Techniques d'Auto Réparation pour les Mémoires à Grandes Densités de Défauts

N.L. Achouri

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

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

T H E S E

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : Microélectronique

préparée au laboratoire TIMA dans le cadre de
l'Ecole Doctorale d'« Electronique, Electrotechnique, Automatique, Télécommunications, Signal »

présentée et soutenue publiquement

par

Nadir ACHOURI

Le 01 avril 2004

Titre :

TECHNIQUES D'AUTO REPARATION POUR LES MEMOIRES A GRANDES DENSITES DE DEFAUTS

Directeur de Thèse : Mihail Nicolaidis

JURY

M. Bernard Courtois	, Président
M. Abbas Dandache	, Rapporteur
M. Yves Crouzet	, Rapporteur
M. Mihail Nicolaidis	, Directeur
Mlle. Lorena Anghel	, Examinateur

TABLE DES MATIERES

Introduction

Les technologies de silicium se rapprochent de leurs limites physiques en termes de réduction de taille des transistors, et de la tension d'alimentation et de seuil, d'augmentation de la vitesse de fonctionnement et du nombre de dispositifs dans une puce (densité d'intégration). En se rapprochant de ces limites, les circuits deviennent très sensibles aux différentes sources de bruit "cross-talk", "ground bounce" ainsi qu'aux particules alpha et aux neutrons atmosphériques. Les mémoires qui constituent aujourd'hui jusqu'à 80% de la surface totale des SoC (System on Chip), concentrent donc à elles seules la plus grande partie des défaillances. Ainsi, des techniques de tolérance aux fautes doivent être utilisées dans les prochaines générations des circuits intégrés et tout particulièrement pour les mémoires, afin de contrecarrer ces problèmes et d'augmenter le rendement en production et en fiabilité. Néanmoins, même en utilisant ces techniques, les technologies actuelles auront atteint leurs limites d'ici l'an 2010. Une telle situation bloquera tout progrès dans le domaine de la micro-électronique et des technologies de l'information mettant en péril un des secteurs les plus dynamiques de l'économie mondiale. Pour pouvoir continuer de progresser, des technologies nouvelles telles que les dispositifs à un électron, les automates cellulaires quantiques ou les dispositifs moléculaires sont actuellement étudiées par de nombreux groupes de recherche, des laboratoires publics et industriels de part le monde.

Les avantages majeurs de ces technologies se situent au niveau de la miniaturisation qui est beaucoup plus grande que dans les circuits CMOS, au niveau de la consommation de puissance et du coût de fabrication qui sont beaucoup plus faibles. Mais leur grande faiblesse réside dans un rendement très faible lié à la grande miniaturisation ainsi qu'aux procédés de fabrication qui sont utilisés pour ces technologies. Il est considéré comme impossible de réaliser de tels circuits sans qu'un très grand nombre de dispositifs soient défaillants [HEA98], [HAN02]. Ainsi, la tolérance aux fautes devient le seul moyen qui permettra dans le futur de réaliser de façon économique des systèmes incluant des centaines de milliards de dispositifs. Néanmoins, pour les très grandes densités de défauts liées à ces technologies, l'hypothèse de base des techniques de tolérance aux fautes conventionnelles, à savoir, un seul composant du circuit est défaillant, n'est plus valable. En fait, à cause des grandes densités de défauts, les parties fonctionnelles ainsi que les parties redondantes contiendront des fautes, nécessitant le développement de techniques de tolérance aux fautes innovantes.

Les derniers développements dans le domaine des techniques de tolérance aux fautes pour les mémoires se situent essentiellement au niveau de l'auto-test intégré ou Built-In Self Test (BIST). Le BIST est une technique dans laquelle la génération des vecteurs de test et l'analyse des résultats sont effectuées d'une manière interne au circuit. Cette approche a l'avantage de se passer d'un équipement de test externe coûteux, de réduire le temps de test en exploitant les parallélismes structurels et fonctionnels qu'offre les architectures BIST, et permet enfin de tester les mémoires à leur fréquence nominale.

Ainsi, le test intégré a permis et continu d'être utilisé pour détecter puis éliminer pendant les phases de production ou de fonctionnement normal du système, les mémoires défaillantes. Mais au fur et à mesure que les densités de défauts augmentent, l'action de test n'est plus suffisante si on veut maintenir des niveaux acceptables de rendement en production. Dans ce cas le test reste certes nécessaire pour détecter les défaillances, mais doit être suivie par une action de réparation.

La réparation des mémoires se faisait traditionnellement d'une manière externe en (de)connectant, à l'aide du laser ou en brûlant des fusibles (ou anti-fusibles), les unités non défaillantes au bus de donnée (Hard Repair). Plusieurs désavantages sont liés à cette méthode. Premièrement, elle ne fait pas partie de la technologie standard CMOS, ce qui implique des coûts en surface importants. De plus la façon très agressive avec laquelle ce dispositif de réparation est implémenté, contribue fortement à la baisse du rendement de production. Un autre inconvénient est que ce type de réparation ne peut être utilisée pour les mémoires embarquées à cause des problèmes d'observabilité et de contrôlabilité. Aussi, l'action de réparation ne peut être effectuée qu'une seule fois, généralement en phase de production, ce qui limite la fiabilité de la mémoire pendant le fonctionnement normal du système. Enfin, son coût est très élevé, surtout lorsque les technologies laser sont employées.

La dernière avancée dans le domaine de la réparation mémoire, comme pour le cas du test, est l'auto réparation intégrée aux puces ou Built-In Self-Repair (BISR). Le BISR est une technique qui comprend la localisation des fautes, la sélection des unités redondantes pour le remplacement des unités défaillantes, et la connexion de ces unités redondantes au bus de donnée. L'utilisation du BISR est adaptée aux mémoires embarquées et pour lesquelles on a peu d'observabilité et de contrôlabilité (accès limité). Elle est aussi intéressante pour éviter un coût conséquent du à une reconfiguration externe basée sur l'utilisation du laser. Enfin, le BISR permet de remédier à la fois aux défauts de fabrication et aux défauts de durée de vie élevée puisqu'il peut être utilisé à tout moment pendant la vie du circuit. Il existe cependant plusieurs contraintes qu'il faut prendre en considération lorsqu'on veut développer une

solution BISR. Tout d'abord il faut veiller à minimiser le plus possible le coût en surface du circuit de réparation, implémenter des algorithmes de réparation qui utilisent efficacement les ressources redondantes pour faire face à des distributions variées de fautes, minimiser l'impact sur le temps de l'accès mémoire, réduire le temps globale de test/réparation et faciliter l'interfaçage avec le BIST. Il est cependant difficile d'allier l'ensemble de ces exigences en même temps, et la solution est plutôt de trouver des compromis, particulièrement entre l'efficacité de la réparation, le surcoût en surface, et la pénalité temporelle.

L'évaluation précise de l'efficacité d'une technique de réparation est une démarche très importante, qui permet soit d'éliminer ou d'améliorer une approche qu'on a développé, ou tout simplement la caractériser en fonction de différents paramètres (nombre des unités redondantes que la mémoire utilise, densité de défaut, …).

Déterminer l'efficacité d'une technique de réparation en présence d'une densité de défaut, est un problème qui s'est souvent posé en termes de formalismes mathématiques. La difficulté d'une approche mathématique, réside dans le fait qu'il faut établir des équations analytiques qui reflètent le comportement de la technique de réparation cible. Ainsi, plus la technique de réparation est complexe et plus la recherche de ces équations est fastidieuse. De plus, la complexité de l'approche analytique ne dépends pas uniquement de l'algorithme de réparation mais aussi du modèle de distribution des fautes qu'on veut adopter, et qui ne manque pas d'être complique des qu'on veut mimer les distributions de fautes les plus réalistes.

Objectif de la thèse

L'objectif de cette thèse est double. Premièrement, proposer des solutions de réparation intégrée efficaces pour un large spectre de densités de défauts, partant des densités de défauts affectant les technologies actuelles ($Dd = x.10^{-6}$ ou quelques cellules défaillantes sur un million de cellules), jusqu'à des densités de défauts extrêmes ($Dd = x.10^{-2}$). Pour de telles densités de défauts, la majorité des parties de la mémoire fonctionnelle ainsi que les parties redondantes sont défaillantes, ce qui implique pour nous le développement de solutions de réparation innovantes. Ce défi est d'autant plus difficile à relever qu'il s'agit non seulement de proposer des algorithmes de réparation mais aussi l'implémentation matérielle optimale en terme de coût de ces algorithmes. Pour réaliser ce premier objectif, nous avons procédé par étapes successives durant lesquelles on a développé des solutions de réparation de plus en plus puissantes.

Le deuxième objectif de cette thèse s'articule autour de l'évaluation de l'efficacité de ces solutions de réparation. Pour ce faire, nous avons développé une nouvelle approche software basée sur des simulations statistiques d'injection de fautes. L'idée est de décrire le comportement d'une technique de réparation donnée, en termes de manipulation des ressources redondantes, et de l'appliquer sur un certain nombre de mémoires ayants des distributions de fautes variées. Cette approche est une alternative à l'élaboration de formules analytiques difficiles à mettre en œuvre particulièrement pour les techniques de réparation complexes. Etant donné le nombre important de simulations à réaliser, nous avons automatisé le processus d'expérimentations et nous l'avons étendu de sorte qu'il puisse tenir compte de la plupart des techniques de réparation et des types de redondances existantes en industrie.
Deux types d'évaluations de l'efficacité ont été considérés, une évaluation au niveau de l'instance mémoire et une évaluation sur un ensemble de mémoires d'un même wafer. Pour cette dernière, un modèle d'amas de fautes a été adopté afin d'approcher le plus possible la réalité.

Organisation du manuscrit

Le manuscrit est organisé de la manière suivante. Dans un premier chapitre nous introduisons le test des mémoires RAM (Random Access Memory), étape préalable à toute action de réparation. Nous commençons dans ce même chapitre par évoquer l'évolution technologique des mémoires RAM, la modélisation fonctionnelle des fautes qui peuvent les toucher, les algorithmes nécessaires à la détection des fautes fonctionnelles, et les principales alternatives d'implémentation des algorithmes de test en vue d'un test intégré.

Le deuxième chapitre présente les principales techniques de réparation qui existent dans la littérature, qu'elles soient intégrées ou pas, ainsi que les modélisations et les algorithmes sur lesquelles elles se basent. Nous conclurons par le fait que le problème de réparation est très difficile traiter en vu d'une réparation intégrée a cause des fortes exigences en termes de coût en surface, d'efficacité et de temps de réparation, et qu'il est intéressant de ne plus traiter le problème de réparation comme étant un problème de recouvrement d'une matrice fautive à l'aide d'un jeu de lignes et de colonnes redondantes (problème NP complet).

Nous présentons dans un troisième chapitre, la réparation intégrée au niveau bit de donnée. Pour simplifier l'analyse du problème et optimiser le coût en surface, nous avons adopté une approche récursive qui permet de reconfigurer les données de la mémoire quelques soit la distribution des fautes sur les parties régulières et/ou redondantes. Afin d'améliorer l'efficacité de réparation de l'approche de base, on a développé un mécanisme de

reconfiguration dynamique qui permet de sélectionner de plus petites tailles d'unités réparables sans modifier la structure de la mémoire. Nous avons également élaboré des fonctions de reconfigurations capables de manipuler des colonnes redondantes singulières (sous ensemble d'une unité responsable de générer un bit de donnée) pour minimiser le coût du aux ressources redondantes.

Le quatrième chapitre propose un certain nombre d'approches de réparation destinées à contrecarrer les très hautes densités de défauts dans les mémoires, d'une part en se basant sur les polarités de fautes et d'autre part en combinant plusieurs techniques de réparation.

Le cinquième chapitre expose les différents problèmes liés à l'évaluation de l'efficacité d'une technique de réparation donnée, et propose une nouvelle approche pour y remédier à la fois au niveau de l'instance mémoire et au niveau wafer (ensemble de mémoires). L'approche a été automatisée à l'aide d'un outil avec lequel on a pu classifier les différentes solutions de réparation du chapitre III et V en termes d'efficacité et de coût en surface.

Notons que cette thèse a été effectuée dans le cadre d'un projet européen « FRACTURE » : Fault-Tolerant Architectures and Nanoelectronic Devices. Ce projet d'une durée de trois ans, comprenait plusieurs axes de recherche et a nécessité une forte collaboration entre des laboratoires publics, universitaires et industriels. Les deux principaux sujets ont été (1) le développement de procèdes économiques (peu coûteux) pour la fabrication des mémoires non volatiles en nanotechnologies, (2) et le développement de techniques de tolérance aux fautes pour les mémoires dans le contexte des nanotechnologies (très grandes densités de défauts).

CHAPITRE I. Test des Mémoires RAM

Introduction

Les systèmes digitaux tels que les ASIC, microprocesseurs et SoC (System on Chip) contiennent de plus en plus de mémoires qui occupent aujourd'hui jusqu'à 80 % de la surface totale de la puce et concentrent à elles seules des millions de transistors. Les mémoires sont construites avec le plus petit transistor qu'une technologie peut fabriquer, ce qui les rend de plus en plus sensibles à toutes les sources de bruit. Elles contiendront par conséquent, la majorité des défauts d'un circuit complexe SoC. Afin de garantir la qualité du système, il est donc décisif de détecter et éventuellement diagnostiquer les différents types de fautes qui peuvent toucher les mémoires.

Tester exhaustivement des mémoires ayant des capacités de l'ordre du Mb n'est pas une chose pratique. Des efforts considérables ont permis de contourner ce problème grâce à la modélisation des fautes. Il s'agit en effet de regrouper en chaque modèle, diverses fautes ayant sur le circuit des manifestations équivalentes. Cette modélisation doit être à la fois simple tout en étant réaliste.

En se basant sur les différents modèles de fautes, une multitude d'algorithmes de test ont vu le jour. Ces algorithmes ont des complexités qui varient essentiellement en fonction des types de fautes et donc de la couverture de fautes que l'on veut.

Dans ce premier nous allons chapitre commencer par introduire la notion de mémoire RAM (Random Access Memory) ainsi que son évolution technologique. Suivra ensuite une revue générale des différentes fautes susceptibles d'affecter les RAM, leur modélisation ainsi que les techniques de test nécessaires à leur détection.

1.1 Principe de la Mémoire RAM

Le principe de la mémoire RAM comme d'ailleurs de tout type de mémoire (ROM, EPROM, FIFO, …, etc.) est de stocker des informations et de les restituer en temps voulu. Les RAM sont également appelées "mémoires volatiles" du fait qu'elles perdent leur contenu après leur mise hors tension. Une particularité des RAM est l'accès aléatoire à n'importe laquelle des cellules mémoire que se soit en lecture ou en écriture.

De part leur spectre d'application très large, les RAM possèdent des implémentations très diversifiées et souvent complexes. Cependant, en les modélisant au niveau fonctionnel, on retrouve un squelette commun donné par la Figure 1.1.

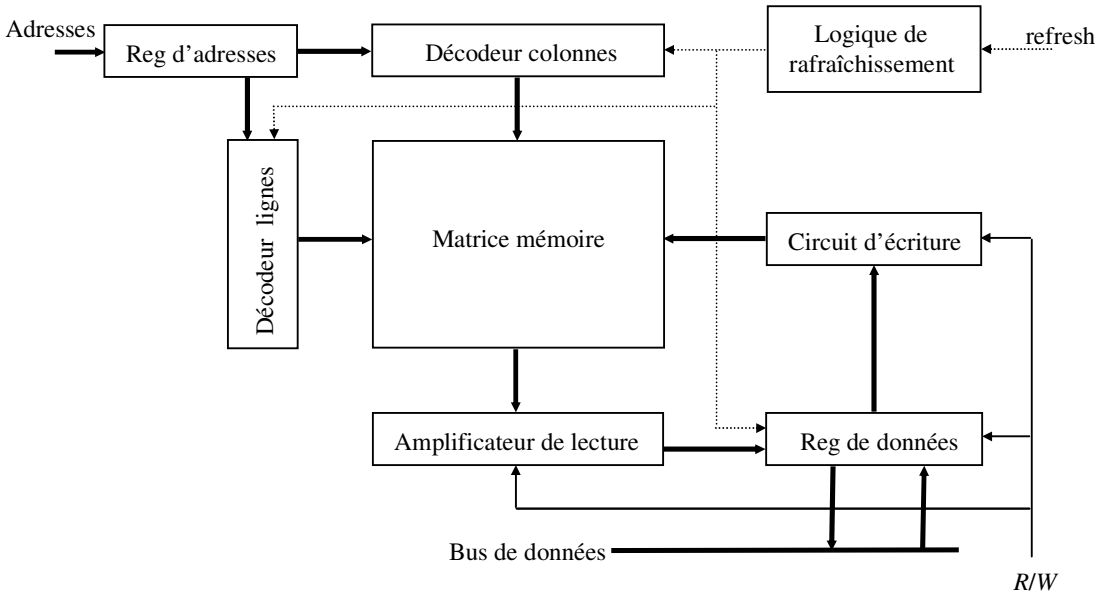


Figure 1.1 Modèle fonctionnel d'une RAM

Les principaux éléments d'une RAM sont donc :

- La matrice mémoire : elle contient les éléments de stockage (cellules) de la RAM. Elle est sous forme carrée ou rectangulaire afin de minimiser la surface et faciliter l'implémentation. Elle peut être organisée en plusieurs sous matrices.
- Le circuit de décodage : il est composé d'un registre d'adresses qui stocke les adresses, du décodeur ligne qui est souvent connecté aux adresses de poids fort et du décodeur colonne connecté aux adresses de poids faible. Le décodeur ligne sélectionne une ligne de la matrice mémoire, ensuite, selon la taille du mot mémoire (*B* bits), le décodeur colonne sélectionne *B* colonnes de cette ligne pour une opération de lecture ou d'écriture.
- La logique de lecture/écriture : le signal *R/W* (lecture/écriture) sélectionne le mode de fonctionnement de la RAM. Dans le cas d'une lecture, le contenu de(s) la cellule(s) sélectionnée(s) est amplifié par l'amplificateur de lecture. Ces données seront ensuite chargées dans le registre de données avant d'être acheminées vers le bus de données. Dans le cas d'une opération d'écriture, les données se trouvant sur

le bus de données sont chargées dans le registre de données, et écrites dans la matrice mémoire via l'amplificateur d'écriture.

- Dans le cas des RAM dynamiques (DRAM), les données sont stockées sous forme de charges capacitives. De ce fait, leurs cellules mémoire ont besoin d'un circuit de rafraîchissement pour restaurer leurs contenus. Ce rafraîchissement est nécessaire après que les cellules mémoires sont accédées en lecture, ou ne sont pas accédées pendant un "long" moment, ce qui entraîne dans un cas comme dans l'autre la décharge des capacités correspondantes.

Les performances d'une RAM, exprimées généralement en nano secondes, sont données par son temps d'accès (en lecture), et sa durée de cycle. Le temps d'accès représente le temps de propagation depuis le temps de prise en compte de l'opération de lecture (i.e au front montant de l'horloge) jusqu'à l'apparition des données sur le bus de sortie. La durée du cycle d'une mémoire est définie comme étant le temps minimal nécessaire entre deux accès successifs à une RAM. Les spécifications temporelles d'une RAM sont généralement données par des chronogrammes. Ceux-ci précisent les spécifications temporelles des signaux d'entrée entre eux, et de ces derniers avec les signaux de sortie (délais). Ces chronogrammes sont spécifiques à chaque type de RAM et varient en fonction de leurs interfaces et de leurs protocoles d'accès.

La section suivante concernera une revue des différentes technologies RAM ainsi que la présentation de quelques modèles électriques pour la compréhension des modèles de fautes basés sur ce niveau d'abstraction (électrique).

Notons que pour la suite de ce mémoire, nous utiliserons les conventions suivantes : le symbole *m* représente la taille des adresses, *n* celui des données.

1.2 La Technologie des Mémoires RAM

L'évolution technologique des mémoires RAM a essentiellement été motivée par deux objectifs : achever les meilleures performances possibles et atteindre de grandes densités d'intégration. Les mémoires SRAM (Static RAM) ont souvent assuré le premier objectif. Tandis que les mémoires DRAM, avec leurs cellules spécifiques atteignent de loin les densités les plus élevées, et sont de ce fait utilisées comme indice et référence dans les nouveaux processus technologiques.

1.2.1 Cellules Mémoire

Les cellules mémoires de base sont différentes pour le cas des RAM dynamiques et statiques. Ainsi, on peut trouver des cellules de mémoire SRAM et DRAM. Leurs caractéristiques sont présentées ci-dessous.

1.2.1.1 La cellule SRAM

La cellule SRAM est conçue sous forme de bascules bistables. Elle peut donc être forcée à un des deux états logiques 0 ou 1, et rester dans cet état après la disparition de ces stimuli. Elle peut être modélisée par deux inverseurs couplés. La connexion avec les lignes électriques qui véhiculent les bits de donnée en écriture ou en lecture (lignes de bits *LB*) est assurée par deux transistors de passage NMOS commandés par un signal *LM*. Le signal est envoyé à toutes les cellules appartenant à la même rangée de la mémoire, permettant ainsi l'accès en écriture ou en lecture à un mot de mémoire particulier.
Il existe différentes approches pour l'implémentation d'une cellule SRAM (Figure 1.2) :

- Cellule mémoire à six transistors NMOS dont deux transistors $Q3$ et $Q4$ de déplétion. Solution peu coûteuse à cause du peu d'étapes technologiques mais consommatrice de courant (Figure 1.2(a)).
- Une amélioration significative de la surface en silicium et de la puissance de dissipation est obtenue en remplaçant les transistors de déplétion ($Q3$ et $Q4$ de la Figure 1.2(a)) par des résistances de charge en poly-silicium (Figure 1.2(b)). Cette approche implique une opération de lecture relativement lente à cause du chargement à travers une résistance de grande densité ohmique et nécessite donc une phase de pré charge.
- Cellule CMOS avec charges en PMOS. C'est la solution la plus optimale au niveau compromis entre vitesse et consommation de courant. Elle est cependant assez coûteuse à cause des étapes de fabrication supplémentaires des transistors PMOS (Figure 1.2(c)). Dans le cas de mémoires intégrées dans un circuit VLSI CMOS, cette solution s'avère intéressante.

Les transistors $Q1$-$Q3$ forment un inverseur, qui est couplé en opposé avec le second inverseur $Q2$-$Q4$, pour former une bascule. Les transistors de passage $Q5$ et $Q6$ permettent l'accès à cette bascule pour une opération de lecture ou d'écriture. Le décodeur ligne

sélectionne une ligne de cellules en activant son signal ligne de mot (*LM*). Ce signal est connecté aux transistors de passage de toutes les cellules de cette ligne. Pour sélectionner une cellule particulière dans cette ligne, le décodeur colonnes active les signaux de ligne de bit ($LB, \overline{LB}$) correspondants.

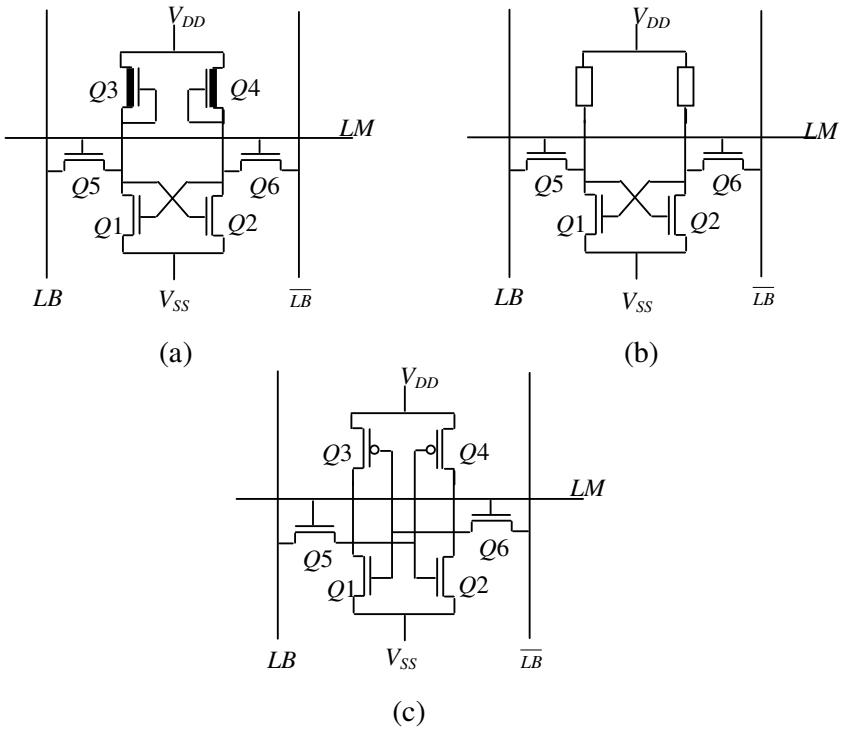


Figure 1.2 Cellules SRAM

Pour écrire dans une cellule appropriée, on doit mettre à 1 le signal LM pour rendre les transistors *Q*5 et *Q*6 passants, et forcer les signaux *LB* et $\overline{LB}$ avec des données complémentaires. La cellule sera forcée à l'état des signaux lignes de bit grâce à leurs charges relativement élevées en comparaison avec celles de la cellule.

Dans une opération de lecture, les signaux *LB* et $\overline{LB}$ sont chargés au même niveau haut. Après l'accès à la cellule, la donnée stockée sera déterminée selon la différence de tension entre *LB* et $\overline{LB}$. Cette différence est due aux valeurs complémentaires stockées dans les transistors *Q*3 et *Q*4 et propagées à *LB* et $\overline{LB}$ en sélectionnant le signal *LM*.

1.2.1.2 La cellule DRAM

La mémorisation dynamique est basée sur le stockage d'une charge (électrons) dans une capacité (parasite ou non) et par conséquent on ne peut garantir l'intégrité de l'information stockée que pendant un temps qui peut être déterminé avec une certaine précision. Cette durée de vie limitée est due aux courants de fuite existant dans les structures MOS et les jonctions inversement polarisées. Plusieurs approches existent pour implémenter cette capacité. La plus courante est la cellule DRAM à un transistor. Conçue en 1973, elle devient un standard afin d'être utilisée pour les DRAM de 16 Kbits jusqu'à celles de 1 Mb. Cette cellule est constituée d'un transistor enrichi et d'une capacité qui occupe environ 30 % de sa surface.

Dans une opération de lecture, le signal *LB* est pré chargé au même niveau de tension que l'amplificateur de lecture. Ensuite, le signal *LM* est mis à 1, permettant le transfert de la charge de la capacité à *LB*. Ceci cause une différence de potentiel sur *LB*, dont la valeur est déterminée suivant la charge de la capacité et de la capacitance de *LB*. Un circuit de lecture très sensible tel que celui de la Figure 1.6 est nécessaire pour détecter cette faible

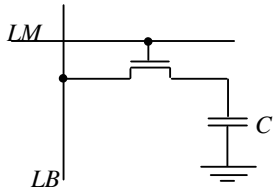


Figure 1.3 Cellule DRAM à un transistor

variation. Cette opération de transfert de charge rend l'opération de lecture destructive, nécessitant ainsi une réécriture des informations lues.

L'écriture consiste à forcer *LB* au niveau désiré, et en même temps mettre *LM* à la valeur logique 1.

1.2.2 Les Décodeurs d'Adresses

Les décodeurs d'adresses sont utilisés afin de pouvoir, à partir d'une adresse donnée, d'accéder à une cellule particulière ou à un groupe de cellules dans la matrice mémoire. On utilise souvent des décodeurs ligne et colonne, car un adressage en deux dimensions permet de réduire par un facteur $0{,}5\sqrt{n}$ (où n est le nombre de bits dans la matrice mémoire) la surface occupée par les décodeurs d'adresses [GOO91]. La Figure 1.4 montre une architecture basée sur un décodage de 8×8 qui illustre le principe de l'adressage en deux dimensions.

Pour les décodeurs ligne, on a généralement le choix entre une structure NOR statique ou dynamique. La première est purement combinatoire (sans horloge), elle consomme peu de

puissance, mais elle est relativement lente pour un grand nombre de bits à décoder et consomme aussi une plus grande surface en silicium. La deuxième est plus rapide, car elle utilise une d'horloge, mais consomme plus de courant [MAZ88], [GOO91].
Le décodeur colonne a pour fonction la sélection d'un certain nombre de bits parmi ceux de la ligne accédée. Il peut être réalisé avec un arbre de transistors (simple mais relativement lent) ou des interrupteurs (CMOS ou NMOS) qui sont plus rapides.

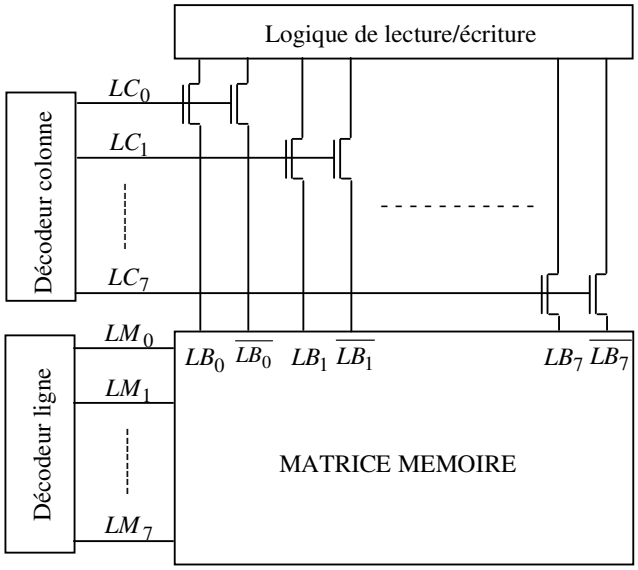


Figure 1.4 Décodage ligne et colonne

1.2.3 Logique de Lecture/Ecriture

Le circuit d'écriture d'une RAM est assez simple comme le montre la Figure 1.5. La donnée en entrée (DI) est transmise aux lignes de bits *LB* et $\overline{LB}$ pour être ensuite stockée dans la cellule mémoire. Cette transmission de la donnée est contrôlée par le signal "Write Enable" (WE).

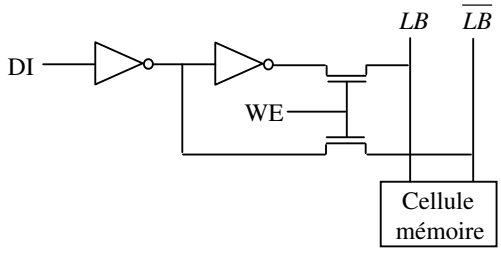


Figure 1.5 Circuit d'écriture RAM.

Le circuit de lecture d'une RAM est donné par la Figure 1.6. On utilise généralement un amplificateur différentiel qui permet une commutation rapide et une bonne détection des faibles variations de tension sur les lignes *LB* et $\overline{LB}$.

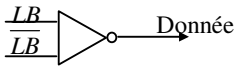


Figure 1.6 Circuits de lecture RAM.

Après avoir passé en revue le principe et les différentes technologies des RAM, nous allons présenter dans la prochaine section les fautes susceptibles d'apparaître dans ces mémoires ainsi que leur modélisation.

1.3 Les Fautes dans les RAM

On parle d'une faute, lorsqu'il existe une différence physique entre le système correct (référence) et le système en question. Cette faute se transforme en erreur si les sorties du système sont différentes des valeurs attendues. Le passage de la faute à l'erreur peut ne jamais se produire (masquage de faute) ou se produire d'une manière différée. L'erreur conduit à une défaillance si elle est propagée jusqu'à la sortie du système (mise à défaut de l'application). Il existe deux grands types de fautes :

- Les fautes permanentes (appelées aussi fautes matérielles), sont présentes dans le système d'une manière permanente. Elles peuvent provenir de plusieurs sources pendant et après la fabrication. Pendant la fabrication, à cause des mauvais alignements des masques, des procédés technologiques instables, et de la contamination par des particules. Elles peuvent également apparaître après la fabrication, à cause d'un environnement agressif qui peut provoquer des courts circuits et des ouvertures dans les pistes métalliques.
- Les fautes non permanentes se produisent à des instants aléatoires et affectent le comportement du système pendant une certaine durée. Les tests de fabrication ne peuvent donc pas les détecter. Si on ajoute à cela le fait qu'elles possèdent un pourcentage élevé par rapport à l'ensemble des types de fautes (88 à 97 % du nombre total de fautes d'après [GOO91]), on comprend que le test des fautes non permanentes est à la fois important et difficile à réaliser. En réalité il nécessite un type de test particulier qui est le teste en ligne. Les fautes non permanentes peuvent à leur tour être

subdivisées en fautes transitoires (appelées aussi fautes soft), causées par les conditions d'environnement tels que les rayons cosmiques, les particules alpha, pollution, pression …, et en fautes intermittentes dues au bruit qui perturbe les signaux dans le système, variations des résistances et des capacitances, irrégularités physiques, … etc.

Remarque : Nous allons dans tout ce qui suit nous intéresser uniquement aux fautes permanentes que ce soit au niveau de la modélisation des fautes, leur test ainsi que leur réparation.

1.3.1 La Modélisation des Fautes pour RAM

Les défauts physiques sont souvent modélisés au niveau logique. Ceci permet, entre autres, de contourner l'examen physique fastidieux du circuit, et permet aussi de développer des tests technologiquement indépendants du circuit sous test.
La modélisation des fautes est une opération complexe qui vise à trouver le meilleur compromis possible entre une reproduction fidèle du comportement du défaut réel et une modélisation simple de celui-ci. Le premier aspect assure une bonne couverture des défauts réels et le second amène à un coût raisonnable pour la simulation et la génération des vecteurs de test.

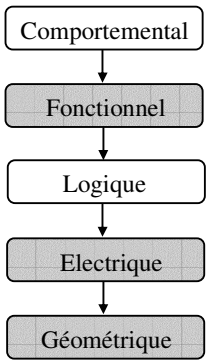


Figure 1.7. Niveaux d'abstraction d'une RAM pour la modélisation des fautes.

Le fonctionnement d'une mémoire RAM peut être modélisé dans différents niveaux d'abstraction (Figure 1.7). Cependant, pour le test des RAM, on n'utilise généralement que des modèles de fautes basés sur des niveaux fonctionnels, électrique ou géométrique. Le niveau comportemental est peu précis par rapport au niveau fonctionnel. Le niveau logique

n'est pas considéré, puisque la RAM est constituée essentiellement de cellules mémoires et non de portes logiques.
Seules les fautes basées sur le modèle fonctionnel de la RAM seront développées dans les sections suivantes.

1.3.2 Les Fautes Fonctionnelles

La Figure 1.8 donne une version simplifiée du modèle fonctionnel de la RAM. Ce modèle, appelé le modèle fonctionnel réduit, est plutôt adapté au cas du test et non à localisation des fautes. Il a été largement utilisé [THA78], [ABA83], et [GOO90] à cause de sa simplicité et sa représentation des fonctions essentielles d'une RAM.
Utilisant ce modèle fonctionnel réduit, l'ensemble des fautes liées aux blocs de la Figure 1.1 se réduit à quatre grands modèles de fautes qu'il suffit de détecter en testant uniquement la matrice mémoire. Une définition de ces modèles de fautes est donnée ci-dessous.

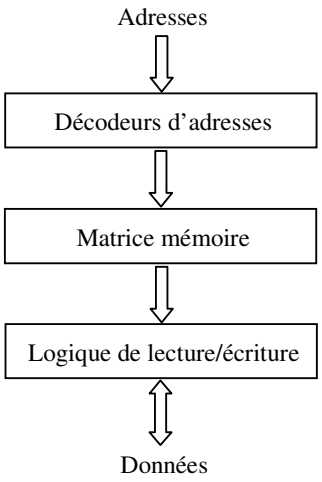


Figure 1.8 Modèle fonctionnel réduit de la RAM

a) Les fautes de collage (Stuck at Faults : SAF) : une cellule mémoire collée à 0 ou à 1 ne peut être mise à 1 ou à 0 respectivement, par une opération d'écriture.

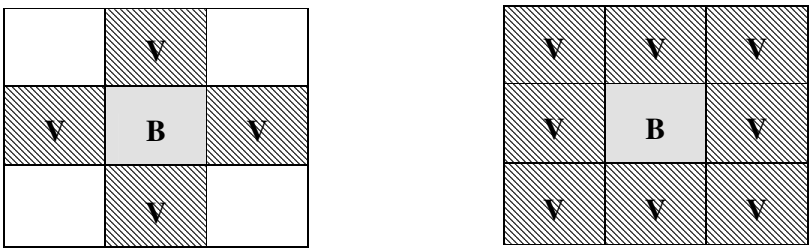
b) Les fautes de transition (Transition Faults : TF) : une cellule qui ne réalise pas la transition $0 \rightarrow 1$ présente une faute de transition positive. De même, une cellule qui ne réalise pas la transition $1 \rightarrow 0$, présente une faute de transition négative.

c) Les fautes de couplage (Coupling Faults : CF) : une faute de couplage implique deux cellules. Une opération d'écriture qui génère les transitions $0 \rightarrow 1$ ou $1 \rightarrow 0$ dans une cellule *j*, change l'état d'une autre cellule *i* de *s* à $\bar{s}$ avec $s \in \{0,1\}$

indépendamment du contenu des autres cellules. Les CF sont causées par les courts-circuits et les effets parasites, tels que les courants de fuite, ou les capacités d'isolement. En réalité il s'agit d'un cas particulier des fautes de *k*-couplage. Celles-ci se manifestent lorsqu'en plus des deux cellules *i* et *j*, il existe *k*–2 cellules se trouvant n'importe où dans la matrice mémoire, avec des états particuliers. La classification des fautes de couplage la plus significative est donnée par Marinescu [MAR82] :

- Fautes de couplage idempotent (Idempotent Coupling Faults : CFid) : une transition positive ou négative d'une certaine cellule force le contenu d'une autre cellule à la valeur 0 ou 1.
- Fautes de couplage inverse (Inverting Coupling Faults : CFinv) : une transition positive ou négative d'une certaine cellule inverse l'état d'une autre cellule indépendamment du contenu de celle-ci.

d) Les fautes sensibles à la configuration du voisinage (Neighborhood Pattern Sensitive Faults : NPSF) : une cellule est sujette à une faute sensible à la configuration du voisinage quand son contenu ou sa capacité de changer d'état dépend de l'état et/ou des transitions des autres cellules de la mémoire. Les PSF sont causées par la grande densité des RAM, ainsi que par des interférences indésirables (e.g. électromagnétiques). Le test de ce genre de fautes requiert l'usage d'algorithmes très complexes et très lents à appliquer. Dans la pratique, on utilise des cas particuliers de PSF qui sont les NPSF. On fait donc l'hypothèse qu'étant donné une cellule B (appelée cellule de base) les cellules pouvant l'influencer se trouvent dans son voisinage immédiat [SUK80].



Voisinage 5-cellules Voisinage 9-cellules

Figure 1.9. La cellule de base et son voisinage.

Deux principaux types de voisinage sont définis : l'un comprenant les quatre cellules situées sur la même ligne et la même colonne de la cellule de base et l'autre incluant

tous les voisins immédiats de B (Figure 1.9). L'hypothèse est raisonnable car il est très probable que les cellules voisines de la cellule de base aient plus d'influence que des cellules éloignées.

Les NPSF peuvent être classées en :

- NPSF actives [SUK80] : la cellule de base change son contenu suite à un changement dans la configuration des cellules avoisinantes (une transition positive ou négative d'une cellule avoisinante, et le reste est dans un certain état). Le test pour ces fautes doit effectuer la lecture, depuis chaque cellule, de l'état 0 et de l'état 1, et ceci pour toutes les transitions possibles du voisinage de la cellule de base.
- NPSF passives [SUK80] : influencé par le contenu des cellules avoisinantes, le contenu de la cellule de base ne peut être changé. Pour tester ces fautes, les valeurs 0 et 1 doivent être écrites et lues pour chaque cellule, et ceci pour toutes les combinaisons possibles du voisinage de la cellule de base.
- NPSF statiques [SAL85] : le contenu de la cellule de base est forcé à une certaine valeur par le contenu de la configuration des cellules avoisinantes. Pour tester ces fautes, on doit lire l'état 0 et 1 de chaque cellule, et ceci pour toutes les combinaisons possibles du voisinage de la cellule de base.

Les fautes présentées ci-dessus s'appliquent aussi bien à la matrice mémoire qu'aux décodeurs et à la logique de lecture/écriture. Il a été prouvé [NAI78], [THA77] que les collages, les fautes de transition et les couplages dans la logique de lecture/écriture peuvent être transposées dans la matrice mémoire sous forme de couplages. Il suffit par conséquent de tester la matrice mémoire pour pouvoir détecter les fautes affectant tous les blocs du modèle fonctionnel de la Figure 1.1. Il en est de même pour le décodeur. Il a été néanmoins démontré [GOO90] que les fautes de type NPSF ne peuvent être transposées à la matrice mémoire.

1.4 Tests Fonctionnels des RAM

L'objectif du test est la détection et éventuellement, la localisation (en vue d'une réparation) des différentes fautes pouvant exister dans le circuit.

Les algorithmes de test sont conçus dans l'objectif de couvrir toutes les fautes d'un modèle de faute particulier, avec le minimum de complexité possible. En général, les algorithmes de test pour RAM sont basés sur les modèles de fautes fonctionnels (la structure électrique de la

mémoire n'est pas prise en compte). Suivant les complexités et les taux de couverture qu'on cible, différents types de tests fonctionnels existent tels que les tests traditionnels, les tests March, et les tests NPSF.

1.4.1 Les Tests Fonctionnels Traditionnels

Se sont historiquement les premiers tests fonctionnels pour RAM. Les tests traditionnels sont également appelés tests libres car bien qu'ils appartiennent à la famille des tests fonctionnels, ils ne sont basés sur aucun modèle de faute. Ils sont plus coûteux mais ils ont quelques avantages comme la détection d'un certain nombre de fautes non fonctionnelles ; quelque uns d'entres eux permettent la localisation précise des fautes détectées. Enfin ils restent efficaces pour les mémoires de taille peu importante. Voici des exemples d'algorithmes qui constituent ce premier type de tests fonctionnels :

- L'échiquier (Checkerboard) : on se limite à diviser la mémoire en deux groupes de cellules comme sur un échiquier. On commence par écrire la valeur 0 sur les cellules du premier groupe et la valeur 1 sur les cellules du second groupe. Ensuite, toutes les cellules sont lues. On répète la même procédure en inversant les valeurs de chaque groupe.
- MSCAN : cet algorithme consiste à écrire 0 sur toute la mémoire, lire toute la mémoire et puis répéter l'opération avec 1.
- GALPAT, Walking 1/0 et Sliding diagonal : GALPAT et Walking 1/0 sont deux algorithmes très semblables. Au début du test, c'est toute la mémoire qui est initialisée à 0 ou à 1 excepté la cellule de base qui est mise à 1 (resp. à 0). Durant le test, la cellule de base parcourt toute la mémoire. La différence entre les deux algorithmes réside dans la lecture de la cellule de base. Pour le Walking 1/0, à chaque étape la cellule de base n'est lue qu'après avoir lu toutes les autres cellules. De la même manière pour le GALPAT, toutes les cellules sont lues, mais cette fois-ci la cellule de base est lue à chaque fois qu'une autre cellule est lue. Enfin l'algorithme Sliding diagonal a été conçu comme une version plus simple de GALPAT, la cellule de base représentant une diagonale se déplaçant dans la matrice mémoire.

Les tests Checkerboard et MSCAN ont une couverture de fautes très limitée puisque les décodeurs ne sont pas testés. Pour les collages par exemple, on ne peut qu'affirmer à la fin du

test qu'une seule cellule est bonne, puisqu'on n'est pas sûr d'avoir accédé à toutes les cellules de la matrice mémoire. GALPAT et Walking 1/0 assurent une couverture de fautes assez élevée (détection et localisation des fautes de collage, de transition et de couplage). La couverture de fautes du Sliding diagonal est moins élevée (tous les couplages ne peuvent être détectés).

Nous présentons ci-dessous les tests March qui à l'inverse des tests traditionnels, sont basés sur les modèles de fautes SAF, TF et CF, et qui assurent à 100% leur détection.

1.4.2 Les Tests March

Suk définit les tests de type March de la façon suivante [SUK80] : "Un test de type March est une séquence finie d'éléments de marche. Un élément de marche est une séquence finie d'opérations appliquées à chaque cellule de la mémoire avant de passer à la cellule suivante".

Exemple de test de type March : l'algorithme MATS. {⇑(W0) ; ⇑(R0, W1) ; ⇑(R1)}.

Le premier élément de marche ⇑(W0) consiste à initialiser toutes les cellules de la mémoire à 0 en effectuant des opérations d'écriture dans l'ordre croissant des adresses. Le deuxième élément de marche ⇑(R0, W1) consiste à lire chaque cellule de la mémoire (on attend un 0) et à écrire ensuite un 1 en parcourant la mémoire dans l'ordre croissant des adresses. Le dernier élément de marche ⇑(R1) consiste à lire toutes les cellules de la mémoire (on attend un 1) en effectuant des opérations de lecture dans l'ordre croissant des adresses. Voici quelques exemples des algorithmes March les plus performants avec ⇓ symbolisant un adressage décroissant et *n* le nombre d'adresses mémoire.

- Algorithme en 11*n* de Marinescu [MAR82] :

  ⇑(W0) ; ⇑(R0, W1) ; ⇑(R1, W0) ; ⇑(R0) ; ⇓(R0, W1) ; ⇓(R1, W0) ; ⇓(R0), détecte les SAF, les TF, les 2-couplages d'inversion et les 2-couplages dynamiques.

- Algorithme en 10*n* de Van de Goor [GOO91] :

  ⇑(W0) ; ⇑(R0, W1) ; ⇑(R1, W0) ; ⇓(R0, W1) ; ⇓(R1, W0) ; ⇓(R0), couverture de fautes identiques à celui du 11*n* de Marinescu.

- Algorithme en 15*n* de Suk [SUK81] :

  ⇑(W0) ; ⇑(R0, W1, W0, W1) ; ⇑(R1, W0, W1) ; ⇓(R1, W0, W1, W0) ; ⇓(R0, W1, W0), détecte les SAF, les TF, les 2-couplages idempotents et certains 2-couplages d'inversion.

1.4.3 Les Tests NPSF

Le problème du test des fautes de type NPSF se divise en deux parties : la génération de séquences de test et la génération d'adresses. Les séquences de test sont généralement déterminées à l'aide de deux méthodes qui visent à assurer le nombre minimum d'opérations d'écriture. Ce sont les séquences de Hamilton (pour le test des SNPSF) et les séquences de Euler (pour le test des ANPSF et les PNPSF).

Il en est de même pour la génération des adresses où des méthodes sont utilisées avec succès. La première est appelée "méthode des groupes" [SUK80], [SAL85] et repose sur le fait qu'une cellule peut être une cellule de base à un instant et une cellule de voisinage à un autre instant. Cette méthode n'est valable que pour les voisinages de type 5 cellules de la Figure 1.9 à cause de problèmes de symétrie. Dans la seconde méthode appelée "tilling ou pavage" [MAZ87], la matrice mémoire est complètement couverte par un graphe de voisinages de type tetromino, qui ne se chevauchent pas. Elle est parcourue de manière à ce que chacune des cellules réalise toutes les transitions et assume les valeurs prévues.

De nombreux algorithmes ont été proposés par différents auteurs. Ils réalisent la détection ou le diagnostic de mémoires pour chacune des trois fautes de type NPSF ou pour des combinaisons de celles-ci.

Après une revue des différents types de fautes dans les mémoires et les algorithmes de test assurant leurs couvertures, nous aborderons dans la section suivante, les différentes approches d'implémentation de ces derniers dans le cadre du test intégré.

1.5 BIST pour RAM

Le BIST est une méthode par laquelle on choisit de générer les vecteurs de test, et procéder à l'analyse des résultats, d'une manière interne au circuit. C'est une technique générale qui peut être appliquée à n'importe quelle logique (aléatoire, structurée : RAM, séquentielle, …).

Plus particulièrement, le BIST pour les mémoires RAM s'avère être un choix important voir nécessaire pour plus d'un cas :

- En remplaçant l'ensemble ou une partie du circuit de test, le BIST permet de se passer d'un équipement de test externe complexe (coûteux).
- Il permet également de réduire d'une façon considérable le temps du test, en générant d'une façon parallèle les vecteurs de stimuli niveau puce (un BIST pour chaque

mémoire, ou un BIST pour un ensemble de mémoires "Shared BIST"). C'est une nécessité pour les RAM dynamiques de grandes capacités (> 64 Mbits) à cause du temps nécessaire à leur test [FRA90].

- Le BIST permet le test de la mémoire à sa vitesse nominale et la possibilité de la tester à la fabrication ainsi qu'en phase d'opération.
- Le BIST est nécessaire pour les RAM enfouies à cause des problèmes d'observabilité et de contrôlabilité [NAD90].

Cependant, l'utilisation du BIST a également son prix. Elle conduira inévitablement à un surcoût en matériel et s'accompagne généralement d'une perte de performance.

1.5.1 Principe

La Figure 1.9 montre le schéma bloc d'un circuit BIST pour les mémoires RAM. Un générateur de données de test, génère la donnée de test, qui sera appliquée ensuite à l'adresse spécifiée par le générateur d'adresses. Le résultat est évalué par le vérificateur de réponse. Enfin, un contrôleur permet de contrôler et synchroniser les différents blocs du BIST. Chacun de ces blocs peut être implémenté de la manière suivante.

- Le générateur d'adresse et le générateur de données de test : pour les implémenter on a le choix entre une multitude de compteurs autonomes et exhaustifs tel qu'un Linear Feed Back Shift Register (LFSR), un automate cellulaire (CA) ou un compteur binaire.

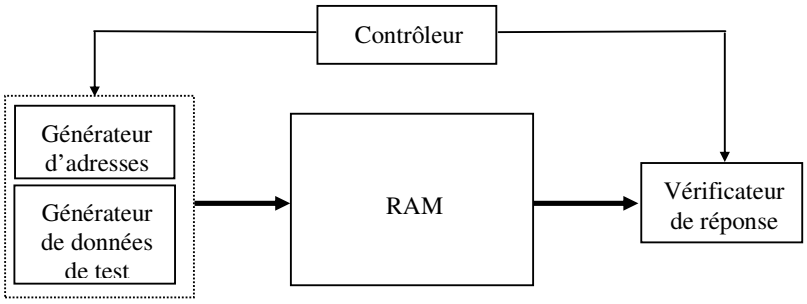


Figure 1.10 Diagramme bloc d'un BIST (parallèle) pour RAM.

- Le vérificateur de réponse : plusieurs stratégies de vérification de la réponse peuvent être utilisées. On peut comparer directement les données obtenues par le test avec celles attendues, c'est la Comparaison Déterministe [NIC85]. Le circuit correspondant

(formé de portes OU exclusives et un arbre binaire de portes OU) est simple à réaliser et peut être facilement testé avec une séquence de *Walking Ones* sur les entrées du comparateur plus la séquence 00…0. Une autre technique de vérification des résultats consiste à les compacter à l'aide d'un analyseur de signature à entrées parallèles multiples (Multiple Input Signature Analyser : MISR), et comparer (extérieurement) la signature obtenue avec la signature de référence, prédéterminée par simulation [BAR87]. Les analyseurs de signature sont auto testables par construction (tout problème dans leur structures se manifestera dans la signature finale). Cependant, le phénomène de masquage (deux séquences différentes produisant la même signature) doit être précisément quantifié et correctement pris en compte.

- Le contrôleur : représente la réalisation physique de l'algorithme de test choisi. Celui-ci peut soit correspondre à un algorithme bien particulier au quel cas, il offrira une cadence de fonctionnement élevée tout en ayant une surface modérée. Soit être conçu de façon à ce qu'il puisse facilement changer d'algorithme de test à implémenter (BIST programmable) pour cibler de nouveaux types de fautes au détriment d'un temps d'exécution plus important.

La section suivante passe en revue les différentes techniques de BIST mémoires utilisées jusqu'à maintenant.

1.5.2 Approches BIST pour RAM

Basés sur des modèles de fautes spécifiques et utilisant ou non des techniques DFT, différents BIST ont été proposés dans la littérature. Nous allons dans ce qui suit répertorier quelques exemples d'implémentation de circuits Built-In Self-Test qui ont été intégrés dans le générateur de BIST par notre équipe de recherche et développement [BOU02]. Ces techniques de BIST seront utilisées par la suite pour implémenter les techniques de réparation que nous proposons.

1.5.2.1 BIST basés sur les tests March

Les tests March sont actuellement très utilisés car ils offrent généralement un bon compromis entre la couverture de fautes, le coût matériel et la longueur de test.
Les architectures de BIST basées sur les tests de type March ont pour architecture de principe les approches introduites en 1985 par Mihail Nicolaidis [NIC85]. Cette architecture a été

formalisée afin de dériver une approche permettant une synthèse efficace de n'importe quel algorithme March. Cette approche est une décomposition hiérarchique de l'algorithme de test March, qui exploite au mieux sa structure régulière. Les simulations de cette structure matérielle ont montré qu'elle est à la fois plus compacte et plus rapide qu'une approche d'implémentation directe (basée sur un circuit séquentiel pour implémenter l'algorithme dans son ensemble). De plus sa nature modulaire simplifie d'une manière considérable le paramétrage et le management de ses blocs matériels par le synthétiseur de BIST.

La Figure 1.11 montre le diagramme bloc de ce modèle. Chaque niveau de l'algorithme March est implémenté par son bloc correspondant, qui peut être soit un bloc combinatoire soit un bloc séquentiel synchrone.

Cet ensemble de blocs interconnectés possède la hiérarchie suivante :

- Au niveau le plus élevé de la hiérarchie, un circuit séquentiel simple appelé Contrôleur de Séquences implémente les différentes séquences de l'algorithme de test March (également appelées *éléments March*).
- A l'intérieur de chaque séquence on a besoin de générer les adresses des différentes cellules mémoires. Ceci est pris en charge par le second niveau de la hiérarchie qui correspond au Générateur d'Adresses. Pour implémenter ce dernier, on a le choix entre plusieurs compteurs exhaustifs autonomes (LFSR, Cellular Automata, générateur de code Gray, …).

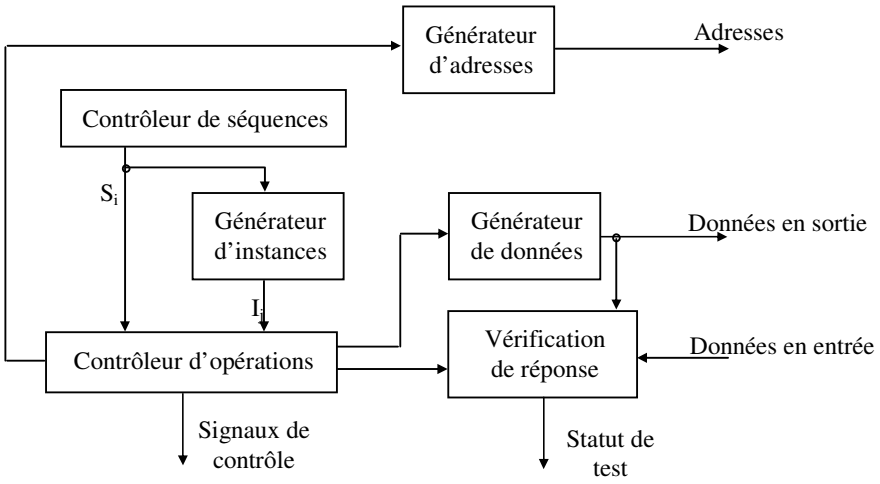


Figure 1.11 Architecture générique des BIST March.

- Au troisième niveau de la hiérarchie, le générateur d'Instances indique le cycle courant afin de pouvoir déterminer le type d'opération à effectuer (une opération de lecture, d'écriture, ou autre …) et la valeur logique des données à appliquer. D'une manière

basique, ce bloc est un compteur circulaire de taille égale au nombre maximal d'opérations contenues dans les éléments March de l'algorithme cible.

- Dans le même niveau de hiérarchie (troisième), le Contrôleur d'Opérations détermine, par le biais de quelques signaux de contrôle, la valeur de la donnée de test à utiliser pour chaque opération d'écriture, les signaux de contrôle de la mémoire (R/W, CS, OEN, …) ainsi que certains signaux de contrôle des autres modules. C'est un bloc combinatoire qui implémente un ensemble d'équations booléennes qui décode la séquence et l'instant en cours.

