses manipulations dans les fichiers source des programmes. L'équation 3.3 donne l'expression générale pour le calcul à l'aide d'un système multiprocesseur. Cette équation est valide peu importe le nombre de processeurs impliqués dans le calcul.

$$\pi_{pr} = \sum_{i=pd}^{pf} \left(\frac{4}{1+\left(\frac{2i+1}{2n}\right)^2} \right) * \frac{1}{n} \quad ; pd < x_i \leq pf \tag{3.3}$$

Les variables pd et pf représentent respectivement le point de départ et le point final de la plage qui est assignée à un processeur. L'utilisation d'un système multiprocesseur pour résoudre ce genre de problème est très efficace à condition que le nombre d'itérations requis soit suffisamment grand. En effet, l'utilisation d'un système multiprocesseur pour un petit nombre d'itérations peut demander plus de temps d'exécution que le temps requis avec un système possédant un seul processeur. Ceci est dû aux délais de communication qui peuvent

devenir non-négligeables dans l'exécution d'une application. Cependant, dans le présent projet, on a négligé de tenir compte de ce facteur puisque le but de ce dernier n'était pas de déterminer le temps nécessaire au système pour exécuter une application, mais plutôt l'efficacité et le comportement de ce dernier lorsqu'il est soumis à l'injection de pannes.

3.3.2. Le problème du placement des reines.

Le second programme utilisé pour caractériser l'efficacité du système d'exploitation est celui du placement des reines. Le problème des reines consiste à déterminer le nombre de possibilités pour la disposition de N reines sur un échiquier de dimension N*N sans que ces dernières se trouvent en possibilité d'être attaquées. Sachant qu'une reine peut attaquer selon l'axe des X, l'axe des Y et dans les directions diagonales, le calcul de la disposition des reines devient un problème typique auquel il n'y a pas de solution analytique (NP complet). En effet, il existe un grand nombre de solutions possibles. Cependant, le seul moyen de déterminer toutes les dispositions possibles de N reines sur un échiquier de dimension N*N est de les essayer toutes.

La difficulté du placement des reines réside dans le grand nombre de possibilités à évaluer. Par exemple, un échiquier de dimension 8*8 possède 64 cases, le nombre total de possibilités pour lesquelles on peut disposer 8 reines peut être obtenu par la formule[IPS90] suivante:

$$N = n! / (m!*(n-m))!$$

où n représente le nombre de cases dans l'échiquier et m représente le nombre de reines qu'on désire disposer. On constate que le nombre de possibilités de placement croît

extrêmement rapidement en fonction de n. Ceci peut exiger des temps de calcul importants pour des ordinateurs conventionnels. Cependant, il est possible de diminuer grandement le nombre de cas pour lequel on doit évaluer la disposition des reines, sachant qu'on ne peut placer plus d'une seule reine par colonne sur l'échiquier. L'approche utilisée pour résoudre le problème des reines est illustrée à la figure 3.2.

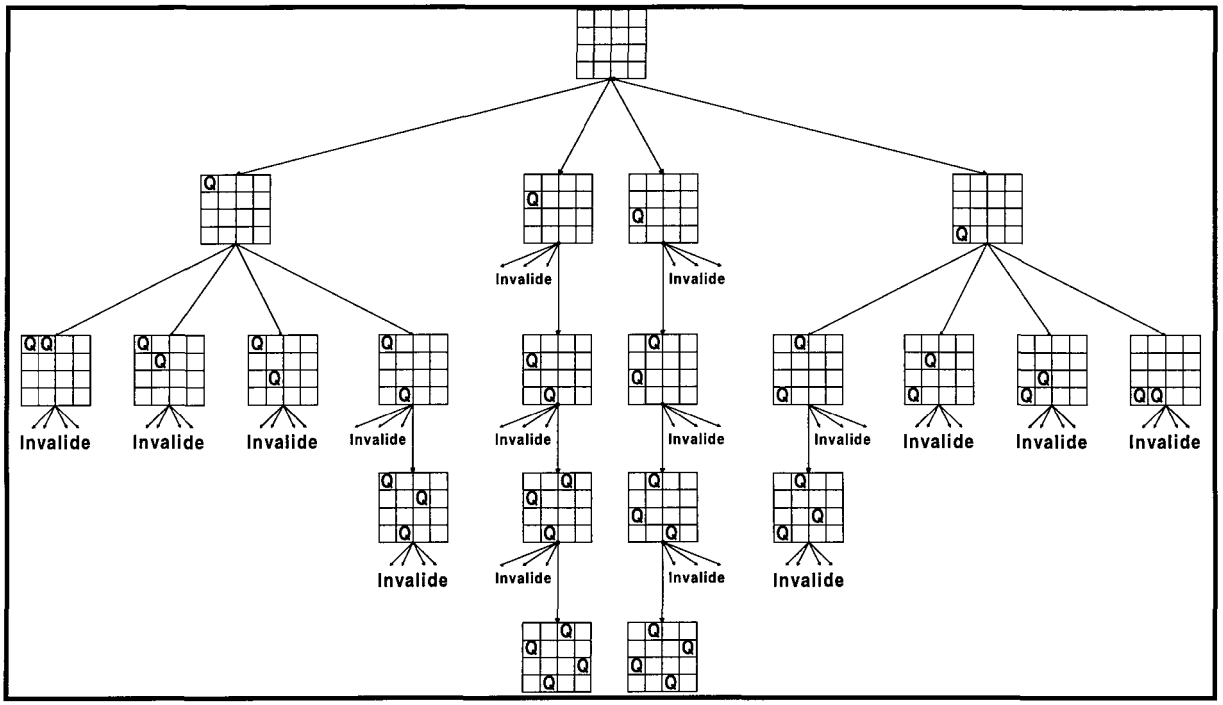

Figure 3.2 : Exemple de placement des reines pour un échiquier de dimension 4*4

Il est encore possible de diminuer de moitié le nombre total de dispositions à évaluer. En effet, en observant la configuration d'un échiquier (figure 3.2), on constate qu'il existe une symétrie selon un axe horizontal (ou vertical) de ce dernier. En effet, il suffit de déterminer le nombre de possibilités pour la première moitié de l'échiquier et de multiplier ce résultat par 2 pour obtenir le nombre total de dispositions pour un échiquier de dimension N*N. Cependant, si le nombre de cases par colonne est impair, il faut effectuer une légère

modification au calcul. En effet, seule la ligne (ou colonne) centrale ne sera pas multipliée par 2 pour obtenir le résultat final puisqu'elle ne possède pas de symétrie selon l'axe horizontal (ou vertical). Cette diminution du nombre de dispositions à évaluer permet d'accélérer considérablement le temps de calcul pour le problème des reines. De plus, cette dernière modification permet d'utiliser moins de processeurs lors de l'utilisation d'un système parallèle pour résoudre ce problème.

3.4. Les mécanismes d'injection de pannes.

L'étude du comportement de FATMOS sous l'effet de pannes implique le développement de mécanismes permettant de créer des pannes similaires à celles pouvant survenir dans un système multiprocesseur. Il existe plusieurs types de pannes pouvant affecter un système multiprocesseur. Dans le présent projet, les efforts ont été concentrés sur trois types de pannes en particulier. Le premier type de panne consiste à inverser un bit dans une adresse mémoire donnée "Bit Flip", le second consiste à simuler une panne dans le décodeur d'instruction du processeur et le troisième consiste à simuler une panne dans le décodeur d'adresse du processeur. Bien sûr, plusieurs autres types de pannes peuvent survenir. Cependant, les trois types choisis permettent d'étudier un système lorsque des pannes surviennent à des endroits critiques.

Comme il a été mentionné dans le chapitre 1, on a préféré une approche logicielle pour réaliser les injections de pannes. On a recours à l'approche logicielle afin d'avoir un meilleur contrôle sur le temps d'injection d'une panne ainsi que l'endroit où elle est introduite dans le

système. Ceci, permet de recréer la même panne autant de fois que nécessaire lorsqu'on désire obtenir plus d'informations sur le comportement réel du système.

Il existe cependant des points importants à souligner lorsqu'on utilise une telle approche pour simuler des pannes. L'injection logicielle de pannes implique certaines contraintes. En effet, contrairement à l'injection matérielle, une injection logicielle influence le déroulement normal d'un traitement. En effet, on doit effectuer une requête (à l'aide d'un message) au processeur ciblé afin qu'il exécute la routine d'injection de panne. Ceci implique que le processeur interrompt le cours normal de son programme. Une telle perturbation peut être critique pour tous les modes de détection de pannes puisque ces derniers font appel à des temporisateurs pour valider les délais de transmission des messages lors de l'exécution de la routine de comparaison. Ainsi la requête d'injection peut retarder l'arrivée d'autres messages et provoquer de fausses erreurs. C'est pourquoi, une routine permettant d'arrêter tous les processeurs du système de façon simultanément a été développée. Les processeurs sont remis en marche après que le processeur ciblé ait confirmé qu'il a injecté la panne ou lorsque le temporisateur du processeur responsable des injections vient à expiration. Ce dernier est très utile afin d'éviter que le système demeure en attente lorsque le processeur ciblé ne répond plus. L'utilisation de cette routine permet de simuler un grand nombre de pannes en assurant une très grande fiabilité au système. On notera cependant que la requête d'injection doit tout de même traverser le réseau de communication avant d'arriver au processeur ciblé et, de ce fait, modifiera le comportement de ce dernier par rapport à ce qu'il aurait été si la requête n'avait pas eu lieu. Il n'y a cependant aucun moyen permettant de compenser cet effet. Ceci rappelle un principe bien connu en physique : le principe d'Heisenberg.

Les mécanismes d'injection de pannes font partie intégrante du système d'exploitation. Les trois types de pannes qui peuvent être simulées modifient le contenu de la mémoire des processeurs. Pour un processeur, il est impossible de déterminer si la panne est survenue dans la mémoire ou dans un de ses registres. Cependant, il existe des mesures de protection permettant de vérifier si l'erreur provient d'une composante interne ou d'une composante externe (Exemple, la vérification de la parité de la mémoire). Les sous-sections suivantes décrivent les opérations réalisées pour chaque type de pannes considérées.

3.4.1. L'inversion d'un bit à une adresse mémoire donnée.

L'inversion d'un bit en mémoire est un symptôme souvent associé à l'effet de radiations ionisantes. Les systèmes lancés dans l'espace sont évidemment très susceptibles de subir de telles inversions. Simuler l'injection d'une inversion de bit demande peu de ressources pour être réalisée de façon logicielle. En effet, il suffit de complémenter un bit dans une adresse mémoire spécifique.

