Numéro d'ordre : 1276

THÈSE

présentée devant

l'Université de Rennes 1

Institut de Formation Supérieure en Informatique et Communication

pour obtenir

Le Titre de Docteur de l'Université de Rennes I
Mention INFORMATIQUE

par

Alain GEFFLAUT

Titre de la thèse

Proposition et évaluation d'une architecture multiprocesseur extensible à mémoire partagée tolérante aux fautes

Soutenue le 5 janvier 1995 devant la commission d'examen :

MM. :	Jean-Pierre	BANÂTRE	**Président**		
	William	JALBY	**Rapporteur**	|	
	Daniel	LITAIZE	**Rapporteur**	|	
	Michel	BANÂTRE		|	**Examinateurs**
	Christine	MORIN		|	
	André	SEZNEC		|	
	Daniel	WINDHEISER		|	

Ce travail de thèse s'est déroulé à l'IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires), au sein du projet SOLIDOR.

Je tiens avant tout à remercier mes parents qui ont su éveiller en moi la curiosité et la perspicacité nécessaires à la réalisation de cette thèse.

Je remercie également mon directeur de thèse, Michel Banâtre, directeur de recherche à l'INRIA et responsable du projet SOLIDOR, ainsi que Christine Morin, chargée de recherche à l'INRIA, pour l'encadrement et les conseils qu'ils m'ont apportés.

J'exprime ma reconnaissance à l'ensemble des membres du jury, président, rapporteurs et examinateurs, pour avoir bien voulu donner une partie d'un temps qui leur est précieux pour lire et commenter ce travail de thèse. Je remercie tout particulièrement André Seznec, responsable du projet CAPS, pour les nombreuses discussions que nous avons eues et qui m'ont bien souvent éclairé.

Je tiens également à remercier l'ensemble des membres du projet SOLIDOR pour les moments de détente que nous avons partagés et l'aide qu'ils m'ont apportée tout au long de cette thèse. Plus particulièrement, je remercie Philippe Joubert pour sa collaboration dans de nombreux aspects de ce travail. Les fréquentes conversations que nous avons eues m'ont souvent permis de trouver les solutions aux problèmes auxquels j'étais confronté. Je remercie aussi Erwan Moysan, avec qui je partageais mon bureau, et qui a contribué pour beaucoup au bon déroulement de ces trois années.

Merci également à Boris Charpiot pour l'aide qu'il m'a apportée dans le développement du simulateur utilisé dans cette thèse. Merci à Anne-Marie Kermarrec et Benoît Dupin pour les lectures et les corrections apportées à ce document.

Enfin et surtout, merci à Caroline pour le soutien qu'elle a su me donner, y compris dans les moments les plus difficiles de l'écriture de cette thèse.

Chapitre I

Introduction

I.1 Multiprocesseurs extensibles à mémoire partagée

Il est de plus en plus difficile d'obtenir une amélioration significative des performances des ordinateurs simplement par l'amélioration des technologies matérielles. Les architectures parallèles fournissent l'opportunité de gains de performance importants en additionnant la puissance de calcul d'un ensemble de processeurs travaillant simultanément. Pour ces architectures, il est impératif de proposer un modèle de programmation à la fois simple et puissant qui permette de tirer facilement parti des caractéristiques de l'architecture.

Les multiprocesseurs utilisant une mémoire partagée offrent une solution à ce problème. À la différence des systèmes communicant par échange de messages, où les processeurs n'ont accès qu'à leur mémoire locale et où les communications sont explicites, un multiprocesseur à mémoire partagée offre à chaque processeur l'accès à la totalité de l'espace mémoire physique de la machine. Ces architectures facilitent ainsi le partitionnement de données et l'équilibrage de charge dynamique qui représentent les difficultés majeures de programmation des applications parallèles. L'utilisation d'un espace d'adressage unique simplifie également la parallélisation automatique de programmes et le portage des applications. Elle permet notamment de proposer des systèmes d'exploitation standard [Schimmel 90].

Alors que les aspects positifs des multiprocesseurs à mémoire partagée sont largement acceptés, les problèmes d'extensibilité qu'ils posent sont aussi souvent cités. A l'origine les multiprocesseurs à mémoire partagée étaient surtout organisés autour d'un bus unique permettant d'accéder à la mémoire. De telles architectures ne peuvent cependant comporter qu'un petit nombre de processeurs (moins de 20) du fait de la bande passante limitée de leur bus. De nombreux travaux sont actuellement menés pour corriger ce défaut et concevoir des architectures à mémoire partagée comportant un grand nombre de processeurs [Agarwal *et al.* 91, Hagersten *et al.* 92, KSR 92, Lenoski *et al.* 92b, Litaize *et al.* 92, Weber 93, Oed 93, Joe & Hennessy 94, J.Kuskin *et al.* 94]. Ces architectures remplacent le bus par un réseau d'interconnexion reliant un ensemble de nœuds de calcul. Elles sont dites **extensibles** (ou

évolutives) car leur puissance de calcul doit augmenter sensiblement avec le nombre de processeurs. Elles sont aussi qualifiées de **machines massivement parallèles** car le nombre de processeurs qu'elles sont supposées supporter peut atteindre plusieurs milliers. Les architectures T3D de Cray et KSR1 de Kendall Square Research, proposent ainsi une mémoire partagée et peuvent atteindre des configurations comprenant jusqu'à 1024 processeurs. Ce sont des candidats sérieux en vue d'assurer des besoins croissants en puissance de calcul tels que ceux nécessaires pour les grands "challenges" du calcul scientifique (modélisation climatique, génome humain, etc).

I.2 Disponibilité des architectures extensibles

Bien que les progrès de la microélectronique aient permis d'améliorer de façon très importante la fiabilité des composants matériels des ordinateurs, les problèmes de fiabilité[1] et de disponibilité[2] d'une machine, souvent ignorés du fait de la rareté des fautes quand le nombre de composants est limité, deviennent cruciaux pour une machine extensible et ceci pour au moins deux raisons. D'une part, avec l'augmentation du nombre de composants d'une architecture, la probabilité d'une défaillance survenant quelque part dans le système augmente aussi. On peut estimer qu'un système comprenant 1000 nœuds ayant chacun un MTBF (temps moyen entre deux pannes) de 30000 heures (soit environ 1 panne tous les 3 ans), aura lui-même un MTBF de 30 heures c'est-à-dire connaîtra en moyenne pratiquement une panne par jour. D'autre part, les applications s'exécutant sur ce type de machine nécessitent des temps d'exécution longs qui ne peuvent être envisagés avec les taux de défaillance annoncés précédemment. Il est donc indispensable que ces architectures intègrent, dès leur conception, des mécanismes leur assurant une disponibilité et une fiabilité suffisantes pour être réellement utilisables. Ce sont les objectifs de la tolérance aux fautes. Dans le cas de machines à mémoire partagée, ce besoin de tolérance aux fautes est de plus renforcé par le fait que la défaillance d'un seul élément de la machine peut rapidement conduire à une défaillance globale de l'architecture. Il est aussi nécessaire d'ajouter à ces fautes matérielles l'ensemble des fautes provenant du logiciel qui représentent habituellement la majorité des fautes rencontrées [Gray 90].

Il apparaît donc clairement que les architectures extensibles à mémoire partagée ont besoin de mécanismes de tolérance aux fautes leur permettant d'assurer une continuité de service, malgré la défaillance de certains de leurs composants. Cette préoccupation apparaît chez tous les constructeurs de ce type de machine qui proposent des mécanismes de tolérance aux fautes minimaux autorisant habituellement une reconfiguration de l'architecture en inhibant un composant défaillant. Bien entendu, les calculs en cours d'exécution au moment de la défaillance sont perdus et doivent être intégralement réexécutés. Ce type de solution ne répond donc qu'au problème de disponibilité de l'architecture, sans résoudre celui de sa fiabi-

[1] La fiabilité est définie comme la probabilité qu'une machine fonctionne jusqu'à un temps t.

[2] La disponibilité est définie comme la probabilité qu'une machine fonctionne à un instant t.

lité. Des calculs de longue durée ne sont en effet pas envisageables sans l'aide de mécanismes de tolérance aux fautes permettant de tolérer la défaillance d'un nœud de l'architecture tout en assurant la continuité des calculs en cours ("*graceful degradation*").

I.3 Objectifs

Les architectures de type COMA (Cache Only Memory Architectures), sont une classe d'architectures extensibles à mémoire partagée [Hagersten *et al.* 92, KSR 92]. Elles ont comme particularité de transformer les mémoires standard des nœuds de l'architecture en caches de grande dimension de l'espace mémoire partagé, offrant ainsi des mécanismes de migration et de réplication automatiques des données dans les mémoire des nœuds de l'architecture. Parallèlement, la récupération arrière est une technique de tolérance aux fautes qui nécessite la conservation et la réplication d'un ensemble de données appelées données de récupération.

L'objectif de cette étude est d'utiliser les mécanismes de réplication de données offerts par une architecture de type COMA pour implémenter une stratégie de récupération arrière de façon à limiter à la fois le coût de son développement matériel et la dégradation de performance qu'elle engendre. La solution proposée consiste à étendre le protocole de cohérence utilisé par les mémoires de l'architecture afin qu'elles intègrent de façon transparente la gestion des données de récupération à celle des données courantes. Cette solution permet de considérer la défaillance de tous les composants d'un nœud (processeur aussi bien que mémoire) en utilisant non pas du matériel spécifique tel que des mémoires stables [Banâtre *et al.* 93a], mais les mémoires traditionnelles et les mécanismes de réplication de données d'une architecture COMA, pour assurer le stockage et la réplication des données de récupération nécessaires à l'implémentation d'une stratégie de récupération arrière. En conservant les données de récupération dans les mémoires des nœuds, une telle implémentation permet également de limiter la dégradation de performance en assurant une sauvegarde des points de récupération efficace.

Bien entendu, il serait illusoire d'envisager la construction de cette architecture sans l'avoir évaluée au préalable. Une partie de l'étude porte donc sur l'évaluation, par simulation, de notre proposition, en vue de caractériser les facteurs influant sur la dégradation de performance engendrée. Cette évaluation est réalisée à l'aide de traces d'adresses d'applications parallèles.

I.4 Organisation du document

Le document se divise en trois parties.

La première partie du document est principalement consacrée à l'étude des mécanismes de réplication de données. Le chapitre II présente l'intérêt des mémoires cache qui permettent

une réplication et une migration automatique des données et limitent les temps d'attente des processeurs. Après une brève introduction à la tolérance aux fautes, le chapitre III s'attarde sur la gestion des données de récupération nécessaires pour la mise en œuvre d'une technique de récupération arrière. Il présente notamment les besoins en réplication de données qu'une telle technique engendre.

La seconde partie de l'étude débute par un rapprochement des mécanismes de réplication de données offerts par une architecture COMA qui n'utilise que des caches, et des besoins de réplication engendrés par la sauvegarde de données de récupération. Il en ressort que les mécanismes de réplication offerts par ces architectures sont tout à fait adaptés aux besoins d'une technique de récupération arrière. Le chapitre V présente alors un protocole de cohérence étendu reprenant ces idées. Ce protocole intègre de façon transparente la gestion des données courantes et la gestion des données de récupération. Il permet de plus d'assurer l'ensemble des propriétés demandées pour tolérer la perte d'un nœud de l'architecture sans nécessiter de développement de matériel coûteux.

La troisième partie du document se focalise sur l'implémentation et l'évaluation de notre proposition. Le chapitre VI décrit les modifications à apporter pour intégrer le protocole de cohérence étendu au sein d'une architecture COMA non-hiérarchique. Le chapitre VII présente des résultats de simulation pour l'architecture considérée. Une étude détaillée de la dégradation de performance et de l'extensibilité de l'approche y sont réalisées. Enfin, le chapitre VIII étudie plusieurs optimisations permettant de minimiser la dégradation de performance ainsi que certains problèmes qui n'ont pas été abordés dans le reste du document.

Première Partie

Réplication de données

Dans les architectures extensibles à mémoire partagée, la réplication de données est utilisée pour réduire les temps d'accès mémoire. L'utilisation de caches permet alors d'assurer cette réplication de façon automatique mais nécessite l'introduction de protocoles de cohérence pour maintenir à jour l'ensemble des répliques des lignes mémoire.

La réplication de données est aussi largement utilisée dans le domaine de la tolérance aux fautes. Ainsi une technique de récupération arrière se base sur la réplication de l'état mémoire courant d'un système lors de la sauvegarde d'un point de récupération. Elle peut également utiliser une réplication des données de récupération conservées pour leur assurer des propriétés de stabilité et limiter ainsi les hypothèses sur les fautes tolérées.

Cette première partie du document présente ces différents types de réplication de données ainsi que leurs mises en œuvre. Le premier chapitre détaille les mécanismes de gestion de la réplication dans les architectures extensibles à mémoire partagée. Après avoir présenté les principes de la récupération arrière, le second chapitre s'intéresse à la réplication de données nécessaire à l'implémentation d'une telle technique de tolérance aux fautes.

Chapitre II

Réplication de données et efficacité

Dans une architecture multiprocesseur, la réplication de données permet de réduire les temps de latences des accès mémoire des processeurs et améliore ainsi leur taux d'utilisation. Généralement, cette réplication est assurée de façon automatique par des **mémoires cache**. Ce chapitre commence par présenter l'intérêt des mémoires cache, en particulier dans le cas d'architectures extensibles à mémoire partagée. Il aborde ensuite la gestion de la réplication introduite par les caches afin de maintenir **cohérent** l'ensemble des copies des lignes mémoire. Les différents types de protocole de maintien de la cohérence ainsi que leur mise en œuvre au sein d'architectures extensibles à mémoire partagée sont présentés. Le chapitre se termine par la présentation des architectures COMA où les mémoires traditionnelles des nœuds sont utilisées comme des caches de grande dimension de l'espace mémoire partagé.

II.1 Multiprocesseurs à mémoire partagée

Un multiprocesseur à mémoire partagée est constitué d'un ensemble de nœuds de calcul connectés par un médium d'interconnexion et partageant un espace mémoire physique unique. Chaque nœud peut contenir un ou plusieurs processeurs. Les communications entre processeurs sont réalisées de façon transparente par des accès directs à la mémoire sans utilisation de messages explicites. Cette propriété facilite la programmation de ce type d'architecture en simplifiant la résolution de problèmes tels que la distribution de données ou l'équilibrage de charge qui sont deux des problèmes majeurs dans la programmation des machines parallèles.

Le médium d'interconnexion d'un multiprocesseur à mémoire partagée peut être physiquement implanté de différentes manières. Les exemples les plus courants sont les bus qui simplifient l'implémentation mais limitent l'extensibilité de l'architecture. Les architectures extensibles à mémoire partagée remplacent les bus par des réseaux d'interconnexion tels que des réseaux directs (grilles, hypercubes...), ou des réseaux multi-étages. Ces réseaux assurent une bande passante mémoire qui augmente avec le nombre de nœuds et ne constituent donc

plus un goulot d'étranglement pour accéder à la mémoire.

II.1.1 Organisation mémoire

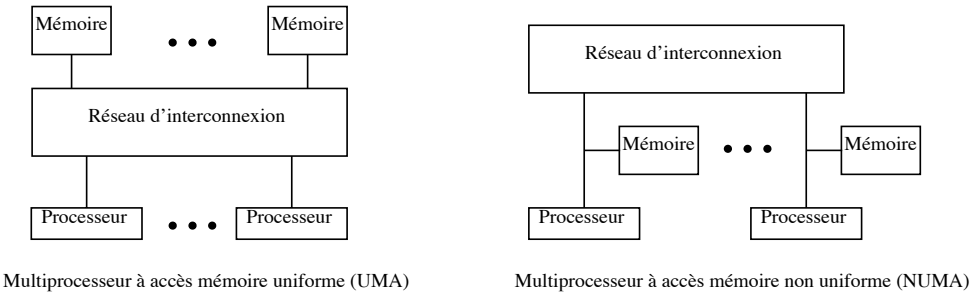


Figure II.1 Classes de multiprocesseurs à mémoire partagée

Pour réduire la contention mémoire et assurer des accès mémoire parallèles, la mémoire physique d'un multiprocesseur à mémoire partagée est habituellement constituée de multiples modules. L'organisation physique de cette mémoire partagée permet de distinguer deux classes d'architecture :

(1) Les machines à accès mémoire uniforme (*Uniform Memory Access*, UMA) où les temps d'accès à la mémoire sont, en l'absence de conflits, égaux pour tous les processeurs. C'est le cas des multiprocesseurs à bus, utilisant des réseau d'interconnexion multi-étages ou des liens séries entre les processeurs et les mémoires [Litaize *et al.* 92].

(2) Les machines à accès mémoire non-uniforme (*Non Uniform Memory Access*, NUMA) où les mémoires sont physiquement distribuées sur les nœuds de l'architecture. Les processeurs mettent alors moins de temps à accéder à leur mémoire locale qu'à une mémoire distante.

Les machines NUMA permettent de limiter les latences mémoire et l'occupation réseau en allouant, dans leur mémoire locale, les données utilisées par les processeurs. Ce type d'organisation mémoire tend à devenir l'organisation standard des multiprocesseurs extensibles à mémoire partagée. Nous nous limiterons à ce type d'architectures dans la suite de notre étude.

II.2 Utilité de la réplication de données

Pour atteindre des performances importantes et assurer une bonne extensibilité à un multiprocesseur, il est primordial d'assurer un taux d'utilisation élevé des processeurs. Pour cela,

un multiprocesseur doit offrir une **bande passante mémoire importante** et **minimiser les temps d'accès mémoire** des processeurs.

Dans un multiprocesseur, une bande passante mémoire élevée, garantie par une minimisation de la contention sur la mémoire et le réseau, assure qu'un grand nombre de requêtes mémoire pourront être traitées simultanément et minimise le temps moyen de traitement d'une requête. Au contraire des architectures à base de bus, les architectures de type NUMA assurent une bande passante mémoire élevée et extensible avec le nombre de nœuds, en distribuant les mémoires sur les nœuds de l'architecture et en utilisant des réseaux d'interconnexion extensibles.

Cependant, dans une architecture multiprocesseur, disposer d'une bande passante mémoire élevée n'est pas suffisant pour garantir à un processeur un taux d'utilisation maximal. Ainsi, même en l'absence de contention, un processeur dont le code est accessible localement, qui exécute une instruction par cycle et réalise un accès mémoire toutes les quatre instructions[1] [J. Hennessy 90], aura un taux d'utilisation inférieur à 4% si ses accès mémoire sont distants et prennent 100 cycles. De tels temps d'accès mémoire sont courants dans les architectures de type NUMA, qui doivent donc utiliser des techniques permettant de réduire les temps de latence des accès mémoire.

II.2.1 Réduction de latences et caches

Parmi les techniques de réduction des latences mémoire [Gupta *et al.* 91, Hagersten *et al.* 92], la réplication automatique des données dans des **mémoires cache** associées aux processeurs est certainement la solution la plus répandue ainsi que la plus efficace. Les caches sont des mémoires rapides et habituellement de faible capacité qui permettent de conserver les données les plus récemment référencées par un processeur. Ils tirent parti des localités **spatiale** et **temporelle** des références mémoire [Denning 72, Smith 82] d'une application pour conserver des données qui seront prochainement réutilisées par le processeur. Dans le cas d'une architecture de type NUMA, la réplication dans un cache d'une donnée située normalement dans une mémoire distante, permet ainsi de servir localement, et donc de façon plus rapide, plusieurs références mémoire.

Avec un cache de données accessible en 1 cycle et une bande passante illimitée, le taux d'utilisation du processeur considéré précédemment est donné par l'équation suivante :

$$Taux\ d'utilisation = \frac{1}{1+\frac{(1-hit)*Latence}{4}}$$

où *Latence* représente la latence de l'accès mémoire distant et *hit* le taux de succès dans le cache de données. Le graphe de la figure II.2.1 donne pour différentes latences mémoire, le taux d'utilisation d'un processeur en fonction du taux de succès dans son cache. Il apparaît clairement au vu de ces résultats que l'utilisation de caches associée à des taux de succès

[1] Ces fréquences sont en moyenne celles que l'on observe dans la plupart des applications.

élevés est une condition incontournable en vue d'atteindre des taux d'utilisation importants des processeurs garantissant une efficacité optimale à un multiprocesseur.

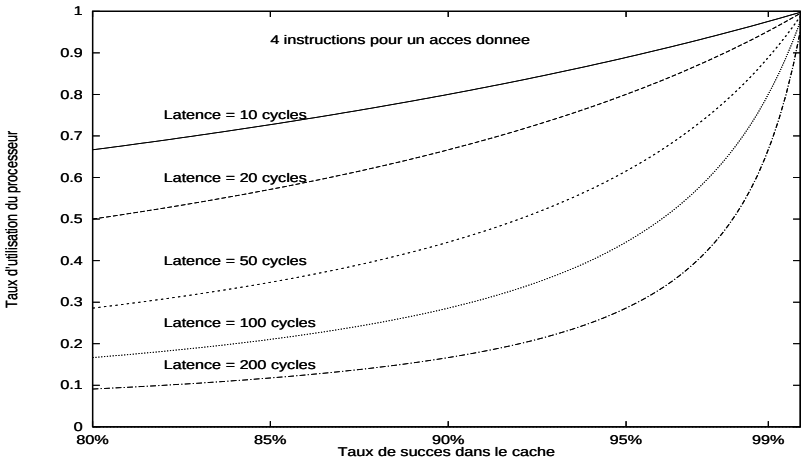


Figure II.2 Taux d'utilisation d'un processeur

La vitesse des processeurs doublant tous les deux ans, et augmentant beaucoup plus rapidement que la vitesse des mémoires ou des liens de communication, il n'est pas prévu de réduction de la durée des accès distants dans les années à venir. Comme le montre la table II.1, les architectures extensibles à mémoire partagée ont, ou auront des temps d'accès distants supérieurs à 100 cycles processeur. Les caches se révèlent donc être des composants indispensables d'une architecture de ce type. Dans un multiprocesseur à mémoire partagée, un autre effet des caches est de limiter l'occupation du réseau puisqu'une partie des données partagées peut être répliquée localement. Cette diminution de la charge du réseau limite à son tour les temps d'attente des processeurs et augmente l'efficacité ainsi que l'extensibilité de la machine.

Machine	Topologie du réseau	Accès distant (cycles processeur)	Accès cache 1er niveau (cycles processeur)
Cray T3D	Tore 3D	120	1
Kendall Square KSR 1	Hiérarchie d'anneaux	130 - 600	2
Stanford DASH	Grille 2D	100-130	1
Stanford FLASH	Grille 2D	111-191	1
DDM	Hiérarchie de bus	55-170	1
Convex Exemplar	Liens SCI [2]	200	1

Table II.1 Exemples d'architectures extensibles à mémoire partagée

Les caches peuvent être mis en œuvre de façon logicielle comme dans les mémoires virtuelles partagées implantées au-dessus d'architectures à mémoire distribuée [Delp 88, Li & Hudak 89, Michel 89, Lahjomri & Priol 92], ou directement par matériel comme les caches intégrés aux processeurs [J. Hennessy 90]. Dans ce document, nous nous intéressons plus

particulièrement aux mises en œuvre matérielles des caches. Les résultats restent cependant valides dans le cas de caches logiciels.

II.3 Gestion de la réplication

II.3.1 Cohérence

L'introduction de caches dans un multiprocesseur n'est pas sans poser un certain nombre de problèmes. Si chaque nœud est autorisé à maintenir une copie locale de n'importe quel emplacement mémoire (ligne mémoire), il est impératif que toutes ces répliques soient maintenues identiques quand l'une d'entre elles est modifiée par l'un des processeurs. Dans le cas contraire, des **incohérences** pourraient apparaître car certains processeurs disposeraient de copies non à jour d'une donnée. La définition généralement utilisée pour la cohérence a été donnée dans [Censier & Feautrier 78]

Définition 1 (Cohérence - première définition)
Un ensemble de caches est dit cohérent si la lecture d'un emplacement mémoire par un processeur retourne la valeur de la dernière écriture effectuée à cette adresse.

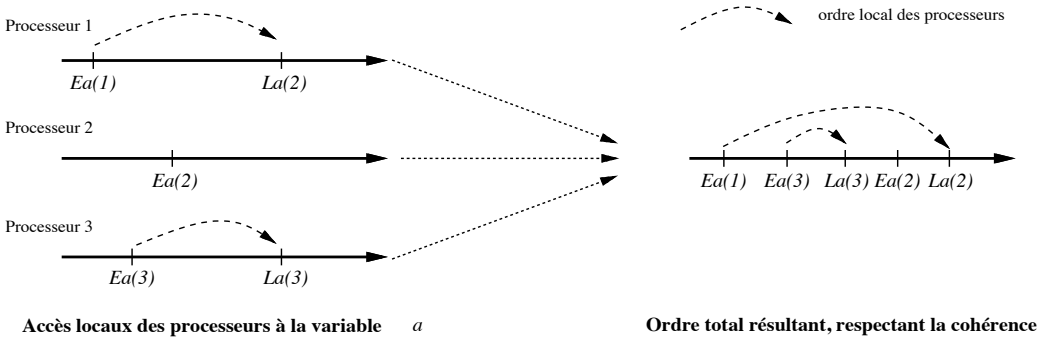
Cette définition implique qu'il existe un ordre physique total des accès à une variable quelconque du système considéré. Dans le cadre d'une architecture multiprocesseur, où les écritures peuvent être retardées et où plusieurs copies de la même donnée existent, la notion de **dernière écriture** reste difficile à définir [Dubois *et al.* 88]. Un accès mémoire n'est pas atomique, c'est-à-dire instantané, mais passe par un certain nombre d'étapes pendant lesquelles il est partiellement résolu. Il arrive donc souvent que des accès mémoire sur une même variable soient réalisés simultanément. Ainsi une lecture dans un cache peut être réalisée pendant qu'un autre processeur est en cours de modification de la même ligne dans un autre cache. Il n'apparaît donc pas clairement de définition de l'instant où l'écriture est considérée comme réalisée. La définition de la cohérence que nous utilisons est donc la suivante.

Définition 2 (Cohérence)
Un ensemble de caches est dit cohérent si pour toute variable, il existe au moins un ordre total des accès des processeurs tel que cet ordre soit compatible avec les ordres locaux des processeurs et que dans cet ordre toute lecture renvoie la valeur de la dernière écriture.

Cette définition est moins contraignante que la définition précédente. Elle n'impose pas d'ordonnancement physique des accès à une variable mais implique que le résultat de l'exécution soit similaire à un ordonnancement total des accès à chaque variable. Pour cela, il est suffisant que l'ensemble des caches de l'architecture observe les modifications de chaque ligne mémoire dans le même ordre [Archibald 87]. Une modification est observée par un cache quand toute lecture ultérieure retourne la nouvelle valeur écrite. Un processeur peut donc

lire une copie locale d'une ligne alors qu'elle est en cours de modification par un autre processeur tant qu'il n'a pas observé cette modification. Dans l'ordre final considéré, la lecture apparaîtra avant l'écriture réalisée par le second processeur. La compatibilité de l'ordre établi avec les ordres des programmes des processeurs se justifie par le fait que les requêtes d'un processeur sur une même ligne sont toujours traitées dans l'ordre spécifié par son programme.

Exemple II.1 (Exemple d'ordre total respectant la cohérence)



II.3.2 Modèles de cohérence

La propriété de cohérence ne doit pas être confondue avec les modèles de cohérence mémoire. Pour un multiprocesseur à mémoire partagée, un modèle de cohérence mémoire est une spécification formelle de la manière dont les accès en lecture et en écriture d'un programme doivent apparaître au programmeur lors d'une exécution [Gharachorloo *et al.* 92]. Plus simplement, il spécifie les ordonnancements des accès légaux, lorsqu'un ensemble de processeurs accède à un ensemble d'emplacements mémoire communs. Un modèle de cohérence ne considère donc pas une seule variable mais l'ensemble des variables accédées lors d'une exécution parallèle.

Parmi les modèles de cohérence mémoire, le plus largement utilisé est la **cohérence séquentielle** [Lamport 78]. Elle impose que l'exécution d'une application parallèle apparaisse comme un entrelacement séquentiel des accès des processeurs de telle sorte que les opérations de chaque processeur apparaissent, dans cette séquence, dans l'ordre spécifié par leur programme. Dans la pratique, le respect de la cohérence séquentielle impose qu'un processeur ne puisse commencer une requête mémoire avant que sa requête précédente ne soit globalement terminée [Dubois *et al.* 88].

D'autres modèles de cohérence mémoire tentent de relâcher les contraintes imposées par la cohérence séquentielle afin d'autoriser plus de parallélisme dans les requêtes d'un processeur

[Dubois *et al.* 88, Gharachorloo *et al.* 90]. Ces modèles sont une des solutions proposées pour masquer les temps de latence des accès mémoire. Il est à noter cependant que tous les modèles de cohérence mémoire, vérifient la propriété minimale de cohérence. Dans un but de simplification, seule la cohérence séquentielle est considérée dans la suite du document. Des modèles de cohérence mémoire plus sophistiqués restent toutefois utilisables.

II.3.3 Protocoles de maintien de la cohérence

L'introduction de mécanismes de réplication automatique des données implique également l'utilisation de mécanismes permettant d'assurer la cohérence des données répliquées. Les architectures multiprocesseurs utilisent habituellement des **protocoles de maintien de la cohérence** qui sont le plus souvent mis en œuvre de façon matérielle. Le rôle de ces protocoles est alors de réguler les permissions de modification des lignes afin d'assurer qu'une modification d'une ligne est observée par l'ensemble des caches avant d'autoriser une autre modification de cette ligne [Archibald 87]. Des accès concurrents en lecture sur différentes copies d'une même ligne restent autorisés.

Un protocole de cohérence définit un certain nombre d'**états des lignes de cache**[3] ainsi qu'un certain nombre de **transitions d'états** causées par les messages transportés par le réseau d'interconnexion (messages de cohérence). Les états des lignes de cache sont utilisés pour identifier les types d'accès autorisés sur une ligne (lecture, écriture, aucun) ainsi que les réponses à donner aux requêtes reçues. Les messages de cohérence sont utilisés pour changer les états des lignes répliquées et maintenir cohérent l'ensemble des copies d'une ligne mémoire.

II.3.3.1 Types de protocoles

Deux types de protocoles de cohérence, correspondant à deux stratégies de mise à jour des données répliquées, peuvent être identifiés [Archibald 87] :

- les **protocoles à diffusion des écritures** où chaque écriture sur une ligne mémoire entraîne la mise à jour de l'ensemble des copies existantes de la ligne.
- les **protocoles à invalidation sur écriture** où la modification, par un cache, d'une ligne répliquée entraîne l'invalidation des copies situées dans d'autres caches. Les nœuds désirant ultérieurement accéder à la ligne doivent alors la recharger.

Des études ont montré que les accès mémoire réalisés dans les applications parallèles favorisent généralement les protocoles à invalidation sur écriture [Weber & Gupta 89]. Les protocoles à diffusion sont avantageux uniquement pour la réduction des latences de lecture

[3] Une ligne de cache représente l'unité d'allocation dans un cache. Les tailles les plus fréquemment utilisées vont de 4 à 64 octets. Dans la suite, ligne mémoire et ligne de cache sont équivalentes.

dans le cas de partage de données de type producteur-consommateur [Eggers & Katz 88]. Dans la suite de ce document, nous ne considérons que des protocoles à invalidation sur écriture. Il est à noter cependant, que certains protocoles dits **adaptatifs** tentent de conserver les avantages des deux techniques en permutant dynamiquement d'un type de protocole à l'autre en fonction des accès mémoire réalisés [Cox & Fowler 93, Nilsson & Stenström 94].

De nombreux protocoles de cohérence ont été proposés en particulier pour des multiprocesseurs à bus. Parmi les plus connus, nous pouvons citer les protocoles BERKELEY [Katz *et al.* 85], WRITE-ONCE [Goodman 83], ILLINOIS [Papamarcos & Patel 84] représentatifs des protocoles à invalidation sur écriture, et les protocoles FIREFLY [Thacker *et al.* 88] et DRAGON [McCreight 84] pour les protocoles à diffusion des écritures.

II.4 Implémentation des protocoles de cohérence

L'implémentation d'un protocole de cohérence dans une architecture laisse place à deux choix de mise en œuvre. La première utilise une diffusion des messages de cohérence et implique que chaque cache de l'architecture puisse **espionner** ces messages. La seconde tente de limiter le trafic du réseau en envoyant sélectivement les messages de cohérence aux caches possédant une copie de la ligne accédée [Stenström 90].

II.4.1 Protocoles à base d'espionnage

Ces protocoles de cohérence sont surtout utilisés dans des architectures fournissant un support efficace pour la diffusion des messages, par exemple un bus. Ils supposent que toute requête peut être espionnée par l'ensemble des caches du système. Leur intérêt provient essentiellement de leur facilité d'implémentation. Seuls, les états des lignes de cache doivent être conservés. Les transitions entre états du protocole de cohérence ne dépendent que de l'état courant d'une ligne et de la transaction espionnée.

Lors d'un défaut, un nœud émet une requête sur le réseau d'interconnexion. Chaque cache espionne la requête et vérifie s'il peut la servir. En cas de lecture, un des caches possédant une copie de la ligne ou éventuellement la mémoire, fournit la ligne au cache demandeur. En cas d'écriture, les caches possédant une copie de la ligne l'invalident. Le nœud propriétaire de la ligne fournit au cache demandeur une copie de la ligne avant d'invalider sa copie. Le cache demandeur devient le nouveau **propriétaire** de la ligne.

II.4.1.1 Utilisation dans le cadre d'architectures extensibles

Bien que les protocoles à base d'espionnage soient souvent associés à des machines à base de bus [Goodman 83, Archibald & Baer 86], certains multiprocesseurs tentent de conserver les avantages qu'ils offrent (absence d'information à conserver) en les utilisant dans des

architectures extensibles. Pour diminuer le coût des diffusions ces architectures limitent alors la portée d'une requête en la diffusant seulement aux parties concernées de l'architecture.

Même si certaines architectures non-hiérarchiques [Goodman & Woest 88], utilisent ce principe, cette solution est généralement implantée à l'aide d'une organisation hiérarchique de l'architecture. Au niveau le plus bas de ces architectures se trouvent les processeurs et leurs caches. Les niveaux supérieurs sont constitués soit de caches, soit de tables ne contenant pas de données mais décrivant le contenu des caches de niveaux inférieurs. Les requêtes (défauts ou invalidations) sont lancées à partir des caches et remontent dans la hiérarchie. La consultation des caches ou des tables de niveau intermédiaire permet d'identifier si la requête doit continuer à remonter ou se trouve limitée à une sous-hiérarchie. Dans le cas des tables, l'accès à une ligne nécessite une descente dans la hiérarchie pour accéder au cache contenant la ligne qui suit alors le chemin inverse pour retourner vers le cache en défaut.

Les architectures décrites dans [Liu 93, Yang *et al.* 92] utilisent une hiérarchies de bus formant un arbre où les niveaux intermédiaires sont constitués de caches. L'architecture DDM [Hagersten *et al.* 92] utilise la même topologie mais avec des tables. L'architecture KSR1 [Frank *et al.* 93] utilise quant à elle une hiérarchie d'anneaux associée à des tables.

Le principal défaut de ces architectures est l'augmentation des temps de latence des défauts de cache, impliquée par la hiérarchie. Dans le cas de l'architecture KSR1, ces temps peuvent atteindre plus de 600 cycles lorsque 2 niveaux d'anneaux sont utilisés [Saavedra *et al.* 93]. L'efficacité de ce type d'architecture reste donc liée à un partitionnement des données permettant de tirer parti de la hiérarchie pour limiter la portée des requêtes. Pour certaines applications, ce partitionnement est difficile et engendre un engorgement au sommet de la hiérarchie.

II.4.2 Protocoles à base de répertoire

Lorsque l'architecture n'est pas hiérarchique, le coût des diffusions devient rapidement prohibitif si le nombre de nœuds est important. Il apparaît alors plus judicieux de limiter la diffusion des messages de cohérence aux seuls nœuds possédant une copie locale de la ligne concernée. Cette idée est utilisée dans les protocoles à base de répertoire [Agarwal & Gupta 88, Weber 93].

II.4.2.1 Principes

Un répertoire est constitué d'une entrée pour chaque ligne mémoire. Chaque entrée est chargée de conserver l'identité du ou des nœuds possédant une copie locale de la ligne correspondante. Avec ce type de protocole, tout défaut de cache passe d'abord par le répertoire. Dans le cas d'un accès en lecture sur une ligne modifiée dans un autre cache, le répertoire fait suivre la requête jusqu'au nœud propriétaire de la ligne. Celui-ci se charge alors d'en transmettre un exemplaire au cache demandeur pendant que l'entrée de répertoire note l'identité

de ce nœud. Dans le cas d'un accès en écriture, le répertoire se charge d'invalider les copies de la ligne avant de transmettre les droits exclusifs au nœud demandeur. Pour assurer la propriété de cohérence, l'entrée de répertoire refuse toute autre requête d'écriture durant toute la durée des invalidations.

Les protocoles à base de répertoire diffèrent par la localisation des entrées de répertoire ainsi que par l'organisation de l'information contenue dans chaque entrée. Nous présentons maintenant ces différentes organisations.

II.4.2.2 Localisation des entrées de répertoire

Il existe essentiellement trois mises en œuvre possibles d'un répertoire qui se différencient par la localisation des entrées de ce répertoire.

Le **répertoire centralisé** est la solution la plus simple et la première qui ait été proposée [Tang 76]. Le répertoire est confié à un seul nœud de l'architecture qui reçoit l'ensemble des requêtes des caches. Un tel répertoire est peu efficace dans le cadre d'une architecture extensible puisqu'il limite le parallélisme des requêtes et crée une contention importante sur le nœud chargé de sa gestion.

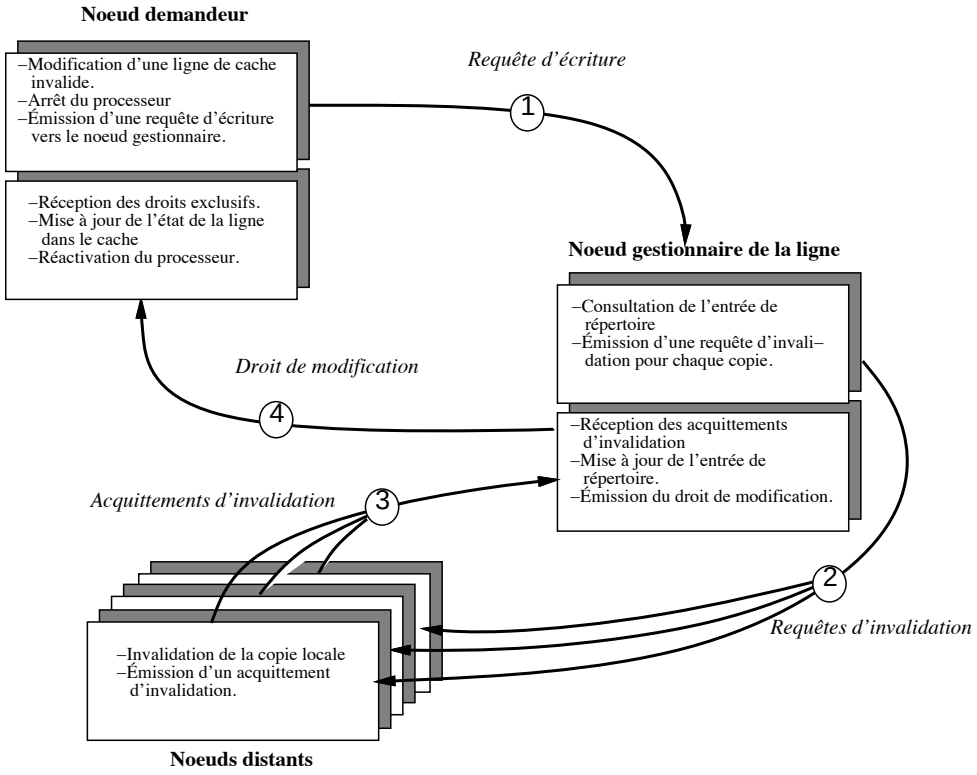


Figure II.3 Traitement d'un défaut d'écriture dans le cas d'un répertoire distribué statiquement

Avec un **répertoire distribué statiquement**, chaque nœud de l'architecture se voit attribuer une partie des entrées du répertoire à l'aide d'une fonction de placement. Les requêtes mémoire ne sont plus envoyées vers un seul nœud mais vers un nœud spécifique à la ligne accédée. Cette approche est utilisée dans la plupart des architectures extensibles à mémoire partagée utilisant des caches maintenus cohérents par matériel. La fonction de placement utilisée est habituellement assez simple. Dans l'architecture DASH [Lenoski *et al.* 92a], les poids forts d'une adresse identifient le nœud chargé de gérer l'entrée de répertoire d'une ligne. La figure II.3 donne l'exemple d'un traitement d'un défaut en écriture sur une ligne partagée par plusieurs processeurs [4]. Cette approche est beaucoup plus efficace que le répertoire centralisé puisqu'elle limite la contention et augmente le parallélisme des requêtes.

La dernière solution est un **répertoire distribué dynamiquement**. L'entrée de répertoire d'une ligne mémoire n'a ici plus de localisation statique, mais se trouve associée au propriétaire courant de la ligne. Chaque nœud dispose ainsi des entrées de répertoire correspondant aux lignes dont il est lui-même le propriétaire. Le problème majeur de cette approche réside dans l'identification du propriétaire courant d'une ligne quand un défaut doit être résolu. La solution la plus simple consiste à utiliser un mécanisme de diffusion qui, de part son coût, s'adapte cependant mal aux architectures extensibles. Une autre solution proposée dans [Li & Hudak 89] consiste à conserver sur chaque nœud l'identité d'un **propriétaire probable** pour chaque ligne de la mémoire partagée. Lors d'un défaut de cache, un nœud s'adresse au propriétaire probable de la ligne qu'il a noté. Si celui-ci est toujours le propriétaire de la ligne, la requête est servie. Dans le cas contraire, il fait suivre la requête vers le propriétaire probable qu'il a lui même noté. L'ensemble des propriétaires probables d'une ligne constitue, dans tous les cas, une chaîne menant au propriétaire courant de la ligne. Une fois l'entrée de répertoire (et donc le propriétaire de la ligne) trouvée, la requête est traitée de façon similaire au cas précédent. L'inconvénient majeur de cette technique est qu'une requête peut avoir besoin de parcourir plusieurs nœuds avant de trouver le propriétaire courant d'une ligne. De plus, le maintien sur chaque nœud d'un propriétaire probable pour chacune des lignes de la mémoire engendre un surcoût mémoire trop important pour qu'elle soit implémentée directement dans une architecture matérielle. Cette solution est généralement utilisée dans le cas de mémoires virtuelles partagées.

II.4.2.3 Organisation des entrées de répertoire

Pour une ligne mémoire, une entrée de répertoire conserve l'identité de l'ensemble des caches possédant une copie de cette ligne. Cette information peut être conservée de différentes façons [Agarwal & Gupta 88, Chaiken *et al.* 90].

Dans un **répertoire à vecteur de bits** chaque entrée de répertoire comprend un bit pour chacun des nœuds de l'architecture (figure II.4) qui indique si la ligne de cache est présente sur le nœud correspondant. Le répertoire a donc une connaissance exacte des processeurs

[4] Une solution similaire aurait été obtenue dans le cas du répertoire centralisé. Seule l'identité du nœud gestionnaire de la ligne change.

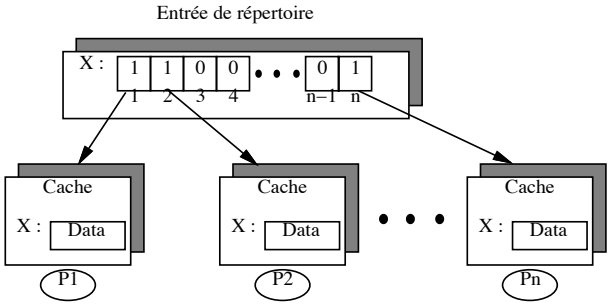


Figure II.4 Répertoire à vecteur de bits

possédant une copie d'une ligne et lors d'une invalidation cette solution produit le nombre minimum de messages de cohérence. Cependant, cette solution limite donc considérablement l'extensibilité d'une architecture, car la quantité de mémoire nécessaire pour le répertoire augmente avec le carré du nombre de nœuds.

Des études sur le comportement des applications parallèles ont montré que la plupart des données partagées ne sont répliquées que dans un petit nombre de nœuds à un instant donné [Weber & Gupta 89]. Les **répertoires limités** tirent parti de cette constatation en limitant le nombre de copies simultanées d'une même ligne de cache. Une entrée de répertoire comprend alors un nombre limité de pointeurs. Chacun de ces pointeurs utilise $\log(P)$ bits (où P est le nombre de processeurs de l'architecture) pour coder l'identité d'un nœud. Les besoins en mémoire pour ce type de répertoire augmente alors en $P \log P$ avec le nombre de processeurs. Cette solution est donc beaucoup plus adéquate que la solution précédente pour une architecture où le nombre de nœud peut être important. Deux alternatives ont

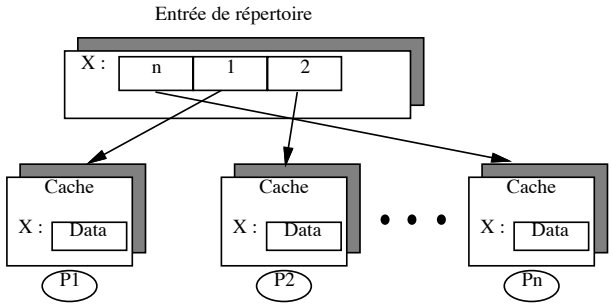


Figure II.5 Répertoire limité

été proposées lorsque le nombre de copies d'une ligne dépasse le nombre de pointeurs d'une entrée. Les **répertoires limités avec diffusion** remplacent les invalidations sélectives par une diffusion globale lorsque cette situation apparaît. Les **répertoires limités sans diffusion** empêchent le dépassement du nombre de copies en invalidant éventuellement une copie avant qu'une nouvelle ne soit créée.

D'autres solutions limitent le nombre de pointeurs utilisés en allouant dynamiquement les pointeurs [Simoni 92], en maintenant une information moins précise sur la localisation des

copies d'une ligne [Agarwal & Gupta 88, Weber 93] ou en émulant par logiciel un vecteur de bits complet lorsqu'un dépassement de capacité survient [Chaiken *et al.* 91]. Le comportement de ces différentes solutions du point de vue du trafic généré sur le réseau est étudié notamment dans [Weber 93].

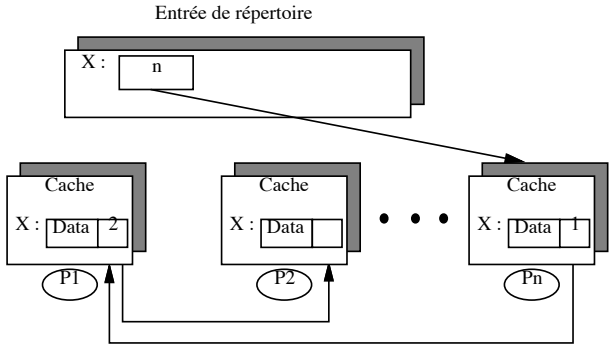


Figure II.6 Répertoire chaîné (simple chaînage)

Les **répertoires chaînés** sont la troisième catégorie de répertoires. Ils corrigent le manque d'extensibilité des répertoires à vecteurs de bits en conservant l'extensibilité des répertoires limités sans restreindre le nombre de copies d'une ligne de cache. Cette organisation émule un répertoire complet en formant une chaîne avec les caches possédant une copie d'une ligne (figure II.6). Les liste peuvent alors utiliser un chaînage simple, tel que dans SDD (Stanford Distributed Directory) [Thapar 92], ou un chaînage double comme dans l'interface SCI [5] (Scalable Coherent Interface) [Gustavson 92] qui simplifie l'éviction d'un cache d'une liste. Même si ces solutions sont les plus extensibles du point de vue de l'occupation mémoire (qui reste fixe), elles ont le défaut de rendre le traitement des invalidations coûteux en nécessitant un parcours séquentiel de la liste pour invalider toutes les copies d'une ligne. Pour limiter le temps d'invalidation, des organisations à base d'arbres ont été proposées [Nilsson & Stenström 93]. La complexité d'une écriture passe alors en $O(\log n)$ alors qu'elle était en $O(n)$ pour une liste.