Le générateur de données est commandé par le contrôleur d'opérations pour dériver les différents backgrounds pour les mémoires orientées mots.
Le bloc de vérification de la réponse permet de vérifier le contenu de la mémoire. Il peut être implémenté en utilisant une comparaison déterministe (i.e. comparateur) ou une compression de données (i.e. MISR). La première implémentation compare la valeur lue avec la valeur attendue, et rapporte l'état de la comparaison. La seconde implémentation compresse les valeurs lues et effectue un décalage sériel vers la sortie de la signature résultante. Pour ce dernier cas, le test se compose de trois phases distinctes : RAZ, Exécution et Extraction de la signature. Lorsque le comparateur est utilisé, le test n'est composé que des phases RAZ et Exécution, puisque l'extraction est réalisée en ligne aussitôt qu'une location mémoire fautive est détectée.

1.5.2.2 Le BIST Transparent

Le BIST transparent évite une sauvegarde ou la perte des données contenues dans la mémoire qu'on veut tester. L'idée est de tester la mémoire avec son contenu et de le restituer une fois la session de test achevée.
Le fait de resituer automatiquement le contenu de la RAM une fois la session de test achevée, fait du BIST Transparent un candidat idéal pour les tests périodiques durant la durée de vie de la RAM.
Le BIST transparent est basé sur un algorithme de test (transparent) qui permet d'avoir la même couverture de fautes que celle de l'algorithme March original avec en plus la possibilité de couvrir des fautes non modélisées. Pour dériver un algorithme transparent à partir d'un algorithme March donné, une procédure décrite par Mihail Nicolaidis [NIC96] a été utilisée. L'architecture adoptée pour le BIST transparent utilise le même concept de décomposition

hiérarchique que celle utilisée pour le BIST March. Son coût en surface est cependant, un peu plus élevé que celui du BIST March.
Cette surface additionnelle est due aux séquences de prédiction, nécessaires pour prédire la signature de l'algorithme transparent. En effet, puisque le contenu initial de la mémoire est inconnu, le BIST transparent effectue une série de lecture, qui est l'image identique de celle qui sera effectuée par l'algorithme transparent, afin de calculer la signature de référence. Cette signature sera ensuite comparée avec la signature finale pour fixer le statut du test (positif/négatif). Les séquences de prédiction impliquent une séquence additionnelle dans le protocole du BIST transparent pour extraire la signature de prédiction. D'autre part, les données lues durant ces séquences doivent être inversées dans certains cas afin de produire les mêmes données lues lors de l'exécution de l'algorithme transparent. Ces exigences impliquent en retour des modifications sur le Générateur de Séquences ainsi que sur le Générateur d'Opérations. En plus de ces modifications, le BIST transparent nécessite un générateur de données spécifique pour dériver les données de test à partir des valeurs existantes dans la mémoire.

1.5.2.3 Le BIST Programmable

Les deux types de BIST présentés précédemment sont des BIST dédiés à un algorithme de test donné. Le BIST programmable constitue une approche alternative qui permet de changer à volonté l'algorithme de test March au niveau de l'implémentation physique, c'est-à-dire même après la fabrication du circuit. Par conséquent, le BIST programmable est le candidat idéal pour les tests de débuggage afin de déterminer les principaux types de fautes relatives à une nouvelle conception de mémoire et à un nouveau processus de fabrication. Le BIST programmable peut être aussi utilisé pour les analyses de défauts lors des retours de la part des clients, afin de diagnostiquer les défauts du circuit qui n'ont pas été détectés par le test de fabrication ou qui apparaissent pendant sa durée de vie.
Une solution originale de BIST programmable, appelé SEMBIST (SEquence based Memory BIST), a été conçue et implémentée [BOU02(a)], [BOU02(b)]. SEMBIST est capable de changer l'algorithme de test implémenté afin de cibler de nouveaux modèles de fautes.
L'architecture SEMBIST est donnée par la Figure 1.12. Mis à part les modules de génération d'adresses et de vérification de la signature, communs aux autres types de BIST, celle-ci est composée de plusieurs modules :

- L'unité de stockage (registre) stocke une séquence entière à exécuter, puisque l'algorithme de test est chargé séquence par séquence dans le BIST. La séquence de test est d'abord chargée d'une manière sérielle dans le registre de scan, avant d'être stockée ensuite dans le registre d'instructions en effectuant un chargement parallèle. Lorsque SEMBIST exécute les opérations contenues dans le registre d'instructions, le contrôleur active une autre opération de scan pour charger une nouvelle séquence de test dans le registre de scan. Une fois que l'exécution de la première séquence est terminée, la seconde séquence est déjà valable pour être exécutée, et peut être chargée dans le registre d'instructions sans perdre aucun cycle d'horloge.

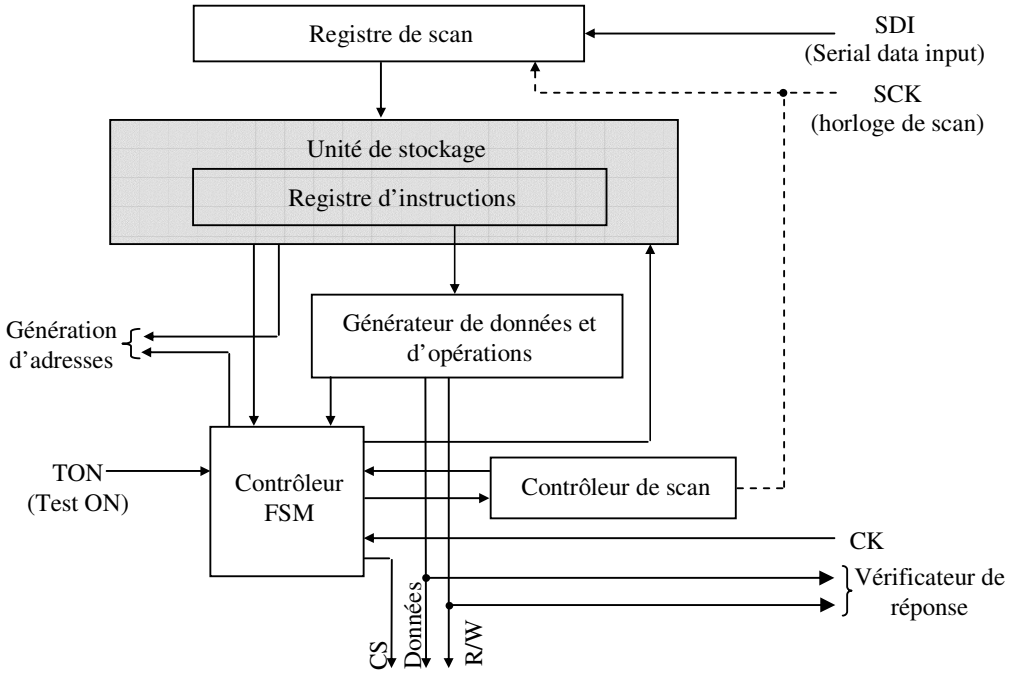


Figure 1.12 L'architecture SEMBIST

- Le générateur d'opérations et de données, est une sorte de séquenceur qui itère à travers les champs des opérations (contenus dans le code instruction de SEMBIST) pour dériver l'opération et la donnée qui correspondent au cycle courant.
- Le contrôleur de scan est un compteur utilisé pour contourner un testeur externe coûteux.
- Le contrôleur FSM est le cœur de SEMBIST. Il a en charge le contrôle et la synchronisation de flux de données interne.

SEMBIST ne présente pas un coût excessif en surface, grâce notamment à la décomposition structurelle de l'algorithme de test. D'autre part, SEMBIST est autonome, et ne nécessite que deux signaux externes (SDI, TON). Tous les autres signaux sont générés localement.

1.6 Le Diagnostic et Réparation Intégré

Le test présenté jusqu'à maintenant s'occupe essentiellement de la détection des fautes. Afin d'augmenter le rendement des RAM de grandes densités, les techniques telles que le Built-In Self-Diagnosis (BISD) et le Built-In Self-Repair (BISR), ont été explorées. Elles sont basées sur la détection, la localisation et la réparation des fautes.

Le diagnostic a pour objectif de relier le comportement erroné observé en sortie d'une unité sous test avec la faute physique correspondante. Le degré de précision dans la localisation de fautes est appelé "la résolution du diagnostic". Plus cette résolution est fine, plus la localisation est précise plus le diagnostic est meilleur.

Le diagnostic de fautes peut être effectué suivant deux approches différentes :

- Une approche "cause effet" qui analyse une cause (faute) possible et détermine les effets (réponses) correspondants. Cette approche nécessite une simulation préalable intensive pour déterminer le dictionnaire de fautes qui est une base de données stockant tous les couples faute-réponse explorés.
- Une approche "effet cause" dans laquelle l'effet (réponse actuelle de l'unité sous test) est traité pour déterminer les causes (fautes) possibles. Dans le cas d'une logique combinatoire, cette approche peut utiliser un algorithme de déduction tel que celui proposé dans [ABR94] pour localiser automatiquement la faute.

Dans le cas des mémoires, l'approche cause effet n'est pas utilisée vu qu'elle nécessite un dictionnaire de fautes de très grande taille. Enfin, dans une logique BISD il convient d'effectuer un bon compromis entre la résolution du diagnostic (complexité de l'algorithme de détection), le temps nécessaire à son exécution et la surface supplémentaire qu'il induit.

Concernant la logique BISR, elle a pour but de traiter les informations venant d'une logique BIST et BISD, pour déterminer les ressources défaillantes à remplacer, et effectuer la reconfiguration adéquate de la mémoire. La réparation intégrée permet donc de maintenir des rendements de production élevés, pour les mémoires embarquées. Nous discuterons en détail les techniques BISR dans les chapitres suivants.

Conclusion

Nous avons passé en revue les principaux modèles de fautes fonctionnels ainsi que les algorithmes les plus performants assurant leur détection. Quelques exemples d'implantation BIST de ces algorithmes ont également été présentés. Les circuits BIST induisent un coût additionnel en surface mais réduisent considérablement le temps de test et éliminent le besoin de stocker et amener des vecteurs de l'extérieur. Un autre avantage est qu'on applique les vecteurs de test à la fréquence nominale du circuit.

Dans les chapitres suivants (Chapitre III, IV, et V), l'ensemble des techniques et algorithmes de réparation qu'on a développé est basé sur l'approche BIST.

CHAPITRE II. Algorithmes et Architectures de Réparation Mémoire

Introduction

L'action de réparation se fait traditionnellement en remplacement les parties défaillantes de la mémoire par des parties redondantes (parties de rechange) correctes. Avant de procéder à tout remplacement, il est nécessaire d'effectuer un test (chapitre I) dans le but de détecter l'ensemble des fautes.

Il existe deux grandes techniques pour la réparation d'une mémoire : la réparation externe à l'aide de la reconfiguration laser [SHI83], [OOK93], [JAY02], et l'auto réparation intégrée [SAW89], [MAZ90], [KIM98].

La technique de la reconfiguration à l'aide du laser se compose généralement de deux phases : (a) une phase logicielle qui détermine les ressources redondantes nécessaires au remplacement des parties défaillantes de la matrice de mémoire, de manière à ce que toutes les fautes soient réparées, (b) une deuxième phase purement matérielle et dans laquelle on déconnecte les parties défaillantes de la mémoire et on connecte les ressources redondantes au bus de données à l'aide de fusibles ou anti-fusibles brûlés par un laser, suivant les résultats de la première phase.

La technique de réparation intégrée, privilégiée dans le cas des mémoires embarquées, est également composée des deux phases : (a) la détermination de la meilleure solution de réparation et (b) de la phase de reconfiguration proprement dite (connexion des unités redondantes au bus de données). Cependant, elles doivent être complètement intégrées avec la mémoire dans le design. Cela signifie que tout le processus de réparation doit être un processus matériel et sans programmation externe par laser.

Dans la première partie de ce chapitre nous présentons les principaux algorithmes de réparation utilisés dans le cadre de la réparation externe ainsi qu'une analyse de leur efficacité. Ensuite, nous présenterons quelques architectures d'auto réparation intégrée que nous analyserons en terme de temps de réparation, de coût en surface et d'efficacité de réparation.

2.1 Algorithmes de Réparation des Mémoires

Le problème de la réparation d'une mémoire se formule souvent comme un problème de réparation d'une matrice de cellules avec un jeu de lignes et de colonnes redondantes. Cependant, il a été démontré que ce problème est de type NP complet [KUO87]. Le but de la plupart des recherches a donc été de réduire le *coût temporel* nécessaire au calcul de la solution de réparation.

Un algorithme de réparation idéal est donc un algorithme qui trouve la solution de réparation (si celle-ci existe) en utilisant le minimum de ressources redondantes en un temps raisonnable (polynomiale) [DAY85], [KUO87]. Pour atteindre cet objectif, plusieurs approches ont été explorées et présentées dans la littérature. La différence entre elles, est la manière dont on modélise le problème de la réparation (arbre, graphe, réseaux de neurones …).

Notons enfin que les algorithmes que nous présentons dans cette section sont surtout utilisés dans le cadre de la réparation externe (par laser). En fait, la reconfiguration depuis l'extérieur, à l'aide du laser, est faite suivant le résultat calculé par l'exécution de ce type d'algorithmes.

Une étude de ces principaux algorithmes est exposée ci-dessous.

2.1.1 Algorithme de Tarr, 1984

Tarr [TAR84] a proposé deux approches différentes pour le remplacement des unités défaillantes. Il s'agit de l'approche "*broadside*" et de l'approche "*repair-most*".

2.1.1.1 L'approche "broadside"

C'est une approche naïve car elle consiste à remplacer chaque faute localisée par une colonne ou une ligne redondante disponible. Cela continue jusqu'à ce que toutes les fautes soient réparées ou bien jusqu'à l'épuisement des ressources redondantes. Cet algorithme utilise de manière inefficace les ressources redondantes. Dans de nombreux cas de distribution des fautes, il peut échouer dans la réparation d'une mémoire théoriquement réparable.

2.1.1.2 L'approche "repair-most"

Plus efficace que l'approche *broadside*, l'approche *repair-most* essaie d'optimiser le nombre de colonnes/lignes à utiliser pour la réparation. Le principe est de remplacer les colonnes et les lignes qui contiennent le plus grand nombre de fautes. Ensuite répéter cette

étape jusqu'à épuisement de la totalité des ressources redondantes ou jusqu'à la réparation de toutes les fautes. Si toutes les ressources redondantes sont utilisées et que des fautes demeurent, alors la mémoire ne peut être réparée.

Cette dernière approche possède deux désavantages. Le premier est qu'elle ne garantit pas l'obtention d'une solution de réparation (Figure 2.1(a)) même si celle-ci existe en théorie (contre exemple dans la Figure 2.1(b)). Le deuxième désavantage est que même dans le cas ou une solution de réparation est trouvée, celle-ci peut ne pas être optimale dans le sens ou elle va utiliser plus de redondance qu'il n'en faut. Par exemple dans la Figure 2.1(c), si l'on dispose de quatre lignes redondantes et d'une colonne redondante, alors C_4 est choisie en premier pour être remplacée et suivie par R_1, R_4, R_3 et R_7. Mais la solution optimale pour ce problème est R_1, R_3, R_4 et R_7 avec une colonne redondante restante comme le montre la Figure 2.1(d).

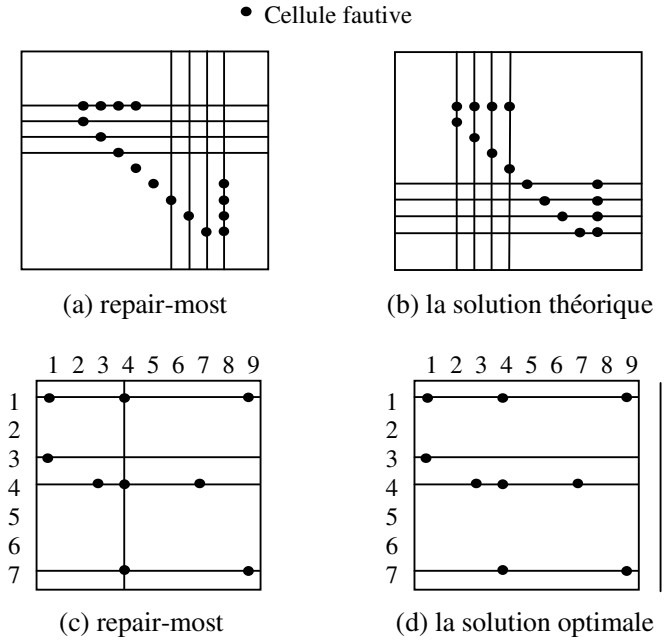


Figure 2.1. Limites de l'approche repair-most.

2.1.2 Algorithme de Day, 1985

L'algorithme de Day [DAY85] est composé de deux phases, la première appelée *forced-repair analysis* durant laquelle on détermine les lignes et les colonnes régulières qui doivent être remplacées par les lignes et les colonnes redondantes. Cette étape permet d'éliminer la majorité si ce n'est la totalité des fautes affectant la matrice mémoire. La seconde phase de l'algorithme appelée *sparse-repair analysis* a pour objectif la réparation du

reste des fautes non réparées durant la première phase en utilisant les ressources redondantes restantes. Les deux étapes de l'algorithme sont décrites ci-dessous.

a) La phase "Forced-repair"

Cette première phase de l'algorithme est basée sur la simple observation qu'une bonne stratégie de réparation consiste à remplacer les colonnes (lignes) affectées par un nombre de fautes supérieures au nombre de lignes (colonnes) redondantes.

Pour cette phase, des informations telles que le nombre de fautes par ligne et par colonne sont nécessaires. Il faut donc deux compteurs indiquant le nombre de lignes et de colonnes redondantes non encore utilisées, LRU et CRU. L'algorithme est décrit ci-dessous :

(1) Chaque ligne (colonne) possède un compteur initialisé à une valeur égale au nombre de fautes affectant cette ligne (colonne). Les deux compteurs LRU et CRU sont initialisés au nombre total de lignes et de colonnes redondantes.

(2) Remplacer chaque ligne affectée par un nombre de fautes supérieur à la valeur du CRU par une ligne redondante et décrémenter le contenu du LRU de 1 après chacun de ces remplacements. Si la valeur du LRU devient nulle et qu'il existe toujours des lignes avec un nombre de fautes supérieur à la valeur du CRU, alors la mémoire est déclarée non réparable.

(3) Remplacer chaque colonne affectée par un nombre de fautes supérieur à la valeur du LRU par une colonne redondante, et décrémenter le contenu du CRU de 1 après chacun de ces remplacements. Si la valeur du CRU devient nulle et qu'il existe toujours des colonnes avec un nombre de fautes supérieur à la valeur du LRU, alors la mémoire est déclarée non réparable.

Le temps d'exécution de cet algorithme est linéaire et dépend du nombre de lignes l et de colonnes c de la matrice mémoire ($O(l+c)$).

b) La phase "Sparse-repair"

La phase *sparse-repair* est invoquée seulement si des fautes demeurent non réparées après la phase *forced-repair*. La réparation de ces fautes est réalisée à l'aide des lignes et/ou colonnes redondantes non utilisées lors de la phase *forced-reair*. Les différentes étapes de l'algorithme sont décrites ci-dessous :

(1) Commencer avec la liste de solutions (structure de donnée) générée par la phase *forced-repair*. Une liste de solutions est composée de deux listes. Une première liste

contenant les numéros des colonnes réparées et une seconde liste qui contient les numéros des lignes réparées.

(2) Scanner une nouvelle faute. L'adresse ligne et l'adresse colonne de cette faute sont ajoutées séparément à la liste de solutions trouvée lors de la phase *forced-repair*, pour générer deux listes de solutions qui correspondent soit à une réparation ligne soit à une réparation colonne. Mais avant de générer une quelconque nouvelle liste, le programme contrôle si l'adresse (ligne ou colonne) à ajouter existe déjà dans la liste de solutions originale. Si c'est le cas l'état de la liste reste inchangé. Le programme vérifie également s'il reste des ressources redondantes disponibles. Si par exemple les lignes redondantes sont épuisées, l'adresse ligne de la faute en cours n'est pas prise en compte. Le même raisonnement est valable pour les colonnes. Une fois que les deux nouvelles listes sont générées, une routine compare ces deux listes pour éliminer parmi elles, celle qui utilise inutilement une ressource redondante.

(3) Répéter l'étape (2) tant que les deux conditions suivantes ne sont pas vérifiées :

   (a) Toutes les lignes et colonnes redondantes sont épuisées et des fautes demeurent non réparées, auquel cas la réparation se termine par un échec.

   (b) Toutes les fautes ont été réparées et toutes les listes de solutions ont été générées sans dépassement des ressources redondantes disponibles. Les listes de solutions ainsi trouvées garantissent la réparation de la mémoire, cependant il appartiendra à l'utilisateur de choisir une de ces solutions.

Nous illustrons dans la Figure 2.2 un exemple d'application de l'algorithme de Day sur une mémoire fautive (Figure 2.2(a)) de taille 10×10, utilisant trois colonnes et trois lignes redondantes. Le résultat de la phase *forced-repair* est montré dans la Figure2.2(b). La phase *sparse-repair* prend ensuite le relais et finit par trouver trois listes de solutions (F, J et H), en cinq étapes (Figure 2.2(c)).

Discussion de l'algorithme

L'algorithme de Day effectue une recherche exhaustive dans l'espace des solutions de réparation, pour une mémoire utilisant un nombre donné de lignes et de colonnes redondantes. Cela permet de trouver toutes les solutions de réparation possibles. Dans la phase *sparse-repair* de cet algorithme, la complexité est exponentielle avec le nombre total de lignes et de colonnes. Afin d'accélérer l'exécution de l'algorithme, Day proposa un test permettant l'arrêt

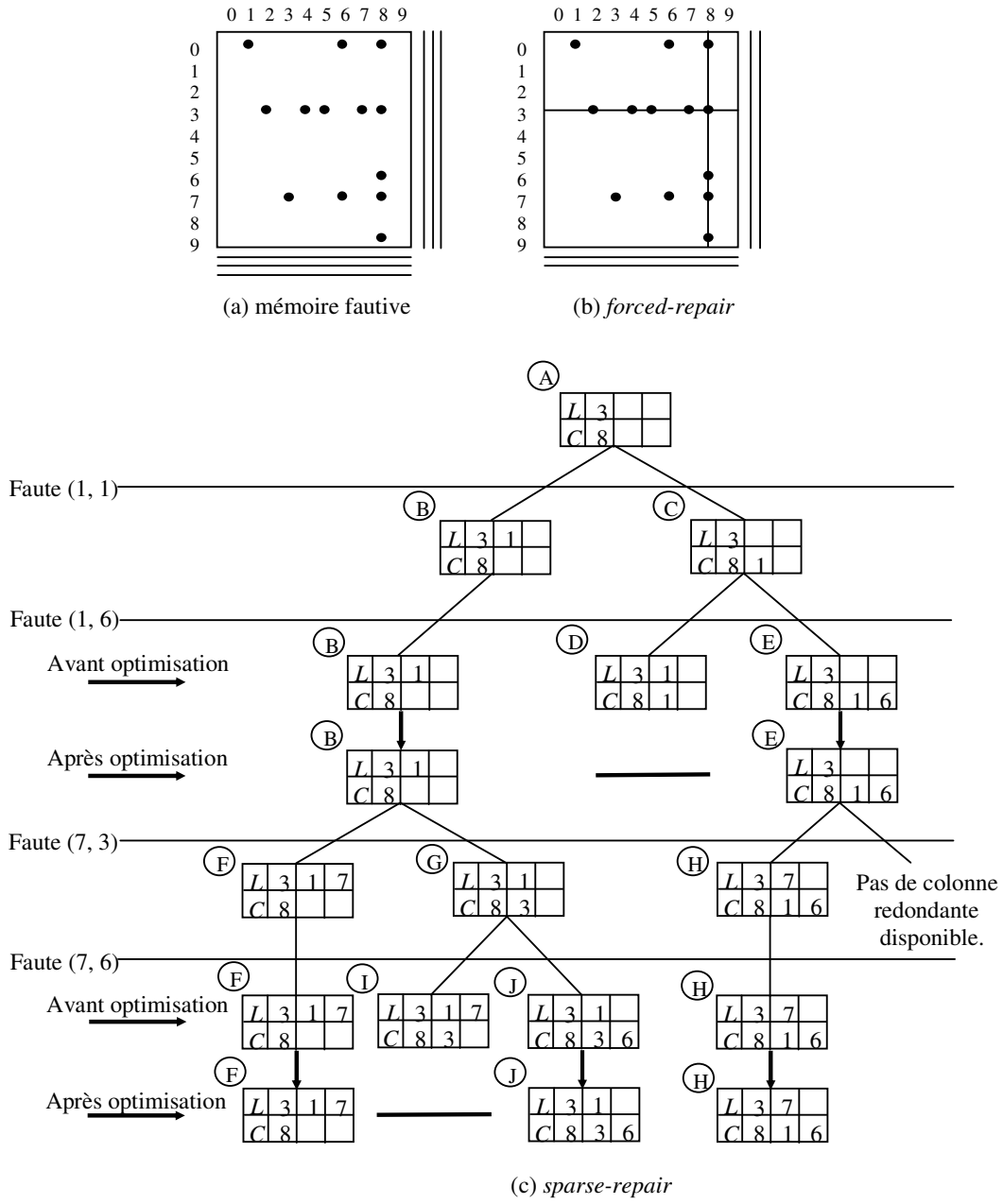


Figure 2.2. Exemple d'application de l'algorithme de Day.

de l'exécution en cas de mémoire non repérable. Si *LN* est le nombre de lignes normales (ou régulières), *CN* le nombre de colonnes normales, *LR* le nombre de lignes redondantes et *CR* le nombre de colonnes redondantes, alors le nombre maximum de fautes pouvant être réparées est (Figure 2.3) :

$$maxF = LN.LR + CN.CR - LR.CR \ldots \textbf{(2.1)}.$$

Par conséquent, une mémoire fautive contenant un nombre de fautes strictement supérieur à *maxF* est automatiquement déclarée non réparable. De même, un test d'arrêt peut être utilisé à chacune des étapes de la phase *sparse-repair*. Si *LRU* et *CRU* désignent respectivement les lignes et colonnes redondantes non encore utilisées après la phase *forced-repair* avec $LRU \leq LR$ et $CRU \leq CR$, alors toute matrice mémoire contenant un nombre de fautes, par région, supérieure à $maxF = 2.LRU.CRU$ (Figure 2.4) est automatiquement déclarée non réparable.

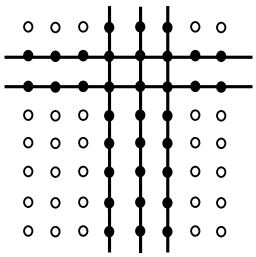


Figure 2.3. Nombre maximal de fautes couvertes par 2 lignes et 3 colonnes redondantes dans une matrice 8×8.

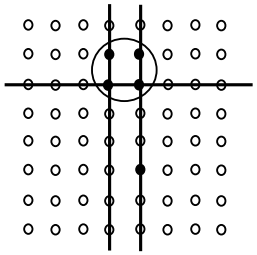


Figure 2.4. Région de 4 fautes.

Etant donné que l'algorithme de Day effectue une recherche exhaustive en générant l'arbre entier de toutes les solutions possibles, son application pour les grandes mémoires affectées d'un grand nombre de fautes, devient inefficace. Prenons pour l'exemple une matrice mémoire de 1000×1000 contenant 100 éléments fautifs. Supposons qu'à la fin de la phase *forced-repair*, il reste 20 éléments fautifs. Par conséquent 2^{20} nœuds de l'arbre doivent être examinés pendant la phase *sparse-repair*. Pour réduire ce temps, Day utilise l'expression (2.1) pour calculer le nombre maximum de cellules fautives possibles pouvant être réparées à l'aide de *LR* lignes redondantes et *CR* colonnes redondantes. Cette expression définit une limite trop grande. Par exemple, si nous reprenons la mémoire de 1 Mbit (1000×1000) avec $LR = CR = 10$, alors le nombre maximum de fautes pouvant être réparées est de $10\times1000 + 10\times1000 - 10\times10 = 19900$. Des valeurs aussi grandes que celles-ci, représentent un nombre de fautes irréaliste pour les technologies actuelles. Par conséquent, ces valeurs ne peuvent être utilisées comme critère d'arrêt de l'algorithme de réparation. Néanmoins, si nous supposons que de telles valeurs sont possibles (nanotechnologies), leurs traitements par la méthode *sparse-repair* nécessitera un temps de calcul exorbitent.

2.1.3 Algorithme de Kuo et Fuchs, 1987

Kuo et Fuchs [KUO87] ont proposé deux algorithmes plus efficaces que celui décrit par [DAY85]. Le premier appelé *branch-and-bound*, utilise une modélisation de type graphe du problème de réparation. Il exploite en particulier quelques propriétés des graphes bipartites pour effectuer des tests d'arrêt rigoureux. Dans la dernière phase de cet algorithme, une analyse de l'arbre des solutions similaire à celle de [DAY85] est utilisée. Cependant, le coût (en temps) de programmation (laser) des ressources redondantes est pris en compte, ce qui permet de réduire considérablement le nombre de nœuds à examiner dans l'arbre.
Le deuxième algorithme uniquement basé sur la modélisation graphe, utilise des heuristiques d'approximation pour trouver une solution de réparation dans le cas des grandes mémoires affectées d'un grand nombre de fautes.
Nous allons commencer par décrire la modélisation de type graphe du problème de la réparation mémoire, puis présenter les deux algorithmes de Kuo et Fuchs [KUO87] utilisant cette modélisation.

Modélisation de type graphe du problème de réparation d'une matrice mémoire

Soit une matrice mémoire rectangulaire ayant $M \times N$ cellule, LR lignes redondantes et CR colonnes redondantes. Considérons un graphe $G = (V, E)$ avec deux ensembles de sommets, le premier appelé L et le second C ($V = L \cup C$). Chaque sommet dans L représente une ligne mémoire qui contient au moins une cellule fautive, et chaque sommet dans C représente une colonne qui contient au moins une cellule fautive. Chaque arrête dans ce graphe a une extrémité dans L et l'autre dans C faisant de ce graphe un graphe bipartite. Une arrête (l, c) qui va d'un sommet l dans L vers un sommet c dans C représente l'existence d'une faute dans une cellule positionnée à la ligne numéro l et à la colonne numéro c. La Figure 2.5 illustre une modélisation de type graphe d'une mémoire contenant cinq fautes, et utilisant trois colonnes redondantes et deux lignes redondantes.
La recherche d'une solution de réparation de la matrice mémoire est alors ramenée à la couverture du graphe bipartite G : autrement dit, trouver le minimum de nœuds L's et C's qui couvrent tout les liens du graphe G. Un nœud couvre un lien si le lien est connecté au nœud.
Si nous considérons l'exemple de la Figure 2.5, on constate que les liens (4,1), (4,4), et (4,9) peuvent être couverts par L_4. Le lien (6,4) peut être couvert soit par L_6 ou par C_4. Le lien (1,1) peut être couvert soit par L_1 ou par C_1.

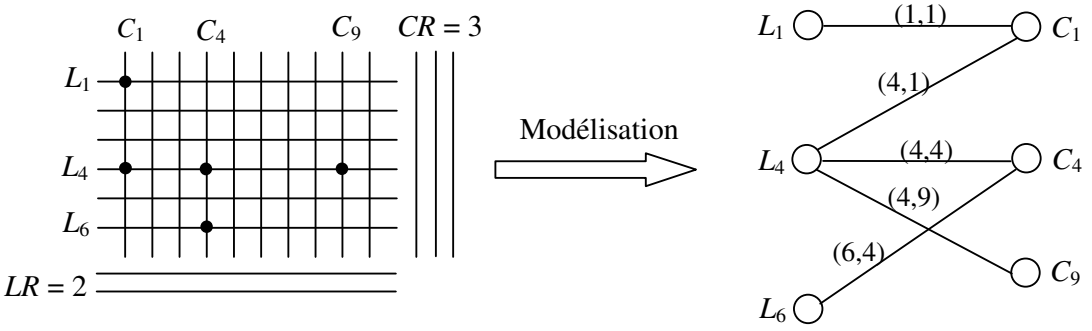


Figure 2.5. Modélisation du problème de la réparation mémoire à l'aide d'un graphe bipartite.

On en déduit les solutions de réparation suivantes :

$\{L_4, L_6, L_1\}$, $\{L_4, L_6, C_1\}$, $\{L_4, C_4, L_1\}$, $\{L_4, C_4, C_1\}$.

Aussi, le nombre de *L*'s à choisir est limité à *LR* et le nombre de *R*'s est limité à *CR*. Le problème de reconfiguration est donc la recherche d'une couverture pour un graphe bipartite avec contraintes. Si nous tenons compte de cette nouvelle exigence dans l'exemple de la Figure 2.5, nous devons alors éliminer la solution de type $\{L_4, L_6, L_1\}$ puisque $LR = 2$.

2.1.3.1 L'approche branch-and-bound

Ce premier algorithme implémenté par Kuo et Fuchs utilise un test d'arrêt basé sur la modélisation graphe du problème de la réparation. Il utilise aussi la contrainte en termes de temps de programmation des ressources redondantes pour trouver rapidement la solution optimale. L'algorithme est constitué de trois phases :

a) La phase "Must-repair"

Cette étape est similaire à la procédure *forced-repair* proposée par [DAY85], et dans laquelle on remplace les colonnes (lignes) fautives affectées par un nombre de fautes supérieures au nombre de lignes (colonnes) redondantes.

b) La phase "Initial screening"

Dans cette phase de l'algorithme, Kuo et Fuchs exploitent une propriété importante des graphes bipartites pour calculer le nombre minimum de lignes et de colonnes redondantes nécessaires à la réparation de la mémoire.

Dans un graphe bipartite, une couverture minimum des nœuds est le nombre minimum de sommets qui couvre toutes les arrêtes du graphe. Le nombre de nœuds dans la couverture minimum des nœuds, est exactement le nombre minimum de lignes et de colonnes redondantes nécessaires à la réparation de la mémoire. D'autre part, ce nombre minimum de nœuds qui couvre toutes les arrêtes est égal au nombre d'arrêtes se trouvant dans n'importe quel couplage maximum du graphe. Un couplage M du graphe $G = (V, E)$ est un sous-ensemble d'arrêtes avec la propriété suivant laquelle il n'existe pas deux arrêtes de M qui partagent le même nœud. M est un couplage maximum si G n'a pas de couplage M' tel que $|M'| > |M|$.

Le temps d'exécution de ce test est polynomiale [HOP73] donnant lieu à un résultat rapide.

c) La phase "Final analysis"

Dans cette étape, un arbre de solutions est généré de la même manière que dans l'algorithme de Day [DAY85]. Pour éviter de traiter exhaustivement tous les nœuds de l'arbre des solutions, Kuo et Fuchs ont introduit une fonction F qui permet de calculer le coût de chacune des solutions partielles. Le nombre de nœuds (solutions) traités peut être ainsi considérablement réduit en tenant compte, à chaque niveau de l'arbre, uniquement de la solution partielle impliquant la plus petite valeur de F.

La fonction coût est donnée par Kuo et Fuchs sous la forme $F = c_L L + c_C C$, ou c_L et c_C représentent respectivement le temps nécessaire à la programmation laser d'une ligne redondante et d'une colonne redondante ; L ($\leq LR$) et C ($\leq CR$) représentent respectivement le nombre total de lignes et de colonnes redondantes déjà impliquées à un certain niveau de l'arbre.

L'algorithme est alors exprimé de la manière suivante :

(1) Calculer la fonction coût de la liste solution générée par la phase *must-repair*.

(2) Générer une succession de listes solution comme dans l'algorithme de Day [DAY85] mais seulement pour le plus petit coût.

(3) Terminer sous l'une des deux conditions ci-dessous, sinon aller à (2) :

(a) Si toutes les lignes et/ou colonnes redondantes sont épuisées et qu'une solution de réparation n'a pas été trouvée. Dans ce cas la mémoire est déclarée non réparable.

(b) La recherche finit par la génération d'une liste de solutions optimale.

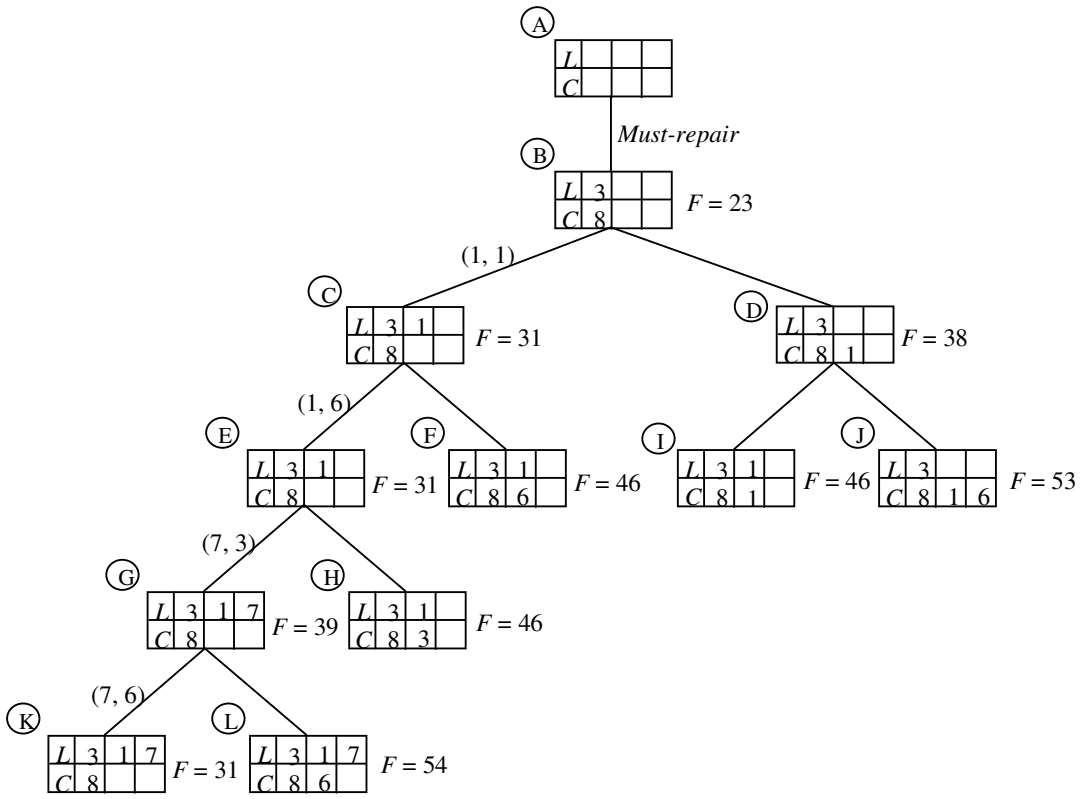


Figure 2.6. Exemple d'application de l'algorithme de Kuo et Fuchs ($c_L = 8$, $c_C = 15$).

La Figure 2.6 illustre un exemple d'application de cet algorithme. Nous avons sélectionné pour cet exemple la mémoire fautive de la Figure 2.2(a), cela nous permettra de comparer les performances des deux algorithmes. La principale différence entre les deux algorithmes, est le nombre de nœuds traités. Dans le présent algorithme on traite 11 nœuds contre 18 dans l'approche de Day [DAY85]. Cette différence est d'autant plus grande quand le problème devient plus grand (en termes de taille de la mémoire et nombre de fautes).

2.1.3.2 L'approche heuristique du branch-and-bound

Bien que l'algorithme *branch-and-bound* soit plus efficace que l'algorithme de Day [DAY85], sa complexité reste exponentielle et ne peut finalement être pratique que pour les mémoires touchées par un nombre modéré de fautes. Afin de traiter des problèmes de plus grande taille, Kuo et Fuchs [KUO87] proposèrent d'utiliser une heuristique d'approximation pour guider le processus de recherche d'une solution de réparation. Bien que l'approche heuristique ne garantisse pas la génération d'une solution optimale et même dans certains cas

de trouver une solution existante, son temps d'exécution est une fonction polynomiale de la taille du problème ce qui donne à cette approche toute son efficacité.
L'algorithme commence d'abord par une phase de *must-repair*. S'en suit alors une phase (*Final analysis*) qui comprend l'approche heuristique proprement dite. Celle-ci est basée sur une méthode de recherche de couplages pour les graphes généraux [CLA83].

L'algorithme est décrit de la manière suivante :

(1) Sélectionner une ligne ou une colonne avec une seule cellule fautive et dont la colonne ou la ligne correspond au coût le plus bas. Si une telle ligne ou colonne existe, lui allouer une ligne ou une colonne redondante. Sinon, sélectionner une ligne ou une colonne avec le minimum de la valeur coût/(nombre de cellules fautives) et la remplacer par une ligne ou par une colonne redondante (si ces dernières existent).
(2) Ajuster le coût courant en lui soustrayant la valeur du (coût courant)/(nombre de cellules fautives couvertes par la ligne ou la colonne redondante qui vient d'être utilisée). En terme de graphe bipartite, il s'agit de supprimer un sommet représentant une ligne (colonne) fautive ainsi que toutes les arrêtes incidentes sur ce sommet. Ces arrêtes représentent toutes les cellules fautives existantes sur cette ligne (colonne). Mettre à jour la liste solution en prenant en compte la ligne (colonne) qui vient juste d'être réparée.
(3) Répéter ces procédures jusqu'à ce qu'il ne reste aucune faute ou ressource redondante.

En réutilisant l'exemple de la mémoire fautive de la Figure 2.2(a), la ligne 3 et la colonne 8 sont programmées pour être remplacées suite à l'analyse *must-repair* comme le montre la Figure 2.2(b) ou la Figure 2.6. Après cela, la colonne 1 a seulement une seule cellule fautive (1,1). Puisque le coût de la ligne 1 est le plus petit, alors une ligne redondante lui est allouée. Chacune des colonnes 3 et 6 a une seule faute, et les deux fautes se trouvent sur la ligne 7. Par conséquent, la ligne 7 est programmée pour un remplacement.

La complexité dans le temps de cet algorithme est $O[(LR + CR).|V|]$, où $|V| = |L| + |C|$ représente le nombre total de lignes et colonnes fautives. Cette approche ne garantit pas toujours une solution optimale, néanmoins une solution optimale est trouvée dans la majorité des tests effectués par Kuo et Fuchs [KUO87]. Le temps d'exécution de cet algorithme est considérablement inférieur à celui de l'algorithme *branch-and-bound* lorsqu'on s'intéresse aux grandes mémoires.

2.1.4 Algorithme de Wey et Lombardi, 1987

Wey et Lombardi [WEY87] considèrent que le surcoût temporel induit par le processus de la réparation mémoire est principalement dû au temps nécessaire à la recherche de la solution de réparation (traitement des nœuds dans l'arbre des solutions, recherche d'une couverture minimale pour les graphes bipartites). Les deux auteurs se sont alors concentrés sur le développement de nouvelles techniques qui leur permette de détecter le plus tôt possible, compte tenu des fautes détectées, si la mémoire peut être réparée ou non. Nous avons déjà discuté certaines de ces techniques dans les sections précédentes comme étant des tests d'arrêt du processus de réparation.

Les auteurs commencent d'abord par construire un "ensemble de réparation" qui contient les coordonnées de toutes les cellules fautives indépendantes. Deux cellules fautives sont dites indépendantes si elles ne partagent pas la même ligne ou la même colonne mémoire.

La construction de l'"ensemble de réparation" se fait de la manière suivante :

Une localisation des cellules mémoire représenté sous la forme d'une matrice $A = (a_{ij})$ de dimension $N \times M$ est utilisé. L'élément a_{ij} sur la i^{eme} ligne et la j^{eme} colonne est à 0 s'il est correct et à 1 s'il est fautif. Afin de simplifier le recensement des cellules fautives indépendantes, la mémoire est parcourue dans un ordre précis. La première faute détectée constitue la première faute indépendante. Les éléments a_{ij} de la ligne et de la colonne auxquelles appartient cette faute, prennent la valeur 0. Aussi, l'adresse de la ligne et l'adresse de la colonne de cette faute indépendante sont immédiatement intégrées dans l'ensemble de réparation. Cette procédure est ainsi répétée jusqu'à ce que tous les éléments a_{ij} deviennent nuls. L'ensemble de paires ordonnées (adresse ligne, adresse colonne) ainsi obtenu forme l'"ensemble de réparation" *S*. Pour la distribution de fautes de la Figure 2.7, l'ensemble de réparation peut être le suivant :

$S = \{(0, 0), (1, 2), (3, 5)\}$.

A partir de l'ensemble *S*, une série de théorèmes démontrés par Wey et Lombardi [WEY87] sont utilisés pour effectuer des tests d'arrêts du processus de réparation.

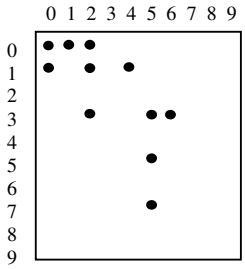


Figure 2.7. Mémoire fautive 3/3 réparable.

Théorème 1

Ce premier théorème stipule que si $s = |S| \leq \max\{LR, CR\}$ alors la mémoire est *LR*/*CR* réparable. Dans l'exemple de la Figure 2.7, $LR = CR = 3$ et $s = \max\{LR, CR\}$, donc la mémoire est bien 3/3 réparable.

De plus, l'ensemble *S* donne une solution approximative pour la réparation, car les lignes et colonnes recensées dans *S* doivent être remplacées. Si nous reprenons alors l'exemple de la Figure 2.7, nous pouvons écrire la solution de réparation comme étant donnée par les ensembles $R_L = \{L_0, L_1, L_3\}$ et $R_C = \{C_0, C_2, C_5\}$, dans lesquels R_i et C_j identifient la i^{eme} ligne et et la j^{eme} colonne à remplacer.

Théorème 2

Ce deuxième théorème introduit un second test d'arrêt. Il stipule que si $s = |S| > LR + CR$, alors la mémoire est *LR*/*CR* non réparable.

L'avantage des deux théorèmes précédents, est qu'ils sont indépendants de la distribution des fautes mais sont uniquement basés sur la cardinalité de *S*. Ils sont utilisés pour établir une région dans laquelle la mémoire est réparable ($s \leq \max\{LR, CR\}$) et une région dans laquelle la mémoire est non réparable ($s > LR + CR$). Un problème de décision demeure pour la région intermédiaire, appelée *région d'incertitude*, dans laquelle $\max\{LR, CR\} < s \leq LR + CR$.

Pour trouver une condition de réparation ou de non réparation dans la région d'incertitude, les auteurs ont défini la notion de *colonne complémentaire* d'un élément de *S*. Une colonne complémentaire d'une faute indépendante $(i, j) \in S$ est constituée de l'ensemble des fautes $a_{ik} = (i, k) \in A$ tel que C_k n'appartient pas à R_C. La *ligne complémentaire* est définie de la même manière. Dans l'exemple de la Figure 2.8., $LR = CR = 3$, $S = \{(1, 2), (3, 0), (4, 4), (5,$

6), (6, 8)}, $R_L = \{1, 3, 4, 5, 6\}$, $R_C = \{0, 2, 4, 6, 8\}$, et $s = 5$. La faute (4, 10) est un élément de la colonne complémentaire de (4, 4), mais (4, 8) n'en est pas un car $C_8 \in R_C$. Aussi, (8, 8) est un élément de la ligne complémentaire de (6, 8).

En se basant sur les notions de ligne et de colonne complémentaire, les auteurs ont énoncé et démontré le théorème suivant.

Théorème 3

Ce troisième théorème stipule que pour $s = |S|$ tel que : $\max\{LR, CR\} < s \leq LR + CR$, si d représente le nombre de fautes indépendantes dans S telles que chacune d'elles ait un élément dans la ligne et dans la colonne complémentaire et que l'ensemble de ces éléments ne partagent ni ligne ni colonne et que $s + d > LR + CR$, alors la mémoire est LR/CR non réparable.

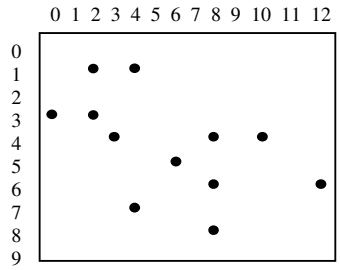


Figure 2.8. Mémoire fautive 3/3 non réparable.

Pour la distribution des fautes de la Figure 2.8, (6, 12) et (8, 8) sont des éléments de la colonne et de la ligne complémentaire de la faute indépendante (6, 8), et $d = 2$; en reprenant le dernier exemple avec $s = 5$, $LR = CR = 3$; on obtient : $s + d > 6 = LR + CR$, ce qui signifie que la mémoire est 3/3 non réparable.

Les deux auteurs ont ensuite étendu leur approche, pour résoudre le problème de la région d'incertitude ($\max\{LR, CR\} < s \leq LR + CR$), à des techniques antérieures de détection qui utilisent les compteurs de fautes (ou *Totalizers*) [DAY85]. Pour se faire, ils utilisent le théorème suivant.

Théorème 4

Soit T le nombre total de fautes dans la mémoire et max $\{LR, CR\} < s \leq LR + CR$; la mémoire est LR/CR réparable si :

$T \leq 2(LR + CR) - s + 1$ pour $LR + CR - s \leq \max\{LR, CR\}$ ou

$T \leq 2 \min\{LR, CR\} + \max\{LR, CR\}$ pour $LR + CR - s > \max\{LR, CR\}$.

Un second théorème qui compte le nombre de fautes a également été utilisé.

Théorème 5

Soit max $\{LR, CR\} < s \leq LR + CR$, $C_F(i)$ le nombre de fautes dans la i^{eme} colonne, $L_F(j)$ le nombre de fautes dans la j^{eme} ligne, $l = |L'| = |\{(i, j) \in S \mid L_F(j) > 1\}|$ et $c = |C'| = |\{(i, j) \in S \mid C_F(i) > 1\}|$; si $l \leq LR$ et $c \leq CR$ alors la mémoire est LR/CR réparable.

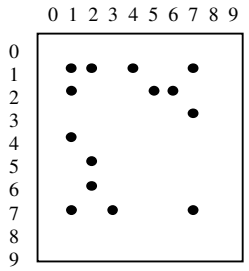


Figure 2.9. Mémoire fautive 3/3 réparable.

Pour illustrer ce dernier théorème considérons la Figure 2.9, dans laquelle $S = \{(1, 1), (2, 5), (3, 7), (5, 2), (7, 3)\}$. Les compteurs ligne indiquent : $L_F(1) = 4$, $L_F(2) = 3$, $L_F(3) = 1$, $L_F(5) = 1$, et $L_F(7) = 3$. Les compteurs colonne indiquent : $C_F(1) = 4$, $C_F(5) = 1$, $C_F(7) = 3$, $C_F(2) = 3$, et $C_F(3) = 1$. Cela implique $l = 3$ et $c = 3$; $l = LR$ et $c = CR$, alors la mémoire est LR/CR réparable ; la solution de réparation est donnée par $L' = \{L_1, L_2, L_7\}$ et $C' = \{C_1, C_2, C_7\}$.

Les résultats de simulation publiés par les auteurs indiquent que les techniques proposées ont une bonne efficacité. D'une part parce qu'elles réduisent considérablement le temps nécessaire à la réparation par l'évaluation des conditions de réparation et de non réparation, et d'autre part parce qu'elles aboutissent à une solution de réparation dans la majorité des cas ou celle-ci existe.

2.1.5 Réparation basée sur les réseaux de neurones

Un système de réseaux de neurones est un ensemble de processeurs simples connectés entre eux. Les interconnexions ont des poids qui leur sont associés tandis que les processeurs appelés aussi nœuds sont prévus avec des entrées. Les poids des interconnexions et les entrées des processeurs peuvent varier pendant la phase de calcul effectuée par le réseau de neurones. Un exemple d'un système de réseau de neurones est montré dans la Figure 2.10.

Pour résoudre le problème de réparation des matrices mémoires, Mazumder et Yih [MAZ90] ont utilisé le modèle de réseau de neurones développé par Hopfield [HOP82] pour le traitement des problèmes d'optimisation combinatoire. Ils proposent également une approche d'implémentation matérielle en vue d'une auto réparation intégrée (basée sur les réseaux de neurones). Cette architecture sera décrite dans la prochaine section. Nous allons nous intéresser ici uniquement à la modélisation et à la résolution du problème de réparation d'une mémoire à l'aide des réseaux de neurones.

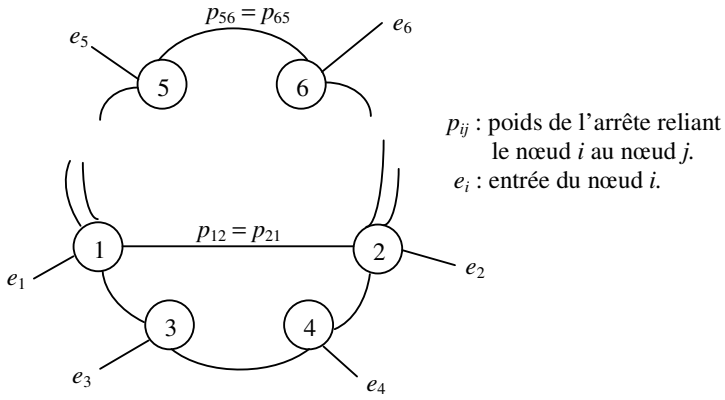


Figure 2.10. Exemple d'un réseau de neurones.

Modélisation en réseau de neurones du problème de la réparation d'une matrice mémoire

Considérons une matrice mémoire de taille $M \times N$ avec p lignes redondantes et q colonnes redondantes. Considérons aussi une configuration de fautes dans laquelle les cellules défectueuses sont aléatoirement distribuées sur m ($< M$) lignes distinctes et n ($< N$) colonnes distinctes de la matrice mémoire, de telle manière que le sous ensemble $m \times n$ représente toutes les lignes et colonnes contenant au moins une faute. Soit la matrice $D = \{d_{ij}\}$, de dimension $m \times n$, la matrice caractéristique de la mémoire telle que d_{ij} correspond à l'état de

la cellule se trouvant à la ligne i et à la colonne j. Si celle si est fautive alors $d_{ij} = 1$ sinon $d_{ij} = 0$.

Une solution ligne consiste en un vecteur U de dimension m tel que $u_i = 0$ si la ligne i est à remplacer sinon $u_i = 1$, $0 \leq i \leq m-1$. Une solution colonne est définie de la même façon, et représentée par le vecteur V à n bits. Le nombre de 0 dans U et V doit être inférieur ou égal à p et q respectivement. La mémoire est dite réparable si les contraintes citées ci-dessus sont respectées, et le problème de la réparation revient essentiellement à savoir comment déterminer U et V tel que $U^T DV = 0$.

En plus de la matrice caractéristique D, considérons l'état des lignes i et des colonnes j à l'aide des variables r_i et k_i respectivement, définies comme suit :

$$r_i = \begin{cases} 1, \text{ si la ligne } i \text{ contient des éléments défectueux} \\ 0, \text{ sinon.} \end{cases}$$

$$k_j = \begin{cases} 1, \text{ si la colonne } j \text{ contient des éléments défectueux} \\ 0, \text{ sinon.} \end{cases}$$

Afin de construire un réseau de neurones pour le problème de réparation, un réseau de dimension $m + n$ est utilisé dans lequel l'état des m premiers neurones est défini par $s_{1i}\,(1 \leq i \leq m)$ et les états des n neurones restant par $s_{2j}\,(1 \leq j \leq n)$. Les premiers m neurones sont appelés les neurones ligne et les n restants sont appelés les neurones colonne. L'état physique de ces neurones est défini comme suit :

$$s_{1i} = \begin{cases} 1, \text{ si la ligne } i \text{ est suggérée pour un remplacement} \\ 0, \text{ sinon.} \end{cases}$$

$$s_{2j} = \begin{cases} 1, \text{ si la colonne } j \text{ est suggérée pour un remplacement} \\ 0, \text{ sinon.} \end{cases}$$

La fonction coût C^{MR} du problème de la réparation mémoire est la somme algébrique de deux fonctions C_1 et C_2 données ci-dessous :

$$C_1 = A/2\left[\left(\sum_{i=1}^{m} s_{1i}\right) - p\right]^2 + A/2\left[\left(\sum_{j=1}^{n} s_{2j}\right) - q\right]^2 \quad \ldots \textbf{(1.2)}$$

$$C_2 = B/2\left[\sum_{i=1}^{m}\sum_{j=1}^{n} d_{ij}\left(1 - s_{1i}\right)\left(1 - s_{2j}\right)\right] + B/2\left[\sum_{j=1}^{m}\sum_{i=1}^{n} d_{ji}\left(1 - s_{1j}\right)\left(1 - s_{2i}\right)\right] \ldots \textbf{(1.3)}$$

L'expression C_1, montre clairement l'intérêt d'une solution de réparation utilisant exactement p lignes redondantes et q colonnes redondantes. Pour les solutions de réparation non faisables, c'est-à-dire nécessitant plus de p lignes te q colonnes redondantes, le coût augmente d'une manière quadratique. L'expression C_2 décroît quand de plus en plus de cellules fautives sont remplacées par une ligne ou une colonne redondante. Si toutes les cellules fautives sont remplacées alors C_2 devient nulle. La fonction d'énergie E d'un réseau de neurones est donnée par l'expression :

$$E = -1/2 \sum_i \sum_j p_{ij} s_i s_j - \sum_i s_i e_i \ \ldots \ \textbf{(1.4)}$$

En réécrivant la fonction C^{MR} sous la même forme que la fonction d'énergie E, on trouve par identification l'expression des poids ainsi que des entrées du réseau de neurones correspondant au problème de réparation. Il en résulte le réseau de neurones défini par :

$p_{1i,1j} = -A(1 - \delta_{ij})$, $p_{2i,2j} = -A(1 - \delta_{ij})$, $p_{1i,2j} = -B\, d_{ij}$, $p_{2i,1j} = -B\, d_{ij}$, $e_{1i} = A(p - 1/2) + B\sum_j d_{ij}$,

et $e_{2j} = A(q - 1/2) + B\sum_i d_{ij}$ avec $\delta_{ij} = 0$ si $i \neq j$, 1 sinon.