L'injection de pannes dans la mémoire permet de simuler un certain nombre de pannes pouvant survenir dans les composantes internes d'un processeur. En effet, il est généralement difficile et souvent impossible pour un processeur de déterminer exactement d'où provient une panne et à quel moment cette dernière s'est introduite dans le système. Ainsi, peut être donc impossible de déterminer si une panne provient de la mémoire d'un processeur ou de l'une de ses composantes internes. Il est plus difficile d'effectuer une inversion de bit dans le décodeur d'instructions du processeur plutôt que d'effectuer cette inversion directement sur la case mémoire qui contient cette instruction. Il existe néanmoins une lacune à l'utilisation de la

mémoire pour réaliser une inversion de bit. La panne introduite dans le système doit être latente pour être efficace. En effet, si l'injection d'une panne se produit après que l'instruction où une panne est injectée ait été exécutée, elle n'aura aucune influence sur le déroulement du programme.

L'inversion de bits permet d'injecter des pannes dans plusieurs composantes du système. Elle permet de simuler des pannes dans la mémoire et le processeur, de même que dans les composantes servant à communiquer avec les autres processeurs du système. De plus, il est possible de simuler des pannes dans le bloc d'alimentation. Cette dernière requiert quelques modifications dans la routine d'inversion de bits. Une panne du bloc d'alimentation implique le changement d'état de plusieurs bits dans des adresses mémoire et dans les différentes composantes du processeur. Le nombre de composantes affectées par une panne du bloc d'alimentation dépend de la durée de la perturbation. En effet, plus la perturbation est longue et plus le nombre de composantes affectées est grand. Afin de simuler ce type de pannes, on a recours à un générateur de nombres aléatoires qui permet de déterminer les adresses qui seront affectées. De plus, il faut que la fonction responsable de simuler ce type de panne tienne compte du temps que doit durer la panne et des statistiques associées à ce genre de panne.

3.4.2. Les pannes dans le décodeur d'instruction.

L'injection logicielle d'une panne dans le décodeur d'instruction est moins efficace que l'injection matérielle. En effet, il est impossible de modifier le fonctionnement d'une composante physique de façon logicielle. On ne peut que tenter de simuler le plus fidèlement

possible certains types de pannes. Dans la présente étude, nous considérerons deux types de pannes pouvant survenir dans le décodeur d'instruction. La première est l'exécution d'une mauvaise instruction et la seconde est le collage ("stuck-at fault") d'un des bits du registre manipulant les instructions.

La simulation de l'exécution d'une mauvaise instruction peut être réalisée en affectant la mémoire de la façon suivante. On recherche toutes les instructions correspondant à celle déterminée par l'utilisateur et on remplace cette dernière par une autre instruction, elle aussi déterminée par l'utilisateur. Il est important de noter que seule la section de mémoire contenant le code du programme doit être affectée par ce type de panne. Il est donc important de forcer le compilateur à regrouper les zones d'instructions des différentes parties du programme en un seul bloc afin d'éviter d'affecter des zones mémoire qui ne contiennent pas d'instruction.

Le collage d'un bit du registre d'instruction demande un peu plus de travail pour être réalisée. Cependant, il exige peu de ressources. En effet, le principe est simple, puisqu'il suffit d'appliquer un masque sur le contenu d'une adresse mémoire. Le masque appliqué permet de forcer à 0 ou à 1 le bit désiré dans l'adresse mémoire. L'opération nécessite que chacune des adresses mémoire soit modifiée. Comme pour la simulation d'une mauvaise instruction, il est important d'exécuter les modifications uniquement sur les zones mémoire comportant des instructions.

3.4.3. Les pannes dans le décodeur d'adresse.

Plusieurs types de panne peuvent survenir dans le décodeur d'adresses. Cependant, nous ne considérerons qu'un seul type de panne, soit celles pouvant être simulées en collant un bit. Comme pour la simulation d'un bit collé dans le décodeur d'instruction, on fait appel à des masques pour réaliser l'opération. Cependant, l'opération effectuée sur l'espace mémoire est bien différente de celle effectuée dans le cas du décodeur d'instruction. L'application du masque s'effectue sur l'adresse mémoire et non pas sur le contenu de cette dernière. En effet, une panne dans le décodeur d'adresse affecte les adresses mémoire. Ainsi on doit utiliser deux pointeurs pour réaliser l'opération. Le premier pointeur pointe sur l'adresse source dans la mémoire. Le deuxième pointeur pointe sur l'adresse obtenue après application du masque désiré sur l'adresse source. Une fois l'adresse du deuxième pointeur obtenue, on copie le contenu du deuxième pointeur dans l'adresse du premier pointeur. Cette opération s'effectue pour l'ensemble de la mémoire du système. En effet, contrairement aux pannes dans le décodeur d'instruction qui sont restreintes à la zone mémoire contenant les instructions, les pannes dans le décodeur d'adresses couvrent toutes les zones mémoire sans distinction particulière.

L'injection logicielle de pannes dans le décodeur d'instructions ne permet pas de simuler tous les types de pannes pouvant être réalisées de façon matérielle. Cependant, il faut bien comprendre que ce type de panne est généralement fatal pour les processeurs, puisque ces derniers ne peuvent exécuter correctement les instructions contenues dans la mémoire du processeur. De plus, l'injection de pannes de façon matérielle sur des systèmes multiprocesseurs s'avèrent très onéreuses et demandent beaucoup de ressources matérielles.

Puisque l'implantation d'outils permettant d'injecter ces pannes de façon logicielle ne nécessite qu'un faible coût et peut être implanté rapidement sur des systèmes multiprocesseurs ou des systèmes distribués tout en donnant une simulation acceptable du phénomène désiré, il est donc dans l'intérêt du concepteur d'utiliser cette approche pour mettre à l'épreuve son système.

3.5. Automatisation des expérimentions.

Le développement du système d'exploitation FATMOS ainsi que les nombreuses expérimentations nécessaires à la réalisation de l'étude statistique du comportement des différents modes d'opérations du système ont nécessité le développement d'outils permettant d'obtenir des résultats fiables en évitant toute erreur de manipulation. En effet, pas moins de 2704 expérimentations par programmes ont été effectuées pour chacun des modes d'opérations du système d'exploitation. De plus, ces expérimentations ont été répétées plusieurs fois lors du développement du système d'exploitation. Deux programmes pour automatiser le processus ont été développés à cette fin. Le premier est un programme de génération des résultats et le second un programme permettant une analyse primaire de ces derniers.

3.5.1. Le programme de génération des résultats.

Le programme de génération des résultats permet de contrôler les expérimentations qui sont effectuées sur les différents programmes de test (Ex, le calcul de la constante π). Étant donné le grand nombre de simulations effectuées afin d'obtenir des résultats significatifs, on se doit de développer de tels outils. Le but de ce programme est d'automatiser les injections de

façon à pouvoir contrôler et recréer les différentes simulations en tout temps. Ceci permet de reprendre l'analyse lorsque le programme d'analyse primaire ne peut fournir toutes les informations désirées.

Le fonctionnement du programme fait appel à plusieurs fonctions qui permettent une grande souplesse à l'utilisateur. Les différentes fonctions permettent de modifier un fichier source contenant le programme à simuler, d'assurer qu'il est compilé dans le bon mode d'opération, de créer les répertoires nécessaires à l'enregistrement de l'information suite à l'expérimentation et de contrôler la génération des résultats. Lorsqu'on désire effectuer une série d'expérimentations, il faut donc spécifier plusieurs paramètres au programme. Une fois les paramètres définis, le programme détermine si ces derniers sont valides. Lorsque tous les paramètres sont vérifiés, le programme crée les répertoires nécessaires à l'enregistrement des données, crée un fichier contenant l'information sur les différentes expérimentations effectuées pendant l'exécution et il effectue les expérimentations. La façon de procéder pour générer les différentes expérimentations est la suivante. Chaque programme étant divisé en 8 plages mémoires, la plage mémoire qui est donnée en paramètre au programme est divisée par 26. Ceci permet de déterminer les 26 adresses mémoire qui subiront une injection de pannes. Chacune des adresses sélectionnées sera évaluée dans le temps à 13 reprises. Ces évaluations temporelles permettront de déterminer à quel moment l'injection d'une panne devient critique pour un processeur et pendant combien de temps il y est sensible selon le programme utilisé.

3.5.2. Le programme d'analyse primaire.

Le programme d'analyse primaire a été développé pour accélérer l'interprétation des résultats et éliminer les risques d'erreurs de manipulation. L'analyse effectuée est primaire puisque l'information contenue dans les résultats n'est que partielle. L'analyseur décortique chaque fichier selon un ordre bien précis. Cet ordre suit l'évolution des injections de pannes dans le temps. De plus, l'analyseur permet de déterminer quels types d'erreurs sont survenus pendant l'exécution. Un registre des analyses effectuées est conservé dans un fichier ainsi que les statistiques qui lui sont associées. Les statistiques recueillies permettent de déterminer le pourcentage des erreurs qui ont affectées le système, les zones de code qui sont les plus sensibles aux pannes et bien d'autres informations qui permettent une meilleure compréhension du comportement du système. Cependant, l'analyseur ne permet pas de tout vérifier. En effet, une vérification minutieuse doit être effectuée pour tous les cas comportant des erreurs. Ces erreurs peuvent en effet provenir du simulateur de système utilisé pour tester FATMOS et ainsi fausser les résultats obtenus. En effet, puisque ce simulateur n'est pas le véritable système, des erreurs de programmation peuvent survenir (un logiciel parfait n'existe pas). De plus, puisque les informations contenues dans les résultats sont partiels il peut s'avérer qu'une erreur soit mal interprétée dû au manque d'information contenue dans les fichiers de données.

CHAPITRE 4

RÉSULTATS ET ANALYSE.

4. PRÉAMBULE

Ce chapitre a pour objectif de quantifier la robustesse du système multiprocesseur avec et sans les mécanismes de tolérance aux pannes décrits au chapitre 2. Un certain nombre de sujets feront l'objet d'une attention particulière : la sensibilité des programmes aux pannes, le gain en fiabilité lorsqu'un mécanisme de tolérance aux pannes est associé à une application, l'effet sur la performance des différents modes d'opération, la sensibilité des différentes zones de mémoire selon le programme exécuté et l'effet de la granularité de l'injection de pannes sur la qualité des résultats obtenus.

Avant d'aller plus loin dans l'analyse des résultats, il est bon de définir quelques termes utilisés tout au long de ce chapitre. Ainsi, on utilisera le terme "erreurs observées" pour définir toutes les erreurs dont on a pu observer la trace et qui n'ont eu aucune influence sur le système. Ce type d'erreurs est difficilement détectable et affecte normalement des portions de code utilisés pour des fin de déverminage ("débugging"). On utilisera également le terme "erreurs non détectées" pour définir les erreurs qui ont affectées le comportement d'un programme mais qui n'ont pu être détectées par le système d'exploitation. Le terme "erreurs détectées" sera utilisé pour définir les erreurs qui ont affectées le comportement d'un programme et qui ont été détectées par le système d'exploitation. Finalement, on utilisera le terme "erreurs perceptibles" pour définir l'ensemble des erreurs qui sont survenues dans un programme.