II.4.2.4 Conclusion

L'implémentation d'un protocole de cohérence à base de répertoire laisse place à un large éventail de possibilités quant à la localisation et à l'organisation des entrées de ce répertoire [Lilja & Yew 93]. Toutes les combinaisons entre ces différentes méthodes de localisation et d'organisation des répertoires, sont envisageables. Même s'il ne ressort pas clairement de solution qui minimise à la fois l'occupation mémoire et les temps d'invalidation, il apparaît cependant que certaines organisations sont à éviter dans le cas d'une machine extensible. Notamment, les répertoires centralisés et les répertoires à vecteurs de bits sont à exclure.

[5] L'interface SCI est utilisée dans l'architecture Exemplar de Convex.

Des solutions à base de répertoires distribués et utilisant des entrées à nombre de pointeurs limités ou à liste chaînée sont beaucoup plus adéquates et permettent une extensibilité bien meilleure des architectures. Une comparaison des performances de ces différents schémas est présentée dans [Chaiken *et al.* 90].

II.5 Architectures COMA

Les architectures COMA représentent l'aboutissement de la politique visant à réduire les latences par l'utilisation de la réplication de données dans des caches locaux. En effet, elles font disparaître les mémoires traditionnelles des nœuds en les remplaçant par des caches de grande dimension de l'espace partagé.

II.5.1 Présentation

Une architecture de type NUMA utilisant des caches locaux maintenus cohérents à l'aide de protocoles de cohérence matériels est une architecture de type CC-NUMA (*Cache Coherent Non Uniform Memory Access machine*). Des exemples de ce type de machines sont l'architecture DASH développée à l'université de Stanford [Lenoski *et al.* 92b] et l'architecture Alewife développée au MIT [Agarwal *et al.* 91]. Dans ces architectures, chaque ligne mémoire possède un emplacement physique fixe sur un des nœuds de l'architecture. Ce nœud appelé **nœud source** sert de refuge à la ligne quand elle est remplacée d'un cache alors qu'elle y était modifiée. Le nœud source est aussi chargé de gérer les entrées de répertoire correspondant aux lignes qu'il possède dans sa mémoire. C'est donc une organisation statiquement distribuée du répertoire que l'on trouve dans ces architectures.

Malgré l'utilisation de caches, les machines CC-NUMA peuvent engendrer un nombre non négligeable d'accès distants dûs à la taille ou à l'associativité limitée de leurs caches. Ces défauts sont appelés **défauts de capacité**. Le reste des défauts de cache sont des **défauts de cohérence** causés par le partage de données entre processeurs. Alors que le nombre des défauts de cohérence est constant et ne peut être réduit que par une réécriture des applications, celui des défauts de capacité peut être réduit soit en utilisant des techniques de placement de pages qui permettent de répliquer dans les mémoires locales certaines pages [D.L. Black 89, R.P. Larowe 92], soit en augmentant la taille des caches locaux.

La première solution implique nécessairement le concours du système d'exploitation. La seconde est utilisée dans les architectures COMA qui, tout en conservant la structure d'une machine NUMA, transforment les mémoires locales des nœuds en de grands caches de l'espace mémoire partagé. Ces mémoires, font donc office de caches de deuxième ou de troisième niveau et offrent un support efficace pour la réplication et la migration automatique de données. Le partitionnement de la mémoire n'est donc plus statique. Chaque ligne peut résider dans n'importe quelle mémoire et au contraire des architectures CC-NUMA, la localisa-

tion d'une ligne est totalement découplée de son adresse. Ce type de mémoire est appelée **Mémoire Attractive** (MA) car elle attire vers elle les lignes que son processeur utilise.

L'implémentation d'une architecture COMA passe par la conversion de la mémoire standard des nœuds en une Mémoire Attractive. Cette transformation est réalisée en ajoutant de l'information à chaque ligne de la mémoire afin de l'identifier et de coder son état. Pour limiter le coût de gestion de ce cache, la taille des lignes d'une MA est souvent supérieure à la taille des lignes rencontrée dans des caches standard[6]. Les mémoires faisant office de caches, l'implémentation d'une machine COMA nécessite aussi l'utilisation d'un protocole de cohérence. Celui-ci est très similaire aux protocoles de cohérence utilisés dans les CC-NUMA. Il peut être implanté en utilisant des techniques à base d'espionnage ou de répertoire.

II.5.2 Gestion des Mémoires Attractives

Suite à la transformation des mémoires en caches, deux problèmes doivent cependant être considérés dans une architecture de type COMA. L'architecture doit tout d'abord **permettre la localisation** d'une ligne en cas de défaut dans une mémoire. Elle doit ensuite **assurer l'existence d'au moins une copie de chaque ligne mémoire** dans l'architecture.

Le problème de localisation d'une ligne peut être résolu par l'utilisation d'une architecture hiérarchique associée à un protocole de cohérence à base d'espionnage. Cette approche est utilisée dans les deux architectures COMA actuellement existantes. L'architecture DDM du Swedish Institut of Computer Science [Hagersten *et al.* 92] utilise une hiérarchie de bus. L'architecture KSR1 de Kendall Square Research [Frank *et al.* 93] utilise une hiérarchie d'anneaux. Les mécanismes de traitement des défauts sont cependant similaires. A chaque racine d'une sous-hiérarchie, un répertoire conserve l'état des lignes mémoire situées dans cette sous-hiérarchie. En cas de défaut en lecture, une requête remonte la hiérarchie jusqu'à ce qu'un répertoire indique qu'il possède une copie valide dans sa sous-hiérarchie. La requête descend alors vers le nœud disposant de la copie et la réponse suit le chemin inverse jusqu'au nœud demandeur. En cas de défaut en écriture, une requête remonte dans la hiérarchie jusqu'à ce qu'un répertoire indique que la copie est exclusive à cette sous-hiérarchie. Le répertoire envoie alors des invalidations à tous les sous-systèmes possédant une copie de la ligne et retourne un acquittement au nœud demandeur. La figure II.7 décrit l'exemple d'un traitement de défaut en lecture sur un bloc modifié, dans l'architecture DDM.

Pour assurer qu'une copie d'une ligne existe toujours dans l'architecture, les architectures COMA utilisent des protocoles de cohérence spécifiques ainsi que des mécanismes d'exportation de données (injection). Le protocole de cohérence de l'architecture DDM utilise trois états, *Invalide*, *Exclusif* et *Partagé*. Lorsque le remplacement d'une ligne *Partagé* est nécessaire dans une MA, une requête remonte dans la hiérarchie. Si cette copie est la dernière copie de la ligne, le répertoire supérieur est dans l'état *Exclusif* et transforme la requête en

[6] 128 octets dans le cas de la KSR1.

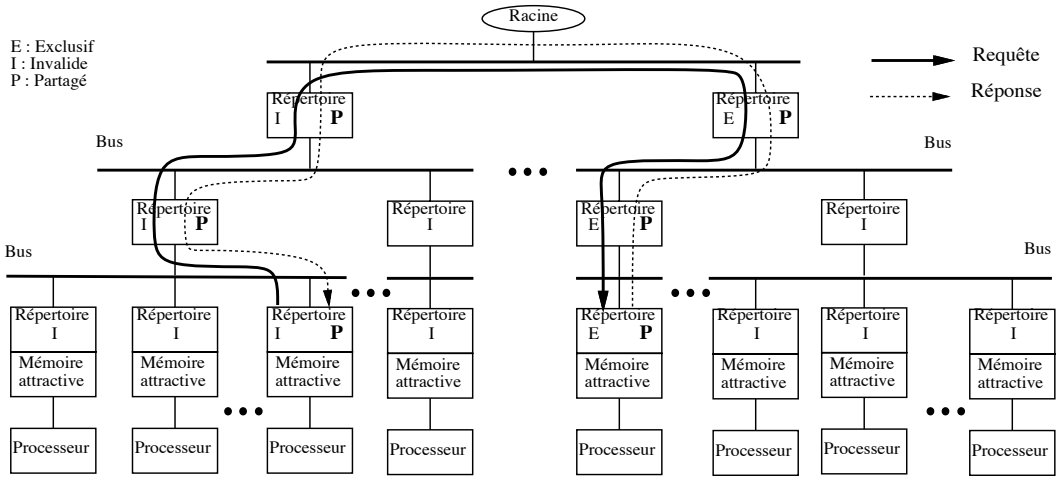


Figure II.7 Exemple de traitement d'un défaut en lecture dans l'architecture DDM

injection dans la sous-hiérarchie qu'il contrôle. Le remplacement d'une ligne marquée *Exclusif* génère automatiquement une injection puisque cette copie est la seule existante. Cette solution nécessite donc un contrôle pour chaque remplacement. Dans l'architecture KSR1, une stratégie différente est utilisée. Au lieu de réaliser une vérification pour chaque copie à remplacer, le protocole de cohérence utilise la notion de propriétaire d'une ligne existant dans le protocole Berkeley. Le propriétaire d'une ligne (possédant la ligne dans l'état *Exclusif* ou *Modifié Partagé*), est responsable de l'injection de la ligne si un remplacement est nécessaire. Les lignes *Partagé* peuvent donc être remplacées sans générer d'injection puisqu'il existe toujours une copie sur le nœud propriétaire de la ligne.

II.5.3 Comparaison

Des études comparatives ont été menées afin de comparer les architectures CC-NUMA aux architectures COMA [Gupta & Weber 92, Hagersten *et al.* 92]. Les résultats sont mitigés et il n'apparaît pas clairement qu'un type de machine soit meilleur que l'autre.

Les avantages des architectures COMA sont multiples. Grâce à la taille de leurs mémoires attractives, elles limitent les défauts de capacité et réduisent ainsi le nombre d'accès distants. Elles réalisent de plus une migration et une réplication automatique des données à un grain plus fin que ne pourrait le réaliser un système d'exploitation et permet notamment de limiter les effets néfastes du faux partage. Le programmeur n'a plus à se soucier du placement des données.

Du point de vue de l'implémentation, les architectures COMA sont plus coûteuses à mettre en œuvre car la transformation des mémoires en MA engendre un surcoût mémoire non négligeable pour coder l'identité et l'état de lignes. Ce surcoût varie selon les hypothèses choisies de 1,5 à 6,5% de la taille de la mémoire attractive [Joe & Hennessy 94, Hagersten

et al. 94]. De plus, ces architectures compliquent les protocoles de cohérence utilisés.

Du point de vue des performances, les résultats montrent que les architectures COMA sont efficaces pour réduire les défauts de capacité. Elles se comportent donc mieux que les architectures CC-NUMA quand les défauts de capacité sont majoritaires. Cependant, du fait de leur organisation hiérarchique, elles souffrent habituellement de temps de traitement des défauts de mémoire élevés. Au contraire, les architectures CC-NUMA limitent les temps de service des défauts mais souffrent quant à elles d'un nombre de défauts de capacité plus élevé, dû à la taille limitée de leurs caches. Elles sont donc plus efficaces lorsque les applications connaissent essentiellement des défauts de cohérence dont le nombre ne peut être réduit et qui sont traités de façon peu efficace par les architecture COMA. Les performances de l'une ou l'autre classe de machines sont donc particulièrement influencées par le comportement de l'application exécutée [Gupta & Weber 92], en particulier la taille de son ensemble de travail et le nombre de défauts de cohérence [Singh *et al.* 93].

Une solution intermédiaire proposée dans [Stenström *et al.* 92] envisage l'implémentation d'une architecture de type COMA non hiérarchique. Le répertoire est organisé de façon similaire à une architecture CC-NUMA, c'est-à-dire statiquement distribué. Cependant les mémoires des nœuds sont des mémoires attractives. Ce type d'architecture combine donc les avantages des COMAs qui réduisent les défauts de capacité, avec ceux des architectures CC-NUMA qui réduisent les temps de traitement des défauts. Les mesures réalisées montrent que les gains en performance d'une architecture COMA, utilisant cette organisation, par rapport à une architecture CC-NUMA, varient alors de 10 à 20% [Stenström *et al.* 92].

D'autres propositions tentent de minimiser le coût d'implémentation d'une architecture de type COMA soit en utilisant des composants programmables permettant de gérer de façon logicielle la mémoire attractive comme dans l'architecture FLASH [J.Kuskin *et al.* 94], soit en utilisant l'unité de gestion mémoire des processeurs pour émuler une mémoire attractive à partir d'une mémoire standard et d'un minimum de matériel spécifique [Hagersten *et al.* 94, Saulsbury *et al.* 95].

II.6 Résumé

Les points importants de ce chapitre sont :

(1) Dans une architecture extensible à mémoire partagée, l'utilisation de mécanismes de réplication de données est indispensable pour limiter les latences des accès mémoire. Les caches assurent cette réplication de façon automatique. Ce sont des composants essentiels de ce type d'architecture.

(2) La réplication de données dans plusieurs caches d'une architecture introduit des problèmes de cohérence de données répliquées. Ces problèmes sont réglés par l'utilisation de protocoles de cohérence.

(3) Il existe deux implémentations possibles d'un protocole de cohérence suivant que les messages de cohérence sont diffusés à tous les nœuds ou sélectivement à ceux qui sont concernés. Dans le cadre d'architectures extensibles, les protocoles à base d'espionnage sont limités aux architectures hiérarchiques. Les protocoles à base de répertoires sont plus indiqués pour les architectures non hiérarchiques mais sont plus complexes à mettre en œuvre.

(4) Les architectures COMA représentent l'aboutissement de la politique d'utilisation de caches locaux. La mémoire locale des nœuds fait alors office de cache de grande dimension de l'espace mémoire partagé. Ces architectures ont pour principal intérêt de réduire le nombre d'accès distants à réaliser en augmentant les possibilités de réplication et de migration automatique des données.

Chapitre III

Réplication de données et récupération arrière

Le nombre élevé de composants des architecture extensibles les rend particulièrement sujettes aux défaillances. Pour leur assurer des temps de fonctionnement suffisamment longs, ces architectures doivent intégrer des mécanismes de tolérance aux fautes leur permettant notamment de tolérer les défaillances des nœuds. Parmi les techniques de tolérance aux fautes utilisables, la récupération arrière est tout à fait adaptée aux contraintes fixées par ces machines.

Ce chapitre débute par une brève introduction aux différentes techniques de traitement d'erreur associées à la tolérance aux fautes. Nous examinons plus particulièrement les techniques de récupération arrière et leurs mises en œuvre au sein de systèmes utilisant une mémoire partagée. Les besoins en réplication de données pour ce type de techniques sont alors clairement identifiés. Une réplication des données courantes permet de conserver une version antérieure de chaque ligne mémoire constituant ainsi un **point de récupération**. Une réplication de ces données de récupération leur assure des propriétés de stabilité qui limitent les hypothèses sur les fautes tolérées.

III.1 Tolérance aux fautes

Nous ne reviendrons pas ici sur la terminologie, relative à la tolérance aux fautes, et employée dans la suite du document. Le lecteur intéressé trouvera celle-ci dans de nombreux ouvrages tels que [Lee & Anderson 90] ou [Laprie 85]. Un résumé du vocabulaire employé est cependant proposé dans l'annexe A.1.

La tolérance aux fautes a pour objectif d'assurer la conception et la réalisation de systèmes sûrs de fonctionnement c'est-à-dire permettant à leurs utilisateurs de placer une confiance justifiée dans les services qu'ils délivrent [Courtois *et al.* 92]. Elle doit donc permettre à un système de continuer à délivrer le service pour lequel il est spécifié malgré le mauvais

fonctionnement d'une partie de ce système.

Après la détection d'une erreur, une technique de tolérance aux fautes doit nécessairement initier une phase de **traitement d'erreur**. Cette phase a pour but est de transformer l'état erroné du système en un état sain afin d'éviter une défaillance du système.

III.1.1 Traitement des erreurs (récupération d'erreurs)

Deux techniques de traitement d'erreurs peuvent être distinguées. Les techniques de **compensation d'erreurs** effectuent un traitement particulier sur l'état du système pour fournir un service conforme en dépit des erreurs qui peuvent affecter cet état. La plupart du temps ces techniques répliquent de façon matérielle ou logicielle les calculs de manière à réaliser un traitement systématique des erreurs même en l'absence d'erreur ; on parle alors de **masquage d'erreur**. Au contraire les techniques fondées sur la **détection suivie du recouvrement d'erreur** attendent la détection d'une erreur pour entreprendre des actions visant à substituer un état exempt d'erreur à l'état erroné. Le choix de l'une ou l'autre de ces techniques se fonde essentiellement sur les objectifs recherchés et les coûts engendrés par leurs mises en œuvre.

III.1.1.1 Masquage d'erreurs

Les techniques de masquage d'erreurs sont fondées sur la comparaison et le vote entre un ensemble de composants répliqués de façon matérielle ou logicielle. Dans le cadre d'architectures massivement parallèles utilisant une mémoire partagée, ces techniques sont difficiles à utiliser.

La réplication active, habituellement utilisée pour des systèmes communicants par messages, masque les erreurs en répliquant les calculs sur plusieurs processeurs. Les exécutions des répliques sont supposées **déterministes** et les messages sont disséminés aux composants d'un groupe de répliques à l'aide de primitives de diffusion atomique [Birman & Joseph 85, Morin 90, Powell 91]. Appliquée à une mémoire partagée, cette technique suppose une synchronisation des répliques à chaque lecture d'une donnée partagée engendrant ainsi une dégradation de performance importante. Elle restreint de plus le nombre de processeurs utilisables pour un calcul. La réplication matérielle de type nMR (n Modular Redundancy) assure quant à elle le masquage d'erreurs par des techniques de vote entre composants matériels répliqués [Bartlett *et al.* 87, Harris 88, Wilson 85]. Dans le cadre d'une architecture massivement parallèle, c'est principalement le coût de son implémentation qui constitue un obstacle à son utilisation.

Pour des architectures extensibles à mémoire partagée, les techniques de masquage d'erreur sont donc peu efficaces ou d'un coût prohibitif. Elles sont de plus inopérantes pour traiter les fautes logicielles ou les fautes temporaires multiples. Ces techniques sont plus

adaptées dans le cas où des contraintes temps réel fortes sont exigées puisqu'elles inhibent la phase de récupération d'erreur.

III.1.1.2 Détection et recouvrement d'erreur

Les contraintes de fiabilité des multiprocesseurs extensibles à mémoire partagée n'étant pas aussi importantes que celles de systèmes de contrôle dont des vies humaines ou des enjeux économiques peuvent dépendre, des techniques moins coûteuses que le masquage d'erreurs peuvent être suffisantes. En effet, les applications s'exécutant sur ces machines ne nécessitent pas une fonctionnalité toujours optimale, elles peuvent accepter des baisses temporaires de puissance pourvu que le système soit toujours en service. Les techniques de détection et de recouvrement d'erreur fournissent ce type de fonctionnalité en impliquant un surcoût important au moment de la détection d'une erreur. Deux classes de méthodes sont alors applicables : la **récupération avant** encore appelée **poursuite** et la **récupération arrière** appelée aussi **reprise**.

La récupération avant vise à reconstruire un état sain à partir d'un état erroné en y apportant les modifications nécessaires pour éliminer les erreurs. Elle s'adresse cependant à des fautes anticipées et reste inutilisable dans le cas de défaillances de processeurs. Son alternative, la récupération arrière est plus adaptée pour les architectures et les fautes considérées.

III.2 Récupération arrière

III.2.1 Principes

La récupération arrière a pour but de substituer, à un état erroné, un état sain du système préalablement sauvegardé. Cette restauration d'état simule un retour dans le temps en ramenant un système dans un état qu'il occupait avant la manifestation d'une faute. Périodiquement un système sauvegarde une image complète de son état sur un support externe, qualifié de **support stable** car résistant lui-même aux défaillances. Cette opération est appelée établissement d'un **point de récupération** (ou **point de reprise**). Un point de récupération est alors défini comme un instant passé auquel il pourra être nécessaire de faire revenir le système en cas de défaillance. Un point de récupération est constitué de toutes les informations nécessaires à sa restauration, ces informations portent le nom de **données de récupération**.

Pour limiter la taille des données de récupération sauvegardées, des mécanismes permettant la suppression de points de récupération devenus inutiles doivent être envisagés. Lorsque un point de récupération est supprimé, le système **valide** celui-ci. Un point de récupération est actif à partir de l'instant où il est établi jusqu'au moment où il est validé. La période

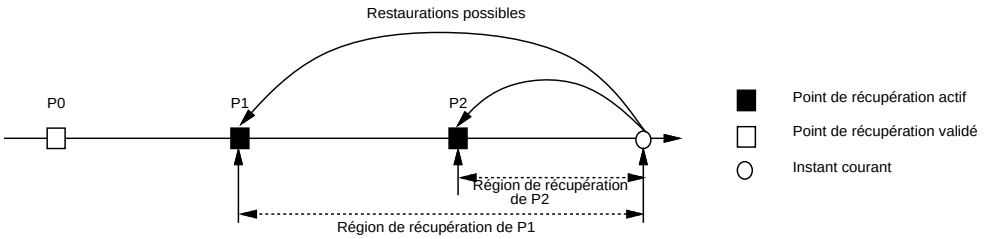


Figure III.1 Récupération arrière

pendant laquelle un point de récupération est actif constitue une **région de récupération**. Si plusieurs points de récupération sont conservés, les régions de récupération peuvent éventuellement être imbriquées. La figure III.1 résume ces définitions.

III.2.2 Récupération arrière et partage de données

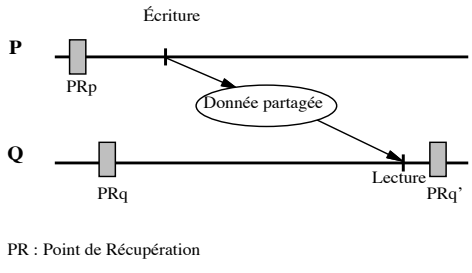
Une technique de récupération arrière est conceptuellement simple à mettre en œuvre dans le cas d'une application séquentielle. Lorsque le système considéré est constitué d'un ensemble de processeurs communicant par l'intermédiaire d'une mémoire partagée, il faut éviter que la restauration du point de récupération d'un processeur[1] n'introduise des incohérences dans les exécutions effectuées sur les autres processeurs. En effet, suite aux communications, les états des processeurs dépendent les uns des autres. Cette inter-dépendance entre les états des processeurs est capturée par la notion **d'état global cohérent** [Chandy & Lamport 85].

III.2.2.1 Présentation du problème

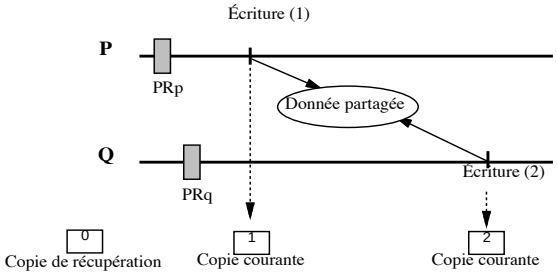
L'état local d'un processeur peut être défini par son état initial suivi de la séquence des accès mémoire qu'il a réalisés. Un état global d'un système de processeurs communicants par une mémoire est constitué de l'ensemble des états locaux des processeurs. Lorsque les processeurs sont situés sur des sites différents et que ces sites disposent de caches locaux, un état global est dit cohérent s'il vérifie la définition de la cohérence donnée au chapitre précédent. Le but d'un mécanisme de récupération arrière est donc d'assurer que l'état global du système reste cohérent, malgré le retour arrière d'un ou de plusieurs processeurs. La figure III.2 décrit les deux situations pouvant introduire une incohérence de l'état global en cas de retour arrière d'un processeur.

Un ensemble de points de récupération actifs dont la restauration ramène le système dans un état global cohérent constitue une **ligne de récupération**. L'objectif d'une technique de récupération arrière est donc de calculer une ligne de récupération permettant, en cas de

[1] Un point de récupération d'un processeur est constitué de l'ensemble des points de récupération des processus qui s'exécutent sur ce processeur.



Le processeur Q lit une donnée préalablement modifiée par le processeur P. Si P effectue un retour arrière après que Q ait lu la donnée alors l'état global résultant n'est pas cohérent car Q a observé une écriture qui n'existe pas. Ce type d'incohérence est à rapprocher de celle définie dans les systèmes communicant par envoi et réception de messages où un état est incohérent si la réception d'une information fait partie de l'état global alors que son émission n'en fait pas partie [Chandy & Lamport 85].



Le processeur Q modifie une donnée partagée précédemment modifiée par le processeur P. Q devient donc le nouveau propriétaire de la donnée et la copie de P est aussitôt invalidée par le protocole de cohérence. Si Q fait un retour arrière, la donnée de récupération (ayant ici pour valeur 0) est restaurée comme valeur courante de la donnée. Cet état global est qualifié d'incohérent puisqu'une lecture de la variable par le processeur P retournerait 0 et non pas 1 comme le spécifie la cohérence (respect de l'ordre du programme de chaque processeur).

Figure III.2 Interactions entre deux processeurs

défaillance, de ramener le système dans un état global cohérent. Pour cela, elle peut forcer certains processeurs à faire un retour arrière.

Il existe deux approches pour la sauvegarde de points de récupération et le calcul d'une ligne de récupération quand une erreur est détectée : les stratégies **optimistes**, encore appelées non planifiées ou asynchrones, et les stratégies **pessimistes**, aussi appelées planifiées ou synchrones.

III.2.2.2 Stratégies optimistes

Avec une stratégie optimiste, les processeurs sauvegardent leurs points de récupération indépendamment les uns des autres. Lorsqu'un retour arrière est nécessaire, une ligne de récupération doit être calculée à partir des points de récupération actifs des processeurs et de l'historique de leurs communications qui doit donc être conservé. Cet enregistrement permet aussi de déterminer les points de récupération actifs et de valider ceux qui n'appartiennent pas à une ligne de récupération.

L'avantage d'une telle approche est d'accélérer l'établissement des points de récupération puisque ce type d'opération ne concerne qu'un seul processeur. Elle présente cependant deux inconvénients. D'une part, la conservation des communications et de plusieurs points de récupération par processeur, sur des supports fiables, engendre un surcoût non négligeable lié à la taille de ce support. D'autre part, une telle stratégie peut souffrir de l'**effet domino**

[Randell 75] dans lequel on assiste à une cascade de retours arrière des processeurs pouvant éventuellement ramener le système dans son état initial

Pour pallier ces deux inconvénients, certains systèmes communicant par échange de messages supposent des exécutions déterministes et utilisent les messages conservés pour limiter le retour arrière seulement aux processeurs en défaillance [Wood 85, Strom & Yemini 85, Goldberg *et al.* 90]. Dans le cas d'une mémoire partagée cette solution nécessite l'enregistrement de tous les accès en lecture réalisés à distance par un processeur [Richard-III & Singhal 93]. Au vue des fréquences des lectures en mémoire partagée, le coût de ces sauvegardes devient rapidement problématique et une telle solution semble donc difficilement envisageable.

III.2.2.3 Stratégies pessimistes

À la différence d'une stratégie optimiste, une stratégie pessimiste coordonne l'établissement des points de récupération des processeurs afin d'assurer que l'ensemble des points de récupération des processeurs forme toujours un état global cohérent. De ce fait, seul le dernier point de récupération d'un processeur doit être conservé, l'effet domino est inhibé et aucune hypothèse sur le comportement déterministe des exécutions n'est nécessaire.

Pour assurer la présence d'une ligne de récupération, des mécanismes de synchronisation d'établissement des points de récupération sont introduits afin de coordonner l'établissement des points de récupération des processeurs de l'architecture [Koo & Toueg 86]. Dans le cas d'une mémoire partagée, trois stratégies de synchronisation ont été proposées.

La **synchronisation globale** force l'ensemble des processeurs d'un système à établir un nouveau point de récupération simultanément [Ahmed *et al.* 90, E. L. Elnozahy & Zwaenepoel 92, Carter *et al.* 93]. Avec une **synchronisation des points de récupération et des communications** la formation de dépendances entre processeurs est empêchée en forçant l'établissement d'un point de récupération par un processeur avant qu'il ne délivre une donnée modifiée à un autre processeur [Ahmed *et al.* 90, Wu *et al.* 90, Wu & Fuchs 90]. La fréquence d'établissement des points de récupération est ici directement dépendante du nombre de communications engendrées par l'application et peut être la cause d'une dégradation de performance importante [Janssens & Fuchs 91, Banâtre *et al.* 93a]. Finalement, la **coordination dynamique** propose une solution intermédiaire où les communications entre les processeurs sont prises en compte de manière à définir dynamiquement l'ensemble des processeurs concernés par l'établissement d'un point de récupération [Joubert 93, Janakiraman & Tamir 94].

Dans la suite du document nous nous limitons aux protocoles de récupération pessimistes qui minimisent la taille des données de récupération et sont les mieux adaptés au cas d'une architecture à mémoire partagée où les communications sont très fréquentes.

III.2.3 Fréquence de sauvegarde des points de récupération

Dans un système implémentant une technique de récupération arrière, la fréquence de sauvegarde des points de récupération intervient pour beaucoup dans la dégradation de performance. Des fréquences élevées augmentent les quantités de données de récupération à traiter et pénalisent l'architecture [Banâtre *et al.* 93a]. Avec une stratégie pessimiste, même dans le cas où les communications forcent l'établissement d'un point de récupération, la décision d'établir un nouveau point de récupération est également liée aux interactions du système avec le monde extérieur.

Le *monde extérieur* est défini ici comme l'ensemble des périphériques avec lesquels les processeurs du système peuvent interagir mais qui ne participent pas à un éventuel retour arrière. Deux problèmes se posent, dans le cas d'une stratégie pessimiste, lorsqu'une requête est envoyée vers le monde extérieur. La requête peut tout d'abord être remise en cause si un retour arrière survient et que les exécutions des processeurs sont différentes (absence de déterminisme). La requête peut également être répliquée si le système est ramené dans un état précédant l'exécution et que celle-ci est réexécutée (voit figure III.2.3).

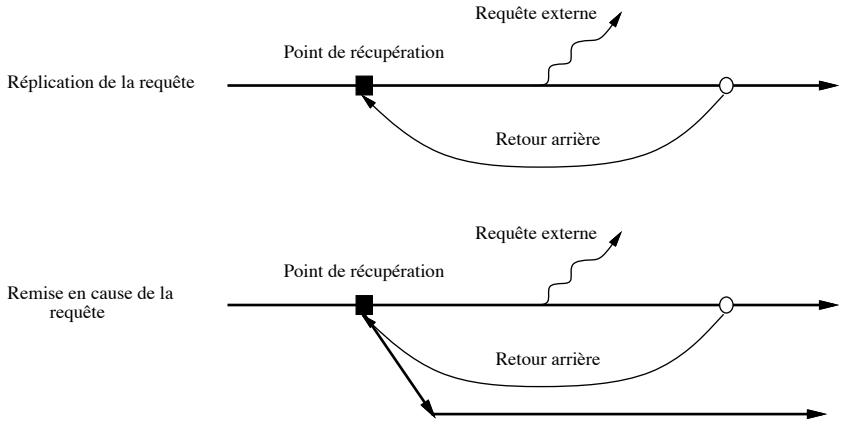


Figure III.3 Interactions avec l'extérieur

Pour éviter ce type de problèmes, il faut s'assurer qu'un retour arrière ne pourra pas répliquer ou remettre en cause l'émission d'une requête vers le monde extérieur. Une solution simple consiste alors à établir un point de récupération à chaque fois qu'une interaction avec le monde extérieur se produit [Joubert 93, Johnson 93]. La fréquence de sauvegarde des points de récupération va donc dépendre essentiellement de la fréquence des interactions avec le monde extérieur. Le système opératoire étant responsable de ces interactions, c'est lui qui se trouve responsable de la décision de sauvegarde d'un nouveau point de récupération.

Toutes les opérations vers le monde extérieur ne nécessitent cependant pas l'établissement d'un nouveau point de récupération. Certaines opérations de consultation (lecture sur un disque) peuvent êtres rejouées ou remises en cause sans poser de problème. D'autres opérations peuvent simplement être rejouées sans toutefois pouvoir être remises en cause. De

telles opérations obligent à sauvegarder un point de récupération. La seule solution, en vue de réduire la fréquence de sauvegarde des points de récupération, est alors de retarder ces requêtes par des techniques de "bufferisation" afin de réaliser un seul accès physique externe pour plusieurs requêtes. La mise en œuvre de telles stratégies ne peut être réalisée qu'au sein du système opératoire de la machine et nécessite un traitement au cas par cas suivant le type d'interaction.

III.2.4 Récupération arrière et architectures extensibles

Dans le cas de l'implémentation de mécanismes de tolérance aux fautes dans une architecture extensible, la récupération arrière est attrayante pour plusieurs raisons. Elle répond tout d'abord aux contraintes fixées par l'utilisation de ce type de machines qui n'ont pas de contraintes temps réel fortes. Elle limite de plus l'introduction de matériel spécifique aux besoins de détection d'erreurs et permet l'utilisation de tous les processeurs pour l'exécution d'une application. Elle ne fait aucune hypothèse sur le déterminisme des exécutions. La récupération arrière évite également la réexécution totale d'une application en cas de faute et tolère notamment les fautes temporaires multiples pourvu que le point de récupération puisse être réinstallé. Finalement, son dernier avantage, et peut être aussi le plus important, est qu'elle permet de tolérer certaines fautes logicielles. Beaucoup d'erreurs causées par des fautes logicielles sont souvent dues à des situations temporaires liées au contexte d'exécution courant, qui ne réapparaissent pas lors d'une phase de réexécution suivant un retour arrière (*Heisenbug* [Gray 86]). Ainsi, les fautes résiduelles d'un système opératoire peuvent entrer dans cette classe de faute et donc être tolérées via une technique de récupération arrière. Des statistiques des systèmes Tandem montrent que la récupération arrière permet de tolérer jusqu'à 99% des fautes logicielles de leur système [Gray 90].

Les techniques de récupération arrière souffrent cependant de plusieurs défauts. La sauvegarde des points de récupération constitue tout d'abord un surcoût pouvant entraîner une dégradation des performances non négligeable. L'implémentation d'une telle technique dans une architecture ayant pour but de fournir une puissance de calcul élevée doit donc s'attacher à minimiser le temps imparti à cette tâche. Ce besoin se trouve exacerbé par des sauvegardes de points de récupération qui peuvent être fréquentes si l'architecture communique avec le monde extérieur. L'introduction de mécanismes de tolérance aux fautes ne doit pas non plus être la cause d'un manque d'extensibilité de l'architecture.

Ces stratégies font ensuite généralement l'hypothèse qu'elles disposent d'un support de stockage stable pour les données de récupération. De tels supports sont habituellement coûteux ou peu efficaces. Il convient donc d'imaginer des supports permettant de limiter le coût de développement tout en assurant un débit élevé pour le traitement des données de récupération. Ceci est particulièrement vrai pour une architecture massivement parallèle dont le coût et les quantités de données de récupération sont élevés.

III.3 Gestion des données de récupération

Indépendamment de la stratégie utilisée pour assurer la présence d'une ligne de récupération, un système à mémoire partagée implémentant une technique de récupération arrière pessimiste, doit avant tout être capable de gérer les données de récupération de l'ensemble des processeurs. Le système doit tout d'abord pourvoir au stockage et à l'identification de ces données. Afin de limiter les hypothèses sur les fautes tolérées, il doit également leur assurer des **propriétés de stabilité**.

III.3.1 Stockage et identification des données de récupération

Tout mécanisme de récupération arrière est fondé sur la possibilité d'établir et de restaurer des points de récupération. Pour cela, le système doit être capable de **sauvegarder** et d'**identifier** les données de récupération. La sauvegarde d'un point de récupération nécessite une **recopie** des données courantes sur un support permettant de les conserver intactes en vue d'un éventuel retour arrière. Cette recopie s'apparente à une réplication des données courantes. Cependant, la copie de récupération est utilisée pour conserver une version antérieure d'une donnée lorsque la copie courante est modifiée. L'identification des données de récupération est indispensable pour pouvoir les réinstaller comme données courantes lors du traitement d'une erreur.

Différentes solutions ont été proposées pour assurer le stockage et l'identification de données de récupération. Elles se distinguent par la manière dont est réalisée cette identification.

III.3.1.1 Schémas utilisant la hiérarchie mémoire

Avec de tels schémas, les données courantes et les données de récupération ne peuvent cohabiter dans un même niveau de la hiérarchie mémoire. Les niveaux les plus bas contiennent les données modifiées (données courantes) et les niveaux les plus hauts, plus éloignés du processeur, contiennent les données de récupération. Les différents niveaux de la hiérarchie mémoire fournissent alors un moyen simple pour sauvegarder et identifier les données de récupération.

Un schéma de ce type est proposé dans les architectures multiprocesseurs Sequoia [Bernstein 88] et CARER [Ahmed *et al.* 90, Wu *et al.* 90]. Les caches et les registres des processeurs contiennent alors les données courantes. Les données de récupération sont stockées dans la mémoire partagée de la machine. Dans une région de récupération, toute modification de donnée doit être réalisée dans un cache sans mise à jour de la mémoire qui détruirait le point de récupération. Cette solution suppose donc l'utilisation de caches à copie retardée [2] dans lesquels les modifications ne sont pas envoyées immédiatement vers la mémoire. La mise à

[2] Write-Back

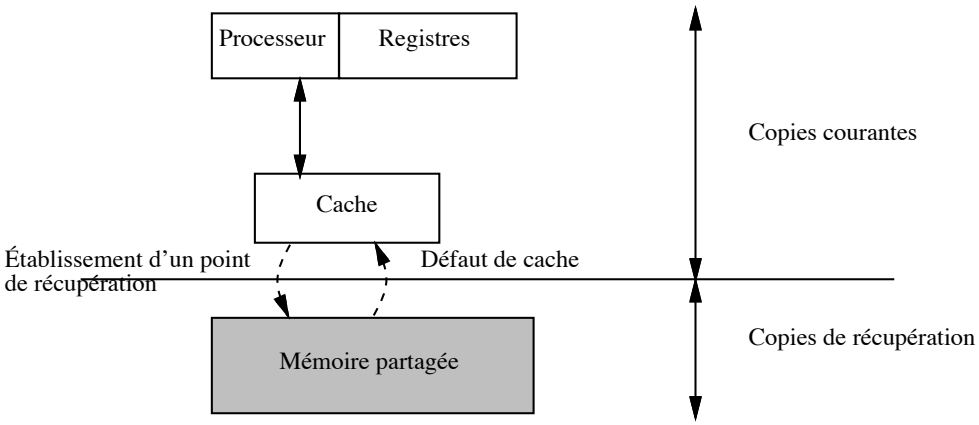


Figure III.4 Exemple de schéma hiérarchique

jour de la mémoire est réalisée lorsqu'une ligne modifiée doit être évacuée d'un cache. Cette opération déclenche alors la prise d'un point de récupération durant laquelle la mémoire est mise à jour en recopiant l'ensemble des lignes modifiées du cache vers la mémoire. Un retour arrière d'un processeur est réalisé en invalidant le cache et les registres du processeur. Les données sont alors fournies par la mémoire. La figure III.4 donne l'exemple d'une telle organisation.

D'autres solutions de ce type utilisent non plus la mémoire mais des disques pour stocker les données de récupération d'un ensemble de nœuds implémentant une mémoire virtuelle partagée recouvrable [E. L. Elnozahy & Zwaenepoel 92].

L'intérêt majeur des schémas utilisant la hiérarchie mémoire réside dans la simplicité de leur mise en œuvre. Ils tirent partie de la réplication déjà existante dans les différents niveaux d'une hiérarchie mémoire pour faciliter la sauvegarde et la restauration des points de récupération. Le défaut de ces schémas réside dans le caractère indécidable du nombre de sauvegardes de points de récupération qu'ils engendrent. Chaque fois qu'une donnée courante modifiée doit être envoyée vers le niveau de la hiérarchie contenant les données de récupération, un nouveau point de récupération doit être établi. Dans le cas de caches, le nombre de ces opérations dépend donc des caractéristiques du cache (taille, associativité...) ainsi que des références mémoires générées par l'application. Il a été montré que cette approche peut sévèrement limiter les performances d'une architecture qui l'utilise [Janssens & Fuchs 91, Banâtre *et al.* 93a].

III.3.1.2 Schémas mixtes avec localisation physique fixe des données de récupération

Au contraire des schémas hiérarchiques, les schémas mixtes autorisent la cohabitation des données de récupération et des données courantes dans un même niveau de la hiérarchie mémoire. Ils corrigent donc le principal défaut des schémas utilisant la hiérarchie mémoire en

rendant la fréquence de sauvegarde des points de récupération indépendante de l'application et des caractéristiques matérielles de la machine.

Avec ce type de schémas, le problème à résoudre est l'identification des données courantes et des données de récupération. Une solution simple consiste à assurer le stockage des données de récupération sur des supports indépendants de ceux utilisés pour les données courantes. L'identification des données de récupération est alors simplement réalisée par leur localisation physique. Ces solutions ont surtout été proposées au niveau des mémoires en vue d'améliorer les performances des techniques de récupération arrière.

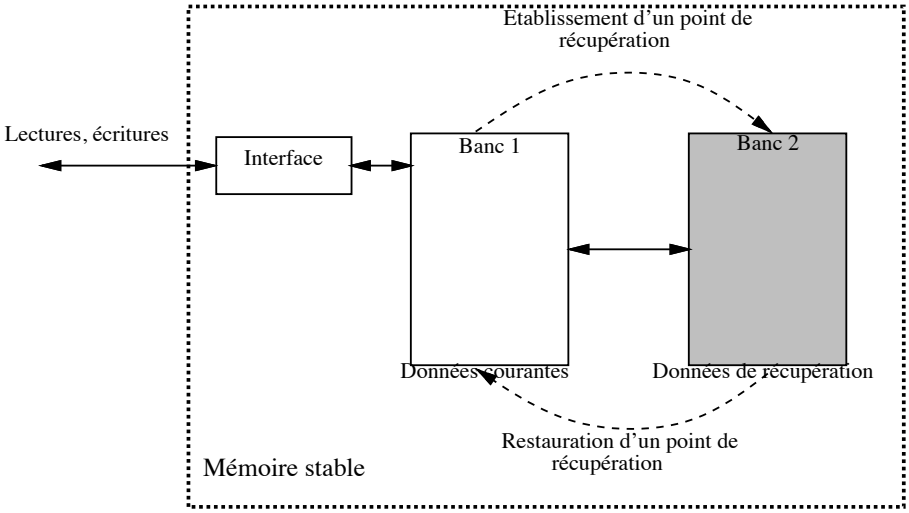


Figure III.5 Synopsis d'une mémoire stable

La mémoire stable est un dispositif matériel proposé pour fournir un support efficace à la récupération arrière. Elle a notamment été utilisée dans une architecture multiprocesseur à bus [Banâtre *et al.* 93a]. Le schéma de la figure III.5 résume l'organisation d'une telle mémoire. Deux bancs mémoire distincts sont utilisés, le premier stocke les données courantes, le second conserve les données de récupération. Lors de l'établissement d'un point de récupération, les lignes courantes modifiées du banc 1 mettent à jour les lignes de récupération correspondantes sur le banc 2. L'opération inverse est réalisée pour la restauration d'un point de récupération. Une telle mémoire est capable de tolérer la défaillance d'un des bancs mémoire sauf dans le cas d'une défaillance du premier banc durant la phase de mise à jour du second banc.

L'utilisation de mémoires stables au sein d'une architecture extensible à mémoire partagée et tolérante aux fautes, est proposée dans l'architecture MEMSY [Din *et al.* 94]. Au contraire de la mémoire stable présentée auparavant, les deux bancs mémoire sont toujours identiques et ne contiennent que des données de récupération[3]. Des mémoires standard sont

[3]La présence de deux copies des données de récupération permet ici d'assurer les propriétés de stabilité définies plus loin.

donc utilisées pour les données courantes. Pour des raisons évidentes de coût, une mémoire stable est associée à plusieurs nœuds de l'architecture. Le traitement d'une défaillance d'un des bancs de la mémoire stable consiste alors à transférer les objets stockés sur celle-ci vers une autre mémoire stable de l'architecture.

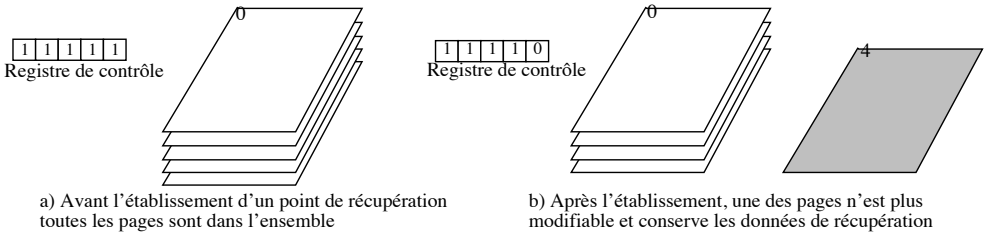


Figure III.6 Mémoire en tranches

Les mémoires *en tranches* [Staknis 89] proposent une organisation où chaque page est en fait constituée d'un ensemble de pages physiques avec la propriété qu'une seule d'entre elles est lisible alors que toutes sont modifiables. L'établissement d'un point de récupération consiste simplement à sortir une page de l'ensemble auquel elle appartenait de manière à ce qu'elle ne soit plus modifiée. La restauration d'un point de récupération consiste à réintroduire la page de récupération dans l'ensemble et à mettre à jour toutes les pages de l'ensemble avec la page de récupération. Ces mémoires ont été plus particulièrement étudiées pour des situations où plusieurs points de récupération sont à conserver [Horning *et al.* 74, Randell 75]. Dans notre cas, des ensembles de deux pages seraient suffisants. Si les différentes tranches correspondent à des bancs mémoire distincts, ces mémoires sont capables de tolérer la perte d'un des bancs.

Les schémas mixtes avec localisation fixe des données de récupération corrigent le défaut des schémas utilisant la hiérarchie mémoire tout en conservant la simplicité d'identification des données de récupération. En séparant physiquement les données de récupération des données courantes sur des supports distincts, ils facilitent la restauration d'état et permettent de tolérer la défaillance des supports utilisés. Ainsi, si un banc de la mémoire stable est défaillant, seules des données courantes ou des données de récupération disparaissent et un état global cohérent existe donc toujours. Cependant, ces schémas sont souvent coûteux car ils utilisent du matériel spécifique nécessitant un coût de développement élevé. De plus, la réplication statique qu'ils utilisent ne leur permet pas de faire un usage optimal du support utilisé pour stocker les données courantes et les données de récupération. Ainsi, dans le cas d'une mémoire stable, la défaillance d'un seul banc inhibe l'utilisation de l'autre.

III.3.1.3 Schémas mixtes sans localisation physique fixe des données de récupération

Comme les schémas précédents, ces schémas autorisent les données courantes et les données de récupération à cohabiter dans un même niveau de la hiérarchie mémoire. Cependant

les données de récupération n'ont pas ici de localisation physique fixe et leur identification est réalisée à l'aide d'informations additionnelles.