La solution de réparation est déterminée par le calcul des différents états des neurones à des instants successifs tout en respectant les contraintes citées ci-dessus. Lorsque le premier minimum d'énergie est atteint, la recherche s'arrête même si la solution trouvée n'est pas optimale. C'est pour cette raison que les auteurs ont proposé dans le même article [MAZ90] une seconde approche plus efficace (*hill-climbing ability*) et qui permet de rechercher la solution optimale parmi toutes celles qui représentent un minima d'énergie. Bien que deux à trois fois plus lente que l'approche basique, elle présente une meilleure efficacité.

La suite de ce chapitre est consacrée à la revue des principales architectures d'auto réparation qui ont été développées au cours de ces deux dernières décennies, époque considérée comme le début de l'intérêt porté à ces techniques.

2.2 Architectures d'Auto Réparation Intégrée pour les Mémoires

L'auto réparation intégrée (Built-In Self-Repair dans la littérature anglaise, ou BISR) pour mémoires est une technique de réparation complètement autonome et dans laquelle toutes les phases de détection, de localisation et de remplacement sont effectuées à l'intérieur du circuit. Contrairement à la réparation externe (laser), l'auto réparation intégrée, ne se fait

pas par programmation mais se réalise d'une manière autonome au niveau matériel. Pour le cas des mémoires RAM embarquées, l'auto réparation intégrée s'avère intéressante voir nécessaire pour plus d'un cas :

- En remplaçant l'ensemble du circuit de réparation, le BISR permet de se passer d'un équipement de réparation spécial souvent basé sur la technologie laser (coûteux). De plus, la technique laser utilise des composants électriques qui ne font pas partie du standard de la technologie CMOS. Par conséquent, introduire ces composants, implique une augmentation significative du coût de fabrication. Notons également que la nature agressive de cet équipement (par rapport a la technologie CMOS) donne à son tour naissance à des défaillances.
- Il permet également de réduire d'une façon considérable le temps de réparation, en évitant le processus de déconnexion (à l'aide du laser) des ressources fonctionnelles fautives et la connexion des ressources régulières au bus de données. C'est une nécessité pour les RAM dynamiques de grandes capacités à cause du temps nécessaire à leur réparation. Ce problème se pose avec plus d'acuité lorsqu'il s'agit de réparer un grand nombre de RAM d'une même carte, mais aussi lorsque la densité de défauts augmente.
- L'approche BISR offre la possibilité de réparer la mémoire pendant la phase de fabrication pour augmenter le rendement en production, ainsi qu'en phase d'opération pour augmenter la fiabilité du système.
- Le BISR est nécessaire pour les RAM enfouies à cause des problèmes d'observabilité et de contrôlabilité.

Cependant, comme dans le cas du BIST, l'utilisation du BISR a également son prix. Elle conduira inévitablement à un surcoût en matériel et d'une perte de performance. La seule solution possible, est de minimiser au maximum ces pénalités temporelles et surfaciques. Cependant d'autres recommandations peuvent s'ajouter à ces dernières, et que si elles sont respectées, feraient du BISR un circuit de réparation idéal. Nous pouvons résumer ces différentes recommandations comme suit :

- Veiller à ce que le temps nécessaire au processus de réparation soit raisonnable,
- Implémenter au niveau matériel des algorithmes qui manipulent efficacement les ressources redondantes pour réparer l'ensemble des fautes, qu'elles soient distribuées sur les unités régulières et/ou redondantes,

- Minimiser le plus possible le coût en surface et la pénalité temporelle induits par l'utilisation d'un BISR,
- Le circuit de réparation doit être intégré avec la mémoire et non dans la mémoire. Cette recommandation est très importante car elle évite la modification de la structure interne de la mémoire (layout).

Compte tenu des recommandations citées ci-dessus, nous pouvons déjà appréhender la difficulté de conception d'un BISR. En effet, respecter en même temps des points antagonistes tels qu'un coût en surface raisonnable, une efficacité de réparation élevée et en plus sans accès à la structure interne de la mémoire, est une tache complexe.
C'est principalement pour cette raison, que les algorithmes de réparation adoptés dans le cas du BISR ont bon nombre de limitations. D'une façon générale, la tendance actuellement est de diminuer le coût en surface au détriment d'une efficacité de réparation élevée. D'un point de vu pratique, ce choix est justifiée puisque dans les technologies actuelles le nombre de fautes par Mbit n'excède pas la dizaine.

L'implémentation d'un circuit BISR pour les mémoires diffère considérablement selon le type de la redondance utilisée. Trois principaux types de redondance sont généralement considérés : la redondance ligne, la redondance colonne et la redondance combinée ligne/colonne.

2.2.1 La Réparation des Ligne de Mémoire (ou Mots de Mémoire)

Ce premier type de réparation est basé sur le remplacement d'un mot de mémoire, ou plus généralement d'une ligne entière de la mémoire, par des mots ou des lignes redondantes. Une ligne redondante est constituée d'une part de cellules mémoire et d'autre par d'un décodeur d'adresses. Ainsi, la réparation ligne permet de réparer les fautes pouvant affecter [KIM98] :

- Le décodeur d'adresse ligne,
- Les cellules mémoires d'un mot, ou d'un ensemble de mots.

Architecture Matérielle

La première architecture matérielle (1989) pour la réparation ligne a été proposée par Sawada et al. [SAW89]. D'une conception très simple, l'architecture proposée était basée sur

le mécanisme d'une mémoire CAM (Content Adressable Memory) dans laquelle on utilise la technique de la comparaison d'adresse. Sawada a montré dans son article pour la première fois la faisabilité de son BISR sur une mémoire DRAM de 1 Mbit (128K × 8). Il a également noté qu'en utilisant ce type d'architecture, les performances sur l'accès mémoire étaient négligeables car l'accès à la mémoire régulière et à la mémoire redondante se faisait en parallèle. Par contre, l'architecture proposée ne permet la réparation que d'une seule ligne défaillante et ne tient pas compte des fautes dans les parties redondantes.
Souvent, les architectures BISR basées sur le principe de fonctionnement de la mémoire CAM ont en commun les mêmes blocs fonctionnels. Le BISR proposé par Sawada [SAW89] peut nous servir de bon exemple afin de résumer cette technique. Ce circuit BISR se compose de deux unités : une unité d'auto test et une unité d'auto réparation.

- L'unité de self test est comparable à un BIST et se compose d'un générateur d'adresse (par exemple un compteur), un générateur de données (par exemple un LFSR), un block de synchronisation et un comparateur de données. Suivant le type de fautes qu'on doit détecter et réparer, on implémente ces blocs suivant divers algorithmes comme le CHECKER-BOARD, MARCH, GALLOP et bien d'autres.
- L'unité de self réparation est composée d'un registre chargé de mémoriser l'adresse défectueuse, d'un comparateur d'adresse et d'une mémoire redondante. On trouve aussi des multiplexeurs d'adresses pour faire la différence entre la phase de test/réparation et la phase de fonctionnement normal de la mémoire.

Un schéma bloc de ce type d'architecture est donné dans la Figure 2.11. Pendant la phase de test si une erreur est détectée, l'adresse présente à la sortie du générateur d'adresse est chargée dans un registre (fail address register). Pendant la phase de fonctionnement normal de la mémoire, l'adresse système, ou un sous ensemble de cette adresse (pour remplacer plusieurs lignes à la fois) est à chaque fois comparée avec l'adresse fautive mémorisée. S'il y a égalité, alors l'adresse courante est reconnue comme fautive, et l'opération de lecture/écriture est alors effectuée sur la mémoire redondante. Sinon, c'est la mémoire régulière qui est sélectionnée.

Basée sur le même principe, des travaux ont été menés pour améliorer l'architecture de Sawada. C'était la plupart du temps des variantes de l'implémentation de base afin de régler les problèmes classiques d'une telle technique à savoir le temps d'accès mémoire et la prise en compte des fautes multiples [SCH01],[TAN92], [BHA94], [BEN00], [SUN93]. Mais ces

derniers travaux ne considèrent qu'un nombre très réduit de fautes (deux fautes dans [BHA94], [SUN93]), ce qui nous laisse dire que les fautes multiples ainsi que les fautes dans

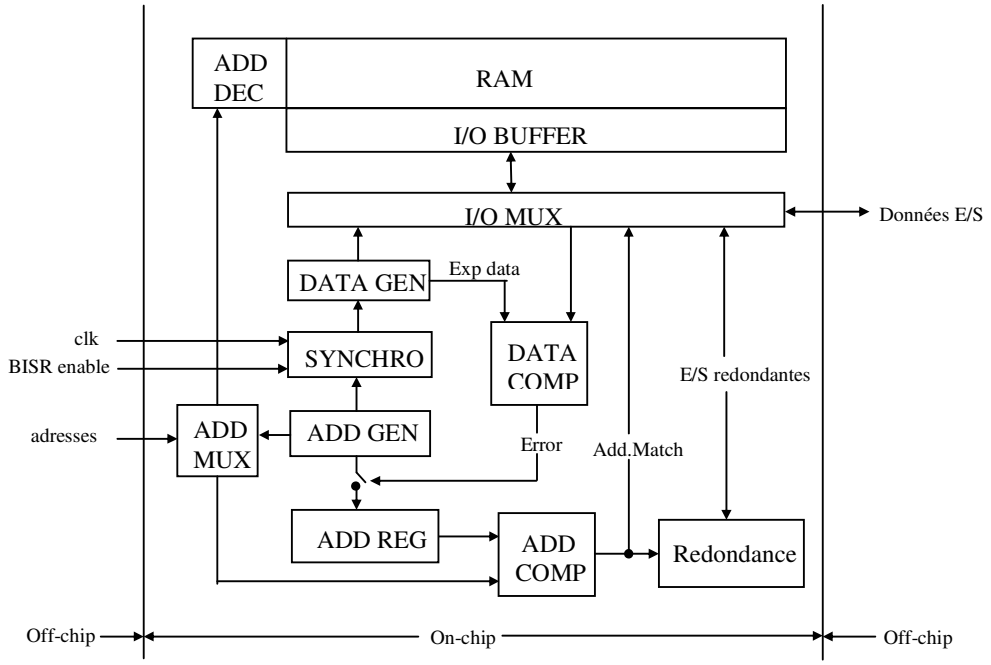


Figure 2.11. Schéma bloc du BISR basé sur le principe de la CAM [SAW89].

les parties redondantes n'ont pas été abordées dans ce genre de techniques. Cela est dû à la difficulté de prendre en compte en même temps ces deux aspects de nature très différente. Dans le chapitre V nous proposerons une architecture capable d'assurer la réparation des fautes multiples affectant les parties redondantes.

2.2.2 La Réparation des Colonnes de Mémoire (ou Bits de Donnée)

L'élément redondant dans la réparation des colonnes de mémoire est généralement l'entière unité responsable de générer un bit de donnée. Cette unité se compose d'éléments de mémoire suivants :

- Un décodeur d'adresses colonnes,
- Des cellules mémoires,
- Des lignes de bits,
- Des multiplexeurs colonne,
- Et des amplificateurs de données.

Compte tenu du nombre important de ces éléments, le remplacement d'une colonne permet de réparer un plus grand nombre de défauts en comparaison avec la réparation ligne. C'est pour cette raison que la réparation colonne demeure la stratégie la plus utilisée par les techniques de reconfiguration laser.

Architecture Matérielle

Les travaux relatifs à la réparation intégrée des colonnes de mémoire sont très récents. I. Kim et al [KIM98] ont présenté l'un des rares travaux dans ce domaine.

La Figure 2.12 montre le schéma bloc de l'architecture BISR que les auteurs proposent pour une mémoire SRAM standard. Dans cette figure, seuls les blocs actifs dans le mode d'opération normal (après la réparation) sont montrés.

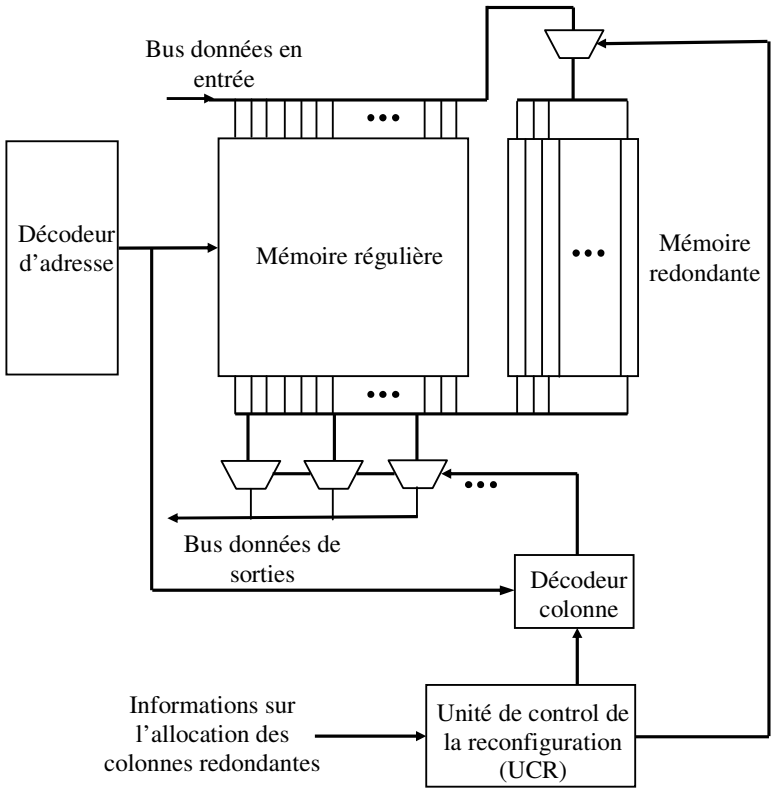


Figure 2.12. Diagramme bloc d'un BISR pour la réparation colonne [KIM98].

Nous citons ci-dessous les différents blocs de ce circuit BISR :

- Une ou plusieurs colonnes redondantes,
- Une unité de contrôle BIST/BISR (UCBB, n'apparaît pas dans la Figure 2.12) : c'est une machine d'état qui contrôle les opérations du BIST et du BISR. Pour contrôler la

fonction BIST, le UCBB utilise un générateur d'adresse pour parcourir la mémoire, un générateur de données pour acheminer les vecteurs de test aux cellules mémoire et un élément d'évaluation de données (comparateur) pour détecter les fautes. Le bloc UCBB code les adresses des cellules défaillantes pour les stocker dans le bloc UCR.

- Une unité de contrôle de la reconfiguration mémoire (UCR) : ce bloc est une mémoire qui délivre des signaux de contrôle des multiplexeurs d'entrée et de sortie de la Figure 2.12. Les valeurs mémorisées dans le bloc UCR sont calculées par le bloc UCBB.
- En plus des trois blocs cités plus haut, une circuiterie additionnelle est également incluse pour connecter les colonnes redondantes à la place des colonnes défaillantes. Cette circuiterie contient deux groupes de multiplexeurs, des multiplexeurs d'entrée pour rediriger les données vers les colonnes correctes et des multiplexeurs de sortie pour acheminer les données de sortie vers le registre de données.

Les auteurs soulignent la possibilité de rendre une partie de leur circuit BISR auto testable en testant le module UCR avec le même algorithme de test dédié à la mémoire régulière.
Afin de diminuer les pénalités temporelles, induites par le circuit BISR, deux dispositions ont été prises :

- Pendant le fonctionnement normal, les adresses courantes passent toujours par le UCR, diminuant ainsi les performances sur l'accès à la mémoire régulière. Les auteurs proposent alors d'utiliser une mémoire très rapide, d'une taille de 128 bits × 30, comme bloc UCR. Ainsi, les signaux de reconfiguration sont disponibles plus rapidement à chaque fois que la mémoire régulière est adressée pour une opération de lecture/écriture pendant le fonctionnement normal.
- Pour remédier au problème du délai au niveau des multiplexeurs surtout dans le cas d'un grand nombre de colonnes, les auteurs proposent d'utiliser un plus grand nombre de multiplexeurs mais de plus petite taille.

Afin de maîtriser la taille et la complexité des blocs UCBB et UCR, deux limitations ont été formulées :

- La première est qu'un nombre très réduit de colonnes fautives est pris en compte pour la réparation ; dans le BISR proposé, seulement deux colonnes pour une mémoire qui en contient 128 peuvent être réparées.

- La seconde limitation est qu'une seule faute est réparée par session de test. A chaque fois qu'une faute est détectée et réparée, le test est arrêté et redémarré pour détecter une nouvelle faute et ainsi de suite. Dans ces conditions, le temps de test et de réparation devient extrêmement long dans le cas des grandes mémoires affectées de fautes multiples.

Un circuit BISR pour une mémoire de 1 Mbit (64K × 16) a été réalisé par les auteurs. Le bloc RCU utilisé qui est une mémoire de 128bits × 30 plus les multiplexeurs de routage des données ont un surcoût de 4 % par rapport a la mémoire 1 Mbit. Les colonnes redondantes présentent quant à elles un surcoût de 2 %.

Nous avons vu dans cette section que les auteurs ont du limité le nombre de colonnes de la mémoire pouvant être réparées au nombre de deux. Cela est dû à la difficulté d'élaborer des fonctions de reconfiguration peu coûteuses capables de connecter vers les données entrée/sortie, les colonnes non fautives. Nous avons pour notre part développé (chapitre III), pour la réparation intégrée au niveau bit de donnée, les fonctions de reconfiguration optimales. Ainsi, on peut atteindre des capacités de réparation élevées moyennant le minimum de coût en surface.

2.2.3 La réparation combinée ligne colonne

Comme pour la réparation colonne, la réparation combinée ligne et colonne n'a pas bénéficié d'un grand nombre d'implémentations matérielles. La plupart des publications dans ce domaine proposent des algorithmes efficaces basés sur des heuristiques d'approximation, mais ne donnent pas les architectures correspondantes [BHA99], [KIM99].
Dans [MAZ90], une implémentation matérielle du BISR basée sur les réseaux de neurones a été présentée. L'approche théorique au problème de réparation a été décrite dans la section précédente.
Dans le circuit BISR, un neurone est réalisé comme un amplificateur de tension qui produit à sa sortie une tension binaire. Pendant le processus de recherche de la solution de réparation, les neurones qui sont suggérés pour le remplacement sont représentés par une tension haute à la sortie de l'amplificateur. Les autres sont représentés par une tension basse. Les influences entre les neurones ainsi que leurs entrées sont représentées par des courants électriques. Donc, une excitation portée à un neurone est un courant électrique injecté à l'entrée de

l'amplificateur. Inversement, un neurone excitateur est représenté par un abaissement du courant à l'entrée de l'amplificateur. Les poids des arrêtes dans le réseau de neurones sont simulées par des résistances qui régulent le courant qui passe à travers. En plus des neurones, le circuit se compose d'une mémoire, dans laquelle est stockée l'état de chaque cellule de la mémoire régulière. Pour ne pas dupliquer la mémoire à réparer, on stocke seulement une matrice compressée de la matrice caractéristique *D* (voir la section 2.1.5). La matrice compressée est obtenue en éliminant toute ligne et colonne ne contenant aucune faute. La solution de réparation obtenue en lisant les sorties des amplificateurs est aussi mémorisée et utilisée pour connecter ou déconnecter les unités redondantes aux unités régulières.

Le circuit qui implémente l'algorithme est compatible avec la technologie CMOS conventionnelle, en effet les auteurs ont pu réaliser son jeu de masques en technologie 2 µm. Le surcoût en surface a été estimé à 10 %. Cependant, le surcoût en surface est très dépendant de la distribution des fautes car même si une matrice compressée à la place de la matrice caractéristique est utilisée, le coût en surface peut être importante. On peut voir aussi que ce type de circuit BISR basé sur les réseaux de neurones est difficilement réalisable au niveau RTL. Il est par conséquent très dépendant de la technologie et c'est pour cette raison que les auteurs l'ont directement réalisé directement au niveau layout. Enfin, un aspect très important, concernant la prise en compte des fautes dans les parties redondantes, n'a pas été abordé. En effet si l'on se place dans le cas des fautes multiples, et l'on doit se placer dans ce cas, puisqu'il est question de réparation combinée ligne et colonne, il y a une probabilité non négligeable que des parties redondantes soient affectées par des fautes.

Conclusion

Dans ce chapitre nous avons présenté les principales techniques de réparation d'une mémoire. Il existe à ce jour deux grandes techniques de réparation : la reconfiguration laser et l'auto réparation intégrée. La première est considérée comme une réparation externe tandis que la deuxième est une réparation interne et autonome.

La reconfiguration laser utilise dans un premier temps des algorithmes programmés pour déterminer la solution de réparation. Ces algorithmes manipulent les unités redondantes avec des efficacités variables. Nous avons vu à ce sujet, qu'il y avait un compromis à faire entre l'efficacité de l'algorithme et son temps d'exécution. En général, un tel compromis est respecté, avec, à la clé, des algorithmes software efficaces avec des temps d'exécution raisonnables. La solution de réparation ainsi calculée est ensuite soit directement utilisée pour

activer les connexions redondantes, soit stockée dans une mémoire de type FLASH pour être utilisée au moment voulu.

Posons nous maintenant la question si ces algorithmes peuvent être implémentés au niveau matériel en vue d'une réparation intégrée ?

Premièrement, l'auto réparation intégrée est à l'opposé de la technique de reconfiguration laser dans le sens où elle est complètement intégrée, et se fait sans intervention extérieure. Le problème qui se pose est que plus un algorithme de réparation est efficace plus son implémentation matérielle est coûteuse et son temps d'exécution est important. Il est par conséquent impensable d'utiliser ces algorithmes très sophistiqués au niveau matériel.

Deuxièmement, étant donné que dans les technologies CMOS le nombre de cellules défaillantes est généralement faible, l'alignement de plusieurs cellules défaillantes dans le même mot la mémoire ou la même colonne est peu probable. Ainsi, le cas des mémoires nécessitant une allocation des ressources redondantes très sophistiquée reste faible. Par conséquent, le gain de rendement de production obtenu par ces algorithmes restera faible, et le coût de l'implémentation matérielle sera injustifié. D'autre part dans un contexte de grande densité de fautes, les meilleurs architectures de réparation que nous avons trouvé combinent des schémas de ressources redondantes en utilisant une allocation séquentielle, c'est-à-dire que les ressources d'un premier schéma sont allouées puis les ressources d'un second schéma sont à leur tour allouées. Ainsi, nous n'avons pas besoin d'utiliser des algorithmes de réparation très sophistiqués, tels que ceux décrits dans ce chapitre.

A la fin de ce chapitre nous avons présenté quelques implémentations hardware de circuits BISR. Bon nombre de ces implémentations restent simplistes à cause de la difficulté d'allier à la fois efficacité et coût modéré en surface. D'ailleurs, bon nombre de chercheurs essaient d'améliorer l'efficacité de leurs circuits BISR, mais ils se heurtent soit à un coût en surface inacceptable, soit à des difficultés d'ordre conceptuel.

CHAPITRE III. Architectures BISR pour la Reconfiguration des Mémoires au Niveau Bit de Donnée

Introduction

Les techniques de réparation intégrée actuelles souffrent de plusieurs limitations. Ces principales limitations concernent, le nombre de fautes pouvant être réparées, et la prise en compte des fautes à la fois dans les parties régulières et redondantes. Ces limitations sont dues à la difficulté de concevoir des circuits BISR qui soient à la fois efficaces et peu coûteux en silicium. Cela est particulièrement vrai pour la technique de réparation colonne. La réparation colonne est appelée également la réparation au niveau bit de donnée. Ce type de réparation est le plus utilisé lors de la reconfiguration avec laser car il permet de corriger un nombre élevé de fautes se situant dans la colonne d'une mémoire. Intégrer ce type de réparation autour de la mémoire reste difficile, car il faut pouvoir corriger d'une manière efficace l'ensemble des fautes en sélectionnant la taille de l'unité redondante et ce sans modifier la structure régulière de la mémoire.

Nous présentons dans ce chapitre, une série d'architectures BISR, développées dans le cadre de notre thèse, concernant la réparation de la mémoire au niveau "bit de donnée". D'une efficacité variable, ces techniques doivent être capables de réparer les défauts avec des densités jusqu'à des ordres de grandeurs deux fois plus élevées que dans les technologies actuelles.

3.1 La Réparation Statique

La réparation au niveau bit de donnée est une approche qui consiste à localiser les colonnes qui contiennent des défauts, et les remplace par des colonnes redondantes. La colonne de la mémoire (régulière ou redondante) est la partie de la mémoire responsable de générer un seul bit de donnée et ce quelque soit l'organisation physique de la mémoire. Nous pouvons schématiser cet aspect par la Figure 3.1.

Dans tout ce qui suit nous allons indiquer par k le nombre de telles unités redondantes et par n le nombre d'unités régulières.

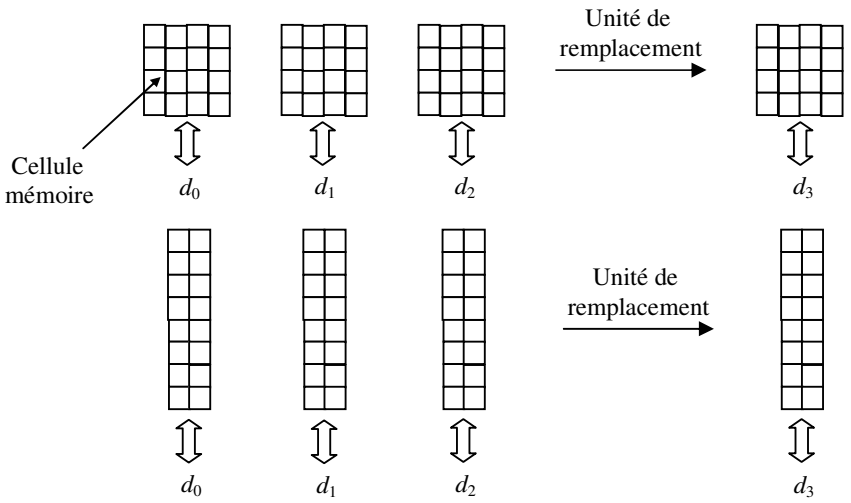


Figure 3.1. L'unité de remplacement pour deux différentes organisations physiques d'une mémoire 16 × 3 bits.

3.1.1 Localisation des Fautes

Tout d'abord nous allons nous intéresser au processus de localisation des fautes, autrement dit la détermination des unités de mémoire erronées. Pour simplifier ce processus, nous pouvons utiliser un BIST comprenant un comparateur de données. Pendant la phase de test, le comparateur du BIST compare les données à la sortie de la mémoire avec les données de référence et indique par le biais d'un seul signal l'existence ou non d'une faute. Ce signal d'indication d'erreur est généré à partir d'une rangée de portes XOR suivie d'un arbre de portes OU. Pour effectuer la réparation, il suffit de disposer des signaux à la sortie des portes XOR. Afin de mémoriser l'état des unités mémoires qu'elles soient régulières ou redondantes, nous disposons de $n+k$ latches FBI (Faulty Bit Indication). Si nous appelons X_i les signaux délivrés par le comparateur, alors nous pouvons calculer le contenu des latchs FBI représenté par les valeurs FBI_i à chaque cycle de la phase de test par l'équation :

(3.1) $(FBI_i)t_{q+1} = (FBI_i + X_i)t_q$, $\forall\ i : n+k-1 \geq i \geq 0$, avec t_q et t_{q+1} deux instances temporelles consécutives.

Par conséquent si une valeur de '1' logique est générée sur le signal X_i (une faute a été détectée sur un bit de position correspondant à la position du bit de donnée d_i), alors il sera mémorisé tout au long de la phase de test. Le dispositif de localisation des bits fautifs est montré dans la Figure 3.2. Le signal 'reset' dans la figure permet d'initialiser les latches FBI, à chaque fois qu'une nouvelle session de test est réalisée.

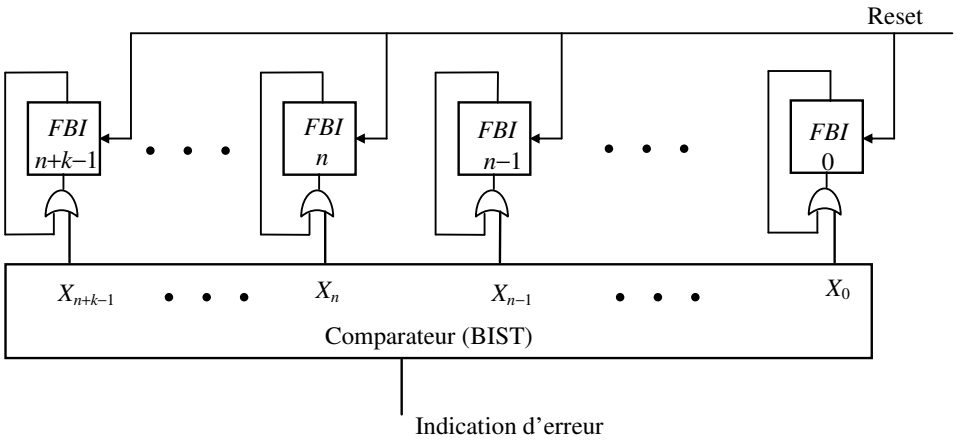


Figure 3.2. Dispositif de localisation des bits fautifs a l'aide du comparateur du BIST.

3.1.2 La Réparation Locale et Distante

Pour effectuer la réparation, nous utiliserons un jeu de multiplexeurs pour remplacer les unités fautives par les unités redondantes. Un bloc implémentant la logique de reconfiguration reçoit à son entrée les sorties des latches FBI et calcule les signaux nécessaires au contrôle des multiplexeurs.

3.1.2.1 La réparation locale

La reconfiguration peut être faite d'une manière locale. Dans ce cas de figure, une unité fautive est isolée et remplacée par l'unité correcte la plus proche se trouvant sur son côté droit (resp. côté gauche). Un tel remplacement nécessite également le décalage vers la droite (resp. vers la gauche) de toutes les connections des unités se trouvant à la droite (resp. à la gauche) de l'unité réparée. Si nous considérons un décalage vers la gauche et en désignant par U_0, U_1, …, U_{n-1} les unités fonctionnelles et par U_n, U_{n+1}, …, U_{n+k-1} les unités redondantes, on commence alors par réparer l'unité fonctionnelle de plus bas ordre (la plus à droite) en décalant sa connexion vers la gauche jusqu'à la première unité correcte. Nous procédons ensuite à la réparation de la suivante unité fautive de plus bas ordre et ainsi de suite. Un exemple de réparation avec décalage vers la gauche pour quatre unités fonctionnelles et trois unités redondantes est montré dans la Figure 3.3. Dans cette figure nous pouvons distinguer quatre multiplexeurs de type 1 parmi k+1. En effet, pour chaque unité régulière, nous devons utiliser un multiplexeur de type 1 parmi k+1 pour que chaque bit de donnée en entrée/sortie puisse être connecté soit à l'unité correspondante soit au k unités voisines.

3.1.2.2 La réparation distante

Une autre option consiste à effectuer une reconfiguration distante. Dans ce cas, les unités fonctionnelles et redondantes forment deux groupes distincts dans le processus de réparation. Chaque unité fautive est remplacée par une unité redondante correcte au lieu d'être remplacée par sa voisine. Dans ce cas, il n'y a pas de décalage des connections des unités correctes. Le même nombre de multiplexeurs est utilisé pour ce type de réparation mais les signaux qui les contrôlent sont calculés différemment. Aussi, un nombre plus petit de lignes d'interconnections est utilisée mais la longueur de chaque ligne est plus grande. Un exemple de réparation distante avec $n = 4$ et $k = 3$ est montré sur la Figure 3.4.

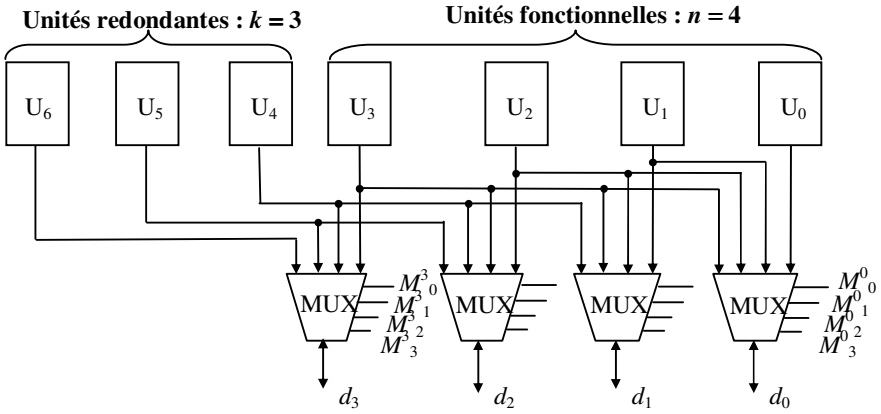


Figure 3.3. Les interconnections dans la réparation locale.

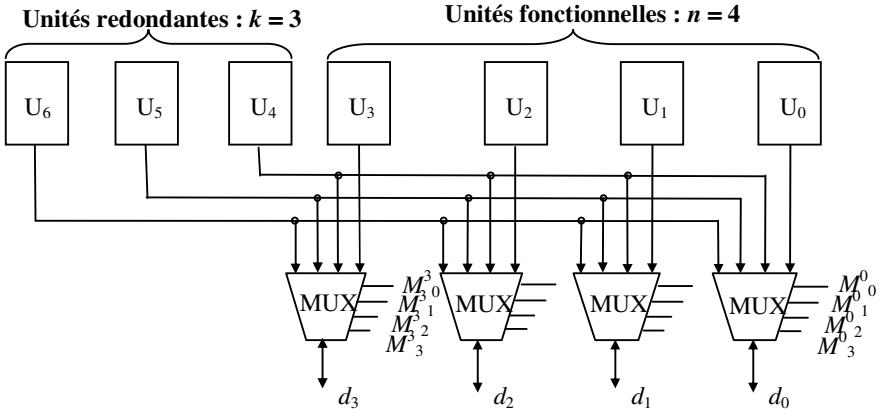


Figure 3.4. Les interconnections dans la réparation distante.

Dans les figures ci-dessus, nous présentons les multiplexeurs comme étant bidirectionnels : un sens pour écrire et un sens pour lire les données dans n unités parmi les $n+k$ disponibles, les n unités ayant été sélectionnées au cours du processus de réparation. Par

conséquent si l'on veut implémenter ces multiplexeurs comme des fonctions logiques, il faut considérer plutôt deux groupes de multiplexeurs unidirectionnels et chaque groupe étant mathématiquement l'inverse de l'autre. La Figure 3.5 montre les deux groupes de multiplexeurs en portes logiques pour une réparation locale avec décalage à droite avec $n = 3$ et $k = 2$. Dans le cas de la réparation distante, on retrouve les mêmes fonctions de multiplexage mais avec des connections différentes. Aussi, nous nous aiderons de cette figure pour établir plus tard l'impact de notre BISR sur les performances d'accès à la mémoire.

NB : Pour des raisons de simplification nous parlerons dorénavant de multiplexeurs à la place de multiplexeurs bidirectionnels.

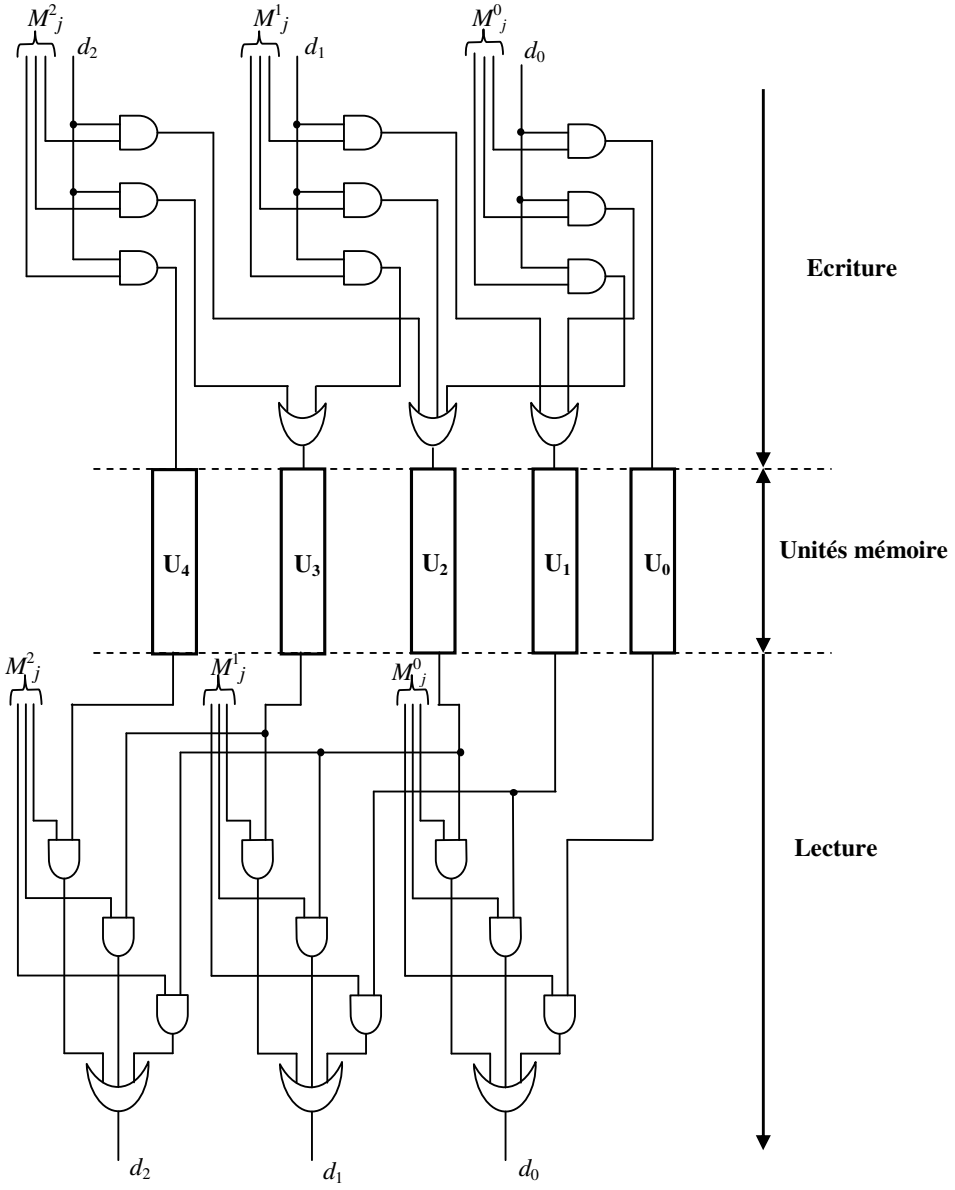


Figure 3.5. Fonctions d'écriture et de lecture dans les unités fonctionnelles et redondantes.

3.2 Fonctions de Reconfiguration pour la Réparation Statique

Dans cette section nous présentons les fonctions de reconfiguration optimales pour les différents types de réparation (locale est distante) décrits dans la section précédente.

Chaque unité fautive U_i est susceptible d'être remplacée par une des k autres unités. Dans la réparation locale, ces k unités sont U_{i+1}, U_{i+2}, …, U_{i+k} pour une réparation avec décalage à gauche, ou bien les unités U_{i-1}, U_{i-2}, …, U_{i-k} pour une réparation avec décalage à droite (Figure 3.3). Dans la réparation distante chaque unité régulière fautive peut être remplacée par une des k unités redondantes (Figure 3.4). Nous désignerons l'ensemble des unités susceptibles de remplacer une unité fautive U_i, par l'ensemble de réparation de U_i.

Pour le remplacement des unités défaillantes, on dispose pour chaque unité régulière d'un multiplexeur de k+1 entrées. Ce multiplexeur a pour fonction de connecter le bit de donnée de position d_i à l'unité régulière U_i si celle-ci est non fautive, sinon il sera connecté à l'une des k unités de l'ensemble de réparation de U_i. Les fonctions de reconfigurations se présentent alors comme les fonctions permettant le calcul des signaux nécessaires au contrôle de chacun de ces multiplexeurs.

Pour déterminer des fonctions de reconfiguration optimales il faut implémenter le circuit qui compte le nombre de décalages K^i correspondant à chaque signal d_i. Cette comptabilisation peut être effectuée par un circuit combinatoire ou séquentiel. Les k+1 signaux de contrôle de chaque multiplexeurs doivent suivre un code précis pour indiquer le nombre K^i : 100…0 pour K^i = 0, 010…0 pour K^i = 1, 001…0 pour K^i = 2, 000…1 pour K^i = k+1. Cependant, des signaux intermédiaires peuvent être utilisés pour un dénombrement suivant n'importe quel autre code mais doivent être au final décodés en 1 parmi k+1 pour produire les signaux de contrôle.

Les fonctions de reconfigurations qui seront décrite dans les sections suivantes peuvent être implémentées dans l'hypothèse que les unités redondantes peuvent être erronées. La description de fonctions de reconfiguration pour une position arbitraire i peut être très complexe puisqu'il faut tenir compte des états (fautifs et non fautifs) des unités se trouvant à la droite et à la gauche de la position i. En effet, pour calculer le nombre de décalages K^i il nous faut compter le nombre de uns logiques contenu dans un premier groupe de latches FBI de position 0 à i (formant un premier groupe de latches FBI) ainsi que le nombre de uns logiques de position supérieure à i (formant un second groupe de latches FBI). De plus, les états du second groupe de latches FBI doivent varier en fonction des états des latches FBI du

premier groupe et en fonction de leur propre état courant. Cette situation donne vite lieu à des fonctions de reconfiguration complexes difficiles à déterminer et très coûteuse à implémenter.

Dans le but de simplifier à la fois l'analyse de ces fonctions et le coût matériel, nous avons adopté une approche récursive, nous permettant de déterminer des équations pouvant être implémentées en une logique itérative.

La complexité des fonctions implémentées en logique itérative est basse mais leur délai est grand. En général, le délai des logiques itératives est linéaire et donne donc lieu à un circuit lent. Cependant, dans le cas présent, le circuit calcule une seule fois les signaux de reconfiguration (à la fin de la phase de réparation), mais ensuite ceux-ci restent stables tout au long du fonctionnement normal de la mémoire. On peut donc estimer que le délai impliqué dans le calcul des signaux de reconfiguration est dépourvu de sens.

Un autre avantage des logiques itératives que nous allons présenter est la facilité avec laquelle on peut réparer le circuit de reconfiguration lui-même. Dans ce chapitre nous considérons que la densité de fautes dans la logique n'est pas élevée. En effet, on peut considérer que la mémoire et la logique sont réalisées dans des substrats différents utilisant un processus standard pour la logique. Nous pouvons donc considérer que le circuit de reconfiguration ne contient pas de fautes et qu'il n'y a seulement que les ressources mémoires, qu'elles soient régulières ou redondantes, qui peuvent être affectées. Dans le cas où cette dernière hypothèse n'est pas valide, réparer le circuit de reconfiguration devient nécessaire. Les solutions développées ici utilisent la logique itérative qui est composée par des blocs logiques identiques excepté aux extrémités (cas limites). Dans ces conditions nous pouvons disposer de blocs logiques redondants et une réparation peut être effectuée en contournant les blocs fautifs.

Nous avons développé deux types de logiques itératives pour le calcul des signaux de reconfiguration : fonctions itératives séquentielles et fonctions itératives combinatoires. Ces variantes sont aussi combinées avec le type d'interconnexions employé, c'est-à-dire la réparation locale ou distante.

3.2.1 Fonctions de Reconfiguration pour la Réparation Locale

Nous allons dans cette section présenter deux alternatives pour l'implémentation des fonctions de reconfiguration pour la réparation locale. Une implémentation d'un circuit de reconfiguration séquentielle et un circuit de reconfiguration combinatoire.

3.2.1.1 Fonctions de reconfiguration séquentielle pour la réparation locale

Afin de compter le nombre de positions avec lesquelles la position d_i doit être décalée, nous pouvons implémenter un circuit qui génère un signal R_i. Ce signal doit demeurer à la valeur 1 pendant les K^i cycles d'horloge correspondant à la recherche de la connexion non fautive la plus proche. Un tel signal peut alors contrôler un compteur qui s'incrémente à chaque cycle d'horloge tant que le signal R_i est égal à 1. Ce compteur peut être un registre à décalage à k+1 étages initialisés à la valeur 100…0. Ce registre génère un signal sur un vecteur de k+1 bits qui contrôle les multiplexeurs de reconfiguration MUX. D'autres types de compteur peuvent être utilisées comme les compteurs binaires ou les LFSR (Linear Feedback Shift Register) afin de réduire le nombre de flips-flops surtout pour les valeurs élevées de k.
L'implémentation du circuit séquentiel qui permet de générer les signaux R_i utilise les latches FBI ainsi qu'une logique disposée tout autour. Dans une réparation avec décalage vers la gauche, le signal R_i dépend du contenu du latche FBI_i mais aussi du contenu des latches FBI_j avec $j < i$. En effet, si une position d_j a été décalée vers la gauche, toutes les positions d_i avec $i > j$ doivent subir le même nombre de décalages. Il dépend aussi du contenu des latches FBI_j avec $j > i$ car à chaque fois que la position d_i est décalée vers la gauche elle peut être connectée à une unité fautive ou non fautive.
Comme nous l'avions expliqué, les équations décrivant les fonctions de reconfiguration pour une position arbitraire i peuvent être très complexes. Et c'est pour simplifier ces équations qu'on va adopter un raisonnement récursif. Afin d'obtenir les équations permettant à chaque signal R_i de compter les K^i décalages nécessaires pour chaque position d_i, prenons en compte les observations suivantes :
(a) Une unité fautive U_i dont le latche FBI_i contient un 1, doit être décalée au moins une fois. Autrement dit, le signal R_i doit être forcée à 1 : $FBI_i = 1 \Rightarrow R_i = 1$.
(b) A chaque fois que le signal R_i est activé, il force les signaux R_{i+1} à l'état actif ('1' logique) car chaque décalage vers la gauche d'une position d_i doit engendrer un décalage vers la gauche de la position d_{i+1}.
(c) De (a) et (b) nous obtenons $R_i = FBI_i + R_{i-1}$.
(d) Quand le signal R_i est activée (R_i = 1) la valeur du latche FBI_{i+1} est transférée au latche FBI_i. Par contre quand R_i = 0, le latche FBI_i prend la valeur 0. En effet, quand une position d_i est connectée à U_{i+1} (position occupée précédemment par d_{i+1}), l'état de U_{i+1} doit

maintenant être indiqué dans le latche FBI_i. Cette analyse peut être traduite en une équation séquentielle pour chaque latche FBI durant la phase de réparation :

$(FBI_i)t_{q+1} = (R_i \cdot FBI_{i+1})t_q$ ou t_q et t_{q+1} sont deux instances temporelles consécutives.

Nous obtenons alors un jeu constitué de deux équations implémentant le circuit séquentiel qui génère les signaux R_i :

(3.2) $R_i = R_{i-1} + FBI_i$, $\forall\ i : n+k-1 \geq i \geq 0$, $R_{-1} = 0$

(3.3) $(FBI_i)t_{q+1} = (R_i \cdot FBI_{i+1})t_q$, $\forall\ i : n+k-1 \geq i \geq 0$, $FBI_{n+k} = 0$.

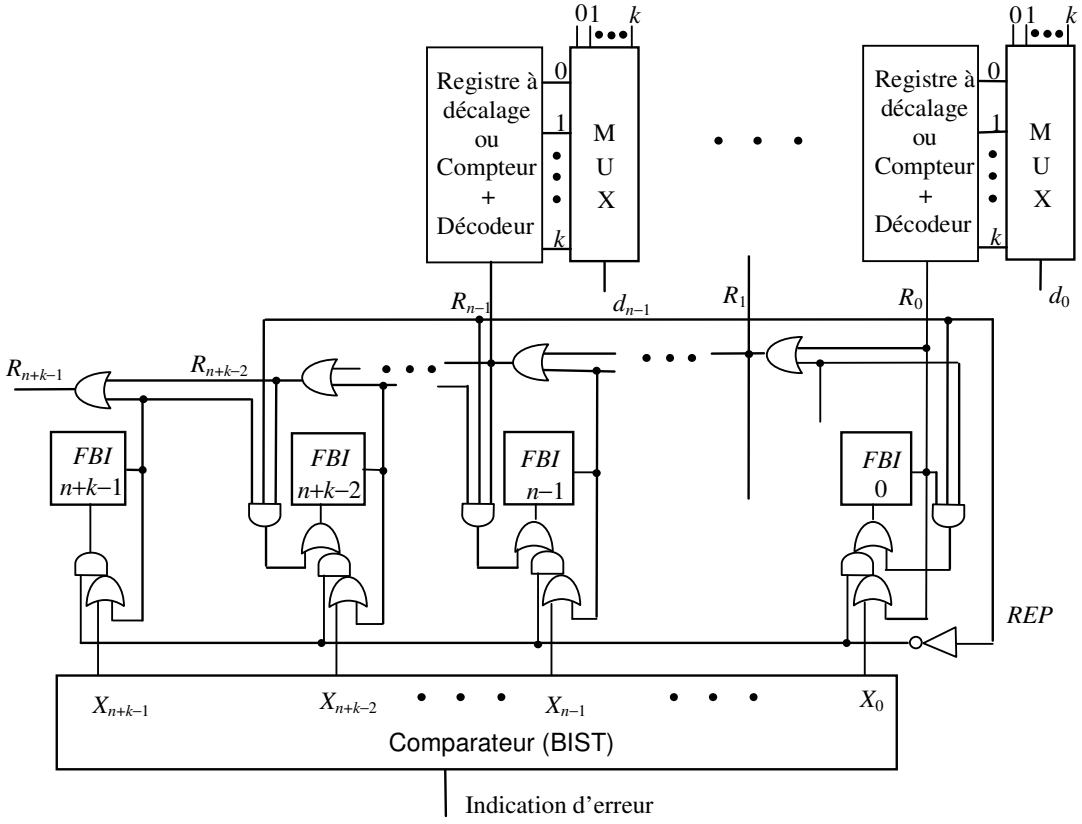


Figure 3.6. Circuit séquentiel pour la réparation locale.

Pour obtenir l'équation finale qui détermine la valeur des latches FBI_i, nous devons prendre en compte l'équation (3.1) $FBI_i = FBI_i + X_i$ dans laquelle le signal X_i indique pendant la phase de test l'état du résultat de lecture à la position d_i ($X_i = 0$ pour une lecture de donnée correcte, $X_i = 1$ pour une lecture d'une donnée erronée). Donc l'équation (3.3) peut être remplacée par l'équation (3.4) dans laquelle le signal *REP* est à 1 durant la phase de réparation et 0 durant la phase de test :

(3.4) $\left(FBI_i\right)t_{q+1} = \left(\overline{REP}.\left(FBI_i + X_i\right) + REP.R_i.FBI_{i+1}\right)t_q$, $\forall i : n+k-1 \geq i \geq 0$, $FBI_{n+k} = 0$.

L'implémentation des équations (3.2) et (3.4) est montrée par la Figure 3.6. Dans cette figure, la chaîne de portes OU interconnectées en série suivant l'équation (3.2) donne lieu à un délai important. Cela implique une phase de réparation opérant à basse fréquence. Mais cela n'altère ni la fréquence de la phase de test ni la fréquence de fonctionnement normal de la mémoire. En effet, nous pouvons utiliser lors de la conception matérielle un signal d'horloge indépendant pour la phase de réparation, ou bien utiliser le même signal d'horloge que celui des autres phases mais en le réduisant pendant la réparation.

Nous pouvons implémenter l'équation (3.2) différemment pour obtenir un circuit plus rapide. Il suffit d'implémenter un arbre de portes OU à $\log_2(n+k)$ niveaux qui génère les signaux R_i avec un délai logarithmique. Une telle implémentation est certes plus optimisée en termes de délai mais requiert un plus grand nombre de portes OU. La Figure 3.7 illustre un exemple d'une telle construction dans le cas de 8 signaux FBI. De telles optimisations peuvent aussi être effectuées automatiquement à l'aide d'outils de synthèse, selon les recommandations des designers en termes de surface, vitesse, consommation de puissance ou tout autre paramètre.

Pour une mémoire ayant n bits entrées/sorties connectés aux unités régulières U_0, U_1, …, U_{n-1}, on utilise n MUX. Chacun de ces MUX connecte une des entrées/sorties à k unités

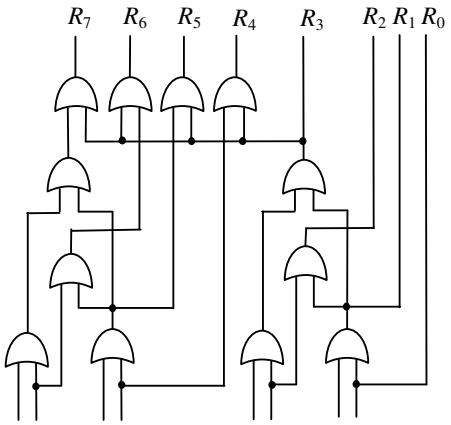


Figure 3.7. Génération des signaux R_i à l'aide d'un arbre a délai logarithmique.

réparables. De cette façon chaque bit de donnée d_i est connecté via un MUX aux unités U_i, U_{i+1}, …, U_{i+k}. Donc, chacun des signaux R_0, R_1, … R_{n-1} est utilisé pour le contrôle d'un compteur mais aussi utilisé dans les équations (3.3) et (3.4). D'autre part, les signaux R_n, …

R_{n+k-1} sont utilisés uniquement dans les équations (3.3) et (3.4) et n'ont pas de compteur qui leurs sont associé.

Le signal R_{n-1} peut être utilisé pour indiquer l'échec ou le succès du processus de réparation. Si $R_{n-1} = 0$ après k cycles d'horloge de la phase de réparation veut dire que la réparation a réussi. Par contre si $R_{n-1} = 1$ après les k cycles d'horloges cela signifie qu'il y a plus de k unités fautives et que par conséquent la mémoire la n'est pas réparable. Le signal R_{n-1} peut aussi être utilisé comme un signal d'achèvement de la réparation. En effet, manière nous n'avons pas besoin d'attendre k cycles de réparations, la réparation peut s'arrêter dès que $R_{n-1} = 0$. L'ensemble des tests sur le signal R_{n-1} indiquant la fin et l'état de la réparation, ainsi que la génération du signal *RPE* de l'équation (3.4) est gérée par une simple machine d'états finis.

3.2.1.2 Fonctions de reconfiguration combinatoire pour la réparation locale

Comme nous l'avons vu plus haut, les entrées du MUX de position i comptent dans un code 1 parmi (k+1) le nombre de positions avec lequel la donnée entrée/sortie d_i doit être décalée. Commençons par établir les fonctions d'un circuit combinatoire pouvant effectuer ce décompte en partant de l'état des latches FBI. Nous allons d'abord établir les équations donnant les signaux de contrôle du MUX de position 0 : M_0^0, M_1^0, …, M_k^0. Le code qu'on va suivre par l'intermédiaire de ces signaux est le même que celui utilisé dans la réparation locale séquentielle décrite plus haut. Un seul signal parmi les M_j^0 ($0 \le j \le k$) signaux doit être égal à 1 tout les autres doivent être à 0 et quand $M_j^0 = 1$ cela signifie que le MUX de position 0 connecte le bit de donnée d_0 à l'unité U_j. Afin de déterminer l'expression générale des signaux M_j^0, il faut donc examiner les cas pour lesquels on va connecter la position d_0 à une des unités parmi les U_0, U_1, …,U_k. Le signal $M_0^0 = 1$ si l'unité U_0 est non fautive ($FBI_0 = 0$), cela implique l'équation $M_0^0 = \overline{FBI_0}$. Le signal M_j^0 est égal à 1 ssi le bit de donnée d_0 doit être décalé avec j positions pour être connecté à la première unité non fautive. Cette situation est représentée par : $FBI_0 = FBI_1 = \ldots = FBI_{j-1} = 1$ et $FBI_j = 0$.