4.1. La sensibilité des programmes aux pannes.

Il existe plusieurs études sur l'effet des pannes dans les composantes d'un système. La majeure partie d'entre elles se consacrent à un élément bien précis du système et non pas à l'effet sur l'ensemble. Cependant, ces études ont permis de mettre en lumière certaines caractéristiques qui

sont propres aux types de fautes injectées ou à la composante sur laquelle ont été effectuées les injections de pannes. Par exemple, l'étude de Karlsson et al. [KAR 94] a démontré que les inversions de bit subies lors de l'exposition d'un microprocesseur MC6809E à des radiations n'étaient observées que pour le passage de l'état logique l à un état logique 0. D'autres études, comme celle de Clark et al. [CLA95], ont permis de caractériser les temps de latence et de dormance ainsi que d'autres paramètres associées à certains types de pannes. Cependant, il semble qu'aucune d'entre elles n'a permis de quantifier l'effet réel de ces dernières sur le comportement d'un programme. Les résultats de cette section apportent un éclairage nouveau sur l'influence de ce type de pannes.

L'étude de la sensibilité aux pannes a donné des résultats plutôt intéressants. Ainsi, le tableau 4.1a présente les pourcentages de pannes ayant produit des erreurs et, par conséquent ayant affecté le fonctionnement des deux programmes qui ont servi à cette étude. Les résultats obtenus dans ce tableau sont classés selon les cinq modes d'opérations et tiennent compte des erreurs recueillies sur la totalité des pannes ayant été injectées lors des expérimentations effectuées. Dans ce tableau, ainsi que tout au long du présent chapitre, l'expression "mode Normal" correspond à un fonctionnement sans mécanisme de tolérance aux pannes. Le tableau 4.1b, quant à lui, donne les pourcentages des pannes ayant produit des erreurs lorsque ces dernières ont été injectées dans des zones mémoire (appelées zones actives) utilisées par les programmes. Ainsi le tableau 4.1a donne une meilleure représentation de la réalité, puisqu'il tient compte des espaces mémoire inoccupé lors de l'exécution d'un programme quelconque. Cependant, le tableau 4.1b permet de déterminer la sensibilité des programmes proprement dits. Toutefois, le tableau 4.1b n'indique pas si les adresses mémoire visées contenaient des données ou des instructions nécessaires à l'exécution des programmes.

Tableau 4.1a : Pourcentage des pannes injectées dans toute la mémoire ayant causé des erreurs.

Programme	Mode d'opération				
	Normal	Triplet Symétrique	Triplet Asymétrique	Triplet Mobile	Duo Symétrique
π	6.2%	4.4%	3.2%	3.4%	9.5%
REINE	2.5%	5.1%	5.0%	4.5%	2.9%

Tableau 4.1b : Pourcentage des pannes injectées dans les zones actives ayant causé des erreurs.

Programme	Mode d'opération				
	Normal	Triplet Symétrique	Triplet Asymétrique	Triplet Mobile	Duo Symétrique
π	13.0%	5.5%	4.1%	4.1%	15.2%
REINE	5.6%	6.6%	6.5%	5.6%	4.9%

Lorsqu'on regarde attentivement les résultats du tableau 4.1a et ceux du tableau 4.1b, on s'aperçoit qu'un très faible pourcentage des pannes injectées dans un programme a une influence lors de l'exécution de ce dernier. En effet, les résultats obtenus pendant les expérimentations sur les différents mode d'exploitation, démontrent que moins de 16% (tableau 4.1b Duo Symétrique) des pannes injectées dans les zones mémoire du programme π et moins de 7% (tableau 4.1b Triplet Symétrique) dans celles du programme des reines ont produit des erreurs de fonctionnement. Si l'on reporte ces résultats sur l'ensemble de la plage mémoire ayant subit des injections de pannes, alors ces nombres deviennent inférieurs à 10% (tableau 4.1a Duo Symétrique) pour le programme π et légèrement supérieurs à 5% (tableau 4.1a Triplet Symétrique) pour le programme des reines. Ceci est très peu si l'on considère les chances qu'un système soit affecté par de telles pannes. Ainsi, on peut

donc dire que les résultats obtenus démontrent que les programmes sont relativement peu sensibles aux pannes.

Un autre point d'intérêt, sur l'étude de la sensibilité aux pannes, est la relation entre l'accroissement de la taille d'un programme et l'augmentation du nombre de pannes qui peuvent affecter son fonctionnement. Les tableaux 4.2a et 4.2b donnent le nombre d'erreurs recueillies dans les cinq modes d'opération pour les deux programmes utilisés. De plus, on y retrouve la taille des programmes selon les différents modes d'opération ainsi que le rapport de l'accroissement de la taille par rapport au mode d'opération normal. Il est important de préciser que, dans le cas présent, on se doit de tenir compte de toutes les erreurs qui ont été observées lors de l'exécution des programmes. Ainsi, on tient compte des erreurs qui ont été corrigées à l'aide des mécanismes de tolérances aux défaillances. En effet, ce genre de faute aurait probablement affecté le fonctionnement du programme si l'accroissement de la taille avait été relié à autre facteur que l'ajout de mécanismes de tolérance aux défaillances.

Tableau 4.2a : % Nombre d'erreurs perceptibles versus taille du code des modes d'opération pour le calcul de la constante π.

	Mode d'opération				
	Normal	**Triplet Symétrique**	**Triplet Asymétrique**	**Triplet Mobile**	**Duo Symétrique**
Nombre d'erreurs perceptibles	170	120	89	92	257
Taille du programme (en hexadécimal)	0x3DD9	0x66E0	0x62E3	0x6687	0x5010
Rapport taille code / mode normal	1.00	1.66	1.60	1.66	1.29

Tableau 4.2a : % Nombre d'erreurs perceptibles versus taille du code des modes d'opération pour le problème des reines.

	Mode d'opération				
	Normal	**Triplet Symétrique**	**Triplet Asymétrique**	**Triplet Mobile**	**Duo Symétrique**
Nombre d'erreurs perceptibles	67	107	136	122	80
Taille du programme (en hexadécimal)	0x39D2	0x62D9	0x5EDC	0x6280	0x4C09
Rapport taille code / mode normal	1.00	1.71	1.64	1.70	1.32

En regardant attentivement les résultats obtenus, on s'aperçoit que le nombre d'erreurs perceptibles ne semble pas uniquement relié à l'augmentation de la taille du programme π. En effet, pour les modes d'opération triplet symétrique, triplet asymétrique et triplet mobile, le nombre d'erreurs perceptibles est inférieur à celui obtenu pour le mode normal et ce malgré une augmentation de la taille qui est supérieure à 1.6 fois celle du mode normal. Cependant, lorsqu'on calcule π en mode duo symétrique, l'augmentation du nombre d'erreurs perceptibles devient supérieure à l'augmentation de sa taille par rapport au mode normal. Lorsqu'on observe les résultats obtenus pour le programme des reines, on a un autre son de cloche. En effet, dans ce cas, l'augmentation du nombre d'erreurs perceptibles est sensiblement la même que l'augmentation de la taille du programme

Les résultats sur le nombre d'erreurs perceptibles en fonction de l'augmentation de la taille d'une application ne permettent pas de déterminer avec certitude la relation entre ces deux facteurs. En effet, une des hypothèses qui pourraient expliquer le manque de cohérence entre les résultats sera discutée à la section 4.6.

4.2. Gain en fiabilité avec l'utilisation de mécanismes de tolérance aux défaillances.

L'objectif principal du présent projet étant de développer un système d'exploitation tolérant aux défaillances, il est donc normal d'y consacrer une section du présent chapitre. Lors des expérimentations chacun des modes d'opérations des deux programmes étaient soumis à 2704 injections de pannes. Les tableaux 4.3a et 4.3b montrent le nombre d'erreurs obtenues pour le calcul de la constante π et pour le problème des reines respectivement. Dans ces tableaux, on a classé les erreurs en cinq catégories. La première catégorie représente les erreurs ayant affecté une partie du programme mais qui n'ont pas eu d'influence sur le résultat final (erreurs observées sans incidence sur le système). Exemple : Lorsqu'un processeur devient fautif après l'exécution de la routine de

comparaison, il se peut qu'il ne complète jamais l'exécution de son programme sans que cela n'influence le système. La deuxième catégorie d'erreurs regroupe celles ayant causé l'arrêt du système sans qu'elles ne soient détectées (erreurs non détectées générant l'arrêt du système). Dans la troisième catégorie d'erreurs, on retrouve celles ayant causé l'arrêt du système malgré leur détection (erreurs détectées générant l'arrêt du système). Comme on peut le constater, les deuxième et troisième catégories sont étroitement liées et dépendent principalement du moment où elles sont survenues durant l'exécution. La quatrième catégorie regroupe les erreurs ayant généré de mauvais résultats à la fin de l'exécution du programme (erreurs détectées générant un mauvais résultat). On retrouve généralement ce genre d'erreurs dans les systèmes qui ne possèdent pas de mécanisme de tolérance aux défaillances. Finalement, dans la cinquième catégorie, on présente le nombre d'erreurs ayant été détectées et récupérées par le système (erreurs détectées et récupérées). Cette catégorie d'erreurs met en évidence l'efficacité des différents mécanismes.

Tableau 4.3a : Résultats des différents modes d'opération pour le calcul de la constante π.

Nom du programme	**Calcul de la constante π**				
Mode d'opération	**Normal**	**Triplet Symétrique**	**Triplet Asymétrique**	**Triplet Mobile**	**Duo Symétrique**
1- Nombre d'erreurs observées sans incidence sur le résultat final	35	29	4	22	13
2- Nombre d'erreurs non-détectées générant l'arrêt du système	82	1	0	2	54
3- Nombre d'erreurs détectées générant l'arrêt du système	0	0	9	0	190
4- Nombre d'erreurs non-détectées générant un mauvais résultat	53	0	0	0	0
5- Nombre d'erreurs détectées et récupérées	0	90	76	68	0
Nombre total d'erreurs	**170**	**120**	**89**	**92**	**257**

Tableau 4.3b : Résultats des différents modes d'opération pour le programme résolvant le problème des reines.

Nom du programme	**Problème des reines**				
Mode d'opération	**Normal**	**Triplet Symétrique**	**Triplet Asymétrique**	**Triplet Mobile**	**Duo Symétrique**
1- Nombre d'erreurs observées sans incidence sur le résultat final	9	28	8	0	2
2- Nombre d'erreurs non-détectées générant l'arrêt du système	46	0	3	0	22
3- Nombre d'erreurs détectées générant l'arrêt du système	0	0	52	1	56
4- Nombre d'erreurs non-détectées générant un mauvais résultat	14	0	0	0	0
5- Nombre d'erreurs détectées et récupérées	0	79	73	121	0
Nombre total d'erreurs	**67**	**107**	**136**	**122**	**80**

Afin de déterminer l'efficacité des différents modes d'opération, on doit comparer les résultats de ces derniers avec ceux obtenus pour le mode d'opération normal. Ainsi, on peut comparer le nombre d'erreurs critiques pour le fonctionnement du programme dans chacun des modes d'opération. Cette comparaison permet d'évaluer la valeur réelle du rendement de chacun des mécanismes de tolérance aux défaillances implantés dans le système d'exploitation. En effet, sans l'utilisation d'unité étalon, il serait impossible de déterminer les performances réelles d'un mécanisme ainsi que sa viabilité.