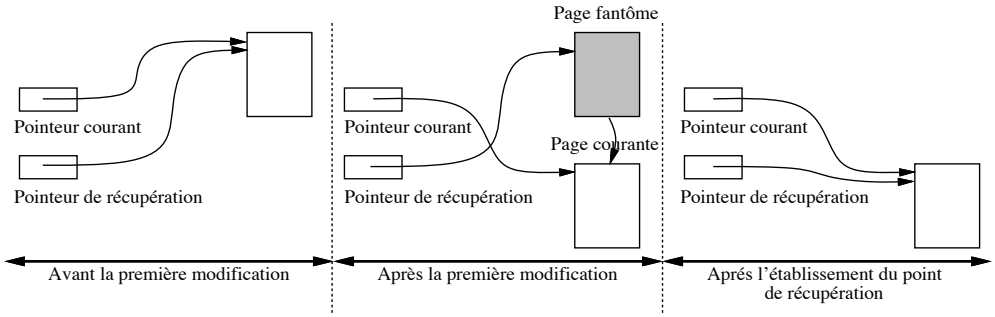


Figure III.7 Technique des pages fantômes (avec mécanisme de copie sur écriture)

Différentes techniques sont envisageables. La technique des **pages fantômes** introduite pour les bases de données [Lorie 77], puis adaptée aux mémoires virtuelles partagées recouvrables [Henskens 93], utilise deux pointeurs identifiant respectivement la version courante et la version de récupération d'une page. Avec la technique des **pages jumelles** [Reuter 80], un seul pointeur sur un emplacement contenant les deux copies d'une page est utilisé. L'identification des pages de récupération est réalisée grâce aux dates de modification de chacune des pages ainsi qu'à la date d'établissement du dernier point de récupération. Des optimisations de ces deux techniques apparaissent dans [Wu & Fuchs 90] pour l'implémentation d'une mémoire virtuelle partagée recouvrable au-dessus d'un réseau de stations de travail ainsi que dans [Bowen & Pradhan 91] où ces deux techniques sont reprises pour l'implémentation d'une mémoire virtuelle recouvrable dans un cadre monoprocesseur.

Ce type de schémas n'impose aucune contrainte sur la fréquence d'établissement des points de récupération. Les données de récupération pouvant être stockées n'importe où. Ils permettent de plus de faire un usage optimal des ressources allouées pour le stockage des données courantes et de récupération. Leurs inconvénients proviennent essentiellement de l'introduction de nouvelles structures de données permettant d'identifier les données de récupération. Ces structures de données fragilisent l'architecture du point de vue des fautes tolérées puisque toute perte d'une partie d'entre elles entraîne inévitablement une défaillance de l'architecture. L'implémentation de tels schémas ne peut donc être réalisée que si ces structures sont conservées sur des supports eux-mêmes tolérants aux fautes ou si l'information perdue peut être recalculée lors de la reprise. Dans la pratique, ils sont souvent accompagnés d'hypothèses simplificatrices sur les fautes tolérées. Ainsi, les techniques des pages fantômes ou des pages jumelles supposent habituellement l'utilisation de disques fiables [Wu *et al.* 90]. Dans le cas de la mémoire virtuelle proposée dans [Bowen & Pradhan 91], la mémoire est supposée fiable et seules les défaillances des processeurs sont considérées. Ces hypothèses simplificatrices permettent dans bien des cas de limiter l'occupation mémoire et d'optimiser l'établissement de points de récupération en utilisant par exemple des techniques de copie sur écriture qui repousse la réplication d'une page de récupération jusqu'au moment de sa

première modification (voir figure III.7).

III.3.2 Stabilité des données de récupération

Une technique de récupération arrière implique toujours que l'ensemble des données de récupération conservé soit accessible après une faute. Pour cela, il est souvent supposé que le système étudié dispose d'un support de stockage **stable**, c'est-à-dire dont le contenu est préservé malgré les défaillances des composants de l'architecture. De tels supports ont pour but d'assurer que toute défaillance, intervenant à n'importe quel instant, peut être gérée par le mécanisme de récupération arrière. La notion de **support de stockage assurant la stabilité** à un ensemble de données a été définie dans [Lampson 81]. Un support stable est alors une abstraction de support persistant ayant la propriété de ne connaître lui-même aucune défaillance et de ne pas être affecté par la défaillance d'un autre composant. Plus précisément, la notion de stabilité d'une donnée peut être définie par trois propriétés.

Définition 3 (Stabilité d'une donnée)

(1) **Accessibilité :** *une donnée stable doit rester accessible quelle que soit la défaillance survenue.*

(2) **Inaltérabilité :** *une donnée stable ne peut être affectée par une défaillance.*

(3) **Atomicité de mise à jour :** *la mise à jour d'une donnée stable est une opération qui, ou bien réussit totalement, ou bien échoue et laisse la donnée dans l'état qu'elle avait avant cette opération.*

Dans la suite du document les propriétés d'accessibilité et d'inaltérabilité seront groupées en une propriété unique appelée **persistance**. La persistance des données de récupération d'un système assure qu'un état global cohérent du système peut toujours être restauré.

Pour un mécanisme de récupération arrière, l'assurance des propriétés de **stabilité des données de récupération** devient indispensable si l'on désire limiter les hypothèses et pouvoir tolérer les fautes de tous les composants, en particulier celles du support utilisé pour le stockage des données de récupération. Dans bien des cas de systèmes tolérants aux fautes, ces propriétés sont assurées grâce à des hypothèses sur les défaillances considérées. Ainsi, dans un multiprocesseur ou un système distribué, un simple disque peut être considéré comme un support assurant la stabilité si seules les défaillances des nœuds de calcul sont prises en compte. La mise en œuvre effective des propriétés de stabilité introduit nécessairement une **réplication** des données concernées.

III.3.2.1 Accessibilité

La propriété d'accessibilité d'une donnée stable est habituellement assurée par des chemins d'accès multiples à cette donnée. Dans l'architecture Sequoia [Bernstein 88] un bus doublé permet d'accéder à la mémoire qui conserve les données de récupération. Dans les multiprocesseurs à base de mémoires stables [Banâtre *et al.* 92, Banâtre *et al.* 93a, Din *et al.* 94], des accès multiples permettent à un processeur valide d'accéder aux données de récupération d'un processeur défaillant. Pour assurer des chemins d'accès multiples à une donnée, une autre solution consiste répliquer cette donnée sur des supports physiques différents. Cette solution est notamment utilisée dans les disques miroirs où les données sont répliquées sur deux disques indépendants.

III.3.2.2 Inaltérabilité

La propriété d'inaltérabilité d'une donnée stable stipule qu'une défaillance dans le système n'affecte pas cette donnée. Dans le cas d'une architecture où le support de stockage utilisé pour conserver des données de façon stable a une probabilité de défaillance d'un ordre de grandeur similaire aux autres composants de l'architecture, il devient nécessaire de considérer aussi les défaillances de ce support. La seule solution permettant d'assurer l'inaltérabilité à des données consiste alors à répliquer les données concernées, sur des supports physiques indépendants vis à vis des défaillances. Le degré de tolérance aux fautes escompté influence le nombre de réplica nécessaires. Avec deux copies, il est possible de tolérer toute défaillance simple du support utilisé. Ce type de réplication de données est fréquemment utilisé dès que la défaillance du support de stockage est prise en compte. Ainsi, les disques miroirs utilisent deux disques contenant les mêmes données.

Dans le cas de données de récupération, la propriété d'inaltérabilité n'est cependant pas toujours nécessaire. Si la défaillance du support de stockage de ces données n'implique pas de retour arrière parce que seules des données de récupération ont été perdues sans endommager l'état global courant du système (cas d'une défaillance simple et d'un schéma mixte avec localisation physique fixe des données de récupération), cette propriété et la réplication qu'elle imposait deviennent inutiles. L'état courant du système peut en effet être utilisé pour recréer un nouveau point de récupération. Au contraire, si cette défaillance nécessite un retour arrière parce que des données courantes ont également été perdues (cas d'un schéma mixte sans localisation fixe des données de récupération), la réplication des données de récupération est alors indispensable.

Très peu de systèmes tolérants aux fautes, concernés par la propriété d'inaltérabilité des données de récupération, l'assurent effectivement. Dans le cas de mémoires virtuelles recouvrables utilisant un disque [Wu *et al.* 90, E. L. Elnozahy & Zwaenepoel 92] ou la mémoire pour stocker les données de récupération, cette propriété est assurée en supposant que le support stockant les données de récupération est fiable et donc non sujet aux défaillances. Cette hypothèse permet de ne conserver qu'un seul exemplaire des copies de récupération

et autorise l'implémentation d'un certain nombre d'optimisations (compteurs de validation, mécanismes de copie sur écriture). Cependant, dans [Wilkinson 93], une mémoire virtuelle distribuée et recouvrable, où données courantes et données de récupération sont stockées dans les mémoires des nœuds du système, assure cette propriété dans le cas de fautes simples en répliquant chaque page de récupération sur deux nœuds.

III.3.2.3 Atomicité des mises à jour

La propriété de mise à jour atomique se justifie par le besoin de tolérer les défaillances également durant la modification de données stables. Dans le cas de la récupération arrière, cette mise à jour a lieu durant la sauvegarde des points de récupération. Même si des hypothèses simplificatrices permettent d'assurer l'inaltérabilité des données de récupération, leur mise à jour atomique reste cependant nécessaire. Dans le cas contraire, une défaillance d'un composant lors de l'établissement d'un point de récupération pourrait laisser celui-ci incomplètement mis à jour et rendrait donc impossible tout retour arrière.

Les techniques permettant d'assurer l'atomicité d'une mise à jour ont surtout été développées pour les systèmes transactionnels [Gray 78]. Généralement ces systèmes utilisent des techniques de journalisation des modifications permettant de défaire ou de refaire des modifications opérées sur les données. Cependant ces techniques supposent l'utilisation de supports assurant eux-mêmes des propriétés de persistance et d'atomicité de mise à jour des journaux.

D'autres approches ne présupposent pas la présence d'un support stable. Elles utilisent une réplication des données pour implémenter un protocole à deux phases [Gray 78, Spector *et al.* 85].

Une telle technique dont les principes apparaissent dans [Lampson 81] est implémentée dans l'architecture Sequoia pour la sauvegarde des points de récupération. L'établissement d'un point de récupération s'effectue par une opération de **vidage** des données modifiées des caches afin de mettre à jour les deux bancs mémoire contenant les données de récupération répliquées. L'atomicité de l'opération est assurée par un protocole à deux phases. Durant la première phase, un vidage met à jour le premier banc mémoire. La seconde phase réalise un autre vidage ayant pour but de mettre à jour le second banc. En cas de défaillance durant le premier vidage, le point de récupération précédent peut être restauré à partir du deuxième banc. En cas de défaillance lors du second vidage, le premier banc est cohérent et contient le nouveau point de récupération. Cette solution assure donc l'atomicité de l'établissement d'un point de reprise en utilisant trois copies d'une même donnée (deux dans les mémoires et une dans un cache). Une technique similaire est utilisée dans l'architecture MEMSY [Din *et al.* 94] pour la mise à jour des mémoires stables.

Une autre solution, assurant l'atomicité, consiste non pas à mettre à jour les données de récupération mais à en créer de nouvelles avant d'invalider les anciennes. Durant la première phase, un nouveau point de récupération **temporaire** tente d'être établi en répliquant les

données courantes sur le support utilisé pour les données de récupération. Si cette première phase termine avec succès, le point de récupération temporaire est transformé en point de récupération **permanent** lors de la seconde phase. L'ancien point de récupération permanent est alors détruit. En cas d'échec, le point de récupération permanent, conservé durant la première phase, permet de réaliser un retour arrière. Ce type de solution est notamment utilisée dans [E. L. Elnozahy & Zwaenepoel 92] pour une mémoire virtuelle distribuée recouvrable. Son coût dépend des hypothèses considérées. Si les points de récupération temporaires et permanents doivent tous deux assurer les propriétés de persistance des données de récupération qu'ils contiennent, il est nécessaire de disposer de quatre copies de récupération lors de la phase de création du point de récupération temporaire (2 copies temporaires, 2 copies permanentes).

III.3.3 Conclusion

L'implémentation réelle d'une stratégie de récupération arrière implique de pouvoir sauvegarder, identifier et répliquer un ensemble de données de récupération.

Lors de l'établissement d'un point de récupération, la sauvegarde des données de récupération est réalisée grâce à une réplication des données courantes. Le choix d'une solution pour le stockage et l'identification des données de récupération dépend alors des objectifs fixés et de la complexité engendrée par leur mise en œuvre. Les schémas hiérarchiques sont simples mais introduisent une dégradation de performance importante. Les schémas mixtes avec localisation statique des données de récupération sont efficaces et limitent les hypothèses sur les fautes tolérées. Ils nécessitent cependant un développement matériel spécifique souvent coûteux et font un usage non optimal des ressources qu'ils utilisent. Finalement, les schémas mixtes sans localisation statique des données de récupération sont moins contraignants mais introduisent habituellement des hypothèses sur le type de fautes tolérées.

Les propriétés de stabilité des données de récupération permettent de traiter tout type de faute intervenant dans le système considéré. Elles ne peuvent être assurées que par des techniques de réplication des données de récupération. Pour des fautes simples, les propriétés d'accessibilité et d'inaltérabilité justifient une réplication des données de récupération seulement dans le cas de schémas mixtes sans localisation fixe des données de récupération. Ces schémas optimisent cependant l'utilisation du support de stockage des données de récupération et limitent de plus le développement matériel. En revanche, la mise en œuvre effective de l'atomicité de mise à jour des données de récupération nécessite toujours une réplication de ces données. Les protocoles de sauvegarde de points de récupération sont alors des protocoles de mise à jour à deux phases.

III.4 Résumé

Les points importants à retenir de ce chapitre sont :

(1) La récupération arrière est une technique de tolérance aux fautes qui répond aux besoins des architectures extensibles à mémoire partagée.

(2) Les stratégies pessimistes de sauvegarde de point de récupération permettent de limiter la taille des données de récupération mais nécessitent une synchronisation des processeurs lors de la sauvegarde des points de récupération.

(3) La conservation des données de récupération nécessite deux types de réplication. Une première pour conserver le point de récupération et continuer l'exécution en cours. Une seconde pour assurer des propriétés de stabilité aux données de récupération.

(4) Le stockage des données de récupération à l'aide de schémas mixtes sans localisation précise des données de récupération permet de limiter la dégradation de performance et le développement matériel. Ils limitent cependant souvent le type de défaillances tolérées.

Deuxième Partie

Un protocole de cohérence étendu

Les mécanismes de réplication permettent d'assurer de bonnes performances à une architecture extensible à mémoire partagée. Parallèlement, une technique de récupération arrière requiert des mécanismes de réplication de données afin de conserver de façon stable un ensemble de données de récupération. Nous examinons dans cette deuxième partie le rapprochement entre les mécanismes de réplication offerts par une architecture COMA et les besoins en réplication d'une stratégie de récupération arrière. La réplication de donnée étant gérée par le protocole de cohérence de l'architecture, nous proposons ensuite un protocole de cohérence étendu qui gère de façon transparente les données courantes et les données de récupération tout en assurant les propriétés de stabilité aux données de récupération.

Chapitre IV

Mécanismes de réplication et retour arrière

Les propriétés de stabilité des données de récupération sont les propriétés minimales qui doivent être vérifiées pour implémenter une technique de récupération arrière sans faire d'hypothèses sur la fiabilité des composants d'une architecture. Ces propriétés impliquent une réplication des données de récupération sur des supports indépendants vis à vis des défaillances. Dans ce chapitre, nous étudions l'adéquation des mécanismes de réplication et de migration de données offerts par les mémoires attractives d'une architecture COMA pour assurer ces propriétés.

IV.1 Rappels des objectifs et hypothèses

Nous rappelons que l'objectif de notre étude est de proposer une solution aux problèmes de disponibilité et de fiabilité d'une architecture extensible à mémoire partagée, qui assure à la fois des critères d'efficacité et de limitation du coût de développement. Le choix d'une technique de récupération arrière est justifié dans le chapitre précédent. Parmi les techniques de traitement des fautes le retour arrière permet de limiter la dégradation de performance et le coût de développement au prix cependant d'interruptions temporaires du service lors des phases de traitement d'erreur et de faute.

Afin de limiter la portée de notre étude, il est cependant nécessaire de fixer un certain nombre d'hypothèses. Comme dans la majorité des techniques de récupération arrière, nous supposons que les nœuds de l'architecture ont un mode de défaillance de type **silence sur défaillance** qui assure le confinement d'erreur et limite donc la complexité de la phase de passivation de faute. L'assurance d'une telle propriété nécessite bien évidemment l'utilisation de matériel supplémentaire [Din *et al.* 94]. La détection de la défaillance d'un nœud est supposée être assurée par des dépassements de temps de réponse lors de communications entre nœuds.

L'unité de défaillance considérée est le nœud et toute défaillance d'un composant interne à un nœud est supposée impliquer une défaillance globale du nœud. Les défaillances des processeurs, des mémoires, ou de tout autre composant d'un nœud doivent ainsi pouvoir être traitées. Seules les défaillances simples sont considérées.

Enfin, nous supposons également que le réseau d'interconnexion est fiable. Cette hypothèse représente la seule hypothèse de fiabilité des composants de l'architecture.

IV.2 Architecture retenue

Dans le cadre d'une architecture extensible à mémoire partagée, l'implémentation d'un mécanisme de récupération arrière laisse place à différentes possibilités quant à la gestion des données de récupération. Beaucoup de propositions de systèmes à mémoire partagée utilisent des disques afin d'assurer le stockage des données de récupération. Cette approche se justifie lorsque les quantités de données de récupération restent peu élevées, c'est-à-dire lorsque les fréquences de sauvegarde de points de récupération sont faibles ou le nombre de nœuds limité. Dans le cas d'une architecture extensible où le nombre de nœuds peut être élevé, l'utilisation de disques, qui limitent les débits de sauvegarde de données de récupération, risque de dégrader fortement les performances de l'architecture. L'utilisation de mémoires semble donc beaucoup plus indiquée. Elle permet de plus une meilleure extensibilité de l'architecture puisque le nombre de mémoires augmente avec le nombre de nœuds.

Au vue des études menées, l'utilisation de schémas hiérarchiques utilisant les mémoires est d'emblée écartée suite à la dégradation de performance qu'ils engendrent. Parmi les schémas mixtes, ceux utilisant une localisation physique fixe des données de récupération nécessitent habituellement un développement matériel important qui les rend eux aussi peu adaptés à des architecture extensibles. Ces schémas font de plus un usage non optimal de la mémoire. Restent donc les schémas mixtes sans localisation physique fixe des données de récupération, où données courantes et données de récupération cohabitent dans les mémoires de l'architecture.

Dans une architecture extensible, l'enjeu est alors de tirer parti de la présence de multiples nœuds disposant de mémoires et ayant la possibilité d'agir les uns à la place des autres, pour implémenter un schéma mixte efficace, limitant le développement matériel et les hypothèses simplificatrices en assurant les propriétés de stabilité aux données de récupération. Les mécanismes proposés doivent de plus permettre une reconfiguration aisée de l'architecture lui permettant de fonctionner en mode dégradé tout en continuant à tolérer les défaillances des nœuds. Le caractère statique des mémoires d'une architecture de type CC-NUMA ne permet pas d'envisager une mise en œuvre qui garantisse à la fois l'efficacité et la simplicité de reconfiguration [Banâtre *et al.* 93b]. Les solutions envisageables utilisent en effet des localisations statiques des données de récupération qui compliquent la reconfiguration. Au contraire le caractère dynamique des mémoires attractives d'une architecture COMA laisse envisager des solutions garantissant à la fois les critères d'efficacité et de simplicité de reconfi-

Besoins pour la récupération arrière		Solutions dans un COMA
Stockage des données de récupération		
Support de sauvegarde des données de récupération	$\rightsquigarrow$	Mémoires attractives
Sauvegarde des points de récupération	$\rightsquigarrow$	Mécanismes d'injection de lignes
Identification des données de récupération	$\rightsquigarrow$	États du protocole de cohérence
Propriété de stabilité		
Inaltérabilité	$\rightsquigarrow$	Réplication des données de récupération sur deux nœuds
Accessibilité	$\rightsquigarrow$	Indépendance des nœuds vis à vis des défaillances
Atomicité de mise à jour	$\rightsquigarrow$	Utilisation d'un protocole à deux phases
Reconfiguration		
Mécanismes de reconfiguration	$\rightsquigarrow$	Absence de localisation physique fixe des lignes

Table IV.1 Mise en œuvre de la récupération arrière et COMA

guration grâce à la propriété d'absence de localisation des lignes mémoire et des mécanismes de réplication de données qu'elles offrent.

IV.3 Architecture COMA et gestion des données de récupération

Grâce à leur architecture et aux mécanismes de gestion de la réplication des données, les architectures COMA présentées au chapitre II fournissent une réponse simple aux problèmes de stockage et de réplication des données de récupération, posés par les schémas mixtes. Ces réponses sont résumées dans la table IV.1.

L'architecture fournit tout d'abord un ensemble de mémoires attractives indépendantes. Pour un nœud donné, l'ensemble des autres mémoires attractives fournit donc un support immédiat pour assurer le stockage de ses données de récupération. Le rôle des mémoires attractives se trouve donc modifié puisqu'elles sont alors utilisées pour conserver des copies courantes ainsi que des copies de récupération de lignes mémoire. La création de ces données de récupération, lors de la sauvegarde de points de récupération, est immédiatement assurée par les mécanismes standard d'exportation de données utilisés pour les injections de lignes. L'absence de localisation fixe des lignes mémoire permet de plus à tout nœud de l'architecture de conserver une copie de récupération de n'importe quelle ligne. La phase d'établissement d'un point de récupération se trouve donc considérablement simplifiée puisque la localisation des copies de récupération n'est pas contrainte. Un nœud peut ainsi utiliser les nœuds les plus proches de lui pour conserver ses données de récupération. L'indépendance des mémoires vis

à vis des défaillances ainsi que leur faculté naturelle de réplication des données fournissent quant à elles une solution simple pour assurer la réplication des données de récupération et leur garantir ainsi les propriétés d'accessibilité et d'inaltérabilité.

Deux problèmes restent cependant non résolus : l'identification des données de récupération et l'assurance de la propriété d'atomicité lors de la sauvegarde d'un nouveau point de récupération. Les lignes des mémoires attractives étant gérées grâce aux états du protocole de cohérence, l'identification des données de récupération peut être réalisée de façon simple en utilisant de nouveaux états chargés d'identifier les lignes correspondant à des copies de récupération. De tels états permettent de gérer de façon transparente les données de récupération sans faire d'hypothèses sur les fautes tolérées. La propriété d'atomicité peut être assurée par un protocole à deux phases similaire à celui utilisé dans [E. L. Elnozahy & Zwaenepoel 92]. L'implémentation d'un tel protocole est alors grandement simplifiée par la possibilité de créer, dans différentes mémoires attractives, autant de répliques d'une ligne qu'il est nécessaire.

IV.4 Avantages de l'approche

Les avantages d'une telle approche sont multiples. Elle permet d'implémenter une technique de récupération arrière de façon simple et efficace tout en limitant les hypothèses sur les fautes tolérées. La simplicité de l'approche réside essentiellement dans l'utilisation de mémoires attractives standard, sans mécanismes particuliers de tolérance aux fautes, qui minimise le développement matériel. L'efficacité est assurée par l'utilisation d'un schéma mixte ne contraignant pas le nombre d'établissements de points de récupération et surtout par l'utilisation des mémoires et du réseau d'interconnexion de l'architecture pour le stockage et l'exportation des données de récupération. L'utilisation des mécanismes de réplication offerts par les architecture COMA, permet d'assurer l'ensemble des propriétés de stabilité et de limiter les hypothèses de fiabilité des composants.

Un autre avantage de cette approche est de pouvoir exploiter la réplication de donnée existante dans les MA pour simplifier la phase d'établissement d'un point de récupération. Ainsi lors de la sauvegarde d'un point de récupération, les lignes courantes déjà répliquées sur plusieurs nœuds de l'architecture ne nécessitent pas de transfert de données pour créer de nouvelles copies de récupération. De plus, les données de récupération étant stockées dans les mémoires, cette solution autorise leur consultation en lecture aussi longtemps qu'elles ne sont pas modifiées dans la région de récupération courante. Pour une ligne, la création de véritables versions de récupération non consultables est ainsi repoussée jusqu'à sa première modification.

Finalement, l'absence de localisation physique fixe des données de récupération simplifie à la fois la sauvegarde des points de récupération et la reconfiguration après défaillance d'un nœud. Lors de la sauvegarde d'un point de récupération, aucune hypothèse n'est faite sur la localisation des données de récupération. Un nœud est libre d'utiliser les nœuds plus proches

de lui pour assurer cette sauvegarde. Lors d'une reconfiguration, l'absence de localisation physique fixe des données de récupération permet de réallouer des lignes mémoire perdues sur n'importe lequel des nœuds valides de l'architecture sans changer leur adresse. Cette propriété assure également une utilisation optimale des mémoires de l'architecture. Les mécanismes de migration automatique de données facilitent de plus la reconfiguration en simplifiant la migration des processus.

IV.5 Approches similaires

Des approches similaires à celle que nous envisageons ont été proposées pour l'introduction de mécanismes de retour arrière dans une mémoire virtuelle partagée dont le comportement reste très proche d'une architecture COMA. L'ensemble des mémoires du système est alors utilisé pour stocker les données courantes et les données de récupération [Wilkinson 93, Brown & Wu 94]. Les solutions proposées se différencient essentiellement par les mécanismes d'identification des données de récupération utilisés. Ces mécanismes d'identification supposent une gestion de la cohérence à base de répertoires et modifient alors les structures de données utilisées par le protocole de cohérence pour assurer l'identification des données de récupération.

L'avantage de la solution que nous présentons dans la suite du document, est de réaliser l'identification des données de récupération au niveau du protocole de cohérence. Ceci nous permet de nous abstraire du type de protocole de cohérence utilisé, et notre solution ne se limite donc pas aux protocoles à base de répertoires mais peut tout aussi bien être utilisée pour des protocoles à base d'espionnage. Elle peut donc être appliquée à un large éventail d'architectures.

IV.6 Résumé

Les points importants de ce chapitre sont :

(1) Les mécanismes de réplication offerts par les mémoires attractives d'une architecture COMA permettent d'assurer l'ensemble des propriétés de stabilité à des données de récupération.

(2) L'utilisation des mémoires attractives pour le stockage des données de récupération assure l'efficacité de la sauvegarde des points de récupération et garantit l'extensibilité de l'approche. Elle facilite de plus la reconfiguration de la mémoire après défaillance.

(3) L'identification et la gestion des données de récupération peuvent être réalisées en modifiant le protocole de cohérence utilisé par les mémoires attractives de l'architecture.

Chapitre V

Protocole de cohérence étendu

Ce chapitre présente un protocole de cohérence étendu intégrant de façon transparente la gestion des données courantes et des données de récupération d'une mémoire partagée, dans une architecture COMA implémentant un mécanisme de récupération arrière. Ce protocole assure le stockage, l'identification et les propriétés de stabilité des données de récupération à partir d'un ensemble de mémoires attractives dont le comportement reste standard. Après une présentation détaillée du protocole et des nouveaux états introduits, les algorithmes de sauvegarde et de restauration de point de récupération sont explicités. Le chapitre se termine par le détail des avantages et des surcoûts engendrés par le protocole de cohérence étendu.

V.1 Rappels

Nous avons montré au chapitre précédent que l'ensemble des mémoires attractives d'une architecture COMA fournit un moyen aisé d'implémenter un mécanisme de récupération arrière. En effet, les mécanismes de réplication de données qui y sont utilisés permettent de conserver et d'assurer à des données de récupération les propriétés d'accessibilité et d'inaltérabilité qu'elles nécessitent. Ainsi une réplication d'une donnée de récupération sur deux mémoires attractives est suffisante pour tolérer la défaillance d'un nœud. Dans ce cadre, données courantes et données de récupération cohabitent dans les MA. La gestion du contenu des MA étant réalisée par le protocole de cohérence, il semble naturel d'y intégrer la gestion des données de récupération. Le protocole de cohérence résultant, appelé protocole de cohérence étendu, doit alors prendre en compte les copies courantes aussi bien que les copies de récupération des lignes de l'espace mémoire partagé. Dans la suite du document, nous appelons **copie courante**, la version courante d'une ligne mémoire, et **copie de récupération** sa version de récupération.

V.2 Protocole de cohérence étendu

Le rôle du protocole de cohérence étendu est double. Comme un protocole de cohérence standard, il doit assurer tout d'abord la cohérence des copies courantes des lignes mémoire. De plus, il doit aussi gérer les copies de récupération en leur assurant notamment les propriétés de stabilité définies précédemment.

D'un autre point de vue, la tâche d'un protocole de cohérence est aussi d'indiquer les copies de lignes mémoire accessibles en consultation ou en modification. Immédiatement après la sauvegarde d'un point de récupération et jusqu'à sa première modification, une copie de récupération contient la version la plus à jour d'une donnée. Le protocole de cohérence doit donc autoriser la consultation et la réplication de cette copie sur les nœuds de l'architecture. Le protocole étendu autorise ce comportement en permettant la lecture, la migration et la réplication d'une copie de récupération non modifiée depuis le dernier point de récupération.

V.2.1 Description du protocole

Le protocole de cohérence étendu peut être vu comme la composition de deux protocoles de cohérence indépendants. Un premier protocole, similaire au protocole standard d'une machine COMA, est utilisé pour les copies courantes. Son rôle se limite à assurer la propriété de cohérence. Un second protocole est utilisé pour les copies de récupération des lignes mémoire. Il a pour but d'assurer une réplication minimum en deux exemplaires de chaque copie de récupération afin que ces copies vérifient les propriétés d'accessibilité et d'inaltérabilité. Ce protocole a aussi pour but d'identifier les copies de récupération afin d'autoriser la lecture et la réplication d'une copie de récupération non modifiée depuis le dernier point de reprise.

V.2.1.1 Protocole gérant les copies courantes (protocole standard)

Le protocole gérant les copies courantes est très similaire au protocole Berkeley. Il utilise les quatre états suivants : *Invalide, Partagé, Modifié Partagé* et *Exclusif*. Les deux derniers états se voient associer la notion de propriétaire de la ligne utilisée dans un COMA pour assurer la présence d'au moins une copie de chaque ligne mémoire. Les transitions entre états sont similaires à celles du protocole Berkeley. Ce protocole est notamment utilisé dans l'architecture KSR1 [Windheiser *et al.* 92].

V.2.1.2 Protocole gérant les copies de récupération

Le protocole gérant les données de récupération utilise également quatre états. Il est toutefois plus simple que le protocole précédent car son rôle se limite à l'autorisation des accès en lecture sur les données de récupération correspondant à des lignes non modifiées depuis le dernier point de récupération. Les quatre états de ce protocole sont les suivants :

- **Invalide (I)** : la ligne n'est accessible ni en lecture ni en écriture. Cette ligne peut être remplacée.
- **Partagé (P)** : la ligne est accessible en lecture. Il existe d'autres copies de cette ligne dans d'autres mémoires. Cette ligne peut être remplacée.
- **Partagé-CK (P-CK)** : cette ligne correspond à une copie de récupération de la donnée. Elle a été créée au dernier point de récupération et n'a pas été modifiée depuis. Elle est accessible en lecture mais pas en écriture. Cette ligne ne peut être remplacée sans être préalablement injectée sur un autre nœud. Il n'existe pas de version plus récente de cette ligne qui sert a la fois de copie courante et de copie de récupération.
- **Invalide-CK (Inv-CK)** : cet état est similaire à l'état invalide dans le sens où la ligne ne peut ni être lue, ni être modifiée. Cependant, cette ligne correspond à une copie de récupération conservée pour un éventuel retour arrière. Une version courante de la ligne existe dans l'architecture. Une telle ligne ne peut être remplacée sans être auparavant injectée sur un autre nœud.

Deux des états de ce protocole (*Invalide* et *Partagé*) sont similaires aux états portant le même nom dans le protocole standard. Ils désignent respectivement des lignes inaccessibles, ou accessibles en lecture uniquement. Dans le protocole étendu, ces deux états sont confondus avec les états équivalents du protocole standard. L'état *Partagé* est donc utilisé pour gérer des répliques de lignes accessibles en lecture uniquement, qu'elles soient des répliques de copies courantes ou des répliques de copies de récupération non modifiées. Les deux autres états du protocole gérant les copies de récupération (*Partagé-CK* et *Invalide-CK*) sont utilisés pour assurer la réplication des copies de récupération afin qu'elles vérifient les propriétés de persistance. Deux copies de ce type sont donc toujours présentes dans l'architecture.

V.2.1.3 Fonctionnement du protocole global

Le protocole global combine la gestion des données de récupération et des données courantes en intégrant les deux protocoles en un protocole unique. Il comprend donc 6 états. Par rapport à un protocole standard, le protocole étendu ajoute aussi deux nouveaux types de transitions entre états, afin de prendre en compte l'établissement et la restauration des points de récupération. Ces transitions sont respectivement nommées *EPR* (Établissement d'un Point de Récupération) et *RPR* (Restauration d'un Point de Récupération). La figure V.1 donne le diagramme de transition du protocole étendu dont le comportement est le suivant :

Succès en lecture ◇ Un succès en lecture sur une copie courante d'une ligne mémoire (*Exclusif*, *Partagé* ou *Modifié Partagé*) est, comme dans un protocole de cohérence standard, servi immédiatement. Si la lecture a lieu sur une copie de récupération dans l'état *Partagé-CK*, le traitement est là aussi immédiat. En effet, une telle copie correspond à la dernière

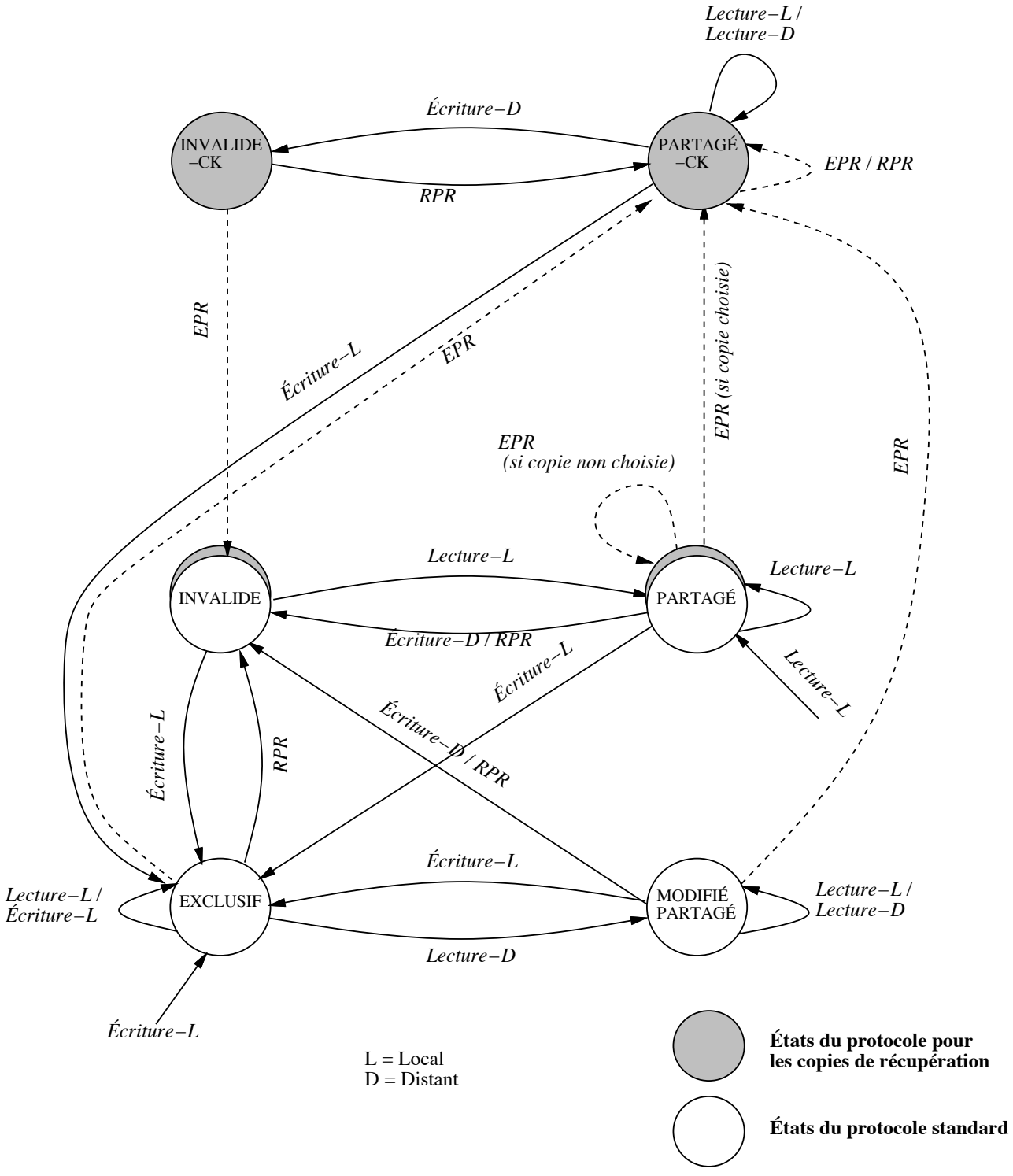


Figure V.1 Protocole de cohérence étendu

version d'une ligne mémoire, elle est donc valide et directement accessible en lecture par le processeur. Au contraire si la copie de récupération se trouve dans l'état *Invalide-CK* un traitement spécifique doit être réalisé. La présence d'une copie *Invalide-CK* indique qu'une modification de la ligne a eu lieu depuis l'établissement du dernier point de récupération. La copie locale n'est donc plus à jour et une copie courante existe dans le système. Les données de récupération étant conservées, il est nécessaire de réaliser une injection de la copie de récupération sur un autre nœud avant de transformer la requête en cours en défaut en lecture.

Défaut en lecture ◇ Un défaut en lecture intervient lors d'un accès à une ligne mémoire non présente dans la mémoire attractive d'un nœud. Si la ligne mémoire concernée a été modifiée depuis le dernier point de récupération, le défaut est traité de façon similaire au protocole standard. Le nœud propriétaire de la ligne est alors contacté et une copie de la ligne est créée localement. Dans le cas d'un défaut en lecture sur une ligne mémoire non modifiée depuis le dernier point de récupération le traitement est similaire mais c'est une des copies *Partagé-CK* qui est utilisée pour servir le défaut. Dans les deux cas, la nouvelle copie créée sur le nœud demandeur est chargée dans l'état *Partagé*.

Succès en écriture ◇ Un succès en écriture intervient lorsqu'un processeur accède en écriture une ligne mémoire que contient déjà sa mémoire attractive. Si la copie est dans l'état *Partagé*, le processeur doit tout d'abord acquérir les droits exclusifs sur la ligne avant de la modifier. Le nœud génère donc une requête d'écriture vers l'extérieur. Suivant l'état de la ligne mémoire, deux cas se présentent :

- Si la ligne a déjà été modifiée depuis le dernier point de récupération, la requête est traitée comme dans le protocole de cohérence standard. L'actuel propriétaire de la ligne (possédant la copie *Modifié Partagé* de la ligne) délivre les droits d'accès exclusifs au nœud demandeur et les autres copies *Partagé* sont invalidées. Dès la réception de ce droit exclusif, le nœud demandeur passe à l'état *Exclusif* et réalise sa modification.
- Si la ligne mémoire n'a pas été modifiée depuis le dernier point de récupération, la requête d'écriture invalide les copies *Partagé* et change les deux copies *Partagé-CK* en copies *Invalide-CK*. Le nœud reçoit les droits exclusifs sur la ligne et réalise sa modification. Le système contient maintenant une copie courante et deux copies de récupération.

Dans le cas d'une écriture sur une copie *Partagé-CK*, le traitement est un peu différent. Cette écriture constitue alors la première modification de la ligne depuis le dernier point de récupération. Afin d'assurer qu'il existe à tout instant deux copies d'une donnée de récupération, une injection de la copie *Partagé-CK* est tout d'abord réalisée. Cette injection a pour but de trouver un nœud permettant d'accueillir la copie de récupération de la ligne. Une fois l'injection réalisée, la requête est traitée comme un défaut en écriture.

Si la copie locale est dans l'état *Invalide-CK*, alors il existe nécessairement une version courante modifiée de la ligne mémoire. Pour assurer la présence de deux copies de récupération une injection de la copie sur un autre nœud est tout d'abord réalisée. Après cette injection, la requête est traitée comme un défaut en écriture.

Défaut en écriture ◇ Lors d'un défaut en écriture, un nœud émet une requête d'écriture vers les autres nœuds de l'architecture. Ici aussi, le traitement de la requête dépend du fait qu'une modification sur la ligne est déjà survenue ou non.

- Dans le cas d'une ligne mémoire modifiée depuis le dernier point de récupération, le traitement est similaire au traitement réalisé dans un protocole de cohérence standard. Avant d'invalider sa propre copie, le propriétaire courant de la ligne en délivre une copie ainsi que les droits de modification. Les copies *Partagé* existantes sont invalidées à la réception de la requête d'écriture. Les deux copies *Invalide-CK* conservent cependant leur état. A la réception de la copie de la ligne et des droits de modification, le nœud demandeur positionne sa copie dans l'état *Exclusif* et réalise sa modification.
- Si aucune modification de la ligne n'a eu lieu depuis le dernier point de récupération, le traitement diffère peu. À la réception de la requête d'écriture, les nœuds possédant les deux copies *Partagé-CK* les changent en copies *Invalide-CK* afin d'assurer la présence de deux copies de récupération. Les copies *Partagé* qui peuvent exister dans le système sont invalidées. À la réception d'une copie de la ligne et des droits exclusifs, délivrés par un des nœuds possédant précédemment une copie *Partagé-CK*, le demandeur crée une copie courante dans l'état *Exclusif* et réalise sa modification.

V.3 Établissement et restauration d'un point de récupération

Dans la présentation du protocole, nous avons simplement traité le cas des accès en lecture et en écriture. Il reste cependant deux transitions du protocole qui ne sont pas explicitées. Ces transitions correspondent à la sauvegarde et à la restauration d'un point de récupération.

V.3.1 Établissement d'un point de récupération

Avec une méthode pessimiste de sauvegarde de points de récupération, le travail à réaliser à chaque nouvel établissement d'un point de récupération dépend de la stratégie de synchronisation utilisée pour assurer la présence d'une ligne de récupération. Nous choisissons ici la stratégie la plus simple, c'est-à-dire celle où l'ensemble des processeurs sauvegardent un point de récupération de façon simultanée.

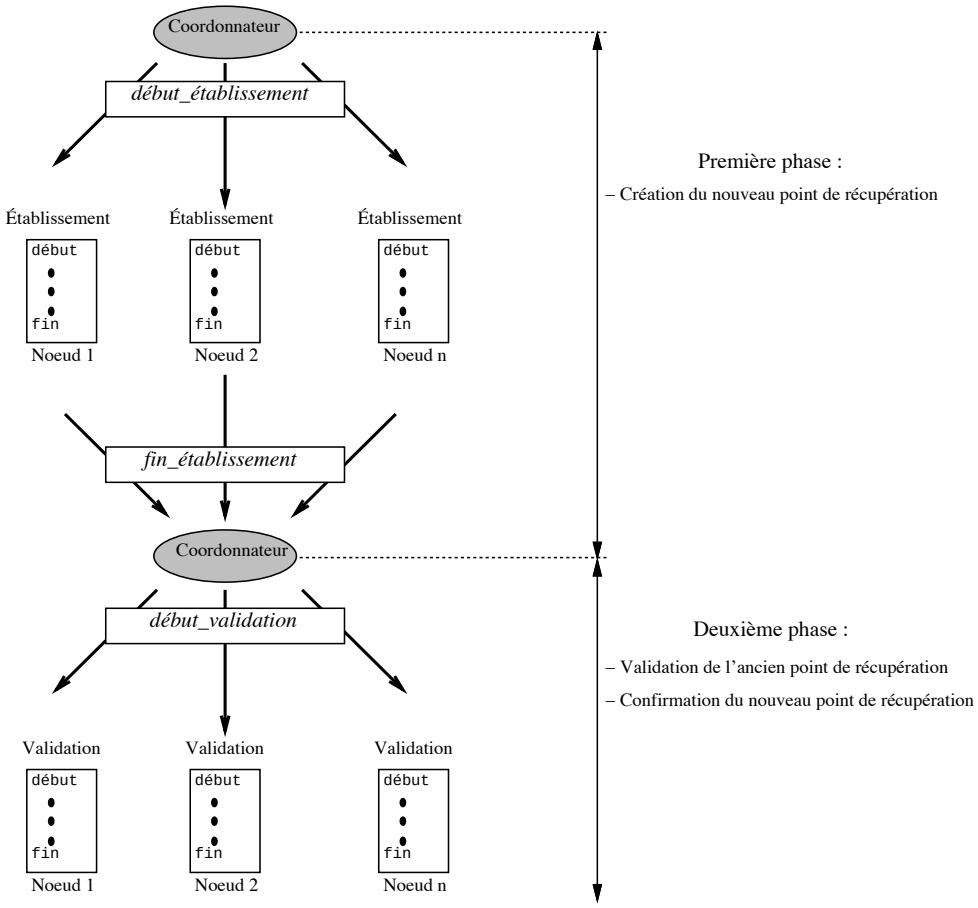


Figure V.2 Algorithme global d'établissement d'un point de récupération

Le travail essentiel à réaliser durant l'établissement d'un point de récupération est la création d'un nouveau point de récupération possédant les propriétés d'accessibilité et d'inaltérabilité introduites au chapitre précédent. Ce nouveau point de récupération se différencie de l'ancien par l'ensemble des lignes mémoires modifiées dans la région de récupération. Sa création nécessite donc seulement la réplication de l'ensemble des copies de lignes modifiées depuis le dernier point de récupération. Les lignes non modifiées, déjà répliquées, peuvent, quant à elles, être conservées dans le nouveau point de récupération. Une telle technique, dite **incrémentale**, limite la quantité de données transférée sur le réseau lors de l'établissement des points de récupération [E. L. Elnozahy & Zwaenepoel 92]. Elle peut facilement être utilisée dans notre cas, grâce aux états du protocole de cohérence.

V.3.1.1 Principes de l'algorithme

La contrainte majeure de l'algorithme d'établissement d'un nouveau point de récupération est d'assurer l'**atomicité** de l'opération. Cette propriété d'atomicité permet de traiter la

défaillance d'un nœud survenant pendant le déroulement de l'algorithme. Afin de garantir cette propriété, l'algorithme utilise un protocole à deux phases similaire à celui décrit au chapitre III et utilisé dans [E. L. Elnozahy & Zwaenepoel 92].

Durant la première phase (phase de création), une nouvelle version persistante d'un point de récupération est construite. Ce nouveau point de récupération est constitué des copies des lignes mémoire non modifiées depuis le dernier point de récupération ainsi que des copies de celles qui ont été modifiées. Au cours de la première phase du protocole, l'ancien point de récupération est conservé de manière à pouvoir être restauré en cas de défaillance. Durant la seconde phase du protocole (phase de validation), l'ancien point de récupération est validé (détruit) et le nouveau confirmé comme point de récupération courant.