Nous obtenons alors les équations :

$$\textbf{(3.5)}\begin{cases} M_0^0 = \overline{FBI_0}\,, \\ M_1^0 = \overline{FBI_1}.FBI_0\,, \\ \quad \ldots, \\ M_k^0 = \overline{FBI_k}\ldots FBI_1.FBI_0 \end{cases}$$

Etant donné la complexité élevée des fonctions de reconfiguration pour une position arbitraire *i*, nous avons adopté une approche récursive pour les simplifier. Considérons les équations pour la position *i*+1. La variable M_j^{i+1} est égale à 1 si le bit de donnée d_{i+1} doit se connecter à l'unité U_{i+j+1} (*j* décalages). Une possibilité pour déterminer la loi que doit suivre la variable M_j^{i+1}, est d'examiner les variables *FBI*. Une manière plus simple serait plutôt de voir le nombre de décalages subis par le bit de donnée de position d_i par l'intermédiaire des variables M_0^i, M_1^i, …, M_k^i. Commençons par établir l'expression de M_0^{i+1}. $M_0^{i+1} = 1$ ssi l'unité U_{i+1} est non fautive ($FBI_{i+1} = 0$) et que d_i n'a pas été connectée à U_{i+1} ($M_0^i = 1$). Cela donne pour résultat l'équation : $M_0^{i+1} = \overline{FBI_{i+1}}.M_0^i$. Continuons maintenant avec M_1^{i+1}, $M_1^{i+1} = 1$ (c'est à dire d_{i+1} connectée à U_{i+2}) dans deux cas : 1/ l'unité U_{i+2} est correcte ($FBI_{i+2} = 0$) et l'unité U_{i+1} est fautive ($FBI_{i+1} = 1$) et que d_i n'a pas été connectée à U_{i+2} ($M_0^i = 1$) ou 2/ l'unité U_{i+2} est correcte ($FBI_{i+2} = 0$) et que d_i a été connecte à U_{i+1} ($M_1^i = 1$). A partir de ces deux conditions on peut écrire : $M_1^{i+1} = \overline{FBI_{i+2}}.\left(M_1^i + M_0^i.FBI_{i+1}\right)$. En tenant compte de l'équation : $M_0^{i+1} = \overline{FBI_{i+1}}.M_0^i$, nous pouvons de la même manière écrire :

$M_2^{i+1} = \overline{FBI_{i+3}}.\left(M_2^i + M_1^i.FBI_{i+2} + M_0^i.FBI_{i+1}.FBI_{i+2}\right)$.

A partir de ces équations, on peut généraliser l'expression de M_j^{i+1} :

$$M_j^{i+1} = \overline{FBI_{i+j+1}}.\left(M_j^i + M_{j-1}^i.FBI_{i+j} + M_{j-2}^i.FBI_{i+j-1}.FBI_{i+j} + \ldots + M_0^i.FBI_{i+1}.FBI_{i+2}\ \ldots\ FBI_{i+j}\right),$$
$$0 \le j \le k,\ 0 \le i \le n-2 \quad \ldots \quad (3.6)$$

Le jeu d'équations récursives (3.5) et (3.6) permet de déterminer la fonction M_j^i pour toutes les valeurs de *i* et *j*. Ces équations peuvent être directement implémentées sous forme d'une logique itérative si on connaît les expressions exactes des variables M_j^i. Pour ce faire, on peut utiliser un algorithme récursif qui commence par déterminer l'expression des équations (3.5) correspondant à $i = 0$. Puis injecter ces expressions dans (3.6) en posant $i+1 = 1$ pour obtenir les variables M_j^1. De la même façon nous obtiendrons les variables M_j^2 en posant $i+2 = 2$ et ainsi de suite pour les variables M_j^3, M_j^4, …, M_j^{n-1}.

Nous pouvons comme dans le cas de la reconfiguration séquentielle, déduire le signal qui indique l'état de la réparation une fois que les signaux M_j^i ont été calculés. Dans le cas de la réparation locale, un échec de la réparation signifie qu'au moins un bit de donnée d_i ne trouve auprès des k+1 unités redondantes auxquelles il est connecté une seule unité qui ne soit pas fautive ou déjà utilisée. Une telle situation se traduit pour une position arbitraire d_i par des variables M_j^i toutes nulles ($M_0^i = M_1^i = \ldots = M_k^i = 0$) c'est-à-dire aucune position n'est choisie. D'après l'équation (3.6), la valeur nulle des variables M_j^i se propage à toutes les positions d_j avec $j > i$. Il suffit donc de détecter une telle situation directement sur les variables M_j^i se trouvant à la dernière position d_{n-1}, on obtient alors :

$$\begin{cases} \overline{\sum_{j=0}^{k} M_j^{n-1}} = 0, \text{ succès de la réparation,} \\ \overline{\sum_{j=0}^{k} M_j^{n-1}} = 1, \text{ échec de la réparation.} \end{cases}$$

Le circuit de reconfiguration crée par l'implémentation des équations combinatoires (3.5) et (3.6) nécessite un cycle de réparation plus court que celui utilisé dans la réparation séquentielle et il est égal au délai nécessaire pour le calcul des signaux M_j^i à partir de l'état des latches FBI. Cependant la taille d'un tel circuit de reconfiguration augmente rapidement avec la valeur de k (nombre d'unités redondantes).

3.2.2 Fonctions de Reconfiguration pour la Réparation Distante

Nous allons dans cette section présenter deux alternatives pour l'implémentation des fonctions de reconfiguration : une implémentation d'un circuit de reconfiguration séquentielle et un circuit de reconfiguration combinatoire.

3.2.2.1 Fonctions de reconfiguration séquentielle pour la réparation distante

Considérons une réparation avec décalage vers la droite. Nous disposerons alors de n unités régulières placées à la gauche de k unités redondantes. Aussi, nous garderons le même ordre des unités que celui utilisé dans les sections précédentes, c'est à dire, U_{n+k-1}, U_{n+k-2}, …, U_0. Ainsi la première unité régulière fautive la plus à gauche est remplacée par la première unité redondante non fautive la plus à gauche. Ensuite, c'est la suivante unité régulière fautive la plus à gauche qui est remplacée par une l'unité redondante "valable" la plus à gauche.

L'unité redondante valable désigne une unité redondante non fautive et non déjà occupée pour réparer une quelconque unité régulière fautive.

Afin de compter le nombre de positions avec lequel d_i doit être décalée, nous allons implémenter un circuit séquentiel autour des latches FBI. Contrairement au cas de la réparation locale, nous devons ici implémenter un circuit séquentiel autour des latches FBI correspondant aux positions régulières et un second circuit autour des latches FBI correspondant aux positions redondantes. Bien entendu seuls les signaux générés par le premier circuit sont utilisés pour la réparation. Pour implémenter le circuit de reconfiguration séquentielle pour une réparation distante, nous avons encore adopté une approche récursive dont le but est de simplifier les fonctions de reconfiguration. En effet, le signal qui doit compter le nombre de décalages pour une position arbitraire d_i, dépend du latche FBI_i mais aussi des latches FBI_j avec $j > i$, et de l'ensemble des latches FBI correspondant au unités redondantes. La prise en compte de tous ces éléments donne lieu à des équations complexes et à un coût élevé en surface.

Partons de l'état initial des latches FBI_i, c'est-à-dire l'état obtenu à la fin de la session de test. Toutes les unités régulières fautives ($FBI_i = 1$) sont connectées en même temps à la première unité redondante (U_{k-1}). Cela peut se faire en incrémentant chaque compteur de ces positions directement à l'aide des signaux FBI_i. Le premier cycle de réparation est dès lors achevé. S'ensuit alors le second cycle de réparation dans lequel on reconfigure le contenu des latches FBI. Les latches FBI pour les positions régulières se reconfigurent différemment de ceux des positions redondantes. Dans ce second cycle, un latche FBI_i d'une unité fautive U_i maintient sa valeur à 1 (ce qui implique une seconde incrémentation du compteur et donc une connexion à la seconde colonne redondante), ssi dans le cycle précédent il existait au moins une seule unité fautive U_j ($j > i$) se trouvant à gauche de U_i ou bien si l'unité redondante U_{k-1} est fautive. A partir de ces observations, nous pouvons écrire :

(3.7) $(FBI_i)t_{q+1} = (FBI_i \,.\, (FBI_{k-1} + R_{i+1}))t_q$, $\forall\ i : n+k-1 \geq i \geq k$, $R_{n+k} = 0$.

Dans l'équation (3.7) le signal intermédiaire R_{i+1} est utilisé pour indiquer s'il y a au moins une unité fautive U_j avec ($j > i$) se trouvant à gauche de U_i, les signaux R_i sont alors définis comme suit :

(3.8) $R_i = FBI_i + R_{i+1}$, $\forall\ i : n+k-1 \geq i \geq k$, $R_{n+k} = 0$.

Comme nous l'avons souligné, à chaque cycle de réparation, la valeur des latches FBI_i des positions régulières dépend de l'état de l'unité redondante utilisée lors du cycle de réparation précédent. Un tel état se trouvera toujours à la sortie du latche FBI_{k-1} si l'on effectue un décalage en série des valeurs des latches FBI_i ($k-1 \geq i \geq 0$). Suite à cette observation nous pouvons écrire :

(3.9) $(FBI_i)t_{q+1} = (FBI_{i-1})t_q, \forall\ i :\ k-1 \geq i \geq 0,\ FBI_{-1} = 0.$

Pour obtenir l'équation finale qui détermine la valeur des latches FBI_i, nous devons prendre en compte l'équation (3.1) $FBI_i = FBI_i + X_i$ dans laquelle le signal X_i indique pendant la phase de test l'état du résultat de lecture à la position d_i ($X_i = 0$ pour une lecture de donnée correcte, $X_i = 1$ pour une lecture d'une fausse donnée). Donc les équations (3.7) et (3.9) peuvent être remplacées respectivement par les équations (3.10) et (3.11) dans lesquelles le signal *REP* est à 1 durant la phase de réparation et à 0 durant la phase de test :

(3.10) $\left(FBI_i\right)t_{q+1} = \left(\overline{REP}.\left(FBI_i + X_i\right) + REP.FBI_i.\left(FBI_{k-1} + R_{i+1}\right)\right)t_q,\ \forall i : n+k-1 \geq i \geq k,\ R_{n+k} = 0.$

(3.11) $\left(FBI_i\right)t_{q+1} = \left(\overline{REP}.\left(FBI_i + X_i\right) + REP.FBI_{i-1}\right)t_q,\ \forall i : k-1 \geq i \geq 0,\ FBI_{-1} = 0.$

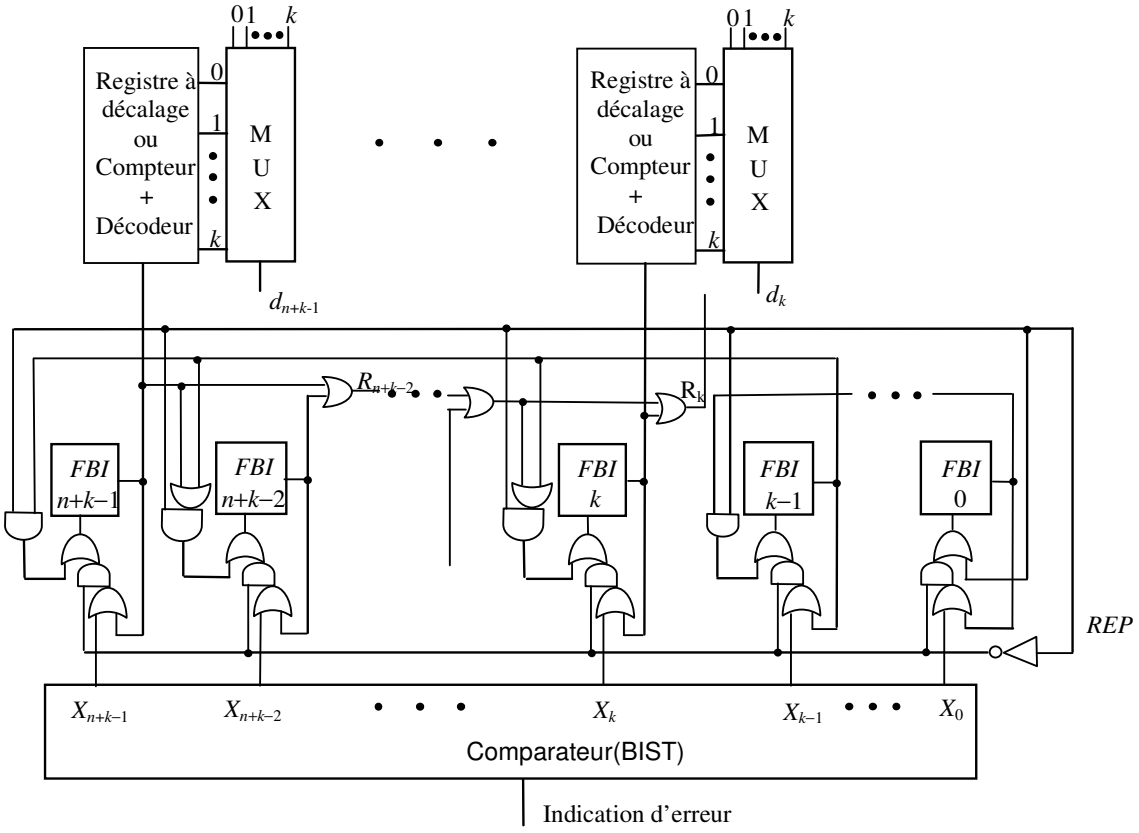


Figure 3.8. Circuit séquentiel pour la réparation distante.

Le schéma décrit par ces équations est montré dans la Figure 3.8. Le signal R_k peut être utilisé pour indiquer l'échec ou le succès du processus de réparation. $R_k = 0$ après k cycles d'horloge de la phase de réparation cela veut dire que la réparation a réussi. Par contre $R_k = 1$ après les k cycles d'horloges signifie qu'il y a plus de k unités fautives dans la mémoire et que par conséquent la réparation a échoué. R_k peut aussi être utilisé comme un signal d'achèvement de la réparation. Le processus de réparation peut s'arrêter dès que $R_k = 0$. L'ensemble de ces tests sur le signal R_k qui indiquent la fin et l'état de la réparation ainsi que la génération du signal *RPE* sont gérés par une simple machine d'états finis.

3.2.2.2 Fonctions de reconfiguration combinatoire pour la réparation distante

Dans la réparation distante une unité régulière fautive est toujours remplacée par une unité redondante correcte et non pas par l'unité correcte la plus proche. Par conséquent, la position une unité régulière correcte n'est jamais décalée. Soient RU_0, RU_1, …RU_{n-1} les n unités régulières et SU_1, SU_2, …, SU_k les k unités redondantes. Les latches FBI correspondant sont respectivement symbolisées par RF_0, RF_1, …, RF_{n-1} et SF_1, SF_2, …, SF_k.

Les équations qui donnent les signaux de contrôle du MUX de position 0 sont les plus simples à obtenir. Le signal M_0^0 est égal à 1 si RU_0 est correcte. Le signale M_j^0 est à 1 si RU_0, SU_1, SU_2, …, SU_{j-1} sont fautives et SU_j est correcte. Nous obtenons donc les équations:

(3.12)
$$\begin{cases} M_0^0 = \overline{RF_0}, \\ M_1^0 = \overline{SF_1}.RF_0, \\ \ldots, \\ M_k^0 = \overline{SF_k}.SF_{k-1} \ldots SF_1.RF_0. \end{cases}$$

Les équations pour un bit de donnée entrée/sortie d_i peuvent être très complexes car elles font intervenir les variables *RF* et *SF* d'une manière compliquée. Afin de réduire la complexité du problème nous avons encore adopté une approche récursive. Cependant, l'analyse pour la détermination des variables M_j^i est plus délicate que dans le cas de la réparation locale. En effet, dans le cas présent, quand nous calculons les variables M_j^{i+1} nous ne pouvons pas utiliser les variables M_j^i avec $0 \le j \le k$ comme indicateur du nombre de positions redondantes occupées après la réparation de quelques unités parmi RU_0, RU_1, …RU_i. C'est parce que dans le cas présent, si RU_i est correcte, alors les variables M_1^i, M_2^i ,…, M_k^i sont toutes à 0, même si quelques unités parmi RU_0, RU_1, …RU_{i-1} sont fautives. C'est pourquoi, nous introduisons

des variables intermédiaires F_j^i, $0 \le i \le k$, qui comptent le nombre d'unités redondantes occupées par les unités fautives RU_r, $0 \le r \le i$. Ces variables sont calculées par les équations suivantes :

Pour la position 0, les variables *F* sont égales aux variables *M*. Nous avons alors :

(3.13) $F_j^0 = M_j^0, \forall\, j \in \{0, 1, \ldots, k\}$.

Pour la position i+1, la variable F_j^{i+1} est égale à la variable F_j^i si l'unité RU_{i+1} est correcte, sinon elle est égale à M_j^{i+1}. Nous avons alors :

(3.14) $F_j^{i+1} = F_j^i . \overline{RF_{i+1}} + M_j^{i+1} . RF_{i+1}, 0 \le i \le n-2, 0 \le j \le k.$

La variable $M_0^{i+1} = 1$ ssi RU_{i+1} est correcte. La variable M_{j+1}^{i+1} est égale à 0 si RU_{i+1} correcte. Si RU_{i+1} est fautive ($RF_{i+1} = 1$), alors, M_{j+1}^{i+1} est égale à 1 si SU_{j+1} et correcte ($SF_{j+1} = 0$) et si on se trouve dans l'une des ces deux situations :

- si le nombre total *r* d'unités redondantes utilisées par les unités régulières RU_0, RU_1, …RU_i est égal à *j* ($F_j^i = 1$).
- si l'unité redondante SU_j est fautive ($SF_j = 1$) et que le nombre total *r* d'unités redondantes utilisées par les unités régulières RU_0, RU_1, …RU_i est égal à $j-1$ ($F_{j-1}^i = 1$). Cette dernière observation doit être répétée jusqu'à ce que nous atteignons la valeur nulle de *r* ($F_0^i = 1$) et que toutes les unités redondantes SU_1, SU_2, …, SU_j sont fautives ($SF_1 = SF_2 = \ldots = SF_j = 1$).

A partir de cette analyse nous pouvons écrire :

(3.15)
$$\begin{cases} M_0^{i+1} = \overline{RF_{i+1}}, \\ M_{j+1}^{i+1} = \overline{SF_{j+1}} . RF_{i+1} . \left(F_j^i + F_{j-1}^i . SF_j + F_{j-2}^i . SF_{j-1} . SF_j + \ldots + F_0^i . SF_1 . SF_2 \ldots SF_j \right), \\ 0 \le j \le k-1, 0 \le i \le n-2. \end{cases}$$

Dans le cas présent, un échec de la réparation peut être directement détectée sur les variables F_j^{n-1} de la manière suivante :

$$\begin{cases} \overline{\sum_{j=0}^{k} F_j^{n-1}} = 0, \text{ succès de la réparation,} \\ \overline{\sum_{j=0}^{k} F_j^{n-1}} = 0, \text{ échec de la réparation.} \end{cases}$$

Les équations (3.12), (3.13), (3.14) et (3.15) définissent entièrement les fonctions de reconfiguration pour la réparation distante. De part la nature récursive de ces équations, le plus simple est de les implémenter directement sous la forme d'une logique itérative.

3.3 La Réparation Dynamique au Niveau Donnée

La taille de l'unité remplaçable est un paramètre majeur dans la réparation. Elle influence le surcoût en surface nécessaire à la réparation des fautes. Si l'unité remplaçable est petite, alors la surface occupée par les éléments de mémorisation, la logique de reconfiguration et les interconnexions devient très grande. Supposons par exemple que l'unité remplaçable est une cellule mémoire alors le nombre possible d'unités fautives et de localités différentes est très élevé et requiert une grande capacité mémoire pour stocker ces informations. Aussi, la logique nécessaire à la déconnexion d'une unité fautive pour la remplacer par une unité redondante devient très grande. D'un autre coté, si la taille de l'unité remplaçable est grande, alors pour réparer une seule cellule fautive, une grande unité redondante sera utilisée rendant le coût de réparation par cellule, élevé. Par conséquent, il y a une taille optimale de l'unité remplaçable pour une taille de mémoire et densité de fautes données. Cependant, même si la taille optimale est déterminée, il n'est pas toujours simple d'implémenter le circuit de réparation avec cette taille à cause des contraintes topologiques liées au design de la mémoire. En effet, les mémoires ont un layout très régulier et très compact et insérer dans le layout le circuit de reconfiguration implique la brisure de cette régularité et un accroissement considérable de la surface de la mémoire. C'est pour cette raison qu'il est important d'effectuer la réparation en utilisant les bits de données entrées/sorties pour reconfigurer les différentes unités. Les solutions présentées précédemment ont ce mérite. Cependant avec ces solutions, chaque unité réparable inclut l'ensemble des cellules connectées à un seul bit de donnée entrée/sortie de la mémoire. La taille d'un tel bloc peut être très grande. Par exemple, une mémoire de 128 K ayant des mots de 32 bits, connecte à chaque bit de donnée entrée/sortie 4 K de cellules mémoire ce qui représente la taille de l'unité remplaçable. Pour réduire cette taille, nous avons développé une solution capable de sélectionner des unités réparables de plus petite taille sans pour autant avoir un accès direct à la cellule en agissant toujours sur les bits de données entrée/sortie.

L'idée est de reconfigurer la sortie mémoire d'une façon dynamique au lieu d'une façon statique. En effet, dans les Figures 3.3 et 3.4, une donnée entrée/sortie est décalée d'une façon permanente si jamais une cellule connectée à cette donnée est fautive. L'approche dynamique

utilise de plus petites unités réparables composées d'un sous ensemble de cellules connectées à un bit de donnée entrée/sortie. Donc au lieu que le décalage d'un bit de donnée se fasse d'une manière permanente, il se fait dynamiquement. C'est à dire qu'une donnée entrée/sortie est décalée seulement quand l'unité réparable contenant une cellule fautive est sélectionnée par l'adresse mémoire. Afin de bien schématiser le concept de la réparation dynamique, nous pouvons reprendre l'exemple de la Figure 3.1(a) dans lequel la réparation statique est montrée. En effet dans cette figure, 16 cellules mémoires sont connectées à chaque bit de donnée entrée/sortie. Dans une réparation dynamique qui utilise de plus petites unités réparables contenant 4 cellules mémoires chacune, on aura alors 4 unités réparables (groupe1, groupe2, groupe3 et groupe4) par bit de donnée entrée/sortie comme le montre la Figure 3.9. Sur cette dernière figure, l'efficacité de la réparation dynamique par rapport à la réparation dynamique est aussi mise en évidence. Pour une distribution de fautes telle quelle est illustrée dans la Figure 3.9, il faudrait quatre unités redondantes pour réparer la mémoire dans le cas statique alors que dans le cas dynamique il suffit d'une seule unité redondante englobant quatre unités réparables pour réussir la réparation. En effet, La réparation dynamique s'effectue en considérant chaque groupe indépendamment des autres. En diminuant encore plus la taille de l'unité réparable, on peut réparer encore plus de fautes avec une seule unité redondante. Plus généralement, si nous disposons de k unités redondantes et que chacune d'entre elles contient r unités réparables, alors le nombre de fautes qu'on peut réparer est de $r \times k$.

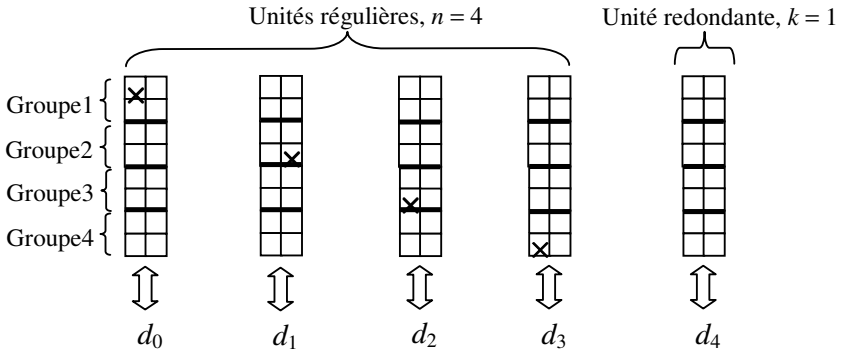


Figure 3.9. Principe de la réparation dynamique.

3.3.1 Implémentation Matèrielle de la Réparation Dynamique

Pour expliquer l'implémentation du circuit relatif à la réparation dynamique, considérons d'abord la Figure 3.10 dans laquelle la mémoire contient n blocs de cellules

chacun correspond à une entrée/sortie régulière et k blocs redondants de la même taille. Si l'on considère une réparation statique, le circuit de reconfiguration se compose d'éléments mémorisants pour la localisation des fautes (n+k latches FBI), n multiplexeurs (MUX) permettant le redirection des données entrées/sorties aux différents blocs de la mémoire et enfin un circuit de reconfiguration qui génère les signaux de contrôle de ces MUX. L'unité réparable dans ce cas est le bloc de cellules connectées à une donnée entrée/sortie.

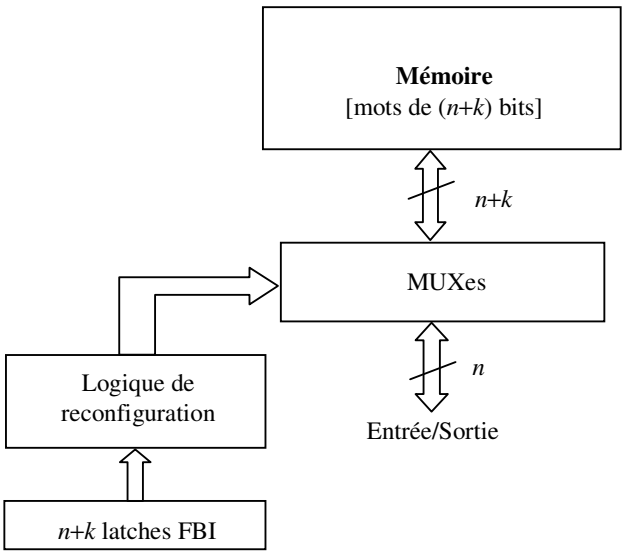


Figure 3.10. Schéma de réparation agissant sur les données entrée/sortie.

Dans l'objectif de diviser la taille de cette unité réparable par le facteur R, la nouvelle solution utilise R blocs de n+k latches FBI. Durant le test, les informations sur les localités fautives sont stockées dans différents blocs de latches FBI suivant la position de la cellule fautive dans l'espace des adresses mémoire. Nous pouvons par exemple considérer un sous ensemble des bits d'adresses de la mémoire noté A_1, A_2, …, A_r, et allouer pour chacune des valeurs de ces bits un bloc de n+k latches FBI. Donc, le bloc 0 de latches FBI va contenir les informations de localisation de toutes les cellules fautives accédées par la valeur A_1, A_2, …, A_r = 00…0, le bloc 1 correspondra à la valeur A_1, A_2, …, A_r = 10….0, et le bloc R−1 correspondra à la valeur A_1, A_2, …, A_r = 11…1. Nous avons alors dans ce cas $R = 2^r$ et nous avons divisé le bloc de cellules connectées à une donnée entrée/sortie en R sous blocs, chacun correspondant à une unité réparable. La Figure 3.11 montre comment les informations de localisation des fautes sont mémorisées dans les différents blocs de latches FBI. Comme dans le cas statique, les latches FBI ont besoin à leur entrée des signaux X_i générés par le comparateur du BIST. A chaque cycle de la phase de test, les signaux X_i entrent dans un bloc de latches FBI

correspondant à la valeur des bits d'adresses A_1, A_2, …, A_r . Cette opération est réalisée à l'aide d'un multiplexeur contrôlé par ces bits d'adresse.

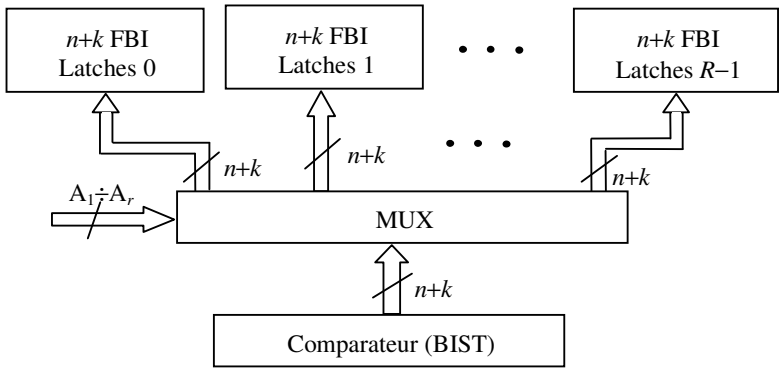


Figure 3.11. Sélection des blocs de latches FBI durant la phase de test.

Durant le fonctionnement normal du système, les bits d'adresses A_1, A_2, …, A_r sont encore une fois utilisés pour déterminer quel bloc de latches FBI sera utilisé pour le pilotage de la logique de reconfiguration durant chaque accès mémoire comme il est montré dans la Figure 3.12.

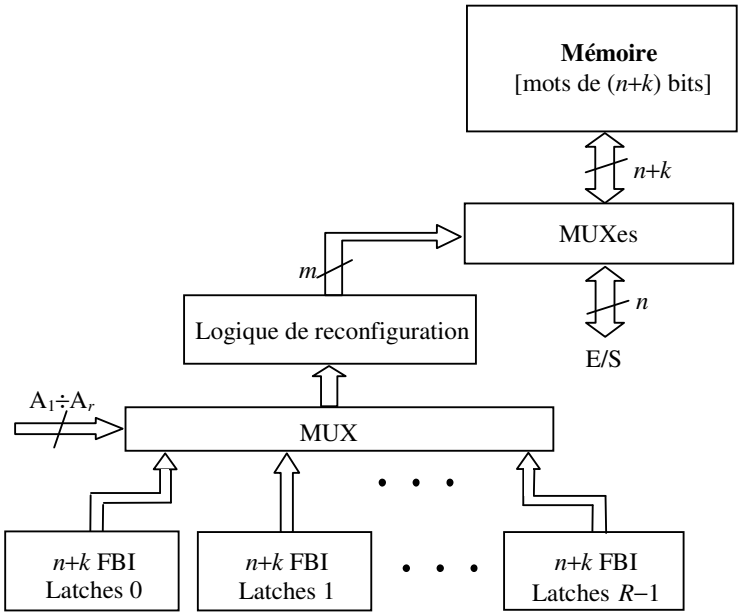


Figure 3.12. Reconfiguration dynamique des données E/S durant le fonctionnement normal.

Une autre possibilité est d'utiliser *R* blocs de logique de reconfiguration au lieu d'un seul et placer des multiplexeurs, contrôlés par les bits d'adresses A_1, A_2, …, A_r aux sorties de ces blocs comme le montre la Figure 3.13. Les *R* blocs de logique de reconfiguration peuvent

être réalisés en logique séquentielle (distante ou locale) et/ou en logique combinatoire (distante ou locale) étant donné que ces blocs, sélectionnés par des valeurs différentes de A_1, A_2, …, A_r,

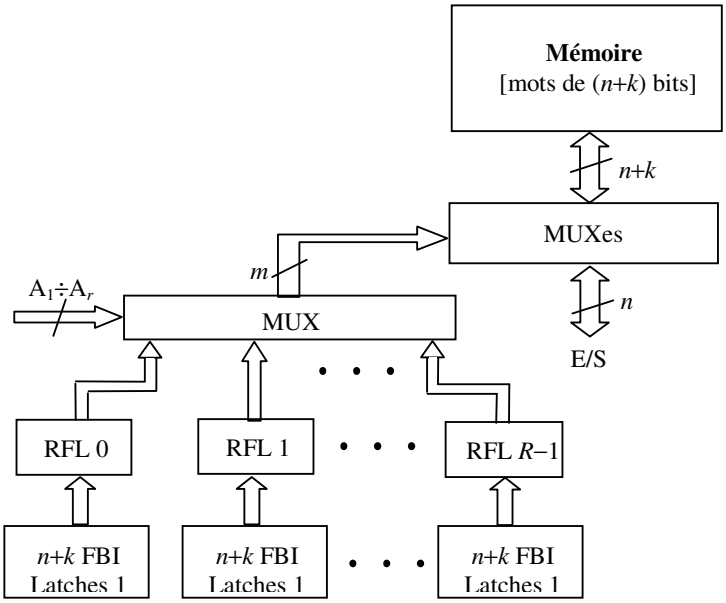


Figure 3.13. Circuit rapide pour la reconfiguration dynamique.

sont indépendants. Dans le cas de la Figure 3.13, le coût matériel est plus élevé mais on augmente la vitesse car les signaux sont prêts à la sortie des blocs de logique de reconfiguration à tout moment. En effet, le plus long chemin critique du circuit de réparation inclut les deux MUX, alors que dans la Figure 3.12 il inclut les deux MUX et la logique de reconfiguration.

3.3.2 Compromis Entre les Paramètres de la Réparation

En utilisant la réparation dynamique nous pouvons augmenter l'efficacité de la réparation puisque nous divisons par *R* la taille de l'unité réparable. Si l'on utilise tous les bits d'adresse mémoire pour contrôler les MUX dans la réparation dynamique, alors l'unité réparable se réduit à la taille de la cellule. Cela réduit le coût des unités redondantes nécessaires.

Cependant, le coût des latches FBI devient excessif. Les paramètres taille de l'unité redondante et taille de la logique de reconfiguration évoluent de façon opposée. C'est pour cette raison qu'il existe un nombre de bits d'adresses *r* optimal qui donne le meilleur

compromis entre ces deux paramètres. Supposons par exemple qu'on veuille réparer 8 fautes dans une mémoire de 1 Mbit ayant des mots de 32 bits. Le bloc de cellules connectées à chaque donnée entrée/sortie inclut 32 K cellules mémoire. C'est la taille de l'unité réparable utilisée dans la réparation statique. Pour réparer les 8 fautes il faudrait donc 8 unités redondantes représentant un total de 256 K cellules redondantes. En prenant $r = 1$, nous divisons par 2 la taille des unités redondantes soit 128 K de cellules redondantes (un gain de 128 K cellules redondantes) mais nous avons besoin de $2\times(32 + 8/2) = 72$ latches FBI au lieu de $32 + 8 = 40$ latches FBI (un accroissement de 32 latches FBI) et un second bloc de logique de reconfiguration (Figure 3.13). Le gain est alors significatif puisque nous gagnons 128 K cellules redondantes en payant seulement 32 latches FBI et un circuit de logique de reconfiguration. Si nous prenons $r = 2$, nous gagnons par rapport au cas $r = 1$, 64 K cellules redondantes en payant 64 latches FBI supplémentaires et deux blocs de logique de reconfiguration. Le gain est encore significatif, mais il est divisé par deux alors que le coût de la surface additionnelle est multiplié par deux. Cette situation se répète à chaque fois qu'on augmente la valeur de *r*. On peut alors arriver à un point où le gain est moins important que le surcoût en surface. Nos expérimentations ont montré que pour un nombre de fautes modéré (ce qui est le cas dans les technologies actuelles), un tel nombre *r* existe et correspond à un surcoût en surface de silicium de quelques pourcentages par rapport à la surface de la mémoire.

3.4 Reconfiguration Dynamique Utilisant des Colonnes Singulières Comme Blocs Redondants

Soit *Nd* le nombre de cellules connectées à chaque bit de donnée entrée/sortie. La Figure 3.1 utilise *k* blocs redondants de cette sorte, composé chacun de *Nd* cellules mémoire. Dans la reconfiguration statique chaque bloc représente une unité réparable. Dans ce cas, pour réparer *q* fautes, on a besoin de $k = q$ unités redondantes résultant en *q* blocs redondants composés de *Nd* cellules chacun. Dans le cas de la reconfiguration dynamique, chaque bloc (de *Nd* cellules) connecté à une donnée entrée/sortie inclut *R* unités réparables. Donc, chaque unité réparable est composée de *Nd*/*R* cellules mémoire. Dans ce cas pour réparer *q* fautes nous avons besoin de $k = q/R$ blocs redondants de *Nd* cellules. Ainsi, nous avons divisé par *R* le coût des blocs redondants. Cependant, pour certaines distributions des *q* fautes, une mémoire donnée ne peut être réparée. En effet, la réparation dynamique utilise les bits d'adresses A_1, A_2, …, A_r et chaque valeur de A_1, A_2, …, A_r sélectionne un groupe de $n+k$

unités réparables. S'il y a plus de *q*/*R* fautes dans le même groupe (sélectionne par la même valeur de A_1, A_2, …, A_r) alors la mémoire n'est pas réparable. Par conséquent, pour une densité de fautes donnée, la réparation statique utilisant *q* blocs redondants de *Nd* cellules peut donner lieu à un rendement plus élevé qu'une

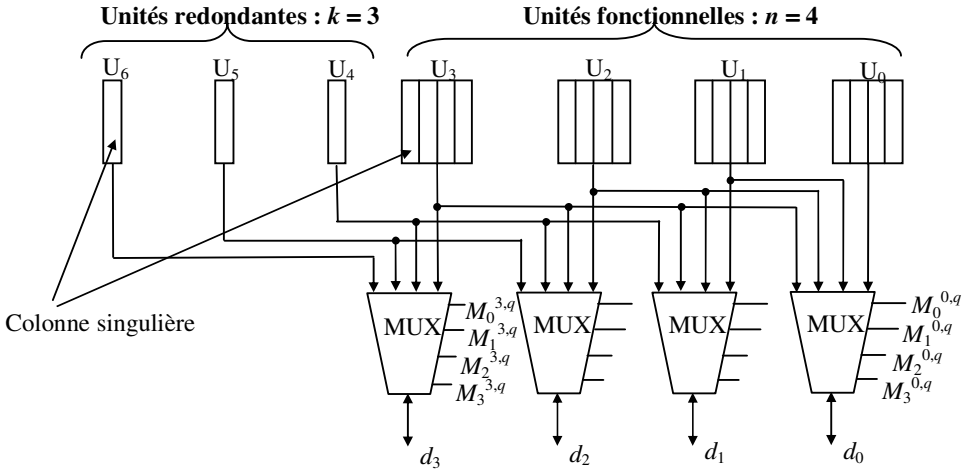


Figure 3.14. Reconfiguration dynamique utilisant des colonnes singulières comme unités redondantes. Exemple avec *n* = 4, *k* = 3, *R* = 4.

réparation dynamique qui utilise *q*/*R* blocs redondants de *Nd* cellules. Pour obtenir le même rendement, il faut utiliser un nombre de blocs redondants supérieur à *q*/*R*.

Nous présentons dans cette section une méthode de reconfiguration dynamique plus performante qui permet de remédier au problème décrit plus haut. La technique va être décrite pour le cas combinatoire local. Une approche similaire peut être utilisée pour les autres types de reconfiguration. Dans cette approche au lieu de disposer de *k* blocs redondants chacun composé d'autant de cellules que de cellules connectées à une donnée entrée/sortie (*Nd* cellules), nous disposons de *k* blocs chacun d'eux est une colonne singulière (Bit Line) comme le montre la Figure 3.14. Pour expliquer la notion de colonne singulière prenons l'exemple d'une mémoire utilisant un multiplexage colonne de 1 parmi 32. Alors, chaque bloc redondant contiendra une seule colonne singulière composée de *Nd*/32 cellules. Nous pouvons alors utiliser une reconfiguration dynamique dans laquelle les bits d'adresse utilisés pour contrôler le MUX de la Figure 3.13 sont exactement les bits d'adresse colonne A_1, A_2, A_3, A_4, A_5 de la mémoire. Nous avons alors $R = 32$ et la taille de l'unité réparable est de *Nd*/32 (une colonne). On a ici la même réduction de taille de l'unité réparable que dans le cas dynamique de la section précédente (*Nd/R*). Aussi, les unités réparables (colonnes singulières) régulières sont sélectionnées par rapport à une valeur donnée de A_1, A_2, A_3, A_4, A_5. Par contre, les bits de données redondants sont générés par des colonnes singulières qui peuvent être

sélectionnées quelque soit la valeur de A_1, A_2, A_3, A_4, A_5. En fait, ces blocs redondants composé chacun d'une colonne, n'utilise pas de multiplexage colonne. De ce fait, chaque groupe d'unités réparables régulières sélectionnées par la même valeur de A_1, A_2, A_3, A_4, A_5 peut disposer des *k* colonnes redondantes, ce qui garantit la réparation de la mémoire incluant *k* fautes quelque soit leur distribution.

Présentons ci-dessous l'architecture matérielle qu'on a développé pour ce type de réparation.

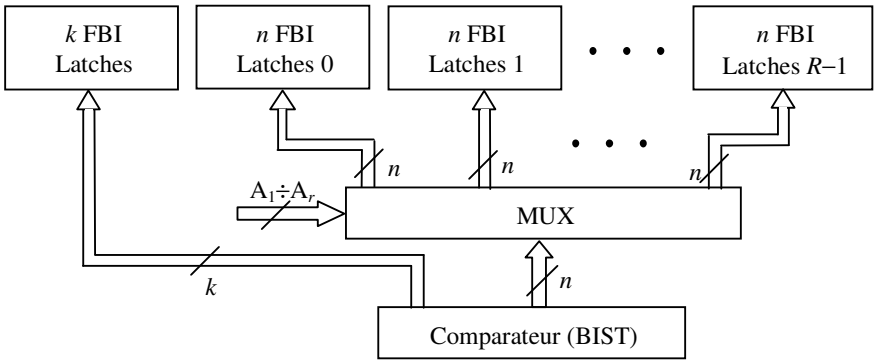


Figure 3.15. Sélection des latches FBI durant le test.

3.4.1 Localisation des Fautes

Pour chaque position régulière nous utilisons *R* FBI latches puisque chaque bloc régulier inclut *R* unités réparables (colonnes) comme dans le cas dynamique. Nous utilisons un seul latche FBI pour chaque bloc redondant puisque qu'un bloc redondant ne contient qu'une colonne. Pendant la phase de test, une faute détectée sur une position redondante force le latche FBI de cette position à 1. D'autre part, une faute détectée sur une position régulière force un des *R* latches FBI de cette position à 1. Ce latch est sélectionné par la valeur des bits d'adresse A_1, A_2, …, A_r à l'aide d'un multiplexeur comme le montre la Figure 3.15.

3.4.2 Fonctions de Reconfiguration

L'implémentation du circuit de reconfiguration passe essentiellement par la détermination des fonctions de reconfiguration qui génèrent les signaux de contrôle des MUX de la Figure 3.14. Nous pouvons illustrer ce type de reconfiguration par la Figure 3.16. La reconfiguration est similaire à celle de la Figure 3.13. Cependant, dans la Figure 3.16, les blocs de logique de reconfiguration partagent quelques entrées venant des latches FBI

correspondants aux positions redondantes. Aussi, dans la Figure 3.13, les blocs de logique de reconfiguration n'échangent aucune information entre eux (ils implémentent des fonctions indépendantes les unes des autres). Dans la Figure 3.16, les blocs de logique de reconfiguration implémentent des fonctions interdépendantes qui échangent des informations.

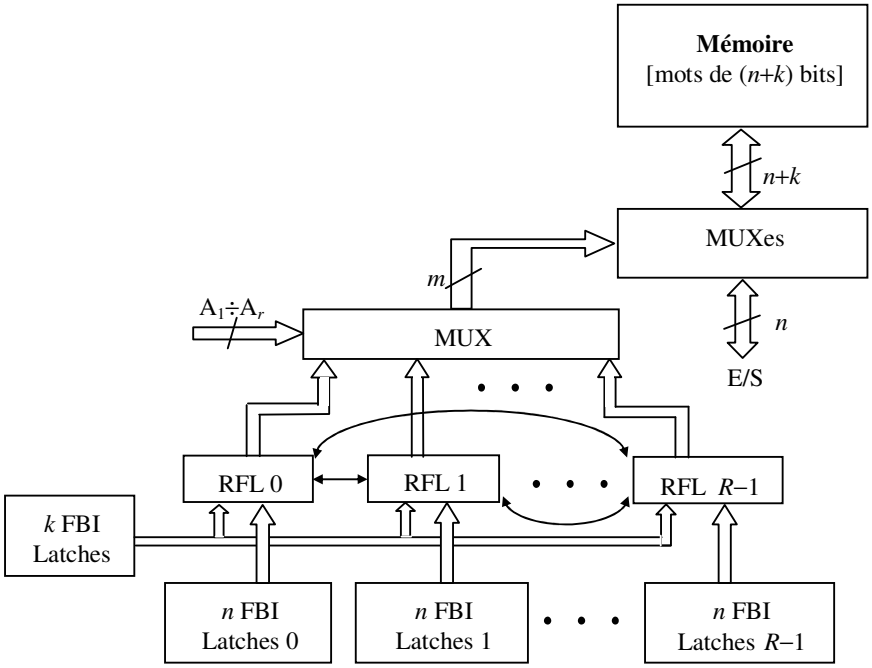


Figure 3.16. Interaction des fonctions de reconfiguration dans la reconfiguration dynamique utilisant des colonnes redondantes singulières.

Ces interdépendances donnent lieu à des fonctions très complexes. Pour simplifier l'analyse nous introduirons quelques variables intermédiaires. Une fois que ces variables sont déterminées, nous pourrons utiliser les équations élaborées dans les sections précédentes. Soit FB_i l'état du latche de la position redondante i ($n \le i \le n+k-1$) et FB_i^q l'état du latche FBI_i correspondant à la colonne q ($0 \le q \le R-1$) et de position régulière i ($0 \le i \le n-1$). Soit $M_j^{i,q}$ le signal de contrôle du MUX de position i qui indique par 1 si la colonne q est décalée de j positions. La variable $M_j^{i,q}$ est calculée par le bloc de logique de reconfiguration RFL_q ($\forall\, j \in \{0, 1, \ldots, k\}$, $\forall\, i \in \{0, 1, \ldots, n-1\}$). L'introduction des variables intermédiaires ne concerne que les positions redondantes. En effet, il s'agit de générer à partir des états FB_i, des variables FB_i^q

($n \le i \le n+k-1$) reflétant l'état de la colonne redondante i par rapport à un groupe de colonnes régulières (toutes de rang q). Suivant l'état de cette variable intermédiaire FB_i^q, la colonne

redondante i va être utilisée ou pas par le groupe de colonnes régulières de rang q, afin d'effectuer la réparation. La colonne redondante i est déclarée non utilisable dans deux cas. Si elle est fautive ($FB_i = 1$). Ou si elle est déjà occupée par une des colonnes de rang inférieur à q appartenant à une des k positions régulières voisines. Nous pouvons formaliser ces deux conditions sous la forme suivante :

$$\textbf{(3.15)}\quad FB_i^q = FB_i + \sum_{x=i-k}^{n-1} \sum_{\substack{y=0 \\ y\neq q}}^{q-1} M_{i-x}^{x,y}\ ,\ \forall\ i : n \le i \le n+k-1,\ \forall\ q : 0 \le q \le R-1.$$

Pour les positions régulières les variables FB_i^q demeurent inchangées. Nous pouvons alors utiliser directement les variables FB_i^q dans les équations établies dans le cas de la reconfiguration statique (équations (3.5) et (3.6)), pour obtenir les fonctions de reconfiguration dynamique utilisant des colonnes comme blocs redondants. Pour toutes les colonnes de position 0, on peut écrire :

$$\textbf{(3.16)}\begin{cases} M_0^{0,q} = \overline{FB_0^q}\ , \\ M_1^{0,q} = \overline{FB_1^q} \cdot FB_0^q\ , \\ \ldots, \\ M_k^{0,q} = \overline{FB_k^q} \cdot FB_{k-1}^q \cdots FB_0^q\ ,\ \forall\ q : 0 \le q \le R-1. \end{cases}$$

Pour les autres positions régulières, nous obtenons :

$$M_j^{i+1,q} = \overline{FB_{i+j+1}^q} \cdot \left(M_j^{i,q} + M_{j-1}^{i,q} \cdot FB_{i+j}^q + M_{j-2}^{i,q} \cdot FB_{i+j-1}^q \cdot FB_{i+j}^q + \cdots + M_0^{i,q} \cdot FB_{i+1}^q \cdot FB_{i+2}^q \cdots FB_{i+j}^q \right),$$

$$\forall\ i : 0 \le i \le n-2,\ \forall\ j : 0 \le j \le k,\ \forall\ q : 0 \le q \le R-1\ \ldots\ \textbf{(3.17)}$$

Les équations ci-dessus peuvent aisément être étendues aux cas de blocs redondants composés d'un jeu complet de colonnes plus des blocs redondants composés de colonnes singulières. Dans ce cas, nous utiliserons pour les positions redondantes à une seule colonne les variables données par l'équation (3.15). Les blocs redondants restants ne nécessitent pas de transformation sur leurs variables d'état FB_i^q.

Notons que la reconfiguration dynamique utilisant des colonnes singulières comme blocs redondants introduit une asymétrie dans le layout de la mémoire. En effet, dans le cas de la reconfiguration dynamique de la section précédente, on utilise des blocs redondants exactement similaires aux blocs réguliers. De ce fait, le même décodeur d'adresse est utilisé

pour tous les blocs durant les opérations d'écriture et de lecture. L'asymétrie dans le cas présent vient du fait que les blocs redondants sont différents des blocs réguliers (colonnes), et que par conséquent aucun décodage colonne n'existe pour ces positions redondantes. Cette situation peut donner lieu à des disfonctionnements lors du fonctionnement normal de la mémoire même après une réussite du processus de réparation. En effet, sans décodage colonne, les blocs redondants vont être accédés à chaque cycle d'écriture. Les données vont donc s'écraser et l'information sera perdue. Pour remédier à ce problème, il faut en plus de la détermination des connections par le calcul des signaux $M_j^{i,q}$, utiliser ces mêmes signaux pour le contrôle des amplificateurs d'écriture correspondants aux colonnes redondantes. Les amplificateurs attaqueront les colonnes redondantes seulement si ces dernières sont sélectionnées ($M_j^{i,q} = 1$) pour remplacer des colonnes régulières fautives.

3.5 Reconfiguration Dynamique Utilisant des Blocs Redondants Composés d'un Groupe de Colonnes Singulières

Dans les deux sections précédentes nous avons présenté deux techniques de reconfiguration dynamique. La première, divise par un facteur R la taille de chaque unité (régulière ou redondante) connectée à un bit de donnée entrée/sortie. Ainsi, les R groupes connectés chacun à $n+k$ bits de données se reconfigurent indépendamment les uns des autres pour générer les n bits de données représentant le résultat de la réparation. Ensuite nous avons présenté une autre technique de reconfiguration dynamique (section précédente) dans laquelle on divise les unités régulières en R groupes de telle manière que chaque groupe régulier soit composé d'une colonne singulière. D'un autre coté, chaque position redondante est composée d'une seule colonne singulière. Ainsi, tous les groupes réguliers (formés chacun d'une colonne singulière), se partagent les colonnes singulières redondantes pour effectuer la réparation. Le meilleur compromis entre la taille de l'unité réparable et le coût de la logique de reconfiguration est une technique intermédiaire entre les deux qui sont décrites ci-dessus. Dans une telle technique, les R groupes réguliers ne seront plus chacun composés d'une seule colonne singulière mais de P colonnes singulières. Chaque sous-groupe de P colonnes utilise k colonnes singulières redondantes. Chacun des R sous-groupes se reconfigure indépendamment des autres, mais dans chaque sous-groupe les $n \times P$ colonnes se partagent k colonnes singulières redondantes exactement comme dans la section précédente.

Pour illustrer cette approche, considérons un multiplexage colonne de 1 parmi 8, et trois blocs réguliers ($n = 3$). Chaque bloc régulier générant un bit de donnée entrée/sortie est donc

composé de 8 colonnes. Ajoutons à cela deux unités redondantes ($k = 2$), chacune composée de 4 colonnes singulières. Le décodeur colonne pour les blocs réguliers utilise 3 bits d'adresses, tandis que le décodeur colonne pour les blocs redondants utilise un sous ensemble de ces bits d'adresses (2 dans cet exemple). Soit A_1, A_2, …, A_r les bits d'adresse utilisés par le décodeur colonne pour les blocs réguliers, et A_1, A_2, …, A_p ($p < r$) les bits d'adresse utilisés par le décodeur colonne pour les blocs redondants. Une valeur donnée des bits A_1, A_2, …, A_p sélectionne la même colonne dans les blocs redondants quelque soit la valeur des bits d'adresses A_{p+1}, A_{p+2}, …, A_r. La sélection de la même colonne redondante par différentes valeurs de certains bits d'adresse donne lieu à des interdépendances dans les fonctions de reconfiguration. Comme précédemment, nous exprimons ces interdépendances en introduisant des variables intermédiaires. Les équations (3.16) et (3.17) restent inchangées. Il nous faut modifier l'équation (3.15) pour l'adapter au cas présent. Considérons le jeu Ra de colonnes appartenant aux blocs redondants et sélectionnées par une valeur des bits d'adresse A_1, A_2, …, A_p. Le jeu Fa de colonnes fonctionnelles sélectionnées par une valeur A_1, A_2, …, A_p différente de Ra ne peut utiliser le jeu de colonnes Ra pour une reconfiguration quelque soit la valeur de A_{p+1}, A_{p+2}, …, A_r. D'autre part, le jeu Fa de colonnes des blocs réguliers sélectionnées par une valeur des bits A_1, A_2, …, A_p et une quelconque valeur de A_{p+1}, A_{p+2}, …, A_r peut être reconfiguré avec les colonnes du jeu Ra. Comme les colonnes du jeu Fa sélectionnées par les bits d'adresses A_{p+1}, A_{p+2}, …, A_r peuvent être reconfigurées avec les mêmes colonnes, nous avons des interdépendances entre les blocs de reconfiguration correspondants à la même valeur de A_1, A_2, …, A_p et à différentes valeurs de A_{p+1}, A_{p+2}, …, A_r. Pour exprimer ces interdépendances, il nous faut modifier l'équation (3.15). Soit $FB_i^{(q1-q2)}$ l'état du latche FBI correspondant à la colonne régulière de position i ($0 \le i \le n-1$) sélectionnée par la valeur $q1$ de A_1, A_2, …, A_p et la valeur $q2$ de A_{p+1}, A_{p+2}, …, A_r. Soit FB_i^{q1} l'état du latche utilisé pour la colonne redondante de position i ($n \le i \le n+k-1$) sélectionnée par la valeur $q1$ de A_1, A_2, …, A_p. Soit $M_j^{i,(q1-q2)}$ la variable de contrôle du MUX de position i, indiquant si la colonne de position i déterminée par la valeur $q1$ de A_1, A_2, …, A_p et la valeur $q2$ de A_{p+1}, A_{p+2}, …, A_r doit être décalée de j positions. Nous pouvons alors exprimer les variables $FB_i^{(q1-q2)}$ de la manière suivante :

(3.18) $$FB_i^{(q1-q2)} = FB_i^{q1} + \sum_{x=i-k}^{n-1} \sum_{\substack{y=0 \\ y \neq q2}}^{q2-1} M_{i-x}^{x,(q1-y)},$$

$\forall\, i : n \le i \le n+k-1$, $\forall\, q1 : 0 \le q1 \le P1$, $\forall\, q2 : 0 \le q2 \le R-1$.