Les résultats obtenus démontrent que le triplet symétrique est très performant pour les deux programmes. En effet, on dénombre seulement une (1) erreur dans le calcul de la constante π ayant été fatale pour son fonctionnement. Cette erreur est causée par la congestion du bus de communication par le processeur ayant subit l'injection de pannes. Ainsi, les processeurs ne peuvent échanger leurs messages normalement sans qu'il y ait désynchronisation complète du système, ce qui est catastrophique puisque des temporisateurs "timeout" sont utilisés par les mécanismes de tolérance. Il est important de noter que ce type d'erreur n'est pas spécifique au triplet symétrique. En effet, cette erreur peut survenir dans n'importe quel mode d'opération, puisqu'aucun mécanisme permettant de contrer ce genre d'erreur dans le système n'a été implanté.

Le triplet asymétrique, comparativement au triplet symétrique, ne donne pas d'aussi bons résultats pour les deux programmes. En effet, les résultats obtenus pour le calcul de la constante π sont très satisfaisants puisqu'il améliore grandement la fiabilité du système. Cependant, les résultats obtenus pour le programme des reines démontrent que cette fiabilité n'est pas constante pour tous les programmes. En effet, le nombre d'erreurs ayant causé un problème de fonctionnement est sensiblement le même que celui recueilli pour le mode normal. Ainsi, on pourrait conclure que l'implantation de ce mode d'opération n'est pas viable pour un système puisqu'on ne peut améliorer la fiabilité de façon significative. On pourrait toujours être tenté d'utiliser un tel mécanisme pour la détection de pannes puisqu'aucune erreur dans les résultats n'a été détectée. Cependant, des résultats antérieurs qui ne figurent pas au tableau ont démontré que le résultat final d'un programme en mode triplet asymétrique n'est pas toujours valide.

On ne peut parler du triplet asymétrique sans parler du triplet mobile qui se veut une réplique de ce dernier avec quelques modifications. En effet, si l'on compare les résultats du triplet mobile avec ceux du triplet asymétrique, on s'aperçoit que les performances du triplet mobile sont supérieures à

tous points de vue. On peut même dire qu'ils se rapprochent davantage à ceux du triplet symétrique. Cette différence entre les résultats permet de mettre en lumière le fait que le triplet asymétrique et le triplet mobile possèdent un point critique de défaillance au niveau du processeur maître. Afin de déterminer l'effet que pouvait avoir ce point critique de défaillance sur les résultats obtenus, les injections de pannes ont été effectuées dans le processeur maître pour le triplet asymétrique et dans le processeur adjoint pour le triplet mobile. Les résultats présentés dans les tableaux 4.3a et 4.3b démontre bien que lorsque le processeur maître est seul responsable de l'envoi d'un message à un autre triplet, ceci a pour effet d'introduire un point critique de défaillance.

Finalement, le duo symétrique a donné les résultats escomptés. En effet, le duo symétrique possède uniquement un mécanisme de détection des pannes. Ainsi, lorsqu'une erreur est détectée, elle occasionne un arrêt systématique du système. Le but de ce mode d'opération est de s'assurer de la fiabilité des résultats obtenus. En effet, lorsqu'une application n'exécute pas de fonction vitale pour un système et que seul le résultat compte, le duo symétrique devient un mode d'opération économique et viable. Cependant, lorsqu'on regarde attentivement les résultats, on ne peut négliger l'augmentation du nombre d'erreurs dans le calcul de la constante π comparativement au mode normal. La cause de cette augmentation sera discutée à la section 4.7.

4.3. Le lieu d'injection des pannes.

L'un des aspects importants de cette étude fût de déterminer l'effet des injections de pannes de type inversion de bits sur les différentes sections du code généré lors de la compilation d'un programme. Ainsi les tableaux 4.4a et 4.4b montrent la répartition (en pourcentage) des différents types d'erreurs observées dans les sections de code des programmes et ce pour les cinq modes d'opération.

Lors de la compilation, le compilateur divise la mémoire qui lui est allouée en six sections de code. Chacune des sections contient un ou plusieurs types d'information bien précis qui sont nécessaires à l'exécution du programme. Afin de mieux caractériser l'effet des injections de pannes sur les différentes sections de code, on a forcé le compilateur à disposer ces dernières dans la mémoire qui lui est allouée selon un ordre bien précis.

La plage mémoire 0x0002 :0x01Bf constitue la section de code BASE, cette section contient les variables globales du programme ainsi que certaines variables de fonctions déclarées "statiques". La plage mémoire 0x8000:0xC7FF constitue la section de code RAM16. Cette section contient les données des fonctions d'un programme ainsi que sa pile (stack). La plage mémoire 0xC800:0xDFFF constitue la section de code RAM8. Cette section contient les adresses des pointeurs de fonction. La plage mémoire 0xE000:0xF7FF constitue la section de code ROM16. Cette section contient les instructions des différentes fonctions d'un programme. La plage mémoire 0xF800:0xFFBE constitue la section de code ROM8. Cette section contient des données de format texte ainsi que différentes informations associées aux librairies utilisées par un programme. Finalement, la plage mémoire 0xFFBE:0xFFFF constitue la section de code VECTEUR. Cette section contient les adresses des fonctions appelées lors de l'interruption du microprocesseur.

Tableau 4.4a : Répartition du pourcentage d'erreurs détectées dans les sections de code du calcul de la constante π.

Nom du programme	Calcul de la constante π				
Mode d'opération	Normal	Triplet Symétrique	Triplet Asymétrique	Triplet Mobile	Duo Symétrique
% dans la zone BASE	0.00	0.00	0.00	0.00	0.00
% dans la zone RAM16	38.24	11.57	24.72	8.70	35.02
% dans la zone RAM8	0.00	0.83	1.12	1.09	0.00
% dans la zone ROM16	61.76	84.30	73.03	88.04	64.59
% dans la zone ROM8	0.00	3.31	1.12	2.17	0.39
% dans la zone VECTEUR	0.00	0.00	0.00	0.00	0.00
% dans la zone inutilisée	0.00	0.00	0.00	0.00	0.00

Tableau 4.4b: Répartition du pourcentage d'erreurs détectées dans les sections de code du problème des reines.

Nom du programme	Problème des reines				
Mode d'opération	Normal	Triplet Symétrique	Triplet Asymétrique	Triplet Mobile	Duo Symétrique
% dans la zone BASE	0.00	0.00	0.00	0.00	0.00
% dans la zone RAM16	4.35	0.00	0.00	0.00	0.00
% dans la zone RAM8	0.00	0.93	0.74	0.82	0.00
% dans la zone ROM16	95.65	96.26	92.65	90.98	97.50
% dans la zone ROM8	0.00	2.80	0.74	8.20	2.50
% dans la zone VECTEUR	0.00	0.00	0.00	0.00	0.00
% dans la zone inutilisée	0.00	0.00	0.00	0.00	0.00

La première constatation qu'on peut effectuer est l'absence d'erreur dans les espaces mémoire BASE et VECTEUR. La raison de cette absence provient du fait qu'il n'y a eu aucune injection

effectuée dans ces espaces mémoire. En effet, l'étude effectuée n'étant que préliminaire, on s'est donc concentré sur les espaces mémoire de grande dimension qui comportent la majeure partie du programme. Cependant, l'injection de pannes sur ces espaces mémoire fera l'objet d'études ultérieures. Ainsi, on pourra déterminer l'influence de ces dernières sur l'ensemble des résultats.

Parmi les résultats du tableau 4.4, figurent les erreurs qui ont été injectées dans des adresses mémoire inutilisées qui ne contiennent aucune information appartenant aux programmes. A priori, ce résultat semble trivial et ne suscite aucun intérêt. Cependant, il permet d'évaluer l'efficacité de l'outil d'analyse utilisé ainsi qu'a démontrer que les erreurs injectées dans des adresses mémoire inutilisées sont sans conséquence pour un programme. De plus, ceci permet de démontrer que le système d'exploitation se comporte normalement et n'introduit aucune erreur lors de l'exécution d'un programme.

Les informations recueillies sur chacune des sections permettent de déterminer leurs sensibilités aux pannes pour un programme donné. Ainsi, on constate que pour les deux programmes étudiés, la section de code ROM 16 contient le plus grand nombre d'erreurs ayant affecté l'exécution d'un programme. En effet, plus de 60% des erreurs recueillies parmi toutes les expérimentations effectuées proviennent de cette section. Puisque cette dernière contient les instructions nécessaires à l'exécution d'un programme, il est donc normal de s'attendre à un tel résultat. La présence d'une seule erreur dans une adresse mémoire de cette section peut affecter le déroulement normal du programme.

Les autres espaces mémoire n'ont pas la même sensibilité pour les deux programmes étudiés. L'effet d'une panne sur ces dernières est fonction de la nature du programme exécuté ainsi que l'utilisation des différentes sections. En effet, on remarque que, dans la section de code RAM 16, on observe jusqu'à 38% des erreurs pour le calcul de la constante π. Cependant, lorsqu'on regarde les

résultats pour le problème des reines, on s'aperçoit qu'aucune panne injectée dans cette section à l'exception du mode d'exécution normal n'a été détectée. La raison principale qui explique de tels résultats pour le programme des reines provient du fait que ce dernier utilise une grande quantité d'espaces mémoire pour sauvegarder les différentes possibilités de disposition des reines sur son échiquier. L'utilisation de la mémoire croît de façon exponentielle avec les dimensions de l'échiquier pour lequel on désire connaître les possibilités. Cependant, il est important de mentionner que bien que le problème des reines utilise une grande quantité de mémoire, seulement un portion de cette dernière contient de l'information pertinente à la solution du problème. Une étude plus approfondie pourrait démontrer l'effet réel sur l'augmentation de l'utilisation de la mémoire pour le problème des reines. Contrairement au problème des reines, le calcul de la constante π utilise très peu d'espaces mémoire pour conserver ses informations. De plus, la quantité d'espaces mémoire utilisée pour le calcul de la constante π est indépendante de la précision qu'on désire obtenir. Cependant toute panne qui affecterait cette espace mémoire serait fatale au processeur puisqu'il s'agit d'un processus itératif et que le résultat de chaque itération fait parti du résultat final.

L'injection de pannes dans les sections de code RAM8 et ROM8 a une influence minime sur l'exécution d'un programme comparativement aux deux sections précédentes. Ce faible taux n'est pas lié à l'information contenue dans ces sections mais plutôt à l'utilisation qui en est faite. La section de code RAM8 contient les adresses des fonctions qui sont capitales pour l'exécution d'un programme. Cependant, elles sont plus ou moins utilisées. En effet, certaines de ces fonctions sont utilisées pour la récupération des erreurs qui surviennent dans le système. Étant donné qu'une seule erreur est injectée dans le système lors de l'exécution, cette dernière ne sera pas détectée puisque la fonction affectée par cette panne ne sera pas exécutée. La section de code ROM8 quant à elle, contient des données qui ne sont pas ou très peu significatives pour le programme. En effet, la principale information contenue dans cette section sert pour fins de déverminage ("débugging").