V.3.1.2 Description de l'algorithme

La décision de sauvegarde d'un point de récupération est prise par un nœud appelé nœud coordonnateur. Le cas où plusieurs nœuds décident simultanément d'établir un nouveau point de récupération se règle simplement en utilisant l'identité des nœuds comme priorité pour élire un coordonnateur unique. À chacune des phases de l'algorithme correspond une action locale réalisée par chaque nœud de l'architecture. L'algorithme est résumé sur la figure V.2. Il fonctionne de la façon suivante :

(1) Le coordonnateur décide de la sauvegarde d'un nouveau point de récupération. Il prévient les autres nœuds de l'architecture par un message *début_d'établissement* diffusé à tous les nœuds.

(2) À la réception du message *début_d'établissement*, un nœud termine sa requête en cours, s'arrête et exécute l'algorithme local correspondant à la première phase du protocole en vue établir le nouveau point de récupération.

(3) Une fois cette phase terminée, un nœud envoie au coordonnateur un message *fin_d'établissement* indiquant que pour sa part la première phase de l'algorithme est terminée.

(4) Le coordonnateur, après avoir lui même exécuté l'algorithme de création du nouveau point de récupération, collecte les messages *fin_d'établissement* envoyés par les autres nœuds de l'architecture. Si tous les nœuds répondent, la première phase de l'algorithme est terminée. Le nouveau point de récupération est créé et vérifie les propriétés de persistance. La seconde phase débute alors et le coordonnateur diffuse un message *début_validation* aux nœuds de l'architecture, leur indiquant de valider le point de récupération précédent. Si l'un des nœuds ne répond pas, c'est qu'une défaillance est survenue lors de la première phase de l'algorithme. Une restauration du point de récupération précédent doit alors être réalisée.

(5) À la réception d'un message *début_validation*, un nœud exécute l'algorithme de validation du précédent point de récupération. Le nœud reprend son exécution à la fin de cette phase.

Les diffusions de message réalisées dans cet algorithme sont standard. Elles ne font pas d'hypothèses sur l'ordonnancement des messages échangés. Dans le cas d'un réseau supposé fiable, la perte de messages ne peut survenir. Sans cette hypothèse, ces pertes peuvent cependant facilement être traitées. La perte d'un message *début_établissement* ou *fin_établissement* est détectée par le coordonnateur qui ne reçoit pas un des messages *fin_établissement*. Celui-ci peut alors réémettre le message *début-validation* vers le nœud qui n'a pas encore répondu. La perte d'un message *début-validation* implique qu'un des nœuds ne débute pas la seconde phase et reste bloqué en attente du message. Cet état peut être débloqué par un autre nœud ayant repris son calcul et qui désire accéder à la mémoire du nœud en attente. Ce dernier peut alors déduire que la phase de validation a été engagée. Il réalise donc de lui-même la phase de validation avant de servir la requête. Nous détaillons maintenant les phases locales exécutées par chacun des nœuds de l'architecture.

V.3.1.3 Algorithme local : phase de création

```
 ensemble_de_copies contient l'ensemble des nœuds possédant une copie de ligne
Réception du message début_d'établissement

début  { Création du nouveau point de récupération }
 Pour chaque ligne dans la mémoire attractive {
  Cas (ligne.état) dans {
   Exclusif :   { Injection de la ligne dans un autre nœud }
     Injecter_ligne(ligne, Pré-validé)
     ligne.état = Pré-validé
   Modifié Partagé :
     Si (ensemble_de_copies(ligne) ≠ ∅ ) alors
     { Une copie est choisie pour constituer la seconde copie de récupération }
     nœud := Choisir_parmi(ensemble_de_copies(ligne))
    { Requête envoyée vers le nœud choisi }
    changer_a_pré_validé(nœud, ligne)
   sinon
     { Injection de la ligne sur un autre nœud }
     Injecter_ligne(ligne, Pré-validé)
    fsi
    ligne.état = Pré-validé
   Défaut :
    Ne rien faire
  }fcas
 }fpour
 envoyer_message(fin_d'établissement)
fin
```

Figure V.3 Première phase de l'algorithme d'établissement

Durant la première phase de l'algorithme, le nouveau point de récupération est créé de façon à ce qu'il soit persistant. Pour cela, les données courantes modifiées depuis le dernier point de récupération doivent être répliquées sur deux nœuds de l'architecture. Cette réplication est réalisée par les nœuds de l'architecture lorsqu'ils reçoivent le message *début_d'établissement*. Afin d'identifier les copies appartenant au nouveau point de récupération, un nouvel état transitoire *Pré-validé* est utilisé. Cet état indique que la copie appartient au nouveau point de récupération sans toutefois qu'il soit confirmé comme point de récupération effectif. La figure V.3 décrit l'algorithme utilisé localement par les nœuds pour créer le nouveau point de récupération.

L'algorithme de création est simple. Il consiste à parcourir la mémoire afin d'identifier les copies courantes dans l'état *Exclusif* ou *Modifié Partagé*, c'est à dire les copies de lignes mémoire modifiées depuis le dernier point de récupération. Pour répliquer chacune de ces copies sur un autre nœud, un mécanisme similaire aux injections de lignes est utilisé. Un nœud qui accepte l'injection d'une copie marque celle-ci à l'état *Pré-validé*. Dans le cas de copies *Modifié Partagé*, une injection n'est pas toujours nécessaire si une copie *Partagé* existe sur un autre nœud. La réplication déjà existante permet alors d'éviter un transfert de données entre nœuds. Cette optimisation n'est réalisable que si l'architecture offre un répertoire permettant de conserver l'identité des nœuds possédant une copie d'une ligne.

A la fin de cette phase, nous avons :

$$\begin{aligned} \textit{Nouveau point de récupération} &= \{\textit{Partagé-CK}\} \bigcup \{\textit{Pré-validé}\} \text{ et} \\ \textit{Ancien point de récupération} &= \{\textit{Partagé-CK}\} \bigcup \{\textit{Invalide-CK}\}. \end{aligned}$$

Toutes ces copies existent en deux exemplaires dans l'architecture. Les copies de récupération non modifiées depuis le dernier point de récupération, identifiées par leur état *Partagé-CK*, ne nécessitent pas de traitement puisqu'elles n'ont pas été modifiées et qu'elles sont déjà répliquées.

V.3.1.4 Algorithme local : phase de validation

La seconde phase du protocole débute aussitôt que le message *début_validation* est émis par le nœud coordonnateur. Une fois que ce message est émis vers tous les nœuds de l'architecture, le nouveau point de récupération est considéré comme confirmé. Cette phase est purement locale et ne nécessite aucune communication inter-nœud. Elle consiste à parcourir une nouvelle fois la mémoire afin de valider (détruire) les copies appartenant à l'ancien point de récupération et à confirmer celles nouvellement créées.

Dans les mémoires attractives, les copies obsolètes désignées par l'état *Invalide-CK* sont invalidées. Les copies dans l'état *Pre-validé* qui correspondent au nouveau point de récupération sont confirmées et passent dans l'état *Partagé-CK*. Toutes les autres copies sont conservées, soit parce qu'elles correspondent à des copies valides (copies *Partagé*) soit parce qu'elles appartiennent à l'ancien aussi bien qu'au nouveau point de récupération (copies *Partagé-CK*). A la fin de cette seconde phase, une mémoire attractive ne contient que des

```
Réception du message début_validation
début { Validation de l'ancien et confirmation du nouveau point de récupération }
 Pour chaque ligne dans la mémoire attractive {
  Cas (ligne.état) dans {
   Pre-validé :  { Copie appartenant au nouveau point de récupération }
     ligne.état = Partagé_CK
   Invalide-CK :  { Copie appartenant à l'ancien point de récupération }
     ligne.état = Invalide
   Partagé :
   Partagé_CK :
     Ne rien faire   { Copies conservées }
   Défaut :
     Erreur   { Aucun autre état n'est valide pendant la seconde phase }
  } fcas
 } fpour
{ Reprise du calcul }
```

Figure V.4 Algorithme local exécuté lors de la phase de validation

copies à l'état *Partagé* ou *Partagé-CK*.

V.3.1.5 Vérification de l'atomicité

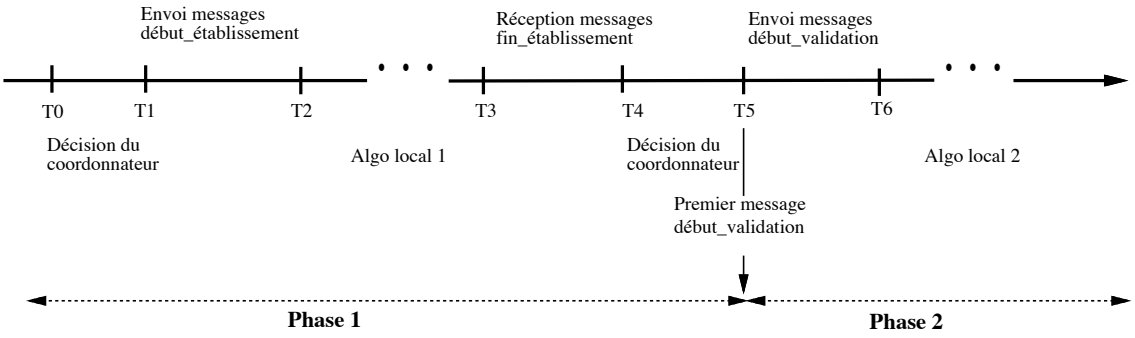


Figure V.5 Découpage de l'algorithme de sauvegarde

Afin de vérifier l'atomicité de l'algorithme d'établissement d'un point de récupération, nous étudions tous les cas de défaillance simple pouvant survenir lors de son exécution. Nous montrons qu'un état global cohérent, correspondant soit au précédent point de récupération, soit au nouveau, peut toujours être restauré.

L'algorithme de sauvegarde d'un point de récupération peut être décomposé en 7 sous-phases délimitées par les dates Ti sur la figure V.5. Soit Td la date de détection d'une erreur survenant sur un des nœuds de l'architecture.

- Si $Td < T4$, les messages *fin_d'établissement* n'ont pas été tous reçus par le coordonnateur et le nouveau point de reprise peut ne pas être complètement établi et persistant. Le point de reprise précédent toujours présent peut être utilisé pour réaliser un retour arrière.

- Si $T4 < Td < T5$, tous les messages *fin_d'établissement* ont été reçus mais le coordonnateur n'a pas eu le temps d'envoyer un message *début_validation*. Deux cas se présentent :

  (1) défaillance du coordonnateur : les autres nœuds peuvent détecter la défaillance du coordonnateur mais ne peuvent savoir si le nouveau point de récupération est totalement établi. Le point de reprise précédent est utilisé pour faire un retour arrière.

  (2) défaillance d'un autre nœud : les deux points de récupération existent et vérifient, avant la défaillance, les propriétés d'inaltérabilité et d'accessibilité. Deux choix sont alors possibles : continuer l'algorithme d'établissement puis traiter la défaillance comme une défaillance pendant le fonctionnement normal, ou stopper l'algorithme et faire immédiatement un retour arrière en restaurant l'ancien point de récupération. Le choix de la première solution est plus risqué puisqu'une nouvelle défaillance durant la fin de l'algorithme entraînerait une défaillance globale de l'architecture. La seconde solution est plus sûre du point de vue des défaillances. Cependant, la durée de la fin de l'établissement du point de récupération est courte et la probabilité d'une défaillance intervenant durant son déroulement est faible.

- Si $T5 < Td < T6$: le nouveau point de récupération est totalement établi et au moins un message *début_validation* a été émis par le coordonnateur. Le réseau étant supposé fiable, au moins un des nœuds va recevoir le message *début_validation*. Ce nœud va débuté la seconde phase et une partie du précédent point de récupération risque d'être détruite. Il faut donc poursuivre cette phase pour que le nouveau point de récupération soit validé. Ici aussi deux cas se présentent :

  (1) défaillance du coordonnateur : s'apercevant de la défaillance du coordonnateur, le nœud ayant reçu le message *début_validation* peut prendre sa place et se charger de diffuser le message aux autres nœuds. Si plusieurs nœuds ont déjà reçu ce message, une élection est nécessaire. La phase de validation est donc poursuivie jusqu'à son terme et la défaillance sera traitée une fois celle-ci terminée.

  (2) défaillance d'un autre nœud : le coordonnateur continue à envoyer les messages et la phase de validation est poursuivie jusqu'à son terme. La faute est traitée à la reprise du calcul et est considérée comme une faute en fonctionnement normal.

- Si $Td > T6$: tous les nœuds ont reçu ou vont recevoir le message *début_validation*. Le nouveau point de récupération est établi et persistant. L'exécution de la phase 2 étant purement locale aux nœuds, elle se poursuit jusqu'à son terme. La faute est traitée une fois celle-ci terminée.

Il apparaît donc que quel que soit l'instant d'occurrence d'une faute durant la phase d'établissement d'un point de récupération, il est toujours possible de restaurer soit le précédent soit le nouveau point de récupération. Cette opération est donc bien atomique ◇.

V.3.2 Restauration d'un point de récupération

V.3.2.1 Principe

Le retour arrière d'un processeur nécessite la restauration de toutes les copies de récupération dont le processeur est actuellement propriétaire des versions courantes. En présence de multiples processeurs communicants, l'algorithme de récupération doit assurer que l'état courant restauré est cohérent en coordonnant le retour arrière des processeurs. Pour cela il peut être amené à forcer certains processeurs à réaliser un retour arrière.

L'utilisation d'une telle stratégie, n'englobant pas obligatoirement tous les processeurs dans un retour arrière, nécessite la sauvegarde d'informations supplémentaires afin d'identifier les lignes mémoires à restaurer [Banâtre *et al.* 93a]. Ce type de solution implique d'une part une modification de l'information conservée par les mémoires attractives, car les copies de récupération d'une ligne doivent conserver l'identité du nœud propriétaire de la copie courante, et d'autre part, un surcoût en fonctionnement normal lié à la gestion de cette information. Compte tenu de la rareté des défaillances, nous lui préférons une stratégie globale qui ne requiert pas d'information autre que l'identification des données de récupération (toutes les données de récupération sont restaurées simultanément). L'utilisation d'une stratégie globale de restauration reste compatible avec toutes les stratégies pessimistes de sauvegarde de points de récupération puisqu'elles garantissent toutes que l'ensemble des points de récupération des processeurs forment une ligne de récupération.

V.3.2.2 Algorithme

Le but de l'algorithme de restauration est de restaurer l'ancien point de récupération comme état global et de détruire l'ensemble des données modifiées dans la région de récupération.

V.3.2.3 Faute temporaire

Nous nous plaçons tout d'abord dans le cas d'une faute temporaire ne demandant pas de reconfiguration de l'architecture. L'algorithme de restauration est déclenché par un nœud après qu'une défaillance ait été détectée. Ce nœud diffuse alors un message indiquant qu'un retour arrière doit être réalisé. Chaque nœud exécute alors l'algorithme décrit sur la figure V.6. Cet algorithme réalise un parcourt séquentiel de la mémoire afin de détruire les copies courantes de lignes modifiées depuis le dernier point de récupération et de restaurer leurs

```
Réception du message début_restauration
début { Restauration du point de récupération courant }
 Pour chaque ligne dans la mémoire attractive {
  Cas (ligne.état) dans {
   Invalide-CK :  { Cette copie appartient au point de récupération }
    ligne.état = Partagé-CK
   Partagé-CK :  { Cette copie appartient au point de récupération }
    Ne rien faire
   Partagé :
   Exclusif :
   Non Modifié Partagé :
   Pré-validé :
    ligne.état = Invalide
   Défaut :
    {Aucun autre état possible }
  } fcas
} fpour
{ Reprise du calcul }
```

Figure V.6 Algorithme local de restauration d'un point de récupération

copies de récupération. La destruction des copies courantes passe par l'invalidation des copies dans l'état *Exclusif*, *Modifié Partagé* et *Partagé*. L'invalidation des copies *Partagé* est nécessaire puisqu'il n'y a aucun moyen de savoir si elles correspondent à des répliques de données courantes ou de données de récupération. Si le retour arrière a lieu durant la première phase d'établissement d'un nouveau point de récupération, les copies *Pré-validées* éventuellement présentes sont elles aussi invalidées car elles correspondent aussi à des copies modifiées depuis le dernier point de récupération. La restauration de l'ancien point de récupération nécessite quant à elle de changer l'état des copies *Invalide-CK* en copies *Partagé-CK*. Les copies *Partagé-CK* restantes sont conservées puisqu'elles appartiennent à la fois à l'ancien et au nouveau point de récupération.

A la fin de cet algorithme, un nœud ne contient que des copies à l'état *Partagé-CK*. L'exécution en cours peut reprendre puisqu'aucune copie n'a été perdue.

V.3.2.4 Faute Permanente

Dans le cas d'une faute permanente, une reconfiguration de la mémoire est aussi nécessaire. Cette reconfiguration a pour but d'assurer la réplication de chaque copie de récupération afin que la machine soit capable de fonctionner en tolérant une nouvelle faute. Durant cette phase, le nœud défaillant doit être identifié puis inhibé afin que la machine puisse continuer à fonctionner en mode dégradé (passivation de la faute).

Pour reconfigurer l'architecture, chaque copie *Partagé-CK* doit être répliquée en deux

exemplaires afin d'assurer les propriétés d'inaltérabilité et d'accessibilité des données de récupération. L'ensemble des copies *Partagé-CK* doit donc être testé pour vérifier que chacune des copies possède une réplique dans l'architecture. La durée de cette phase est très dépendante de l'implémentation du protocole de cohérence. Si le protocole est à base d'espionnage du médium d'interconnexion, une diffusion est nécessaire pour chaque ligne mémoire testée. Un protocole à base de répertoires permet de simplifier la reconfiguration en fournissant des informations sur la localisation des copies *Partagé-CK*. Même avec ce type d'optimisation, la reconfiguration mémoire peut être relativement longue. C'est le prix à payer pour une solution où aucune hypothèse sur la localisation des données de récupération n'est faite. Cette constatation est cependant minimisée par le fait que l'occurrence d'une faute permanente est un événement relativement rare par rapport à une faute temporaire. Cette reconfiguration ne doit donc pas jouer un rôle essentiel dans la dégradation de performance mesurée. Il est cependant certain que ce type de reconfiguration peut poser problème dans le cas où un temps de restauration borné est imposé. Nous reviendrons dans le chapitre suivant sur cette phase de reconfiguration.

V.4 Caractéristiques du protocole

V.4.1 Avantages

Potentiellement, le protocole de cohérence étendu possède un certain nombre d'avantages sur d'autres mises en œuvre de mémoires partagées recouvrables. Ces avantages sont :

- Une limitation des hypothèses de fiabilité des composants de l'architecture. Les processeurs, les caches, les mémoires peuvent être défaillants.
- Un développement matériel limité à l'ajout et à la gestion des nouveaux états du protocole de cohérence. Les mémoires attractives conservent leur fonctionnalité originelle.
- Une efficacité de l'établissement des points de récupération grâce à :
  (1) l'utilisation d'une technique incrémentale de sauvegarde des points de récupération où seules les données modifiées sont transférées [E. L. Elnozahy & Zwaenepoel 92, Banâtre *et al.* 93a].
  (2) l'utilisation potentielle de la réplication de données existante pour limiter les transferts de lignes au moment de la sauvegarde d'un nouveau point de récupération.
  (3) l'utilisation d'un réseau d'interconnexion fournissant une bande passante et un débit élevés qui assurent un temps de transfert de données faible pour l'établissement des points de récupération.
- Une faible perturbation des mémoires attractives de l'architecture en autorisant la consultation en lecture et la réplication des copies de récupération aussi longtemps qu'elles ne sont pas modifiées depuis le dernier point de récupération.

- Une simplification de l'identification des données de récupération grâce aux nouveaux états introduits par le protocole de cohérence étendu. Cette propriété laisse présager d'une bonne portabilité de l'approche.
- Un usage optimal de l'ensemble des mémoires de l'architecture. La défaillance d'une mémoire n'engendre la perte que de cette mémoire. Les données qui y étaient stockées se répartissent sur l'ensemble des mémoires encore valides de l'architecture.
- Une reconfiguration de l'architecture facilitée par l'absence de localisation physique fixe des lignes mémoire.

L'annexe 2 décrit une vérification du protocole de cohérence qui montre, sous certaines hypothèses, que le protocole assure l'ensemble des propriétés de cohérence et de stabilité.

V.4.2 Surcoûts engendrés par le protocole

Le fonctionnement du protocole de cohérence étendu est très similaire à un protocole standard. Lors d'accès en lecture ou en écriture, les actions se limitent à l'envoi d'une requête vers les autres nœuds. Ce sont principalement les actions entreprises par les nœuds qui sont modifiées. Cependant, le protocole engendre un certain nombre de nouvelles opérations à réaliser ainsi qu'une occupation mémoire plus importante pour sauvegarder les données de récupération.

V.4.2.1 Nouvelles injections

Dans le protocole tel qu'il est décrit ici, un nœud ne peut posséder, à un instant donné, qu'une seule copie (courante ou de récupération) d'une ligne mémoire. Cette contrainte provient essentiellement de l'associativité limitée des mémoires attractives. De nouvelles injections sont donc introduites par le protocole de cohérence étendu.

Dans une architecture COMA traditionnelle, les injections ne sont utilisées que pour des copies dans l'état *Exclusif* ou *Modifié Partagé*. Elles ont pour but d'assurer la présence d'une copie de chaque ligne mémoire et n'interviennent que lorsqu'un manque de place nécessite l'évacuation d'une ligne d'une mémoire attractive.

Le protocole de cohérence étendu introduit cinq nouveaux cas d'injection liés à la présence des données de récupération. Deux d'entre eux sont générés par le manque de place et le besoin de conserver les copies de récupération. Les trois autres correspondent à des cas où un processeur désire accéder à une ligne alors qu'il possède déjà une copie de récupération de celle-ci dans sa MA. Toutes ces nouvelles injections ont pour but d'assurer la présence de deux copies de récupération, de chaque ligne, dans l'architecture. Les cinq nouveaux cas d'injections sont résumés dans la table V.1. Les injections causées par un accès en lecture ou en écriture sont immédiatement suivies par le traitement du défaut correspondant.

Cause	État de la copie locale	Actions
Remplacement	*Partagé-CK*	Injection sur un autre nœud
Remplacement	*Invalide-CK*	Injection sur un autre nœud
Accès en lecture	*Invalide-CK*	Injection + défaut en lecture
Accès en écriture	*Invalide-CK*	Injection + défaut en écriture
Accès en écriture	*Partagé-CK*	Injection + défaut en écriture

Table V.1 Nouveaux cas d'injection introduits par le protocole étendu

Au contraire des données courantes qui peuvent être manipulées sans précaution particulière, nous verrons dans le chapitre suivant que l'implantation de ces mécanismes d'injection dans une architecture est complexe car les propriétés de stabilité d'une copie de récupération doivent être vérifiées à tout moment, y compris pendant une injection.

V.4.2.2 Surcoût mémoire

La sauvegarde des copies de récupération dans les mémoires attractives de l'architecture engendre bien évidemment une occupation mémoire plus importante. Cette augmentation de l'occupation mémoire a pour effet d'augmenter le taux de défauts des mémoires attractives. Les défauts de capacité et les défauts de conflit [Hill & Smith 89] sont les deux catégories de défauts concernées par cet effet.

Pour une ligne mémoire donnée, le nombre de copies présentes dans l'architecture varie au cours du temps. Le diagramme de la figure V.7 donne, en fonction de l'instant, le nombre minimum de copies nécessaires pour une ligne mémoire.

Après la première modification d'une ligne depuis le dernier point de récupération, au moins trois copies de cette ligne existent dans l'architecture. Une copie *Exclusif* ou *Modifié Partagé*, puisque la ligne est modifiée, ainsi que deux copies de récupération de type *Invalide-CK*. Pendant la première phase de l'établissement d'un point de récupération, c'est un minimum de quatre copies qui sont nécessaires pour chaque ligne mémoire. Ce nombre est cependant transitoire car la seconde phase du protocole détruit les copies *Invalide-CK* appartenant à l'ancien point de récupération. A la fin de la seconde phase du protocole de sauvegarde d'un point de récupération, le nombre minimum de copies redescend à deux, correspondant aux deux copies *Partagé-CK*. Dans chacun de ces cas, le nombre donné correspond au nombre minimum de copies nécessaires, d'autres copies *Partagé* peuvent également exister.

Par rapport à une architecture standard, ce nombre minimum de copies ne reflète cependant pas réellement le surcoût mémoire engendré par le protocole de cohérence étendu. Les lignes mémoire partagées souvent allouées sur différentes mémoires de l'architecture peuvent bénéficier de cette allocation de multiple copies pour limiter le surcoût mémoire. Ces lignes peuvent en effet utiliser des copies *Invalide* pour conserver les copies de récupération.

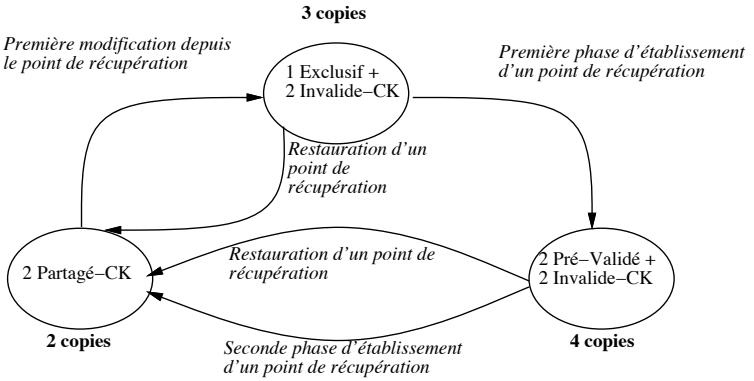


Figure V.7 Nombre minimum de copies d'une ligne mémoire

Dans le cas de lignes privées, accédées par un seul processeur, le nombre de copies minimum représente le surcoût mémoire réel puisqu'aucune autre copie n'est allouée dans la version standard de l'architecture.

V.5 Résumé

Les caractéristiques du protocole de cohérence étendu sont les suivantes :

(1) Il intègre de façon transparente la gestion des données courantes et des données de récupération d'une stratégie de récupération arrière.

(2) Il assure l'ensemble des propriétés de stabilité aux données de récupération en utilisant les mécanismes standard de gestion de la réplication et permet ainsi de limiter le développement de matériel et les hypothèses sur les fautes tolérées.

(3) Il minimise la dégradation de performance en assurant une sauvegarde des points de récupération incrémentale, en autorisant l'accès aux données de récupération non modifiées et en permettant l'utilisation de la réplication de données lors de la sauvegarde des points de récupération.

(4) Il engendre un certain nombre de surcoûts qui proviennent de la sauvegarde des points de récupération, de la création de nouveaux cas d'injection de lignes et d'une occupation mémoire plus importante.

Troisième Partie

Éléments de mise en œuvre et évaluation

Le protocole de cohérence proposé peut être utilisé dans n'importe quel type d'architecture COMA. Nous avons choisi d'étudier ce protocole au sein d'une architecture utilisant un protocole à base de répertoire. Ce type d'architecture laisse en effet présager d'une meilleure extensibilité ainsi que d'une meilleure fiabilité qu'une architecture utilisant un protocole à base d'espionnage et une organisation hiérarchique. Ce choix complique cependant le travail de reconfiguration nécessaire après la défaillance d'un nœud. Nous présentons dans un premier temps les problèmes d'intégration dans l'architecture considérée. Nous proposons ensuite une évaluation par simulation de cette architecture. Nous terminons cette partie par une présentation de différentes optimisations ainsi que de certains problèmes qui n'ont pas été étudiés.

Chapitre VI

Intégration du protocole dans une architecture de type COMA

Au chapitre précédent, nous avons présenté un protocole de cohérence étendu, gérant les données courantes et les données de récupération d'une technique de récupération arrière. Dans ce chapitre nous examinons comment ce protocole de cohérence peut être intégré au sein d'une architecture COMA réaliste. Après avoir présenté l'architecture considérée, nous analysons les modifications à lui apporter pour implémenter le protocole de cohérence étendu de façon efficace. Nous nous intéressons ensuite au traitement des injections de lignes. Le chapitre se termine par une description du travail à réaliser lors d'une phase de traitement de fautes.

VI.1 Architecture considérée

Une architecture COMA peut être implémentée soit en utilisant un protocole de cohérence à base d'espionnage [Hagersten *et al.* 92, KSR 92], soit en utilisant un protocole de cohérence à base de répertoire [Stenström *et al.* 92, Joe & Hennessy 94]. Les architectures utilisant un protocole à base d'espionnage nécessitent une organisation hiérarchique qui les fragilise du point de vue des fautes tolérées. Nous lui préférons une architecture non-hiérarchique utilisant un protocole à base de répertoire. Une évaluation du protocole de cohérence étendu dans une architecture implémentant le protocole par espionnage est réalisée dans [Gefflaut *et al.* 94].

La figure VI.1 représente l'architecture considérée. Elle s'inspire de l'architecture COMA-F[1] présentée dans [Stenström *et al.* 92]. Elle comprend un ensemble de nœuds de calcul reliés par un réseau d'interconnexion. Chaque nœud dispose d'une interface réseau lui permettant de recevoir et d'envoyer des requêtes ou des lignes mémoire. Le réseau d'interconnexion choisi est une grille à deux dimensions composée de deux sous-grilles servant respectivement

[1] "Flat COMA", c'est-à-dire COMA non hiérarchique.

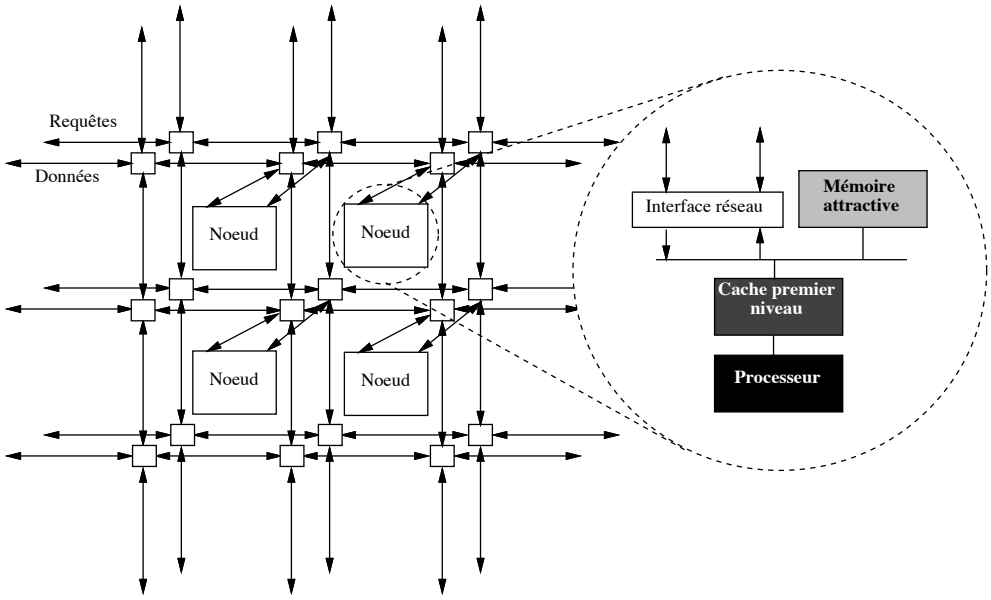


Figure VI.1 Architecture COMA non-hiérarchique

à transférer les requêtes et les données. D'autres réseaux directs utilisant des topologies différentes telles que des tores[2] ou des hypercubes [NI & McKinley 93] pourraient être envisagés sans problème. La grille à deux dimensions est cependant une topologie largement répandue et facilement extensible. L'organisation de l'architecture est très similaire aux architectures CC-NUMA telles que DASH [Lenoski *et al.* 92b] ou Alewife [Agarwal *et al.* 91].

Le détail de l'architecture d'un nœud apparaît sur la figure VI.2. Le processeur est connecté à un cache de premier niveau, lui-même connecté par un bus local à la mémoire contenant les données. Le cache de premier niveau se décompose en un cache d'instructions et un cache de données. La mémoire attractive est constituée de deux unités, la mémoire contenant les données, similaire à une mémoire standard, et la **mémoire d'états** qui permet de transformer la mémoire de données en un cache. Pour chaque ligne de la mémoire, la mémoire d'état maintient une étiquette identifiant la ligne, ainsi que l'état de la ligne dans la mémoire attractive. Les états utilisés sont définis par le protocole de cohérence.

Le cache de données de premier niveau est un cache à écriture retardée[3]. Il vérifie la propriété d'inclusion avec la mémoire attractive du nœud qui implique qu'une ligne mémoire ne peut se trouver dans le cache si elle n'existe pas déjà dans la mémoire attractive.

L'utilisation de plusieurs processeurs par nœud est parfaitement envisageable. Leur implémentation est cependant plus complexe car il est nécessaire d'assurer la cohérence des caches à l'intérieur d'un nœud.

[2] tore : grille dont les extrémités sont rebouclées sur elles-mêmes.

[3] Write-back cache

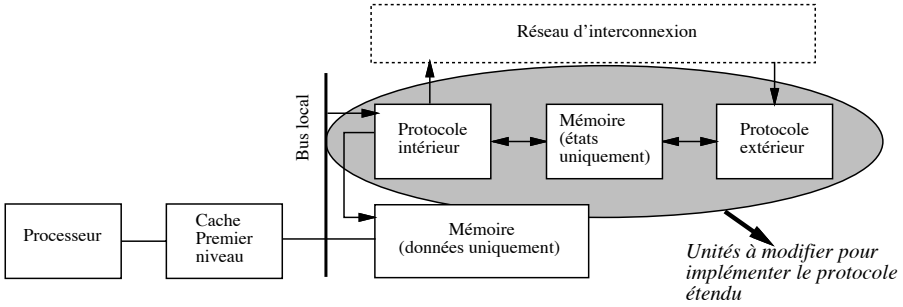


Figure VI.2 Organisation d'un nœud d'une architecture COMA

VI.1.1 Gestion de la cohérence

Le protocole de cohérence utilisé par les mémoires attractives se décompose physiquement en deux sous-protocoles mis en œuvre par deux unités supplémentaires.

- L'unité contenant le **protocole intérieur** (unité PI) espionne sur le bus local les requêtes en provenance du processeur. Elle vérifie dans la mémoire d'état à l'aide de l'adresse de la ligne accédée, si celle-ci est accessible dans la mémoire de données. Si la ligne est absente ou si son état n'autorise pas l'accès spécifié par la requête, l'unité PI annule la transaction en cours et se charge alors d'initier les transactions externes nécessaires pour que la requête puisse être servie (défaut de mémoire).
- L'unité contenant le **protocole extérieur** (unité PE) gère les requêtes et les réponses aux requêtes en provenance de l'extérieur d'un nœud. Elle partage logiquement la mémoire d'états avec l'unité PI. Physiquement, la mémoire d'états est souvent répliquée de manière à permettre des accès concurrents de l'unité PI et de l'unité PE [KSR 92].

Dans leur implémentation, ces deux unités se résument habituellement à des automates d'états finis chargés de définir une réponse ainsi qu'un nouvel état en fonction de la requête reçue et de l'état courant de la ligne accédée.

Lorsqu'une requête externe est initiée, il est nécessaire de localiser la ligne mémoire accédée. L'architecture utilise une solution similaire à une distribution statique des entrées de répertoire. Chaque ligne mémoire possède un pointeur de localisation (PL) conservant l'identité du propriétaire courant de la ligne. Les pointeurs de localisation sont distribués sur l'ensemble des nœuds de l'architecture à l'aide d'une fonction de placement. Pour simplifier la reconfiguration après défaillance, l'entrée de répertoire d'une ligne est associée à la ligne elle-même. Elle se trouve donc toujours sur le nœud actuellement propriétaire de la ligne. Lors d'un changement de propriétaire, l'entrée de répertoire migre avec la ligne. Sur chaque nœud, les entrées de répertoire sont stockées dans une mémoire qui peut être associée à la mémoire d'états de la mémoire attractive. Chaque nœud est capable d'allouer autant d'entrées de répertoire que de lignes se trouvant dans sa mémoire attractive. L'organisation

physique des entrées de répertoire peut utiliser n'importe laquelle des techniques présentées au chapitre II.1.

Le traitement d'un défaut de mémoire attractive est très similaire au traitement d'un défaut de cache dans une architecture CC-NUMA. Lors d'un défaut en lecture, un nœud envoie sa requête vers le nœud possédant le pointeur de localisation de la ligne accédée. Celui-ci fait suivre la requête vers le propriétaire courant de la ligne qui répond au nœud demandeur en lui envoyant une copie de la ligne. Dans le cas d'un défaut en écriture, le principe est similaire. Le propriétaire de la ligne se charge cependant des invalidations des copies de la ligne dans l'architecture avant de renvoyer la ligne et les droits exclusifs au nœud demandeur. Il se charge également de la mise à jour du pointeur de localisation de la ligne accédée. L'entrée de répertoire de la ligne est allouée sur le nouveau nœud propriétaire de la ligne.

VI.1.2 Gestion des mémoires attractives

VI.1.2.1 Type de mémoire

Différentes versions d'une mémoire attractive peuvent être implémentées [Hagersten *et al.* 94]. La version la plus simple est une mémoire attractive à correspondance directe[4] qui permet de débuter l'accès à la mémoire de données simultanément à l'accès à la mémoire d'états. Ce type de mémoire souffre cependant d'un manque d'associativité qui augmente ses taux de défaut. Une organisation associative par ensembles permet de corriger ce désavantage au prix, cependant, d'accès mémoire plus coûteux puisque pour une ligne, toutes les étiquettes d'un ensemble doivent être vérifiées avant que l'emplacement physique de la ligne dans la mémoire de données ne soit déterminé. Les architectures considérées utilisent ce dernier type d'organisation.

VI.1.2.2 Gestion des étiquettes

La gestion d'une étiquette pour chaque ligne d'une mémoire attractive peut nécessiter l'utilisation d'une mémoire d'état de grande taille augmentant le coût d'implémentation et le temps de vérification des étiquettes. Ainsi, pour une architecture contenant 1024 nœuds et utilisant des mémoires d'associativité 16, le nombre de bits nécessaires pour une étiquette est de 14 bits ($\log_2(1024 * 16) = 14$) [Joe & Hennessy 94]. Dans le cas de mémoires de 32 Mo avec des lignes de 128 octets, la mémoire nécessaire pour le stockage des étiquettes représente 448 Koctets. Il faut de plus rajouter à cette quantité, les bits codant l'état de chaque ligne.

Pour limiter la taille de la mémoire d'état, une solution consiste à utiliser une seule étiquette pour un ensemble de lignes contiguës groupées en page. Lorsqu'une allocation de ligne est nécessaire, une page entière est allouée dans la MA. Cette technique est utilisée

[4] Direct-mapped

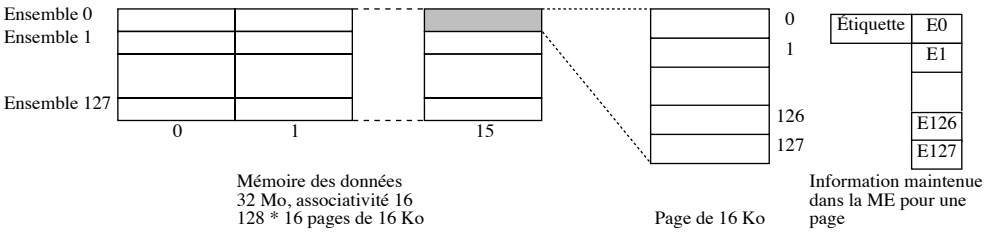


Figure VI.3 Organisation de la mémoire attractive dans l'architecture KSR1

dans l'architecture KSR1 (figure VI.1.2.2). Dans les MA, l'unité d'allocation est une page de $16 Koctets$ mais la cohérence est gérée sur des lignes de 128 octets. Un état est donc conservé pour chaque ligne mais une étiquette est utilisée pour un ensemble de 128 lignes. Dans le cas de la mémoire précédente, l'occupation mémoire des étiquettes se limite alors $28 Koctets$. Cette quantité est beaucoup plus adéquate pour assurer une vérification rapide des étiquettes. Elle possède cependant le désavantage de forcer l'allocation d'une page entière lorsqu'une seule ligne peut être nécessaire.

VI.1.2.3 Allocation de pages

Pour assurer la présence d'une copie de chaque ligne mémoire, les architectures COMA utilisent des mécanismes d'injection de lignes qui permettent de transférer une ligne d'un nœud à un autre lorsque la dernière copie d'une ligne doit être évacuée d'une MA. Des problèmes de place peuvent cependant survenir s'il n'existe pas sur un autre nœud, un emplacement prêt à accueillir la ligne injectée. Dans l'architecture KSR1, ce problème est résolu en allouant, pour chaque page accédée, une page **refuge** non remplaçable qui assure qu'il existe toujours un emplacement prêt à accueillir une injection pour les lignes de cette page.

L'implémentation du protocole de cohérence étendu pose le même type de problème. Lors des phases d'établissement des points de récupération, au minimum quatre copies sont nécessaires pour chaque ligne mémoire modifiée. Deux stratégies d'allocation peuvent alors être envisagées.

L'**allocation dynamique** est la stratégie la moins coûteuse en terme d'occupation mémoire. Elle peut cependant être pénalisante en temps lorsque les mémoires sont pleines et que des remplacements de pages sont nécessaires pour libérer de la place. Ces remplacements risquent notamment de rallonger les phases d'établissement des points de récupération et donc d'augmenter la dégradation de performance. Au contraire l'**allocation statique** de quatre pages refuge permet de faire face aux besoins de place mémoire lors de la sauvegarde des points de récupération. Elle augmente cependant l'occupation mémoire en particulier pour les pages non partagées qui peuvent se contenter de deux ou trois copies à l'intérieur d'une région de récupération. Les pages partagées souvent allouées en plusieurs exemplaires dans l'architecture peuvent ne pas créer de surcoût mémoire.

Dans la suite, nous considérons une allocation statique de quatre pages pour chaque page mémoire accédée. Nous montrons dans le chapitre dédié à l'évaluation le surcoût mémoire engendré par cette allocation.

VI.2 Intégration du protocole de cohérence étendu

Nous nous intéressons maintenant à l'intégration du protocole de cohérence étendu au sein de l'architecture considérée. Beaucoup des difficultés présentées ont été identifiées lors de la mise en œuvre du simulateur utilisé pour l'évaluation du protocole présentée au chapitre VII.

VI.2.1 Localisation des modifications

L'intégration du protocole de cohérence étendu implique certaines modifications des composants de base des nœuds. Ces modifications se limitent cependant aux composants utilisés pour la gestion des mémoires attractives (composants grisés sur la figure VI.2). Le comportement des caches et des processeurs ne se trouvent en rien modifiés.

Le composant le plus simple à adapter est la mémoire d'états dont la capacité doit simplement être augmentée pour pouvoir prendre en compte un nombre d'états plus important. L'augmentation précise de sa taille dépend du nombre d'états utilisés pour l'implémentation du protocole.

Le reste des modifications est relatif aux unités PI et PE qui doivent être capables de gérer les nouvelles requêtes d'injection ainsi que les algorithmes d'établissement et de restauration des points de récupération. Ces algorithmes restent toutefois simples à implémenter puisqu'ils se limitent à un parcours séquentiel des mémoires attractives.

VI.2.2 Nouveaux états

Dans un système où les écritures ne sont pas atomiques, la propriété de cohérence est assurée en autorisant, à un instant donné, un seul nœud à pouvoir modifier une ligne mémoire, c'est-à-dire en assurant la présence d'un seul propriétaire de la ligne. Avec le protocole étendu, une différenciation des deux copies *Partagé-CK* devient nécessaire de façon qu'une seule d'entre elles puisse délivrer les droits exclusifs sur une ligne. Deux états *Partagé-CK1* et *Partagé-CK2* doivent donc être utilisés. La copie *Partagé-CK1* est alors la seule apte à délivrer les droits exclusifs sur la ligne correspondante. Le nœud possédant cette copie possède également l'entrée de répertoire associée. Les copies *Invalide-CK* pouvant être restaurées comme copies *Partagé-CK*, deux états *Invalide-CK1* et *Invalide-CK2* doivent également être utilisés.

VI.2.3 Modifications du protocole intérieur

La version du protocole intérieur pour le protocole de cohérence étendu reste relativement similaire à celle que l'on pourrait avoir avec un protocole de cohérence standard. Du point de vue des accès autorisés, seuls, les accès en lecture sur des copies *Partagé-CK1* ou *Partagé-CK2* doivent être rajoutés. L'essentiel des modifications intervient lors d'accès non autorisés sur des copies de récupération qui doivent alors générer des injections de ligne. Ainsi tout accès en lecture ou en écriture sur une copie *Invalide-CK*[5] génère une injection suivie d'un traitement de défaut standard. L'accès en écriture sur une copie *Partagé-CK1* est traité en réalisant d'abord les invalidations puis en les faisant suivre d'une injection de copie *Invalide-CK1*. L'accès en écriture à une copie *Partagé-CK2* est traité en réalisant directement une injection de copie *Invalide-CK2*, puisque la ligne va être modifiée, suivie d'un défaut en écriture sur la ligne.

Les requêtes engendrées par le protocole de cohérence intérieur n'étant pas atomiques, leur implémentation nécessite l'introduction d'un certain nombre d'états transitoires chargés de gérer des situations où une requête est en cours de résolution [Pong *et al.* 94]. Par exemple, pendant qu'un nœud réalise les invalidations nécessaires pour un accès en écriture, il change l'état de la ligne concernée pour empêcher son accès par d'autres nœuds. Tout accès à la ligne par un nœud distant est alors traité en répondant avec un acquittement négatif qui indique que le nœud n'est pas prêt à répondre. Par rapport à un protocole de cohérence standard, de nouveaux états transitoires sont utilisés notamment pour traiter les injections de copies de récupération.

VI.2.4 Modifications du protocole extérieur

Avec le protocole de cohérence étendu, le protocole extérieur doit incorporer la gestion des requêtes de lecture et d'écriture sur des copies de récupération non modifiées. Leur traitement est relativement simple puisqu'il est similaire au traitement réalisé pour des données courantes. Le nœud propriétaire de la ligne est le nœud possédant la copie *Partagé-CK1*. Dans le cas d'un accès en lecture, le propriétaire est contacté par un nœud en défaut. Il lui délivre une copie de la ligne et note l'identité du nœud demandeur dans l'entrée de répertoire de la ligne. Pour une requête d'écriture, le propriétaire se charge d'envoyer les messages d'invalidation des copies de la ligne (en particulier vers la copie *Partagé-CK2*) avant de changer son état à *Invalide-CK1* et de donner les droits exclusifs au nœud demandeur. Comme pour les données courantes, l'accès à la ligne est inhibé pendant tout le temps des invalidations.

Il arrive fréquemment, avec les architectures considérées, que plusieurs processeurs désirent accéder simultanément à une même ligne mémoire. Le but du protocole extérieur est également de gérer ces accès concurrents de façon à assurer la propriété de cohérence et l'absence d'interblocage. Pour cela de nouveaux états transitoires sont généralement utilisés

[5] Dans la suite, une copie *Invalide-CK* désigne une copie *Invalide-CK1* ou *Invalide-CK2*.

[Pong *et al.* 94]. Les états transitoires définis par le protocole extérieur dépendent des accès mémoire conflictuels qui peuvent survenir dans l'architecture. La majorité des états transitoires rajoutés par le protocole de cohérence étendu sont introduits pour gérer les injections de copies de récupération de lignes mémoire. La gestion des requêtes sur des copies courantes de lignes reste similaire à celle que l'on peut trouver dans une architecture COMA standard [Pong *et al.* 94].