Les équations (3.16) et (3.17) deviennent :

(3.19)
$$\begin{cases} M_0^{0,(q1-q2)} = \overline{FB_0^{(q1-q2)}}, \\ M_1^{0,(q1-q2)} = \overline{FB_1^{(q1-q2)}} \cdot FB_0^{(q1-q2)}, \\ \ldots, \\ M_k^{0,(q1-q2)} = \overline{FB_k^{(q1-q2)}} \cdot FB_{k-1}^{(q1-q2)} \cdots FB_0^{(q1-q2)} \end{cases}$$

$$M_j^{i+1,(q1-q2)} = \overline{FB_{i+j+1}^{(q1-q2)}} \cdot \left(\begin{array}{l} M_j^{i,(q1-q2)} + M_{j-1}^{i,(q1-q2)} \cdot FB_{i+j}^{(q1-q2)} + M_{j-2}^{i,(q1-q2)} \cdot FB_{i+j-1}^{(q1-q2)} \cdot FB_{i+j}^{(q1-q2)} \\ + \cdots + M_0^{i,(q1-q2)} \cdot FB_{i+1}^{(q1-q2)} \cdot FB_{i+2}^{(q1-q2)} \cdots FB_{i+j}^{(q1-q2)} \end{array} \right),$$

$\forall\, i : 0 \le i \le n-2$, $\forall\, j : 0 \le j \le k$, $\forall\, q1 : 0 \le q1 \le P-1$, $\forall\, q2 : 0 \le q2 \le R-1 \ldots$ **(3.20)**

La Figure 3.17 montre l'implémentation de cette nouvelle approche de réparation. $P = 2^p$ copies des latches FBI de la Figure 3.15, sont utilisées pour les positions redondantes, et P copies des R groupes de latches FBI de la même figure sont également utilisées pour les unités régulières. Aussi, nous utilisons P copies des R blocs de fonctions de reconfiguration montrés dans la Figure 3.17 et décrits par les équations (3.18), (3.19) et (3.20). Au total les $P \times R$ blocs de fonctions de reconfiguration donnent lieu à $m \times P \times R$ signaux de sortie. Un premier multiplexeur contrôlé par les bits d'adresses A_{p+1}, A_{p+2}, …, A_r réduit ces signaux en P copies de m signaux. Un second multiplexeur contrôlé par les bits d'adresses A_1, A_2, …, A_p réduit ces $P \times m$ signaux en m signaux utilisés pour le contrôle des multiplexeurs MUX responsables de reconfigurer les unités régulières et redondantes. Notons que les deux multiplexeurs, contrôlés par les bits d'adresses A_{p+1}, A_{p+2}, …, A_r et A_1, A_2, …, A_p, peuvent être remplacés par un seul multiplexeur contrôlé par les bits d'adresses $A_1 \div A_r$.

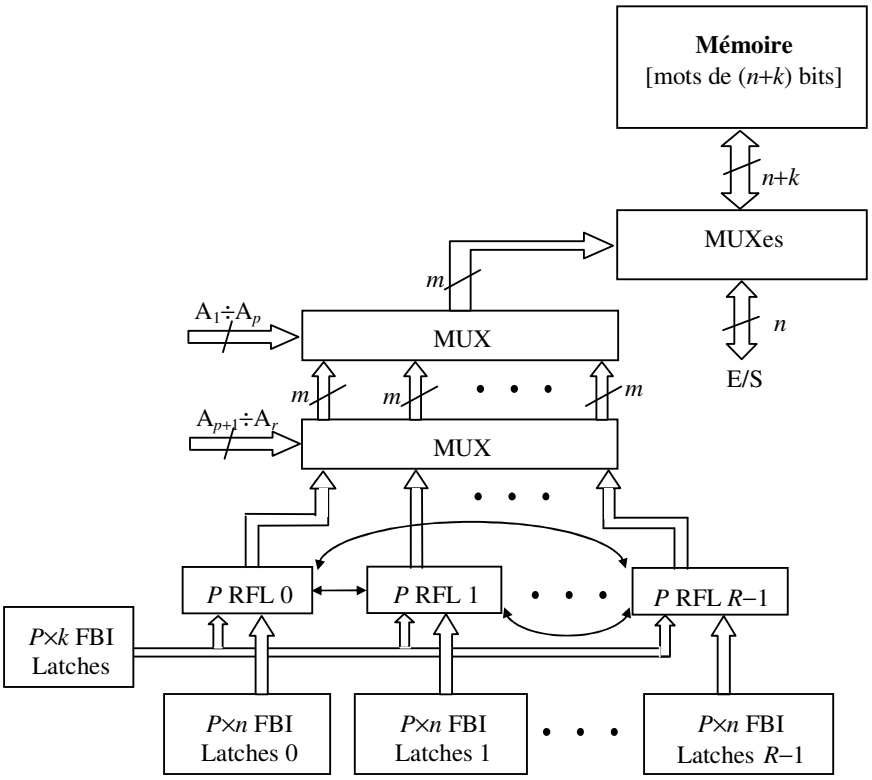


Figure 3.18. Interaction des fonctions de reconfiguration dans la reconfiguration dynamique utilisant des colonnes redondantes.

Conclusion

Nous avons présenté dans ce chapitre, une nouvelle approche d'auto réparation intégrée pour les mémoires au niveau bit de donnée entrée/sortie. Cette approche permet le remplacement de l'ensemble du bloc responsable de la génération d'une donnée entrée/sortie incluant l'amplificateur de lecture/écriture, le multiplexeur colonne qui sont des parties sensibles fréquemment touchées par des fautes. Ainsi, les fautes affectant les cellules mémoires et la logique de lecture/écriture peuvent être réparées. Puisque nous nous intéressons aux grands nombres de fautes, l'hypothèse traditionnelle dans les techniques de tolérance aux fautes et qui veut que les fautes surviennent sur les parties régulières ou sur les parties redondantes, n'est pas adoptée ici. Les techniques proposées tiennent compte à la fois des fautes dans les parties régulières et les fautes dans les parties redondantes. Les fonctions de reconfiguration pour de telles techniques sont très complexes. Par conséquent, l'élaboration des équations de reconfiguration devient une tache difficile et peut donner lieu à une implémentation au coût matériel élevé. Afin de simplifier l'analyse et réduire le coût

matériel, nous avons adopté une approche récursive pour établir ces équations de reconfiguration. Cela permet d'obtenir une logique régulière, compact et peu coûteuse.

Les circuits BISR développés effectuent la réparation par l'unique accès aux entrées et aux sorties de la mémoire. Ceci est très important, étant donné que la mémoire est d'une structure dense et régulière, et que l'intégration des fonctions de reconfiguration à l'intérieure de celle-ci est difficile. D'un autre coté, le fait qu'on accède à la mémoire depuis l'extérieur, ne permet pas de sélectionner la taille de l'unité réparable. Pour réduire cette taille et ainsi améliorer l'efficacité de la réparation, nous avons développé une technique de reconfiguration qui décale dynamiquement les connections des données entrée/sortie. Dans ce cas une connexion d'une donnée entrée/sortie vers un autre bloc de mémoire seulement lorsqu'on accède à une unité réparable fautive. De cette manière, n'importe quelle taille de l'unité réparable peut être sélectionnée tout en permettant un compromis entre le coût des unités redondantes et le coût des fonctions de reconfiguration. La reconfiguration à l'aide d'une telle approche à certaines limitations. Pour y remédier, nous avons développé une nouvelle approche qui consiste à utiliser des colonnes singulières comme unités redondantes au lieu d'un jeu de colonnes singulières constituant un bloc régulier. Cette technique permet d'obtenir des efficacités de réparation encore plus élevées. Elle introduit cependant des interactions complexes entre les fonctions de reconfiguration. Pour maîtriser cette complexité, nous avons introduit des variables intermédiaires pour exprimer ces interactions. En remplaçant des variables dans les fonctions de reconfiguration de base, on obtient les nouvelles équations. Bien que cette technique soit très efficace, elle peut être sujette à une limitation pratique importante. En effet, la plupart des compilateurs mémoire existants ne génèrent pas ce type de redondance basée sur les colonnes singulières, redondance qui introduit une brisure de symétrie dans le design de la mémoire.

Les techniques développées dans ce chapitre sont capables de réparer des mémoires affectées par des densités de défauts élevées (jusqu'à deux ordres de grandeur plus élevées que dans les technologies actuelles) moyennant un coût modéré en silicium. L'étude de l'efficacité de ces techniques, dans le dernier chapitre, confirmera ces capacités de réparation.

CHAPITRE IV. Techniques de Tolérance aux Fautes pour les Grandes Densités de Défauts : Application aux Nanotechnologies

Introduction

Les circuits nanoélectroniques nécessiteront de nouvelles solutions de tolérance aux fautes capables de faire face à une très grande densité de défauts. Les techniques traditionnelles de tolérance aux fautes développées ces dernières années sont destinées à faire face à seulement quelques fautes (typiquement une seule faute par module). Il est donc clair qu'elles ne sont pas adéquates aux situations dans lesquelles un grand nombre de fautes peuvent affecter le circuit. Nous allons dans ce chapitre présenter des techniques de tolérance au fautes adéquates pour les mémoires affectées par une concentration de fautes élevée.

Dans le chapitre précédent, nous avons utilisé les ressources redondantes comme ressources de remplacement. C'est l'approche communément utilisée pour la réparation des mémoires. Ces ressources redondantes remplacent pendant le fonctionnement normal de la mémoire, les ressources fautives détectées pendant la phase de test. Nous avons cependant considéré des densités de fautes plus élevées que celles rencontrées dans les technologies actuelles (plusieurs dizaines à plusieurs centaines de défauts par Mbit au lieu de quelques défauts pour les technologies actuelles) mais moins élevées que les densités de défaut qu'on peut rencontrer dans les nanotechnologies. Par conséquent, les solutions développées dans le chapitre précédent permettent de faire face aux densités de défauts pouvant affecter les mémoires appartenant à plusieurs futures générations du processus technologique incluant le processus technologique actuel. Pour prendre en compte des densités de défauts similaires à celles qu'on pourrait rencontrer dans les nanotechnologies (quelques milliers de défauts par Mbit), nous avons dans cette thèse exploré et développé deux approches de réparation. Dans la première, la réparation est effectuée en combinant deux unités fautives, au lieu de remplacer une unité fautive, par une unité correcte. La seconde approche est basée sur la réparation diversifiée dans laquelle deux (ou plus) techniques de réparation sont utilisées de façon séquentielle afin de réparer l'ensemble des défauts.

4.1 La Réparation Basée sur les Polarités d'Erreurs

Les approches statiques et dynamiques développées dans la section précédente, utilisent une unité correcte pour remplacer une unité défaillante (réparation statique), ou une partie correcte d'une unité redondante pour remplacer une partie fautive d'une unité régulière. L'approche statique développée dans le chapitre précédent est innovante par le fait qu'on considère que les unités régulières ainsi que les redondantes peuvent être fautives et aussi par le fait qu'elle utilise des fonctions de reconfiguration optimales. De plus, elle utilise une seule session de test pour détecter et réparer toutes les fautes au lieu de la répétition des sessions de test après chaque faute détectée et corrigée [KIM98]. L'approche de réparation dynamique est innovante par le fait qu'on puisse effectuer la réparation en utilisant une plus petite unité réparable que les unités régulières ou redondantes. Nous avons montré expérimentalement (chapitre V) que ces approches sont efficaces pour des densités de défauts un à deux ordres de grandeurs plus élevés que les densités de défauts rencontrées dans les technologies actuelles. Malgré ces performances, ces approches ne sont pas suffisantes pour faire face à des densités de défauts qui devraient émerger dans les nanotechnologies. Afin d'améliorer les capacités de réparation, nous avons développé une approche permettant de réparer une unité régulière fautive par une unité redondante fautive. L'idée est d'exploiter le fait que dans une unité régulière ou redondante fautive il y a seulement quelques cellules mémoires qui sont fautives, donc si nous utilisons les cellules non fautives de chacune des unités nous pourrions effectuer la réparation avec succès. Cependant, effectuer la réparation en remplaçant individuellement les cellules fautives, donne lieu à un circuit de reconfiguration très complexe au coût très élevé. Pour éviter une telle opération nous exploitons les polarités des erreurs produites aux sorties des unités fautives pour combiner ces unités en utilisant des fonctions de reconfiguration qui vont masquer ces polarités d'erreurs.

4.1.1 Réparation pour une Distribution Equilibrée des Polarités d'Erreurs

Une cellule fautive produit des erreurs d'une polarité donnée pour la majorité des types de fautes (voir section 1.3.1.2) : collage à 0 ou collage à 1, fautes de transition (cellule qui ne peut effectuer la transition $0 \rightarrow 1$ ou $1 \rightarrow 0$), fautes de couplage (l'état ou la transition d'une cellule d'un état a un autre modifie la valeur d'une autres cellule de 0 à 1 ou de 1 à 0), fautes statiques et dynamiques sensibles au voisinage (l'état d'un ensemble de cellules combiné ou

non à la transition d'une autre cellule modifie l'état d'une tierce cellule de 1 à 0 ou de 0 à 1 ou l'empêche de faire la transition $0 \rightarrow 1$ ou $1 \rightarrow 0$).

Pour ce qui suit, nous dirons qu'une cellule mémoire est touchée par une polarité d'erreur $x \rightarrow y$, lorsqu'au cours d'une opération de lecture de cette cellule, on obtient la valeur x au lieu de la valeur y, et ce quelles qu'en soient les raisons.

Dans une mémoire utilisant des unités réparables régulières et redondantes, lors d'une reconfiguration dynamique, d'une taille de 128 cellules avec une densité de défauts de 0,5 %, alors, la majorité de ces unités va contenir une faute, quelques unes d'entre elles vont contenir plus d'une faute et quelques unes aussi vont être correctes. Nous pouvons alors classifier les différentes unités régulières et redondantes dans quatre catégories :

- La catégorie 01 génère des cellules fautives, quelques unes de ces cellules créent des polarités d'erreurs $0 \rightarrow 1$, d'autres créent des polarités d'erreurs $1 \rightarrow 0$ et enfin des cellules qui créent les deux polarités d'erreurs $0 \rightarrow 1$ et $1 \rightarrow 0$. Une cellule peut effectivement créer les deux types d'erreurs si elle est touchée d'un rare défaut ayant un effet bidirectionnel (défaut double).
- La catégorie 0 contient une ou plusieurs cellules qui génèrent des erreurs de polarité $1 \rightarrow 0$ seulement.
- La catégorie 1 contient une ou plusieurs cellules qui génèrent des erreurs de polarité $0 \rightarrow 1$ seulement.
- La catégorie 00 non fautive contient que des cellules non fautives.

Nous allons exposer dans les sections suivantes la façon avec laquelle on va exploiter cette classification des unités réparables, basée sur les polarités d'erreurs, pour reconfigurer la mémoire au niveau bit de donnée.

4.1.1.1 Masquage des polarités d'erreurs

En se basant sur la classification ci-dessus, nous utiliserons les parties redondantes de catégorie 0 pour réparer les parties régulières de la catégorie 0 et les partie redondantes de catégorie 1 pour réparer les parties régulières de la catégorie 1. Pour ce faire, au lieu de remplacer une partie régulière fautive par une partie redondante non fautive, nous utilisons la fonction OU pour combiner les données en sortie de deux unités fautives de catégorie 0 et une fonction ET pour combiner les données en sortie de deux unités fautives de catégorie 1. Cette

combinaison permet de réparer les deux parties à la seule condition que les deux parties n'aient pas des cellules fautives à la même position. Dans ce cas, la fonction OU combinera toujours les deux bits de données en sortie issus de deux cellules correctes ou bien d'une cellule contenant un 1 correct et un 0 erroné. Donc le résultat sera toujours un bit de donnée correct. Les mêmes remarques sont valables pour la fonction ET qui combine deux parties de catégorie 1. Les parties régulières de catégorie 01 seront réparées par des unités redondantes non fautives non pas par combinaison mais par remplacement.

Si pour quelques unités régulières de catégorie 0 ou 1, il n'y a plus de parties redondantes de la même catégorie, nous pouvons alors utiliser des parties redondantes non fautives pour les réparer. Pour ce faire, nous pouvons d'abord réparer les parties de catégorie 0 et 1 et ensuite répéter le test une seconde fois. Durant ce test, toutes les parties non réparées lors de la première phase de réparation, seront déclarées comme appartenant à la catégorie 01 et seront par conséquent réparées par des unités redondantes non fautives. Ce test permettra la conversion des parties régulières de catégorie 0 ou 1 à la catégorie 01 si ces parties ont eu avec les parties redondantes des cellules fautives à la même position.

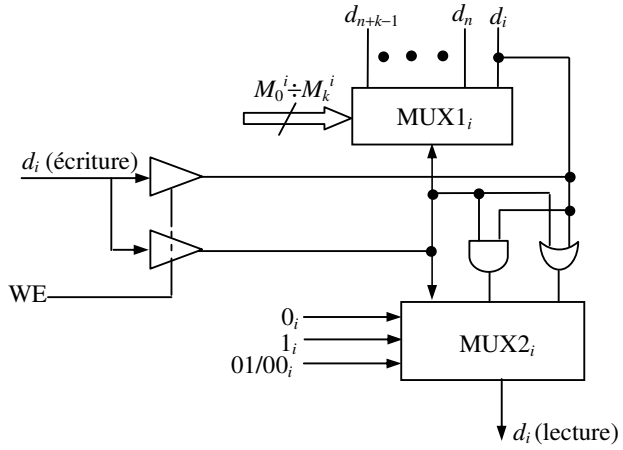


Figure 4.2. Circuit de sélection dans la réparation par combinaison.

La Figure 4.2 montre le circuit utilisé pour combiner les unités régulières avec les unités redondantes comme il a été décrit plus haut. Le multiplexeur MUX1 de position *i* (MUX1i) est utilisé pour sélectionner l'unité redondante employée pour réparer l'unité régulière de position *i*. L'opération de ce MUX est identique à celle des MUX utilisés pour les différentes reconfigurations du chapitre précédent. La circuiterie additionnelle est utilisée pour combiner la sortie de l'unité redondante sélectionnée avec la sortie de l'unité régulière de position *i* pour créer un bit de donnée correct à cette position. Cette combinaison est contrôlée par les signaux

0_i et 1_i, qui déterminent si la position *i* est de catégorie 0 ou 1, et par le signal $01/00_i$ qui détermine si la position *i* est de catégorie 01 ou bien 00 (non fautive).
L'écriture des données doit se faire sur l'unité redondante sélectionnée pour la réparation mais aussi sur l'unité fautive de position régulière *i* pour éviter toute perte de l'information. L'opération d'écriture peut bien sûr être effectuée à l'aide de deux portes trois états contrôlées par le signal "Write Enable" (WE), voir Figure 4.2.

4.1.1.2 Détection et localisation des erreurs suivant leurs polarités

Pour rendre opérationnel le circuit de la Figure 4.2, nous devons d'abord déterminer les signaux 0_i, 1_i, et $01/00_i$ puis les utiliser pour le calcul des signaux M_j^i. Pour générer les signaux 0_i, 1_i, et $01/00_i$ nous devons disposer de deux latches pour stocker les résultats de test pour chaque unité régulière et redondante au lieu d'un seul latche utilisé dans le chapitre précédent. Soit F_i et D_i les états de ces latches, soit d_i la donnée lue depuis la position *i* et X_i la sortie du XOR de position *i* dans le comparateur du BIST. Les variables F_i et D_i, pour les positions redondantes sont à chaque cycle de la phase de test, exprimées comme suit :

(4.1) $F_i = \left(\overline{X_i}.F_i + X_i.\overline{F_i}.\overline{D_i} + X_i.F_i.(d_i \oplus D_i)\right).\overline{T_2} + F_i.T_2$,

(4.2) $D_i = X_i.d_i + D_i$.

Pour les positions régulières nous obtenons :

(4.3) $F_i = \left(\overline{X_i}.F_i + X_i.\overline{F_i}.\overline{D_i} + X_i.F_i.(d_i \oplus D_i)\right).\left(\overline{T_2} + \overline{X_i}\right)$

(4.4) $D_i = X_i.d_i + D_i + T_2.X_i$

La variable supplémentaire *T*2 utilisée dans ces équations prend la valeur 1 durant la seconde phase de test. Elle est employée pour transformer les unités régulières, non réparées pendant la première phase de test et de réparation, en unités de catégories 01. Durant la seconde phase de test, les variables F_i et D_i sont bloquées à leurs valeurs obtenues pendant la première phase de test. A partir des équations ci-dessus, la concaténation des variables F_i et D_i détermine les différentes catégories :

F_iD_i = 00 pour la catégorie 00,

F_iD_i = 10 pour la catégorie 0,

F_iD_i = 11 pour la catégorie 1,

F_iD_i = 01 pour la catégorie 01.

Les signaux 0_i, 1_i, et $01/00_i$ de la Figure 4.2 sont donc obtenus de la manière suivante :

$0_i = F_i.\overline{D_i}$, $1_i = F_i.D_i$, $01/00_i = \overline{F_i}$, $01_i = \overline{F_i}.D_i$, $00_i = \overline{F_i}.\overline{D_i}$.

Pour différencier les positions régulières des positions redondantes, nous allons utiliser pour les positions régulières de position *i* les signaux $0R_i$, $1R_i$, $01R_i$, et $00R_i$ à la place des signaux 0_i, 1_i, 01_i et 00_i. Pour les positions redondantes de position *i* nous utiliserons les signaux $0S_i$, $1S_i$, $01S_i$, et $00S_i$.

4.1.1.3 Fonctions de reconfiguration

Pour effectuer la réparation par combinaison, de nouvelles fonctions de reconfiguration doivent être élaborées. Nous allons commencer par l'approche statique et prendre pour l'exemple le cas de la réparation distante. Nous devons alors établir les équations régissant les signaux M_0^i, M_1^i, …, M_k^i et qui contrôlent le MUX1$_i$ de la Figure 4.2 de position régulière *i*. Pour ce faire nous pouvons reprendre les fonctions de reconfiguration développées dans le chapitre précédent pour la réparation distante. Pour rappel, nous donnons ci-dessous les expressions de ces fonctions :

(4.5) $M_0^0 = \overline{RF_0}$, $M_1^0 = \overline{SF_1}.RF_0$, …, $M_k^0 = \overline{SF_k}.SF_{k-1} \ldots SF_1.RF_0$

(4.6) $F_j^0 = M_j^0, \forall\, j \in \{0, 1, \ldots, k\}$.

(4.7) $F_j^{i+1} = F_j^i.\overline{RF_{i+1}} + M_j^{i+1}.RF_{i+1}, 0 \le i \le n-2, 0 \le j \le k.$

(4.8)
$$\begin{cases} M_0^{i+1} = \overline{RF_{i+1}}, \\ M_{j+1}^{i+1} = \overline{SF_{j+1}}.RF_{i+1}.\left(F_j^i + F_{j-1}^i.SF_j + F_{j-2}^i.SF_{j-1}.SF_j + \ldots + F_0^i.SF_1.SF_2 \ldots SF_j\right), \\ 0 \le j \le k-1, 0 \le i \le n-2. \end{cases}$$

Ces équations considèrent seulement une seule catégorie de fautes alors que l'approche développée ici utilise trois catégories d'unités fautives. Donc pour établir les nouvelles fonctions de reconfiguration, nous pouvons implémenter ces équations pour chaque catégorie de fautes. La difficulté réside dans le fait qu'on ne connaît pas à l'avance la catégorie de telle ou telle partie de la mémoire. Pour remédier à ce problème nous allons utiliser les signaux $0R_i$, $1R_i$, $01R_i$, $00R_i$ et $0S_i$, $1S_i$, $01S_i$, $00S_i$ à la place des signaux *RF* et *SF* afin de générer les signaux de contrôle des MUX1$_i$ en considérant uniquement les positions régulières et redondantes de la même catégorie que celle de la position *i*. En effet, lorsque une position *i* est de catégorie 0, les nouvelles fonctions de reconfiguration qui calculent les signaux de contrôle

du MUX1$_i$, considèrent les autres unités régulières de catégorie 0 comme fautives, les unités régulières de catégorie différente de 0 comme non fautives, les unités redondantes de catégorie 0 comme correctes et les unités redondantes de catégorie différente de 0 comme fautives. Ainsi, si l'unité régulière de position *i* est de catégorie 0, le circuit BISR utilisera uniquement les unités redondantes de catégorie 0 pour réparer la position *i*.

Les équations pour les signaux de contrôle du MUX1 de position 0 sont les suivantes :

(4.9)
$$\begin{cases} M_0^0 = 00R_0, \\ M_1^0 = 0S_1.0R_0 + 1S_1.1R_0 + 00S_1.01R_0, \\ \ldots, \\ M_k^0 = 0S_k.\overline{0S_{k-1}}\ldots\overline{0S_1}.0R_0 + 1S_k.\overline{1S_{k-1}}\ldots\overline{1S_1}.1R_0 + 00S_k.\overline{00S_{k-1}}\ldots\overline{00S_1}.01R_0. \end{cases}$$

Pour déterminer les équations régissant le calcul des signaux M_j^i ($i > 0$) pour une position arbitraire *i* il nous faut déterminer la première unité redondante non utilisée et de même catégorie que l'unité régulière de position *i*. Dans le chapitre précédent, nous avons introduit la variable intermédiaire F_j^i qui donne l'état de l'unité redondante de position *j* (fautive ou déjà occupée par une unité régulière de position inférieure à *i*). Nous devons alors pour chaque catégorie réécrire les équations donnant les variables intermédiaires $0F_j^i$, $1F_j^i$ et $01F_j^i$, en conformité avec les équations (4.6) et (4.7), comme suit :

$$0F_j^0 = 1F_j^0 = 01F_j^0 = M_j^0, \forall j \in \{0, 1, ..., k\},$$

Pour la position *i*+1, la variable $0F_j^{i+1}$ est égale à la variable $0F_j^i$ si l'unité régulière RU$_{i+1}$ est de catégorie 0, sinon elle égale à M_j^{i+1}, nous obtenons alors :

$$0F_j^{i+1} = 0F_j^i.\overline{0R_{i+1}} + M_j^{i+1}.0R_{i+1},\ 0 \le i \le n-2,\ 0 \le j \le k-1.$$

Les équations pour les variables $1F_j^i$ et $01F_j^i$ sont similaires à l'équation ci-dessus.

Aussi, les signaux de contrôle des MUX1$_{i+1}$ sont données par :

(4.10)
$$\begin{cases} M_0^{i+1} = 00R_{i+1}, \\ M_{j+1}^{i+1} = 0S_{j+1}.0R_{i+1}.\left(0F_j^i + 0F_{j-1}^i.\overline{0S_j} + 0F_{j-2}^i.\overline{0S_{j-1}}.\overline{0S_j} + \ldots + 0F_0^i.\overline{0S_1}.\overline{0S_2}\ldots\overline{0S_j}\right) + \\ 1S_{j+1}.1R_{i+1}.\left(1F_j^i + 1F_{j-1}^i.\overline{1S_j} + 1F_{j-2}^i.\overline{1S_{j-1}}.\overline{1S_j} + \ldots + 1F_0^i.\overline{1S_1}.\overline{1S_2}\ldots\overline{1S_j}\right) + \\ 00S_{j+1}.01R_{i+1}.\left(01F_j^i + 01F_{j-1}^i.\overline{00S_j} + 01F_{j-2}^i.\overline{00S_{j-1}}.\overline{00S_j} + \ldots + 01F_0^i.\overline{00S_1}.\overline{00S_2}\ldots\overline{00S_j}\right). \\ 0 \le i \le n-2,\ 0 \le j \le k-1. \end{cases}$$

Dans les équations (4.9) et (4.10) certains termes sont répétés par comparaison aux équations (3.12) et (3.15). Cette répétition correspond aux trois catégories de fautes 0, 1 et 01. Aussi, on

peut remarquer des changements de polarité (inversion) de certains symboles toujours par rapport aux équations (3.12) et (3.15). En effet, dans ces dernières, quand une unité redondante est correcte on a $SF_j = 0$ alors que dans (4.9) et (4.10), une unité redondante est considérée correcte pour la catégorie 0 quand $0S_j = 1$, correcte pour la catégorie 1 quand $1S_j = 1$ et correcte pour la catégorie 01 quand $01S_j = 1$. Enfin, dans les termes de la catégorie 01 nous utilisons l'état des unités redondantes de catégorie 00 c'est a dire $00S_j$ car se sont les seules aptes à réparer les positions régulières de catégorie 01.

Comme il a été discuté plus haut, nous effectuons une première phase de test pour générer les quatre catégories des positions régulières et redondantes. Ensuite, nous procédons à la première phase de réparation pour réparer les unités fautives de catégorie 0, 1, et 01. Or, lors de cette réparation, il se peut que quelques unités régulières de catégorie 0 et 1 n'aient pas suffisamment d'unités redondantes de la même catégorie pour être combinées avec. Il se peut aussi que certaines unités redondantes utilisées pour la combinaison avec des unités régulières de même catégorie aient des cellules à la même position. Nous devons alors à l'aide d'un second test identifier de telles positions régulières et utiliser des unités redondantes correctes, non utilisées pour réparer des unités régulières de catégorie 01 lors de la première phase, pour les remplacer. En effet, les unités redondantes non fautives peuvent remplacer (pas de combinaison) des unités régulières fautives de n'importe quelle catégorie (0, 1 et 01). Pour garder le même principe de réparation après la première phase de test et la seconde phase de test, c'est à dire les catégories 00 réparent les catégories 01, on doit transformer toutes les unités régulières de catégorie 0 et 1 non réparées pendant la première phase de réparation en catégorie 01 à la fin du second test. Ceci est effectué par les équations (a), (b), (c) et (d). Pour ce faire, nous activons la seconde phase de test avec $T2 = 1$, et à la fin nous activons la seconde phase de réparation qui permet de réparer toutes les unités régulières non réparées par la première phase.

4.1.1.4 Approche dynamique

Afin d'améliorer l'efficacité de la réparation tout en contrôlant le surcoût en surface qui pourrait s'ajouter au circuit BISR, nous pouvons appliquer le principe de la réparation dynamique présenté dans le chapitre précédent au cas présent comme le montre la Figure 4.3 et la Figure 4.4.

Pour ce faire, on utilise R blocs de latches F&D. A chaque cycle de la phase de test, les signaux X_i issus du comparateur du circuit BIST, sont capturés par le bloc F&D sélectionné

par la valeur des bits d'adresses A_1, A_2, …,A_r. Ceci est effectué par un multiplexeur MUX contrôlé par ces bits d'adresses. Nous disposons également de R blocs de fonctions de reconfiguration basés sur le principe de combinaison. Chaque bloc a comme entrée le contenu des latches F&D correspondants et calcule ainsi les signaux M_j^i pour le contrôle du $MUX1_i$ qui servira à combiner les bonnes unités entre elles. Les signaux M_j^i sont auparavant sélectionnés par un multiplexeur MUX contrôlé par les bits d'adresse système A_1, A_2, …,A_r. Ainsi chaque groupe se répare indépendamment des autres, et on aura de cette manière réduit chaque unité régulière et redondante de taille Nd en $Nd/R = Nd/2^r$ unités réparables (plus petites). En même temps nous avons multiplié par 2^r le nombre d'unités redondantes.

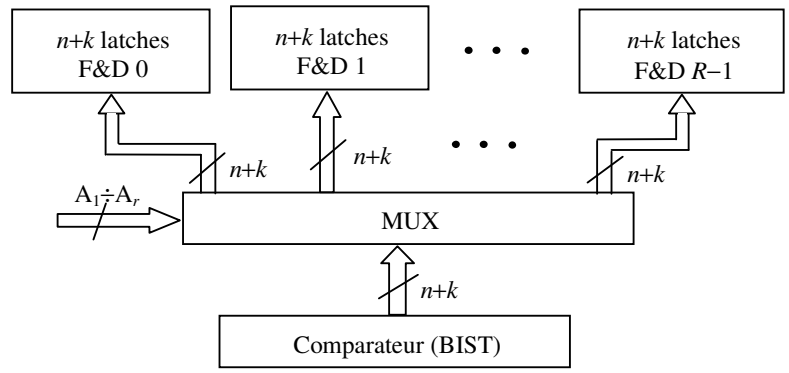


Figure 4.3. Sélection des blocs de latches F&D durant la phase de test.

Il est tout à fait possible d'augmenter encore plus l'efficacité du circuit BISR tout en contrôlant le surcoût en surface dû à la fois à la taille des unités redondantes et à la taille du circuit de reconfiguration. En effet, la solution présentée dans le chapitre précédent utilisant k colonnes redondantes singulières partagées entre P groupes de n unités régulières réparables (chaque unité réparable est composée de R/P colonnes singulières avec $R \geq P$), peut être appliquée au cas présent. Il faut bien sûr étudier les surcoûts induits par de telles techniques de reconfiguration dynamique, car les fonctions de reconfigurations dans ces cas deviennent de plus en plus coûteuses.

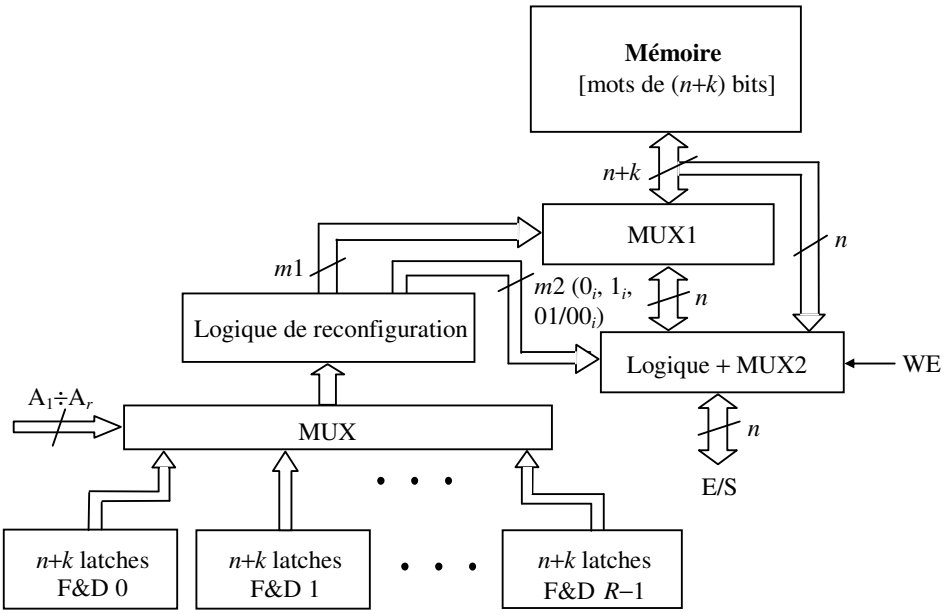


Figure 4.4. Reconfiguration dynamique des données E/S durant le fonctionnement normal.

4.1.2 Fonctions de Reconfiguration Fixes

L'approche présentée ci-dessus améliore les capacités de réparation, mais celle-ci peut se révéler inefficace si la densité de défauts devient très élevée. En effet, dans ce cas la plupart des unités régulières et redondantes vont inclure des fautes avec les deux types de polarités $0 \rightarrow$ et $1 \rightarrow 0$, ce qui va nous ramener aux mêmes limitations de la réparation par remplacement dans la situation où le nombre d'unités régulières fautives est plus grand que le nombre d'unités redondantes. Nous pouvons remédier à ce problème en utilisant de plus petites unités redondantes et ce à l'aide des différentes approches de la réparation dynamique. Cependant, la logique de reconfiguration va devenir très complexe. Dans de telles situations, il est possible de contourner la complexité des fonctions de reconfiguration en employant une augmentation significative des unités redondantes comme le montre la Figure 4.5.

Dans cette figure, au lieu de reconfigurer les unités entre elles pour ensuite effectuer des combinaisons, on dispose de deux paires d'unités mémoires dont les sorties sont combinées pour générer un seul bit de donnée en sortie. Dans la Figure 4.5, le bit de donnée en sortie de chaque paire est généré par la combinaison de deux unités mémoires à l'aide d'une porte OU. La porte OU permet de masquer les éventuelles erreurs de type $1 \rightarrow 0$ à condition que dans chaque paire ces fautes n'occupent pas la même position. Les deux sorties des portes OU sont combinées à l'aide d'une porte ET qui va masquer les fautes de type $0 \rightarrow 1$. Ce masquage est correctement effectué si seulement une seule entrée de la porte OU transporte la faute de type

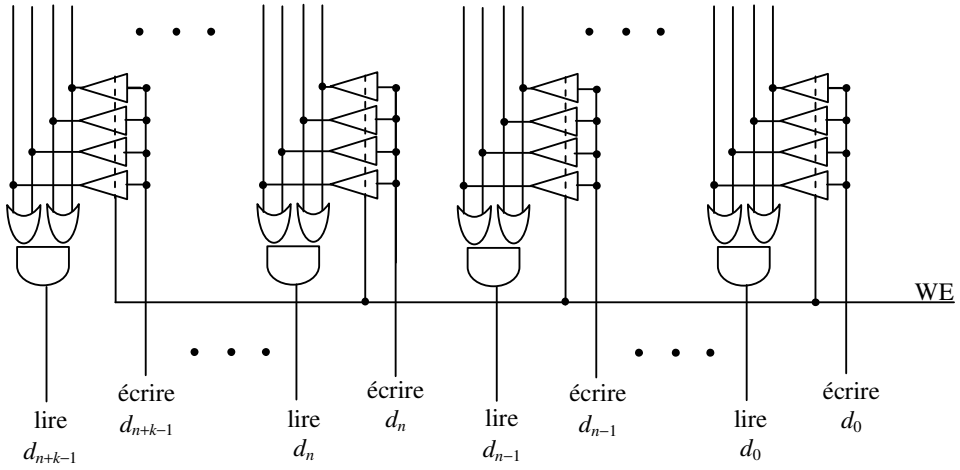


Figure 4.5. Réparation utilisant des fonctions de reconfiguration fixes.

$0 \rightarrow 1$. Cela revient à dire que pour une position (d'une cellule) donnée une seule unité sur les quatre peut comporter une faute de type $0 \rightarrow 1$ pour pouvoir la masquer à l'aide de la porte ET. Trois situations basées sur la probabilité d'occurrence de types de fautes sont donc à considérer afin de maximiser l'efficacité de réparation du circuit BISR :

- Si les fautes de type $1 \rightarrow 0$ sont plus probables que les fautes de type $0 \rightarrow 1$. Le schéma de la Figure 4.5 serait donc le plus efficace.
- Si les fautes de types $0 \rightarrow 1$ sont plus probables que les fautes de type $1 \rightarrow 0$, on utilise un schéma dans lequel on utilise deux portes ET pour masquer les fautes de types $0 \rightarrow 1$ et une porte OU pour masquer les fautes de type $1 \rightarrow 0$.
- Si les deux types deux types de fautes $1 \rightarrow 0$ et $0 \rightarrow 1$ ont la même probabilité d'occurrence, alors un des deux schémas cités plus haut peut être utilisé.

Le schéma de réparation proposé dans la Figure 4.5 (et celui qui utilise deux portes ET et une porte OU), corrige la plupart des cellules fautives donnant lieu à des bits de données en sortie pour la plupart correctes. En effet, dans le cas de densités de défauts élevées, cette technique ne garantit pas la réparation de tous les bits de données fonctionnels pour les diverses raisons de distribution de ces fautes. Pour remédier à ce problème, nous pouvons utiliser k extra unités comme il est montré sur la Figure 4.5. Ces unités sont construites de la même manière que les n unités fonctionnelles. Cela permet d'obtenir un plus grand nombre d'unités réparées pour les utiliser au remplacement des unités qui n'ont pas été réparées à l'aide de la technique

de réparation par combinaison. Le remplacement peut alors être effectué d'une manière statique ou dynamique exactement comme cela a été décrit dans la section 3.3.

4.1.3 Fonctions de Reconfiguration pour une Distribution Déséquilibrée des Fautes

La distribution des types de défauts peut être équilibrée (les défauts créant des erreurs de polarités $0 \rightarrow 1$ et $1 \rightarrow 0$ ont la même probabilité d'occurrence) ou déséquilibrée (la probabilité d'occurrence d'un défaut créant une erreur d'un certain type de polarité est beaucoup plus élevée que les défauts donnant lieu à des erreurs de l'autre type de polarité). Cette situation va dépendre du processus de fabrication et des causes des défaillances. Dans la fabrication des technologies CMOS, les défauts créant des fautes de collages à 1 ou à 0 ont une probabilité du même ordre résultant en une situation équilibrée. Cependant, si nous considérons les softs erreurs dans les DRAM, la probabilité des erreurs de polarité $1 \rightarrow 0$ est beaucoup plus élevée que les erreurs de polarité opposée. Cela est dû au fait qu'une soft erreur est créée quand une particule ionisante heurte le drain d'un transistor au repos. Quand le transistor est de type NMOS, un pulse de courant négatif est crée, alors que pour un transistor PMOS un pulse de courant positif est créé. Or, le transistor lié à la cellule DRAM est un transistor NMOS. Alors, les particules ionisantes créent un pulse de courant négatif sur le drain de ce transistor déchargeant ainsi la capacité de la cellule mémoire et créent une erreur de polarité $1 \rightarrow 0$.

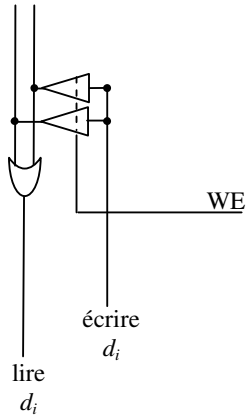


Figure 4.6. Fonctions de reconfiguration fixes pour les fautes déséquilibrées.

Parce qu'on ne peut aujourd'hui avoir des données sur l'analyse des défaillances pour les nanotechnologies, nous ne pouvons pas estimer si les défauts vont être équilibrés ou déséquilibrés. C'est pourquoi nous proposons dans cette section une solution pouvant

améliorer significativement l'efficacité de réparation dans le cas de la distribution déséquilibrée de défauts. Ces distributions déséquilibrées peuvent être considérées dans le cas des cellules mémoires asymétriques (cellules DRAM) ayant des étapes critiques dans la phase de fabrication affectant d'une manière sévère le rendement de production. Cette situation peut donner lieu à une probabilité élevé de défauts créant des erreurs d'une certaine polarité tandis que les erreurs de la seconde polarité ont des probabilités similaires à celles des technologies actuelles (e.g. densité de défauts de 10^{-2} qui créent des erreurs de polarité $1 \rightarrow 0$ et une densité de défauts de 10^{-4} qui créent des erreurs de polarité $0 \rightarrow 1$). Dans ce cas, on peut simplifier le circuit de réparation de la Figure 4.5 en réduisant le nombre d'unités combinées à seulement deux unités combinées à l'aide d'une porte OU pour générer les bits de donnée réguliers et redondants comme le montre la Figure 4.6. Cette combinaison permet de masquer la majorité des défauts créant des erreurs de polarité $1 \rightarrow 0$. Un test peut alors être effectué pour déterminer les positions touchées par des fautes $1 \rightarrow 0$ qui n'ont pu être masquées et/ou les positions touchées par des fautes $0 \rightarrow 1$. Chacun de ces défauts non réparés par combinaison ont de faibles probabilités d'occurrence et peuvent être réparés en utilisant l'approche dynamique développé dans le chapitre III.

4.2 Approches de Réparations Diversifiées

L'idée de base de cette approche consiste en l'utilisation d'une première technique de tolérance aux fautes pour réparer la plupart des défauts, puis l'utilisation d'une seconde technique pour réparer les défauts restants. Une telle approche distribue les ressources redondantes entre plusieurs techniques de tolérance aux fautes. A cause d'une telle distribution, chaque technique de tolérance aux fautes disposera de moins de ressources redondantes pour le même nombre total de ressources redondantes. D'où l'objection : pour quelle raison la distribution des ressources redondantes entre plusieurs techniques de tolérance aux fautes est plus efficace que d'allouer toutes les ressources redondantes à une seule technique de tolérance aux fautes ? La justification est basée sur le fait qu'il faut réparer toutes les fautes affectant une mémoire autrement cette mémoire est déclarée irréparable. Considérons alors une mémoire composée de plusieurs parties. Une densité de défauts donnée résulte en une distribution du nombre de défauts telle que, seulement quelques parties de la mémoire sont affectées par un nombre de défauts beaucoup plus élevé que la plupart des parties de la mémoire. Ces parties sont montrées sur la partie droite de la courbe de distribution des défauts (voir la Figure 4.7). En effet, la grande majorité des parties sera

affectée par un nombre de défauts se trouvant dans la partie centrale de la courbe de distribution des défauts.

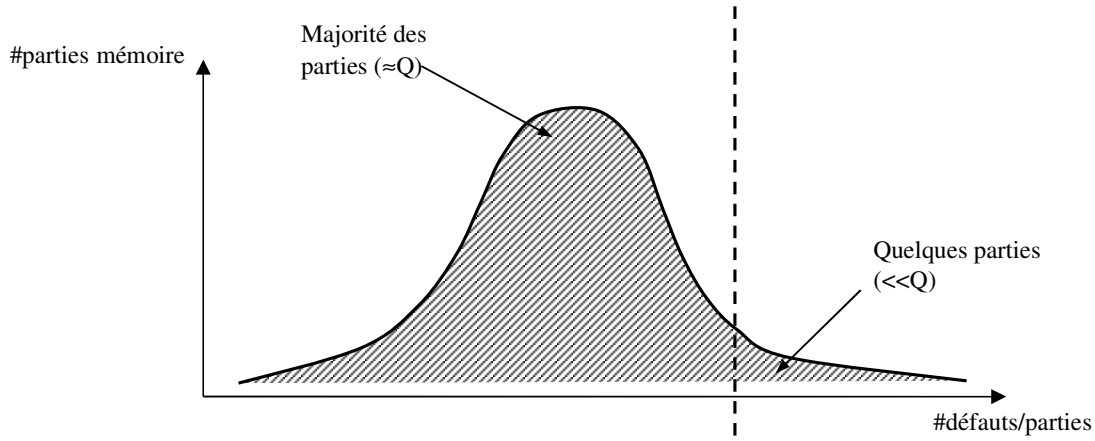


Figure 4.1. Répartition des défauts dans les parties mémoire.

Etant donné qu'on doit réparer toutes les parties de la mémoire et qu'on ne connaît pas à l'avance quelles parties vont être touchées par un grand nombre fautes, nous devons attribuer à chaque partie une redondance suffisante afin d'effectuer la réparation avec succès. La situation devient critique lorsqu'on dispose d'un grand nombre de parties mémoire, cela va augmenter la probabilité d'avoir quelques parties mémoire avec un nombre de défauts plus élevé que celui touchant la plupart des parties. Dans ce cas, il sera plus efficace de disposer pour chaque partie d'un niveau de redondance capable de réparer un nombre de défauts modéré, pour réparer ainsi la majorité des parties, puis ajouter des parties redondantes pour remplacer les quelques parties régulières touchées par un plus grand nombre de fautes. Nous illustrerons ces idées par des exemples tout au long de ce chapitre.

4.2.1 Réparation Diversifiée Combinant la Réparation Dynamique au Niveau Données et la Réparation Bloc

Les techniques de réparation dynamique au niveau données présentées dans le chapitre III, divise la mémoire en 2^r groupes, chacun peut être sélectionné à l'aide d'une valeur donnée des r bits d'adresse. Chaque groupe et composé de 2^{m-r} mots mémoire (m représente la taille du bus d'adresse) et chaque mot mémoire est composé de n bits réguliers et k bits redondants. La réparation dynamique (section 3.3) permet de réparer chaque groupe indépendamment des autres. En effet, cette reconfiguration est équivalente à une réparation statique de chaque

groupe. Ainsi, si k bits redondants sont utilisés et qu'il n'existe pas plus de k bits défectueux dans un groupe alors celui-ci est réparé.

Dans le cas des hautes densités de défauts et d'après la distribution des fautes évoquée sur la Figure 4.1, il est possible que la majorité des 2^r groupes n'aient pas plus de k bits fautifs (donc réparables) et un petit nombre de groupes (exemple : 2 blocs) ayant un nombre de bits fautifs supérieur à k (exemple : k+3). Au lieu d'ajouter trois bits de données redondants aux k déjà existants pour chacun des 2^r groupes, nous ajoutons 2 blocs redondants ayant la même taille que les 2^r groupes (ou blocs) pour les utiliser afin de réparer les deux blocs non réparés par la reconfiguration dynamique au niveau bits de données. Nous pouvons à titre d'exemple effectuer la comparaison du surcoût en surface entre le choix d'ajouter 3 bits redondants supplémentaires et le choix de disposer de 2 blocs redondants. En ajoutant 3 bits redondants à une mémoire possédant des mots de 32 bits fonctionnels et utilisant une réparation dynamique avec $r = 7$, on a une augmentation de la surface de l'ordre de 3/32 = 9,4 % tandis que l'ajout

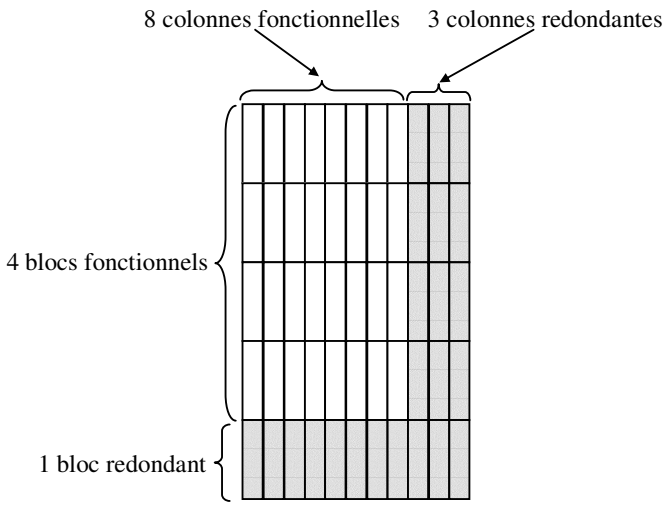


Figure 4.8. Organisation des unités ressources redondantes pour la réparation diversifiée bits de donnée et bloc avec $n = 8$, $k = 3$, $r = 2$ et $q = 1$.

de 2 blocs redondants implique une augmentation de seulement $2/2^7$ = 1,55 %. Cet exemple montre bien l'intérêt de combiner les deux techniques de réparation.

La Figure 4.8 illustre un exemple d'organisation de trois bits redondants et un bloc redondant.

4.2.1.1 La réparation au niveau bloc

Pour effectuer la réparation au niveau bloc, on dispose de q blocs redondants chacun d'eux incluant 2^{m-r} mots mémoires. Nous utilisons r bits d'adresse mémoire comme adresse de chaque bloc. Nous utilisons également une mémoire CAM (Content Adressable Memory) composée de q mots de r bits. Le système de réparation fonctionne de la manière suivante.

Test, réparation et fonctionnement de la mémoire

Durant le test de la mémoire régulière, la détection d'une faute, permet d'écrire dans un mot de la CAM la valeur des r bits de l'adresse fautive. La valeur de ces r bits écrits dans un mot de la CAM représente ainsi un bloc fautif parmi les 2^r blocs fonctionnels. Ainsi, lorsque la mémoire est accédée par une valeur donnée des m bits d'adresse lors des opérations normales du système, les r bits d'adresse parmi les m sont systématiquement comparés à tous les mots de la CAM. Si la comparaison réussie avec un des mots, un signal hit est activé. Ce signal permet de sélectionner un des q blocs redondants et interdit l'accès au bloc régulier (fautif) de la mémoire. Ainsi, l'opération courante de lecture ou d'écriture se fait sur le bloc redondant et plus exactement sur le mot (de ce bloc) sélectionné par la valeur courante des $m-r$ bits d'adresse.

Il faut noter que des fautes différentes ou la même faute dans un bloc régulier peuvent être détectées lors de plusieurs cycles de test. Par conséquent, les mêmes r bits d'adresse correspondant à ce bloc, seront écrits plusieurs fois dans les mots de la CAM. Ainsi plusieurs blocs redondants seront sélectionnés pour réparer un seul bloc régulier fautif. Pour éviter ce phénomène, on compare à chaque détection d'une faute l'adresse courante avec les adresses stockées dans les mots de la CAM. Si la comparaison réussit (signal hit), cela veut dire que l'adresse (r bits) du bloc fautif concerné a déjà été stockée dans la CAM. Dans ce cas, le signal hit est activé pour interdire l'écriture de la valeur des r bits d'adresse courante dans la CAM.

Test et réparation des blocs redondants

Si l'on considère une haute densité de défauts, alors chaque bloc redondant va inclure des fautes. Il est donc nécessaire de réparer les blocs redondants avant de les utiliser pour la réparation des blocs fonctionnels. Il s'agit alors de les réparer au niveau bit de donnée et pour ce faire, chaque bloc redondant doit dispose de k bits redondants comme pour les blocs

fonctionnels[1]. C'est pourquoi une première session de test est réalisée sur les q blocs redondants pour ensuite les réparer, à l'aide d'une reconfiguration statique pour chacun d'entre eux.

Cependant, et similairement au blocs fonctionnels, une fois qu'on a reconfiguré les blocs redondants au niveau bits de donnée, quelque uns d'entre eux demeurent non réparés. Il est alors question d'interdire l'utilisation de tels blocs redondants pour le remplacement des blocs fonctionnels. C'est pourquoi une seconde session de test est réalisée au niveau des blocs redondants pour déterminer ceux qui n'ont pas été réparés précédemment. Pour chaque bloc redondant, nous utilisons un flag (flag1) initialisé à 0. Si une faute est détectée dans un bloc donné, on charge la valeur 1 dans le flag1 correspondant. La valeur 1 de ce flag1 va interdire l'écriture dans le mot CAM correspondant c'est à dire l'utilisation de ce bloc redondant lors du test et la réparation des blocs fonctionnels. On peut éviter la seconde session de test pour les blocs redondants en chargeant directement le signal R_k (voir la section 3.2) de chaque bloc redondant dans le flag1 correspondant. Pour rappel ce signal renseigne, à la fin de la reconfiguration au niveau donnée, sur le résultat de la réparation (la réparation a réussi implique $R_k = 0$ sinon $R_k = 1$).

Test et réparation de la CAM

Encore une fois à cause de la haute densité de défauts, des fautes peuvent être présentes dans la mémoire CAM. Pour les couvrir, on réalise une session de test pour la CAM. Si pendant ce test, une erreur est détectée dans un mot de la CAM, on charge la valeur 1 dans le flag associé à ce mot. Cette valeur 1 va interdire l'utilisation de ce mot de la CAM même si le bloc redondant correspondant va se révéler réparable. Pour éviter une telle situation, nous pouvons utiliser plus d'un mot CAM par bloc redondant. Ainsi, seul le flag d'un des mots CAM d'un bloc redondant est mis à 1, lorsqu'une faute est détectée. De cette façon, un des mots CAM ayant un flag à 0 peut être utilisé pour réparer les blocs fonctionnels. L'utilisation de plusieurs mots CAM par bloc redondant modifie quelque peu le test des blocs redondants. En effet, on doit forcer à 1 les flags des tous les mots CAM de chaque bloc redondant demeurant non réparé après la reconfiguration au niveau bits de donnée.

Le test de la CAM et la prise en compte des fautes pouvant toucher le flag, seront discutés dans la prochaine section qui traitera de la réparation diversifiée mot/ECC. En effet la réparation mot utilise une mémoire CAM non seulement pour stocker des adresses mais aussi

[1] Il est préférable d'utiliser le même nombre k de bits redondants pour les blocs fonctionnels et redondants pour ne pas briser la symétrie dans le layout de la mémoire.

des données, son fonctionnement englobera donc celui de la CAM utilisée dans cette section et par conséquent une analyse plus complète sera présentée.

4.2.1.2 Combinaison de la réparation dynamique au niveau données et la réparation au niveau bloc

La technique de réparation bloc qui a été proposée dans la section précédente, peut aisément être combinée avec la reconfiguration dynamique au niveau bit de donnée, dans le but d'améliorer l'efficacité de la réparation. Pour ce faire, nous sélectionnons les mêmes *r* bit d'adresse, pour effectuer la reconfiguration dynamique au niveau bit de donnée et pour effectuer la réparation au niveau bloc. Nous effectuons également la réparation dynamique au niveau de chaque bloc redondant exactement de la même manière que pour les blocs réguliers de la mémoire. La réparation dynamique, divise la mémoire en 2^r blocs sur lesquels une reconfiguration au niveau bit de donnée est effectuée. Si pour une valeur donnée des *r* bits d'adresse, le blocs mémoire correspondant ne peut être réparé (au niveau bit de donnée), pour les raisons illustrées dans la Figure 4.7, alors la mémorisation de ces *r* bits dans la CAM permettra le remplacement de ce bloc non réparé par un bloc redondant.