Certaines des données qu'elle contient sont accessibles lors de l'analyse, ce qui permet de déterminer avec une plus grande exactitude le nombre réel de pannes qui ont affectées le système.

4.4. Évolution des erreurs dans le temps

L'un des objectifs de cette étude consistait à déterminer l'effet que pouvait avoir le temps d'injection d'une panne sur le comportement du système. Étant donné le grand nombre de graphiques que cette section nécessite, ces derniers ont été placés à l'annexe A. Les résultats présentés dans ces graphiques tiennent compte de toutes les erreurs ayant affecté un programme lors de son exécution. Les termes utilisés pour classifier les erreurs sont les mêmes que ceux utilisés dans la section 4.2. Chaque graphique comporte un titre qui donne le nom du programme exécuté, le mode d'opération ainsi que le type de pannes qu'on y a injectées. L'ordonnée contient la somme des différentes erreurs recueillies pour un temps d'injection donné. Finalement, l'abscisse indique les différents temps auxquels on a injecté les pannes.

Ces graphiques ont permis de mettre en lumière différents comportements. Les observations recueillies au cours des expérimentations démontrent que, contrairement à ce qu'on pourrait croire, le nombre d'erreurs recueillies ne décroît pas toujours en fonction du temps d'injection. En effet, en regardant attentivement les résultats obtenus, on peut classifier les différentes courbes obtenues en trois catégories. La première catégorie regroupe les courbes décroissantes. Ces courbes peuvent décroître de façon exponentielle, décroître de façon linéaire ou encore décroître par palier. Cependant, il est important de spécifier que l'interprétation des courbes est faite selon leur tendance. Ainsi, le graphique du programme des reines dans le mode d'opération triplet mobile montre une courbe à décroissance linéaire, alors que le graphique du calcul de la constante π dans le mode triplet asymétrique montre une courbe à décroissance exponentielle. La deuxième catégorie de courbe tend à décrire une cloche. Le graphique du programme des reines en mode triplet asymétrique fait partie

de cette catégorie. La dernière catégorie ne décrit pas une courbe mais plutôt un plateau comportant de légères fluctuations. Le graphique du calcul de la constante π en mode normal fait partie de cette catégorie. Cette dernière catégorie suscite beaucoup d'intérêt dans l'étude du comportement du système en fonction du temps d'injection.

Suite aux résultats obtenus, nous nous sommes penchés sur les causes pouvant provoquer de si grandes différences entre les deux programmes et leurs modes d'opération. L'une des premières observations effectuées démontre que le lieu d'une injection est un facteur important dans la sensibilité du programme en fonction du temps. Par exemple, si l'on injecte une panne dans une fonction qui est exécutée à la fin du programme, la probabilité que l'on retrouve une trace de cette panne pour les différents temps d'injection est très élevée, comparativement à une fonction exécutée au début du programme. Seule la nature des données touchées lors de l'injection de pannes déterminera si oui ou non l'erreur sera observée pour un temps d'injection donné.

Cette dernière affirmation nous amène à la deuxième observation effectuée lors de l'analyse des résultats. En effet, le moment où on injecte une panne influence les résultats obtenus. Si l'on injecte une panne dans une des variables utilisées par une fonction, cette dernière ne sera probablement pas affectée par l'injection de panne si celle-ci est effectuée avant qu'on l'ait exécutée. Cette affirmation est vraie uniquement si la fonction initialise ses données lors de son exécution. Ainsi, toute panne injectée après l'initialisation d'une variable et avant qu'elle soit accédée lors de l'exécution de cette fonction a une chance d'être observée. Si l'injection est effectuée dans une instruction de la fonction, la probabilité que la panne soit observée lorsque l'injection est effectuée avant l'exécution de cette instruction est très élevée.

La troisième observation effectuée tient compte de la fréquence d'utilisation d'une même fonction. En effet, si une fonction est utilisée fréquemment, comme c'est le cas de certaines dans le

programme des reines, la probabilité que le système soit affecté par une injection de pannes augmente en fonction de la durée d'exécution de la fonction et de sa fréquence d'utilisation. La probabilité d'observation devient quasi indépendante du moment d'injection de la panne.

Finalement, la dernière observation porte sur l'effet des injections de pannes sur une variable globale du système. En effet, l'injection de pannes sur des variables globales augmente les chances d'affecter le système indépendamment du temps d'injection. Cependant, l'utilisation qu'on fait de la variable ainsi que le nombre de fonctions qui y accèdent sont des facteurs importants dans les résultats qu'on observe. En effet, une variable globale dont on modifie fréquemment le contenu a moins de chances d'affecter le déroulement d'un programme qu'une variable qui est initialisée au début de l'exécution.

4.5. Effet de la granularité sur les résultats obtenus.

Lors du développement du système d'exploitation, une série de tests a du être effectuée avant d'atteindre le rendement désiré pour chacun des modes d'opérations. Au cours de cette série de tests, on a remarqué que le pourcentage d'erreurs recueillies pouvait varier de façon considérable d'une expérimentation à l'autre pour des conditions similaires, notamment lors de l'analyse des résultats du calcul de la constante π. En regardant attentivement le nombre total d'erreurs dans le tableau 4.2, on s'aperçoit qu'il ne semble pas y avoir de corrélation entre la taille du programme exécuté et le nombre d'erreurs observé.

Afin de mieux comprendre ce phénomène, une étude approfondie des différents fichiers générés lors de la compilation du programme a été réalisée. Les informations recueillies lors de cette étude ont permis d'émettre une hypothèse sur la cause possible de tels résultats. En effet, les premières observations effectuées lors des premières séries de tests ont démontré que, lors de modifications

effectuées au système d'exploitation, le compilateur peut effectuer une restructuration complète du code généré. Cette restructuration peut consister à déplacer les instructions associées à une fonction ou encore déplacer les données qui lui sont associées. Lors de la dernière analyse des résultats, on a constaté que l'organisation du code pour les différents modes d'opérations étaient très différentes d'un mode d'opération à l'autre. Suite à ces observations, on s'est interrogé sur l'influence que pouvait avoir une réorganisation du code sur la qualité des résultats obtenus. Ainsi, on a voulu déterminer quels étaient les facteurs ayant conduit à une si grande variabilité dans les résultats pour le calcul de la constante π. Afin de déterminer avec exactitude les causes réelles de ces résultats, on a donc procédé à une analyse approfondie du contenu des espaces mémoire touchés par les injections de pannes. Ainsi, on a observé que les sections de codes étaient plus "dense" pour le mode normal et le mode triplet asymétrique que pour les autres modes d'opération. Le terme "dense" signifie que les instructions sont regroupées dans une même région. Afin, de vérifier si ce facteur pouvait jouer un rôle dans les résultats obtenus, on a procédé à une étude similaire sur les différents modes d'opérations du programmes des reines. On a remarqué que le mode normal et le mode duo symétrique subissaient eux aussi une restructuration importante au niveau du code. Cependant les résultats recueillis pour le programme des reines présentent une corrélation entre la taille du code et le nombre d'erreurs recueillies ce qui a pour effet de remettre en cause la validité de l'hypothèse de la restructuration du code comme cause principale des différences entre les résultats obtenus.

D'autres observations effectuées au cours de cette analyse ont démontré que certaines fonctions vitales du calcul de la constante π étaient plus souvent la cible d'injection de pannes pour le mode duo symétrique et le mode normal que pour les autres modes d'opérations. En effet, les erreurs recueillies pour ces deux modes d'opération se trouvent dans des sections de code qui sont sensibles tout au long de l'exécution du programme. Ceci expliquerait l'allure des graphiques de cette série de test. Cette constatation évoque l'hypothèse que la granularité (échantillonnage) des expérimentations

effectuées n'est pas suffisamment petite pour évaluer le comportement réel d'un système multiprocesseur lorsqu'il est soumis à des injections de pannes. Un autre argument vient renforcer l'hypothèse d'une trop grande granularité, soit le nombre d'adresses mémoire ayant subies des injections de pannes. En effet, seulement 208 adresses mémoire ont subi des injections de pannes lors d'une série expérimentale. De plus, chaque adresse a été soumis à 13 injections de pannes à différent moment dans le temps. Ce qui donne un total de 2704 injections par programme par mode d'opération. Les pourcentages observés varient de 2.5% à 15.2% avec une moyenne de 5.9%. Le nombre d'adresses ayant subi des injections de pannes est faible lorsqu'on le compare à la taille d'un des programmes compilés en mode normal. Afin de déterminer le comportement réel du système lorsqu'il est soumis à une injection de pannes, il serait souhaitable d'effectuer de nouvelles séries expérimentales comportant un plus grand nombre d'injections dans le système. Ainsi, on pourra déterminer la granularité minimale (c'est-à-dire, le rapport du nombre d'injections divisé par la taille du programme) à employer pour caractériser la robustesse d'un programme et des mécanismes de tolérance inclus.

CHAPITRE 5

CONCLUSION

5. CONCLUSION

Ce chapitre résume les résultats importants du projet réalisé.

5.1. Le système d'exploitation.

À la lumière des résultats obtenus, on peut dire que le système d'exploitation répond bien aux spécifications établies dans le premier chapitre. En effet, on constate que quatre des cinq modes d'opération possèdent des mécanismes de détection de panne et trois d'entre eux possèdent des mécanismes de récupération de pannes. On peut dire également que, dans sa version actuelle, il peut être porté sur n'importe quel système existant, moyennant de très légères modifications. Il est raisonnable de penser qu'on puisse utiliser la même approche pour des systèmes distribués non-homogènes. Ceci ne peut se faire sans introduire des mécanismes qui gèrent l'utilisation des stations au sein du système. Finalement, on peut dire que le système d'exploitation "FATMOS" peut être implanté rapidement et à faible coût.

5.1.1. Les modes d'opération.

Des quatre modes d'opération comportant des mécanismes de détection et de récupération de pannes, c'est le triplet symétrique qui s'est avéré être le plus robuste. Ce dernier atteint un rendement de tolérance aux défaillances approchant les 100% pour les injections de pannes simples. Il est raisonnable de croire qu'une fois combiné à d'autres mécanismes de tolérance aux défaillances, l'objectif de 100% pour des injections de pannes simples pourrait être obtenu. De plus, l'ajout de tels mécanismes aurait pour effet d'augmenter le rendement si le système était soumis à des injections de pannes multiples.

Le mode duo symétrique a lui aussi donné de bonnes performances pour la détection des pannes. Puisqu'il n'est pas muni de mécanisme de récupération de pannes, il ne peut donc pas être comparé

au triplet symétrique. Cependant, ce mode demeure très efficace pour diagnostiquer toute erreur pouvant affecter un système. En effet, les résultats ont démontré qu'aucune erreur n'a pu affecter le système sans être détectée.