Une situation particulièrement problématique survient cependant lors de la première modification d'une ligne dans une région de récupération. Un cas d'interblocage apparaît en effet lorsqu'un nœud, désirant modifier une ligne dans l'état *Partagé-CK1* dans sa mémoire attractive, réalise les invalidations puis tente une injection de la ligne alors que tous les nœuds susceptibles d'accepter l'injection sont également en attente d'écriture sur cette ligne. Le nœud possédant les droits exclusifs ne peut alors réaliser son injection et aucun des autres nœuds ne peut recevoir les droits exclusifs. Une situation similaire apparaîtrait si l'injection était causée par un remplacement de page. Une solution à ce problème consiste alors à utiliser un état transitoire qui indique que le nœud est bien en cours d'injection mais qu'il dispose des droits exclusifs sur la ligne. En autorisant le nœud à délivrer ces droits à un des nœuds en attente d'écriture, la situation se débloque. Un des nœuds en attente d'écriture devient alors *Exclusif* et l'injection peut être transformée en injection standard de copie *Invalide-CK1*.

D'autres situations problématiques sont également traitées par l'utilisation d'états transitoires. Il apparaît cependant que les modifications à réaliser pour l'implémentation du protocole de cohérence étendu restent faibles. Aucune nouvelle fonctionnalité n'est introduite. Le traitement des accès à des copies de récupération est similaire au traitement des accès aux données courantes. La majorité des états transitoires introduits sont utilisés pour traiter les injections de copies de récupération.

VI.2.5 Gestion des injections de lignes

Une des particularités du protocole de cohérence étendu est d'introduire de nouveaux cas d'injection de copies de lignes dans l'état *Invalide-CK* ou *Partagé-CK*. La dégradation de performance engendrée par le protocole de cohérence étendu dépend en partie de l'efficacité du traitement de ces injections. Les injections de copies de lignes privées sont peu coûteuses car elles n'ont lieu qu'une seule fois après l'établissement d'un point de récupération. Au contraire les injections de copies de lignes partagées doivent être réalisées avec soin car les mouvements de la copie courante d'une ligne peuvent provoquer de multiples injections.

Dans l'architecture considérée, les injections sont traitées en deux étapes. Dans un premier temps, une requête recherche un nœud acceptant la copie de ligne à injecter. Une fois celui-ci trouvé, la copie de la ligne est effectivement envoyée vers ce nœud.

VI.2.5.1 Recherche d'un nœud

Pour une copie de ligne partagée, la recherche d'un nœud susceptible d'accepter une injection est primordiale pour minimiser la dégradation de performance. En particulier ce choix doit être réalisé de façon à minimiser le nombre global d'injections. Les injections de copies courantes de lignes peuvent être réalisées en utilisant des optimisations telle que celle présentée dans [Joe & Hennessy 94]. Le traitement des injections de copies de récupération nécessite cependant une autre stratégie. Vu la technique d'allocation dans les mémoires attractives, beaucoup de lignes mémoire appartenant à des pages partagées, risquent d'être allouées suite à du faux partage et donc de rester inutilisées. De telles lignes sont des candidates idéales pour la conservation des copies de récupération de type *Invalide-CK* inutiles pour le calcul en cours. Lors d'une injection l'objectif est donc de trouver une ligne de ce type afin de profiter au maximum des emplacements mémoire alloués qui restent inutilisés.

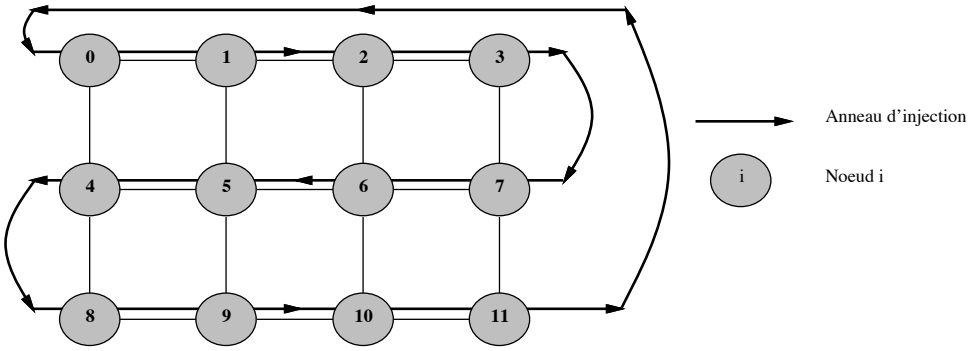


Figure VI.4 Exemple d'anneau d'injection sur une grille 4x3

À la différence d'une architecture utilisant l'espionnage où une requête visite l'ensemble des nœuds, il n'existe aucun moyen, avec l'architecture considérée, d'identifier un nœud possédant une copie de ligne allouée et non utilisée. Le seul algorithme envisageable est alors de tester les nœuds un à un. Pour réaliser ces tests, un anneau logique, que nous appelons **anneau d'injection**, peut être plaqué sur le réseau d'interconnexion physique utilisé par l'architecture (figure VI.4). Un nœud désirant injecter une ligne envoie donc sa requête vers le nœud suivant sur l'anneau d'injection. Si celui-ci accepte l'injection, il acquitte la requête. Dans le cas contraire la requête est transmise au nœud suivant sur l'anneau d'injection. L'utilisation de l'anneau d'injection permet à une copie de récupération d'atteindre, en une ou plusieurs injections, un nœud où elle occupera finalement un emplacement mémoire alloué mais non utilisé. Bien entendu, l'anneau d'injection doit être reconfiguré en cas de défaillance permanente d'un nœud.

Afin de minimiser les temps d'injection de lignes, l'anneau d'injection peut également être utilisé pour allouer les quatre copies d'une page mémoire. Dans le cas de pages privées, les quatre copies sont allouées de façon contiguës selon l'anneau à partir du nœud accédant à la page. Une injection d'une ligne de cette page est alors réalisée, en moyenne, après la visite de 2,5 nœuds.

VI.2.5.2 Politique d'injection

La politique d'injection a pour but d'indiquer si un nœud est autorisé à accepter une injection de ligne. Cette politique est un compromis entre une minimisation des temps d'injection et une minimisation de leur nombre. La politique choisie vise à limiter le temps des injections en remplaçant éventuellement des copies valides par des copies de récupération. Elle se définit par quatre règles résumées dans la table V.1 :

- Une injection de ligne ne peut être acceptée par un nœud que si la ligne correspondante est déjà allouée dans sa mémoire attractive. Cette règle inhibe l'allocation dynamique de pages. L'allocation statique de quatre pages refuge dans le système garantit qu'une injection pourra toujours être réalisée.
- Une injection de ligne *Exclusif* ou *Modifié Partagé* ne peut être acceptée par un nœud contenant déjà une copie de récupération de cette ligne. Dans tout autre état, un nœud accepte ce type d'injection, même s'il est en cours d'accès sur la même ligne, l'accès se trouvant ainsi servi.
- Les injections de copies *Invalide-CK* sont acceptées par toute copie *Invalide* ou *Partagé* d'une ligne. L'utilisation de copies *Partagé* augmente les chances que l'injection soit traitée rapidement et assure de plus que l'injection pourra être réalisée. Pour ne pas complexifier l'implémentation du protocole de cohérence, ces injections ne sont pas acceptées par des mémoires possédant une copie de la ligne dans un état transitoire. Aucun risque d'interblocage n'est cependant à craindre car la présence d'un minimum de quatre copies d'une même ligne assure qu'il existera tôt ou tard une copie *Invalide* ou *Partagé* permettant d'accepter l'injection. Ce type d'injection est donc réitéré jusqu'à ce que l'injection soit servie.
- Les injections de copies *Partagé-CK* sont aussi acceptées par toute copie *Invalide* ou *Partagé* d'une ligne. De plus, tout nœud en attente de lecture sur cette ligne est autorisé à accepter ce type d'injection, la requête se trouvant ainsi servie.

Il est important de noter que l'injection d'une copie *Invalide-CK* ne peut engendrer un interblocage car ces copies ne disposent pas des droits exclusifs sur une ligne, elles ne peuvent donc pas perturber les accès aux copies courantes. Seules les injections de copies *Partagé-CK* sont délicates à traiter.

VI.2.5.3 Injections et défaillances

Si les injections de copies courantes de lignes mémoire ne posent aucun problème puisqu'elles sont de toutes façons invalidées en cas de faute, les injections deviennent problématiques lorsque les données manipulées sont critiques pour le recouvrement. C'est le cas des copies de récupération dont les propriétés de stabilité doivent être assurées même lors des

	Invalide	Partagé	Invalide-CK	Partagé-CK	Lecture en cours	Écriture en cours	Autres états transitoires
Exclusif	oui	oui	non	∅	oui	oui	non
Modifié Partagé	oui	oui	non	∅	oui	oui	non
Invalide-CK	oui	oui	non	non	non	non	non
Partagé-CK	oui	oui	non	non	oui	non	non

∅ : situation impossible

Table VI.1 Table des injections

injections. En effet si une défaillance entraîne la perte d'une copie de récupération d'une ligne alors que la seconde copie est en cours d'injection, le système peut se retrouver dans l'impossibilité de restaurer le point de récupération par suite de la perte des deux copies de récupération.

Une façon simple de traiter ce problème consiste à conserver une copie d'une ligne injectée sur le nœud réalisant l'injection jusqu'à la réception d'un acquittement de l'injection. Cette solution implique qu'il existe à un instant donné plus de deux copies de récupération d'une ligne présentes dans l'architecture. Cette possibilité est donc à considérer durant une phase de reconfiguration dont le rôle est d'assurer la présence d'exactement deux exemplaires d'une copie de récupération de chaque ligne mémoire. L'utilisation d'états transitoires permet toutefois d'identifier de tels cas.

Les injections de copies *Partagé-CK* sont un peu plus délicates à traiter puisqu'elles doivent de plus assurer la correction de l'information de cohérence localisée dans l'entrée de répertoire associée à la copie *Partagé-CK1*. En particulier le pointeur sur la copie *Partagé-CK2* doit toujours être correct pour que la cohérence soit respectée. Ainsi, si l'injection d'une copie *Partagé-CK1* ne pose pas de problème, l'injection d'une ligne *Partagé-CK2* nécessite une phase de concertation afin que la copie *Partagé-CK1* mette correctement à jour son pointeur. Les échanges de messages entre les différents nœuds impliqués dans cette opération sont représentés sur la figure VI.5. À partir du moment où l'injection est autorisée et jusqu'à la mise à jour du pointeur de la copie *Partagé-CK1*, celle-ci ne peut être injectée sur un autre nœud. Le coût d'une injection de copie *Partagé-CK2* est donc supérieur à celui d'une injection standard.

VI.3 Sauvegarde d'un point de récupération

L'algorithme de sauvegarde d'un point de récupération se décompose en deux phases. La première phase a pour but de répliquer l'ensemble des lignes modifiées. La seconde permet de collecter les lignes inutiles et de confirmer les nouvelles copies de récupération. Même si ces algorithmes sont simples, leur intégration dans des architectures réelles peut nécessiter l'introduction de matériel spécifique pour limiter leur durée.

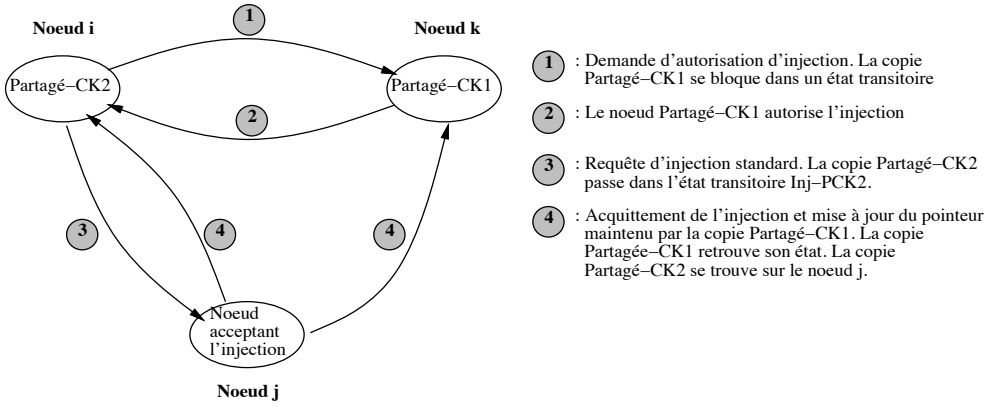


Figure VI.5 Étapes d'une injection de copie *Partagé-CK2*

VI.3.1 Première phase

Sur chaque nœud, l'algorithme exécuté durant la première phase de la sauvegarde d'un point de récupération se résume à un parcours séquentiel de la mémoire pour identifier et répliquer les lignes modifiées. Si cet algorithme est simple, il devient rapidement coûteux dès que la taille des mémoires est importante. À titre d'exemple, une mémoire attractive de 32 Mo utilisant des lignes de 128 octets possède 256K lignes à tester. Parmi ces lignes, seule une faible quantité risque d'être modifiée.

Pour remédier à des temps de recherche excessifs des lignes modifiées, un dispositif simple, représenté sur la figure VI.6, peut être utilisé. L'information indiquant si une ligne est modifiée est organisée en arbre. Chaque niveau de l'arbre résume les informations situées aux niveaux inférieurs. Ainsi le premier niveau divise la mémoire d'un nœud en n parties (n étant le nombre de bits) et chacun des bits indique alors si cette partie de la mémoire contient une ligne modifiée. Les mots se trouvant à la base de l'arbre indique pour chaque ligne si elle est modifiée.

Sur l'exemple présenté, l'information est codée sur des mots de 16 bits. Elle occupe un peu plus de 8Ko. Si l'on considère que cette information est stockée dans des mémoires vives statiques (SRAM), et qu'il faut 1 cycle pour lire et 1 cycle pour tester un mot de 16 bits, 8 cycles sont nécessaires pour identifier une ligne modifiée. Une organisation similaire est utilisée au niveau des pages de l'architecture KSR1, où les états des 128 lignes d'une page sont résumés dans un arbre à 3 niveaux.

En fonctionnement normal d'un nœud, le coût de mise à jour de cette information n'est pas prohibitif. Une mise à jour est en effet nécessaire seulement lorsqu'un nœud acquiert ou perd la propriété d'une ligne modifiée dans la région de récupération courante. Dans de tels cas, la mise à jour peut être réalisée simultanément à l'accès mémoire nécessaire pour lire ou écrire la ligne dans la mémoire. Un tel dispositif est indispensable si l'on désire assurer une dégradation de performance minimale et que les tailles des mémoires attractives sont

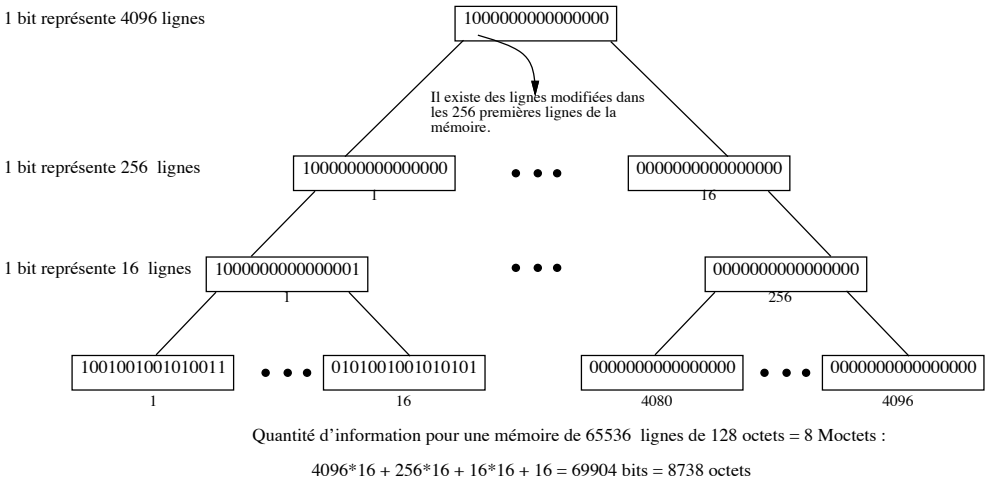


Figure VI.6 Organisation de l'information pour une recherche rapide des lignes modifiées

importantes.

VI.3.1.1 Réplication des lignes

La réplication de lignes nécessaire lors de la sauvegarde des points de récupération est relativement simple, elle n'introduit pas de nouvelles fonctionnalités. En effet, lors de la sauvegarde d'un point de récupération, les requêtes utilisées sont similaires à des injections de lignes. La seule différence est que la ligne injectée est conservée.

Au contraire d'une architecture qui utiliserait un protocole de cohérence à base d'espionnage, l'information de cohérence maintenue par le propriétaire d'une ligne permet de plus d'utiliser la réplication de donnée existante pour limiter les transferts de données. Ainsi, pour une ligne *Modifié Partagé*, la présence d'une autre copie *Partagé*, facilement détectée dans l'entrée de répertoire, permet de limiter la réplication à un simple envoi de requête. Cette propriété affecte le temps d'établissement d'un point de récupération en fonction du taux de réplication des lignes mémoire partagées. Ce taux dépend lui-même complètement du comportement des applications.

Lorsque la ligne n'est pas répliquée (cas d'une ligne *Exclusif*), la requête d'injection utilise l'anneau d'injection pour trouver une place dans une des mémoires de l'architecture.

VI.3.2 Seconde phase

L'algorithme local exécuté durant la seconde phase de l'établissement d'un point de récupération nécessite également un parcours séquentiel de la mémoire d'état. La durée de cette phase peut bien évidemment être la cause d'une dégradation de performance importante.

Les solutions envisageables pour limiter sa durée sont multiples. Un dispositif matériel similaire à celui présenté pour identifier les lignes modifiées peut être envisagé. En l'absence d'une telle optimisation, les tests peuvent être limités aux pages allouées ou modifiées des mémoires attractives. Le chapitre VIII décrit une optimisation permettant de se passer du parcours séquentiel de la mémoire d'état lors de cette phase.

VI.4 Traitement des fautes

La phase de traitement de faute a pour objectif de reconfigurer l'architecture afin qu'elle puisse reprendre son calcul tout en tolérant toujours les fautes des nœuds encore valides. Le type de faute est le principal facteur qui influence sa complexité. Les fautes temporaires sont ainsi plus simples à traiter puisqu'aucune mémoire n'est perdue. Dans ce paragraphe, nous nous limitons aux aspects matériels du traitement d'une faute. Nous ne considérons pas le travail à réaliser au niveau du système d'exploitation pour redémarrer les calculs.

VI.4.1 Faute temporaire

En l'absence de perte d'une des mémoires de l'architecture, le traitement d'une faute temporaire se limite à une restauration du point de récupération sauvegardé. Aucune allocation de mémoire n'est ici nécessaire.

Par rapport à une architecture utilisant un protocole à base d'espionnage, le traitement des fautes temporaires est plus complexe dans le cas d'une architecture utilisant un protocole à base de répertoire. En effet, dans ce type d'architecture, deux types d'information sont également nécessaires : des **informations de cohérence** maintenues dans les entrées de répertoire, et des **informations de localisation** utilisées pour retrouver une ligne. Après un retour arrière une partie de ces informations peut avoir besoin d'être remis à jour.

VI.4.1.1 Informations de cohérence

Les informations de cohérence maintenues pour des copies courantes d'une ligne ne sont pas vitales car ces lignes sont invalidées lors d'un retour arrière. Au contraire pour des copies de récupération ces données doivent absolument être à jour. Ainsi, pour maintenir la cohérence, une copie *Partagé-CK1* d'une ligne est supposée connaître l'identité du nœud possédant la copie *Partagé-CK2*. Après un retour arrière, les lignes *Invalide-CK1* restaurées en lignes *Partagé-CK1* peuvent cependant avoir perdu cette information.

Une première solution à ce problème consiste à conserver des pointeurs pour les copies *Invalide-CK*. La copie *Invalide-CK1* maintient alors, dans une entrée de répertoire, un pointeur sur la copie *Invalide-CK2*. La copie *Invalide-CK2* fait de même en conservant un

pointeur sur la copie *Invalide-CK1*. La gestion de ces pointeurs alourdit cependant l'algorithme d'injection des lignes *Invalide-CK* qui doit alors utiliser une concertation similaire à celle présentée pour l'injection de copies *Partagé-CK2*. Une seconde solution consiste à utiliser un mécanisme de diffusion lorsque l'identité de la copie *Partagé-CK2* n'est pas connue. Aucun pointeur n'est alors conservé par les copies *Invalide-CK* et leurs injections restent donc simples. Après un retour arrière, la première modification d'une ligne *Partagé-CK1* ne connaissant pas l'identité du nœud possédant la copie *Partagé-CK2* engendre alors une diffusion d'un message d'invalidation. L'occurrence d'une telle situation étant rare ce surcoût est tout à fait acceptable. Pour une ligne, il se trouve de plus limité à la première modification après un retour arrière puisque des pointeurs sont recréés lors de la prochaine sauvegarde d'un point de récupération.

VI.4.1.2 Informations de localisation

Pour l'architecture considérée ici, l'information de localisation se résume aux pointeurs de localisation des lignes. Après un retour arrière, l'ensemble des lignes modifiées dans la région de récupération, possède des pointeurs de localisation erronés. En effet ces derniers conservaient l'identité des copies courantes de ces lignes. Ici aussi, différentes solutions sont envisageables. La plus simple consiste à remettre à jour, lors du retour arrière, les pointeurs de localisation de l'ensemble des lignes *Invalide-CK1* restaurées comme lignes *Partagé-CK1*. Cette solution est la plus coûteuse au moment du retour arrière puisque pour chaque ligne *Partagé-CK1* restaurée, un nœud doit envoyer un message vers le nœud possédant le pointeur de localisation de la ligne. Une autre solution, moins coûteuse lors du traitement de faute, consiste à reprendre les calculs et à utiliser un mécanisme de diffusion lorsqu'un tel pointeur est erroné. La restauration est donc simplifiée mais le calcul est ralenti par les diffusions nécessaires. Ce type de diffusion n'est cependant nécessaire que lors du premier accès à une ligne après une faute. Finalement la dernière solution envisageable consiste à maintenir deux pointeurs de localisation pour une ligne mémoire ; un pointeur sur le propriétaire courant et un pointeur sur le nœud possédant la copie *Invalide-CK1* de cette ligne. Cette dernière solution nécessite une mise à jour du pointeur de localisation à chaque injection d'une copie *Invalide-CK1* et est donc plus coûteuse en fonctionnement normal. Le choix d'une de ces solutions est un compromis à trouver entre le temps alloué à la reconfiguration, et la dégradation de performance engendrée.

VI.4.2 Faute permanente

Dans le cas d'une faute permanente la restauration du point de récupération doit être accompagnée d'une étape de passivation de la faute permettant d'inhiber le nœud défaillant et de permettre à l'architecture de poursuivre son calcul. Une conséquence directe de l'éviction d'un nœud de l'architecture est la perte de toutes les copies de lignes situées dans sa mémoire attractive. Une reconfiguration mémoire est donc nécessaire pour assurer la réplication de

chaque ligne mémoire.

Dans l'architecture étudiée, l'information de cohérence maintenue par les copies *Partagé-CK1* permet de limiter la durée de cette phase. Ainsi une copie *Partagé-CK1*, non modifiée dans la dernière région de récupération, peut utiliser le pointeur sur la copie *Partagé-CK2* afin de vérifier si cette copie a disparu. Dans le cas où aucun pointeur n'est conservé par les copies *Invalide-CK*, la reconfiguration nécessite une vérification pour chaque copie *Invalide-CK* restaurée. Cette vérification implique la diffusion d'un message afin de vérifier la présence de la seconde copie d'une ligne dans l'architecture. Dans le cas d'une réponse négative, une nouvelle copie de la ligne doit être allouée sur un nœud valide. Au contraire si les copies de récupération maintiennent constamment des pointeurs l'une sur l'autre, la phase de réallocation mémoire peut être considérablement réduite. Chaque nœud est alors capable d'identifier la perte d'une copie de ligne sans engendrer de message de diffusion.

La perte d'un nœud de l'architecture entraîne également la disparition d'une partie des pointeurs de localisation de l'architecture. Une réallocation des pointeurs perdus est donc également nécessaire. Lorsqu'aucun nœud de secours n'est prévu pour remplacer le nœud défaillant, cette réallocation suppose que les nœuds possèdent des pointeurs de localisation de secours non utilisés en fonctionnement normal ou qu'ils sont capables d'en allouer de nouveaux. Une solution simple consiste alors à affecter à un nœud l'ensemble des pointeurs préalablement alloués au nœud défaillant. On peut ainsi tolérer la défaillance de la moitié des nœuds de l'architecture. L'identité d'un nœud possédant le pointeur de localisation d'une ligne pouvant changer, il est nécessaire que cette information puisse être modifiée. Si la distribution des pointeurs de localisation utilise une fonction du type $modulo(N)$ sur l'adresse des lignes, où N est le nombre de nœuds, cette information nécessite, par nœud, une table contenant N entrées. Lors d'un défaut cette table indique le nœud à contacter pour accéder au pointeur de localisation de la ligne.

VI.4.3 Conclusion

Le travail à réaliser lors du traitement d'une faute est conceptuellement simple, il peut cependant être coûteux car l'absence de localisation physique fixe des copies de récupération nécessite une restauration des informations de cohérence et de localisation. Dans le cas de fautes temporaires, il semble réaliste d'envisager l'utilisation de mécanismes de diffusion lorsqu'un pointeur de localisation n'est pas à jour ou lorsque l'information de cohérence est perdue. En effet, ces diffusions ne seront nécessaires qu'une seule fois après une faute. Pour les fautes permanentes, il est impératif de vérifier la présence de deux copies de chaque ligne. Sans information supplémentaire, la phase de reconfiguration mémoire peut être longue car chaque ligne restaurée doit être testée. En ajoutant de l'information aux copies de récupération de type *Invalide-CK*, la reconfiguration mémoire peut être beaucoup plus rapide. Le prix à payer est cependant une dégradation de performance plus importante en fonctionnement normal. L'utilisation de telles informations ne peut donc se justifier que si des contraintes de temps sont fixées pour la reconfiguration.

VI.5 Résumé

Dans ce chapitre nous avons étudié comment implémenter le protocole de cohérence étendu présenté au chapitre V.1 dans une architecture de type COMA non hiérarchique. Il ressort de cette étude que :

(1) Les modifications à réaliser sur une architecture standard sont faibles et localisées aux unités de gestion des mémoires attractives.

(2) Le protocole de cohérence étendu doit être enrichi d'un certain nombre d'états transitoires permettant de régler les situations de conflit d'accès à une même ligne mémoire. La majorité des états transitoires nécessaires sont utilisés pour gérer des cas d'injection de ligne.

(3) La politique d'injection utilisée permet aux données de récupération de tirer parti de l'ensemble des emplacements mémoire alloués.

(4) Même s'il est simple, l'algorithme de sauvegarde des points de récupération nécessite du matériel supplémentaire pour être efficace.

(5) Le traitement d'une faute temporaire est simple car les informations de cohérence et de localisation ne sont pas nécessaires si une diffusion peut être réalisée. Dans le cas d'une faute permanente, l'information de localisation maintenue dans les entrées de répertoires peut être utilisée pour limiter la durée de la reconfiguration mémoire. Le maintien d'informations supplémentaires associées aux copies *Invalide-CK* permet de limiter la durée d'une reconfiguration mémoire au prix, toutefois, d'injections de lignes plus coûteuses.

Chapitre VII

Évaluation du protocole

Ce chapitre présente une étude détaillée du comportement du protocole de cohérence dans l'architecture COMA présentée précédemment. L'étude est menée à l'aide d'un simulateur et de traces d'applications parallèles. Une fois la technique de simulation et les paramètres de l'architecture fixés, nous étudions dans un premier temps l'ensemble des surcoûts temporels et spatiaux engendrés par le protocole de cohérence étendu avec une machine comportant 30 nœuds. Nous nous intéressons ensuite à l'extensibilité de notre approche en présentant des dégradations de performance pour un nombre de nœuds variant de 9 à 56. Le chapitre se termine par des résultats de simulation obtenus pour une architecture ayant des caractéristiques envisageables dans un futur proche.

VII.1 Technique de simulation

L'évaluation d'une architecture multiprocesseur n'est pas une chose aisée. En l'absence d'architecture réelle, elle ne peut être réalisée que par simulation ou modélisation. La modélisation est généralement employée lorsque le comportement de l'architecture est prévisible. Dans notre cas, la simulation semble plus indiquée car l'influence du protocole sur le comportement des mémoires attractives n'est pas connu.

Un simulateur d'architecture se compose généralement de trois composants. Un générateur d'adresses chargé de produire les références mémoire, un modèle de l'architecture qui modélise le fonctionnement, les opérations et les interconnexions entre les composants, et un noyau de simulation qui synchronise la consommation des références mémoire produites par le générateur.

VII.1.1 Générateur d'adresses

Le choix de la charge de travail utilisée pour évaluer les performances d'une architecture intervient pour une part importante dans la validité et l'extrapolation des résultats de

simulation permettant de prédire le comportement général de la machine. En l'absence de générateur de références mémoire synthétiques adaptées à des architectures multiprocesseurs, notre étude est menée à l'aide de traces d'adresses d'applications parallèles réelles.

L'évaluation que nous présentons dans ce chapitre est réalisée à l'aide du noyau de simulation SPAM [Gefflaut & Joubert 96]. SPAM assure le traçage des applications parallèles, et permet également de mettre en œuvre de façon efficace une technique de **simulation coordonnée à l'exécution** (*Execution Driven Simulation*) [Covington *et al.* 88, Davis *et al.* 91]. Ce type de technique de simulation corrige les défauts des techniques consistant à collecter une trace d'adresse sur une machine puis à l'utiliser pour en simuler une autre (*Trace Driven Simulation*). En effet, si ces dernières techniques sont valides dans le cas séquentiel, elles se heurtent dans le cas parallèle au problème de **décalage de traces** (*Trace Shifting*) [Dubois *et al.* 86]. Ce problème apparaît lorsque les traces des processus ne sont plus en phase avec l'exécution que l'on devrait obtenir sur l'architecture simulée. Il est engendré par les différences existant dans les exécutions possibles d'une même application parallèle sur des machines possédant des caractéristiques différentes. Une technique de simulation coordonnée à l'exécution inhibe le décalage de trace et permet d'obtenir des résultats de simulation plus réalistes.

Avec une technique de simulation coordonnée à l'exécution, la production des traces et l'exécution de l'application, s'effectuent sous le contrôle du simulateur, de façon simultanée à la simulation de l'architecture cible. À chaque fois qu'un processus de l'application tracée désire effectuer une opération pouvant avoir un impact sur la trace générée, l'exécution du processus et donc la production de sa trace d'adresse est arrêtée avant l'exécution de l'opération. Lorsque la trace du processus a été simulée jusqu'à cette opération, l'opération est effectuée par le simulateur qui libère alors le processus. L'exécution parallèle étant controlée par le simulateur, la trace d'adresse obtenue est exactement celle qui serait produite si l'application avait été réellement exécutée sur la machine simulée.

Le noyau SPAM fournit toutes les primitives nécessaires au contrôle de l'exécution des processus d'une application parallèle s'exécutant sur une station de travail de type SUN. Il permet donc de simuler de façon réaliste des architectures multiprocesseur à partir d'une machine monoprocesseur.

VII.1.2 Simulateur et noyau de simulation

Le corps du simulateur utilisé est écrit en C++ et utilise une librairie de simulation à événements discrets [Schwetman 92]. Cette librairie fournit un ensemble de primitives permettant la création, la gestion et la synchronisation de processus légers au sein d'un même processus UNIX. Chacun des composants de l'architecture cible (processeur, cache, mémoire attractive...) est modélisé par un objet C++. Chaque objet est indépendant et possède une interface qui lui permet d'envoyer ou de recevoir des requêtes. Un objet n'a pas connaissance des composants auxquels il est connecté, lorsqu'il désire envoyer un message

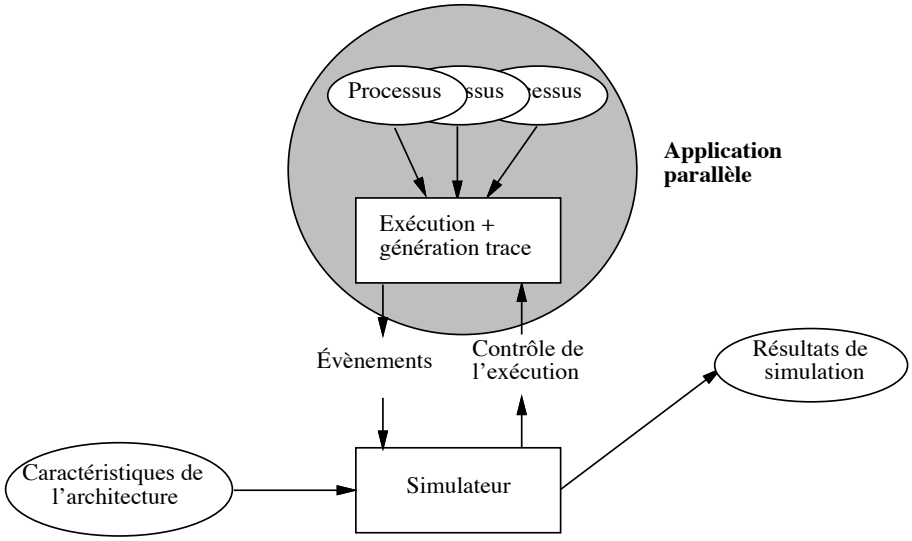


Figure VII.1 Simulation coordonnée à l'exécution

son interface se charge de délivrer le message à l'objet auquel il est connecté. Il est ainsi fort simple d'ajouter de nouveaux composants dans l'architecture simulée.

Chaque nœud de l'architecture est simulé grâce à un processus léger interagissant avec l'ensemble des composants de l'architecture. L'interfaçage entre le simulateur et le noyau de simulation SPAM est réalisée de façon simple grâce aux primitives offertes par SPAM. Chacun des objets processeur est connecté à un des processus de l'application tracée de sorte que chacun des nœuds de l'architecture correspond à un des processus de l'application. Aucune migration de processus n'est considérée pendant les simulations.

VII.1.3 Charge de travail

Il existe à l'heure actuelle assez peu d'applications pour des multiprocesseurs à mémoire partagée permettant de construire un ensemble d'applications représentatif et cohérent. La suite d'applications SPLASH (*Stanford Parallel Applications for Shared Memory*) [Singh *et al.* 91] a été développée afin de servir de base aux évaluations de performance des multiprocesseurs à mémoire partagée. Elle est constituée de sept applications scientifiques provenant de domaines variés tels que le calcul mathématique, l'aéronautique, l'océanographie ou la CAO de circuits. Cet ensemble d'applications a déjà été utilisé dans un très grand nombre d'études de performances [Joe & Hennessy 94, Hagersten *et al.* 91, Gupta & Weber 92, Wilkinson 93, Stenström *et al.* 92] et tend à devenir un standard dans le domaine de l'évaluation des multiprocesseurs à mémoire partagée.

VII.1.3.1 Applications choisies

Dans notre évaluation, nous retenons quatre applications de SPLASH : **Barnes-Hut**, **Cholesky**, **Mp3d** et **Water**. Ces applications parallèles fournissent un panel de comportement vis à vis de la mémoire partagée qui donne à notre évaluation toute sa validité. Toutes ces applications sont écrites en C et utilisent les macro-instructions du *Argonne National Laboratory* (ANL) pour la gestion de la mémoire partagée et de la synchronisation. Elles ont été initialement développées pour des machines à base de bus garantissant un temps d'accès uniforme à la mémoire (UMA). Elles ne font donc aucune hypothèse sur le placement des données par rapport aux processeurs. Les applications se composent d'un ensemble de processus travaillant sur un espace mémoire partagé. Un processus **père** est chargé de l'initialisation des structures de données et du lancement d'un nombre paramétrable de processus **fils** identiques. Les différents processus d'une application se synchronisent à l'aide de verrous d'exclusion mutuelle (*locks*) et de barrières assurant des **rendez-vous**.

VII.1.3.2 Description des applications

Ce paragraphe décrit brièvement les applications retenues pour notre évaluation. Il donne aussi leur comportement et les paramètres d'entrée utilisés dans les simulations.

- **Barnes-Hut** simule l'évolution d'un système de corps sous l'influence de forces de gravitation. C'est une simulation de N corps classique, où chacun des corps est modélisé par un point et une masse, et exerce des forces sur l'ensemble des autres corps du système. L'exécution se divise en étapes de calcul des forces suivie d'une mise à jour des positions des différents corps. Pour limiter les recherches des corps, l'algorithme utilise une représentation hiérarchique de l'espace sous la forme d'un arbre. Les étapes de l'exécution sont séparées par des barrières. Au début de chaque étape, les corps sont répartis selon les processeurs afin d'améliorer l'équilibrage de charge. Les simulations utilisent 1536 corps durant 11 étapes de simulation.

- **Cholesky** réalise la factorisation d'une matrice creuse. La répartition du travail entre les processeurs est dynamique et est réalisée à l'aide d'une file de travaux partagés protégée par des verrous d'exclusion mutuelle. Les simulations utilisent la matrice *bcsstk14* de taille 512x512.

- **Mp3d** simule les contraintes de pression et de température s'exerçant sur un objet volant à grande vitesse dans les couches hautes de l'atmosphère. Le programme simule le déplacement et les collisions d'un ensemble de molécules d'air dans un "tunnel" représenté par un ensemble de cellules. Les collisions peuvent se produire avec les autres molécules, le véhicule ou contre les bords du tunnel. L'algorithme est parallélisé en divisant statiquement les molécules suivant les processeurs qui se synchronisent à chaque étape de calcul. Nous utilisons 50000 particules dans un espace de taille 14x24x7 cellules durant 8 étapes de calcul.

- **Water** simule les interactions à l'intérieur d'un système de molécules d'eau à l'état liquide. La répartition des molécules entre les processeurs est réalisée statiquement. Des barrières synchronisent les étapes de calcul. Le temps d'exécution de cette application variant en $O(n^2)$ en nombre de molécules, les simulations sont lancées avec un ensemble de travail réduit variant de 120 à 144 molécules suivant le nombre de processeurs, pendant 2 pas de simulation.

Le choix des paramètres d'entrée des applications est essentiellement dicté par le temps nécessaire aux simulations. Les simulations les plus longues ($Mp3d$) prennent en moyenne 13 heures sur une station SUN SS10.

Les caractéristiques des applications utilisées sont présentées dans la table VII.1. Leur comportement vis à vis des caches est étudié notamment dans [Gupta & Weber 92].

Applications	Paramètres	Instructions (millions)	Lectures	Écritures	Lectures partagées	Écritures partagées
Barnes-Hut	1536 corps 11 itérations	190	49,5 (18,4%)	28,7 (10,7%)	11,1 (4,2%)	0,27 (0,1%)
Cholesky	bcsstk14	53,1	17.6 (23,3%)	4.7 (6,2%)	14,2 (18,8%)	2,5 (3,3%)
Mp3d	50 K molécules 8 pas	48,3	10,6 (16,3%)	6,3 (9,7%)	8,6 (13,1%)	5,4 (8,3%)
Water	120/144 molécules 2 itérations	78,6	26,8 (23,7%)	7,8 (6,9%)	4,9 (4,3%)	0,58 (0,5%)

Table VII.1 Caractéristiques des applications simulées

VII.2 Paramètres de simulation

Dans un souci de correction des simulations, les caractéristiques des architectures simulées s'inspirent de l'architecture KSR1 qui est à l'heure actuelle la seule machine COMA commercialisée.

VII.2.1 Caractéristiques d'un nœud

Chaque nœud est constitué d'un processeur, d'un cache primaire, d'une mémoire attractive et d'une interface réseau. Le processeur utilise une horloge à 20Mhz (50 ns de temps de cycle) et exécute une seule instruction par cycle [1].

Le cache de premier niveau est constitué d'un cache d'instructions et d'un cache de données chacun d'une taille de 256Ko. Dans les simulations, le cache d'instructions n'est

[1] Pour des raisons de simplicité des simulations nous n'avons pas considéré de processeurs superscalaires comme dans l'architecture KSR1.

pas considéré et nous supposons un taux de succès de 100% pour les instructions. Cette hypothèse est tout à fait réaliste étant donnée la taille du cache d'instructions. Le cache de données est associatif par ensembles de huit. Les lignes de cache font 64 octets mais l'allocation dans le cache utilise des pages de 2Ko. La politique de remplacement des pages est aléatoire (*random*). Le temps d'accès au cache primaire est de 1 cycle processeur[2].

Pour des raisons de place mémoire utilisée par le simulateur, la taille des mémoires attractives est fixée à 8Mo. Les lignes des MA font 128 octets et l'allocation dans la mémoire attractive utilise des pages de 16Ko (128 lignes). La latence d'une requête servie par une MA est de 18 cycles processeur. Les caches de premier niveau vérifient la propriété d'inclusion dans la mémoire attractive.

Le protocole de cohérence est celui décrit dans le chapitre précédent. Le protocole intérieur et le protocole extérieur sont mis en œuvre par des automates d'états finis.

VII.2.2 Réseau d'interconnexion

L'architecture étudiée utilise une organisation similaire à celle de l'architecture DASH. Deux réseaux en grille, un pour les requêtes et un pour les données, sont utilisés. Chacun de ces réseaux est similaire à celui décrit dans [Joe & Hennessy 94], ils utilisent des chemins de données de 32 bits pour les liens et un routage *wormehole* de type XY [NI & McKinley 93], c'est-à-dire suivant les dimensions. Les latences du réseau sont de 1 cycle (50ns) pour le transfert d'un *flit* (flow control digit) de 32 bits (soit un débit de 76 Mo/s entre deux nœuds pour chaque réseau). Une requête de 16 octets met donc 4 cycles pour aller d'un nœud à un autre et une donnée de 144 octets (128+16) 36 cycles. L'accès au répertoire pour sa mise à jour ou sa consultation prend 4 cycles.

Nous donnons dans la table VII.2 les temps de latence d'une requête de lecture servie à différents niveaux de la hiérarchie mémoire pour une architecture utilisant une grille 4x4. Pour calculer ces temps, les temps d'accès mémoire ont été fixés à 20 cycles en supposant la présence d'un bus mémoire de 64 bits, soit 4 cycles (200 ns) d'accès mémoire et 16 cycles pour transférer les données. Un saut correspond à la traversée de la grille. Une requête servie en un saut correspond donc à une requête où le nœud chargé du pointeur de localisation de la ligne est également propriétaire de la ligne demandée. Deux sauts sont nécessaires si ce nœud doit faire suivre la requête vers le nœud actuellement propriétaire de la ligne.

Les temps qui ont été fixés ici ne sont pas optimisés. Par exemple, il n'existe aucun parallélisme entre le traitement des requêtes et le transfert des données vers la mémoire d'un nœud. De plus, une requête interne et une requête externe ne peuvent être traitées simultanément. Il est à noter que la faible fréquence d'horloge utilisée pour les processeurs engendre des temps de latence longs en comparaison de ceux utilisés dans d'autres architectures (calculés en nombre de cycles processeur, ces temps sont généralement plus courts). A titre

[2]L'architecture réelle de la KSR1 a un temps d'accès au cache primaire de 2 cycles mais peut exécuter deux instructions par cycle.

Lecture	Latence (cycles processeur)
Accès locaux	
Cache de premier niveau	1
Mémoire Attractive locale	18
Accès distants	
Nœud distant (1 saut)	116
Nœud distant (2 sauts)	124

Table VII.2 Latences mémoire d'une opération de lecture (sans contention)

d'indication, ces temps sont au moins 3 fois plus longs que ceux utilisés pour l'architecture FLASH [Heinrich *et al.* 94].

VII.2.3 Traitement des injections

Une injection est réalisée en deux étapes. Une requête, utilisant l'anneau d'injection, est tout d'abord chargée de trouver un nœud acceptant l'injection. Une fois celui-ci trouvé, la ligne mémoire est directement envoyée vers ce nœud. Aucun pointeur n'est maintenu par les copies *Invalide-CK*.

Lors d'une injection de ligne, le temps de transfert d'une ligne de l'unité PE vers la mémoire du nœud acceptant l'injection prend un accès mémoire, soit 20 cycles. Ce temps de transfert ne perturbe cependant que le nœud réalisant le transfert. L'acquittement de l'injection est renvoyé 5 cycles après l'arrivée de la ligne sur le nœud (temps d'accès et de mise à jour du répertoire). Cette solution suppose donc la présence de tampons permettant de conserver les copies des lignes en cours d'injection [J.Kuskin *et al.* 94].

VII.2.4 Sauvegarde des points de récupération

Durant la première phase de l'algorithme d'établissement d'un nouveau point de récupération, un nœud attend toujours la fin d'une injection pour en commencer une autre. Il est supposé que l'architecture dispose d'un mécanisme matériel permettant d'accélérer la recherche des lignes modifiées. Le temps entre deux injections est donc suffisamment long pour qu'une ligne à injecter soit prête à être envoyée dès que l'injection précédente est terminée (cette supposition est vérifiée par les temps d'injection mesurés lors des simulations).

La seconde phase de cet algorithme réalise un test pour chacune des pages de la mémoire. Si la page est allouée, l'ensemble de ses lignes est testé. Un test prend un cycle et un cycle supplémentaire est nécessaire lorsque l'état doit être modifié. Il est supposé que, comme dans l'architecture KSR1, quatre unités indépendantes se partagent la gestion de la mémoire attractive [KSR 92].

VII.2.5 Allocation mémoire

Les simulations ne prennent pas en compte les temps d'allocation de pages dans les mémoires attractives. La politique suivie pour leur allocation varie suivant que la page est partagée ou privée. Les pages privées sont allouées sur le nœud qui les accèdent. Les pages partagées sont distribuées sur l'ensemble des nœuds de l'architecture suivant une politique de type *round-robin*. Dans le cas de l'architecture tolérante aux fautes, les répliques des pages sont allouées sur les nœuds suivants sur l'anneau d'injection. Cette allocation des pages répliquées en suivant l'anneau d'injection permet de limiter le temps d'établissement des points de récupération.

VII.3 Évaluation des surcoûts

Nous nous intéressons dans ce paragraphe à l'étude des surcoûts engendrés par l'utilisation du protocole de cohérence étendu ainsi que la sauvegarde de points de récupération dans l'architecture COMA étudiée.

Les simulations considèrent une architecture composée de 30 nœuds organisés en une grille 6x5. La fréquence de sauvegarde des points de récupération est un des paramètres les plus importants puisqu'il conditionne la dégradation de performance par rapport à une architecture standard. Pour notre architecture, c'est la fréquence des opérations non récupérables (entrée-sortie) qui fixe la fréquence d'établissement des points de récupération. Ne connaissant pas a priori la fréquence de telles opérations, la dégradation de performance est mesurée en faisant varier cette fréquence de 400 à 5 points de récupération par seconde.

De nombreuses évaluations de mécanismes de récupération arrière ne prennent pas en compte les opérations d'entrée-sortie. Les évaluations réalisées supposent alors des fréquences d'établissement de points de récupération beaucoup plus faibles (un point de récupération toutes les deux minutes dans [Carter *et al.* 93]) qui permettent alors de limiter la dégradation de performance mesurée. Appliqués à notre architecture, de tels taux donneraient bien évidemment des résultats similaires. Ces suppositions limitent cependant les applications considérées, puisque seules des applications ayant des fréquences d'entrée-sortie très faibles sont envisageables.

Dans les simulations, les dates d'établissement des points de récupération sont fixées à l'aide d'une loi exponentielle de moyenne la fréquence fixée.

VII.3.1 Surcoûts temporels

Le premier surcoût engendré par le protocole de cohérence est un surcoût temporel. Une application s'exécutant sur une architecture utilisant le protocole de cohérence étendu et

sauvegardant des points de récupération voit son temps d'exécution augmenter par rapport à une exécution sur la même architecture utilisant un protocole de cohérence standard.