4.2.2 Réparation Diversifiée Combinant Code Correcteur d'Erreur (ECC) et la Réparation mot

Les codes correcteurs d'erreur sont utilisés pour corriger les fautes non permanentes telles que les erreurs transitoires induites par l'incidence de particules ionisantes sur la mémoire [BLA83], [CHE84]. De telles erreurs survenant d'une manière aléatoire à la fois dans le temps et dans l'espace ne peuvent être corrigées par les techniques de réparation qui reconfigurent la mémoire suivant un schéma préétabli puisqu'elles surviennent. D'un autre coté, quand la mémoire est affectée par quelques fautes de fabrication, l'utilisation de la réparation implique moins de surcoût en surface que dans le cas des ECC. A titre d'exemple, pour corriger 8 fautes dans une mémoire de 32×512 Kbit, une réparation dynamique au niveau bit de donnée utilisant 2 bits redondants et 3 bits d'adresses ($r = 3$), induit un surcoût en surface de 6,9 % tandis que la technique ECC la moins coûteuse et permettant de corriger un seul bit fautif par mot mémoire (code de Hamming) coûte 18,7 % de surface additionnelle. De plus les pénalités temporelles qu'implique la technique ECC sont significatives. Il est donc préférable d'utiliser les codes ECC pour la correction des fautes transitoires et d'utiliser les techniques de réparation pour éliminer les fautes de fabrication.

Lorsque la densité de défauts augmente, le coût du circuit de réparation augmente aussi et à un certain niveau de densité de défauts, ce coût devient plus élevé que celui d'un ECC. Il est donc intéressant de rechercher un champ d'utilisation d'une telle technique dans le cas des hautes densités de défauts. Il faut néanmoins rester prudent quant à l'évolution de la complexité et donc du coût additionnel des circuits ECC. En effet, ces codes deviennent rapidement inutilisables en termes de surcoût en surface et de pénalité sur la vitesse de fonctionnement de la mémoire, au fur et à mesure que le nombre d'erreurs corrigibles augmente. A cause de ces limitations et à cause de la faible probabilité que les fautes transitoires donnent lieu à de multiples bits erronés, les applications actuelles sont protégées par un code correcteur d'un seul bit erroné (par mot mémoire) comme le code de Hamming.
Dans le contexte particulier des mémoires affectées par de hautes densités de défauts, effectuer la réparation en utilisant des codes correcteurs d'erreurs devient inefficace à cause du nombre limité de fautes pouvant être réparées. En effet, dans ce cas de hautes densités de défauts, plusieurs mots mémoire peuvent inclure plusieurs erreurs. Et utiliser un ECC, peut laisser sans réparation ces mots et la mémoire sera rejetée. Cependant, l'utilisation d'une technique ECC, peut être intéressante si elle est combinée avec une seconde technique de réparation pour couvrir l'ensemble des fautes de la Figure 4.7. En effet, on pourrait utiliser une technique ECC qui permette de corriger la plupart des mots fautifs affectés par un nombre précis de fautes, puis utiliser une technique de réparation pouvant réparer des mots mémoire affectés par plusieurs bits fautifs. Il est clair que la meilleure technique pour réparer des mots contenant de multiples bits fautifs, est la réparation mot (ou ligne).
Dans ce qui suit nous n'allons pas nous attarder sur les techniques ECC étant donné qu'elles sont aujourd'hui bien connues du monde du test et de la fiabilité des mémoires [BLA83], [CHE84]. Nous allons par contre présenter une solution de réparation mot dans le contexte des hautes densités de défauts, dans lequel les mots redondants ainsi que le circuit de réparation peuvent être fautifs. Le circuit de réparation mot doit alors considérer toutes ces fautes si l'on veut éviter un échec de la réparation.

4.2.2.1 La réparation mot considérant les fautes dans la redondance

La réparation au niveau mot mémoire est une technique bien connue qui fut introduite dans [SAW89]. Elle utilise une mémoire associative (ou mémoire CAM pour « Content Addressable Memory ») de *Nmc* mots. Chaque mot est composé d'un champ d'adresse (descripteur ou clef) et d'un champ de donnée, comme l'illustre la Figure 4.9. Le champ

d'adresse de chaque mot dispose d'un comparateur. Cela permet la comparaison parallèle de l'adresse courante avec les champs d'adresse de chaque mot de la CAM. Durant les modes de test et de réparation, un compteur est utilisé pour sélectionner un mot de la CAM. Quand une faute est détectée, l'adresse mémoire courante est stockée dans le mot sélectionné de la CAM et le compteur est incrémenté. Durant une opération de lecture ou d'écriture dans le mode de fonctionnement normal de la mémoire, l'adresse mémoire courante est comparée en parallèle avec l'ensemble des champs d'adresses. Une réussite de la comparaison avec un champ d'adresse donné active un signal hit qui va permettre la lecture ou l'écriture dans le champ de donnée correspondant. En même temps, le signal hit contrôle un multiplexeur qui connecte sur le bus de données en sortie, la donnée venant de la CAM comme le montre la Figure 4.10. Sinon et par défaut, le multiplexeur connecte sur la donnée en sortie, la donnée provenant de la mémoire. A cause du fait qu'un algorithme de test puisse sensibiliser par l'intermédiaire de l'adresse mémoire plusieurs fois le même mot, ce dispositif de réparation peut stocker dans plusieurs champs d'adresses, la même adresse du mot mémoire fautif. Cette situation doit être évitée car elle utilise les ressources redondantes très inefficacement. Pour ce faire, on utilise la comparaison de l'adresse également dans la phase de test et de réparation. Ainsi, si une faute est détectée et que le signal hit s'active, on interdit le stockage de l'adresse courante dans le mot sélectionné de la CAM et on interdit l'incrémentation du compteur.

Dans les solutions de réparation mot existantes, les fautes dans la CAM ne sont pas prises en compte et donc ne sont pas réparées. Dans ce cas, le seul moyen possible est de tester la CAM et la rejeter dès qu'une faute est détectée. Cependant, dans le contexte de la présente

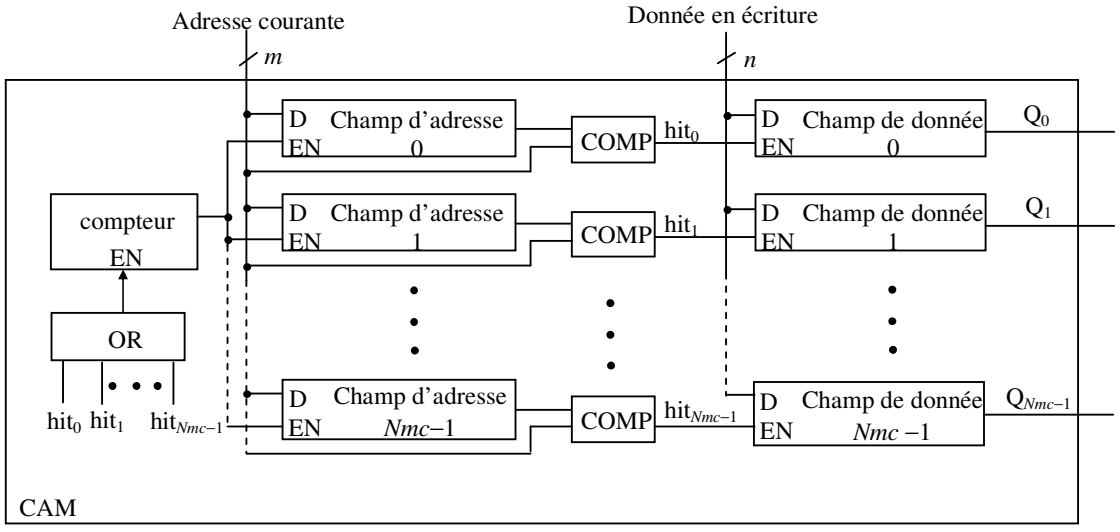


Figure 4.9. Organisation de la CAM.

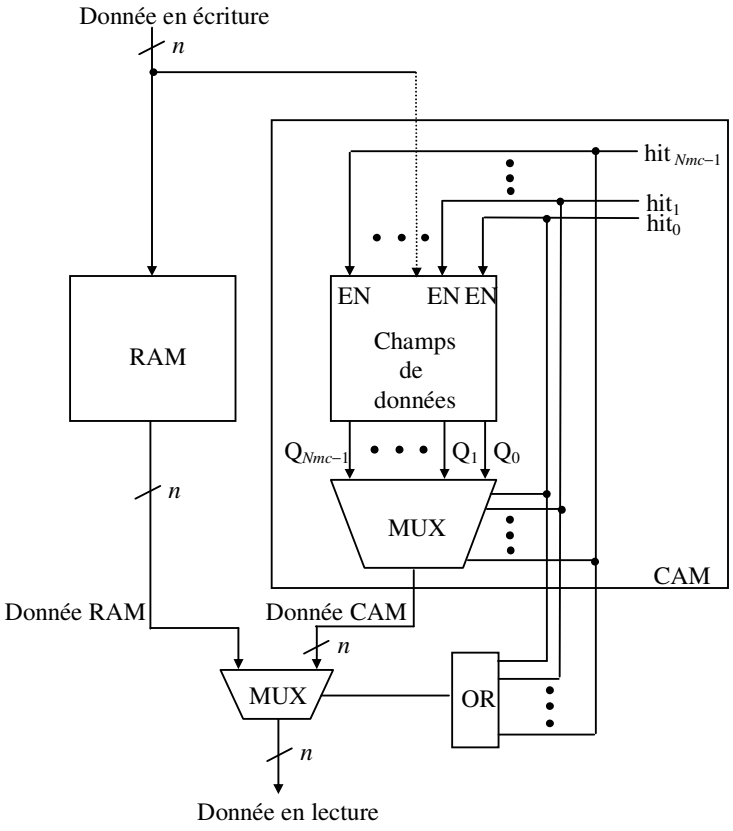


Figure 4.10. Interface RAM/CAM.

application, la mémoire peut inclure des milliers de mots mémoires fautifs non réparés par la technique ECC. Donc, la taille de la mémoire CAM sera importante, et la probabilité qu'elle soit non fautive est négligeable, particulièrement dans les technologies avec des hautes densités de défauts. Par conséquent, rejeter la mémoire à chaque fois qu'une faute dans la CAM est détectée, donne lieu à un rendement nul.

Une solution possible pour masquer les fautes de la CAM consiste à réparer les mots de la CAM fautifs. Cependant, ceci peut entraîner une augmentation significative du surcoût en surface si l'on ajoute de la redondance mais aussi les circuits de reconfiguration. Une solution plus économique peut être élaborée si l'on exploite les spécificités opérationnelles d'une CAM. En effet, étant donné que tout mot de la CAM peut être utilisé pour remplacer un mot mémoire fautif, nous pouvons juste interdire l'utilisation des mots CAM fautifs et n'utiliser que ceux qui sont non fautifs pour réparer la mémoire. En se basant sur ce concept, nous avons développé deux solutions permettant la réparation de la CAM. La première utilise pour cela un BIST spécifique pour la CAM et la seconde permet la réparation de la CAM sans utiliser aucun circuit BIST pour CAM. Nous allons dans ce qui suit exposer ces deux solutions.

4.2.2.2 Réparation de la CAM en utilisant un BIST spécifique au test des CAM

Dans cette section nous allons décrire le fonctionnement du circuit de réparation mot utilisé pour l'approche de réparation diversifiée ECC/réparation mot mémoire (CAM). La réparation mot dont il est question ici prend en compte les fautes dans la CAM et utilise pour cela un BIST spécifique pour les CAM. Un BIST pour CAM doit être capable de détecter toutes les fautes se trouvant dans les mots CAM que se soit dans le champ d'adresse, le champ de donnée ou dans le comparateur (Figure 4.9). Plusieurs architectures de BIST pour CAM ont été développées [ALA94], [LIN98], [SID99] et il n'est donc pas question ici de discuter de cet aspect mais seulement de l'utilisation des résultats de ce test pour la réparation de la CAM qui sera alors utilisée pour la réparation de la RAM.

Dans ce circuit, nous utilisons un flag (flag1) pour chaque mot de la CAM. Durant la phase de test spécifique à la CAM, nous écrivons la valeur 0 dans le flag1 si le mot CAM correspondant est non fautif et la valeur 1 s'il est fautif. La valeur 1 du flag1 sera donc utilisée pour interdire l'utilisation du mot CAM fautif. Cela peut se faire de la manière suivante. Lorsqu'une faute est détectée durant le mode de test et de réparation de la mémoire et que l'adresse courante n'est pas déjà mémorisée dans la CAM, alors le BIST est immédiatement arrêté. Cet arrêt du BIST permet la prise en compte uniquement de la faute en cours. En effet, pour cette faute, on va commencer un cycle de recherche du premier mot CAM non fautif afin de stocker dans le champ d'adresse de celui ci, l'adresse courante (liée à la faute en cours). Pour illustrer l'intérêt de cette action, prenons comme exemple une succession de deux fautes détectées dans la RAM. Dans le cas ou les mots de la CAM sont non fautifs, il n'y a aucun problème à stocker les deux adresses courantes successives dans deux mots CAM successifs. Mais lorsque les mots CAM sont fautifs, on ne peut stocker l'adresse de la seconde faute tant que celle de la première ne l'a été préalablement. Etant donné qu'on ne connaît pas à l'avance le nombre de mots CAM fautifs à parcourir pour stocker la première faute, on se trouve dans l'impossibilité de garder la seconde faute et de la stocker par la suite. L'arrêt du BIST est donc suivi de la recherche du premier mot CAM non fautif. Cela peut se faire en incrémentant de 1 le compteur tant que le compteur sélectionne un mot CAM dont le flag1 est à 1 (mot fautif de la CAM). Une fois un mot CAM avec flag1 = 0 est sélectionné, on l'utilise pour stocker l'adresse courante. Le BIST est aussitôt redémarré. Deux modes de redémarrage sont possibles. On peut redémarrer le BIST à partir du dernier état dans lequel il a été laissé ou bien le redémarrer depuis le début. Le premier mode a l'avantage de minimiser le temps de test mais implique un certain effort de conception pour le contrôle du BIST. Le deuxième

mode est plus facile à contrôler ; de plus il permet le test de la RAM à la vitesse nominale[2] en évitant les arrêts et les redémarrages, mais d'un autre côté, il augmente considérablement le temps de test dans le cas des fautes multiples. Les fautes de la RAM déjà prises en compte sont donc à chaque session de test re-détectées mais cela n'est pas gênant puisqu'un test sur l'existence préalable d'une adresse fautive dans la CAM est disponible dans tous les cas de figure.

Durant le fonctionnement normal de la mémoire, la valeur 1 du flag1, force également à 0 la sortie du comparateur du champ d'adresse. Cette action assure, en présence d'une faute dans un mot de la CAM, que le champ de donnée correspondant ne sera jamais utilisé pour une opération d'écriture ou de lecture et assure aussi que le signal hit ne déconnecte pas la mémoire au bus de données.

4.2.2.3 Réparation de la CAM sans utilisation d'un BIST spécifique à la CAM

Une seconde solution serait de ne pas utiliser une phase de test spécifique pour la CAM. L'idée est de commencer par effectuer un test de la mémoire. Dès qu'une erreur est détectée, on mémorise l'adresse courante dans le champ d'adresse courant sans se soucier de l'état fautif ou non fautif du mot CAM. Pendant cette phase de test tous les flags1 sont initialisés à 0. Ensuite, on re-teste la mémoire mais cette fois-ci en utilisant les adresses des mots fautifs stockées dans les mots CAM. Deux situations peuvent alors se présenter :

- Si une erreur est détectée, et qu'un signal hit est activé alors on écrit la valeur 1 dans le flag1 du mot CAM dont le hit s'est activé. De plus on stocke l'adresse courante dans un nouvel emplacement de la CAM sans se soucier de son état. Pour ce faire, on commence cette session de test en utilisant le compteur d'adresse avec comme état initial la dernière valeur à laquelle il a été incrémenté lors de la session de test précédente. Le flag1 du nouvel emplacement CAM reste quand à lui à 0.
- Si une erreur est détectée sans que le signal hit ne s'active, on stocke l'adresse courante dans un nouvel emplacement de la CAM sans se soucier de son état.

De cette façon, touts les mots CAM fautifs sélectionnés lors de la première phase de test sont déclarés à la fin de la seconde phase de test comme inutilisables pour la réparation et sont repérés par le flag1 à 1. Cependant, il se peut que certains nouveaux emplacements de la CAM sélectionnés lors de la seconde phase de test soient à leur tour fautifs. On doit alors

[2] Le test de la RAM à la vitesse nominale permet de détecter une certaine classe de fautes (fautes de délais).

activer un troisième test similaire au second. Nous devons répéter ce test jusqu'à ce qu'aucune faute ne soit détectée parmi les mots CAM sélectionnés (succès de la réparation) ou bien jusqu'à épuisement des mots CAM sans pouvoir réparer toutes les fautes (échec de la réparation).

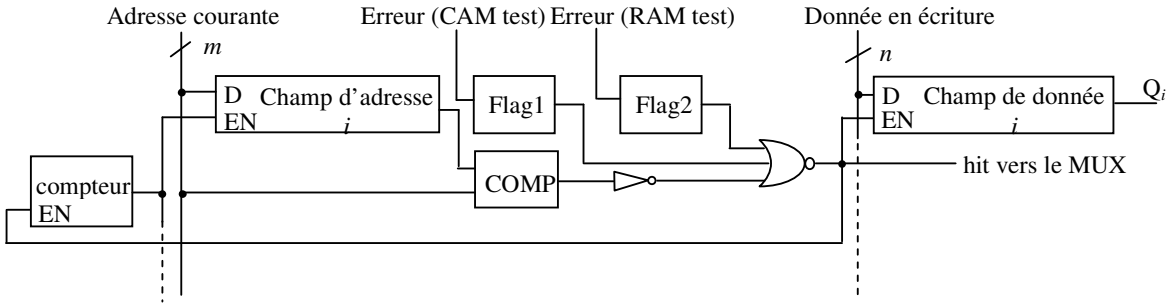


Figure 4.11. Exemple d'implémentation d'une CAM auto-réparable.

Cette solution requiert aussi une simple machine d'états finis qui contrôle les différentes sessions de test et de réparation. Notons aussi que dans les deux solutions proposées, on doit utiliser un second flag (flag2) à coté de chaque mot de la CAM. Ce flag se met automatiquement à 1 des qu'une faute est détectée pendant le test de la mémoire. Il sera pris en compte à la fois avec le signal hit et la valeur du flag1 pour sélectionner la CAM pendant le fonctionnement normal (Figure 4.11). En effet, si les champs d'adresses sont initialisés à une valeur INIT alors pendant le fonctionnement normal, la valeur INIT de l'adresse courante peut activer plusieurs signaux hit correspondant à des mots CAM non sélectionnés pour la réparation donnant lieu à un disfonctionnement du circuit BISR. L'ajout d'un bit (flag2) permet alors de différencier entre une valeur INIT fautive et non fautive de l'adresse courante.

4.2.2.4 Discussion et comparaison des deux solutions relatives à la réparation de la CAM

Les deux solutions développées plus haut sont très différentes au niveau conceptuel et cela se répercute au niveau de leur implémentation. En effet, la solution la plus naturelle et qui utilise un BIST spécifique au test des CAM, implique un effort de conception surtout relatif au BIST implémentant des algorithmes de test pour CAM. Cela implique bien entendu un coût additionnel en termes de silicium. Dans une des approches possibles, toujours dans la même solution, le temps total de test/réparation $T1$ est la somme du temps de test de la CAM et le temps de test de la RAM, nous avons alors : $T1 = t_{CAM} + t_{RAM}$.

La seconde solution qui utilise le contenu des mots de la CAM (plus précisément les champs d'adresse) pour le test de la CAM, n'utilise pas de BIST pour CAMs et nous épargne alors tout un effort de conception. Par contre elle implique un temps de test $T2$ donné par :

$T2 = 2.t_{RAM\ test} + F$(nombre de mots CAM fautifs) . $t_{RAM\ test}$

Le premier terme désigne le temps de test de la RAM et de la CAM. Le second terme concerne la répétition des tests de la CAM selon la disposition et le nombre des mots CAM fautifs. Notons que $\min(T2) = 2.t_{RAM\ test}$ et obtenu si aucun mot CAM n'est fautif ($F = 0$). Si on compare $T1$ et $T2$, on a dans le meilleur des cas ($F = 0$) : $T2 = 2.T1$ avec $t_{CAM} << t_{RAM}$. Il est donc question de choisir entre une solution qui nécessite l'implémentation d'un BIST spécifique aux CAM mais qui effectue les opérations de test/réparation en un temps raisonnable pratiquement équivalent au temps de test de la mémoire seule, ou bien une solution ne nécessitant pas de BIST spécifique aux CAM mais effectuant une session de test/réparation au moins deux fois plus longue que la première.

Conclusion

Le travail présenté dans ce chapitre concerne le développement de diverses approches Built-In Self-Repair (BISR) capables de réparer des mémoires affectées par de très hautes densités de défauts (plus de quelques pour cent par cellule mémoire). Pour atteindre de telles efficacités de réparation, nous avons adopté deux approches.

La première approche combine entre elles les unités fautives pour générer des unités réparées, au lieu de remplacer les unités défaillantes par des unités correctes comme il a été question avec les approches présentées dans le chapitre précédent. Cette approche devrait mieux fonctionner dans le cas de très hautes densités de défauts dans lequel il est difficile de disposer d'unités non fautives. La combinaison est basée sur le fait que dans la majorité des situations, les cellules fautives ne vont pas occuper les mêmes positions dans deux unités différentes. La difficulté d'appliquer un tel schéma de réparation réside dans la sélection des cellules correctes dans chaque unité. C'est donc un schéma de réparation agissant au niveau de la cellule mémoire et peut donc requérir un circuit de reconfiguration extrêmement complexe. Pour simplifier ce schéma, nous avons utilisé une approche basée sur les polarités d'erreurs produites à la sortie de chaque unité mémoire. Nous avons alors combiné les unités fautives en utilisant des fonctions de reconfiguration qui masquent une polarité d'erreur spécifique aux deux unités combinées. Trois schémas basés sur le concept de combinaison des unités, ont été présentés.

Un premier qui combine les unités fautives en utilisant des fonctions de reconfigurations fixes mais requiert l'utilisation d'un nombre important d'unités redondantes. L'autre schéma utilise des fonctions de reconfiguration qui effectuent les combinaisons suivant les polarités d'erreurs de chaque unité. Ce dernier schéma permet de réduire le nombre d'unités redondantes nécessaires à la réparation au détriment d'une logique de reconfiguration plus complexe. Un dernier schéma utilisant des fonctions de reconfiguration fixes a été proposé pour le cas des mémoires avec une distribution déséquilibrée de défauts. Il considère la situation où une polarité d'erreur particulière et nettement plus probable que toutes les autres. Pour de pareilles situations, on peut utiliser des fonctions de reconfiguration fixes utilisant un nombre réduit d'unités redondantes.

La seconde approche est basée sur la distribution des défauts dans les différentes unités réparables de la mémoire. Pour la majorité de ces unités, la distribution des défauts résulte en un nombre de fautes oscillant autour du nombre moyen de fautes par unité réparable. Ce dernier étant engendré par la densité de défauts considérée. D'un autre cote, quelques unités réparables, peuvent contenir un grand nombre de fautes supérieur au nombre moyen de fautes par unité réparable. Du fait qu'on ne connaît pas à l'avance quelles unités qui concentreront le plus grand nombre de fautes, nous devons disposer d'un nombre suffisant d'unités redondantes pour chaque unité fautive afin de garantir leur réparation. Une telle disposition des ressources redondantes donnerait rapidement lieu à un coût de réparation très important. En tenant compte de cette remarque, nous améliorons l'efficacité de la réparation en proposant une approche de réparation diversifiée (ou mixte). Celle-ci utilise un premier schéma de réparation pour réparer la majorité des unités fautives (incluant un nombre de faute qui oscille autour du nombre moyen), et utilise aussi un second schéma de réparation qui remplace les unités non réparées par des unités redondantes correctes ou bien réparées. Afin d'illustrer cette approche nous avons développé une technique qui utilise la réparation au niveau bit de donnée combinée à la réparation au niveau bloc. La réparation au niveau bit de donnée a été présentée dans le chapitre précédent, tandis que la réparation bloc a été introduite ici en se basant sur certaines propriétés des mémoires associatives. Une seconde technique de réparation mixte utilise les codes ECC pour corriger la majorité des mots mémoire fautifs, et une réparation mot pour réparer les mots mémoire non corrigés parce qu'ils contiennent un grand nombre de fautes. Nous nous sommes concentrés dans ce dernier volet sur le développement d'une nouvelle technique de réparation mot, basée sur le fonctionnement des CAM (Content Adressable Memories), permettant de masquer les erreurs se trouvant dans la CAM, condition nécessaire pour le succès de l'ensemble de la technique. Nous avons aussi

montré qu'une telle technique est efficace pour mémoires atteintes de très hautes densités de défauts, tandis que la seule utilisation d'un code ECC résulte en un rendement nul même avec une redondance plus élevée que celle utilisée dans l'approche mixte. Bien sûr d'autres combinaisons de deux techniques de réparation, ou plus, peuvent être utilisées pour l'approche mixte. La condition importante pour rendre l'approche efficace, est de sélectionner un premier schéma qui répare efficacement la majorité des unités fautives, et un second schéma qui répare toutes les unités fautives demeurant non réparées par le premier schéma. Cela peut être répété plusieurs fois suivant le degré de mixture de la technique en tenant compte du surcoût en surface que peut engendrer de telles additions.

Les techniques de tolérance aux fautes développées dans ce chapitre concernent des mémoires affectées par des densités de défauts supérieures jusqu'à deux ordres de grandeur aux densités discutées dans le chapitre précédent. Ces techniques peuvent très bien remplacer les techniques de réparation développées dans le chapitre précédent avec un moindre surcoût en surface. Les différentes évaluations expérimentales présentées dans le chapitre suivant nous confirment ces capacités.

CHAPITRE V. Outil d'Injection de Fautes pour l'Evaluation de l'Efficacité de Réparation des Architectures BISR

Introduction

Dans le contexte du test et de la réparation intégrée, il est important d'évaluer l'efficacité des différentes architectures BISR proposées. Cette évaluation consiste à déterminer pour diverses densités de défauts, le rendement induit par l'utilisation d'un schéma de réparation et le surcoût matériel de la circuiterie correspondante. Souvent, une architecture BISR met en jeu un certain nombre de paramètres qu'on appelle paramètres de réparation. Il s'agit alors de déterminer à l'aide d'un grand nombre de simulations, le meilleur compromis entre l'efficacité de réparation et le surcoût matériel pour chaque densité de défauts. Nous appelons efficacité de réparation, l'amélioration du rendement de fabrication obtenu en utilisant un schéma de réparation donné.

L'objectif de ce chapitre est double.

Dans un premier temps, il est question d'évaluer l'efficacité de l'ensemble des architectures BISR développées dans les deux chapitres précédents et démontrer leur réelle qualité de techniques de tolérance aux fautes pour les mémoires à très hautes densités de défauts. Pour évaluer le coût matériel des différentes implémentations nous avons implémenté chacune d'entre elles (C++, [BOU02b]) en vue d'une génération automatique hautement paramétrable. L'évaluation de l'efficacité de chaque implémentation en présence d'une densité de défauts particulière peut quant à elle être réalisée à l'aide de deux approches. Une approche théorique traditionnellement utilisée, et qui passe par l'établissement de formules probabilistes d'estimation du rendement de fabrication. Une approche expérimentale pour laquelle on a développé un outil d'injection statistique des fautes. Le désavantage de la première approche est qu'elle requiert le développement des formules probabilistes pour chaque nouveau schéma de réparation. Une telle opération n'est pas triviale et dépasse souvent le cadre de compétences d'un designer software ou hardware. La deuxième approche est plus flexible, car à l'aide d'une spécification précise du comportement du BISR concerné, on peut effectuer un grand nombre de simulations permettant d'estimer son efficacité. Le point faible de cette approche est son temps de simulation non négligeable.

Le second objectif de ce chapitre est de concevoir cet outil de simulation et d'injection de fautes de manière à lui procurer une flexibilité suffisante pour qu'il soit non seulement

capable d'évaluer l'efficacité de nos propres architectures BISR, mais aussi d'évaluer n'importe quel autre schéma de réparation. Un modèle précis de distribution des défauts est également implémenté dans l'outil. Il s'agit du modèle de distribution de défauts basé sur les clusters (amas de fautes). Ce dernier est nécessaire pour une bonne prédiction du rendement de fabrication au niveau wafer.

5.1 Travaux Antérieurs

L'intérêt porté à l'évaluation du rendement de fabrication par la simulation est très récent [HUA02], [SEH03] . En effet, dans le contexte technologique actuel, la fabrication des mémoires subit des chutes de plus en plus importantes au niveau du rendement. Les fabricants s'accordent à dire que la réparation par la voie de la redondance matérielle est une solution incontournable pour rehausser le rendement de fabrication des mémoires embarquées. Reste à estimer l'efficacité des techniques de réparation pour faire le choix final.

Dans l'article [SEH03], un outil d'analyse du rendement de fabrication des mémoires a été présenté. Cet outil ne peut pas vraiment s'inscrire dans le contexte des mémoires sous-microniques et manométriques utilisant l'auto réparation intégrée pour plusieurs raisons.

1) La méthodologie d'analyse, qui consiste uniquement à estimer l'efficacité d'un mécanisme de réparation sans se soucier du coût en surface, est une évaluation acceptable mais seulement pour les technologies actuelles.
2) La densité de défaut considérée dans [SEH03] est de l'ordre d'une dizaine de défauts/cm^2 donnant lieu à une probabilité globale d'occurrence d'une faute dans une cellule de l'ordre de 5.10^{-7}. Avec de telles probabilités, les techniques de réparation basées sur les algorithmes de réparation traditionnels (algorithme de Day [DAY85], et "*broadside approach*" de Tarr [TAR84], voir aussi la section 2.1), sont suffisantes pour corriger les fautes. Aussi, une estimation du coût en surface induit par le circuit de réparation, n'est pas importante surtout lorsqu'on n'est pas dans le contexte d'une auto réparation intégrée.

Nous voudrions pour notre part, analyser le rendement de fabrication pour des mémoires utilisant des techniques de tolérance aux fautes intégrées [NIC01], [ACH01], [NIC03(a)], [NIC03(b)], [ACH03], [NIC03(c)], [NIC03(d)], [NIC04] dédiées au cas des très hautes densités de défauts (la probabilité globale pour une occurrence d'une faute dans une cellule allant de 10^{-6} à 10^{-2}). Deux principales exigences découlent de cet objectif. Premièrement et

comme nous l'avons vu précédemment, ces approches utilisent un certain nombre de paramètres (paramètres de réparation) sur lesquels on peut agir pour augmenter l'efficacité de la réparation mais au détriment d'une certaine augmentation de la surface. Il est par conséquent nécessaire ici d'estimer conjointement coût en surface et efficacité pour une densité de défauts donnée. Deuxièmement, étant donné que le nombre de mémoires par puce ne cesse d'augmenter d'un processus technologique à un autre, il est important d'utiliser un modèle de distribution de fautes assez précis. Cette dernière exigence est importante si nous voulons correctement évaluer le rendement de fabrication par wafer. C'est pour cette raison qu'on a adopté et intégré dans l'outil, un modèle basé sur les densités de défauts "clusterisés".

5.2 Génération Automatique de structures BIST/BISR Synthétisables

Dans l'objectif d'exploiter les schémas Built-In Self-Repair décrits dans les deux chapitres précédents, nous avons besoin d'un outil de génération automatique de structures matérielles BISR. La génération automatique est nécessaire pour deux raisons.

Premièrement, pour les besoins de notre étude et pour évaluer les mérites de nos schémas de réparation, il nous faut implémenter un grand nombre de cas BISR en agissant sur les paramètres de réparation, afin de déterminer l'efficacité de la réparation et le coût matériel pour des densités de défauts variées. Implémenter ces configurations manuellement requiert un temps excessif limitant alors le nombre d'expérimentations et la qualité de l'évaluation.

Deuxièmement, pour des raisons de gain en temps de conception, l'automatisation de la génération de solutions BIST et BISR, est une condition nécessaire pour percer le marché dans le domaine de la tolérance aux fautes.

Le générateur de BIST/BISR appelé M-BISTeR a été développé à l'aide du langage CHDL (C++ based Hardware Description Language) [BOU02(b)]. C'est un langage de description matérielle qui utilise le langage logiciel C++ pour implémenter les différentes structures. Une des principales propriétés du langage CHDL est sa grande capacité de modélisation. Cette modélisation repose sur le paramétrage de l'architecture du circuit, mais aussi sur celle de son interface (entité). Il permet aussi une manipulation facile des blocs matériels par les parties logicielles de M-BISTeR. D'un autre coté, le langage CHDL fournit un vérificateur sémantique qui permet de détecter des constructions syntaxiques pouvant être fausses une fois translatées dans le langage matériel cible. Ce vérificateur opère dans la phase de translation HDL, c'est-à-dire bien avant la compilation des fichiers HDL générés. Cette caractéristique

réduit considérablement le temps global de développement et contribue ainsi à améliorer le facteur *time-to-market* des nouvelles solutions à intégrer.
En résumé, les principales caractéristiques du langage CHDL sont :

- Une hiérarchie de classes dynamiques et flexible (~ 60 classes) qui décrit les circuits matériels et supporte jusqu'à 70 % des constructions sémantiques des langages HDL (i.e. VHDL/VERILOG).
- Une grande capacité de modélisation (e.g. paramétrage, gestion facile des variables entrées/sorties et manipulation de fichiers …) qui permet une réduction importante de la capacité mémoire nécessaire au stockage des librairies locales.
- La syntaxe CHDL est inspirée de celle des langages de description du matériel afin de faciliter l'apprentissage des concepteurs ayant pour charge la maintenance et la mise à jour des librairies des circuits.
- Offre une vérification de syntaxe orientée matériel.
- Génère des descriptions VHDL et/ou VERILOG synthétisables.
- Peut être réutilisé avec n'importe quel outil de CAO manipulant ou générant des descriptions matérielles.

Pour générer un modèle BIST/BISR à synthétiser, l'utilisateur commence par spécifier la mémoire, l'algorithme de test ainsi que les caractéristiques de l'architecture de test/réparation désirée. A partir de ces spécifications (sous forme de fichiers), un bloc fonctionnel spécifique constitué essentiellement de parseurs, extrait les différentes caractéristiques relatives à l'architecture test/réparation cible et les stocke dans des structures de données appropriées (classes), en utilisant les utilitaires FLEX/BISON [LEV92]. Ces classes internes (copies d'exécution) sont nécessaires pour accélérer l'exécution du flot et faciliter la gestion des données par les différents algorithmes software implémentés par l'outil. Ces classes sont ensuite traitées par une cascade d'outils internes, qui constituent le noyau de M-BISTeR : le module de génération de BIST, le module génération de BISR et le module d'interconnections. En utilisant des conceptions RTL (mais aussi structurelles et comportementales pour d'autres) fournis par la librairie des cellules et macro cellules, le module de génération du BISR génère l'architecture BISR cible. Ainsi, on peut avoir un code RTL synthétisable décrit en langage VHDL et/ou VERILOG.

Nous n'allons pas nous étendre davantage sur le sujet de l'automatisation des architectures BIST, BISD ou BISR, car l'étude a déjà fait l'objet de plusieurs thèses [KEB92],[BOU02a]. La suite de ce chapitre sera consacrée à l'évaluation des solutions d'auto réparation que nous avons développé et présenté dans les chapitres III et IV.

5.3 Choix des Solutions BISR pour la Génération Automatique et Estimation du Surcoût en Surface

Une riche collection de solutions BISR a été présentée dans les deux chapitres III et IV. Dans le chapitre III, on a commencé par présenter des techniques de réparation au niveau bit de donnée. Nous pouvons classer ces dernières, selon le type de la redondance, en deux grandes catégories. Une catégorie qui utilise des unités redondantes entières (similaires aux unités fonctionnelles), et une catégorie qui utilise des colonnes singulières comme éléments de redondance. Toutes les implémentations de la première catégorie qu'elles soient statiques/dynamiques, séquentielle/combinatoire ou locale/distante ont été intégrées dans le générateur BISR pour offrir une bonne flexibilité de l'outil M-BISTeR. L'intégration de ces techniques efficaces pour quelques futurs processus technologiques à venir, a été nécessaire pour assurer une place confortable de l'outil dans le marché des techniques de tolérance aux fautes pour les mémoires. L'outil M-BISTeR s'intègre aisément dans un flux de conception [BOU02(b)], en faisant notamment des appels directs aux simulateurs et/ou aux synthétiseurs logiques. Ajouté à cela la génération automatique, nous pouvons alors estimer avec précision le surcoût en surface de toutes ces architectures BISR en fonction des paramètres de réparation. Pour ce qui est des techniques de réparation utilisant des colonnes singulières, aucune d'elle n'a encore été intégrée dans l'outil pour une génération automatique. La raison tient au fait que ces schémas de réparation requièrent pour être valides des compilateurs de mémoires spéciaux ayant la capacité de générer un jeu de colonnes singulières autour de la mémoire régulière. Souvent, c'est une contrainte qui reste posée pour la plupart des générateurs de mémoires. Cependant, nous pouvons sans peine estimer le coût en surface de ces schémas BISR (utilisant des colonnes redondantes singulières), en prenant comme référence les meilleurs résultats issus de la synthèse logique de la première catégorie de schémas BISR (utilisant des unités redondantes entières). En effet, cela est tout à fait possible car on retrouve les fonctions de reconfiguration de base, comme étant un sous ensemble non seulement pour les BISR de la seconde catégorie mais même pour la réparation basée sur les polarités d'erreurs.

Un second type d'architecture de réparation a été intégré dans le générateur BISR, c'est le cas de la réparation du mot de mémoire. L'intégration de ce type de réparation est très importante car elle ajoute une flexibilité supplémentaire de l'outil par rapport à la distribution des fautes (dans le prolongement des lignes mémoire dans ce cas) mais aussi vis-à-vis des générateurs de mémoires qui ne produisent aucune redondance autour de la mémoire fonctionnelle. En effet, la redondance est dans ce cas entièrement incluse dans le circuit de réparation (champ de données dans les mots CAM). Une mesure précise du coût en surface, via un synthétiseur logique, est alors effectuée pour le circuit de réparation mot (généré en fonction des paramètres de réparation). Ces résultats seront à leur tour exploités pour estimer les deux approches de réparation diversifiée : la réparation bit de donnée/bloc ainsi que la réparation ECC/mot mémoire. Pour la seconde technique, une bonne approximation serait d'additionner le coût dû à la technique ECC (généré par l'outil M-ROKIT® de la société iRoC Technologies [iRoURL]) au coût dû à la réparation mot. En ce qui concerne la première technique de réparation diversifiée, on remarquera que le circuit de réparation bloc composé d'une CAM contenant uniquement des champs d'adresses, est un sous ensemble du circuit de réparation mot. Il suffira alors d'additionner le coût correspondant à un certain pourcentage du circuit de réparation mot au coût de la réparation au niveau bits de données.

5.4 Description Générale de l'Outil d'Injection de Fautes

L'outil a été principalement développé à l'aide du langage C++. Ce choix a été fait pour deux raisons. La première c'est de rester pour des raisons pratiques conforme avec le générateur d'architectures matérielles BIST/BISR et qui utilise le même langage. La seconde tient au fait de l'organisation même de l'outil qu'on veut modulaire. Deux modes de simulation sont possibles : une simulation pour déterminer l'efficacité d'une seule instance mémoire et une seconde qui concerne le calcul de l'efficacité de réparation au niveau *wafer* (contenant *M* mémoires). Pour simuler une seule instance mémoire il nous faut passer par les étapes suivantes (Figure 5.1(a)) :

- La génération d'un modèle de mémoire régulière et redondante.
- Injection des fautes dans la mémoire.
- Déroulement d'un algorithme de test et de réparation.
- Calcul de l'efficacité de réparation après avoir répété un certain nombre de fois les trois étapes précédentes.

Si on passe au niveau wafer, il nous faut rajouter les étapes ci-dessous (Figure 5.1(b)):

- Génération d'un modèle de wafer avec une distribution de densités de défaut de type clusters.
- Calculer l'efficacité de chaque mémoire du wafer.
- Calcul de l'efficacité au niveau wafer après avoir répété un certain nombre de fois les deux étapes précédentes.

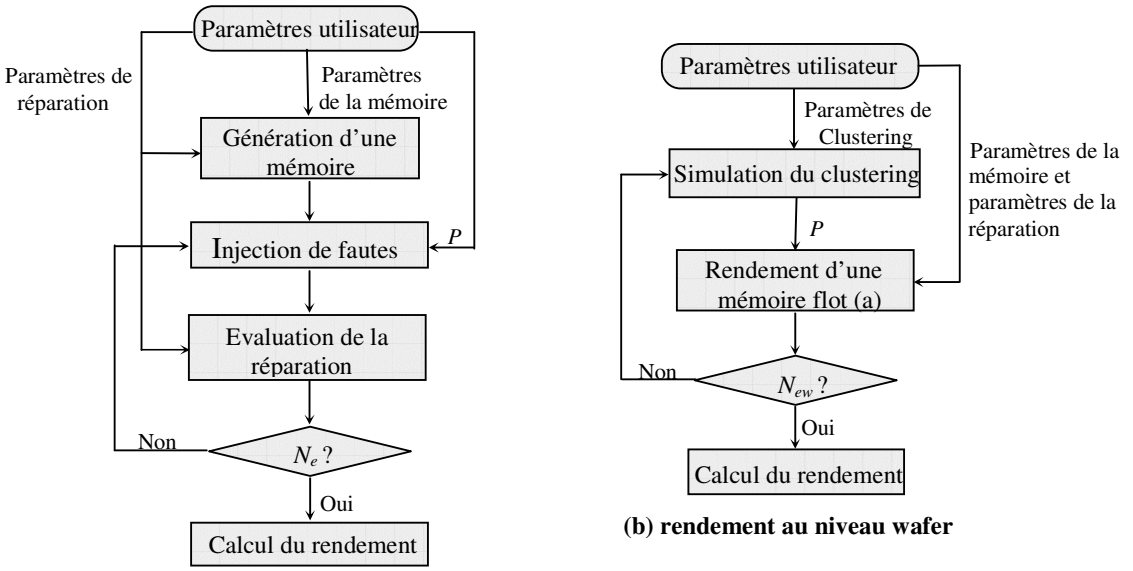


Figure 5.1. Flot de simulation et organisation de l'outil.

Le but d'une organisation modulaire basée sur l'utilisation de classes, est d'intégrer par l'intermédiaire de modules indépendants, de nouvelles alternatives à chaque étape du flot de simulation. La dépendance entre les modules appartenant à des étapes de simulation différentes est décrite par un programme principal (« main »). Ce dernier propose à l'utilisateur de choisir un des divers flots possibles avant de commencer les simulations. L'utilisateur choisit ensuite le(s) flot(s) de simulation qu'il désire.

Nous avons jusqu'à présent implémenté un certain nombre de modules pour les étapes d'injection de fautes qui sont utilisables par tous les flots de simulation. Il s'agit de la génération d'une probabilité d'occurrence d'un défaut. Celle-ci est un facteur déterminant pour la qualité des simulations. Deux méthodes pour la génération d'une probabilité sont utilisables au choix à chaque fois qu'une simulation est invoquée :

- Génération d'une probabilité à l'aide d'un LFSR software,
- Génération d'une probabilité à l'aide de la fonction rand() du langage software C.

En ce qui concerne l'étape de Test/Réparation, celle-ci supporte toutes les approches de réparations développées dans cette thèse, soient les sept schémas suivants :

- Réparation statique au niveau bit de donnée,
- Réparation dynamique au niveau bit de donnée,
- Réparation au niveau bit de donnée utilisant des colonnes redondantes singulières,
- Réparation dynamique au niveau bit de donnée utilisant des colonnes redondantes singulières,
- Réparation basée sur les polarités d'erreurs,
- Réparation diversifiée bloc/bit de donnée,
- Réparation diversifiée ECC/mot mémoire.

Par contre on a généralisé la génération d'un modèle de mémoire et le calcul de l'efficacité à tous les flots de simulation possibles.
Nous allons dans ci-après détailler les différents modules utilisés dans les flots de simulation de l'outil d'injection de fautes.

5.5 Composantes de l'Outil d'Injection de Fautes

5.5.1 Les Paramètres Utilisateur

Se sont tous les paramètres nécessaires à une simulation en vue de calculer l'efficacité d'une technique de réparation. Nous pouvons classer les paramètres utilisateurs en trois catégories :

- Les paramètres propres à la mémoire régulière sur laquelle on veut appliquer l'algorithme de réparation, par exemple : le nombre de mots, le nombre de bits de données.
- Les paramètres propres à une technique de réparation donnée (il faut bien sûr faire un choix préalable de cette technique de réparation) pour déterminer le type, la taille et le comportement des éléments redondants. Par exemple, dans le cas de la réparation dynamique au niveau bit de donnée, l'utilisateur doit spécifier deux paramètres : le nombre de bits de données redondants (k) et le facteur (R) par lequel on divise une unité redondante en R unités réparables par bit de donnée. Dans le cas d'une réparation mixte ECC/mot mémoire, on doit spécifier le nombre de bits de données pouvant être

réparés par le code ECC ainsi que le nombre de mots CAM disponibles et le nombre de flags disponibles pour la réparation mot.

- Les paramètres correspondant au processus d'injection de fautes. On retrouve dans ce répertoire la probabilité d'occurrence des défauts ainsi que le nombre d'expériences (ou d'itérations) à effectuer. Ce dernier paramètre constitue la base de l'approche statistique.

5.5.2 Modèle Générique de la Mémoire

Dans un premier temps, une matrice mémoire software est générée en utilisant des tableaux en deux dimensions. Cette mémoire est ensuite entourée par de la redondance en fonction du type d'algorithme de réparation cible, toujours en utilisant des tableaux. L'injection de fautes est ainsi effectuée dans les cellules de la mémoire régulière et dans les cellules redondantes. La Figure 5.2 montre le modèle de mémoire générique utilisé pour la génération. Il regroupe tous les types de redondances dont on a besoin. La mémoire régulière peut être vue comme une matrice de Nm mots chacun de n bits, le nombre de bits d'adresse et alors donné par $m = \lceil \log_2 Nm \rceil$. A partir de ces paramètres (de la mémoire régulière), nous pouvant générer les divers redondances suivant la technique de réparation voulue :

- Réparation statique au niveau bit de donnée : en plus de la mémoire régulière, un tableau a deux dimensions de taille $k \times Nm$ (avec k représentant le nombre de bits de donner redondantes, choisi par l'utilisateur).
- Réparation dynamique au niveau bit de donnée : dans ce cas, la mémoire régulière ainsi que le tableau $k \times Nm$ sont divisés en R groupes de taille égale. Ainsi, chaque bit de donnée dispose de R unités réparables de taille Nm/R (en nombre de cellules) chacune. Le facteur R est en général une puissance de deux et peut exprimer de la manière suivante : $R = 2^r$, avec $1 \leq r \leq n$. Notons que le cas $r = 0$ correspond au cas de la réparation statique (point précédent). Aussi, le cas $r = n$ correspond à une taille de l'unité réparable égale à la taille de la cellule mémoire.
- Réparation au niveau bit de donnée utilisant des colonnes redondantes singulières : dans ce schéma de réparation, on dispose aux cotés de la mémoire régulière d'un nombre k de colonnes singulières. Par rapport à une colonne entière contenant N_m cellules mémoire, une colonne singulière va contenir $Nm/2^{nc}$ (nc = nombre de bits de l'adresse colonne) cellules mémoire. Ce schéma de redondance est le même que celui de la

réparation statique au niveau bit de donnée mais avec k unités redondantes de taille $Nm/2^{nc}$ chacune. Le comportement est par contre différent et nous le décrirons plus loin.

- Réparation dynamique au niveau bit de donnée utilisant des colonnes redondantes singulières : dans ce cas de figure on dispose de k unités redondantes (de Nm cellules mémoire chacune). Chacune de ces unités contient 2^{nc} colonnes singulières (de taille $Nm/2^{nc}$ chacune). On procède ensuite à un découpage de la mémoire régulière en R groupes avec $R = 2^r$, avec $1 \leq r \leq nc$. Chacun de ces R groupes disposera alors de $k \times 2^{nc-r}$. On notera que dans ce schéma de réparation, que le cas $r = 0$, correspond au schéma de redondance de la réparation statique utilisant $k \times 2^{nc}$ colonnes redondantes singulières (vu dans le point précédent). Aussi, le cas $r = nc$, est équivalent au schéma de redondance de la réparation dynamique au niveau bit de donnée (avec colonnes entières, vue dans le deuxième point) avec une taille de l'unité réparable égale à $Nm/2^{nc}$.
- Réparation basée sur les polarités d'erreurs : dans ce cas, le schéma de redondance est exactement le même que celui utilisé pour les quatre schémas précédents. En effet, il s'agit ici toujours d'une réparation au niveau bit de donnée. Par contre, le principe de réparation est complètement différent.
- Réparation diversifiée bloc/bit de donnée : en plus de l'un des quatre types de redondance correspondant à la réparation au niveau bit de donnée (exposées dans les quatre premiers points), il faut rajouter la redondance relative à la réparation au niveau bloc. Pour ce faire on va tout d'abord diviser la mémoire en $R = 2^r$ groupes (blocs) chacun contenant Nm/R mots mémoire (comme dans le cas dynamique), chaque mot mémoire comporte à son tour $n+k$ bits de données. La redondance bloc est réalisée en disposant d'un nombre q de ces blocs. Nous avons vu dans le chapitre précédent, que la réparation au niveau bloc nécessite une CAM. Or, cette CAM peut contenir des fautes, il faut donc l'ajouter comme faisant partie de la redondance afin de pouvoir injecter dans celle-ci des fautes. Cette CAM dans laquelle on stocke les r bits d'adresses des blocs réguliers fautifs, peut être représentée à l'aide d'un tableau à deux dimensions de taille $q \times (r+f)$, avec f représentant le nombre flags utilisés pour chaque mot de la CAM.
- Réparation diversifiée ECC/mot mémoire : dans ce cas de figure, le schéma de redondance est constitué d'une CAM. Celle-ci, et conformément à son architecture

matérielle présentée dans le chapitre précédent, peut être modélisée par un tableau à deux dimensions de taille $Nmc\times(n+m+f)$, avec Nmc représentant le nombre de mots mémoires redondants, n la taille de la donnée (pour les champs de donnée), m la taille de l'adresse (pour les champs d'adresse) et f le nombre de flags utilisés par chaque mot CAM.

Les schémas de redondance décrits ci-dessus, peuvent être assemblés en un seul modèle qui comprend quatre matrices à deux dimensions. Des dimensions sur lesquelles on peut agir en variant les paramètres cités plus haut (Nm, n, m, nc, k, r, q, Nmc, f). La première matrice, appelée *matrice régulière* et de dimension $n\times Nm$, correspond à la mémoire régulière et est obtenue en variant les paramètres Nm, n et m. La deuxième matrice, appelée *matrice de reconfiguration bit* et de dimension $k\times Nm$ dans le cas des colonnes redondantes entières et $k\times(Nm/2^{nc})$ dans le cas des colonnes redondantes singulières, correspond à l'ensemble des techniques de réparation au niveau bits de données, elle est obtenue en variant les paramètres nc et k (Nm étant déjà fixé par la matrice régulière).

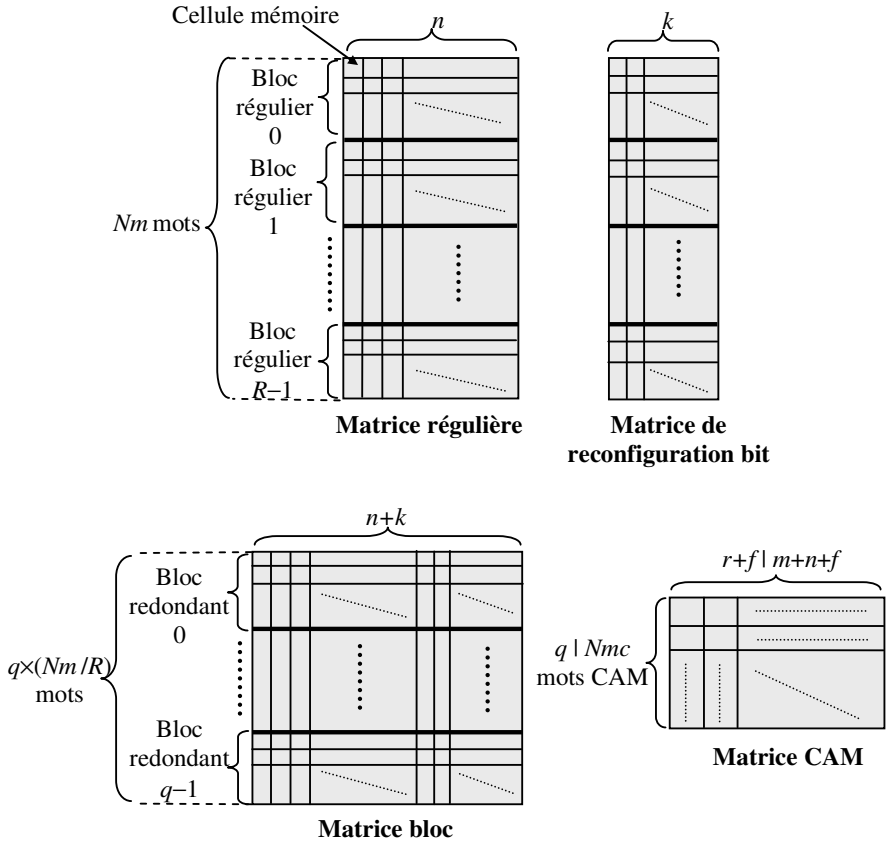


Figure 5.2. Modèle général de la redondance mémoire.

La troisième matrice, appelée *matrice bloc* et de dimension $(n+k)\times q$, correspond à la redondance bloc et est paramétrable en fonction de q (n étant déjà fixé par la matrice régulière et k par la matrice de reconfiguration bit). La dernière matrice, appelée matrice CAM et de dimension $(r+f)\times q$ dans le cas de la réparation bloc et $(n+m+f)\times Nmc$ dans le cas de la réparation mot. Ce modèle général de la mémoire tel qu'il est implémenté dans l'outil est présenté dans la Figure 5.2.

5.5.3 Le Module d'Injection de Fautes

Après avoir discuté la génération de la mémoire et de sa redondance pour chaque technique de réparation, on va s'intéresser à l'aspect injection de fautes proprement à parlé. L'injection de fautes ici est en fonction d'une probabilité globale de défaut par cellule mémoire (redondante ou régulier).

Dans [SEH03], l'injection de faute est basée sur la densité de défaut (D_d) exprimée en nombre de défauts par cm^2, celle-ci est ensuite utilisée par la formule (surface totale de la matrice). D_d / (le nombre total localités possibles du défaut). A partir de cette expression, la probabilité peut être calculée avec une bonne précision si D_d est ramenée au niveau de la cellule mémoire, car à ce moment là, le nombre de localités mémoire susceptibles d'être défectueuses est plus facile à estimer.

En effet, dans notre cas il ne s'agit pas seulement de défauts dans les cellules mais dans toutes les parties mémoire comme la logique de lecture écriture, les multiplexeurs, les lignes de bits (mot et colonne), …, etc. C'est pour éviter donc toute imprécision, que nous avons choisi de partir d'une probabilité globale calculée par les technologues, qui sont à même de connaître les surfaces mémoire (régulière et redondante) ainsi que le nombre de localités possibles pour un défaut. Ils pourront alors aisément fournir une probabilité de défaut globale (p) ramenée au niveau de la cellule mémoire. Nous pouvons des lors pratiquer une injection de fautes seulement sur les cellules de la matrice mémoire sans nous soucier du reste des parties logiques.

En partant de la probabilité de défaut globale p, une variable binaire (de valeur 0 ou 1) est générée pour chaque cellule mémoire. La valeur 1 et obtenue avec la probabilité p et désignera une cellule fautive, la valeur 0 est quant à elle obtenue avec la probabilité $1-p$ et désignera une cellule correcte. La mémoire générée est par conséquent initialisée à 0 avant de commencer toute expérience d'injection de faute.

Nous avons également implémenté la possibilité d'avoir une probabilité de faute spécifique aux éléments mémoire autres que la mémoire sous réparation. Nous pouvons prendre comme exemple les cellules mémoire des champs d'adresse de la CAM (utilisée dans la réparation mot et la réparation bloc). En considérant que la probabilité est la même pour deux éléments mémoire de tailles égales, on peut exprimer la probabilité de défaut pour cette cellule de la CAM comme étant $1-(1-p)^{\alpha}$ avec α représentant la proportion de taille entre la cellule CAM est la cellule mémoire sous réparation (de probabilité de défaut p).