Le mode triplet asymétrique quant à lui, offre des performances médiocres. En effet, ce dernier n'offre pas de gain significatif qui justifie son utilisation dans un système. De plus, il utilise un grand nombre de processeurs sans pour autant augmenter de façon significative la fiabilité d'un système. Cette lacune du triplet asymétrique était prévisible dès le début du projet. Cependant, on a cru bon d'utiliser ce dernier comme moyen de comparaison pour le triplet symétrique.

Le triplet mobile n'est pas tellement plus efficace que le triplet asymétrique. Cette efficacité peut être augmentée si l'on évite d'affecter la tâche de processeur maître au processeur soupçonné d'être défectueux. L'utilisation d'autres mécanismes de détection et de récupération de pannes aurait pu rendre ce mode d'opération très efficace. En effet, il aurait été plus efficace d'utiliser le principe du triplet symétrique comme mécanisme de détection et de récupération des pannes ce qui aurait eu pour effet d'éliminer le point critique de défaillance au niveau du processeur maître. Cependant, ceci aurait augmenté la complexité dans la routine de gestion des échanges de messages.

5.1.2. L'approche logicielle.

Il est difficile d'évaluer avec précision l'impact réel de l'utilisation d'une approche logicielle pour la tolérance aux défaillances. En effet, puisque le développement ainsi que les tests ont été effectués sur un simulateur plutôt que sur un prototype et qu'aucun système comparable n'utilise une approche matérielle, il nous a donc été impossible de déterminer quels sont les effets réels encourus par l'utilisation intensive de messages pour la détection d'erreurs. Cependant, il est raisonnable de croire que cette utilisation intensive des communications n'est pas un facteur déterminant. En effet,

comme on l'a mentionné dans le chapitre 1, il est impensable de créer un système tolérant aux défaillances sans utiliser une redondance des éléments qui le composent, ce qui implique qu'il faut, tôt ou tard, effectuer une comparaison des données du système.

On peut affirmer que l'utilisation d'une approche logicielle pour les injections de pannes s'avère être un instrument de grande précision lorsque vient le temps de recréer les conditions qui ont mené à une panne. Ceci simplifie le processus de développement, puisque l'on peut répéter autant de fois que nécessaire une panne et ainsi diagnostiquer le problème. Cependant, il faut bien comprendre que cette technique a des lacunes. En effet, elle ne permet pas d'évaluer certaines caractéristiques qui ne peuvent être déterminées qu'en utilisant une approche matérielle.

5.2. Comportement du système.

Le chapitre 4 a permis d'approfondir nos connaissances sur le comportement d'un système multiprocesseur lorsqu'il est soumis à des injections de pannes. Dans ce chapitre, on a étudié quatre sujets qui ne sont généralement pas abordées par d'autres études.

Le premier sujet touchait la sensibilité d'un système multiprocesseur à des injections de pannes. Ainsi, on a pu déterminer que les chances qu'un programme soit affecté lorsqu'une panne est injectée sont faibles. Dans le pire des scénarios observés, moins de 16% des pannes injectées ont affecté le système de différentes façons. Il faut également ajouter que ce nombre fut obtenu suite à 2704 injections dans le système. Si on considère les chances qu'un système subisse réellement une perturbation, la probabilité que ce dernier soit affecté est donc faible.

Le second sujet touchait l'efficacité des mécanismes de tolérance aux défaillances pour accroître la fiabilité d'un système. Les résultats ont démontré que l'utilisation de mécanismes tels que le triplet symétrique et le duo symétrique peut contribuer à augmenter de façon significative la fiabilité du

système et ainsi assurer une qualité des résultats. De plus, le triplet symétrique a démontré que 100% des résultats "obtenus" étaient fiable pour des injections de pannes simples. Cependant, on ne peut être sûr à 100% "d'obtenir" un résultat de la part du système. En effet, une congestion du bus de données peut conduire à une erreur au niveau du système.

Le troisième sujet portait sur l'effet du lieu d'une injection de panne. L'étude réalisée sur ce sujet était des plus intéressante, puisqu'elle a permis de démontrer que l'effet d'une panne injectée dans le système dépend en grande partie des caractéristiques de la région touchée. En effet, on a remarqué que la zone mémoire contenant les instructions d'un programme et du système d'exploitation s'avère être très sensible comparativement aux autres zones.

Le dernier sujet traitait de la sensibilité d'un système à des pannes injectées à différents moment de l'exécution d'un programme. Dans cette section, on a déterminé que la sensibilité d'un programme varie en fonction de ses caractéristiques ainsi que du mécanisme de tolérance utilisé. De plus, on a pu établir des hypothèses qui permettaient d'expliquer pourquoi les programmes étaient en général peu sensibles. L'une des explications présentées était basée sur la nature des informations contenues dans l'adresse mémoire affectée par la panne. En effet, le programme devient plus sensible s'il s'agit d'une fonction qui est exécutée souvent ou en fin de programme lorsqu'il s'agit d'une fonction utilisée au début du programme. Une autre hypothèse soulevée portait sur l'utilisation des variables globales. Ces dernières sont plus ou moins sensibles si elles sont réinitialisées fréquemment et que l'information contenue lors de l'initialisation n'est pas utilisée pour le calcul de leurs futurs valeurs.

Liste bibliographique.

1. [CLA95] J. A. Clard, and D. K. Pradhan, « Fault Injection Amethod For Validation Computer-System Dependability, » IEEE Computer, June 1995, pp. 47-56.

2. [KAR94] J. Karlsson, P. Liden, P. Dahlgren, and R. Johansson, « Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms, » IEEE Micro, Feb. 1994, pp. 8-29.

3. [POW94] D. Powell, « Distributed Fault Tolerance : Lessons from Delta-4, » IEEE Micro, Feb. 1994, pp. 36-47.

4. [ARE94] N. Arel, D. Audet, and Y. Savaria, « Functional Simulator and Executable Models Version 2, » Contract Report (W2207-0-AF14/01-SS), Apr. 1994, 44 pages.

5. [SOS94] J. Sosnowski, « Transient Fault Tolerance in Digital Systems, » IEEE Micro, Feb 1994, pp. 24-35.

6. [BOW93] N. S. Bowen, and D. K. Pradhan, « Processor- and Memory-Based Checkpoint and Rollback Recovery, » IEEE Computer, Feb. 1993, pp. 22-30.

7. [SAV93] Y. Savaria, D. Audet, and N. Arel, « State-of-the-art in Large Area Integration and Scalability to ULSI, » Contract Report (W2207-0-AF14/01-SS), Dec. 1993, 93 pages.

8. [CLA93] Clark and D.K. Pradhan, « React : A Synthesis and Evaluation Tool for Fault-Tolerant Multiprocessor Architectures, » Proc. 1993 Annual Reliability and Maintainability Symp., IEEE Press, Piscataway, N.J., 1993, pp. 428-435.

9. [KAN92] Kanawati, N.A. Kanawati, and J.A. Abraham, « Ferrari : A Tool for the Validation of System Dependability Properties, » Proc 22nd Int'l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, Calif., Order No. 2876, 1992, pp. 336-344.

10. [CHO92] Choi and R.K. Iyer, « Focus : An Experimental Environement for Fault Sensitivity Analysis, » IEEE Trans. Computers, Vol. 41. No. 12 Déc 1992. pp.1,515-1,526.

11. [CLA92] Clark and D.K. Pradhan, « Reliability Analysis of Unidirectional Voting TMR Systems Through Simulated Fault Injection, » Proc. 1992 Vorkshop Fault-Tolerant Parallel and Distributed Systems, IEEE CS Press, Los Alamitos, Calif., Order No. 2871, 1992, pp. 72-81.

12. [CZE92] E.W. Cseck, and D. P. Siewiorek, « Observations on the Effects of Fault Manifestation as a Function of Workload, » IEEE Transactions on computers, Vol. 41, NO. 5. May 1992, pp. 559-566.

13. [KAN92] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, « FERRARI : A Tool for The alidation of System Dependability Properties, » Proc. 22nd Int'l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, Calif, 1992, pp. 336-344.

14. [SAV91] Y. Savaria, D. Audet and N. Arel, « Methods of On-line Error Detection and Correction in ULSI Architectures, » Contract Report (W2207-0-AF14/01-SS), jun. 1991, 70 pages

15. [ARE91] N. Arel, D. Audet, Y. Savaria, « Report on a Functional Simulator for the Proposed ULSI Architechture, » Contract Report (W2207-0-AF14/01-SS), Sept. 1991, 44 pages.

16. [KAR91] J.Karlsson et al.' « Two Fault-Injection Techniques for Test of Fault-Handling Mechanisms, » Proc. Int'l Test Conf. ,IEEE CS Press, Los Alamitos, Calif., Order No. 2156, 1991,pp.140-149.

17. [GOS91] K.K Goswami and R.K. Iyer, « A Simulation-Base Study of a Triple-Modular Redundant System Using Depend, » Proc. Fifth Int'l Conf. Fault-Tolerant Computing Systems' IEEE Press, Piscataway, N.J., 1991, pp. 300-311.

18. [ARL90] Arlat et al., « Fault Injection for Dependability Validation : A Methodology and Some Applications, » IEEE Trans. Software Engineering, Vol. 16, No. 2, Feb. 1990, pp. 166-182.

19. [INT90] iPSC/2 and iPSC/860 User's Guide, Intel Corporation, June 1990.

20. [CZE90] Czeck and D.P. Siewiorek, « Effects of Transient GateLevel Faults on Program Behavior, » Proc. 20th Int'l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, Calif.' Order No. 2051, 1990, pp. 236-243.

21. [MAD90] H Madeira, G. Quadros, and J. G. Silva « Experimantal evaluation of a set of simple error detection mechanisms, » North-Holland Microprocessing and Microprogramming 30, 1990, pp. 513-520.

22. [BOW89] R.Chillarege and N.S. Bowen, « Undestanding Large-System Failures : A Fault-Injection Experiment, » Proc. 19th Int'l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, Calif.' Order No. 1959, 1989, pp. 356-363.

23. [CHI89] R.Chillarege and R.K. Iyer, « An Experimental Study of Memory Fault Latency, » IEEE Trans. Computers, Vol. 38, No. 6, June 1989, pp. 869-874.

24. [GER89] J. Ph. Gerardin, « The 'DEF.Injector' Test Instrument, Assistance in the Design of Reliable and Safe Systems, » North-Holland Computers in Industry 11, 1989. pp 311-319.

25. [GUN89] U. Gunneflo, J. Karlsson, and J. Torin, « Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation, » Fault Tolerant Computer Symposium, Chicago, 21-23, June 1989, pp. 340-347.

26. [ARL89] J. Arlat, Y. Crouzet, and J-C Laprie, « FAULT INJECTION FOR DEPENDABILITY VALIDATION OF FAULT-TOLERANT COMPUTING SYSTEMS, » Fault Tolerant Computer Symposium, Chicago, 21-23, June 1989, pp. 348-355.