Le temps d'exécution d'une application utilisant le protocole de cohérence étendu peut s'exprimer comme la somme de quatre termes soit :

$$T_{protocole-etendu} = T_{standard} + T_{phase1} + T_{phase2} + T_{pollution} \quad (1)$$

où $T_{standard}$ représente le temps d'exécution de l'application sur l'architecture standard, T_{phase1} et T_{phase2} représentent la somme des temps passés respectivement dans la première phase (transfert des lignes mémoire) et la seconde phase (modification de l'état des lignes) de l'algorithme d'établissement de points de récupération, et $T_{pollution}$ le surcoût engendré par l'augmentation du nombre de défauts de mémoires attractives et l'augmentation du nombre d'injections de lignes. La figure VII.2 donne ces surcoûts pour les différentes applications utilisées, en fonction de la fréquence de sauvegarde des points de récupération. Nous présentons maintenant une étude détaillée des différents surcoûts engendrés par le protocole de cohérence étendu.

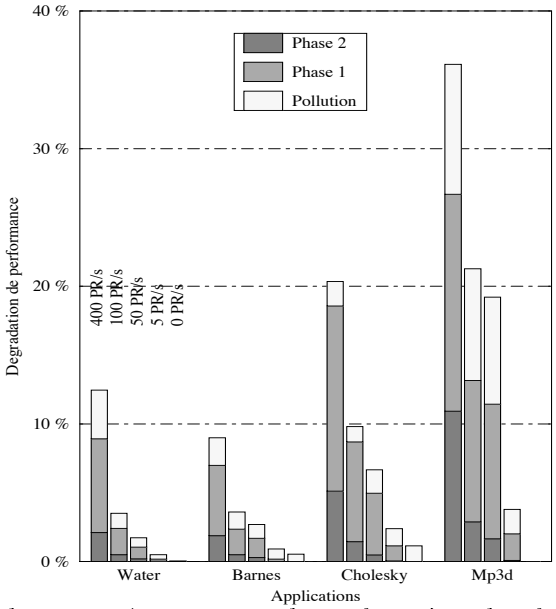


Figure VII.2 Variation des surcoûts temporels en fonction des fréquences de sauvegarde des points de récupération

VII.3.1.1 Étude de la phase 1

Il est difficile de quantifier le surcoût exact lié à cette phase car elle dépend de nombreux paramètres. Les valeurs données sur la figure VII.2 ne donnent que des exemples valables uniquement pour les applications et les paramètres d'entrée qu'elles utilisent.

Cette phase constitue généralement le principal surcoût engendré par le protocole de cohérence étendu. Suivant les applications et les fréquences de sauvegarde de points de

récupération, elle représente un surcoût variant de 1 à 2 % dans le meilleur des cas, pour atteindre plus de 15% dans le pire des cas ($Mp3d$ et 400 points de récupération par seconde). Son importance dépend essentiellement de la quantité de données de récupération à transférer lors des sauvegardes des points de récupération. Cette quantité de données dépend elle-même de plusieurs facteurs :

(1) **Fréquence de sauvegarde des points de récupération.** Pour toutes les applications, le surcoût engendré par la phase 1 diminue avec la fréquence de ces opérations. Cette diminution s'explique par une baisse de la quantité globale de données de récupération transférées. Les fréquences plus faibles permettent aux lignes d'être modifiées plusieurs fois avant un point de récupération. Au contraire les fréquences élevées demandent un transfert fréquent de ces lignes. La figure VII.3 présente la variation de la taille des données de récupération, pour 10 000 références mémoire, en fonction des fréquences de sauvegarde des points de récupération. Dans le cas de $Cholesky$, la quantité de données de récupération est divisée par 8 entre une fréquence de 400 et une fréquence de 5 points de récupération par seconde. Globalement, la quantité de données de récupération traitée passe alors de 10 Mo à 1,2 Mo.

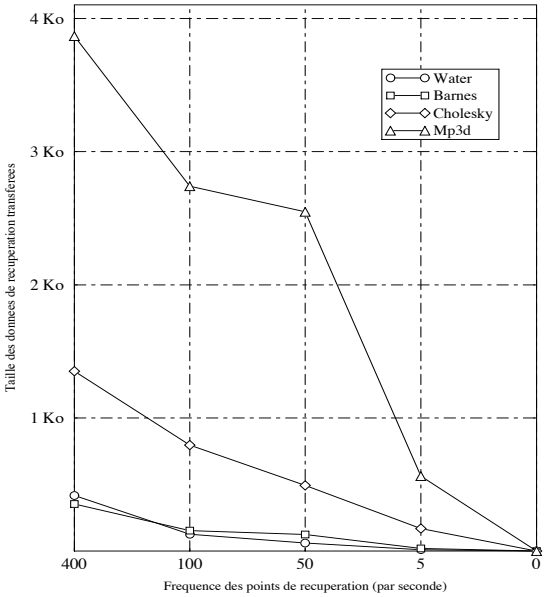


Figure VII.3 Taille des données de récupération, par processeur, pour 10 000 références mémoire

(2) **Taux de modification des applications.** Le taux de modification des applications est un autre critère qui intervient dans la quantité de données de récupération à traiter lors des sauvegardes des points de récupération. $Mp3d$ par exemple possède un taux d'écriture important (10%) qui contribue ainsi à lui donner une quantité de données de récupération traitées élevée (près de 4Ko par processeur pour 10000 références mémoire à 400 points de récupération par seconde).

(3) **Taille de l'espace de travail.** La taille de l'espace de travail utilisé par l'application intervient également dans la quantité de données de récupération à traiter. Plus l'espace de travail d'une application est important, plus elle a la possibilité de modifier de lignes mémoires entre deux points de récupération. Ainsi les différences de surcoûts engendrés par la phase 1 pour *Mp3d* et *Barnes*, qui ont des taux de modification de données équivalents, peuvent s'expliquer par la différence de l'espace mémoire partagé utilisé par ces applications. *Mp3d* utilise ainsi approximativement 3,7 Mo de données partagées avec 50000 molécules alors que *Barnes* en utilise seulement 421 Ko.

(4) **Localité des références mémoire.** Finalement, la localité des accès mémoire est le dernier facteur intervenant sur la quantité de données modifiées entre deux points de récupération. Les applications possédant un fort taux de localité ne modifient que peu de données. Au contraire une application telle que *Mp3d* dans laquelle le partage de données est intense et les taux de défaut des mémoires attractives atteignent 10% (essentiellement des défauts de cohérence), risque de modifier une grande quantité de lignes mémoire.

Recherche d'une borne maximale. La quantité de données de récupération et donc la dégradation de performance engendrée par la phase 1, dépend de nombreux facteurs. Il semble donc difficile de donner une borne maximale à cette valeur. Dans le but d'approcher cette borne, la quantité de données modifiées générée entre deux points de récupération est mesurée pour l'application *Mp3d*, à l'aide d'un simulateur simplifié. Nous donnons également une estimation de la dégradation de performance correspondante. Le choix de l'application *Mp3d* vient du fait que cette application rassemble à elle seule tous les critères nécessaires pour générer une quantité de données de récupération importante.

Les courbes VII.4 et VII.5 résument les résultats obtenus pour les différentes fréquences de sauvegarde de points de récupération et les différentes tailles de problème (de 10000 à 100000 molécules). Il ressort de cette étude que la quantité de données de récupération générée entre deux points de récupération tend vers une asymptote (celle-ci n'est pas atteinte à 5 points de récupération par seconde). Ce comportement est normal car une application réalise au plus un nombre borné de modifications uniques entre deux instants. La dégradation engendrée par la phase 1 tend donc elle aussi vers une asymptote. Il apparaît également que cette valeur asymptotique est atteinte pour l'application *Mp3d* même avec une taille de problème réduite à 50000 molécules. L'application *Mp3d* correspondant à un cas d'application très défavorable, la dégradation de performance engendrée par la phase 1 pour cette application, donne donc une bonne approximation d'une borne maximale (sauf à 5 points de récupération par seconde où l'asymptote n'est pas atteinte).

Autres facteurs. D'autres facteurs, plus dépendants de l'architecture utilisée, influencent également la durée de la phase 1. Ainsi l'utilisation du réseau d'interconnexion et des mémoires attractives pour le transfert et le stockage des données de récupération est en grande

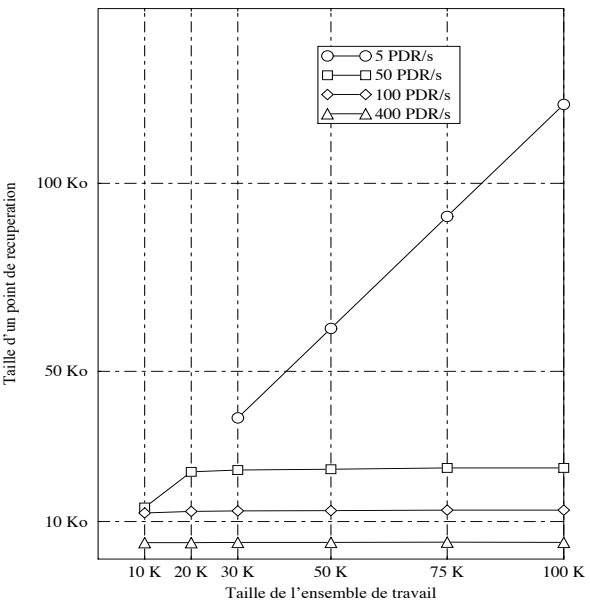


Figure VII.4: Variation de la taille des points de récupération en fonction de la taille de l'ensemble de travail (application $Mp3d$)

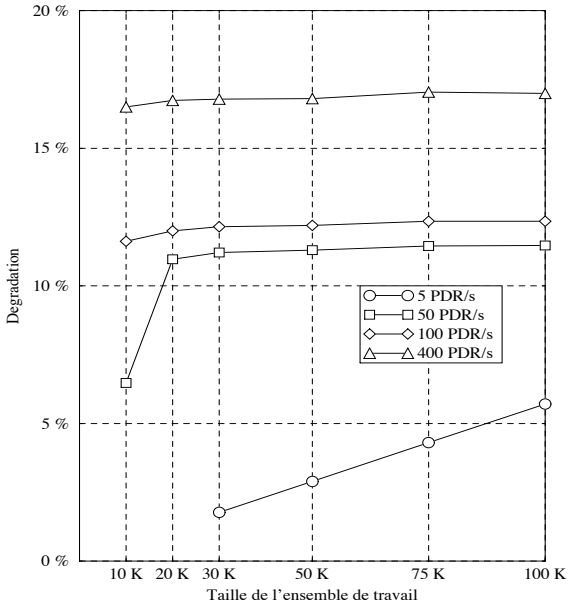


Figure VII.5: Variation de la dégradation de performance engendrée par la phase 1 pour différentes tailles d'ensemble de travail (application $Mp3d$)

partie responsable de la faible dégradation de performance engendrée par cette phase malgré les fréquences élevées de sauvegarde des points de récupération. Elles assurent en effet, à chaque nœud de l'architecture, un débit élevé de réplication des données modifiées. Celui-ci se situe aux alentours de 20 Mo/s par nœud pour l'architecture étudiée soit globalement avec 30 nœuds, un débit de 600 Mo/s. Cette propriété permet de garantir des temps de création de points de récupération courts qui limitent la dégradation de performance et permettent de supporter des sauvegardes de points de récupération fréquentes.

L'architecture étudiée bénéficie d'un autre avantage. Grâce à l'utilisation de l'information contenue dans les entrées de répertoires, elle peut en effet utiliser la réplication de données existante dans l'architecture pour limiter les transferts de données. Ainsi lorsqu'une seconde copie d'une ligne modifiée dans une région de récupération, existe déjà, une simple requête est suffisante lors du traitement de la réplication de cette ligne. Bien entendu cette propriété n'est intéressante que dans le cas d'applications utilisant beaucoup d'objets partagés en lecture (*mostly-read* objects [Weber 93]). Parmi les quatre applications utilisées seule *Barnes* utilise ce type d'objet de façon fréquente. Les autres applications utilisent généralement des objets de type **migratoire**, manipulés, à un instant, par un seul et unique processeur. Lors des phases de réplication ces applications ont donc une majorité de lignes *Exclusif* à répliquer (plus de 99% dans le cas de $Mp3d$). Pour Barnes l'impact de cette propriété augmente lorsque la fréquence des sauvegardes des points de récupération baisse car la proportion de lignes modifiées et répliquées augmente aussi. Ainsi à 400 points de récupération par seconde, seulement 4,7% des lignes à répliquer ont déjà une copie dans l'architecture alors

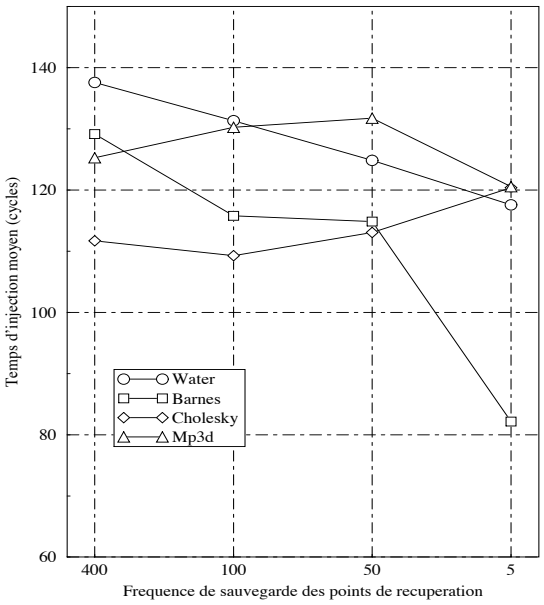


Figure VII.6: Temps moyen de réplication d'une ligne

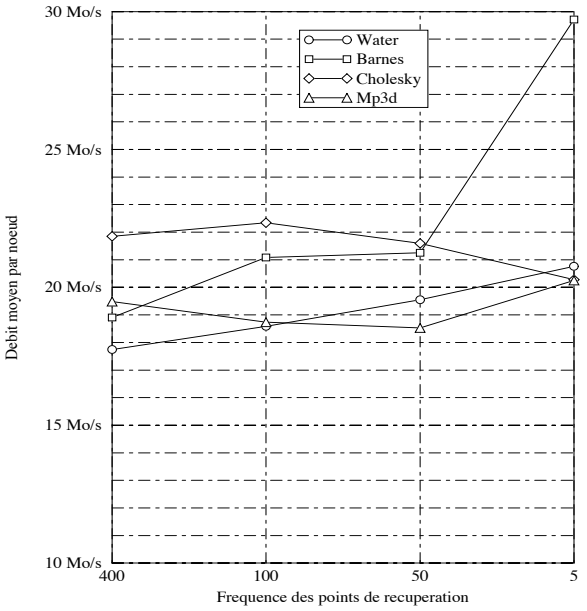


Figure VII.7: Débit de réplication moyen des données modifiées sur un nœud

qu'à 5 points de récupération par seconde, cette proportion passe à 52%. Le temps moyen de réplication d'une ligne passe alors de 130 à 82 cycles (figure VII.6). Cette diminution du temps moyen de réplication des lignes partagées permet d'augmenter le débit de traitement des données à répliquer qui atteint alors presque 30 Mo/s (figure VII.7).

VII.3.1.2 Étude de la phase 2

La durée de la phase 2 ne dépend que du nombre de pages allouées sur les nœuds de l'architecture. La dégradation qu'elle engendre est donc directement proportionnelle à la taille de l'espace de travail et à la fréquence de sauvegarde des points de récupération. Les applications *Mp3d* et *Cholesky* possédant les espaces de travail les plus grands, il est normal qu'elles connaissent aussi une dégradation de performance plus importante causée par cette phase. Sans précautions particulières, cette phase peut devenir la cause principale de dégradation de performance lorsque les mémoires sont grandes et les fréquences de sauvegarde des points de récupération élevées. Avec des fréquences de sauvegarde de points de récupération faibles, le coût de cette phase est négligeable pour toutes les applications.

Certaines optimisations peuvent cependant être envisagées. Par exemple, il serait intéressant de ne tester que les pages modifiées dans la région de récupération. L'utilisation d'un dispositif matériel similaire à celui présenté pour identifier les lignes mémoires modifiées pourrait aussi être envisagée. Le chapitre VIII propose une solution permettant d'éviter le parcours de la mémoire nécessaire pendant cette phase.

VII.3.1.3 Étude de la pollution

L'introduction des données de récupération dans les mémoires de l'architecture entraîne bien évidemment un surcoût qui se caractérise par deux effets : (i) une variation des taux de défauts de cache et de mémoire attractive, (ii) une création de nouvelles injections de lignes. Ces deux effets peuvent être conjugués lorsque la présence d'une copie de récupération sur un nœud nécessite une injection suivie d'un défaut sur la copie courante associée (accès en lecture ou en écriture sur une copie *Invalide-CK*). Suivant les fréquences de sauvegarde des points de récupération, le surcoût engendré par la pollution varie de 10% à moins de 2%, il reste donc limité.

Variation des taux de défauts. À chaque sauvegarde d'un point de récupération, les données modifiées des caches sont recopiées en mémoire. La première augmentation du nombre des défauts intervient donc dans les caches des nœuds. Le graphe VII.8 donne cette augmentation pour les différentes fréquences de sauvegarde des points de récupération. Le comportement global indique une légère augmentation des défauts en écriture alors que les défauts en lecture restent stables. Ce comportement est tout à fait normal car les lignes modifiées, recopiées en mémoire lors de sauvegardes de points de récupération, restent accessibles en lecture dans le cache. Le nombre de défauts en lecture reste donc stable alors que celui des défauts en écriture augmente avec la fréquence de sauvegarde des points de récupération. En proportion cette augmentation est importante pour les applications qui génèrent peu de défauts. À 400 points de récupération par seconde, *Water* multiplie par 2,6 le nombre de défauts en écriture. Au contraire, elle reste faible pour des applications générant déjà beaucoup de défauts. À 400 points de récupération par seconde, *Mp3d* multiplie par 1,08 le nombre de défauts en écriture. Son effet sur les performances reste donc limité.

Le graphe de la figure VII.9 donne le taux de défauts moyen d'un nœud, en fonction de la fréquence de sauvegarde des points de récupération. La constatation qui s'impose est que la variation des taux de défauts par nœud est négligeable et ceci quelle que soit la fréquence de sauvegarde des points de récupération. La constance du nombre de défauts en lecture confirme un des avantages du protocole qui autorise la lecture des données de récupération non modifiées. Pour *Barnes*, à 400 points de récupération par seconde, le nombre de lectures de lignes *Partagé-CK* représente 33% des requêtes de lecture en provenance du cache. Dans certains cas, le nombre de défauts peut même diminuer avec l'augmentation des fréquences des points de récupération si l'application tire parti de la réplication réalisée lors de la création des nouveaux points de récupération. Ainsi *Water* passe d'un taux de défauts en lecture de 1.13% dans la version standard à un taux de 1.09% à 400 points de récupération par seconde. La faible variation du taux de défauts en écriture s'explique par le fait que les applications utilisent le plus souvent des objets migratoires qui auraient de toutes façons généré des défauts en écriture.

L'augmentation du nombre de défauts ne justifie pas l'effet de pollution mesuré sur les courbes de la figure VII.2. La dégradation de performance engendrée par le stockage des

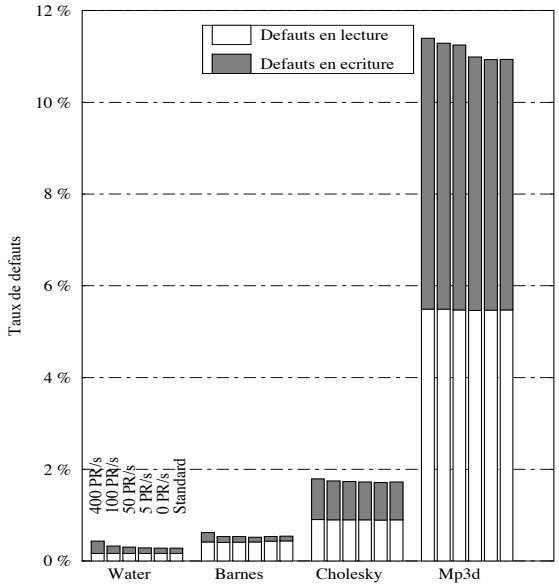


Figure VII.8: Variation des taux de défauts dans les caches primaires en fonction de la fréquence de sauvegarde des points de récupération

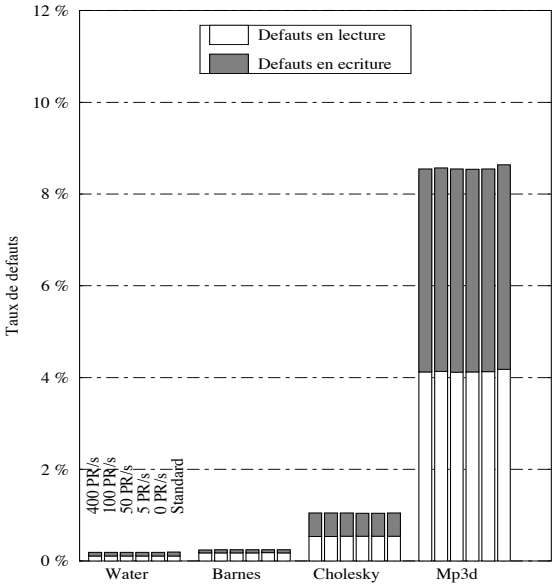


Figure VII.9: Variation des taux de défauts par nœud en fonction de la fréquence de sauvegarde des points de récupération

données de récupération dans les mémoires est en fait essentiellement dûe aux nouvelles injections de lignes.

Injections de lignes. Le graphe de la figure VII.10 donne la variation du nombre d'injections par mémoire attractive pour 10000 références mémoire en fonction des fréquences de sauvegarde des points de récupération. La première constatation est que le nombre global d'injection est faible, il atteint au maximum 25 injections pour 10000 références mémoire. Il apparaît de plus que le nombre d'injections sur lecture varie peu avec la fréquence de sauvegarde des points de récupération. Ici aussi, ce comportement s'explique par le fait que le protocole de cohérence étendu autorise la lecture des données de récupération non modifiées. Au contraire, le nombre d'injections sur écriture augmente de façon importante avec la fréquence de sauvegarde des points de récupération. La transformation fréquente de données courantes modifiées en données de récupération explique ce comportement. À 400 points de récupération par seconde, le nombre d'injections causées par l'accès en écriture à des copies *Partagé-CK1* représente, suivant les applications, de 88 à 98 % du nombre total des injections sur écriture d'un nœud. Ce type d'injection constitue donc la principale cause de l'effet de pollution mesuré. Le faible nombre d'injections en lecture ou en écriture lorsque les fréquences de sauvegarde des points de récupération sont plus faibles, indique que la politique d'injection utilisée est efficace pour placer les données de récupération sur des nœuds où elles ne gênent pas les calculs. Ces résultats confirment l'utilité de l'anneau d'injection.

Les différences du nombre d'injections entre les applications proviennent essentiellement des taux de communications des applications. Des applications telles que *Water* ou *Barnes*, où la localité des accès mémoire est importante et où les taux de défauts sont faibles, vont générer peu d'injections. Au contraire, des applications telles que *Mp3d*, où le partage de données est intense, génèrent beaucoup plus de mouvements de lignes entre les nœuds et ont donc une grande probabilité d'avoir à injecter des lignes. À ce titre, cette application fournit une bonne idée d'une borne maximale de l'effet de pollution.

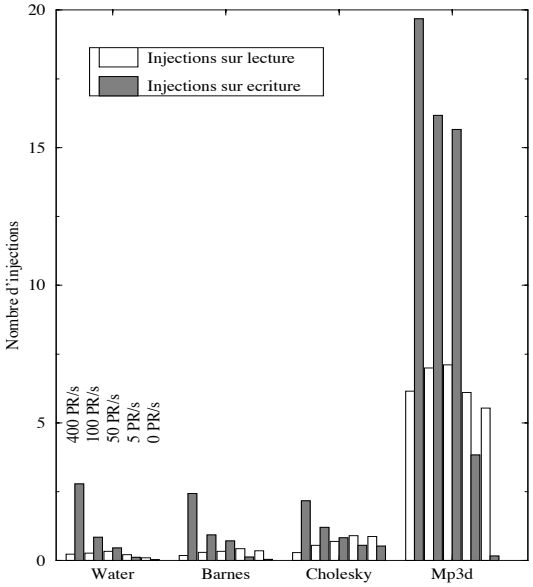


Figure VII.10: Nombre d'injections sur lecture et sur écriture pour 10000 références en fonction de la fréquence de sauvegarde des points de récupération

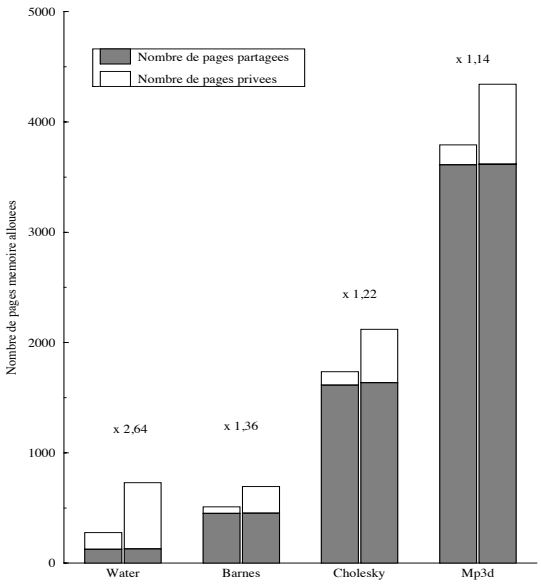


Figure VII.11: Comparaison du nombre de pages allouées par une architecture utilisant un protocole de cohérence standard et une architecture utilisant le protocole de cohérence étendu

VII.3.1.4 Conclusion

L'évaluation réalisée a montré que les surcoûts temporels engendrés par le protocole de cohérence étendu restent limités et dépendent directement de la fréquence de sauvegarde des points de récupération. L'utilisation du réseau d'interconnexion de la machine et des mémoires pour assurer une réplication rapide des données modifiées permet de limiter la durée de la phase 1 de la sauvegarde d'un point de récupération. Celle-ci bénéficie également de la réplication de donnée existante pour limiter les transferts de données. La phase 2 dépend essentiellement des moyens utilisés pour son implémentation. Une recherche séquentielle peut être coûteuse. L'utilisation de matériel spécifique peut permettre de limiter sa durée. L'effet de pollution engendré par le stockage des données de récupération dans les mémoires attractives est essentiellement causé par les injections sur écriture lors d'accès à des copies

Partagé-CK1. Grâce à la politique d'injection utilisée, les injections de copies *Invalide-CK* représentent en effet une faible proportion des injections. La taille importante des pages mémoire, qui favorise le faux partage, est en partie responsable du faible nombre d'injections de ce type de copies. De plus, en autorisant l'accès en lecture aux données de récupération non modifiées, le protocole de cohérence étendu assure une augmentation négligeable du nombre de défauts par nœud.

VII.3.2 Surcoût spatial

Le second surcoût engendré par le protocole de cohérence étendu est une occupation mémoire plus importante dûe au stockage des données de récupération. Le graphe de la figure VII.11 compare le nombre de pages mémoire allouées pour une architecture utilisant un protocole de cohérence standard avec celui obtenu pour une architecture utilisant le protocole de cohérence étendu.

Suivant les applications, ce surcoût varie de 1,1 à 2,6 fois le nombre de pages. Les pages contenant des données partagées ne produisent quasiment aucun surcoût et ce, même si quatre copies sont systématiquement allouées avec le protocole de cohérence étendu. Ce résultat s'explique principalement par la taille des pages (16 Ko) qui favorise le faux partage et donc la réplication des pages sur les nœuds. Il montre que pour les pages partagées, le protocole est capable d'utiliser la réplication mémoire déjà existante pour stocker les données de récupération partagées sans engendrer de surcoût mémoire. Les pages contenant des données privées, allouées sur un seul nœud avec un protocole standard, voient évidemment leur nombre multiplié par quatre. Globalement, les applications utilisant majoritairement des pages partagées ont donc le surcoût mémoire le plus faible. *Mp3d*, *Barnes* et *Cholesky* nécessitent une quantité de mémoire inférieure à 1,5 fois celle allouée en utilisant un protocole de cohérence standard.

Il est à noter que ces résultats correspondent à une situation particulièrement favorable. En effet la taille limitée des applications utilisées a pour conséquence de ne jamais nécessiter de remplacement de page dans les mémoires de l'architecture. Une page allouée sur un nœud reste donc sur ce nœud jusqu'à la fin de l'exécution. Dans le cas d'une page partagée, les quatre copies nécessaires au protocole de cohérence étendu sont donc généralement allouées avec l'architecture utilisant un protocole standard. Le surcoût spatial lié à l'utilisation du protocole de cohérence étendu est donc quasiment nul pour ce type de page. Il est clair qu'avec des applications de taille plus importante, engendrant des remplacements de pages et n'assurant pas que les copies nécessaires au protocole de cohérence étendu soient déjà allouées avec un protocole de cohérence standard, le surcoût spatial engendré peut être plus important. La quantité de mémoire nécessaire au protocole de cohérence étendu reste cependant difficile à évaluer car elle dépend du comportement des applications. La capacité du protocole de cohérence étendu à utiliser les espaces mémoire alloués par le vrai ou le faux partage de données laisse penser que cette quantité sera inférieure à 4 fois la quantité de mémoire utilisée avec un protocole de cohérence standard.

VII.4 Étude d'extensibilité

Pour le type d'architecture considéré dans cette étude, l'extensibilité reste un des objectifs primordiaux à assurer. L'introduction de mécanismes de tolérance aux fautes ne doit pas être la cause d'un manque d'extensibilité de l'architecture finale. Nous évaluons dans ce paragraphe l'extensibilité de notre proposition en faisant varier le nombre de nœuds de 9 à 56 et en mesurant les surcoûts engendrés avec une fréquence de points de récupération fixée à 100 par seconde. Nous présentons respectivement le surcoût engendré par la réplication de données (phase 1) puis par la pollution des mémoires attractives. Le surcoût engendré par la phase 2 de l'algorithme d'établissement des points de récupération n'est pas présenté car il ne dépend pas du nombre de processeurs de l'architecture.

VII.4.1 Phase 1

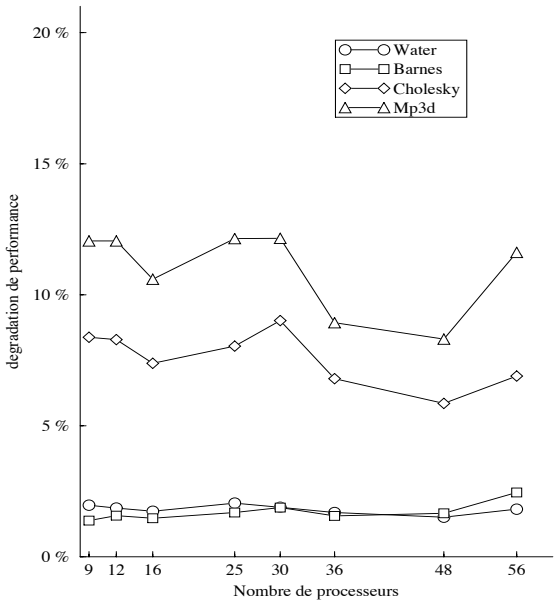


Figure VII.12: Variation du coût de la phase 1 en fonction du nombre de processeurs

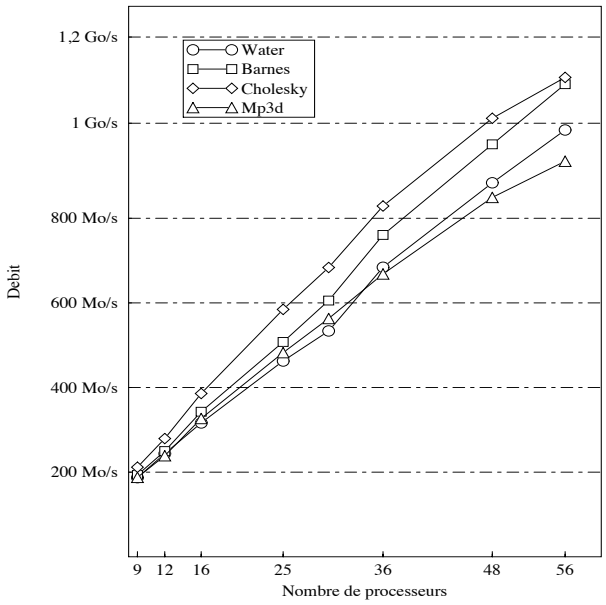


Figure VII.13: Débit global de réplication des données de récupération en fonction du nombre de processeurs

Le graphe de la figure VII.12 présente la variation du surcoût engendré par la phase 1 en fonction du nombre de processeurs. Les résultats montrent que la dégradation de performance engendrée par cette phase reste constante voire diminue avec le nombre de processeurs. Ce comportement a deux explications. La première est qu'avec des applications de taille fixe et une augmentation du nombre de processeurs, la quantité de données de récupération traitée par processeur, à chaque point de récupération, diminue. Ainsi pour *Mp3d*, cette taille passe de 9,6 Ko avec 30 processeurs, à 6,8 Ko avec 56 processeurs.

La seconde explication, qui est aussi le facteur le plus intéressant, est une augmentation pratiquement linéaire du débit de réplication des données de récupération (voir figure VII.13). Dans le cas de *Cholesky*, le débit global passe de 211 Mo/s avec 9 processeurs à plus de 1,1 Go/s avec 56 processeurs. Cette propriété garantit l'extensibilité de l'architecture car la dégradation de performance engendrée par la phase 1, qui dépend essentiellement des débits de réplication des données de récupération, reste pratiquement constante quand le nombre de processeurs augmente. Une propriété similaire serait difficile à assurer en utilisant des disques pour stocker les données de récupération, à moins d'augmenter leur nombre avec celui des nœuds.

Pour certaines applications, l'augmentation du nombre de processeurs entraîne une légère augmentation des temps de réplication des lignes partagées. Les débits moyens de réplication par nœud restent cependant supérieurs à 16 Mo/s.

VII.4.2 Pollution

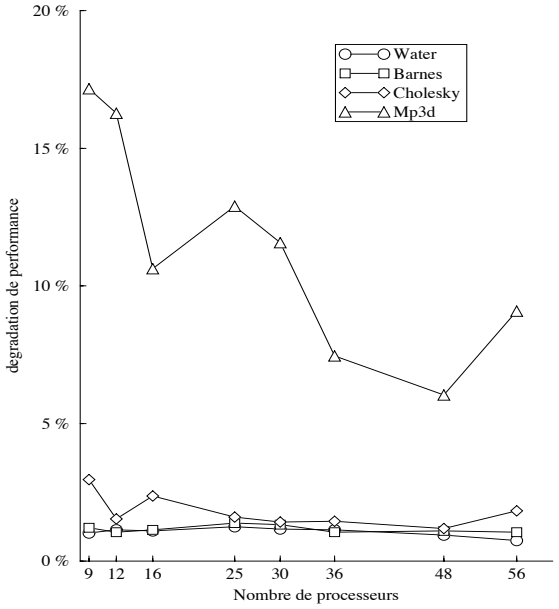


Figure VII.14: Variation de l'effet de pollution en fonction du nombre de processeurs

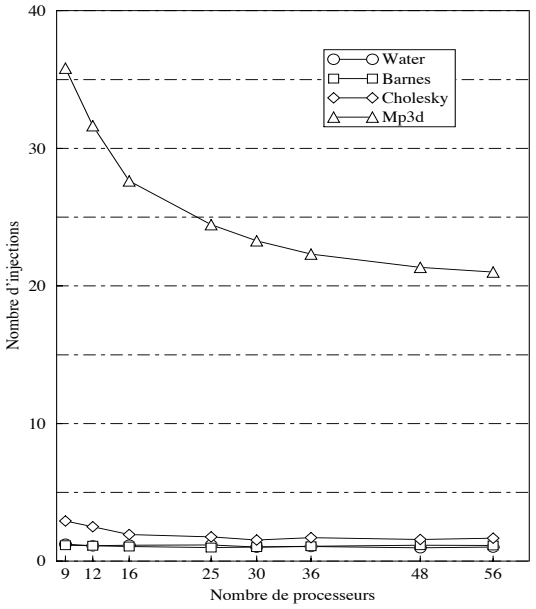


Figure VII.15: Nombre moyen d'injections (sue écriture et sur lecture) par nœud pour 10 000 références mémoires

Sur le graphe de la figure VII.14 est représentée la variation de l'effet de pollution en fonction du nombre de processeurs. Dans ce cas également, l'extensibilité de l'architecture est préservée car ce surcoût reste constant ou diminue avec le nombre de processeurs. La raison principale de ce comportement apparaît sur la figure VII.15. qui donne le nombre moyen d'injections par nœud. Ce nombre soit reste stable, lorsque sa valeur est déjà faible, soit diminue avec le nombre de processeurs lorsque sa valeur est déjà élevée ($Mp3d$). Alors que le

nombre d'injections sur écriture reste pratiquement constant car celles-ci sont essentiellement causées par des accès en écriture sur des copies *Partagé-CK1*, le nombre d'injections sur lecture diminue avec l'augmentation du nombre de processeurs. Dans le cas de *Mp3d*, le nombre d'injections sur lecture, pour 10000 références mémoire, passe ainsi de 16,6 avec 9 processeurs, à 3,3 avec 56 processeurs. Cette diminution est causée par les lignes partagées qui, en bénéficiant d'un nombre de copies de pages plus important lorsque le nombre de processeurs augmente, ont plus de chance d'occuper un emplacement mémoire non utilisé.

VII.4.3 Conclusion

Pour une architecture non-hiérarchique, l'utilisation du protocole de cohérence étendu et des mémoires attractives, pour le stockage des données de récupération, ne remet pas en cause l'extensibilité de l'architecture. L'augmentation du nombre de mémoires et de la bande passante du réseau assure un débit moyen de réplication des données de récupération par nœud qui reste élevé. L'augmentation du nombre de mémoires assure également un effet de pollution plus faible en tirant parti du nombre de copies de pages partagées plus important.

VII.5 Impact de l'augmentation de la fréquence d'horloge

Les caractéristiques de l'architecture étudiée jusqu'à présent sont peu optimistes. Notamment la fréquence d'horloge de 20 Mhz semble bien lente comparée à celle de certains multiprocesseurs existants. Par exemple l'architecture T3D de Cray [Koeninger *et al.* 94] utilise une fréquence de base de 150 Mhz pour ses processeurs et son réseau.

Dans ce paragraphe, nous nous intéressons à une architecture beaucoup plus agressive du point de vue de ses caractéristiques. L'organisation physique de l'architecture reste identique, mais l'horloge de base de l'architecture passe à 100 Mhz (10 ns de temps de cycle). Le réseau d'interconnexion reste une grille. Cependant, à cette fréquence, une grille unique est utilisée pour les requêtes et les réponses. Le temps de transfert sur un lien de communication est fixé à un cycle, ce qui représente un débit par lien de 400 Mo/s. Ce réseau possède les caractéristiques du réseau d'interconnexion proposé pour l'architecture FLASH [Heinrich *et al.* 94]. Les temps d'accès à différents niveaux de la hiérarchie mémoire sont résumés dans la table VII.3. Pour calculer ces temps, les temps d'accès mémoire sont fixés à 16 cycles soit 160 ns.

La figure VII.16 présente la dégradation de performance mesurée pour les différentes applications et fréquences de sauvegarde des points de récupération. Par rapport à l'architecture précédente, la dégradation de performance diminue pour toutes les applications. L'explication de ce comportement se trouve essentiellement dans l'augmentation de la fréquence d'horloge de l'architecture qui permet de réaliser plus de calcul entre deux points

Lecture	Latence (cycle = 10 ns)
Accès locaux	
Cache de premier niveau	1
Mémoire Attractive locale	30
Accès distants	
Nœud distant (1 saut)	192
Nœud distant (2 sauts)	203

Table VII.3 Latences mémoire d'une opération de lecture, sans contention, dans une architecture 4x4 à 100 Mhz

de récupération. À fréquence de sauvegarde de points de récupération égale, l'architecture à 100 Mhz a le temps d'exécuter 5 fois plus de cycles, entre deux points de récupération, que l'architecture à 20 Mhz. À 400 points de récupération par seconde, elle devrait donc, du point de vue de la dégradation de performance, être équivalente à l'architecture à 20 Mhz et 80 points de récupération par seconde. Les mesures réalisées montrent qu'il n'en est rien. Les dégradations de performance mesurées pour l'architecture à 100 Mhz et 400 points de récupération par seconde correspondent grossièrement à la dégradation de performance mesurée avec l'architecture à 20 Mhz et une fréquence de 250 points de récupération par seconde.

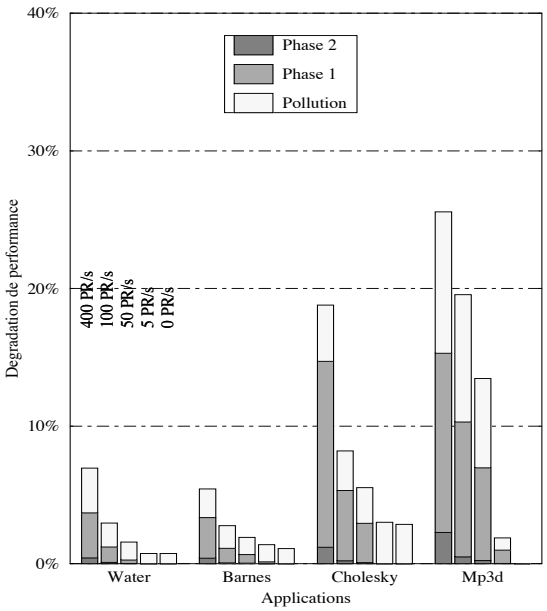


Figure VII.16: Dégradation de performance mesurée pour l'architecture à 100 Mhz

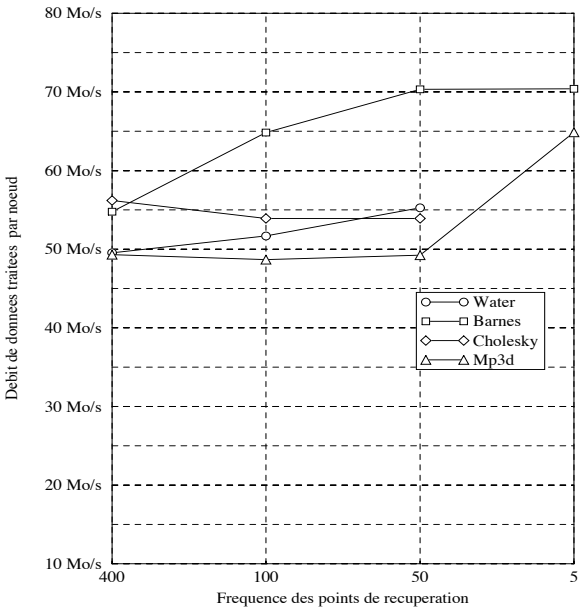


Figure VII.17: Débit moyen de traitement des données de récupération pour l'architecture à 100 Mhz

La raison principale de ce comportement se trouve dans l'augmentation, en nombre de cycles, des temps d'injection de lignes. Cette augmentation s'explique par une hausse, en

nombre de cycles, des temps d'accès à la mémoire, ainsi que par l'utilisation d'un seul réseau d'interconnexion. Ces temps varient ainsi entre 180 et 250 cycles pour la nouvelle architecture alors qu'il se situaient autour des 130 cycles pour l'architecture à 20 Mhz. Les débits de traitement des données de récupération ont donc bien augmenté, comme l'indique la figure VII.17, sans toutefois être multipliés par 5. Ce comportement explique également pourquoi l'importance de la dégradation de performance engendrée par la phase 1 a augmenté avec l'architecture à 100 Mhz.

VII.6 Validité de l'évaluation

Il convient toujours d'être prudent avec des résultats de simulation car leur extrapolation est délicate. Dans le cas de l'évaluation réalisée dans ce chapitre, deux aspects de l'architecture ne sont pas considérés. Tout d'abord l'absence de prise en compte d'un système opératoire dans les simulations ne permet pas de donner la dégradation de performance réelle. En effet, le système opératoire risque d'augmenter les quantités de données de récupération en modifiant certaines structures de données ou en mettant à jour des pages lors de leur création [Torrellas *et al.* 92]. De plus, en raison de la taille réduite des applications et de la taille des mémoires attractives simulées, aucun remplacement de page n'est réalisé. Cet état de fait a deux conséquences. D'une part, il inhibe les injections de copies de récupération qu'il serait nécessaire de réaliser lorsqu'une page contenant des données de récupération est remplacée. D'autre part, les pages partagées restant allouées sur de multiples nœuds et les emplacements mémoire alloués par le faux partage étant nombreux, il minimise le nombre d'injections de copies de récupération. L'utilisation d'applications de taille plus importante risque donc d'augmenter l'effet de pollution et donc la dégradation de performance mesurée. Une solution à ce problème consiste à augmenter la taille des mémoires pour limiter ces effets. Le coût entraîné reste cependant peu élevé pour une architecture tolérante aux fautes.

D'autres aspects des mesures peuvent cependant être considérés comme donnant une bonne approximation de ce que l'on obtiendrait avec une architecture réelle. Ainsi les débits de traitement des données de récupération semblent tout à fait réalistes. Ils permettent d'envisager des sauvegardes de points de récupération rapides et fréquentes. De même, le faible effet de pollution mesuré donne une bonne idée de ce que l'on pourrait obtenir avec une architecture réelle.

VII.7 Résumé

L'évaluation de performance présentée dans ce chapitre a permis de mettre en évidence les propriétés suivantes du protocole de cohérence étendu :

(1) Le surcoût créé par la réplication de données lors de la sauvegarde des points de récupération dépend essentiellement de la fréquence de sauvegarde des points de récupération.

Son effet reste cependant limité grâce à l'utilisation des mémoires et du réseau d'interconnexion qui assure des débits élevés de réplication des données de récupération. Dans le cas d'une architecture utilisant un protocole de cohérence à base de répertoire, le protocole de cohérence étendu permet de plus de tirer parti de la réplication de données existante pour limiter les transferts de données et augmenter ces débits.

(2) L'effet de pollution engendré par l'introduction des données de récupération dans les mémoires des nœuds reste faible. La lecture des données de récupération non modifiées rend les taux de défauts des nœuds pratiquement indépendants de la fréquence de sauvegarde des points de récupération. L'effet de pollution est essentiellement causé par les injections de lignes, en particulier les injections sur écriture dont le nombre augmente avec cette fréquence.

(3) Pour les pages partagées, l'utilisation des espaces mémoire alloués par le vrai et le faux partage de données permet au protocole de cohérence étendu de limiter le surcoût spatial engendré par la conservation et la réplication des données de récupération.

(4) L'utilisation du protocole de cohérence étendu au sein d'une architecture non hiérarchique ne constitue pas un obstacle à l'extensibilité de cette architecture.

(5) L'augmentation de la fréquence d'horloge de l'architecture permet de diminuer la dégradation de performance engendrée par le protocole de cohérence étendu. Cette diminution n'est cependant pas aussi importante que celle que l'on pourrait envisager car le coût des injections, en nombre de cycles, est plus important lorsque la fréquence d'horloge augmente.

Chapitre VIII

Optimisations et problèmes ouverts

Nous profitons de ce dernier chapitre pour présenter certaines optimisations envisageables et pour aborder un certain nombre de problèmes que nous n'avons pas considérés. Les optimisations présentées ont bien sûr pour but de réduire la dégradation de performance. La première permet d'éliminer la phase 2 de l'algorithme de sauvegarde d'un point de récupération. La deuxième envisage une sauvegarde des points de récupération en tâche de fond sans arrêter les calculs des processeurs. La dernière optimisation est liée au réseau d'interconnexion utilisé. La deuxième partie du chapitre s'intéresse à différents problèmes que nous n'avons pas considérés. En particulier nous discutons de l'utilisation de dépendances pour limiter le nombre de processeurs concernés par la sauvegarde d'un point de récupération. Nous y abordons également les problèmes d'extensibilité de notre proposition lorsque le nombre de nœuds est très important.