Typiquement la probabilité p est construite via la génération d'un nombre tiré aléatoirement, puis en regardant si ce dernier appartient à une sous partie l d'un intervalle L. Les valeurs l et L doivent être judicieusement choisies de telle manière que $l = p.L$. Dans le cas particulier de la technique de réparation basée sur les polarités d'erreurs, on peut par exemple désigner par la valeur 1 la polarité d'erreur $0 \rightarrow 1$, par la valeur 2 la polarité d'erreur $1 \rightarrow 0$, par la valeur 3 la polarité d'erreur double $0 \leftrightarrow 1$, et toujours par la valeur 0 la cellule mémoire non fautive. Dans ce cas, la probabilité p est une somme de trois probabilités (p_1, p_2, et p_3) chacune liée à une polarité d'erreur, tandis que la probabilité $1-p$ désigne toujours la probabilité d'une cellule non fautive. De la même façon l'intervalle l est découpé en trois intervalles l_1, l_2, et l_3 de telle manière que $l_1 = p_1 .L$, $l_2 = p_2 .L$, et $l_3 = p_3 .L$.

Une attention particulière doit être octroyée à la construction de la probabilité p. En effet, la qualité de cette construction peut largement influencer les résultats des expérimentations. Il est donc important de générer des nombres qui soient le plus aléatoires possible, pour construire p. Nous avons intégré dans l'outil deux méthodes pour générer des nombres aléatoires. Une première méthode basée sur l'implémentation software d'un LFSR, et qui donne plutôt des nombres pseudo aléatoires. La deuxième méthode donne des nombres aléatoires de meilleure qualité et passe par la manipulation de la fonction rand() du langage C.

5.5.3.1 Types de fautes

Plusieurs types de fautes sont modélisés pour traduire l'occurrence de défauts dans la mémoire. L'outil a été adapté pour inclure les modèles de fautes suivants :

- Une cellule fautive singulière,
- Deux cellules fautives horizontales ou verticales, voisines du premier ordre,
- Un voisinage de quatre cellules fautives, sur l'horizontale et la verticale,
- Une ligne ou une colonne fautive.

D'autres types de fautes peuvent être implémentés dans l'outil en agissant sur le nombre de cellules fautives et leur disposition. Les cellules fautives sont comme convenu marquées par la valeur 1 et les cellules non fautives par la valeur 0.

Ces fautes apparaissent parce que les salles blanches ne peuvent être complètement décontaminées. Par conséquent, il y a une certaine probabilité qu'une particule de taille suffisante se dépose sur le wafer durant une des étapes du processus de fabrication. Mais comme la densité de grandes particules décroît rapidement, les fautes leur incombant (deux cellules fautives ou plus) ont une probabilité d'occurrence significativement moins élevée comparée aux fautes singulières.

Dans la pratique, l'injection des fautes se fait en considérant une densité de défaut globale D_d qu'on distribue avec différentes proportions (pr_i) suivant les types de fautes i que l'utilisateur désire. L'utilisateur peut être guidé dans le choix des proportions de telle manière que le nombre moyen des fautes ($n_c.D_d$, avec n_c représentant le nombre total des cellules mémoire) reste à peu près maintenu, c'est-à-dire respectant l'équation :

$$n_c . D_d \approx \sum_i \left[(n_c / n_i) . pr_i . D_d \right].$$

5.5.4 Module d'évaluation et de Réparation

Une fois l'injection de fautes achevée, on évalue si la technique de réparation choisie réussit à réparer la mémoire fautive. Pour ce faire on décrit par des programmes software le comportement des différentes architectures matérielles présentées dans les chapitres III et IV. Ces programmes ont été intégrés dans le *module d'évaluation de la réparation*. Ce module lit séquentiellement les cellules des matrices mémoire, simule le comportement du BIST et du BISR, teste les conditions de réussite de la réparation, et enfin déclare si la mémoire fautive est réparable ou non.

La nature software du module d'évaluation de la réparation a l'avantage d'effectuer une simulation beaucoup plus rapide que la simulation matérielle et permet ainsi de réduire le temps global d'expérimentations. En effet, dans une simulation purement matérielle, le circuit BIST exécute un algorithme de test afin de détecter les différents types de fautes. Il peut aussi être utilisé plusieurs fois notamment dans la réparation bloc et mot. Cependant, les techniques de réparation qu'on a développé sont indépendantes des types de fautes (collages, fautes de transitions, couplages, …) et les différentes sessions de test sont évitées par une simple injection de faute sur les matrices mémoire suivie d'une seule lecture de ces dernières.

A titre d'exemple, nous présentons ci-dessous quelques algorithmes qui décrivent les conditions de réussite du processus de réparation pour quelques unes des techniques BISR du chapitre III et IV.

La Figure 5.3 montre l'algorithme de la réparation statique au niveau bit de donnée. Dans cet algorithme on dispose d'un compteur (count1) qui compte le nombre de bits de donnée erronés. Un bit de donnée est déclaré fautif dès qu'une cellule liée à ce bit est fautive (break CellScan). Dès que la valeur de count1 dépasse le nombre d'unités redondantes (k), il y a échec de la réparation (break BitScan).

```
BitScan: Pour chaque colonne de (matrice régulière + matrice de reconfiguration bit) do
    CellScan: for j: 0 to Nm−1 do
      if (cell[j] = 1) then count1 = count1+1, break CellScan;
    end CellScan;
    if (count1 > k ) then repair fails, break BitScan;
end BitScan;
```

Figure 5.3. Algorithme de la réparation statique au niveau bit de donnée.

La Figure 5.4 montre l'algorithme de la réparation dynamique au niveau bit de donnée. Cet algorithme agit d'une manière similaire à l'algorithme précédent mais sur des parties plus petites (R blocs). Il y a une condition supplémentaire par rapport à l'algorithme précédent et qui stipule que chaque bloc doit être réparable (break BlockScan).

```
BlockScan: for i: 0 to R−1 do
  count2 = 0;
  BitScan: Pour chaque colonne de (matrice régulière + matrice de reconfiguration bit) do
    CellScan: for j: i.Nm/R to ((i+1).Nm /R)−1 do
      if (cell[j] = 1) then count1 = count1+1, break CellScan;
    end CellScan;
    if (count2 > k ) then repair fails, break BitScan, break BlockScan;
  end BitScan;
end BlockScan;
```

Figure 5.4. Algorithme de la réparation dynamique au niveau bit de donnée.

La Figure 5.5 montre un dernier exemple de description des techniques de réparation intégrée. Il s'agit de la technique qui combine la réparation au niveau bit de donnée et la réparation au niveau bloc. Dans cet algorithme, on compte (count2) le nombre de blocs réguliers non réparables à l'aide des k colonnes redondantes et on voit s'ils peuvent être remplacés par les blocs redondants en tenant compte des fautes dans la CAM. Un mot CAM est considéré comme erroné sous les conditions suivantes sont satisfaites : (1) au moins une

cellule fautive existe dans le champ d'adresse de ce mot (break AdrressScan), (2) il existe moins de deux flags corrects dans ce mot (break FlagScan).

```
BlockScan: for i: 0 to R+q−1 do//q : nombre de blocs redondants
  count1 = 0;
  BitScan: for each column of the main and spare data matrixes do
    CellScan: for j: i.Nm/R to ((i+1).Nm /R)−1 do
     if (cell[j] = 1) then count1 = count1+1, break CellScan;
    end CellScan;
    if (count1 > k ) then count2 = count2+1, break BitScan;
  end BitScan;
if (i = R−1 and count2 = 0) then repair succeed;
if (count2 > q) then repair fails, break BlockScan;
end BlockScan;
if(count2 = 0) then repair succeed;
else if (count2 > q) then repair fails;
else{//0 < count2 ≤ q
   CamScan: for i: 0 to q−1 do
     if(count2 ≤ q) then
       AddressScan: for j: 0 to r−1 do
         if (CAM[i][j] = 1) then count2 = count2+1, break AddressScan;
       end AddressScan;
     else repair fails, break CamScan;
     count3 = 0;
     if(count2 ≤ q) then
       FlagScan: for s: r to r+f−1 do
         if (CAM[i][s] = 1) then count3 = count3+1;
         if (coun3 > 2) then count2 = count2+1, break FlagScan;
       end FlagScan;
     else repair fails, break CamScan;
   end CamScan;
    }
```

Figure 5.5. Algorithme de la réparation diversifiée bit de donnée/bloc.

5.5.5 Module d'Evaluation Statistique de l'Efficacité de Réparation pour une Instance Mémoire

Cette évaluation statistique passe par la réalisation d'un certain nombre d'expérimentations au bout desquelles, on calcule le taux de réussite du processus de réparation. Une expérimentation commence par la création des parties mémoire régulières et redondantes suivant des dimensions fixes, suivie d'une injection de fautes selon une probabilité constante, et finit par le déroulement de l'algorithme de réparation qu'on désire évaluer (Figure 5.1(a)). Ainsi, la seule variable d'une expérimentation à une autre, est la configuration des fautes dans les parties mémoire.

Dans la pratique, après chaque expérience d'injection de faute et d'évaluation, le module de calcul de l'efficacité, stocke le nombre de mémoires non réparables dans la variable N_n. Le

module lit également le nombre d'expérimentations a réaliser N_e. A la fin des N_e expériences, le module calcule l'efficacité (rendement) comme suit :

$$Rend(RAM_i, p_i, N_e) = 1 - \frac{N_n}{N_e} \ldots \textbf{(5.1)}$$

Le choix de N_e est très important car il représente le nombre minimum d'expérimentations pour lesquelles on obtient des résultats (efficacité) convergents. Ce nombre peut aussi bien être choisi par l'utilisateur, ou bien être généré par l'outil. Dans ce dernier cas, l'outil calcule et enregistre après chaque expérience d'injection de fautes, l'efficacité de la réparation et la compare avec les valeurs antérieures afin de vérifier la convergence des résultats. Lorsque après un certain nombre d'expériences, l'efficacité se stabilise à plus ou moins une valeur δ (que l'utilisateur spécifie), l'expérience s'arrête et la valeur courante de l'efficacité est reportée.

5.5.6 Modèle d'Amas de Fautes (Clustering Model)

Plusieurs travaux de recherche sur le modèle d'amas de fautes ont été effectués et présentés dans la littérature [BLO93], [BLO96], [YAN72], [STA80], [STA86], [NOW87], [STA89a], [STA89b], [STA92]. Pour notre étude nous avons considéré le modèle décrit dans [STA89a], [STA89b] qui parait être l'approche la plus précise pour modéliser la distribution des fautes sous forme d'amas. Afin d'introduire ce modèle nous avons développé un programme qui injecte des fautes dans les puces du wafer, en utilisant plusieurs itérations d'injection de fautes. A chaque itération, le programme détermine la probabilité avec laquelle on effectue l'injection de fautes dans une puce. Cette probabilité est une fonction linéaire du nombre de fautes injectées dans la puce et dans ses voisins durant l'itération précédente [STA89b]. Nous avons adopté dans notre outil un voisinage de quatre puces comme cela a été proposé dans [STA 89a]. L'équation utilisée à chaque itération pour produire cet effet est :

$$P_{f1}(N_{fp}, N_{fvp}) = a + bN_{fp} + cN_{fvp} \ldots \textbf{(5.2)}$$

Dans cette expression, N_{fp} représente le nombre de fautes injectées dans la puce en question durant l'itération précédente, et N_{fvp} représente le nombre de fautes injectées dans les puces voisines durant l'itération précédente. Les constantes a, b, et c représentent respectivement l'effet sans amas de fautes, l'effet clustering des fautes de la puce, et l'effet clustering dû aux fautes concentrées dans les puces voisines. Si pour une puce, la valeur de P_{f1} devient supérieure à une certaine valeur $maxP_{f1}$ (choisie par l'utilisateur), nous divisons les

probabilités de toutes les puces par cette valeur. Ceci est effectué au début de chaque itération avant l'injection de fautes.

La difficulté dans cette approche réside dans le choix des paramètres a, b, et c ainsi que le nombre d'itérations. Ceci peut être effectué avec une bonne précision uniquement si nous possédons des données statistiques relatives à la distribution des fautes. A cause du manque de telles données, considérées comme strictement confidentielles par les différentes compagnies, mais aussi parce que les données actuelles ne sont pas pertinentes pour le cas des nanotechnologies pour lesquelles notre étude est destinée, nous avons sélectionné quelques valeurs qui donnent des effets de clustering raisonnables ($a = 1/40$, $b = 0.5a$, $b = 0.2a$, et $maxP_{f1} = 0.5$). Néanmoins, toutes ces valeurs peuvent être sélectionnées par l'utilisateur. Les itérations sont stoppées lorsque le nombre total de fautes injectées dans le wafer atteint une certaine valeur.

Lorsque l'injection de faute au niveau wafer est finie, on calcule la densité de défaut p_{fi} pour la puce i comme une fonction de la densité de défaut du wafer p_{fw} :

$$P_{fi} = e + (P_f - e)N_iN_p/N_t \quad \ldots \textbf{(5.3)}$$

Dans cette expression N_i est le nombre de fautes injectées dans la puce i, N_p le nombre de puces dans le wafer, et N_t le nombre total des fautes injectées dans le wafer. Cette expression maintient la valeur moyenne de la densité de défaut pour chaque puce égale à la densité de défaut du wafer p_{fw}. Le terme e annule l'effet cluster pour les fautes injectées dans les cellules d'une puce. Le facteur de clustering N_iN_p/N_t est proportionnel au nombre de fautes N_i injectées dans la puce i durant les itérations de clustering.

[STA89a] a également suggéré la possibilité d'introduire trois niveau de clustering (grandes surfaces, surfaces intermédiaires, et petites surfaces). A chaque niveau le même programme de clustering peut être utilisé mais avec des paramètres différents. Nous avons introduit un niveau intermédiaire clustering en partitionnant chaque puce en $U \times V$ modules, pour effectuer l'injection de fautes au niveau puce. Pour la puce i du wafer, à chaque itération une faute est injectée dans le module j en utilisant la probabilité d'injection de faute donnée par l'expression :

$$P_{f2}(N_{fm}, N_{fvm}) = N_iN_p/N_t + b' N_{fm} + c' N_{fvm} \quad \ldots \textbf{(5.4)}$$

Dans cette expression, N_{fm} représente le nombre de fautes injectées dans le module durant la précédente itération, et N_{fvm} le nombre de fautes injectées dans les modules voisins durant l'itération précédente. Dans ce niveau de clustering nous pouvons utiliser des constantes b' et

c' similaires à c et b, ou différentes si des données statistiques indiquent un effet de clustering différent au niveau module.

Lorsque ce calcul est achevé, nous effectuons une injection de fautes sur les cellules de la puce i. Durant ce processus d'injection, les fautes sont injectées dans les cellules des différents modules en utilisant différentes densités de défauts. Pour ce faire, nous utilisons pour les cellules du module j des densités de défauts calculées comme expliqué précédemment suivant l'expression :

$$P_{fj} = e + (P_{fc} - e)N_j N_{mo}/N_{tm} \quad \ldots \textbf{(5.5)}$$

Dans cette l'expression (5.5), N_j représente le nombre de fautes injectées dans le module j durant le processus de clustering au niveau module, N_{tm} le nombre total de fautes injectées dans la puce durant le même processus, et N_{mo} le nombre de modules qui constituent la puce ($U.V$). Les valeurs P_{fj} ainsi calculées sont utilisées pour l'injection de fautes au niveau mémoire, en considérant que chaque module représente une instance mémoire.

5.5.6 Evaluation Statistique du Rendement au Niveau Wafer

Le but de cette section est de décrire une approche software qui permet d'évaluer le rendement induit par les techniques BISR, au niveau wafer. Pour modéliser un wafer de mémoires, nous utilisons une matrice de dimension $N_{mo}.N_{ch}$, chacune des cellules de cette matrice représentant une instance mémoire. On applique sur cette matrice le programme de clustering, comme décrit ci-dessus, et qui détermine pour chaque instance mémoire du wafer la densité de défaut (P_{fj}) avec laquelle on va effectuer l'injection de fautes. Il reste alors à déterminer individuellement pour chaque mémoire, le rendement statistique exactement suivant la méthode décrite dans la section 5.4 et représentée par la Figure 5.1(a).

Le rendement statistique d'un seul wafer y, peut être exprimé par l'expression suivante :

$$Rend^{y}_{wafer} = \frac{\sum_x Rend_x}{N_{mo}.N_{ch}}, \text{ avec } x \in \{1, 2, \ldots, N_{mo}.N_{ch}\} \quad \ldots \textbf{(5.6)}$$

Dans l'équation (5.6), $Rend_x$ représente l'efficacité (ou le rendement) d'une instance mémoire x, calculé suivant l'expression (5.1).

L'expression (5.6) correspond à une seule configuration des densités de défauts dans le wafer. Afin d'étudier l'effet d'une technique BISR sur différentes configurations des densités de défauts dans le même wafer, il convient de répéter le même programme de clustering un

certain nombre de fois. Le rendement final au niveau wafer, serait alors le résultat d'une combinaison de deux expériences statistiques : (1) une expérience statistique au niveau wafer, réalisée par l'application du même programme de clustering, et au cours de laquelle différentes distributions de densités de défauts sont considérées, (2) une expérience statistique au niveau de l'instance mémoire, réalisée par une série d'injections de fautes pour une densité de défaut donnée, et cours de laquelle différentes distributions de fautes sont considérées. La Figure 5.1(b) illustre la relation entre les deux expériences statistiques pour le calcul du rendement final.

Considérons que N_{ew} expériences ont été réalisées en utilisant le même programme de clustering, nous pouvons alors exprimer le rendement final au niveau wafer comme :

$$\frac{\sum_y Rend_{wafer}^{y}}{N_{ew}}, \text{ avec } y \in \{1, 2, \ldots, N_{ew}\} \ldots \textbf{(5.7)}$$

Dans l'équation (5.7), $Rend_{wafer}^{y}$ représente le rendement d'un seule wafer et dont l'expression est donnée par l'équation (5.6).

5.5.7 Mixage des Approches Statistique et Analytique pour le Calcul du Rendement au Niveau Wafer

Nous avons vu dans les sections précédentes, l'utilité de développer un programme de clustering pour l'évaluation du rendement au niveau wafer. Néanmoins, la réalisation d'une expérience purement statistique au niveau wafer, prendrait dans la pratique un temps considérable. Nous pouvons illustrer ce problème en prenant comme exemple un wafer d'une taille moyenne contenant 100 mémoires, si on considère que pour chacune de ces mémoires on réalise 100 expériences par densité de défaut, on obtient 10000 expériences par wafer. Si on multiplie ce chiffre par le nombre de wafers (par exemple 100) de configurations différentes (en densités de défauts), on obtient un nombre total d'expériences égal à 100000. En supposant que chaque expérience dure 10 secondes, il nous faut donc compter approximativement 270 heures !

Afin de réduire ce temps de simulation, nous pouvons utiliser des formules analytiques pour calculer le rendement en fonction de la densité de défaut. Ainsi nous éviterons de réaliser les expériences statistiques (consommatrices de temps) relatives à chaque instance mémoire. Nous devons alors élaborer pour chaque technique de réparation qu'on veut évaluer, une

formule mathématique spécifique. Nous avons déjà mentionné qu'un tel procédé n'est pas toujours aisé surtout quand une technique de réparation devient compliquée.

Nous avons pu établir pour certaines des techniques de réparation qu'on a développé, les formules mathématiques permettant le calcul de la probabilité qu'une mémoire soit réparable. Cette probabilité est une fonction de la densité de défaut (au niveau de la cellule mémoire) et le nombre de ressources redondantes. L'expression théorique de cette probabilité passe par le dénombrement de toutes les configurations de distribution de fautes pour lesquelles la mémoire est réparable en présence d'une technique de réparation donnée (bien entendu on peut raisonner sur les configurations de fautes résultants en une mémoire non réparable). Le rendement est alors obtenu en exprimant cette probabilité en terme de pourcentages.

Nous donnons ci-dessous quelques une de ces formules analytiques[3] :

- Réparation statique au niveau bits de donnée :

$$Rend_{stat} = \sum_{s=0}^{k} C_{n+k}^{n+s} \left(1-p_f\right)^{Nm(n+s)} \left(1-\left(1-p_f\right)^{Nm}\right)^{k-s} \quad \ldots \textbf{(5.8),}$$

k représente le nombre de bits (de donnée) redondants.

Nm représente le nombre total de mots de la mémoire.

- Réparation dynamique au niveau bits de donnée :

$$Rend_{Dyn} = \left[\sum_{s=0}^{k} C_{n+k}^{n+s} \left(1-p_f\right)^{B(n+s)} \left(1-\left(1-p_f\right)^{B}\right)^{k-s} \right]^{2^r} \quad \ldots \textbf{(5.9)},$$

r est le nombre de bits d'adresses utilisés pour la réparation dynamique.

$B = Nm/2^r$.

- Réparation combinée bits de donnée et bloc :

$$Rend_{Dyn\&bloc} = \sum_{t=0}^{q} C_{2^r+q}^{2^r+t} \left[\sum_{s=0}^{k} C_{n+k}^{n+s} \left(1-p_f\right)^{B(n+s)} \left(1-\left(1-p_f\right)^{B}\right)^{k-s} \right]^{2^r+t} *$$

$$\left[1-\sum_{s=0}^{k} C_{n+k}^{n+s} \left(1-p_f\right)^{B(n+s)} \left(1-\left(1-p_f\right)^{B}\right)^{k-s} \right]^{q-t} \quad \ldots \textbf{(5.10)},$$

q représente le nombre de blocs redondants.

[3] Dans ces formules, les doubles fautes et les quadri fautes ne sont pas prises en compte. Seules les fautes singulières sont considérées.
Pour rappel : $C_a^{\ b} = a!/[b!(a-b)!]$.

Une limitation importante de ces formules survient lorsque p prends des petites valeurs ($< 10^{-3}$) rendant les termes … très petits, dépassant ainsi les capacités de calcul des ordinateurs. La solution est donc d'utiliser, quand elles sont faciles à élaborer, systématiquement les formules analytiques pour une mémoire donnée et si jamais on ne peut obtenir un résultat à cause des limitations de calcul, alors on emploie une série d'expériences statistiques. Ainsi, l'approche mixte analytique/statistique permet la détermination du rendement au niveau wafer quelque soit les densités de défauts mises en jeu et cela en un temps raisonnable.

5.6 Expérimentations et Résultats

Afin d'évaluer l'efficacité de réparation des techniques BISR pour de grandes densités de défauts ainsi que le coût en surface, nous avons effectué un grand nombre d'expériences sur une mémoire de 1 Mbit utilisant des mots de 32 bit.

Dans ces expériences, nous avons considéré les densités de défauts suivantes : $D_d = 1\times10^{-5}$, $D_d = 2\times10^{-5}$, $D_d = 1\times10^{-4}$, $D_d = 3\times10^{-4}$, $D_d = 1\times10^{-3}$, $D_d = 3\times10^{-3}$, et $D_d = 1\times10^{-2}$. De telles densités de défauts sont supérieures de deux ordres de grandeur par rapport aux densités de défauts qu'on peut rencontrer dans les technologies des mémoires actuelles. Pour ces densités de défauts, les expérimentations ont montré que le rendement des mémoires, non munies de réparation, était nul.

5.6.1 Evaluation de la Technique de Réparation Combinée Dynamique/Bloc

Les Tableaux de 5.1 à 5.5 montrent les résultats pour la technique combinant la réparation dynamique au niveau bit de donnée avec la réparation bloc. Dans ces expériences, nous avons considéré les éclats de fautes (ou burst faults) de telle manière que la densité de défaut des fautes doubles représentent 10% de la densité de défaut des fautes singulières, et que la densités de défaut des quadri fautes représentent 10% de la densité de défaut des fautes doubles. La somme de ces densités de défaut de ces fautes donne la densité de défaut globale D_d. Nous indiquons au-dessus de chaque tableau la taille de la mémoire, la longueur du mot, la densité de défaut globale D_d, et le nombre moyen de fautes dans la mémoire régulière[4] pour la densité de défaut globale considérée. Les deux premières colonnes de ces tableaux contiennent les valeurs des paramètres de la réparation dynamique au niveau bit : k (#bits redondants), et r (#bits d'adresse utilisés pour diviser une colonne redondante en R unités

[4] Le nombre moyen de fautes dans les parties redondantes de la mémoire n'est pas indiqué.

réparables, $R = 2^r$). Dans le reste des colonnes, on donne l'efficacité et le coût en surface en termes de pourcentage par rapport à la mémoire sans technique de réparation. Ces pourcentages sont donnés pour différentes valeurs du nombre q de blocs redondants[5].

Les résultats du tableau 1 sont donnés pour la densité de défaut globale $D_d = 1\times10^{-5}$. Le rendement sans réparation est de seulement 2%. Nous observons que pour les petites valeurs de r, la réparation dynamique au niveau bit de donnée est plus efficace que la réparation combinée. Par exemple pour $r = 2$, la réparation diversifiée enregistre un rendement de 62% avec un surcoût en surface de 37% (cas $k = 3$, $r = 2$, $q = 1$), tandis qu'une simple réparation dynamique obtient un rendement de 88% pour un surcoût en surface estimée a 13% (cas $k = 4$, $r = 2$, $q = 0$). Cela est dû au fait que pour les petites valeurs de r, la taille d'un bloc (égale a la taille de la mémoire divisée par 2^r) est plus grande, rendant inefficace l'utilisation des blocs redondants. Nous pouvons d'ailleurs voir que l'efficacité de la technique combinée ($q \neq 0$) augmente considérablement dès qu'on augmente le paramètre r. Toutefois, pour $D_d = 1\times10^{-5}$ la réparation combinée n'apporte aucune amélioration à l'efficacité de réparation en comparaison avec la seule réparation dynamique au niveau bit. Les deux techniques arrivent à faire passer le rendement d'une mémoire sans réparation de 2% à un rendement approchant les 100% pour un surcoût modéré en surface. Par exemple pour $k = 2$, $r = 5$, $q = 1$, la technique combinée réalise un rendement de 97% pour un surcoût en surface de 14%, tandis que pour $k = 3$, $r = 5$, $q = 0$, la réparation dynamique simple obtient un rendement de 96% pour un surcoût en surface de 13,7%.

Tableau 5.1 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-5}$ (~ 10 défauts)

k	*r*	*q* = 0	*q* = 1	*q* = 2	*q* = 3	*q* = 4	*q* = 5	*q* = 6	*q* = 7
1	**2**	1 /3,66	4/29,58	6 /55,49	14/81,41	24/107.3	29/133.24	38/159.16	43/185,1
1	**3**	3/4,20	14/17,23	35/30,25	53 /43,28	66/56,31	75/69,33	81/82,36	86/95,38
1	**4**	11/5,28	37/11,86	60/18,44	73/25,03	87/31,61	95/38,19	99/44,77	100/51,35
1	**5**	17/7,44	56/10,80	77/14,16	91/17,52	97/20,88	100/24,23	100/27,59	100/30,95
1	**6**	27/11,75	73/13,51	85/15,25	97/17	99/18,75	100/20,5	100/22,24	100/23,99
1	**7**	39/20,40	79/21,34	90/22,28	99/23,23	100/24,17	100/25,11	100/26,05	100/26,99
1	**8**	46/37,68	83/38,22	94/38,76	100/39,3	100/39,84	100/40,37	100/40,91	100/41,45
2	**2**	13/6,79	30/33,48	43/60,18	56/86,88	73/113,6	83/140,3	91/167	95/193,7
2	**3**	38/7,33	71/20,74	83/34,16	93/47,58	98/61	100/74,41	100/87,83	100/101,2
2	**4**	62/8,41	92/15,18	95/21,96	100/28,74	100/35,51	100/42,29	100/49,06	100/55,84
2	**5**	78/10,57	97/14,02	100/17,48	100/20,94	100/24,39	100/27,85	100/31,3	100/34,76
2	**6**	91/14,89	98/16,68	100/18,48	100/20,28	100/22,07	100/23,87	100/25,66	100/27,46
2	**7**	95/23,53	100/24,49	100/25,46	100/26,43	100/27,39	100/28,36	100/29,32	100/30,29

[5] La colonne de valeur $q = 0$ représente l'efficacité de la simple réparation dynamique au niveau bit.

2	**8**	97/40,81	100/41,36	100/41,91	100/42,46	100/43,01	100/43,56	100/44,11	100/44,66
3	**2**	38/9,91	67/37,39	85/64,87	90/92,35	95/119,8	99/147,3	99/174,8	99/202,3
3	**3**	62/10,45	93/24,26	99/38,06	100/51,88	100/65,68	100/79,49	100/93,3	100/107,1
3	**4**	84/11,53	99/18,50	100/25,47	100/32,45	100/39,42	100/46,39	100/53,36	100/60,33
3	**5**	96/13,69	99/17,24	100/20,80	100/24,35	100/27,91	100/31,5	100/35	100/38,57
3	**6**	98/18,01	100/19,85	100/21,70	100/23,55	100/25,39	100/27,2	100/29,1	100/30,92
3	**7**	100/26,65	100/27,64	100/28,63	100/29,62	100/30,61	100/31,6	100/32,6	100/33,58
4	**2**	88/13,04	99/41,3	100/69,56	100/97,82	100/126,1	100/154,3	100/182,6	100/210,9
4	**3**	100/13,58	100/27,77	100/41,97	100/56,1	100/70,37	100/84,57	100/98,77	100/113
4	**4**	100/14,66	100/21,82	100/28,99	100/36,16	100/43,33	100/50,49	100/57,66	100/64,82
4	**5**	100/16,82	100/20,47	100/24,12	100/27,77	100/31,42	100/35,07	100/38,72	100/42,37

Dans le Tableau 5.2, la même mémoire est considérée mais la densité de défaut globale est deux fois plus grande (D_d = 2×10^{-5}). Dans ce cas aussi nous observons que les meilleurs résultats obtenus pour la réparation dynamique simple, sont similaires à ceux obtenus avec la réparation combinée. Par exemple, pour k = 2, r = 6, q = 2, la technique combinée donne un rendement de 99% pour un surcoût en surface de 18,48%, tandis que pour k = 4, r = 5, q = 0, la réparation dynamique simple obtient un rendement de 98% pour un surcoût en surface de 16,82%.

Tableau 5.2 Mémoire de 1Mbit, mot de 32 bit, D_d = 2×10^{-5} (~ 20 défauts)

k	*r*	*q* = 0		*q* = 1		*q* = 2		*q* = 3		*q* = 4		*q* = 5		*q* = 6		*q* = 7	
1	**3**	0	4,20	0	17,23	0	30,26	2	43,28	3	56,31	4	69,33	6	82,36	7	95,38
1	**4**	1	5,28	3	11,87	6	18,45	10	25,03	18	31,61	25	38,19	36	44,77	44	51,35
1	**5**	3	7,44	8	10,8	17	14,16	28	17,52	46	20,88	60	24,23	76	27,59	86	30,95
1	**6**	8	11,77	21	13,51	36	15,26	64	17	82	18,75	92	20,5	98	22,24	98	23,99
1	**7**	17	20,41	36	21,35	66	22,29	82	23,23	94	24,17	98	25,11	99	26,05	100	26,99
1	**8**	21	37,69	54	38,22	77	38,76	91	39,3	97	39,84	99	40,37	100	40,91	100	41,45
2	**2**	0	6,79	0	33,49	0	60,19	0	86,88	1	113,6	3	140,3	4	167	5	193,7
2	**3**	3	7,33	5	20,75	8	34,16	20	47,58	30	61	35	74,41	48	87,83	61	101,2
2	**4**	11	8,41	36	15,19	54	21,96	74	28,74	87	35,51	96	42,29	100	49,06	100	55,84
2	**5**	33	10,57	66	14,03	86	17,48	98	20,94	100	24,39	100	27,85	100	31,3	100	34,76
2	**6**	67	14,89	92	16,69	99	18,48	100	20,28	100	22,07	100	23,87	100	25,66	100	27,46
2	**7**	86	23,53	99	24,5	100	25,46	100	26,43	100	27,39	100	28,36	100	29,32	100	30,29
2	**8**	92	40,81	99	41,36	100	41,91	100	42,46	100	43,01	100	43,56	100	44,11	100	44,66
3	**2**	1	9,915	3	37,39	4	64,87	9	92,35	12	119,8	18	147,3	23	174,8	33	202,3
3	**3**	11	10,46	34	24,26	56	38,07	73	51,88	89	65,68	96	79,49	97	93,3	98	107,1
3	**4**	51	11,54	81	18,51	93	25,48	98	32,45	100	39,42	100	46,39	100	53,36	100	60,33
3	**5**	86	13,7	96	17,25	99	20,8	100	24,35	100	27,91	100	31,46	100	35,01	100	38,57
3	**6**	97	18,02	100	19,86	100	21,7	100	23,55	100	25,39	100	27,23	100	29,08	100	30,92

3	**7**	99	26,66	100	27,64	100	28,63	100	29,62	100	30,61	100	31,6	100	32,59	100	33,58
3	**8**	100	43,94	100	44,5	100	45,06	100	45,62	100	46,18	100	46,75	100	47,31	100	47,87
4	**2**	4	13,04	11	41,3	22	69,56	38	97,82	45	126,1	58	154,3	67	182,6	76	210,9
4	**3**	33	13,58	72	27,78	89	41,98	97	56,17	98	70,37	100	84,57	100	98,77	100	113
4	**4**	83	14,66	97	21,83	100	28,99	100	36,16	100	43,33	100	50,49	100	57,66	100	64,82
4	**5**	98	16,82	99	20,47	100	24,12	100	27,77	100	31,42	100	35,07	100	38,72	100	42,37
5	**2**	10	16,17	33	45,21	48	74,25	65	103,3	74	132,3	87	161,4	95	190,4	97	219,5
5	**3**	65	16,71	92	31,29	98	45,88	100	60,47	100	75,06	100	89,65	100	104,2	100	118,8
5	**4**	96	17,79	100	25,15	100	32,51	100	39,87	100	47,23	100	54,59	100	61,95	100	69,32
6	**2**	27	19,29	64	49,11	79	78,94	89	108,8	96	138,6	98	168,4	99	198,2	100	228
6	**3**	85	19,83	98	34,81	100	49,79	100	64,77	100	79,75	100	94,72	100	109,7	100	124,7
6	**4**	100	20,91	100	28,47	100	36,02	100	43,58	100	51,14	100	58,69	100	66,25	100	73,81

Dans le Tableau 5.3, la même mémoire est toujours considérée mais avec une densité de défaut plus grande ($D_d = 10^{-4}$). Dans ce cas nous observons que la technique de réparation combinée est légèrement meilleure que la simple réparation dynamique. Par exemple, pour $k = 4$, $r = 6$, $q = 4$ la technique combinée donne un rendement de 98% pour un coût en surface de 28,71%, tandis que pour $k = 7$, $r = 6$, $q = 0$, la réparation dynamique simple obtient un rendement de 97% pour un surcoût en surface de 30,52%.

Au delà de ces considérations, nous pouvons souligner le fait que pour une telle haute densité de défauts (10^{-4}), notre BISR atteint un rendement supérieur à 98% avec un surcoût modéré en surface (28,71%).

Tableau 5.3 Mémoire de 1Mbit, mot de 32 bit, $D_d = 10^{-4}$ (~ 100 défauts)

k	***r***	***q* = 0**		***q* = 1**		***q* = 2**		***q* = 3**		***q* = 4**		***q* = 5**		***q* = 6**		***q* = 7**	
2	**6**	0	14,89	0	16,69	0	18,48	0	20,28	0	22,07	0	23,87	0	25,66	0	27,46
2	**7**	0	23,53	0	24,5	2	25,46	5	26,43	10	27,39	17	28,36	39	29,32	63	30,29
2	**8**	5	40,81	18	41,36	38	41,91	64	42,46	79	43,01	92	43,56	97	44,11	98	44,66
2	**9**	24	75,37	61	75,71	86	76,06	99	76,4	100	76,74	100	77,08	100	77,43	100	77,77
2	**10**	55	144,5	87	144,7	98	145	99	145,2	100	145,4	100	145,7	100	145,9	100	146,2
3	**5**	0	13,7	0	17,25	0	20,8	0	24,35	0	27,91	0	31,46	0	35,01	0	38,57
3	**6**	1	18,02	1	19,86	3	21,7	7	23,55	26	25,39	39	27,23	55	29,08	72	30,92
3	**7**	12	26,66	37	27,64	73	28,63	87	29,62	99	30,61	99	31,6	100	32,59	100	33,58
3	**8**	57	43,94	89	44,5	100	45,06	100	45,62	100	46,18	100	46,75	100	47,31	100	47,87
3	**9**	84	78,5	100	78,84	100	79,19	100	79,54	100	79,89	100	80,24	100	80,59	100	80,94
3	**10**	94	147,6	100	147,9	100	148,1	100	148,3	100	148,6	100	148,8	100	149,1	100	149,3
4	**5**	0	16,82	0	20,47	0	24,12	1	27,77	6	31,42	8	35,07	14	38,72	24	42,37

4	**6**	13	21,14	35	23,03	64	24,93	83	26,82	98	28,71	100	30,6	100	32,5	100	34,39
4	**7**	71	29,78	94	30,79	100	31,81	100	32,82	100	33,84	100	34,85	100	35,86	100	36,88
4	**8**	94	47,06	100	47,63	100	48,21	100	48,78	100	49,36	100	49,93	100	50,51	100	51,08
4	**9**	99	81,62	100	82	100	82,3	100	82,7	100	83	100	83,39	100	83,75	100	84,1
4	**10**	100	150,7	100	151	100	151	100	151	100	152	100	152	100	152,2	100	152
5	**5**	2	19,95	4	23,69	20	27,44	36	31,19	56	34,94	68	38,69	84	42,43	92	46,18
5	**6**	55	24,27	84	26,21	96	28,15	100	30,09	100	32,03	100	33,97	100	35,91	100	37,86
5	**7**	91	32,91	100	33,94	100	34,98	100	36,02	100	37,06	100	38,1	100	39,13	100	40,17
5	**8**	99	50,19	100	50,77	100	51,36	100	51,94	100	52,53	100	53,12	100	53,7	100	54,29
6	**5**	11	23,07	36	26,92	61	30,76	82	34,61	93	38,45	100	42,3	100	46,15	100	49,99
6	**6**	85	27,39	98	29,38	100	31,37	100	33,36	100	35,35	100	37,34	100	39,33	100	41,32
6	**7**	99	36,03	100	37,09	100	38,16	100	39,22	100	40,28	100	41,34	100	42,41	100	43,47
6	**8**	100	53,31	100	53,91	100	54,51	100	55,11	100	55,71	100	56,3	100	56,9	100	57,5
7	**5**	42	26,2	76	30,14	77	34,08	93	38,03	99	41,97	100	45,91	100	49,86	100	53,8
7	**6**	97	30,52	100	32,55	100	34,59	100	36,63	100	38,67	100	40,71	100	42,75	100	44,79
8	**5**	70	29,32	95	33,36	100	37,4	100	41,44	100	45,49	100	49,53	100	53,57	100	57,61

Dans le Tableau 5.4, on considère toujours la même mémoire mais avec une densité de défaut encore plus élevée ($D_d = 3\times10^{-4}$). Dans ce cas la réparation combinée commence à montrer des améliorations visibles. Le meilleur résultat pour cette technique est obtenu avec les valeurs $k = 4$, $r = 7$, $q = 6$ pour lesquelles le rendement est de 99% avec un coût en surface de 42,41%, tandis que le meilleur résultat de la réparation dynamique simple est obtenu avec les valeurs $k = 10$, $r = 6$, $q = 0$, pour lesquelles on enregistre un rendement de 98% avec un surcoût en surface de 48,53%.

Table 5.4 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-4}$ (~ 300 défauts)

k	*r*	*q* = 0		*q* = 1		*q* = 2		*q* = 3		*q* = 4		*q* = 5		*q* = 6		*q* = 7	
2	**9**	0	75,37	0	75,71	0	76,06	0	76,4	0	76,74	1	77,08	1	77,43	2	77,77
2	**10**	0	144,5	2	144,7	5	145	13	145,2	20	145,4	39	145,7	47	145,9	62	146,2
3	**10**	49	147,6	85	147,9	93	148,1	93	148,3	99	148,6	100	148,8	100	149,1	100	149,3
3	**11**	100	285,9	100	286	100	286,2	100	286,4	100	286,6	100	286,8	100	287	100	287,2
4	**7**	0	29,78	0	30,79	0	31,81	0	32,82	0	33,84	0	34,85	1	35,86	3	36,88
4	**8**	4	47,06	22	47,63	52	48,21	68	48,78	84	49,36	90	49,93	94	50,51	97	51,08
4	**9**	57	81,62	98	81,97	97	82,33	100	82,68	100	83,04	100	83,39	100	83,75	100	84,1
4	**10**	90	150,7	100	151	100	151,2	100	151,5	100	151,7	100	152	100	152,2	100	152,5
5	**7**	0	32,91	2	33,94	15	34,98	27	36,02	37	37,06	53	38,1	65	39,13	78	40,17
5	**8**	45	50,19	80	50,77	90	51,36	98	51,94	99	52,53	100	53,12	100	53,7	100	54,29
5	**9**	94	84,75	100	85,11	100	85,47	100	85,83	100	86,19	100	86,55	100	86,91	100	87,27
5	**10**	98	153,9	100	154,1	100	154,4	100	154,6	100	154,9	100	155,1	100	155,4	100	155,6

6	**7**	16	36,03	34	37,09	59	38,16	79	39,22	92	40,28	96	41,34	99	42,41	99	43,47
6	**8**	78	53,31	97	53,91	99	54,51	100	55,11	100	55,71	100	56,3	100	56,9	100	57,5
6	**9**	98	87,87	100	88,24	100	88,6	100	88,97	100	89,34	100	89,7	100	90,07	100	90,44
6	**10**	99	157	100	157,2	100	157,5	100	157,7	100	158	100	158,2	100	158,5	100	158,7
7	**7**	45	39,16	82	40,24	96	41,33	98	42,42	99	43,5	100	44,59	100	45,68	100	46,77
7	**8**	95	56,44	100	57,05	100	57,66	100	58,27	100	58,88	100	59,49	100	60,1	100	60,71
8	**7**	79	42,28	96	43,39	99	44,5	99	45,61	99	46,73	100	47,84	100	48,95	100	50,06
8	**8**	99	59,56	100	60,18	100	60,81	100	61,43	100	62,05	100	62,68	100	63,3	100	63,92
9	**6**	4	36,77	19	38,9	44	41,04	66	43,18	82	45,31	95	47,45	97	49,59	100	51,72
9	**7**	93	45,41	99	46,54	100	47,68	100	48,81	100	49,95	100	51,08	100	52,22	100	53,36
10	**6**	22	39,89	55	42,08	80	44,26	94	46,45	99	48,63	100	50,82	100	53	100	55,19
10	**7**	98	48,53	100	49,69	100	50,85	100	52,01	100	53,17	100	54,33	100	55,49	100	56,65

Le Tableau 5.5 considère une densité de défauts encore plus élevée ($D_d = 10^{-3}$). La réparation combinée montre ici une amélioration supplémentaire par rapport à la réparation dynamique simple, illustrant ainsi sa supériorité face à de plus grandes densités de défauts. Nous pouvons voir que pour les valeurs $k = 10$, $r = 8$, $q = 4$, la technique combinée atteint un rendement de 93% pour 68,4% de coût en surface, tandis que le meilleur résultat pour la réparation dynamique seule est enregistré pour les valeur $k = 9$, $r = 9$, $q = 0$ avec un rendement de 91% et 97,5% de coût en surface.

Table 5.5 Mémoire de 1Mbit, mot de 32 bit, $D_d = 10^{-3}$ (~ 1000 défauts)

k	***r***	***q* = 0**		***q* = 1**		***q* = 2**		***q* = 3**		***q* = 4**	
4	**9**	0	81,62	0	81,97	0	82,33	0	82,68	0	83,04
4	**10**	0	150,7	2	151	10	151,2	16	151,5	24	151,7
5	**10**	0	153,9	48	154,1	80	154,4	98	154,6	100	154,9
6	**8**	0	53,31	0	53,91	0	54,51	0	55,11	0	55,71
6	**9**	0	87,87	8	88,24	16	88,6	33	88,97	47	89,34
6	**10**	67	157	96	157,2	100	157,5	100	157,7	100	158
7	**8**	0	56,44	0	57,05	0	57,66	0	58,27	0	58,88
7	**9**	24	91	56	91,37	77	91,74	91	92,11	98	92,49
7	**10**	92	160,1	99	160,4	100	160,6	100	160,9	100	161,1
8	**8**	0	59,56	0	60,18	0	60,81	1	61,43	2	62,05
8	**9**	72	94,12	91	94,5	99	94,88	100	95,26	100	95,64
8	**10**	99	163,2	100	163,5	100	163,8	100	164	100	164,3
9	**8**	0	62,69	4	63,32	14	63,96	23	64,59	44	65,23
9	**9**	91	97,25	99	97,63	100	98,02	100	98,4	100	98,79
9	**10**	100	166,4	100	166,6	100	166,9	100	167,1	100	167,4
10	**8**	10	65,81	31	66,46	61	67,11	78	67,75	93	68,4

10	**9**	98	100,4	100	100,8	100	101,2	100	101,5	100	101,9
10	**10**	100	169,5	100	169,8	100	170	100	170,3	100	170,5
11	**8**	39	68,94	75	69,59	96	70,25	100	70,91	100	71,57
11	**9**	100	103,5	99	103,9	100	104,3	100	104,7	100	105,1
11	**10**	100	172,6	100	172,9	100	173,1	100	173,4	100	173,7
12	**6**	0	46,14	0	48,42	0	50,71	0	52,99	0	55,27
12	**7**	0	54,78	0	55,99	0	57,2	0	58,41	0	59,62
12	**8**	72	72,06	96	72,73	98	73,4	100	74,08	100	74,75
12	**9**	100	106,6	100	107	100	107,4	100	107,8	100	108,2
13	**7**	0	57,91	0	59,14	0	60,37	100	61,61	0	62,84
13	**8**	90	75,19	98	75,87	100	76,55	100	77,24	100	77,92
13	**9**	100	109,7	100	110,2	100	110,6	100	111	100	111,4
14	**7**	0	61,03	0	62,29	2	63,55	4	64,8	10	66,06
14	**8**	96	78,31	100	79,01	100	79,7	100	80,4	100	81,1
14	**9**	100	112,9	100	113,3	100	113,7	100	114,1	100	114,5
15	**7**	0	64,16	4	65,44	15	66,72	34	68	52	69,28
15	**8**	99	81,44	100	82,14	100	82,85	100	83,56	100	84,27
15	**9**	100	116	100	116,4	100	116,8	100	117,3	100	117,7
16	**7**	8	67,28	32	68,59	55	69,89	72	71,2	86	72,51
16	**8**	99	84,56	100	85,28	100	86	100	86,72	100	87,44

Le Tableau 5.6 considère une densité de défaut encore plus élevée ($D_d = 3\times10^{-3}$). Dans ce cas la réparation combinée continue à démontrer supériorité en enregistrant un rendement de 100% pour un surcoût en surface estimée a 116,2% et ce pour les valeurs $k = 14$, $r = 9$, $q = 8$, tandis que le meilleur résultat atteint par la réparation dynamique est un rendement de 100% pour 182% de surcoût en surface, obtenus pour $k = 14$, $r = 10$, $q = 0$.

Tableau 5.6 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-3}$ (~ 3000 défauts)

k	***r***	***q* = 0**		***q* = 4**		***q* = 5**		***q* = 6**		***q* = 7**		***q* = 8**		***q* = 9**		***q* = 10**	
7	**10**	0	160,1	0	161,1	0	161,4	0	161,6	1	161,9	1	162,1	2	162,4	2	162,7
7	**11**	28	298,4	99	299,1	100	299,3	100	299,5	100	299,7	100	299,9	100	300,1	100	300,3
8	**10**	0	163,2	17	164,3	30	164,5	38	164,8	46	165	60	165,3	71	165,6	83	165,8
8	**11**	76	301,5	100	302,3	100	302,5	100	302,7	100	302,9	100	303	100	303,2	100	303,4
9	**10**	5	166,4	88	167,4	94	167,7	97	167,9	98	168,2	100	168,4	100	168,7	100	169
9	**11**	95	304,6	100	305,4	100	305,6	100	305,8	100	306	100	306,2	100	306,4	100	306,6
10	**10**	34	169,5	100	170,5	100	170,8	100	171,1	100	171,3	100	171,6	100	171,9	100	172,1
10	**11**	99	307,7	100	308,5	100	308,7	100	308,9	100	309,1	100	309,3	100	309,5	100	309,7
11	**10**	77	172,6	100	173,7	100	173,9	100	174,2	100	174,5	100	174,7	100	175	100	175,3
11	**11**	100	310,9	100	311,7	100	311,9	100	312,1	100	312,3	100	312,5	100	312,7	100	312,9
12	**9**	0	106,6	3	108,2	5	108,6	7	109	11	109,4	16	109,8	25	110,3	42	110,7

12	**10**	88	175,7	100	176,8	100	177,1	100	177,4	100	177,6	100	177,9	100	178,2	100	178,4
13	**9**	0	109,7	36	111,4	50	111,8	66	112,2	77	112,6	84	113	89	113,4	94	113,8
13	**10**	96	178,9	100	180	100	180,2	100	180,5	100	180,8	100	181	100	181,3	100	181,6
14	**9**	0	112,9	84	114,5	90	114,9	96	115,4	97	115,8	100	116,2	100	116,6	100	117
14	**10**	100	182	100	183,1	100	183,4	100	183,6	100	183,9	100	184,2	100	184,5	100	184,7

Les Tableaux 5.7, 5.8, 5.9, et 5.10 présentent les résultats du rendement théorique (calculés à partir de l'équation (5.10)) pour une instance mémoire atteinte par les mêmes densités de défauts que celles utilisées pour les Tableaux 5.1, 5.2, 5.3, et 5.4.

En comparant les rendements théoriques donnés par les Tableaux 5.7, 5.8, 5.9, et 5.10 avec les rendements simulés présentes dans les Tableaux 5.1, 5.2, 5.3, et 5.4, nous trouvons une très bonne corrélation correspondant à une différence de quelques pour cents dans la plupart des cas. Dans d'autres cas la différence est plus importante, mais reste raisonnable. Cela peut être expliquée en considérant les facteurs suivants :

- La variation du degré de précision pendant lors du calcul des grandes valeurs dues aux termes en factoriels et aux petites valeurs dues aux puissances des probabilités.
- Les éclats de fautes ne sont pas pris en compte dans les formules analytiques. En effet, seules les fautes singulières sont considérées, pour des raisons de simplification.
- L'imperfection dans la génération des nombres aléatoires par la fonction rand() du langage C.
- L'erreur statistique induite par la réalisation d'un nombre limité de simulations.

Cette corrélation très raisonnable entre les résultats théoriques et expérimentaux, valide notre méthodologie de simulation qui a été intégrée dans l'outil, et nous rend confiant sur la précision des résultats obtenus.

Tableau 5.7 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-5}$ (~ 10 défauts)

k	*r*	*q* = 0	*q* = 1	*q* = 2	*q* = 3	*q* = 4	*q* = 5	*q* = 6	*q* = 7
2	**4**	60.18	90.26	98.24	99.74	99.96	99.99	99.99	100
2	**5**	85.18	98.81	99.93	99.99	99.99	100	100	100
2	**6**	95.56	99.99	99.99	100	100	100	100	100
2	**7**	98.8	99.99	100	100	100	100	100	100
3	**3**	66.04	92.74	98.81	99.83	99.98	99.99	99.99	100
3	**4**	91.76	99.63	99.98	99.99	100	100	100	100
3	**5**	98.6	99.99	100	100	100	100	100	100
3	**6**	99.79	99.99	100	100	100	100	100	100

3	**7**	99.97	100	100	100	100	100	100	100

Tableau 5.8 Mémoire de 1Mbit, mot de 32 bit, $D_d = 2\times10^{-5}$ (~ 20 défauts)

k	*r*	*q* = 0	*q* = 1	*q* = 2	*q* = 3	*q* = 4	*q* = 5	*q* = 6	*q* = 7
2	**5**	36.21	72.42	91.08	97.69	99.49	99.90	99.98	99.99
2	**6**	72.56	95.77	99.54	99.96	99.99	99.99	100	100
2	**7**	91.33	99.61	99.99	99.99	100	100	100	100
3	**4**	43.62	78.88	94.03	98.62	99.73	99.95	99.99	99.99
3	**5**	84.21	98.64	99.91	99.99	99.99	100	100	100
3	**6**	97.23	99.96	99.99	100	100	100	100	100
3	**7**	99.59	99.99	100	100	100	100	100	100
4	**3**	27.37	60.12	82.15	93.12	97.64	99.25	99.78	99.94
4	**4**	79.92	97.7	99.81	99.98	99.99	100	100	100
4	**5**	97.58	99.96	99.99	100	100	100	100	100
4	**6**	99.79	100	100	100	100	100	100	100
4	**7**	99.98	100	100	100	100	100	100	100

Tableau 5.9 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-4}$ (~ 100 défauts)

k	*r*	*q* = 0	*q* = 1	*q* = 2	*q* = 3	*q* = 4	*q* = 5	*q* = 6	*q* = 7
3	**6**	0.16	1.16	4.27	10.79	21.2	34.68	49.46	63.54
4	**6**	12.34	37.75	64.30	83.09	93.21	97.63	99.27	99.79
4	**7**	77.16	97.14	99.75	99.84	99.99	100	100	100
5	**6**	54.4	87.36	97.50	99.61	99.95	99.99	99.99	100
5	**7**	96.12	99.92	99.99	100	100	100	100	100
6	**5**	9.38	30.78	55.95	76.28	88.96	95.47	98.33	99.44
6	**6**	85.3	98.84	99.93	99.99	99.99	100	100	100
7	**5**	35.49	71.66	90.68	98.45	99.54	99.89	99.98	99.99
7	**6**	96.3	99.92	99.99	100	100	100	100	100
7	**7**	99.39	100	100	100	100	100	100	100
8	**4**	0.22	1.38	4.49	10.38	19.23	30.43	42.82	55.13
8	**5**	65.48	93.02	98.99	99.88	99.98	99.99	99.99	100

Tableau 5.10 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-4}$ (~ 300 défauts)

k	*r*	*q* = 0	*q* = 1	*q* = 2	*q* = 3	*q* = 4	*q* = 5	*q* = 6	*q* = 7
6	**7**	7.58	26.95	51.87	73.40	87.47	94.88	98.15	99.4
7	**6**	0	0	0	0	0	0.82	1.93	3.95
7	**7**	41.24	77.64	93.83	93.83	98.67	99.76	99.99	99.99
8	**6**	0.07	0.58	2.34	6.47	13.82	24.45	37.44	51.27
8	**7**	75.5	96.69	99.69	99.97	99.99	99.99	100	100

9	**7**	92.07	99.67	99.99	99.99	100	100	100	100
10	**6**	14.32	41.74	68.38	85.92	94.7	98.27	99.5	99.87
10	**7**	97.73	99.97	99.99	100	100	100	100	100

5.6.2 Evaluation de la Technique de Réparation Basée sur les Polarités d'Erreurs

Les résultats expérimentaux sont montrés dans les Tableaux 5.11 à 5.15. Nous avons considéré les densités de défauts suivantes : $D_d = 1\times10^{-4}$, $D_d = 3\times10^{-4}$, $D_d = 1\times10^{-3}$, $D_d = 3\times10^{-3}$, et $D_d = 1\times10^{-2}$. Nous indiquons au-dessus de chaque tableau la taille de la mémoire, la longueur du mot, la densité de défaut globale D_d, et le nombre moyen de fautes dans la mémoire régulière[6] pour la densité de défaut globale considérée. Les deux premières colonnes de ces tableaux contiennent les valeurs des paramètres de la réparation dynamique au niveau bit : k (#bits redondants), et r (#bits d'adresse utilisés pour diviser une colonne redondante en R unités réparables, $R = 2^r$). Dans le reste des colonnes, on donne l'efficacité et le coût en surface en termes de pourcentage par rapport à la mémoire sans technique de réparation. Ces pourcentages sont donnés pour différentes valeurs du nombre q de blocs redondants[7]. En plus de l'introduction des différentes polarités d'erreurs, nous avons considéré les éclats de fautes comme dans le cas de la réparation combinée dynamique/bloc.

Au vu des résultats obtenus pour cette technique, nous pouvons dire qu'elle est moins avantageuse que celle qui n'est pas basée sur les polarités d'erreurs. En effet, la présente technique donne lieu à des améliorations significatives du rendement en comparaison avec la technique précédente, mais en même temps elle induit un accroissement considérable du surcoût en surface. Cet accroissement de la surface est dû à la haute complexité des fonctions de reconfigurations. Cet accroissement du coût abaisse l'efficacité générale de la réparation.