27. [SEG88] Segal et al., « Fiat : Fault-Injection-Based Automated Testing Environnement, » Proc 18th Int'l Symp. Fault-Tolerant Compution, IEEE CS Press, Los Alamitos, Calif., Order No. 867, 1988, pp. 102-107

28. [CHI87] R.Chillarege and R.K. Iyer, « Measurement-Based Analysis of Error Latency, » IEEE Trans. Computers, Vol. C-36, No. 5, May 1987, pp. 529-537.

29. [AND81] T. Anderson and P.A. Lee, « Fault Tolerance, » Prentice-Hall, Englewood Cliffs, N.J. 1981

ANNEXE A

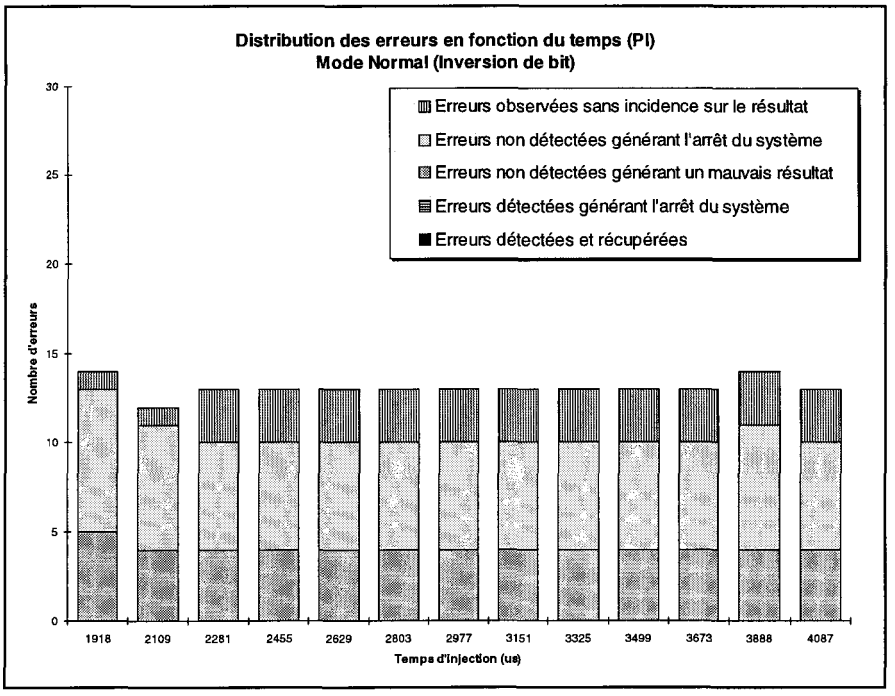

Figure A.1 : Distribution des erreurs en fonction du temps (Constante π) Mode Normal (Inversion de bit)

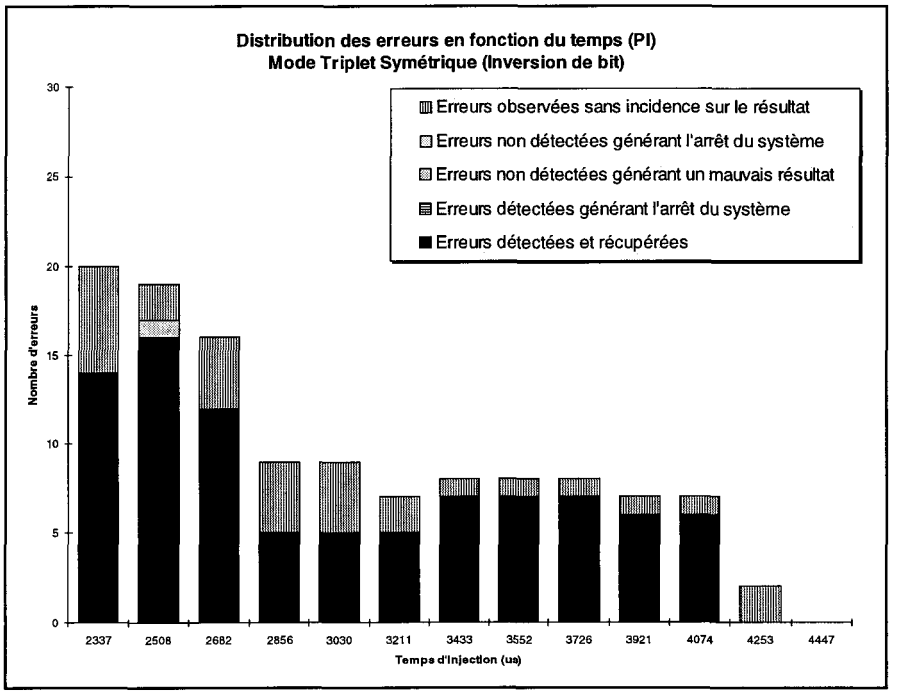

Figure A.2 : Distribution des erreurs en fonction du temps (Constante π) Mode Triplet Symétrique (Inversion de bit)

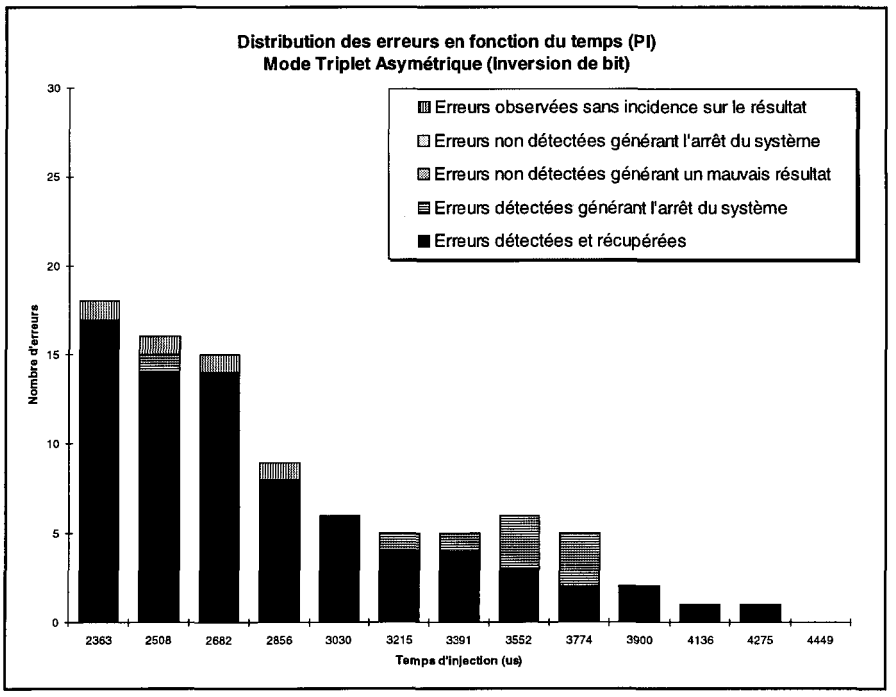

Figure A.3 : Distribution des erreurs en fonction du temps (Constante π) Mode Triplet Asymétrique (Inversion de bit)

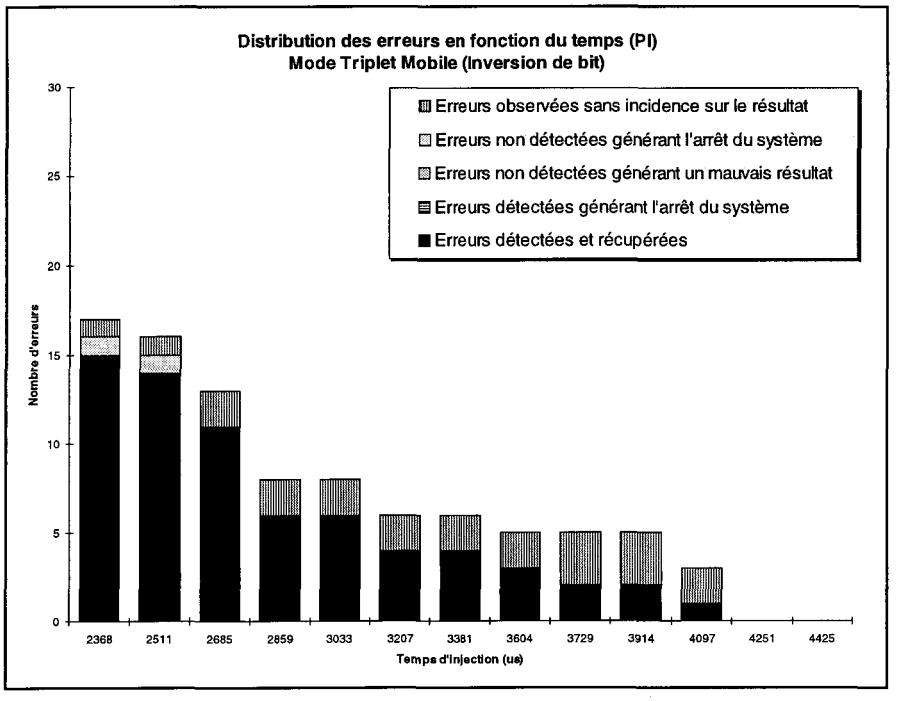

Figure A.4 : Distribution des erreurs en fonction du temps (Constante π) Mode Triplet Mobile (Inversion de bit)

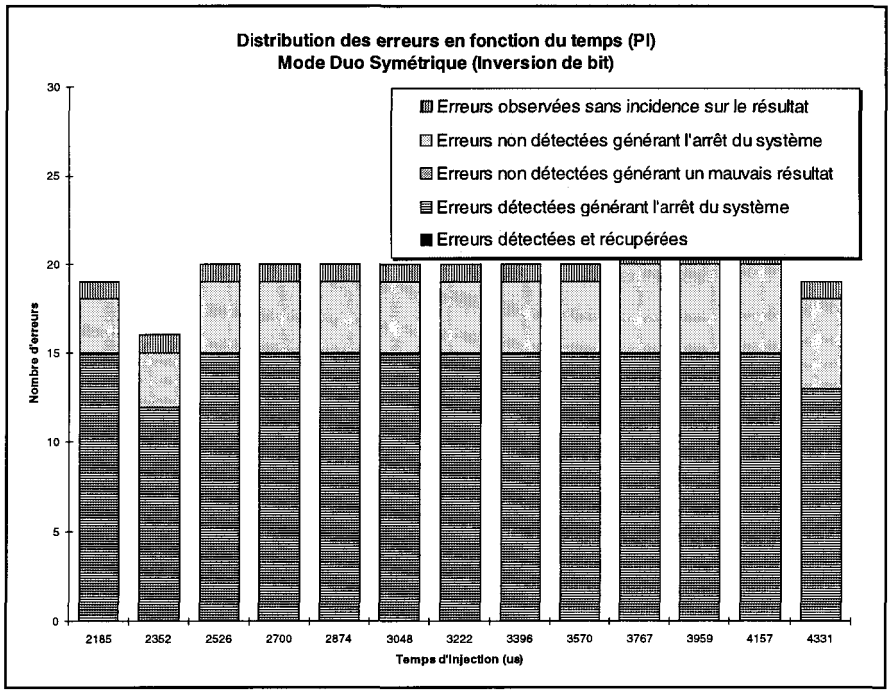

Figure A.5 : Distribution des erreurs en fonction du temps (Constante π) Mode Duo Symétrique (Inversion de bit)