VIII.1 Optimisations

VIII.1.1 Compteurs de validation

La seconde phase de l'algorithme de sauvegarde d'un point de récupération a deux objectifs. Elle collecte tout d'abord l'ensemble des lignes mémoire *Invalide-CK* devenues inutiles pour changer leur état à *Invalide*. Elle est également chargée de changer l'état des lignes *Pré-validé* à *Partagé-CK*. L'algorithme utilisé est fort simple, puisqu'il s'agit d'un parcours séquentiel des états des lignes mémoire. Son coût est cependant proportionnel au nombre de lignes à tester et la dégradation de performance qu'il engendre peut être élevée si les mémoires et les fréquences de sauvegarde des points de récupération sont importantes. Pour limiter son coût, une première optimisation consiste à ne tester que les pages allouées ou mieux, les pages modifiées dans une région de récupération. Le coût de cette phase reste cependant élevé si ce nombre de pages est important.

Nous envisageons ici une solution différente permettant de se passer du parcours séquen-

tiel de la mémoire lors de la seconde phase de l'algorithme d'établissement d'un point de récupération. La modification des états des lignes est alors réalisée lors des accès mémoire ultérieurs faits par les processeurs.

VIII.1.1.1 Principes

Dans cette approche, un compteur de validation C_i est associé à chaque nœud de l'architecture. Ce compteur mémorise le nombre de points de récupération sauvegardés et est identique pour l'ensemble des processeurs. Nous l'appelons dans la suite compteur de validation global. Chaque ligne mémoire se voit aussi associer un compteur de validation C_{ligne}. Durant le calcul, ce compteur est mis à jour avec la valeur du compteur de validation global lorsque la ligne est modifiée ou injectée par un processeur. Ainsi les lignes *Invalide-CK* créées lors de la première modification d'une ligne se voient affecter un compteur de validation égal à la valeur du compteur de validation global.

Dans la phase d'établissement d'un point de récupération, les lignes modifiées dans la région de récupération sont répliquées et marquées directement *Partagé-CK*. Leurs compteurs sont cependant affectés à $C_i + 1$ indiquant que ces copies appartiennent au nouveau point de récupération. L'état *Partagé-CK* associé à un compteur de validation C_{ligne} égal à $C_i + 1$ permet ainsi de coder l'ancien état *Pré-validé*. C'est le seul cas où le compteur de validation d'une ligne peut être supérieur au compteur de validation d'un nœud.

La seconde phase de l'algorithme de sauvegarde d'un point de récupération se limite à un simple incrément du compteur de validation de chaque nœud. Instantanément, toute lignes *Invalide-CK*, créées dans la région de récupération précédente, deviennent alors invalides car leur compteur de validation est inférieur au compteur de validation global. Les lignes *Partagé-CK*, nouvellement répliquées, sont quant à elles confirmées comme copies de récupération car leur compteur de validation devient égal au compteur de validation global.

Durant le calcul, les accès mémoire nécessitent une consultation du compteur de validation de la ligne seulement lorsque l'état de cette ligne est *Invalide-CK*. Cette consultation permet alors de déterminer si la ligne appartient au point de récupération courant ($C_{ligne} = C_i$), ou est en fait une ancienne ligne de récupération maintenant inutile ($C_{ligne} < C_i$). L'accès à une ligne dans l'état *Partagé-CK* est autorisé quelle que soit la valeur de son compteur de validation. En effet, celui-ci est de toutes façons inférieur ou égal au compteur de validation du nœud. Un compteur égal indique que cette ligne a été confirmée au dernier point de récupération. Un compteur inférieur indique que la ligne a été confirmée lors d'un point de récupération antérieur mais qu'elle n'a pas été modifiée depuis. L'ensemble des autres accès est traité de façon standard. L'algorithme utilisé lors d'un accès à une ligne est représenté sur la figure VIII.1

La valeur du compteur de validation étant codé sur p bits, il est nécessaire de réinitialiser les compteurs tous les $2^p - 1$ points de récupération. Dans le cas contraire, le nouveau point de récupération ferait passer la valeur du compteur à zéro et il deviendrait impossible

```
Accès à une ligne {
 Cas (ligne.état) dans {
  Invalide-CK1 :
  Invalide-CK2 :
    Si C_i > C_ligne alors   {Cette ligne appartient à un ancien point de récupération}
      ligne.état = Invalide
      Traitement standard
    sinon
      Traitement standard
    Fsi
  Défaut :
    Traitement standard
 Fcas
}
```

Figure VIII.1 Algorithme d'accès à une ligne

de distinguer les lignes *Invalide-CK* devenues invalides. Quand cette situation se produit, les nœuds remplacent la seconde phase de l'algorithme de sauvegarde d'un point de récupération par un parcours séquentiel des états des lignes mémoire. Durant ce parcours, les lignes *Invalide-CK* sont invalidées et les lignes *Partagé-CK* se voient affecter un compteur de validation égal à zéro. Le compteur de validation global est ensuite affecté à zéro. Bien entendu la taille du compteur de validation influence la fréquence des parcours de la mémoire à réaliser.

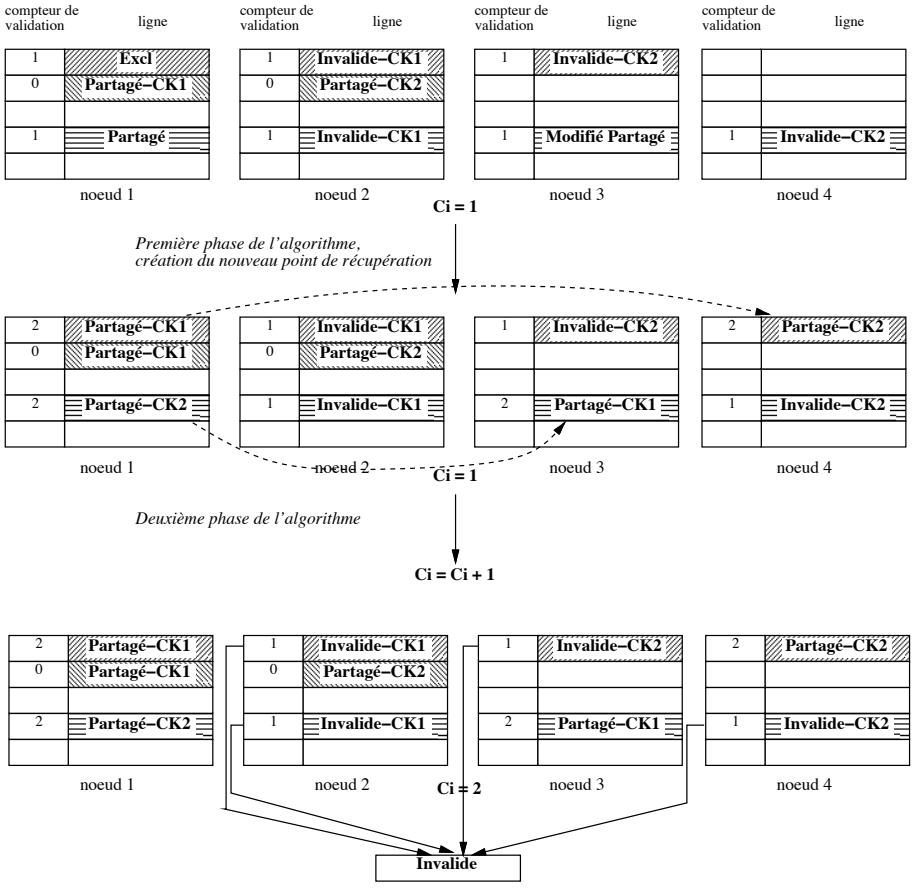
VIII.1.1.2 Restauration des points de récupération

L'algorithme de restauration d'un point de récupération se trouve sensiblement modifié puisque certaines lignes *Invalide-CK* sont en fait des lignes invalides et, en cas de faute durant l'établissement d'un nouveau point de récupération, certaines lignes *Partagé-CK* appartiennent au nouveau point de récupération et doivent donc être invalidées. L'algorithme de restauration apparaît sur la figure VIII.2. Comme dans l'algorithme originel, l'ensemble des copies courantes est invalidé. Les copies *Invalide-CK* sont restaurées dans l'état *Partagé-CK* seulement si elles correspondent à des copies de récupération courantes. Les copies *Partagé-CK* sont invalidées seulement si la faute a lieu pendant la phase d'établissement et que la copie appartient au point de récupération temporaire ($C_{ligne} > C_i$).

VIII.1.1.3 Coût d'implémentation

L'utilisation de compteurs de validation associés à chaque ligne de la mémoire permet donc d'éviter un parcours séquentiel des mémoires lors de la seconde phase de l'algorithme d'établissement d'un point de récupération. Elle engendre cependant une augmentation pos-

Exemple VIII.1 (Exemple d'utilisation de compteurs de validation)



Taille du compteur (en bits)	Surcoût mémoire	Occupation mémoire des compteurs (mémoire 32 Mo, ligne 128 octets)
8	0,78%	256 Ko
16	1,56%	512 Ko
32	3,12%	1Mo

Table VIII.1 Surcoût mémoire engendré par l'utilisation de compteurs de validation

```
Réception du message début_restauration
début { Restauration du point de récupération courant }
 Pour chaque ligne dans la mémoire attractive {
  Cas (ligne.état) dans {
   Invalide-CKi : {i = 1 ou 2}
    Si C_i > C_ligne alors {Cette ligne n'est plus valide}
      ligne.état = Invalide
    sinon { Cette ligne appartient au point de récupération courant}
      ligne.état = Partagé-CKi
    Fsi
   Partagé-CKi : {i = 1 ou 2}
    Si C_i > C_ligne alors {La faute a eu lieu pendant la sauvegarde d'un point de récupération}
      ligne.état = Invalide
    sinon {La faute a lieu pendant le calcul}
      ligne.état = Partagé-CKi
    Fsi
   Partagé :
   Exclusif :
   Non Modifié Partagé :
    ligne.état = Invalide
   Défaut :
    {Aucun autre état possible }
 } fcas
} fpour
{ Reprise du calcul }
```

Figure VIII.2 Algorithme de restauration avec utilisation des compteurs de validation

sible des temps d'accès aux lignes ainsi qu'un surcoût mémoire pour le stockage des compteurs. L'augmentation des temps d'accès mémoire peut être considérée comme négligeable car seuls les accès à des lignes *Invalide-CK* sont concernés. Le surcoût mémoire est certainement le surcoût le plus important. La table VIII.1 donne, pour des lignes de 128 octets, le surcoût mémoire en fonction de la taille des compteurs. L'utilisation de compteurs de 8 bits introduit un surcoût inférieur à 1% mais nécessite un parcours de la mémoire tous les 255 points de récupération. Des compteurs de 32 bits représentent un surcoût mémoire un peu supérieur à 3% mais permettent de limiter le nombre de parcours de la mémoire à un tous les 4 milliards points de récupération.

VIII.1.1.4 Gain en performance

Pour évaluer cette proposition, nous n'avons pas réalisé de nouvelles simulations. Le gain peut simplement être évalué en enlevant la phase 2 de l'algorithme des graphes de la figure VII.2. Ainsi pour *Mp3d* et 400 points de récupération par seconde, le gain représente un peu plus de 10%. L'intérêt de cette optimisation augmente avec la taille des mémoires et la

fréquence des sauvegardes des points de récupération.

VIII.1.2 Sauvegarde des points de récupération en arrière plan

L'utilisation de compteurs de validation permet de réduire la seconde phase de l'algorithme d'établissement d'un point de récupération, à un simple incrément de compteur. Cependant les processeurs restent toujours stoppés pendant la première phase de cet algorithme (phase de création du point de récupération temporaire). Une seconde optimisation envisageable consiste à sauvegarder les points de récupération en parallèle avec le calcul de sorte que les processeurs ne sont plus arrêtés lors de la création du point de récupération temporaire (figure VIII.3). La seconde phase de l'algorithme utilise des compteurs de validation et ne perturbe donc pas le calcul.

VIII.1.2.1 Principes

Pour autoriser les processeurs à continuer leurs calculs durant la sauvegarde d'un point de récupération, la phase 1 de l'algorithme se limite à la mise à jour de la mémoire locale et au lancement d'une tâche de fond chargée de créer le nouveau point de récupération. Les calculs sont aussitôt repris par les processeurs.

Pendant l'établissement d'un nouveau point de récupération, trois versions d'une ligne mémoire peuvent alors cohabiter dans l'architecture : (i) une version de la ligne correspondant au point de récupération permanent, utilisée en cas de retour arrière, (ii) une version appartenant au point de récupération en cours de création (point de récupération temporaire), (iii) une version courante si la ligne a été modifiée pendant la création du point de récupération temporaire.

VIII.1.2.2 Implémentation

Nous ne présentons pas de façon très détaillée cette optimisation car aucune évaluation de performance n'a été réalisée. Nous nous contentons d'en donner les principales idées.

Comme dans les autres versions, l'algorithme de sauvegarde d'un point de récupération utilise deux phases. Les nœuds sont cependant arrêtés moins longtemps que dans la version utilisant simplement les compteurs de validation. Après synchronisation des nœuds pour débuter la sauvegarde d'un nouveau point de récupération, la première phase de l'algorithme débute. Pour mettre à jour la mémoire, chaque nœud vide en mémoire les lignes modifiées se trouvant dans son cache ainsi que les registres de son processeur. Une fois cette mise à jour réalisée, l'algorithme de création du point de récupération temporaire est lancé en tâche de fond au niveau des unités de gestion de la mémoire attractive et les processeurs reprennent leurs calculs. Cet algorithme se contente de parcourir les mémoires et de répliquer les lignes modifiées dans la dernière région de récupération.

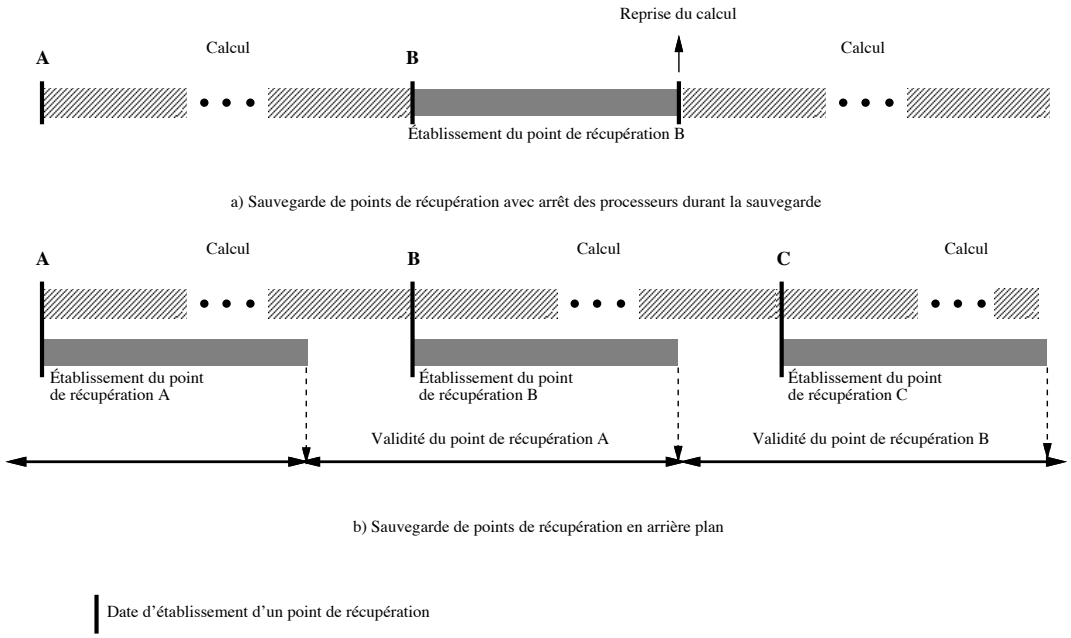


Figure VIII.3 Principe de la sauvegarde des points de récupération en arrière plan

Durant la phase d'établissement du point de récupération temporaire, les processeurs sont autorisés à poursuivre leur calcul. Les accès en lecture sont traités de façon standard car ils ne modifient pas le point de récupération à sauvegarder. Les accès en écriture sont plus délicats. En effet, il se peut alors qu'un processeur désire accéder en écriture à une ligne faisant partie du point de récupération temporaire mais qui n'a pas encore été répliquée. Dans ce cas, la réplication doit être réalisée avant que la copie courante de la ligne ne soit créée. L'état du système après ces opérations apparaît sur la figure VIII.4. Cinq copies de la ligne, existent alors. Une copie courante, deux copies appartenant au point de récupération temporaire et deux copies appartenant au point de récupération permanent.

L'identification des lignes à répliquer est réalisée grâce aux compteurs de validation associés aux lignes. Dans une phase d'établissement d'un point de récupération, une ligne *Exclusif* ou *Modifié Partagé*, dont le compteur C_{ligne} est égal au compteur de validation global C_i, doit être répliquée puisqu'elle a été modifiée dans la région de récupération courante. Les lignes courantes créées lors d'une phase d'établissement d'un point de récupération sont marquées avec un compteur de validation égal à $C_i + 1$ indiquant qu'elles ont été modifiées dans la région de récupération suivante. La première phase se termine quand les tâches de fond des nœuds ont terminé les réplications nécessaires pour assurer la stabilité du point de récupération temporaire. La seconde phase de l'algorithme valide le point de récupération permanent en incrémentant le compteur de validation des nœuds.

La restauration d'un point de récupération est réalisée de façon relativement similaire à celle décrite lorsque seuls les compteurs de validation sont utilisés.

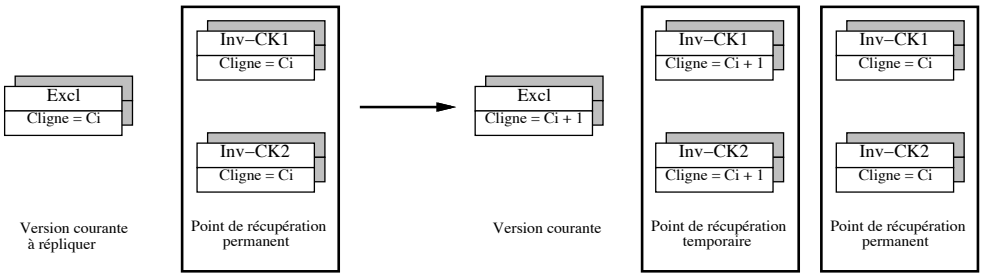


Figure VIII.4 Traitement des accès en écriture durant une phase d'établissement d'un point de récupération

VIII.1.2.3 Coût d'implémentation

Cette nouvelle optimisation ne peut être implémentée sans engendrer un certain surcoût. Du point de vue du surcoût mémoire engendré par l'utilisation de compteurs de validation, le coût est identique à la solution utilisant uniquement les compteurs de validation. La gestion des différentes copies d'une même ligne mémoire complique cependant les accès mémoire et le protocole de cohérence qui doit utiliser de nouveaux états. Un autre effet de cette optimisation est d'augmenter les besoins en mémoire en nécessitant jusqu'à cinq copies pour une même ligne durant l'établissement d'un nouveau point de récupération. Finalement, le principal surcoût matériel engendré par cette optimisation est l'introduction, au niveau de la mémoire attractive, d'une unité matérielle indépendante chargée d'implémenter l'algorithme lancé en tâche de fond lors de la sauvegarde d'un point de récupération.

Il est à noter que les processeurs peuvent être ralentis durant les phases d'établissement des points de récupération. Ce ralentissement est essentiellement dû aux problèmes causés par les accès en écriture décrits précédemment. La plupart des accès des processeurs étant des accès en lecture, ce ralentissement doit cependant être minime.

VIII.1.3 Utilisation d'un réseau en tore

Cette dernière optimisation nous est apparue après l'évaluation de performance réalisée avec le réseau en grille. En effet, l'utilisation de l'anneau d'injection pénalise les processeurs situés dans le coin en bas à droite de la grille car leurs injections nécessitent un transfert de message vers le nœud situé dans le coin en haut à gauche. Dans les cas de $Mp3d$ par exemple, les nœuds situés dans le coin en bas à droite de la grille ont des temps d'injection de lignes partagées de l'ordre de 185 cycles alors que les autres nœuds ne dépassent pas les 140 cycles. Cette différence a pour effet de rallonger la durée globale de la phase 1 de l'algorithme de sauvegarde d'un point de récupération car celle-ci exige une synchronisation des processeurs

pour se terminer.

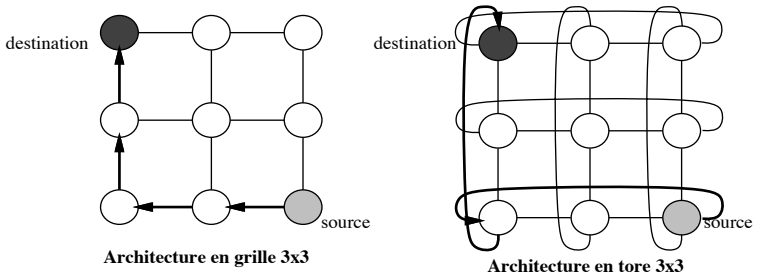


Figure VIII.5 Architecture en grille et architecture en tore

Pour résoudre un tel problème, une solution consiste à utiliser un réseau en tore, c'est-à-dire une grille où les extrémités sont reliées (figure VIII.5). Cette topologie permet au nœud situé auparavant dans le coin en bas à droite de la grille d'accéder au nœud en haut à gauche en au plus deux sauts. Elle minimise donc les différences de temps d'injection entre les nœuds et par conséquent la dégradation de performance.

VIII.2 Problèmes ouverts

Ce paragraphe a pour objectif de discuter de certains aspects de notre proposition qui n'ont pas été abordés auparavant.

VIII.2.1 Gestion de dépendances

Jusqu'ici, nous avons considéré une stratégie globale de sauvegarde des points de récupération. Cette stratégie a cependant le désavantage de forcer l'ensemble des processeurs à sauvegarder un nouveau point de récupération. Pour limiter la quantité de données de récupération traitée, une autre stratégie de synchronisation moins contraignante est cependant parfaitement envisageable. Elle consiste à prendre en compte les communications entre les processeurs afin de définir dynamiquement l'ensemble des processeurs concernés par la sauvegarde d'un point de récupération.

Considérons les deux interactions qui posent problème, présentées sur la figure III.2. Lorsqu'un processeur Q lit une donnée modifiée par un autre processeur P, il se crée une dépendance $Q \rightarrow P$ qui oblige P à sauvegarder un point de récupération si Q désire en établir un nouveau. Dans le cas contraire, l'état formé par le nouveau point de récupération de Q et l'ancien de P ne serait pas cohérent.

De même, quand Q modifie une donnée précédemment modifiée par P, il se crée une dépendance $P \rightarrow Q$ qui oblige Q à établir un point de récupération si P désire en établir un nouveau. Dans le cas contraire, une incohérence pourrait apparaître si Q fait un retour

arrière et fait restaurer une valeur de la variable plus ancienne que l'écriture de P. Dans ce dernier cas d'interaction, une dépendance $Q \rightarrow P$ doit aussi être notée si la taille des lignes mémoire est supérieure à un mot. En effet, dans la ligne que Q modifie peut se trouver un autre mot modifié par P dont la lecture par Q ne peut être détectée.

Lorsqu'un processeur décide de sauvegarder un nouveau point de récupération, l'ensemble des processeurs concernés par cette sauvegarde se limite à l'ensemble des nœuds ayant une dépendance directe ou indirecte avec ce processeur [Joubert 93].

VIII.2.1.1 Mise en œuvre

Dans les architectures considérées, l'enregistrement de ces dépendances est assez simple à réaliser si le propriétaire d'une ligne est toujours utilisé pour servir un défaut. Lorsqu'un nœud fournit une ligne modifiée dans une région de récupération (état *Exclusif* ou *Modifié Partagé*), il note une dépendance avec le nœud à l'origine de la requête. Dans le cas d'un défaut en écriture, les deux nœuds notent une dépendance l'un avec l'autre. L'information conservée sur chaque nœud peut se limiter à un bit par nœud de l'architecture.

Lorsqu'un nœud désire établir un nouveau point de récupération, il doit contacter l'ensemble des nœuds avec lesquels il possède une dépendance directe ou indirecte. L'algorithme utilise alors deux phases. Dans la première phase, l'ensemble des nœuds est contacté en construisant un arbre, chaque nœud contactant ses dépendants directs. Avant de contacter les nœuds avec lesquels il a noté une dépendance directe, un nœud termine sa requête en cours et bloque ses accès externes pour éviter la création de nouvelles dépendances. Il répond alors à toutes les requêtes externes par un acquittement négatif. Lorsqu'il a reçu une réponse à sa requête (éventuellement négative), il contacte effectivement les nœuds avec lesquels il possède une dépendance directe et commence la réplication des lignes mémoire modifiées de sa mémoire attractive. Lorsque cette phase de réplication est terminée, un nœud attend les réponses de ses descendants avant d'envoyer lui-même une réponse à son ascendant. Les réponses remontent ainsi jusqu'au nœud à l'origine de l'établissement du point de récupération.

Quand il a reçu les réponses de ses descendants directs, le nœud à l'origine du point de récupération envoie au groupe de nœuds formé l'ordre de passage dans la seconde phase. Cette phase consiste alors à valider les données appartenant à l'ancien point de récupération et à confirmer les nouvelles. Les dépendances des nœuds du groupe sont effacées et les nœuds peuvent reprendre leur calcul.

VIII.2.1.2 Problèmes de cette solution

Même si conceptuellement la gestion de dépendances assure une quantité minimale de données de récupération [Banâtre *et al.* 93a], elle possède plusieurs inconvénients dans le cadre de l'implémentation proposée.

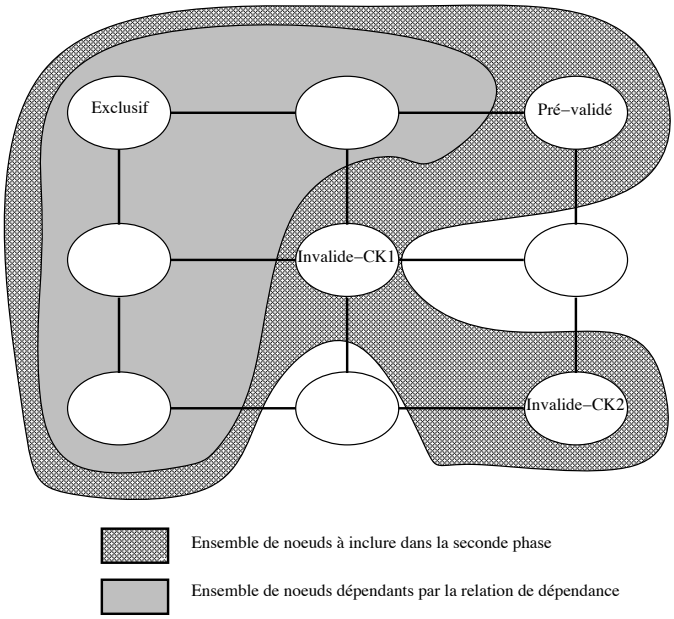


Figure VIII.6 Exemple d'utilisation de dépendances dans une architecture utilisant une grille 3x3

L'inconvénient majeur concerne les injections de lignes. Durant la seconde phase de l'algorithme d'établissement d'un point de récupération, les copies *Pré-validées* et *Invalide-CK* des lignes modifiées par les nœuds du groupe doivent changer leur état. L'absence de localisation fixe de ces copies peut impliquer qu'un certain nombre d'autres nœuds, utilisés pour recevoir ces copies et n'appartenant pas nécessairement au groupe calculé, soient ajoutés pour participer à la seconde phase de l'algorithme (figure VIII.6). Il existe trois solutions à ce problème.

- La première consiste à contraindre la localisation de copies *Pré-validées* ou *Invalide-CK* seulement sur des nœuds ayant déjà une dépendance avec le nœud à l'origine de l'injection. Cette solution a pour conséquence d'augmenter l'effet de pollution. De plus, elle peut poser des problèmes de place mémoire car il devient difficile d'assurer qu'un groupe de nœuds contiendra assez de pages allouées pour assurer la réplication des données.

- Une seconde solution consiste à ajouter une dépendance losrqu'une ligne de ce type est injectée sur un autre nœud. Cet ajout de dépendance va cependant à l'encontre de l'objectif qui est de réduire le nombre de nœuds impliqués dans l'établissement d'un point de récupération. Cette solution peut de plus rendre rapidement dépendant l'ensemble des processeurs en particulier si un anneau d'injection est utilisé.

- La troisième solution envisageable consiste quant à elle à noter l'identité des nœuds qui ont accepté une de ces injections et à les inclure dans la seconde phase de l'algorithme.

Beaucoup de nœuds risquent donc d'être perturbés durant la seconde phase de l'algorithme de sauvegarde d'un point de récupération. Cette solution nécessite de plus que les copies concernées maintiennent l'identité du propriétaire de la ligne afin que seules ces copies changent leur état lors de la seconde phase de l'établissement d'un point de récupération.

Ces trois solutions ont en commun de nécessiter l'ajout de nœuds lors de la première ou de la seconde phase de la sauvegarde d'un point de récupération. Elles minimisent donc l'intérêt de l'utilisation de dépendances. De plus, l'ensemble des calculs de l'architecture sont perturbés car les nœuds en cours de sauvegarde d'un point de récupération ne peuvent pas répondre aux autres nœuds.

Les autres inconvénients de l'utilisation de dépendances sont liés aux calculs de groupes mis en œuvre par des algorithmes distribués. Ces algorithmes doivent en effet pouvoir prendre en compte les défaillances pendant leur déroulement ainsi que des situations où des nœuds appartiennent simultanément à plusieurs groupes de processeurs. Ces situations ont pour effet de compliquer l'implémentation de la récupération arrière.

Pour finir, l'introduction de dépendances rend difficile l'utilisation des optimisations proposées précédemment car le nombre de points de récupération sauvegardés par les nœuds varie suivant les nœuds. L'intérêt des dépendances est de plus discutable si l'architecture utilise un système d'exploitation unique (cas de la machine KSR1). Il semble en effet probable que le partage des structures de données de ce système opératoire rende rapidement dépendant l'ensemble des processeurs. Il suffit ainsi d'un verrou d'exclusion mutuelle accédé par tous les processeurs pour que tous les nœuds soient dépendants.

VIII.2.2 Utilisation du protocole dans une architecture à grand nombre de nœuds

Même si nous avons montré dans le chapitre VII que le protocole de cohérence étendu conserve l'extensibilité de l'architecture étudiée, le nombre de nœuds considéré reste encore relativement faible. Il convient donc de se poser la question de l'utilisation du protocole dans une architecture comportant un très grand nombre de nœuds (plusieurs milliers). L'implémentation du protocole telle qu'elle est décrite dans les chapitre VI pose en effet deux types de problèmes lorsque le nombre de nœuds devient important.

Elle peut, tout d'abord, impliquer des temps de sauvegarde des points de récupération et un effet de pollution qui augmentent de façon importante. L'utilisation de l'anneau d'injection peut en effet se révéler coûteuse si une injection de ligne nécessite la visite d'un grand nombre de nœuds avant de trouver un emplacement mémoire prêt à accepter une ligne. Des techniques d'allocation dynamique des lignes peuvent être envisagées. Elles peuvent cependant créer une occupation mémoire importante et plus complexe à gérer. De plus l'allocation dynamique de ligne ne corrige pas le second problème engendré par l'utilisation d'un nombre

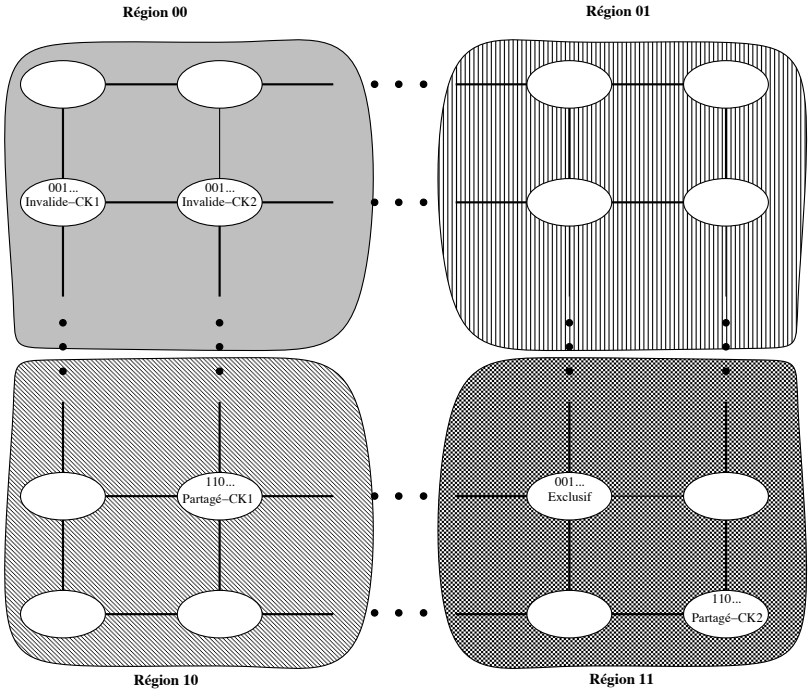


Figure VIII.7 Découpage d'une architecture en régions de localisation pour les données de récupération

important de nœuds. Ce problème est une augmentation importante du temps de reconfiguration de la mémoire. En effet dans le cas où les copies *Invalide-CK* ne conservent pas de pointeurs l'une sur l'autre, la vérification de leur présence, dans l'architecture, devient très coûteuse s'il est nécessaire de tester un grand nombre de nœuds.

Une solution à ces deux problèmes consiste à diviser l'architecture en régions, chaque région comprenant plusieurs nœuds, et à forcer la localisation des copies de récupération des lignes dans une région fixée de l'architecture en fonction, par exemple, de leurs adresses. On peut ainsi imaginer un découpage d'une architecture en plusieurs régions (figure VIII.7). L'identification de la région dans laquelle les copies *Invalide-CK* d'une ligne doivent se trouver peut alors être réalisée à l'aide d'une fonction similaire à celle utilisée pour la distribution des pointeurs de localisation. Avec cette solution, seule les copies de récupération de type *Partagé-CK1* sont autorisées à résider hors de leur région de localisation. Ces copies ne posent toutefois pas de problème car le pointeur de localisation de cette ligne conserve l'identité du nœud qui la possède.

Ce découpage a plusieurs avantages. Il permet tout d'abord de limiter les temps d'injection de lignes de récupération. Lorsque ce type d'injection est nécessaire, la requête est directement envoyée vers la région concernée, et le nombre de nœuds à tester est donc limité. L'autre avantage est de limiter les temps de reconfiguration mémoire. Lors d'une défaillance d'un nœud, seule sa région est concernée par les vérifications de présence des copies de récu-

pération *Invalide-CK* restaurées. Pour les copies *Partagé-CK*, seuls les nœuds possédant une copie *Partagé-CK1* dont la copie *Partagé-CK2* appartient à la région du nœud en défaillance, doivent réaliser une vérification. Il reste alors à régler le cas de copies *Partagé-CK2* situées dans des régions saines qui doivent vérifier la présence de la copie *Partagé-CK1*.

Le dernier avantage de cette solution est d'augmenter la probabilité de tolérer les défaillances multiples. En effet, avec cette solution, la défaillance de deux nœuds situés dans des régions différentes est tolérée sauf dans le cas où ces deux nœuds possèdent les deux copies *Partagé-CK* d'une ligne. La probabilité d'occurrence d'une telle situation reste cependant relativement faible.

Le seul défaut de cette solution est de minimiser l'utilisation de la réplication de données existante lorsqu'un point de récupération doit être sauvegardé. Ainsi une copie *Modifié Partagé* ne peut utiliser une copie *Partagé* pour la changer en copie *Partagé-CK2*, si elle n'est pas située dans la bonne région de l'architecture.

VIII.2.3 Travaux relatifs

Dans le reste du document nous n'avons pas considéré l'ensemble des problèmes relatifs à une technique de tolérance aux fautes dans une architecture extensible à mémoire partagée. Il est notamment supposé que les nœuds défaillent par écroulement et que le réseau est fiable.

La première hypothèse assure l'isolation des nœuds vis à vis des défaillances et garantit la détection de la défaillance par les autres nœuds de l'architecture. Un nœud qui possède ces caractéristiques défaille sans émettre d'informations incorrectes vers le monde extérieur. Il est certain que l'implémentation d'une telle propriété ne peut être assurée sans induire un certain surcoût. Ce surcoût est généralement matériel car les mécanismes de détection d'erreur doivent être rapides. Assurer cette propriété au niveau d'un nœud semble cependant plus aisé que de l'assurer au niveau d'un composant. En effet, il s'agit alors d'empêcher un nœud d'émettre une information erronée vers les autres nœuds. Ainsi les temps de détection d'une erreur ne sont pas critiques puisqu'il est simplement nécessaire de détecter une erreur avant qu'un nœud n'émette une requête vers l'extérieur. L'interface réseau fournit alors un moyen d'isolation d'un nœud vis à vis du monde extérieur. Cette approche est utilisée dans l'architecture Delta-4 [Powell 94]. À l'intérieur d'un nœud, des moyens de détections d'erreur doivent cependant être envisagés. Les mémoires et les bus bénéficient habituellement de codes correcteurs d'erreurs (ECC) permettant de détecter les erreurs. Ce type d'erreur sur les données peut être transmis au système d'exploitation qui se charge alors de prendre les actions d'isolation qui s'imposent. Les fautes des processeurs peuvent être détectées soit grâce à des mécanismes internes de détection d'erreur soit par une technique beaucoup plus coûteuse mais plus sûre, de réplication du processeur [Schneider 87].

La seconde hypothèse concerne le réseau d'interconnexion qui est supposé fiable. Cette hypothèse n'est pas irréaliste. De nombreux travaux ont pour objectif d'assurer la fiabilité d'un réseau d'interconnexion tel qu'une grille. Généralement les solutions proposées se basent

sur un routage adaptatif capable de prendre en compte les défaillances des liens et des nœuds [Allen *et al.* 94, Bolding & Yost 94, Dally *et al.* 94]. Dans [Dally *et al.* 94], un routeur permettant de tolérer les défaillances des nœuds ou des liens de communication et assurant qu'un paquet sera délivré exactement une fois, est proposé. Pour notre architecture, ce type de réseau conviendrait parfaitement. Les performances envisagées par ces réseaux sont similaires à celles de réseaux standard. Les liens du réseau présenté dans [Dally *et al.* 94] permettent ainsi des débits de 400 Mo/s.

VIII.3 Résumé

Les points à retenir de ce chapitre sont :

(1) L'utilisation de compteurs de validation permet de limiter la seconde phase de l'algorithme d'établissement d'un point de récupération à une simple incrémentation de compteur. Le gain de performance apporté par cette optimisation peut être important si les mémoires attractives sont de taille importante.

(2) La sauvegarde de points de récupération en arrière plan permet de continuer les calculs pendant la sauvegarde d'un point de récupération. Elle nécessite cependant une occupation mémoire plus importante ainsi qu'un matériel plus complexe.

(3) L'utilisation d'un réseau en tore permettrait de minimiser la dégradation de performance car les temps d'injection de lignes deviendraient identiques pour tous les processeurs.

(4) L'utilisation de dépendances permet de limiter la taille des données de récupération transférées lors des sauvegardes de points de récupération. Lorsque les données de récupération n'ont pas de localisation physique fixe, cette optimisation est beaucoup moins intéressante puisqu'il devient nécessaire d'inclure d'autres nœuds dans la seconde phase de l'algorithme de sauvegarde des points de récupération.

(5) Lorsque le nombre de nœuds est important, le découpage de l'architecture en régions de localisation des données de récupération permet de limiter les temps d'injection de lignes et les temps de reconfiguration.

Chapitre IX

Conclusion

IX.1 Bilan

Les architectures extensibles à mémoire partagée fournissent à la fois une réponse aux besoins grandissants en puissance de calcul et aux besoins de facilité de programmation des machines massivement parallèles. Elles nécessitent cependant l'introduction de mécanismes de tolérance aux fautes pour être réellement utilisables. L'objet de cette étude était la proposition de mécanismes de tolérance aux fautes qui permettent à ces architectures d'assurer une continuité de service malgré l'occurrence de défaillances des nœuds.

Dans un premier temps, nous avons étudié les mécanismes de réplication de données indispensables pour assurer une bonne efficacité à des architectures extensibles. Ces mécanismes reposent sur l'utilisation de mémoires cache locales qui permettent la réplication et la migration automatiques des données accédées par les processeurs. La présence de multiples répliques d'une même donnée nécessite l'utilisation de protocoles de cohérence dont l'intégration dans des architectures extensibles a été présentée en détail. Les architectures de type COMA constituent l'aboutissement de la politique visant à utiliser des caches pour minimiser les temps de latence puisqu'elles proposent une transformation des mémoires de l'architecture en caches de grande dimension.

Nous nous sommes ensuite intéressés aux techniques de tolérance aux fautes et en particulier à la récupération arrière. Nous avons montré que la gestion de données de récupération nécessite des mécanismes de réplication de données. Une première réplication a pour but de conserver intact l'ensemble des données appartenant au point de récupération. Une seconde réplication permet quant à elle d'assurer, aux données de récupération, des propriétés de stabilité qui minimisent les hypothèses sur les défaillances tolérées. Elles permettent en particulier de considérer la défaillance du support utilisé pour les stocker.

Nous nous sommes alors placés dans le cadre d'architectures de type COMA qui, en garantissant l'absence de localisation physique fixe d'une ligne mémoire, simplifient la reconfiguration mémoire après la défaillance d'un nœud. Nous avons montré que les mécanismes

de gestion de la réplication de données des architectures COMA permettent d'assurer, aux données de récupération, la réplication nécessaire pour vérifier l'ensemble des propriétés de stabilité. La réplication de données étant habituellement gérée par le protocole de cohérence de l'architecture, nous avons alors proposé un protocole de cohérence étendu qui intègre de façon transparente la gestion des données courantes et des données de récupération. Ce protocole permet d'implémenter une technique de récupération arrière de type pessimiste de façon simple en utilisant les mécanismes de réplication et les mémoires standard d'une architecture COMA pour assurer le stockage des données courantes et des données de récupération. Les avantages de ce protocole sont multiples. Il limite tout d'abord les hypothèses simplificatrices en assurant l'ensemble des propriétés de stabilité aux données de récupération. Il fait de plus un usage optimum de la place mémoire en autorisant notamment la lecture et la réplication d'une donnée de récupération tant que celle-ci n'est pas modifiée. En utilisant le réseau d'interconnexion de la machine, il assure également des temps d'établissement des points de récupération faibles. Finalement, il permet de tirer parti de la réplication de données existante pour limiter les temps de sauvegarde des points de récupération.

L'intégration de ce protocole dans une architecture de type COMA a ensuite été étudiée. Il ressort de cette étude que les modifications matérielles à réaliser, par rapport à une architecture standard, sont faibles et localisées aux unités chargées de la gestion de la mémoire. Aucune fonctionnalité nouvelle n'est introduite pour gérer les données de récupération et les principales modifications proviennent des nouvelles injections de copies de récupération et des mécanismes de traitement des fautes.

L'évaluation de performance, menée par simulation, a permis de valider notre proposition en montrant que l'utilisation du protocole de cohérence étendu induit une dégradation de performance faible en comparaison d'une architecture utilisant un protocole de cohérence standard et ceci même lorsque les fréquences de sauvegarde des points de récupération sont élevées. Deux phénomènes expliquent ce résultat. D'une part, l'utilisation des mémoires et du réseau d'interconnexion de l'architecture, pour assurer le stockage et le transfert des données de récupération, assure des débits de transferts de données élevés qui garantissent des sauvegardes de points de récupération efficaces. L'implémentation du protocole à l'aide de répertoires permet de plus de tirer parti de la réplication de données existante pour limiter les transferts de données lors de la sauvegarde de points de récupération et diminuer ainsi la durée de ces opérations. D'autre part, en autorisant la lecture des données de récupération non modifiées, le protocole de cohérence étendu perturbe peu le comportement des mémoires attractives dont les taux de défauts restent similaires à ceux observés avec un protocole de cohérence standard. L'effet de pollution des mémoires attractives, causé par le stockage des données de récupération, est en fait essentiellement dû à l'introduction de nouveaux cas d'injection de lignes traités de façon efficace par l'architecture. Finalement, il a également été montré que l'utilisation du protocole de cohérence étendu n'est pas un obstacle à l'extensibilité d'une architecture notamment parce que les débits de transferts de données augmentent avec le nombre de nœuds de l'architecture.

Différentes optimisations permettant de limiter la dégradation de performance ont été

proposées. L'utilisation de compteurs de validation inhibe le parcours séquentiel des mémoires attractives pendant la seconde phase de l'algorithme de sauvegarde d'un point de récupération. La sauvegarde des points de récupération en arrière plan permet aux processeurs de continuer leur calcul durant la sauvegarde d'un point de récupération, au prix toutefois, d'un nouveau surcoût mémoire et d'une complexification du matériel.

IX.2 Perspectives

Différentes perspectives sont envisageables pour prolonger ce travail de thèse.

La construction réelle d'une architecture utilisant le protocole de cohérence étendu nécessite le développement de matériel spécifique notamment pour l'implémentation du protocole et des algorithmes de sauvegarde et de restauration des points de récupération qui lui sont associés. Même si le matériel est peu complexe, c'est un des principaux obstacles au développement d'une machine réelle.

Une approche différente peut être suivie en utilisant une architecture telle que l'architecture FLASH proposée par l'université de Stanford [J.Kuskin *et al.* 94, Heinrich *et al.* 94]. Cette architecture propose le remplacement des contrôleurs matériels utilisés dans les nœuds d'un multiprocesseur extensible, par un contrôleur programmable chargé de servir les accès venant du processeur aussi bien que de l'extérieur d'un nœud. De plus, pour pouvoir implémenter des protocoles dont les besoins en mémoire sont différents, un nœud ne contient pas de mémoire spécifique pour le protocole. Au lieu de cela, il utilise une partie de la mémoire principale du nœud pour stocker le code du protocole mis en œuvre ainsi que les données qui lui sont nécessaires. Un telle architecture peut être utilisée pour implémenter différents protocoles correspondant à différentes architectures de type CC-NUMA ou COMA. Dans ce cas, l'utilisation de la mémoire principale du nœud permet de conserver toute l'information nécessaire à la transformation de la mémoire en mémoire attractive. Pour le protocole de cohérence étendu, l'utilisation de la mémoire standard pour stocker les informations de cohérence permet de simplifier les mécanismes de reconfiguration. Des pointeurs de localisation peuvent ainsi être réalloués sans problème sur un nœud sain. Elle permet également d'implémenter facilement des optimisations telles que les compteurs de validation. Ce type d'architecture permet donc de mettre en œuvre le protocole de cohérence étendu sans le moindre développement de matériel spécifique. Les phases de mise au point sont de plus considérablement simplifiées puisque tous les algorithmes sont mis en œuvre par logiciel.

Du fait de son indépendance vis à vis d'une architecture, le protocole de cohérence proposé peut également être utilisé pour l'implémentation d'une mémoire virtuelle partagée recouvrable [Wu & Fuchs 89, Wu & Fuchs 90, E. L. Elnozahy & Zwaenepoel 92, Brown & Wu 94]. Les avantages de cette approche sont alors similaires à ceux observés pour les architectures de type COMA à ceci près que le grain de cohérence est ici beaucoup plus gros (une page). Cependant, l'utilisation de mémoires jouant le rôle de caches complètement associatifs permet d'inhiber les injections de copies de récupération. Nous travaillons actuellement sur

le développement d'un premier prototype sur l'architecture Paragon d'Intel [Cabillic *et al.* 94].