Les expériences du Tableau 5.11 ont été réalisées pour $D_d = 1\times10^{-4}$. Le rendement sans réparation est de 0%. Dans ce cas, le meilleur résultat est obtenu avec $k = 4$, $r = 6$, $q = 3$, et pour lequel le rendement est de 96% et le surcoût en surface est de 35,83%. Pour les mêmes valeurs de k, r, et q, la technique qui n'utilise pas les polarités d'erreurs, atteint un rendement de seulement 83% mais avec un surcoût moindre en surface (26,82%). Pour $k = 4$, $r = 6$, $q = 4$, cette dernière technique atteint un rendement de 98% avec seulement 28,7% de surcoût en surface.

[6] Le nombre moyen de fautes dans les parties redondantes de la mémoire n'est pas indiqué.
[7] La colonne de valeur $q = 0$ représente l'efficacité de la simple réparation dynamique au niveau bit.

Tableau 1.11 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-4}$ (~ 100 défauts)

k	*r*	*q* = 0		*q* = 1		*q* = 2		*q* = 3		*q* = 4	
1	**9**	0	141,4	0	141,8	0	142,3	1	142,8	1	143,3
1	**10**	0	279,6	0	280	1	280,3	4	280,7	7	281,1
2	**7**	0	40,81	0	41,91	3	43,01	9	44,11	18	45,21
2	**8**	8	75,37	27	76,06	51	76,74	69	77,43	84	78,11
2	**9**	24	144,5	62	145	87	145,4	95	145,9	98	146,4
2	**10**	50	282,7	85	283,1	97	283,5	100	283,9	100	284,2
3	**5**	0	18,02	0	21,7	0	25,39	0	29,08	0	32,77
3	**6**	1	26,66	7	28,63	18	30,61	28	32,59	46	34,57
3	**7**	24	43,94	55	45,06	83	46,18	93	47,31	98	48,43
3	**8**	63	78,5	96	79,19	100	79,89	100	80,59	100	81,28
3	**9**	89	147,6	99	148,1	100	148,6	100	149,1	100	149,5
3	**10**	95	285,9	100	286,2	100	286,6	100	287	100	287,4
4	**4**	0	16,82	0	24,12	0	31,42	0	38,72	0	46,03
4	**5**	0	21,14	3	24,93	6	28,71	15	32,5	32	36,28
4	**6**	24	29,78	55	31,81	81	33,84	96	35,86	100	37,89
4	**7**	80	47,06	96	48,21	100	49,36	100	50,51	100	51,66
4	**8**	96	81,62	100	82,33	100	83,04	100	83,75	100	84,46
4	**9**	99	150,7	100	151,2	100	151,7	100	152,2	100	152,7
4	**10**	100	289	100	289,4	100	289,7	100	290,1	100	290,5
5	**4**	0	19,95	0	27,44	0	34,94	0	42,43	1	49,93
5	**5**	3	24,27	25	28,15	49	32,03	68	35,91	82	39,8
5	**6**	68	32,91	96	34,98	99	37,06	100	39,13	100	41,21
5	**7**	97	50,19	100	51,36	100	52,53	100	53,7	100	54,88
5	**8**	100	84,75	100	85,47	100	86,19	100	86,91	100	87,63
6	**3**	0	20,91	0	36,02	0	51,14	0	66,25	1	81,37
6	**4**	0	23,07	2	30,76	5	38,45	12	46,15	22	53,84
6	**5**	46	27,39	75	31,37	94	35,35	99	39,33	100	43,31
6	**6**	94	36,03	100	38,16	100	40,28	100	42,41	100	44,53
6	**7**	100	53,31	100	54,51	100	55,71	100	56,9	100	58,1
7	**3**	0	24,04	0	39,54	0	55,04	0	70,55	1	86,05
7	**4**	5	26,2	17	34,08	30	41,97	47	49,86	66	57,74
7	**5**	71	30,52	94	34,59	100	38,67	100	42,75	100	46,83
7	**6**	99	39,16	100	41,33	100	43,5	100	45,68	100	47,85
7	**7**	100	56,44	100	57,66	100	58,88	100	60,1	100	61,32
8	**3**	0	27,16	0	43,06	0	58,95	0	74,85	1	90,74
8	**4**	19	29,32	49	37,4	69	45,49	84	53,57	94	61,65
8	**5**	90	33,64	99	37,82	100	41,99	100	46,17	100	50,35
8	**6**	100	42,28	100	44,5	100	46,73	100	48,95	100	51,17

Le Tableau 1.12 concerne la même mémoire avec une densité de défaut plus élevée (D_d = 3×10^{-4}). Dans ce cas, le meilleur résultat correspond aux valeurs k = 6, r = 7, q = 3 pour lesquelles le rendement est de 97% avec 56,9% de surcoût en surface. C'est néanmoins un moins bon résultat si on compare avec la technique n'utilisant pas les polarités d'erreurs (99% de rendement, 42,42% de coût en surface, pour k = 6, r = 7, q = 6).

Tableau 1.12 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-4}$ (~ 300 défauts)

k	r	q = 0		q = 1		q = 2		q = 3		q = 4	
2	**8**	0	75,37	0	76,06	0	76,74	2	77,43	2	78,11
2	**9**	0	144,5	0	145	4	145,4	12	145,9	16	146,4
2	**10**	2	282,7	4	283,1	6	283,5	17	283,9	26	284,2
3	**7**	0	43,94	0	45,06	0	46,18	2	47,31	2	48,43
3	**8**	0	78,5	1	79,19	3	79,89	5	80,59	6	81,28
3	**9**	7	147,6	34	148,1	59	148,6	79	149,1	94	149,5
3	**10**	49	285,9	90	286,2	100	286,6	100	287	100	287,4
4	**6**	0	29,78	0	31,81	0	33,84	0	35,86	0	37,89
4	**7**	0	47,06	0	48,21	0	49,36	1	50,51	4	51,66
4	**8**	11	81,62	34	82,33	63	83,04	82	83,75	94	84,46
4	**9**	72	150,7	94	151,2	100	151,7	100	152,2	100	152,7
4	**10**	94	289	100	289,4	100	289,7	100	290,1	100	290,5
5	**5**	0	24,27	0	28,15	0	32,03	0	35,91	0	39,8
5	**6**	0	32,91	0	34,98	0	37,06	0	39,13	1	41,21
5	**7**	7	50,19	10	51,36	27	52,53	44	53,7	63	54,88
5	**8**	59	84,75	92	85,47	98	86,19	99	86,91	100	87,63
5	**9**	97	153,9	100	154,4	100	154,9	100	155,4	100	155,8
5	**10**	100	292,1	100	292,5	100	292,9	100	293,3	100	293,6
6	**6**	0	36,03	0	38,16	1	40,28	3	42,41	6	44,53
6	**7**	32	53,31	71	54,51	90	55,71	97	56,9	99	58,1
6	**8**	95	87,87	99	88,6	100	89,34	100	90,07	100	90,81
6	**9**	99	157	100	157,5	100	158	100	158,5	100	159
6	**10**	100	295,2	100	295,6	100	296	100	296,4	100	296,8
7	**5**	0	30,52	0	34,59	0	38,67	0	42,75	0	46,83
7	**6**	1	39,16	6	41,33	17	43,5	26	45,68	46	47,85
7	**7**	75	56,44	98	57,66	100	58,88	100	60,1	100	61,32
7	**8**	99	91	100	91,74	100	92,49	100	93,23	100	93,98
7	**9**	100	160,1	100	160,6	100	161,1	100	161,6	100	162,1
8	**5**	0	33,64	0	37,82	0	41,99	0	46,17	0	50,35

8	**6**	17	42,28	41	44,5	67	46,73	86	48,95	95	51,17
8	**7**	92	59,56	100	60,81	100	62,05	100	63,3	100	64,55
8	**8**	100	94,12	100	94,88	100	95,64	100	96,39	100	97,15
9	**5**	0	36,77	0	41,04	0	45,31	0	49,59	1	53,86
9	**6**	57	45,41	80	47,68	94	49,95	99	52,22	100	54,49
9	**7**	96	62,69	100	63,96	100	65,23	100	66,5	100	67,77
9	**8**	100	97,25	100	98,02	100	98,79	100	99,56	100	100,3
10	**5**	0	39,89	2	44,26	6	48,63	15	53	26	57,38
10	**6**	81	48,53	96	50,85	100	53,17	100	55,49	95	57,81
10	**7**	100	65,81	100	67,11	100	68,4	100	69,7	100	70,99

Le Tableau 5.13 considère la valeur $D_d = 1\times10^{-3}$. Dans ce tableau le meilleur résultat est obtenu pour $k = 8$, $r = 8$, $q = 4$ pour atteindre 88% de rendement avec 97,5% de coût en surface. Ce résultat est moins bon que celui obtenu pour la technique qui n'utilise pas les polarités d'erreurs (93% de rendement, et 68,4% de coût en surface, pour $k = 10$, $r = 8$, $q = 4$).

Tableau 5.13 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-3}$ (~ 1000 défauts)

k	***r***	***q* = 0**		***q* = 1**		***q* = 2**		***q* = 3**		***q* = 4**	
3	**11**	0	562,3	2	562,7	7	563	19	563,3	38	563,6
4	**10**	1	289	8	289,4	19	289,7	35	290,1	58	290,5
5	**9**	0	153,9	2	154,4	2	154,9	5	155,4	7	155,8
5	**10**	44	292,1	77	292,5	89	292,9	99	293,3	100	293,6
6	**9**	18	157	38	157,5	69	158	86	158,5	95	159
6	**10**	99	295,2	100	295,6	100	296	100	296,4	100	296,8
7	**8**	0	91	0	91,74	0	92,49	5	93,23	13	93,98
7	**9**	63	160,1	93	160,6	99	161,1	100	161,6	100	162,1
7	**10**	97	298,4	100	298,7	100	299,1	100	299,5	100	299,9
8	**8**	6	94,12	18	94,88	47	95,64	72	96,39	88	97,15
8	**9**	90	163,2	99	163,8	100	164,3	100	164,8	100	165,3

Le Tableau 5.14 considère une valeur de densité de défaut plus élevée ($D_d = 3\times10^{-3}$). Les meilleurs paramètres pour ce cas de figure sont donnés par $k = 9$, $r = 10$, $q = 3$, pour un rendement de 98% et un coût de 305,8%

Tableau 5.14 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-3}$ (~ 3000 défauts)

k	*r*	*q* = 0		*q* = 1		*q* = 2		*q* = 3		*q* = 4	
3	**11**	0	562,3	2	562,7	7	563	19	563,3	38	563,6
4	**10**	1	289	8	289,4	19	289,7	35	290,1	58	290,5
5	**9**	0	153,9	2	154,4	2	154,9	5	155,4	7	155,8
5	**10**	44	292,1	77	292,5	89	292,9	99	293,3	100	293,6
6	**9**	18	157	38	157,5	69	158	86	158,5	95	159
6	**10**	99	295,2	100	295,6	100	296	100	296,4	100	296,8
7	**8**	0	91	0	91,74	0	92,49	5	93,23	13	93,98
7	**9**	63	160,1	93	160,6	99	161,1	100	161,6	100	162,1
7	**10**	97	298,4	100	298,7	100	299,1	100	299,5	100	299,9
8	**8**	6	94,12	18	94,88	47	95,64	72	96,39	88	97,15
8	**9**	90	163,2	99	163,8	100	164,3	100	164,8	100	165,3

Le Tableau 3.15 montre les résultats pour $D_d = 1\times10^{-2}$. Les meilleurs paramètres pour ce cas sont $k = 10$, $r = 11$, $q = 4$, pour un rendement de 97% et un coût de 328%. Au vu de ces résultats, il est clair que la réparation basée sur les polarités d'erreurs n'est pas souhaitable pour les très hautes densités de défauts.

Tableau 5.15 Mémoire de 1Mbit, mot de 32 bit, $D_d = 2\times10^{-2}$ (~ 10000 défauts)

k	*r*	*q* = 0		*q* = 1		*q* = 2		*q* = 3		*q* = 4	
11	**11**	5	587,3	15	587,7	25	588	52	588,7	71	588,7
12	**11**	34	590,5	66	590,8	87	591,1	98	591,8	99	591,8
13	**11**	74	593,6	96	593,9	99	594,3	100	594,9	100	594,9
14	**10**	0	320,2	0	320,6	0	321,1	0	321,9	0	321,9
14	**11**	97	596,7	100	597,1	100	597,4	100	598,1	100	598,1
15	**10**	0	323,4	1	323,8	8	324,2	17	325	37	325
15	**11**	99	599,8	100	600,2	100	600,5	100	601,2	100	601,2
16	**10**	18	326,5	50	326,9	75	327,3	91	328,1	97	328,1
16	**11**	99	603	100	603,3	100	603,6	100	604,3	100	604,3

5.6.3 Evaluation de la Réparation Combinée Mot/ECC

Les résultats expérimentaux pour cette technique sont montrés dans les Tableaux 5.16 à 5.21. Nous considérons les densités de défauts suivantes : $D_d = 1\times10^{-4}$, $D_d = 3\times10^{-4}$, $D_d = 1\times10^{-3}$, $D_d = 3\times10^{-3}$, $D_d = 1\times10^{-2}$, et $D_d = 1\times10^{-2}$. Nous indiquons au-dessus de chaque

tableau la taille de la mémoire, la densité de défaut globale D_d, et le nombre moyen de fautes dans la mémoire régulière[8] pour la densité de défaut globale considérée. La première colonne des tableaux contient le nombre de mots CAM et le coût en surface (surface du ECC et surface des mots CAM). La seconde colonne donne le rendement. Dans tous les cas nous avons utilisé un seul flag par mot de la CAM. La raison est qu'on a constaté que sans flag le rendement était quasiment nul, et que d'ajouter plus d'un seul flag n'augmentait pas le rendement.

Nous pouvons voir dans ces tableaux que l'emploi de la technique ECC seule, donne toujours lieu à des rendements 0. Mais en ajoutant une CAM de petite taille, on obtient des rendements approchant les 100%.

Nous constatons également, que pour l'ensemble des cas considérés, la réparation combinée ECC et mot est la technique de réparation la plus efficace si on la compare avec les techniques discutées précédemment. Par exemple pour $D_d = 10^{-4}$, la présente technique requiert un coût en surface de 19,1%, tandis que la meilleure technique parmi celles discutées auparavant, enregistre un coût de 28,7%. Cependant, pour des plus petites densités de défauts, les techniques précédentes enregistrent un coût plus petit. En effet, pour $D_d = 10^{-5}$, et $D_d = 3\times10^{-5}$, les techniques précédentes requièrent respectivement un coût de 14,2% et 18,4%, contre un coût de 19% pour la combinaison du ECC et de la réparation mot. Cette dernière technique convient donc mieux aux densités de défauts supérieures à 10^{-4}. Cela reste valable pour des mots de 32 bits, car pour des mots plus petits (e.g. 16 bits, 8 bits, 4 bits), les techniques précédentes deviennent intéressantes même pour des densités de défauts plus élevées que 10^{-4}.

Le tableau 5.20 montre que la technique combinée ECC/mot est capable de réparer des mémoires affectées par une densité de défaut allant jusqu'à 10^{-2} pour un coût supplémentaire raisonnable (98% de rendement pour 37% de coût). Ainsi, cette technique permettra de réaliser des mémoires en nanotechnologie, qui pourront être affectées par de tels niveaux en densités de défauts.

Pour des densités de défauts plus élevées telles que 3×10^{-2} (tableau 5.21), le coût de réparation augmente considérablement (94%). La raison en est que le nombre moyen de cellules fautives par mot devient $32\times3\times10^{-2} = 0{,}96$. Ce nombre moyen approchant une faute par mot, représente la situation la plus répandue dans la mémoire (Figure 5.1), et représente en même temps le nombre limite pour laquelle la technique ECC de Haming peut corriger.

[8] Le nombre moyen de fautes dans les parties redondantes de la mémoire n'est pas indiqué.

Cependant, toute la moitié droite (nombre de fautes par mot > 1) de la courbe dans la Figure 5.1 doit être réparée à l'aide d'un nombre élevé de mots CAM (25% des mots mémoire et 75% de la surface mémoire). Pour réduire ce coût nous pouvons utiliser un double code correcteur d'erreur, et corriger ainsi le double de 0,96 c'est-à-dire 1,92 fautes par mot mémoire. Le problème est qu'une telle logique requiert un encodeur/décodeur très complexe est très lent à la fois. Une meilleure solution est de diviser les mots mémoire en deux parties de 16 bits chacune, et d'appliquer le code de Haming pour chacune d'elles. Dans ce cas, le nombre moyen de fautes par mot mémoire est de 0,48. La plupart des fautes sont ainsi corrigées par ECC, et une petite partie par la CAM. Le coût en surface est alors principalement absorbé par la technique ECC. Les résultats de simulation par notre outil, pour cette technique, sont donnés par le tableau 5.22. Une réduction significative du coût est obtenue, puisqu'on atteint un rendement de 100% pour 61,5% de coût, au lieu de 94% de coût dans le tableau 5.21. Cette solution peut très bien être appliquée aux mots mémoire plus grands que 32 bits, car ils contiendront un nombre de fautes plus élevé même pour des densités de défauts inférieures à 3×10^{-2}.

Tableau 5.16 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-4}$ (~ 100 défauts)

#Mots redondants (%)/ coût en surface (%)	Rendement (%)
0/19	1
0,01/19,03	19
0,03/19,09	89
0,05/19,15	100

Tableau 5.16 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-4}$ (~ 300 défauts)

#Mots redondants (%)/ coût en surface (%)	Rendement (%)
0/19	0
0,01/19,03	0
0,05/19,15	30
0,07/19,21	80
0,1/19,3	99
0,5/20,5	100

Tableau 5.17 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-3}$ (~ 1000 défauts)

#Mots redondants (%)/ coût en surface (%)	Rendement (%)
0/19	0
0,1/19,3	0

0,2/19,6	10
0,3/19,9	99
0,5/20,5	100

Tableau 5.18 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-3}$ (~ 3000 défauts)

#Mots redondants (%)/ coût en surface (%)	Rendement (%)
0,1/19,3	0
0,9/21,7	27
1/22	98
5/34	100

Tableau 5.19 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-2}$ (~ 10000 défauts)

#Mots redondants (%)/ coût en surface (%)	Rendement (%)
1/22	0
5/34	5
6/37	100

Tableau 5.20 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-2}$ (~ 30000 défauts)

0/19	0	0
5/34	0	0
10/49	0	0
15/54	0	0
20/79	0	0
25/94	100	100
30/119	100	100
50/220	100	100

Tableau 5.21 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-2}$ (~ 30000 défauts), 1 ECC par 16 bits

#Mots redondants (%)/ coût en surface (%)	Rendement(%) (#flags = 1)
0/31,5	0
5/46,5	0
10/61,5	100
15/> 61,5	100
20/> 61,5	100

5.6.4 Evaluation Basée sur le Modèle du Clustering

Le calcul des densités de défauts qui affectent les instances mémoire d'un même wafer, selon le modèle de clustering, dépend d'un certain nombre de paramètres variés. Ces paramètres peuvent être obtenus uniquement si on dispose d'un grand nombre de données statistiques sur la distribution des densités de défauts. De telles statistiques existent chez les

technologues mais ne sont pas publiquement divulguées. Par conséquent, il ne nous est pas possible d'avoir des paramètres extraits de données réalistes. De plus, même si nous disposions de données réalistes sur les technologies actuelles, celles-ci ne seraient pas pertinentes pour notre étude qui s'adresse aux nano-technologies.

Nous avons quand même sélectionné des paramètres qui donnent des résultats raisonnables (les mêmes proposées dans la section 5.5.6, avec en plus $e = 0{,}3$). Nous avons considéré dans ces simulations un wafer de 256 mémoires (16×16). A cause du problème de temps en CPU nécessaire à la réalisation d'un grand nombre de simulations purement statistiques sur ce wafer, nous avons adopté l'approche mixte analytique/ statistique, présentée dans la section 5.5.7. Cependant, nous n'avons pu réaliser ces simulations que pour l'approche de réparation combinant la reconfiguration dynamique au niveau bit de donnée avec la réparation bloc. En effet, à cause des limitations de calcul des ordinateurs, il nous a pas été possible d'utiliser les formules analytiques liées à la technique combinée ECC/réparation mot. Un autre problème est qu'on a utilisé le modèle de clustering uniquement au niveau wafer mais pas au niveau de la puce (dans notre cas une puce représente une instance mémoire). Ce type de simulation introduit différentes densités de défauts pour différentes surfaces de la mémoire, et pour lesquelles nous ne disposons pas de formules analytiques.

Les résultats de simulations sont présentés dans les Tableaux 5.22 à 5.25 pour la technique combinée dynamique/bloc. En comparaison avec les résultats de la même technique sans les effets de clustering (Tableaux 5.1 à 5.5), nous observons que les paramètres de réparation (k, r, et q) qui enregistraient de faibles rendements, enregistrent avec le modèle de clustering une augmentation du rendement. Ce phénomène était prévisible à cause de l'apparition de plusieurs instances mémoires affectées par des densités de défauts inférieures à la densité de défaut moyenne (voir la Figure 5.6 qui montre la distribution des densités de défauts dans un wafer atteint par une densité de défaut moyenne de 10^{-5}). La situation inverse est plutôt désavantageuse. En effet, pour les paramètres de réparation qui enregistraient des rendements élevés sans le modèle de clustering, on remarque une baisse du rendement à cause de l'apparition de plusieurs instances mémoires avec des densités de défauts supérieures à la densité de défaut moyenne du wafer. Cet effet n'est pas souhaitable puisque nous visons des rendements élevés. Cependant il est possible d'obtenir de bons rendements en acceptant d'attribuer un plus grand coût en surface.

Tableau 5.22 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-5}$ (~ 10 défauts), modèle de clustering

k	r	$q = 0$	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$	$q = 7$
2	**4**	60.66	82.03	91.14	95.48	97.65	98.77	99.36	99.66
2	**5**	81.10	95.37	98.76	99.66	99.90	99.97	99.99	99.99
2	**6**	93.18	99.34	99.93	99.99	99.99	99.99	100	100
2	**7**	97.98	99.94	99.99	99.99	100	100	100	100
3	**3**	66.32	82.88	90.76	94.71	96.86	98.09	98.82	99.26
3	**4**	86.47	96.92	99.18	99.77	99.93	99.98	99.99	99.99
3	**5**	96.85	99.80	99.98	99.99	99.99	100	100	100
3	**6**	99.47	99.99	99.99	100	100	100	100	100
3	**7**	99.92	99.99	100	100	100	100	100	100

Tableau 5.23 Mémoire de 1Mbit, mot de 32 bit, $D_d = 2\times10^{-5}$ (~ 20 défauts), modèle de clustering

k	r	$q = 0$	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$	$q = 7$
2	**5**	42.37	67.82	81.74	89.56	94.05	96.63	98.11	98.94
2	**6**	70.12	90.87	97.14	99.10	99.72	99.91	99.97	99.99
2	**7**	88.87	98.66	99.83	99.97	99.99	99.99	100	100
3	**4**	48.30	71.35	83.17	89.82	93.77	96.16	97.62	98.52
3	**5**	79.48	94.69	98.49	99.55	99.86	99.95	99.98	99.99
3	**6**	95.30	99.67	99.97	99.99	99.99	100	100	100
3	**7**	99.24	99.99	99.99	100	100	100	100	100
4	**3**	38.54	59.16	71.10	78.72	83.94	87.69	90.45	92.53
4	**4**	74.22	90.85	96.37	98.48	99.34	99.70	99.86	99.93
4	**5**	94.99	99.54	99.94	99.99	99.99	99.99	100	100
4	**6**	99.47	99.99	99.99	100	100	100	100	100
4	**7**	99.95	100	100	100	100	100	100	100

Tableau 5.24 Mémoire de 1Mbit, mot de 32 bit, $D_d = 1\times10^{-4}$ (~ 100 défauts), modèle de clustering

k	r	$q = 0$	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$	$q = 7$
3	**6**	20.62	33.33	41.69	47.81	52.57	56.37	59.45	61.97
4	**6**	41.35	55.05	62.20	66.71	69.93	72.47	74.63	76.52
4	**7**	68.45	80.20	86.14	90.13	92.97	95	96.43	97.44
5	**6**	58.68	69.87	75.30	78.98	81.83	84.20	86.24	88.06
5	**7**	84.67	93.57	97.04	98.60	99.34	99.70	99.87	99.95
6	**5**	42.53	54.26	60.45	64.37	67.13	69.27	71.04	72.58
6	**6**	71.36	80.76	85.72	89.19	91.79	93.74	95.18	96.25
7	**5**	53.58	63.82	68.72	71.90	74.35	76.36	78.10	79.59
7	**6**	81.32	89.71	93.79	96.11	97.48	98.34	98.92	99.31
7	**7**	98.39	99.88	99.99	99.99	100	100	100	100
8	**4**	31.21	41.15	47.02	51.22	54.45	57.00	59.05	60.73
8	**5**	62.38	71.03	75.36	78.36	80.69	82.58	84.20	85.64

Tableau 5.25 Mémoire de 1Mbit, mot de 32 bit, $D_d = 3\times10^{-4}$ (~ 300 défauts), modèle de clustering

k	r	$q = 0$	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q = 5$	$q = 6$	$q = 7$
6	**7**	35.80	46.86	53.69	58.95	63.32	67.04	70.24	72.99
7	**6**	17.80	26.08	31.15	34.79	37.62	39.96	41.98	43.80
7	**7**	49.75	62.19	69.71	74.92	78.67	81.43	83.54	86.22
8	**6**	26.21	34.87	39.85	43.45	46.41	48.98	51.29	53.40
8	**7**	63.61	75.93	82.01	85.59	87.99	89.75	91.12	92.25
9	**7**	75.45	85.48	89.69	92.16	93.90	95.23	96.26	97.04
10	**6**	41.98	51.27	57.07	61.52	65.16	68.24	70.88	73.15
10	**7**	84.23	91.78	94.82	96.54	97.57	98.17	98.55	98.81

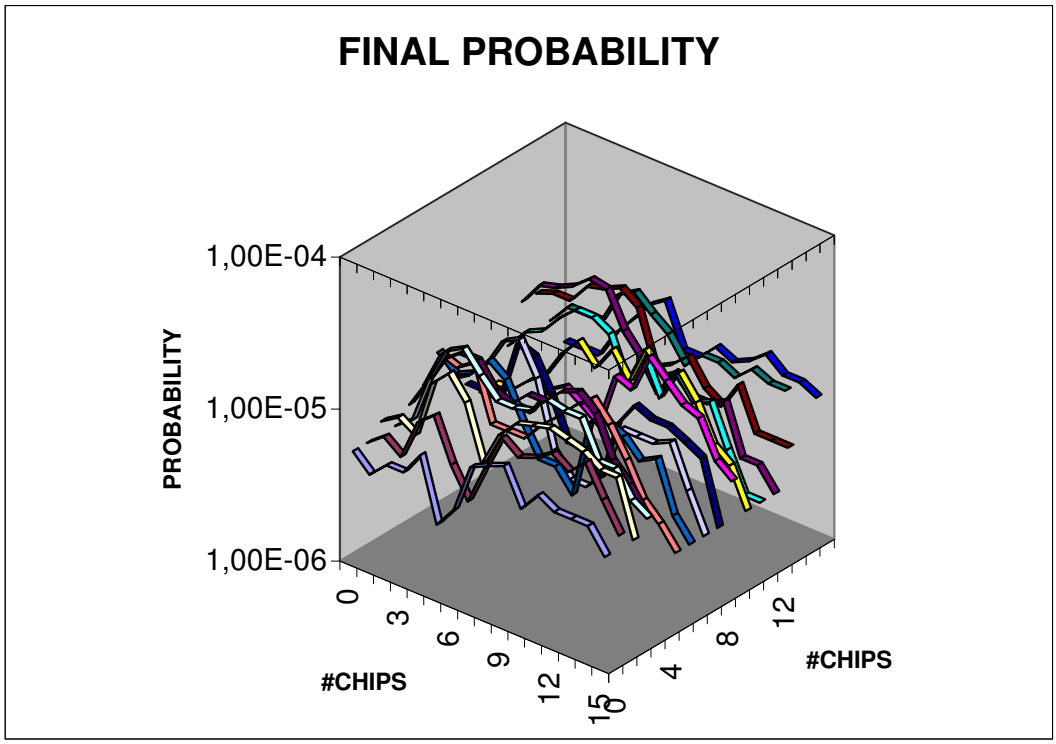


Figure 5.6. Distribution des densités de défauts dans un wafer contenant 256 mémoires.

Conclusion

Dans ce chapitre nous avons présenté un outil d'injection de fautes pour l'évaluation du rendement induit par l'utilisation d'une technique de réparation donnée. Grâce à son organisation modulaire, cet outil peut aisément être adapté à de nouvelles techniques de réparation, en décrivant correctement les modules cibles indépendamment les uns des autres. On peut citer parmi les principaux modules, le module de génération software de la mémoire et de ses différents types de redondances. Un module d'injection de fautes, dans les cellules mémoire, sous forme d'amas de fautes (clustered faults) et en fonction d'une densité de défaut globale du wafer. Un module contenant un programme et qui reflète le comportement de la technique de réparation choisie. Ce dernier module est responsable de correctement manipuler les ressources redondantes disponibles et de déclarer si la mémoire fautive est réparable ou non. Enfin, un dernier module basée sur le modèle de clustering, pour l'étude du rendement au niveau wafer. L'évaluation purement statistique du rendement au niveau wafer nécessite un temps de simulation considérable. Pour remédier à ce problème, nous avons développé et intégré dans l'outil, une approche mixant une évaluation statistique et analytique du rendement. Le recours à une évaluation statistique dans cette dernière approche n'a lieu que si les calculs utilisant les formules analytiques ne peuvent être effectuées à cause des capacités limitées des ordinateurs (en termes de précision). Les formules analytiques qu'on a développé, ont également été utiles pour valider l'ensemble de notre approche statistique. Les résultats ont montré une bonne corrélation entre le rendement théorique (analytique) et expérimental (statistique). L'emploi de l'approche statistique seule reste extrêmement pratique pour les techniques de réparation ayant un comportement complexe et pour lesquelles il est très difficile d'élaborer des formules analytiques.

L'ensemble des techniques de réparation présentées dans les deux chapitres précédents a été intégré dans l'outil et ainsi un grand nombre de simulations a été effectuée pour évaluer le rendement. Parallèlement, à la détermination du rendement, il a fallu estimer la surface d'un grand nombre d'architectures suivant divers paramètres. Cela a été possible grâce à l'intégration d'une description générique (en C++) de ces architectures dans un générateur automatique de structures matérielles de test et de réparation pour mémoire (M-BISTeR [BOU02]).

Les expérimentations montrent que la réparation basée sur les polarités de fautes n'améliore pas le rendement, car ses bonnes capacités de réparation sont contrebalancées par

le coût élevé de ses fonctions de reconfiguration. D'un autre côté, l'efficacité de réparation a été améliorée grâce à l'approche diversifiée. Entre les deux techniques de réparation diversifiée, c'est celle qui combine le ECC avec la réparation mot qui est la plus efficace face aux très hautes densités de défauts, tout en restant dans des coûts modérés en terme de surface. Cette technique ouvre donc la voie vers la fabrication de mémoires dans des technologies susceptibles de véhiculer des densités de défauts très élevées.

La technique de réparation diversifiée combinant la reconfiguration dynamique et le remplacement bloc, est plus efficace pour les densités de défauts inférieures a 10^{-4}, mais elle peut être plus efficace face à des densités de défauts plus élevées pour des mémoires ayant des mots plus petits que 32 bits.

Conclusion et Perspectives

La course vers la miniaturisation des circuits intégrés engendre d'inévitables chutes du rendement en production et de fiabilité, dus à des défauts de fabrication ou à différentes sources de bruit. Les mémoires qui constituent jusqu'à 80 % des SoC, et qui utilisent la technologie la plus fine du transistor MOS, sont les plus propices à ces problèmes. Le BIST a été et continu d'être l'approche la plus répandue pour le test des mémoires embarquées. Mais à cause d'une densité de plus en plus croissante des défauts, l'action de test ne suffit plus et doit être complétée par une action de réparation. Le BISR semble aujourd'hui être l'alternative la plus intéressante pour la réparation des mémoires embarquées. Cependant, des exigences fortes en termes de coût en surface, d'efficacité de réparation et de pénalité temporelle induite sont à considérer pour concevoir des circuits BISR. A travers la première partie de nos travaux réalisés dans cette thèse, nous avons tenté de répondre à l'ensemble de ces exigences en présentant un large spectre de solutions de réparation intégrées pour les mémoires RAM. Ces différentes solutions permettent en effet de réparer les mémoires affectées par des densités de défauts très élevées, similaires à ce qu'on pourrait rencontrer dans les nanotechnologies, moyennant un coût en surface raisonnable et une faible pénalité temporelle.

Nous avons commencé dans cette thèse par développer la réparation intégrée au niveau bit de donnée. Ce type de réparation est très efficace car il permet de venir à bout d'un grand nombre de fautes pouvant survenir dans plusieurs blocs se situant sur les colonnes de la mémoire (cellules mémoire, multiplexeur colonne, circuit de lecture/écriture, décodeur des adresses colonnes, …). Les techniques de reptation colonne existantes, se basent généralement sur des machines d'états finies (FSM) pour répertorier tous les cas de défaillance possibles et générer les signaux nécessaires à la reconfiguration de la mémoire. A cause de son coût très élevé, on se limite souvent dans cette technique à réparer un nombre restreint de bits fautifs (un à deux bits fautifs), et à ne pas considérer les fautes dans la redondance. Nous avons pour notre part, adopte un raisonnement par récurrence qui nous a aisément permis d'élaborer les fonctions optimales (en termes de coût) pour la reconfiguration de la mémoire au niveau bit de donnée. Ces fonctions nous permettent facilement de concevoir le circuit de réparation pour n'importe quel nombre d'unités redondantes, et traitent à la fois les fautes sur les parties régulières et redondantes. Afin d'améliorer l'efficacité de la réparation au niveau bit, nous avons développé une technique de réparation dynamique qui permet de sélectionner des parties remplaçables de plus petite taille par simple accès externe

aux données de la mémoire et donc sans modification de la structure interne de celle-ci. A partir d'une certaine taille de l'unité remplaçable, les fonctions de reconfiguration deviennent plus coûteuses que les unités redondantes. Pour réduire le coût dû aux unités redondantes nous avons développé un mécanisme de réparation, toujours au niveau bit de donnée, qui utilise des colonnes redondantes singulières. Une colonne singulière est un sous ensemble d'une unité qui génère un bit de donnée. Les fonctions de reconfiguration d'une telle redondance introduit des interactions complexes entre les différents blocs de reconfiguration de base. Afin de maîtriser cette complexité nous avons introduit des variables intermédiaires pour exprimer ces interactions et nous les avons ensuite inséré dans les équations de la technique de base, pour obtenir les nouvelles équations. Enfin, si nous voulons à la fois réduire le coût des unités redondantes et améliorer l'efficacité de réparation, nous pouvons utiliser une reconfiguration dynamique qui utilise des colonnes redondantes singulières. Bien que très efficaces, la redondance en colonnes singulières a le désavantage de ne pas être disponible dans la plupart des compilateurs standards de mémoires. En effet, l'addition d'un tel type de redondance brise la symétrie de la mémoire et implique des changements conséquents au niveau de son layout. L'évaluation de l'efficacité de réparation au niveau bit de donnée a donné de très bons résultats en termes de rendement et de coût en surface, pouvant réparer des mémoires atteintes par des densités de défauts un à deux ordres de grandeurs supérieurs aux densités de défauts qui existent dans les technologies actuelles, mais insuffisantes si on veut considérer des densités de défauts qu'on pourrai rencontrer dans les nanotechnologies.

Dans les nanotechnologies, les parties régulières ainsi que les parties redondantes seront défaillantes et la réparation au niveau bit de donnée qui est basée sur le remplacement des unités régulières défaillantes par des unités redondantes correctes ne serai plus suffisante. Notre solution durant cette thèse a été de se baser sur un aspect particulier dans ces unités fautives, qui est la polarités des erreurs, pour pouvoir non plus remplacer mais combiner entre elles ces unités fautives pour générer des bits de données correctes. En se basant sur les polarités d'erreurs tout en adoptant un raisonnement récursif, nous avons pu élaborer les fonctions de reconfiguration qui permettent la réparation de la mémoire au niveau bit de donnée. Bien entendu, la reconfiguration dynamique a été appliquée dans ce cas de figure pour combiner entre elles, des unités défaillantes de plus petite taille, afin d'augmenter l'efficacité de la réparation.

Nous avons mis en place une deuxième approche, appelée réparation diversifiée de réparation pour faire face aux très hautes densités de défauts dans les nanotechnologies. Il

s'agit d'employer judicieusement plusieurs (deux dans notre cas) techniques de réparation à la fois. L'idée a été d'utiliser une première technique de réparation pour réparer la plupart des parties défaillantes, puis en utiliser une seconde pour réparer les parties défaillantes non réparées par la première technique. Dans le registre des réparations diversifiées, deux techniques distinctes ont été développées dans le cadre de cette thèse :

- La réparation dynamique au niveau bit de donnée (par remplacement ou par combinaison) avec la réparation bloc. Pour ce faire, nous avons du résoudre le problème complexe de la réparation bloc, qui utilise d'une part des blocs redondants pouvant être fautifs et d'autre part une mémoire CAM pour la sélection de ces blocs qui elle même peut contenir des fautes (très hautes densités de défauts).
- La technique des codes correcteurs d'erreurs (ECC) avec la réparation mot. L'élaboration de cette technique est passée par la résolution du problème de la réparation mot qui est plus général que le problème de la réparation bloc. En effet, dans ce cas une mémoire CAM est également utilisée mais dans laquelle on stocke à la fois les adresses des mots réguliers fautifs et les mots redondants.

Pendant notre étude, une des démarches importantes a été d'évaluer, pour chacune des solutions BISR, son efficacité de réparation et son coût matériel. Cela nous a permis d'améliorer progressivement nos solutions et relever ainsi le défi de développer des techniques de tolérance aux fautes pour les mémoires en nanotechnologies. Notre approche pour l'évaluation de l'efficacité a été essentiellement une approche statistique dans laquelle on a réalisé un grand nombre de simulations software d'injection de fautes dans les mémoires suivant une densité de défaut donnée. Nous simulons ensuite sur ces mémoires fautives, une technique de réparation précise puis nous recensons le nombre de mémoires qui ont été réparées, et ce nombre qui détermine la l'efficacité de la technique BISR en question. Lorsque nous voulons passer à l'évaluation du rendement d'un wafer de mémoires, nous avons du développer un programme qui donne la distribution des densités de défauts au sein du wafer suivant le modèle d'amas (clustering model). Cependant, à cause du grand nombre de simulations qu'exige une évaluation purement statistique du rendement au niveau d'un wafer, le temps des expérimentations devient considérable, d'où l'idée d'utiliser une approche mixte analytique/statistique.

Les simulations ont montré l'efficacité de nos solutions BISR pour des densités de défauts affectant les technologies actuelles ($x.10^{-6}$) jusqu'à des densités de défauts extrêmes ($x.10^{-2}$) qui pourraient affecter un jour les nanotechnologies. Un résultat important de cette étude est

que même pour des technologies d'une aussi mauvaise qualité, on pourra obtenir un rendement de production élevé en payant un coût en surface modéré (~ 60%) ; en comparaison avec les techniques de réparation des parties logiques nécessitant un coût en surface exorbitant (~ 25000%) pour une densité de défaut de 10^{-3} [HAN02].

Cette étude constitue une première base pour de futurs développements possibles. Les techniques BISR qu'on a développe dans le cadre de cette thèse sont généralistes car nous n'avons considéré aucune particularité fonctionnelle ou structurelle des mémoires RAM. Il serait donc intéressant d'exploiter des spécificité fonctionnelles et/ou structurelles de certaines mémoires pour réduire encore plus le coût en surface et si possible augmenter l'efficacité de la réparation, pour certaines mémoires RAM (mémoires séquentielles, mémoires asynchrones, schémas de redondances particuliers, …). Nous proposons également le développement d'outils permettant l'automatisation de ces techniques ou bien leur intégration dans des flux de conception existants. Un autre axe de développement possible est l'intégration de notre méthodologie d'injection de fautes et d'évaluation de l'efficacité dans un flux d'aide à la conception. Cela permettrais au concepteur de bien choisir l'ensemble des paramètres de réparation pour avoir le maximum de rendement possible en fonction de la densité de défaut. Un axe de développement supplémentaire est alors envisageable et qui concerne la prise en compte des différents modèles de distribution des fautes et qui s'appuient sur les données statistiques les plus à jour.

Notons à cette occasion que nous avons intégré certaines des solutions présentées dans ce manuscrit dans un outil industriel pour la génération automatique de descriptions RTL synthétisables [iroURL]. Elles ont par la même occasion été validées sur un grand nombre de mémoires (STMicrolectronics, Infineon, et Magma).

Bibliographie

[ABA83] M. S. Abadir, Hassan K. Reghbati: Functional Testing of Semiconductor Random Access Memories. ACM Computing Surveys 15(3): 175-198 (1983)

[ABR94] M. Abramovici, M.A. Breuer, A.D. Friedman, "Digital Systems Testing and Testable Design", IEEE Computer Science Press, September 1994, IBN: 0780310624.

[ACH01] N. Achouri, S. Boutobza, "Conception d'une Structure Built-In Self-Repair (BISR) pour Mémoires RAM". *IVème Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM)*, Strasbourg, France, 24-25 Avril 2001.

[ACH03] N. Achouri, "Fonctions Optimales pour la Reconfiguration des Mémoires au Niveau Données". *VIème Journées Nationales du Réseau Doctoral en Microélectronique (JNRDM)*, Toulouse, France, 14-16 Mai 2003.

[ALA94] W.K. Al-Assadi, A.P. Jayasumana, Y.K. Malaiya, "On Fault Modeling and Testing of Content Addressable Memories", Proc. Int. Memory Tech. Design and Test Workshop, Aug. 1994. pp. 78-83.

[BAR87] P. H. Bardell, et al., "Built-In Test for VLSI: Pseudorandom Techniques", John Wiley & Sons, New York, N. Y.

[BEN00] A. Benso et al, "A Family of Self-Repair SRAM Cores", IEEE International On-Line Testing Workshop, July 3-5, 2000, pp. 214-218.

[BHA94] D. K. Bhavsar, John H. Edmondson: Testability Strategy of the ALPHA AXP 21164 Microprocessor. International Test Conference, 1994, pp. 50-59.

[BHA99] D. k. Bhavsar, "An Algorithm for Row-Column Self-Repair of RAMs and Its Implementation in the Alpha 21264", International Test Conference, pp. 311-318, 1999.

[BLA83] R.E. Blahut, "Theory and Practise of Error Control Codes", Addison-Wesley Publishing Company, Reading, MA, 1983.

[BLO93] D.M. Blough and A. Pelc, "A Clustered Failure Model For The Memory Array Reconfiguration Problem", IEEE Transactions on Computers, Vol. 42, No. 5, pp. 518-528, May 1993.

[BLO96] D.M. Blough, "Performance Evaluation of a Reconfiguration-Algorithm for Memory Arrays Containing Clustered Faults", IEEE Transaction on Reliability, Vol.45, No. 2, pp. 274-284, June 1996.

[BOU02a] S.Boutobza, M. Nicolaidis, "Programmable Test for Memories", European Patent Office, Mars 2002, patent No 02354040.4-.

[BOU02b] S.Boutobza, "Outils de génération de Structures BIST/BISR pour Mémoires", Manuscrit de Thèse, December 2002.

[CHE84] C.L. Chen, and M. Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-art Review", IBM J. Res. Dev., vol. 28, no. 2, 1984, pp. 124-138.

[CLA83] K.L. Clarkson, "A modification of the greedy algorithm for vertex cover", Information Processing Letters, Vol.16, Jan. 1983, pp. 23-25.

[DAY85] Day, J. R., "A Fault-Driven, Comprehensive Redundancy Algorithm", IEEE Design and Test of Computers, pp. 35-44, June 1985.

[FRA90] Manoj Franklin, Kewal K. Saluja, "Built-in Self-testing of Random-Access Memories". IEEE Computer 23(10): 45-56 (1990).

[GOO90] A. J. van de Goor, C. A. Verruijt, "An Overview of Deterministic Functional RAM Chip Testing". ACM Comput. Surv. 22(1): 5-33 (1990).

[GOO91] Goor, A.J. van de, "Testing Semiconductor Memories, Theory and Practice", (536 pages) John Wiley & Sons, Chichester, UK, 1991.

[HAN02] J. Han, D. Jonker, "A system Architecture Solution for Unreliable Nanoelectronic Devices", IEEE Transactions on Nanotechnology, vol.1, No. 4, December 1992.

[HEA98] J.R. Heath, Kuekes P.J., Snider G.S., Stanley Williams R., "A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology", SCIENCE, Vol. 280, June 12, 1998.

[HOP73] J. Hopcroft and R. Karp, "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs", SIAM J. Computing, Vol. 2, Dec. 1973, pp. 225-231.

[HOP82] J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", Proc. Natl. Acad. Sci. U.S.A., vol. 79, 1982, pp. 2554-2558.

[HUA02] R. Huang, and al., "A Simulator for Evaluating Redundancy Analysis Algorithms of Repairable Embedded Memories", In Proc. IEEE International Workshop on Memory Technology, Design and Testing (MTDT 2002),

[HUA90] W. K. Huang, Y. N. Shen and F. Lombardi, "New Approaches for the Repair of Memories With Redundancy by Row/Column Deletion for Yield Enhancement", IEEE Transaction on CAD of Integrated Circuits and Systems, pp. 323-328, Mars 1990.

[iRoURL] www.iroctech.com .

[JAY02] J. Jayabalan, J. Povazanec, "Integration of SRAM Redundancy into Production Test", IEEE International Test Conference, pp. 187-193, 2002

[KEB92] O. Kebichi, M. Nicolaidis, "A Tool for Automatic Generation of BISTEed and Transparent BISTed RAMs", International Conference on Computer Design, Cambridge, USA. October 1992, p. 570-575.

[KIM99] H. C. Kim, D. S. Yi, J. Y. Park, C. H. Cho, "A BISR (Built-In Self-Repair) circuit for embedded memory with multiple redundancies", IEEE International Conference on VLSI and CAD, pp. 602-605, Seoul, Korea, October 1999.

[KIM98] I. Kim, Y. Zorian, G. Komoriya, H. Pham, F. Higgins, J. Lewandowski, "Built-In Self-Repair for Embedded High Density SRAM", International Test Conference, pp. 1112-1119, 1998.

[KUO87] S.Y. Kuo and W.K. Fuchs, "Efficient Spare Allocation in Reconfigurable Arrays", IEEE Design & Test, Vol. 4,Issue. 9, pp. 15-22, February 1987.

[LEV92] J. R. Levine, and al., "Lex & Yacc", 2nd Updated Edition O'Reilly & Associates, October 1992, ISBN: 1565920007.

[LIN98] K. Lin, C-W. Wu, "Functional Testing of Content-Addressable Memories", IEEE Memory Technology, Design and Test Workshop 1998, pp. 70-5.

[MAR82] M. Marinescu, "Simple and Efficient Algorithms for Functional RAM Testing," Proc. Int'l Test Conf, pp. 236-239, 1982.

[MAZ87] P. Mazumder, J. Patel, "An Efficient Built-In Self Testing for Random-Access Memory", IEEE Transactions on Industrial Electronics, Vol. 36, No. 2, May. 1989, pp. 246-253.

[MAZ88] P. Mazumder, "Parallel Testing of Parametric Faults in a Three-Dimensional Dynamic Random-Access Memory," IEEE Journal of Solid-State Circuits, Vol. 23, No. 4, Aug. 1988, pp. 933-942.

[MAZ90] P. Mazumder, J.S. Yih, "A Novel Built-In Self-Repair Approach to VLSI Memory Yield Enhancement", Proc. IEEE Int. Test Conf., 1990, pp. 833-841.

[MCP00] R.J. McPartland, D.J. Loeper et al., "SRAM Embedded Memory with Low Cost, FLASH EEPROM-Switch-Controlled Redundancy", Custom Integrated Circuits Conference, 2000. CICC. Proceedings of the IEEE 200, Vol. 36, No. 11, pp. 287-289, May 2000.

[NAD90] B. Nadeau-Dostie, A. Sulburt, and V.K. Agrawal, "Serial Interfacing for Embedded-Memory Testing," IEEE Design and Test of Computers, vol. 7, no. 2, pp. 52-63, Apr. 1990.

[NAI78] Ravindra Nair, Satish M. Thatte, Jacob A. Abraham: Efficient Algorithms for Testing Semiconductor Random-Access Memories. IEEE Trans. Computers 27(6): 572-576 (1978).

[NIC85] M. Nicolaidis, "An Efficient Built-In Self-Test Scheme for Functional Test of Embedded RAMs", Proc. Of the IEEE Fault Tolerant Computer Systems Conf., 1985, pp. 118-123.

[NIC96] Michael Nicolaidis: Theory of Transparent BIST for RAMs. IEEE Trans. Computers 45(10): 1141-1156 (1996)

[NIC01] M. Nicolaidis, S. Boutobza, M. N. Achouri, "Designing and Implementing Efficient BISR Techniques for Embedded RAMs", 2nd IEEE Latin America Test Workshop (LATW 2001), 11-14 February 2001, Cancun, Mexico.

[NIC03a] M. Nicolaidis, N. Achouri, S. Boutobza: Optimal Reconfiguration Functions for Column or Data-bit Built-In Self-Repair. IEEE Design, Automation and Test in Europe (DATE2003), 03-07 March 2003, Munich, Germany, ISBN 0-7695-1870-2.

[NIC03b] M. Nicolaidis, N. Achouri, L. Anghel : Memory Built-In Self-Repair for nanotechnologies, 9th IEEE International On-Line Testing Symposium (IOLTS2003), 07-09 July 2003, Kos Island, Greece, ISBN 0-7695-1968-7.

[NIC03c] M. Nicolaidis, N. Achouri, L. Anghel: A Memory Built-In Self-Repair for High Defect Densities Based on Error Polarities,18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), 03 – 05 November 2003, Cambridge, MA, U.S.A.

[NIC03d] M. Nicolaidis, N. Achouri, S. Boutobza: Dynamic Data-bit Memory Built-In Self-Repair, IEEE International Conference on Computer Aided Design (ICCAD-2003), 09-13 November 2003, San Jose, CA, U.S.A.

[NIC04] M. Nicolaidis, N. Achouri, L. Anghel: Diversified Memory Repair approach Based on ECC and Word Repair, 22nd IEEE VLSI Test Symposium (VTS 2004), 25-29 April 2004, Napa Valley, CA, U.S.A.

[NOW87] E.J. Nowak, W.M. Trickle, "Analysis and solutions of a yield-limiting patterned fault mechanism in a 1-Mb DRAM", VLSI Symposium, May 1987.

[OOK93] S. Ookuma, et al., "An Effective Defect-repair Scheme for a High Speed SRAM", IEICE Trans. Electron., vol. 76-C, no. 11, 1993, pp. 1620-1625.

[SAL85] Kewal K. Saluja, Kozo Kinoshita: Test Pattern Generation for API Faults in RAM. IEEE Trans. Computers 34(3): 284-287 (1985)

[SAW89] K. Sawada, T. Sakurai, Y. Uchino, and K. Yamada, "Built-In Self-Repair Circuit for High-Density ASMIC", IEEE Custom Integrated Circuits Conference, 1989, pp. 26.1.1-26.1.2.

[SCH01] V. Schober, S. Paul, O. Picot, "Memory Built-In Self-Repair using redundant words", IEEE International Test Conference, pp. 995-1001, 2001.

[SEH03] A. Sehgal, and al., "Yield Analysis for Repairable Embedded Memories", In Proc. of IEEE VLSI Test Symposium, April 2003, CA, USA.

[SHI83] K. Shimohigashi, et al., "Redundancy Techniques for Dynamic RAMs", Japan J. Applied Phys., vol. 22, suppl. 22-1, 1983, pp. 63-67.

[SID99] P. R. Sidorowicz, "Modeling and Testing Transistor Faults in Content-Addressable Memories", IEEE Memory Technology, Design and Test Workshop 1999, pp. 83-90.

[STA80] C. H. Stapper, A. N. McLaren and M. Dreckmann, "Yield Model for Productivity Optimization of VLSI Memory Chips with Redundancy and Partially Good Product", IBM J. Research & Development, Vol. 24, pp. 398-409, May 1980.

[STA86] C. H. Stapper, "On Yield, Fault Distributions, and Clustering of Particles", IBM J. Research & Development, Vol. 30, No. 3, pp. 326-338, May 1986.

[STA89a] C.H.Stapper, "Fault-simulation programs for integrated-circuit yield estimations", IBM J. Res. Develop, vol. 33, no 6, 1989.

[STA89b] C.H. Stapper, "Fault-simulation for fault-tolerant multi-Mbit RAMs", IEEE Int. Conference on Microelectronic Test Structures, vol. 2, no 1., March 1989.

[STA92] C.H. Stapper, H. S. Lee, "Synergetic Fault-Tolerance for Memory Chips", IEEE Transaction on Computers, Vol. 41, No. 9, pp. 1078-1087, September 1992.

[SUK80] D. S. Suk, Sudhakar M. Reddy, "Test Procedures for a Class of Pattern-Sensitive Faults in Semiconductor Random-Access Memories". IEEE Transaction on Computers, C-29, pp. 419-429.

[SUN93] T. Chen, G. Sunada, "Design of a self-testing and Self-repairing Structure for Highly Hierarchical Ultra-large Capacity Memory Chips", IEEE Trans. VLSI Syst. Vol. 1, no. 2, 1993, pp. 88-97.

[TAN92] A. Tanabe, T.Takeshima, H. Koike, Y. Aimoto, M. Takada, et. al, "A 30 ns 64-Mb DRAM with Built-In Self-Test and a Self-Repair Function", IEEE Journal on Solid-State Circuits, Vol. 27, November 1992.

[TAR84] M. Tarr, et al., "Defect Analysis System Speeds Test and Repair of Redundant Memories", Electronics, 1984, pp. 175-179.

[THA77] S.M. THATTE, ABRAHAM, J. A. 1977. Testing of semiconductor random access memories. In Proceedings of the 7th Annual International Conference on Fault-Tolerant Computing, pp. 81-87.

[WEY87] Wey, C. and Lombardi, F., “On the Repair of Redundant RAMs”, IEEE Trans. CAD/ICAS, vol. 6, no. 2, 1987, pp. 222-231.

[YAN72] T. Yanagawa, “Yield degradation of integrated circuits due to spot defects”, IEEE trans. Electronic Devices, vol. ED-19, February 1972.

LISTE DES PUBLICATIONS PERSONNELLES

L.Anghel, N. Achouri, M. Nicolaidis, "Evaluation of Memory Built-In Self-Repair Techniques for High Defect Density Technologies" to be published in Proceedings on *IEEE Reliable Dependable Computing 2004*, Tahiti, French Polynesia, March 2004.

M. Nicolaidis, N. Achouri, L. Anghel: "A Diversified Memory Built-In Self-Repair Approach for Nanotechnologies", *22nd IEEE VLSI Test Symposium (VTS 2004)*, 25-29 April 2004, Napa Valley, CA, U.S.A.

M. Nicolaidis, N. Achouri, S. Boutobza: "Dynamic Data-bit Memory Built-In Self-Repair", *IEEE International Conference on Computer Aided Design (ICCAD-2003)*, 09-13 November 2003, San Jose, CA, U.S.A.

M. Nicolaidis, N. Achouri, L. Anghel: "A Memory Built-In Self-Repair for High Defect Densities Based on Error Polarities",*18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03)*, 03 – 05 November 2003, Cambridge, MA, U.S.A.

M. Nicolaidis, N. Achouri, L. Anghel : "Memory Built-In Self-Repair for Nanotechnologies", 9th *IEEE International On-Line Testing Symposium (IOLTS2003)*, 07-09 July 2003, Kos Island, Greece, ISBN 0-7695-1968-7.

M. Nicolaidis, N. Achouri, S. Boutobza: "Optimal Reconfiguration Functions for Column or Data-bit Built-In Self-Repair". *IEEE Design, Automation and Test in Europe (DATE2003)*, 03-07 March 2003, Munich, Germany, ISBN 0-7695-1870-2.

M. Nicolaidis, S. Boutobza, N. Achouri: "Designing and Implementing Efficient BISR Techniques for Embedded RAMs", *2nd IEEE Latin America Test Workshop (LATW 2001)*, 11-14 February 2001, Cancun, Mexico.