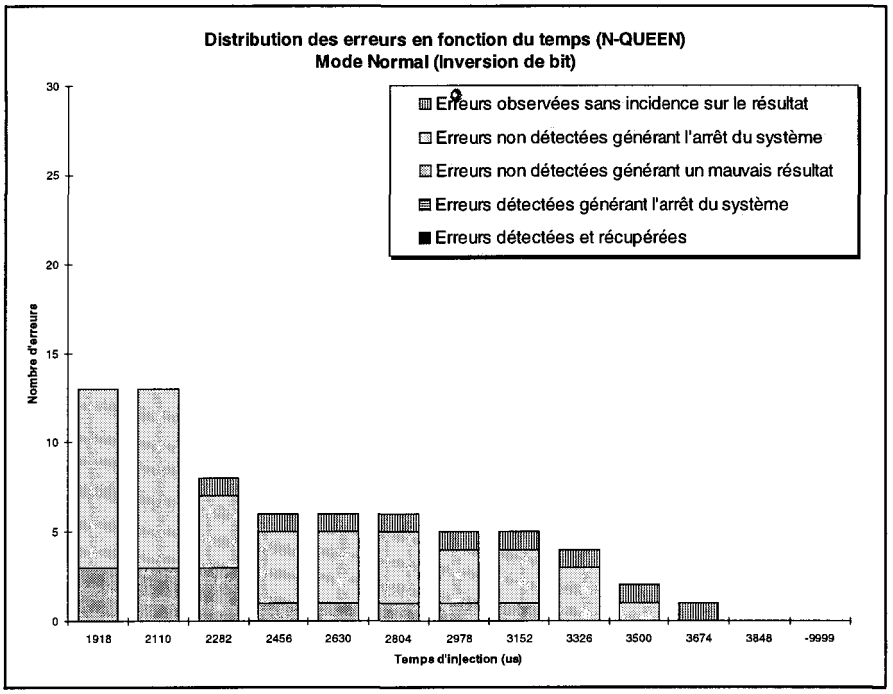

Figure A.6 : Distribution des erreurs en fonction du temps (Problème des reines) Mode Normal (Inversion de bit)

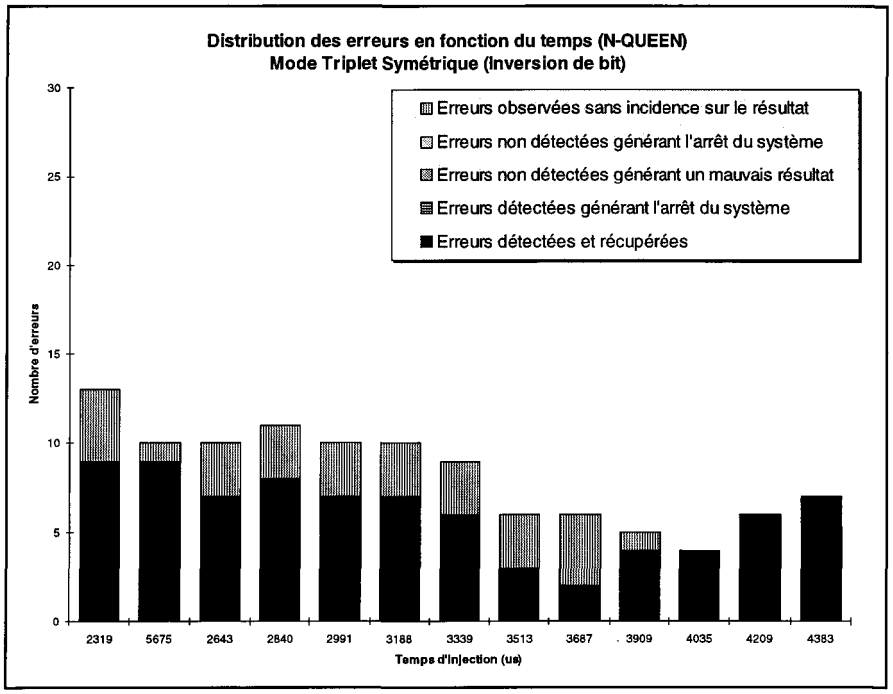

Figure A.7 : Distribution des erreurs en fonction du temps (Problème des reines) Mode Triplet Symétrique (Inversion de bit)

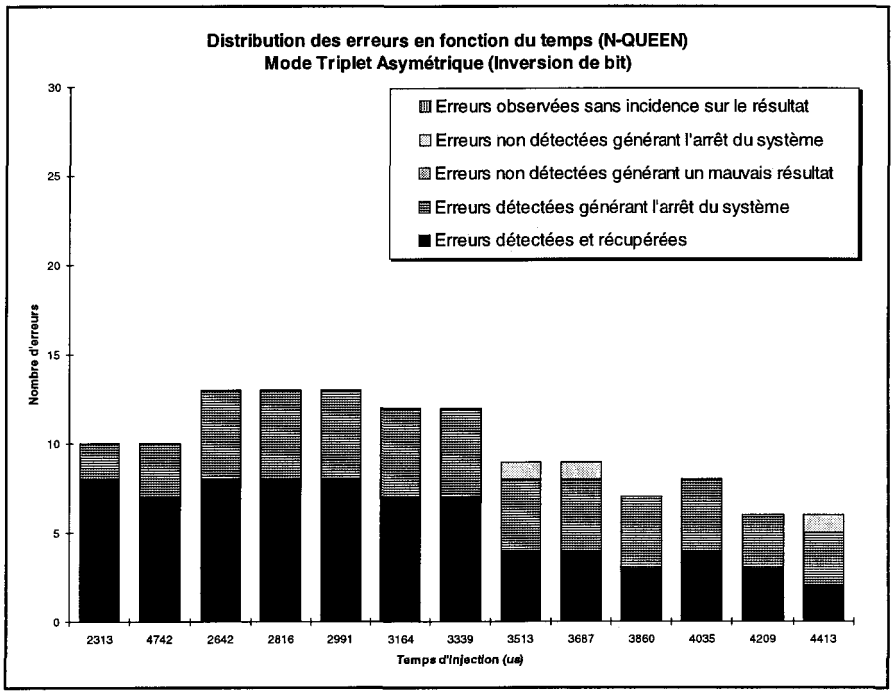

Figure A.8 : Distribution des erreurs en fonction du temps (Problème des reines) Triplet Asymétrique (Inversion de bit)

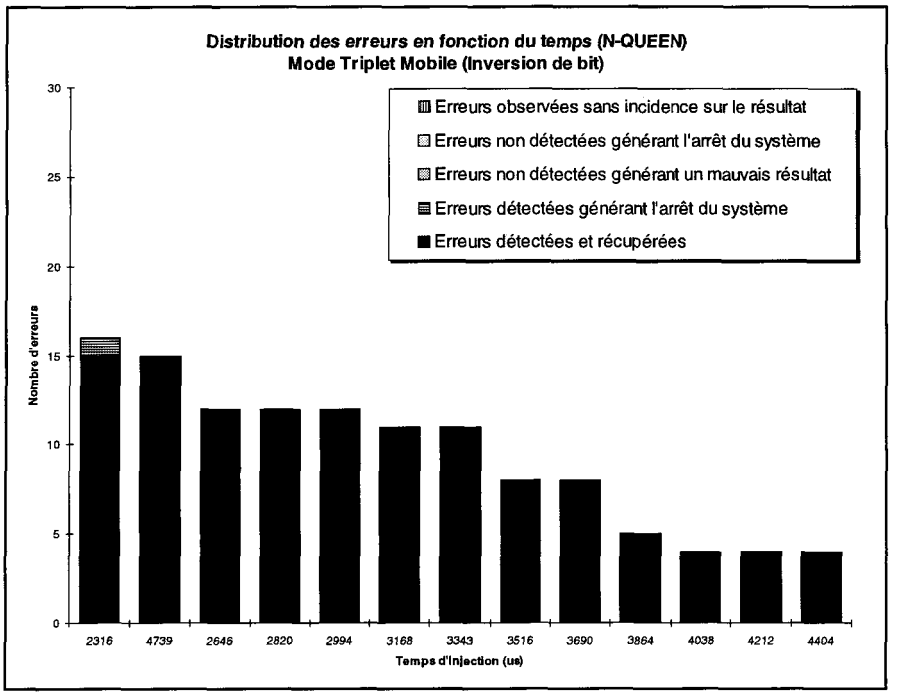

Figure A.9 : Distribution des erreurs en fonction du temps (Problème des reines) Triplet Mobile (Inversion de bit)

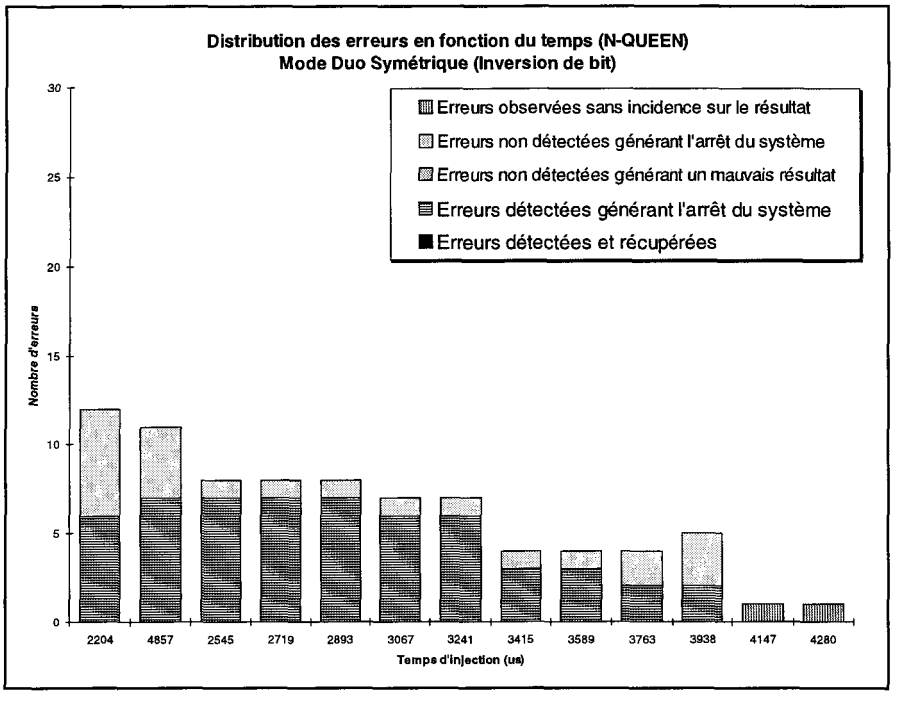

Figure A.10 : Distribution des erreurs en fonction du temps (Problème des reines) Mode Duo Symétrique (Inversion de bit)