Du point de vue de l'évaluation de notre proposition, il serait intéressant de simuler des applications de taille beaucoup plus importante qu'il n'est possible de le faire par simulation. L'utilisation d'une machine telle que l'Intel Paragon associée à une mémoire virtuelle partagée laisse envisager des possibilités d'émulation d'architectures COMA qui permettraient de corriger ce défaut [Reinhardt *et al.* 93]. Bien que difficile, la prise en compte d'un système d'exploitation dans l'évaluation est aussi quelque chose d'indispensable pour avoir des performances tout à fait réalistes. Des approches à base d'émulation telles que celles réalisables avec le *banc de test* développé par l'*University of Southern California* [Barroso *et al.* 95] permettraient d'atteindre cet objectif. Il serait également intéressant d'évaluer de façon plus précise le coût de certains mécanismes permettant de minimiser les temps de traitement de fautes et de reconfiguration de l'architecture. Ainsi la gestion de pointeurs par les copies de récupération de type *Invalide-CK* permettrait de minimiser les temps de reconfiguration après la défaillance d'un nœud au prix cependant d'une dégradation de performance plus importante.

Bibliographie

[Agarwal & Gupta 88] A. Agarwal et A. Gupta. Memory Reference Characteristics of Multiprocessor Applications under MACH. Dans *Proc. of ACM SIGMetrics International Conference on Measurement and Modeling of Computer Systems*, pp 215–225. ACM, Mai 1988.

[Agarwal *et al.* 91] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Ma, et D. Nussbaum. The MIT Alewife Machine : A Large-Scale Distributed Memory Multiprocessor. Rapport Technique MIT/LCS/TM-454, MIT Laboratory for Computer Science, Juin 1991.

[Ahmed *et al.* 90] R.E. Ahmed, R.C. Frazier, et P.N. Marinos. Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems. Dans *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, pp 82–88, Newcastle, Juin 1990.

[Allen *et al.* 94] J.D. Allen, P.T. Gaughan, D.E. Schimmel, et S. Yalamanchili. Ariadne : An Adaptative Router for Fault-tolerant Multicomputers. Dans *Proc. of 21th Annual International Symposium on Computer Architecture*, pp 278–288, Avril 1994.

[Anderson & Lee 81] T. Anderson et P. A. Lee. *Fault Tolerance : Principles and Practice.* Prentice-Hall International, 1981.

[Archibald & Baer 86] J. Archibald et J. L. Baer. Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, Novembre 1986.

[Archibald 87] J. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors.* Thèse de doctorat, University of Washington, Décembre 1987.

[Banâtre *et al.* 92] M. Banâtre, M. Jégado, P. Joubert, et C. Morin. Communicating Processes and Fault Tolerance: A Shared Memory Multiprocessor Experience. Rapport Technique 1649, INRIA, Mars 1992.

[Banâtre *et al.* 93a] M. Banâtre, A. Gefflaut, P. Joubert, P.A. Lee, et C. Morin. An Architecture For Tolerating Processor Failures In Shared-Memory Multiprocessors. Rapport Technique 1965, INRIA, Mars 1993.

[Banâtre *et al.* 93b] M. Banâtre, A. Gefflaut, et C. Morin. Scalable Shared Memory Multiprocessors: Some Ideas to Make them Reliable. Dans *Proceedings of the Workshop on Hardware and Software Architectures for Fault Tolerance: Perpectives and Towards a Synthesis*, Le Mont Saint-Michel, Juin 1993.

[Barroso *et al.* 95] L. Barroso, S. Iman, J. Jeong, K. Öner, K. Ramamurthy, et M. Dubois. RPM: ARapid Prototyping Engine for Multiprocessor Systems. *IEEE Computer*, Février 1995.

[Bartlett *et al.* 87] J. Bartlett, J. Gray, et B. Horst. Fault Tolerance in Tandem Computer Systems. Dans A. Avizienis, H. Kopetz, et J.C. Laprie, éditeurs, *The Evolution of Fault-Tolerant Computing*, volume 1, pp 55–76. Springer Verlag, 1987.

[Bernstein 88] Ph. A. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 21(2):37–45, Février 1988.

[Birman & Joseph 85] K. Birman et T. Joseph. Reliable Communication in an Unreliable Environment. Rapport Technique TR 85-694, Computer Science Department, Cornell University, Ithaca, New York, Juillet 1985.

[Bolding & Yost 94] K. Bolding et W. Yost. Design of a Router for Fault-Tolerant Networks. Dans K. Bolding et L. Snyder, éditeurs, *Parallel Computer Routing and Communication*, volume 853 de *Lecture Notes in Computer Science*, pp 226–240, Mai 1994.

[Bowen & Pradhan 91] N.S. Bowen et D.K. Pradhan. A virtual Memory Translation Mechanism to Support Checkpoint and Rollback Recovery. Dans *Proc. of Supercomputing'91*, pp 890–899, Novembre 1991.

[Brown & Wu 94] L. Brown et J. Wu. Dynamic Snooping in a Fault-Tolerant Distributed Shared Memory. Dans *Proc. of 14th International Conference on Distributed Computing Systems*, pp 218–226, Juin 1994.

[Cabillic *et al.* 94] G. Cabillic, T. Priol, et I. Puaut. MYOAN: an Implementation of the KOAN Shared Virtual Memory on the Intel Paragon. Rapport Technique 812, IRISA, Avril 1994.

[Carter *et al.* 93] J.B. Carter, A.L. Cox, S. Dwarkadas, E.N. Elnozahy, D.B. Johnson, P. Keleher, S.Rodrigues, W. Yu, et W. Zwaenepoeo. Network Multicomputing Using Recoverable Distributed Shared Memory. Dans *Proc. of Spring Compcon93*, Février 1993.

[Censier & Feautrier 78] L. M. Censier et P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, Décembre 1978.

[Chaiken *et al.* 90] D. Chaiken, C. Fields, K. Kurihara, et A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, Juin 1990.

[Chaiken *et al.* 91] D. Chaiken, J. Kubiatowicz, et A. Agarwal. LimitLESS Directories : A Scalable Cache Coherence Scheme. Dans *Proc. of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 224–234, 1991.

[Chandy & Lamport 85] K. M. Chandy et L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Février 1985.

[Courtois *et al.* 92] B. Courtois, M.C. Gaudel, J.C. Laprie, et D. Powell. Sûreté de Fonctionnement Informatique. Evolutions 1987-1992 Tendances et Perspectives, Décembre 1992.

[Covington *et al.* 88] R. C. Covington, S. Madala, V. Metha, J.R. Jump, et J. B. Sinclair. The Rice Parallel Processing Testbed. Dans *Proc. of ACM SIGMetrics International Conference on Measurement and Modeling of Computer Systems*. ACM, 1988.

[Cox & Fowler 93] A.L. Cox et R.J. Fowler. Adaptative Cache Coherency for Detecting Migratory Shared Data. Dans *Proc. of 20th Annual International Symposium on Computer Architecture*, pp 98–108, 1993.

[Dally *et al.* 94] W.J. Dally, L.R. Dennison, D. Harris, K. Kan, et T Xanthopoulos. The Reliable Router : A Reliable and High-Performance Substrate for Parallel Computers. Dans K. Bolding et L. Snyder, éditeurs, *Parallel Computer Routing and Communication*, volume 853 de *Lecture Notes in Computer Science*, pp 241–280, Mai 1994.

[Davis *et al.* 91] H. Davis, S.R. Goldschmidt, et J. Hennessy. Multiprocessor Simulation Using Tango. Dans *Proc. of 1991 International Conference on Parallel Processing*, volume II, pp 99–107, Août 1991.

[Delp 88] G. Delp. *The Architecture and Implementation of MemNet : A High Speed Shared Memory Computer Communication Network*. Thèse de doctorat, University of Delaware, Computer Science Department, 1988.

[Denning 72] P. J. Denning. On Modeling Program Behavior. Dans *Proc. of Fall Joint Computer Conference*, pp 937–944, Atlantic City, Mai 1972. AFIPS.

[Dill 92] D.L. Dill. Protocol Verification as a Hardware Design Aid. Dans *Int'l Conf. on Computer Design: VLSI in Computers and Processors*, Octobre 1992.

[Din *et al.* 94] M.C. Din, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hönig, W. Hohl, E. Michel, et A. Pataricza. Fault Tolerance in Distributed Shared Memory Multiprocessors. Dans A. Bode et M.D. Cin, éditeurs, *Parallel Computer Architectures*, volume 732 de *Lecture Notes in Computer Science*, pp 31–48. Springer Verlag, 1994.

[D.L. Black 89] W.D. Weber D.L. Black, A. Gupta. Competitive Management of Distributed Shared Memory. Dans *In Spring COMPCON 89 Dig. Papers*, pp 184–190, Février 1989.

[Dubois *et al.* 86] M. Dubois, F.A. Briggs, I. Patil, et M. Balakrishnan. Trace-Driven Simulation of Parallel and Distributed Algorithms in Multiprocessors. Dans *Proc. of 1986 International Conference on Parallel Processing*, volume I, pp 505–508, Août 1986.

[Dubois *et al.* 88] M. Dubois, C. Scheurich, et F. A. Briggs. Synchronisation, Coherence and Event Ordering in Multiprocessors. *IEEE Computer Survey, Tutorial Series*, pp 9–21, Février 1988.

[E. L. Elnozahy & Zwaenepoel 92] D. B. Johnson E. L. Elnozahy et W. Zwaenepoel. The Performance of Consistent Checkpoint. Dans *Proc. of 11th Symposium on Reliable Distributed Systems*, pp 39–47, Octobre 1992.

[Eggers & Katz 88] S.J. Eggers et R.H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. Dans *Proc. of 15th Annual International Symposium on Computer Architecture*, pp 373–383, Honolulu, Mai 1988.

[Frank *et al.* 93] S. Frank, H. Burkhardt, et J. Rothnie. The KSR1 : Bridging the Gap Between Shared Memory and MPPs. Dans IEEE Computer Society, éditeur, *Proc. of spring COMPCON'93*, pp 285–294, Février 1993.

[Gefflaut & Joubert 96] A. Gefflaut et P. Joubert. SPAM : A Multiprocessor Execution Driven Simulation Kernel. *International Journal in Computer Simulation*, Á paraître, Janvier 1996.

[Gefflaut *et al.* 94] A. Gefflaut, C. Morin, et M. Banâtre. Tolerating Node Failures in Cache Only Memory Architectures. Dans *Proc. of Supercomputing'94*, Novembre 1994.

[Gharachorloo *et al.* 90] K. Gharachorloo, D. Lenosky, J. Laudon, P. Gibbons, A. Gupta, et J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. Dans *Proc. of 17th Annual International Symposium on Computer Architecture*, pp 15–26, 1990.

[Gharachorloo *et al.* 92] K. Gharachorloo, S.V. Adve, A. Gupta, et J.L. Hennessy ans M.D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, Août 1992.

[Goldberg *et al.* 90] A. Goldberg, A. Gopal, K. Li, R. Strom, et D. Bacon. Transparent Recovery of MACH Applications. Rapport Technique RC16242, IBM Research Division, T.J. Watson Research Center, Octobre 1990.

[Goodman & Woest 88] J.R. Goodman et P.J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. Dans *Proc. of 15th Annual International Symposium on Computer Architecture*, pp 422–431, Juin 1988.

[Goodman 83] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. Dans *Proc. of 10th Annual International Symposium on Computer Architecture*, pp 124–131, Stockholm, Juin 1983. ACM.

[Gray 78] J. Gray. *Notes on Database Operating Systems.*, volume 60 de *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[Gray 86] J.N. Gray. Why do computers stop and what can be done about it? Dans *Proc. of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pp 3–12, 1986.

[Gray 90] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 34(4):409–418, Octobre 1990.

[Gupta & Weber 92] A. Gupta et W.D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, Juillet 1992.

[Gupta *et al.* 91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, et W.D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. Dans *Proc. of 18th Annual International Symposium on Computer Architecture*, pp 254–263, 1991.

[Gustavson 92] D.B. Gustavson. The Scalable Coherent Interface and Related Projects. *IEEE Micro*, Février 1992.

[Hagersten *et al.* 91] E. Hagersten, P. Andersson, A. Landin, et S. Haridi. A Performance Study of the DDM - a Cache-Only Memory Architecture. Rapport Technique R91:17, Swedish Institute of Computer Science, Novembre 1991.

[Hagersten *et al.* 92] E. Hagersten, A. Landin, et S. Haridi. DDM - a Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, Septembre 1992.

[Hagersten *et al.* 94] E. Hagersten, A. Saulsbury, et A. Landin. Simple COMA Node Implementation. Dans *Hawaii International Conference on System Science*, Janvier 1994.

[Harris 88] David L. Harris. An Input/Output Subsystem for the Hawk Operating System Kernel. *Operating System Review*, 22(2):32–44, Avril 1988.

[Heinrich *et al.* 94] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, et J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. Dans *Proc. of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Octobre 1994.

[Henskens 93] F.A. Henskens. *A Capability-Based Persistent Distributed Shared Memory.* Thèse de doctorat, The University of Sydney, N.S.W. Australia 2006, Mars 1993.

[Hill & Smith 89] M. Hill et A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Décembre 1989.

[Horning *et al.* 74] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, et B. Randell. A Program Structure for Error Detection and Recovery. Dans *International Symposium on Operating Systems*, pp 171–187, Rocquencourt, France, Avril 1974.

[J. Hennessy 90] D. Patterson J. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[Jalote 94] P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

[Janakiraman & Tamir 94] G. Janakiraman et Y. Tamir. Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers. Dans *Proc. of 13th Symposium on Reliable Distributed Systems*, Dana Point, California, Octobre 1994.

[Janssens & Fuchs 91] B. Janssens et W.K. Fuchs. Experimental Evaluation of Multiprocessor Cache-Based Error Recovery. Dans *Proc. of 1991 International Conference on Parallel Processing*, volume I, pp 505–508, Août 1991.

[J.Kuskin *et al.* 94] J.Kuskin, D. Ofelt, M.Heinrich, J. Heinlein, R.Simoni, K.Gharachorloo, J.Chapin, D.Nakahira, J.Baxter, M. Horowitz, A.Gupta, M.Rosenblum, et J.Hennesy. The Stanford FLASH Multiprocessor. Dans *Proc. of 21th Annual International Symposium on Computer Architecture*, pp 302–313, Chicago, Illinois, Avril 1994.

[Joe & Hennessy 94] T. Joe et J.L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. Dans *Proc. of 21th Annual International Symposium on Computer Architecture*, pp 82–93, Chicago, Illinois, Avril 1994.

[Johnson 93] D.B. Johnson. Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs. Rapport Technique CMU-CS-93-117, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Mars 1993.

[Joubert 93] P. Joubert. *Conception et évaluation d'une architecture multiprocesseur à mémoire partagée tolérante aux fautes.* Thèse de doctorat, université de Rennes I, Janvier 1993.

[Katz *et al.* 85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, et R. G. Sheldon. Implementing a Cache Consistency Protocol. Dans *Proc. of 12th Annual International Symposium on Computer Architecture*, pp 276–283, Boston, 1985. IEEE.

[Koeninger *et al.* 94] R. K. Koeninger, M. Furtney, et M. Walker. A Shared Memory MPP from Cray Research. *Digital Technical Journal*, 6(2), 1994.

[Koo & Toueg 86] R. Koo et S. Toueg. Checkpointing and Rollback Recovery for Distributed Systems. Dans *Proc. of Fall Joint Computer Conference*, pp 1150–1158, Dallas, 1986.

[KSR 92] Technical Summary, 1992. Kendall Square Research.

[Lahjomri & Priol 92] Z. Lahjomri et T. Priol. KOAN: a Shared Virtual Memory for the iPSC/2 hypercube. Dans *CONPAR/VAPP92*, 1992.

[Lamport 78] L. Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114, 1978.

[Lampson 81] B. Lampson. Atomic Transactions. Dans *Distributed Systems and Architecture and Implementation : an Advanced Course*, volume 105 de *Lecture Notes in Computer Science*, pp 246–265. Springer Verlag, 1981.

[Laprie 85] J.C. Laprie. Dependable Computing and Fault-Tolerance: Concepts and Terminology. Dans *Proc. of 15th International Symposium on Fault-Tolerant Computing Systems*, pp 2–11, Ann Arbor, Michigan, Juin 1985.

[Lee & Anderson 90] P.A. Lee et T. Anderson. *Fault Tolerance: Principles and Practice*, volume 3 de *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, second revised edition, 1990.

[Lenoski *et al.* 92a] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, et M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, Mars 1992.

[Lenoski *et al.* 92b] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, et J. Hennessy. The DASH Prototype: Implementation and Performance. Dans *Proc. of 19th Annual International Symposium on Computer Architecture*, pp 92–103, Mai 1992.

[Li & Hudak 89] K. Li et P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–357, Novembre 1989.

[Lilja & Yew 93] D.J. Lilja et P.C. Yew. Improving Memory Utilization in Cache Coherence Directories. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):1130–1146, Octobre 1993.

[Litaize *et al.* 92] D. Litaize, A. Mzoughi, C. Rochange, et P. Sainrat. Towards a Shared Memory Massively Parallel Multiprocessor. Dans *Proc. of 19th Annual International Symposium on Computer Architecture*, pp 70–79, Mai 1992.

[Liu 93] L. Liu. Managing Coherence for Multi-Level Caches. Rapport technique, IBM, Yorktown Heights, NY 10598, 1993.

[Lorie 77] R.A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, Mars 1977.

[McCreight 84] E. M. McCreight. The Dragon Computer System, an Early Overview. Dans *Microarchitecture VLSI Computer*. NATO Advance Study Institute, Juillet 1984.

[McMillan 92] K.L. McMillan. *Symbolic Model Checking: An Approach to State Explosion Problem.* Thèse de doctorat, Carnegie Mellon University, Mai 1992.

[Michel 89] B. Michel. *Conception et réalisation de la mémoire virtuelle de GOTHIC.* Thèse de doctorat, université de Rennes I, Septembre 1989.

[Morin 90] C. Morin. *Protocole d'appel de multiprocédure à distance dans le système Gothic : définition et mise en œuvre.* Thèse de doctorat, université de Rennes I, Décembre 1990.

[NI & McKinley 93] L.M. NI et P.K McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pp 62–76, Février 1993.

[Nilsson & Stenström 93] H. Nilsson et P. Stenström. Performance Evaluation of Link-Based Cache Coherence Schemes. *Proc. of the 26th Annual Hawaï International Cpnference on System Sciences*, pp 486–495, 1993.

[Nilsson & Stenström 94] H. Nilsson et P. Stenström. An Adaptative Update-Based Cache Coherence Protocol for Reduction of Miss Rate and Traffic. Dans *Proc. of 6th International PARLE Conference*, volume 817 de *Lecture Notes in Computer Science*, pp 363–374. Springer Verlag, Juillet 1994.

[Oed 93] W. Oed. The Cray Research Massively Parallel Processor System CRAY T3D. Rapport technique, Cray Research GmbH, Novembre 1993.

[Papamarcos & Patel 84] M. S. Papamarcos et J. H. Patel. A Low Overhead Coherence Solution fo Multiprocessors with Private Cache Memories. Dans *Proc. of 11th Annual International Symposium on Computer Architecture*, pp 348–354, Ann Arbor, Juin 1984. IEEE.

[Pong & Dubois 93] F. Pong et M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. Rapport technique, Department of Electrical Engineering - Systems, University of Southern California, Avril 1993.

[Pong *et al.* 94] F. Pong, P. Stenström, et M. Dubois. An Integrated Methodology for the Verification of Directory-based Cache Protocols. Rapport technique, University of Southern California, 1994.

[Powell 91] D. Powell. *Delta-4 a Generic Architecture for Dependable Distributed Computing.* Springer Verlag, ISBN 3-540-54985-4, Berlin, 1991.

[Powell 94] D. Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, Février 1994.

[Randell 75] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(SE-2):220–232, Juin 1975.

[Reinhardt *et al.* 93] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, et D. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. Dans *Proc. of the 1993 ACM SIGMETRICS Conference*, Mai 1993.

[Reuter 80] A. Reuter. A Fast Transaction-Oriented Logging Scheme for Undo Recovery. *IEEE Transactions on Software Engineering*, SE-6(7):348–356, Juillet 1980.

[Richard-III & Singhal 93] G.G. Richard-III et M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. Dans *Proc. of 12th Symposium on Reliable Distributed Systems*, pp 58–67, Octobre 1993.

[R.P. Larowe 92] M. Holliday R.P. Larowe, C.S. Ellis. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Transactions on Parallel and Distributed Systems*, 3:686–701, Novembre 1992.

[Saavedra *et al.* 93] R.H. Saavedra, R. Stockton Gaines, et M.J. Carlton. Micro Benchmark Analysis of the KSR1. Dans *Supercomputing'93*, pp 202–213, Novembre 1993.

[Saulsbury *et al.* 95] A. Saulsbury, T. Wilkinson, J. Carter, et A. Landin. An Argument for Simple COMA. Dans *Proc. of 1st IEEE Symposium on High-Performance Computer Architecture*, Janvier 1995.

[Schimmel 90] C. Schimmel. UNIX on Modern Architectures. Dans *Summer 1990 USENIX Conference*, Anaheim, Juin 1990.

[Schneider 87] F. B. Schneider. The Fail-Stop Processor Approach. Dans *Concurency control and reliability in distributed systems, Chapitre 13*, pp 370–394. Barghava, 1987.

[Schwetman 92] H. Schwetman. CSIM User's Guide, Rev. 2. Rapport Technique ACT-126-90, Rev. 2, Microelectronics and Computer Technology Corporation, 1992.

[Simoni 92] R. Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. Thèse de doctorat, Standford University, Octobre 1992.

[Singh *et al.* 91] J.P. Singh, W.D. Weber, et A. Gupta. SPLASH : Stanford Parallel Applications for Shared-Memory. Rapport Technique CSL-TR-91-469, Computer Systems Laboratory, Stanford University, Avril 1991.

[Singh *et al.* 93] J.P. Singh, T. Joe, A. Gupta, et J.L. Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and the Stanford DASH Multiprocessors. Dans *Supercomputing'93*, pp 214–225, Novembre 1993.

[Smith 82] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, Septembre 1982.

[Sosnowski 94] J. Sosnowski. Transient Fault Tolerance in Digital Systems. *IEEE Micro*, pp 24–35, Février 1994.

[Spector *et al.* 85] A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, et R. Paush. Distributed Transactions for Reliable Systems. Dans *Proc. of 10th ACM Symposium on Operating Systems Principles*, Washington, 1985.

[Staknis 89] M.E. Staknis. Sheaved Memory: Architectural Support for State Saving and Restoration in Paged Systems. Dans *Proc. of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 96–102, Avril 1989.

[Stenström *et al.* 92] P. Stenström, T. Joe, et A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. Dans *Proc. of 19th Annual International Symposium on Computer Architecture*, pp 80–91, Mai 1992.

[Stenström 90] P. Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, Juin 1990.

[Strom & Yemini 85] R. E. Strom et S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Août 1985.

[Tang 76] C. K. Tang. Cache System Design in the Tightly Coupled Multiprocessor System. Dans *AFIPS Proceedings*, volume 45, 1976.

[Thacker *et al.* 88] C. P. Thacker, L. C. Stewart, et E. H. Satterthwaite. Firefly : A multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, Août 1988.

[Thapar 92] M. Thapar. *Cache Coherence for Scalable Shared Memory Multiprocessors.* Thèse de doctorat, Computer Systems Laboratory, Stanford University, Mai 1992.

[Torrellas *et al.* 92] J. Torrellas, A. Gupta, et J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. Rapport technique, Computer Systems Laboratory, Stanford University, 1992.

[Weber & Gupta 89] W.D. Weber et A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. Dans *Proc. of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Avril 1989.

[Weber 93] W.D. Weber. *Scalable Directories For Cache-Coherent Shared-Memory Multiprocessors.* Phd thesis, Computer Systems Laboratory, Stanford University, Janvier 1993.

[Wilkinson 93] T.J. Wilkinson. *Implementing Fault Tolerance in a 64-bit Distributed Operating System.* Thèse de doctorat, City University, London, Juillet 1993.

[Wilson 85] D. Wilson. The STRATUS Computer System. Dans T. Anderson, éditeur, *Resilient Computer Systems*, pp 208–231, 1985.

[Windheiser *et al.* 92] D. Windheiser, E.L. Boyd, E.H., et S.G. Abraham. KSR1 Multiprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver. Rapport technique, University of Michigan, Ann Arbor, 1992.

[Wood 85] W.G. Wood. Recovery Control of Communicating Processes in Distributed Systems. Dans S.K. Shrivastava, éditeur, *Reliable Computer Systems*, pp 448–484. Springer Verlag, 1985.

[Wu & Fuchs 89] K. L. Wu et W. K. Fuchs. Recoverable Distributed Shared Memory : Memory Coherence and Storage Structures. Dans *Proc. of 19th International Symposium on Fault-Tolerant Computing Systems*, pp 520–527, Chicago, Juin 1989.

[Wu & Fuchs 90] K.L Wu et W.K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, Avril 1990.

[Wu *et al.* 90] K.L. Wu, W.K. Fuchs, et J.H. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, Avril 1990.

[Yang *et al.* 92] Q. Yang, G. Thangadurai, et L.N. Bhuyan. Design of an Adaptative Cache Coherence Protocol for Large Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):281–293, Mai 1992.

Annexe A

Terminologie et principes de la tolérance aux fautes

La tolérance aux fautes a pour but d'assurer la conception et la réalisation de systèmes sûrs de fonctionnement c'est-à-dire permettant à leurs utilisateurs de placer une confiance justifiée dans les services qu'ils délivrent [Courtois *et al.* 92]. Elle a pour objectif de permettre à un système de continuer à délivrer le service pour lequel il est spécifié malgré le mauvais fonctionnement d'une partie de ce système.

A.1 Terminologie

La tolérance aux fautes utilise une terminologie précise. Nous donnons ici quelques définitions de termes utilisés dans le document.

Un système est dit **défaillant** quand le service qu'il délivre ne se conforme plus à sa spécification externe. A la base d'une défaillance se trouve une **faute**. Quand une partie fautive d'un système est utilisée, elle donne naissance à une **erreur** qui correspond à un état pouvant éventuellement mener à une défaillance du système. Une erreur est donc la manifestation d'une faute dans le système, alors qu'une défaillance est la manifestation d'une erreur sur le service délivré par le système.

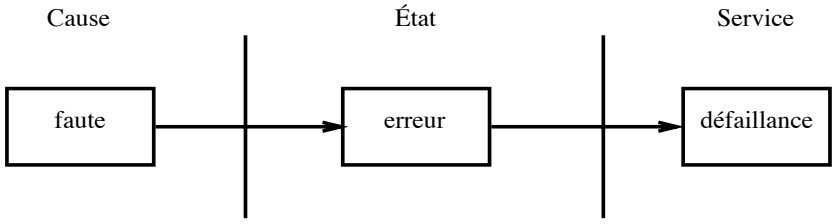


Figure A.1 Terminologie de la tolérance aux fautes

Ces définitions s'appliquent de façon récursive à des systèmes hiérarchiquement constitués. Ainsi la défaillance d'un sous-composant peut être considérée comme une faute du composant qui l'englobe. Dans une architecture multiprocesseur, la défaillance d'un nœud peut donc être vue comme une faute au niveau de l'architecture globale. Si cette faute n'est pas traitée elle peut mener à la défaillance globale de la machine. Le but de la tolérance aux fautes est donc de permettre à un système de tolérer la défaillance de ses composants afin que lui même ne connaisse pas une défaillance et que le service qu'il délivre ne soit pas affecté.

Les fautes sont extrêmement diverses [Laprie 85]. Elles peuvent notamment être classées en deux catégories suivant la durée de leurs effets. Une faute est dite **temporaire** si elle est présente dans un système pour une période limitée après laquelle elle disparaît spontanément. Dans le cas contraire, une faute est dite **permanente** [Lee & Anderson 90]. Les fautes temporaires sont le plus souvent dues à des distorsions externes au système considéré telles que des interférences électromagnétiques, des particules alpha ou des chutes de tension [Sosnowski 94]. On les appelle alors fautes **transitoires**. Les fautes temporaires sont habituellement beaucoup plus fréquentes que les fautes permanentes (90 à 98% des fautes matérielles détectées sont des fautes transitoires).

A.2 Étapes d'une technique de tolérance aux fautes

De façon simplifiée, toute technique de tolérance aux fautes se compose généralement des quatre étapes suivantes [Anderson & Lee 81, Jalote 94] :

(1) **Détection d'erreur.** C'est le point de départ de la tolérance aux fautes. La présence d'une faute est déduite d'une erreur détectée dans l'état d'un sous-système. Une erreur détectée implique généralement la défaillance du composant concerné.

(2) **Estimation des dommages.** Entre l'instant d'apparition d'une faute et la détection de sa manifestation, une faute a pu se propager à d'autres composants. Une fois l'erreur détectée, il est donc nécessaire d'évaluer l'étendu de l'extension de l'erreur dans le système. Cette phase peut être facilitée par des mécanismes de détection rapide d'erreur et de **confinement** d'erreur permettant de limiter l'extension des dommages.

(3) **Traitement d'erreur.** Cette étape a pour but de transformer l'état erroné du système en un état sain de façon à éviter que l'erreur ne conduise à une défaillance du système. Cette phase peut revêtir deux formes : le **recouvrement d'erreur** et la **compensation d'erreur**.

(4) **Traitement de fautes.** Cette phase a pour but d'assurer la non récurrence immédiate de l'erreur en empêchant une nouvelle activation de la faute qui peut encore être présente, c'est-à-dire en assurant sa **passivation**. Habituellement la passivation d'une faute est réalisée soit en réparant le composant défectueux, soit en en l'inhibant et en

déléguant sa charge du service aux composants encore valides (reconfiguration). Elle n'est donc pas nécessaire dans le cas d'une faute temporaire.

Dans la pratique, ces différentes phases sont souvent imbriquées ou interdépendantes. Certaines d'entre elles peuvent même ne pas exister

Ce travail de thèse s'intéresse surtout à la phase de traitement d'erreur en particulier aux techniques de recouvrement d'erreur. Les autres aspects d'une technique de tolérance aux fautes ne sont pas abordés. En particulier l'architecture étudiée suppose des nœuds ayant un mode de défaillance de type *silence sur défaillance* qui règle les problèmes de détection et de confinement d'erreur.

Annexe B

Vérification du protocole

Il est essentiel qu'un protocole de cohérence vérifie certaines propriétés invariantes. Habituellement, ces propriétés sont liées au respect de la cohérence [Pong *et al.* 94] et imposent certaines contraintes sur les états du système de cache considéré. Par exemple deux copies dans l'état *Exclusif* ne doivent jamais exister simultanément dans deux caches de l'architecture.

Dans le cas du protocole de cohérence étendu, d'autres invariants liés à la stabilité des données de récupération nous intéressent également. En particulier la présence de deux copies de récupération doit être assurée quelles que soient les fautes et y compris pendant les phases d'établissement de points de récupération.

B.1 Technique d'expansion des états

Faire la preuve de propriétés sur un protocole de cohérence n'est pas une tâche aisée. Les techniques de validation à base de simulation sont conceptuellement simples mais restent cependant insuffisantes car une séquence aléatoire de tests ne peut représenter l'ensemble des états accessibles par un protocole. Il est donc peu probable qu'une validation à partir d'une simulation détecte toutes les erreurs d'un protocole.

Une classe importante de techniques de vérification de protocoles dérive des méthodes d'énumération d'états (analyse d'accessibilité) [Dill 92]. Ces techniques consistent à explorer, de façon exhaustive, l'ensemble des états accessibles par le système considéré. Généralement, le processus d'expansion débute d'un état initial à partir duquel toutes les transitions tirables sont considérées produisant ainsi un ensemble de nouveaux états. Le procédé est appliqué de façon récursive pour chaque nouvel état jusqu'à ce qu'aucun nouvel état ne soit généré. L'ensemble des états accessibles forment alors un diagramme donnant les transitions entre états du système. Dans ce diagramme, les états ne vérifiant pas certaines propriétés sont appelés états erronés. Si un état erroné est accessible alors le protocole est incorrect.

Le problème majeur d'une technique d'énumération d'états, réside dans l'explosion de l'espace des états résultant de l'exploration exhaustive. Dans le cas de la vérification d'un protocole de cohérence, cette explosion d'états limite souvent le nombre de caches considérés pour la vérification du protocole.

Des travaux de recherche récents se sont intéressés à ce problème. Les techniques proposées diffèrent alors des méthodes d'énumération explicite des états en utilisant les équivalences entre états pour minimiser leur nombre. Ainsi un seul état global est utilisé pour représenter plusieurs états équivalents du système. Ces techniques portent le nom de **techniques d'expansion symbolique** des états [McMillan 92].

B.2 Technique d'expansion symbolique des états

Pour vérifier notre protocole de cohérence étendu, nous utilisons une technique d'analyse d'accessibilité décrite dans [Pong & Dubois 93]. Cette technique, dédiée aux protocoles de cohérence, ne considère qu'une seule ligne mémoire, ce qui est suffisant pour vérifier les propriétés qui nous intéressent. L'état du système est alors donné par l'état de la ligne dans chacun des caches de l'architecture. Cependant, l'espace des états est symboliquement étendu et représenté.

B.2.1 Équivalences entre états

La localisation particulière d'une copie étant sans intérêt du point de vue de la vérification du protocole, les caches dans le même état sont combinés dans une classe d'équivalence. Chaque classe d'équivalence correspond à un état du protocole de cohérence. Un état global est alors composé de classes d'équivalence auxquelles est ajouté le nombre d'éléments dans chacune des classes. Ainsi un état contenant une copie *Modifié Partagé*, deux copies *Partagé* et sept copies *Invalide* est codé par l'état global (*Modifié Partagé* ^{1}, *Partagé* ^{2}, *Invalide* ^{7}).

Dans tout protocole de cohérence, la cohérence est maintenue soit en diffusant les écritures à toutes les copies d'une ligne soit en les invalidant. Le nombre exact de copies d'une ligne dans un état particulier n'est donc pas utile du point de vue de la correction du protocole. En revanche, il peut être essentiel de savoir s'il existe 0, 1 ou plusieurs copies dans le même état (par exemple, plusieurs propriétaires pour une ligne indique une erreur dans le protocole).

Partant de cette constatation, les équivalences entre états peuvent être raffinées en groupant les états en états plus abstraits où le nombre exact de copies dans une classe d'équivalence n'est plus conservé mais se trouve codé par un opérateur. Dans notre cas, les opérateurs de répétition suivants sont utilisés pour représenter les états globaux[1].

[1]Les opérateurs 2 et +2 ont été rajoutés par rapport à la technique présentée dans [Pong & Dubois 93] pour pouvoir prouver la propriété de stabilité.

Définition 4 (Opérateurs de répétition)

1. *L'opérateur Nul (0) indique aucune instance.*
2. *L'opérateur Singleton (1) indique une et une seule instance.*
3. *L'opérateur Double (2) indique exactement deux instances.*
4. *L'opérateur Plus1 (+1) indique au moins une instance.*
5. *L'opérateur Plus2 (+2) indique au moins deux instances.*
6. *L'opérateur Étoile (*) indique zéro, une, deux ou plus de deux instances.*

Dans un système où le nombre de caches n'est pas spécifié, un ensemble de caches dans le même état est groupé dans une classe d'équivalence et le nombre de caches dans chaque classe est spécifié par un des opérateurs de répétition. Par exemple l'ensemble des états globaux du protocole tels que "au moins un cache possède une copie *Invalide*, exactement un cache possède une copie *Modifié Partagé* et aucun, un, deux, ou de multiples cache possèdent une copie *Partagé*" est symbolisé par l'état ($Invalide^{+1}$, $Modifié\ Partagé^{1}$, $Partagé^{*}$).

Cette représentation inclut un large nombre d'états qui, avec une méthode d'énumération traditionnelle, auraient été explicitement énumérés. De plus, elle est indépendante de la taille du système et permet donc de considérer des systèmes composés d'un nombre quelconque de caches.

Définition 5 (État composé) *Un état composé représente l'état d'un système de caches pour un nombre arbitraire de caches. Il est construit à partir de la classe d'états de la forme* $(q_1^{r_1}, q_2^{r_2}, ..., q_n^{r_n})$ *où* n *est le nombre d'états du protocole de cohérence,* $r_i \in [0, 1, 2, +1, +2, *]$ *et* q_i *sont les états du protocole.*

La représentation d'un état composé comporte toute l'information nécessaire à la vérification d'un protocole. Par exemple, elle permet de déterminer qu'un cache dans l'état *Partagé* ne cœxiste jamais avec un cache dans l'état *Exclusif*. Cette propriété peut se formuler par le fait qu'un état composé n'est jamais de la forme *(Exclusif* ^{1}*, Partagé* ^{1}*,...)*.

Les opérateurs de répétition peuvent être ordonnés suivant les états qu'ils spécifient. L'ordre résultant pour les opérateurs définis est alors le suivant :

$$1 < +1 < *$$
$$2 < +2 < *$$
$$+2 < +1$$
$$0 < *$$

Cet ordre débouche sur la définition de **couverture d'état** qui permet de minimiser le nombre d'états à conserver lors du processus d'expansion.

Définition 6 (Couverture) *Un état composé* S_2 *couvre un état composé* S_1 *, ou* $S_1 \subseteq S_2$*, si* $\forall q^{r_1} \in S_1, \exists q^{r_2} \in S_2$ *tel que* $q^{r_1} \leq q^{r_2}$ *i.e.* $r_1 \leq r_2$ *où* r_1 *et* r_2 *sont des opérateurs de répétition.*

Nous dirons aussi que S_1 est inclus dans S_2.

Une conséquence de la propriété de couverture est que si $S_1 \subseteq S_2$ alors la famille d'états représentée par S_2 inclut la famille d'états représentée par S_1. De plus, S_1 peut être détruit durant l'étape d'expansion pourvu que S_2 soit conservé. Dans [Pong & Dubois 93], il est démontré que le processus d'expansion des états est monotone sur l'ensemble des états composés, c'est-à-dire que si $S_1 \subseteq S_2$ alors $\tau(S_1) \subseteq \tau(S_2)$, où τ est un opérateur représentant une transition du protocole de cohérence. Ainsi pour tout état $\overline{S_1}$ accessible depuis S_1, il existe un état $\overline{S_2}$ accessible depuis S_2 tel que $\overline{S_1} \subseteq \overline{S_2}$. L'introduction de nouveaux opérateurs mènerait à une preuve similaire à celle qui est alors donnée.

Le processus d'expansion utilisé par cette méthode est conventionnel excepté que l'opération de comparaison entre états générés utilise la relation de couverture. Durant le processus d'expansion, de nouveaux états composés sont créés. Un nouvel état est détruit s'il est contenu dans un état déjà visité. De façon similaire, tout état déjà visité et inclus dans un nouvel état est détruit. A la fin du processus d'expansion, tous les états visités sont des états essentiels. Le nombre d'états générés est donc minimum.

Définition 7 (État essentiel) *Un état composé S est essentiel si et seulement si il n'existe pas un état composé $\overline{S}$ tel que $S \subseteq \overline{S}$.*

B.3 Expansion symbolique du protocole de cohérence étendu

B.3.1 Hypothèses et limites

Le protocole considéré est le protocole représenté sur la figure V.1 auquel l'état *Prevalidé*, utilisé lors de la sauvegarde d'un point de récupération, a été ajouté. Nous supposons les transitions du protocole de cohérence atomiques, c'est-à-dire que le temps nécessaire pour le changement d'états est nul. Cette hypothèse a pour but de simplifier la vérification des propriétés du protocole. Elle est valide dans le cas d'un multiprocesseur utilisant un bus où chaque transaction conserve le bus jusqu'à ce qu'elle soit terminée. Dans le cas d'une implémentation dans une architecture extensible, cette hypothèse n'est pas vérifiée et le protocole doit alors être enrichi d'états transitoires. Cependant la vérification devient beaucoup plus complexe car le modèle doit aussi prendre en compte les envois et réceptions de messages [Pong *et al.* 94]. De plus, le protocole utilisé dépend alors directement de la machine cible et une vérification devient nécessaire pour chaque architecture.

B.3.1.1 Protocole de cohérence

Pour la vérification, le protocole de cohérence étendu est défini par une machine d'états finis ayant une structure $M = (\mathcal{Q}, \Sigma, \sigma)$ où :

	Requêtes externes						**Requêtes internes**	
	Lec	Ecr	Inj-Exc	Inj-MP	Inj-PCK	Inj-ICK	Lec	Ecr
Inv	Inv	Inv	Exc ou Inv	MP	PCK	ICK	Par/Lec	Exc/Ecr
Par	Par	Inv	Exc ou Par	MP	PCK	ICK	Par	Exc/Ecr
Exc	MP	Inv	$\emptyset$	$\emptyset$	$\emptyset$	Excl	Exc	Exc
MP	MP	Inv	$\emptyset$	$\emptyset$	$\emptyset$	MP	MP	Exc/Ecr
PCK	PCK	ICK	$\emptyset$	$\emptyset$	PCK	PCK	PCK	Inv/Inj-PCK
ICK	ICK	ICK	ICK	ICK	ICK	ICK	Inv/Inj-ICK	Inv/Inj-ICK

Requêtes de gestion des points de récupération			
	EPR1	EPR2	RPR
Inv	Inv	Inv	Inv
Par	Pre ou Par	Par	Inv
Exc	Pre	PCK	Inv
MP	Pre	PCK	Inv
PCK	PCK	PCK	PCK
ICK	ICK	Inv	PCK

Abbréviations	
Inv	Invalide
Par	Partagé
Exc	Exclusif
MP	Modifié Partagé
PCK	Partagé-CK
ICK	Invalide-CK
Pre	Pré-validé
Lec	Lecture
Ecr	Écriture
Inj	Injection
$\emptyset$	Impossible

Table B.1 Transitions du protocole de cohérence étendu

- $\mathcal{Q}$ ={ *Exclusif, Modifié Partagé, Partagé, Invalide, Partagé-CK, Invalide-CK, Pré-Validé*}, l'ensemble des états définis par le protocole.

- Σ ={*Lecture, Écriture, EPR1, EPR2, RPR*}, l'ensemble des opérations responsables des transitions entre états, où EPR1 et EPR2 indiquent respectivement la première et la seconde phase du protocole d'établissement d'un point de récupération et RPR la restauration d'un point de récupération.

- σ fonction de transition $\mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$. Cette fonction est donnée par la table B.1 où chaque case représente le nouvel état de la ligne suivi éventuellement de la requête envoyée.

B.4 Expansion du protocole étendu

L'algorithme d'expansion symbolique des états est décrit dans [Pong & Dubois 93]. À partir d'un état initial, il permet de générer l'ensemble des états essentiels accessibles par un système de caches utilisant un protocole de cohérence.

Le processus d'expansion prend en compte tous les événements décrits par le protocole de cohérence étendu, c'est-à-dire l'ensemble de transitions définies par la fonction $\mathcal{F}$. L'établissement d'un point de récupération peut intervenir à tout moment. Cette opération se décompose en deux phases correspondant respectivement à l'établissement du nouveau point de récupération et à la validation du précédent. Entre ces deux phases aucune autre requête ne peut être réalisée. Lors de la phase de création, une copie *Partagé* est toujours choisie en priorité sur une copie *Invalide*, pour constituer la seconde copie *Pré-Validé*. Les fautes peuvent aussi intervenir à tout moment même durant l'établissement d'un point de récupération. Nous supposons cependant seulement l'occurrence de fautes temporaires car les fautes permanentes nécessitent une réallocation mémoire. L'ensemble des injections de lignes est également considéré. Lors d'une injection, une copie *Invalide* est toujours choisie en priorité sur une copie *Partagé*.

L'expansion débute de l'état initial (*Invalide*$^{+2}$, *Partagé-CK*^{2}) qui correspond à l'état de départ dans lequel une ligne mémoire est chargée dans l'architecture. La présence d'au moins deux copies *Invalide* assure qu'il existera toujours suffisamment de place pour la sauvegarde d'un point de récupération. Le processus d'expansion est automatique. Après 172 étapes, 12 états essentiels sont conservés et représentés dans le tableau B.4. L'ensemble des étapes du processus d'expansion est détaillé en B.6.

S0	$(Partage^*, Invalide^{+2}, PCK^2)$
S1	$(Partage^{+2}, Invalide^*, MP^1, ICK^2)$
S2	$(Invalide^{+1}, MP^1, ICK^2)$
S3	$(Invalide^{+1}, Exclusif^1, ICK^2)$
S4	$(Partage^1, Invalide^*, MP^1, ICK^2)$
S5	$(Invalide^*, ICK^2, PRE^2)$
S6	$(Partage^{+2}, Invalide^*, PCK^2)$
S7	$(Partage^{+1}, Invalide^{+1}, PCK^2)$
S8	$(Partage^{+1}, Invalide^{+1}, MP^1, ICK^2)$
S9	$(Partage^{+1}, Invalide^*, ICK^2, PRE^2)$
S10	$(Partage^*, Invalide^{+2}, MP^1, ICK^2)$
S11	$(Partage^*, Invalide^{+1}, ICK^2, PRE^2)$

Table B.2 Ensemble des états essentiels accessibles

La figure B.1 donne le diagramme des états essentiels accessibles ainsi que les transitions entre ces états. Sur ce diagramme, seules les transitions générant un changement d'état sont représentées. Par exemple, la lecture d'une copie *Partagé* n'apparaît pas sur ce diagramme.

Figure B.1 Diagramme global des états accessibles du protocole

B.5 Vérification des propriétés

Le diagramme des états essentiels est fortement connexe, ce qui indique qu'il n'existe pas d'états puits et donc garantit la vivacité du protocole et l'absence d'interblocage. Les propriétés que l'on souhaite vérifier sont de deux ordres. Les propriétés liées au respect de la cohérence, et les propriétés liées au respect de la stabilité des données de récupération. Le tableau B.3 résume l'ensemble de ces propriétés invariantes.

Propriétés de cohérence	
Propriété 1 :	il existe au plus une seule copie dans l'état *Exclusif*.
Propriété 2 :	il ne peut exister une copie *Exclusif* simultanément à des copies *Partagé*.
Propriété 3 :	il ne peut exister une copie *Exclusif* simultanément à des copies *Partagé-CK*.
Propriété 4 :	il ne peut exister une copie *Modifié Partagé* simultanément à des copies *Partagé-CK*.
Propriétés de stabilité	
Propriété 1 :	il existe toujours deux copies *Invalide-CK* ou deux copies *Partagé-CK*.
Propriété 2 :	il ne peut exister de copies *Invalide-CK* simultanément à des copies *Partagé-CK*.

Table B.3 Invariants à vérifier

En regardant les états essentiels générés par le protocole de cohérence, il apparaît qu'aucun de ces invariants n'est violé. Le protocole de cohérence étendu vérifie donc, avec les hypothèses d'atomicité des transitions, les propriétés de cohérence et de stabilité des données de récupération et ceci quel que soit l'état du système.

Bien entendu, cette vérification réalisée en supposant les transitions entre états atomiques, ne convient pas pour une implémentation réelle du protocole étendu dans une architecture extensible où la propriété d'atomicité des transitions n'est plus vérifiée. Elle valide cependant le protocole et montre que l'approche suivie est correcte. De plus, l'expansion des états réalisée a aussi pour intérêt de donner une représentation simple de l'ensemble des états accessibles par un système utilisant ce protocole de cohérence étendu.

B.6 Étapes du processus d'expansion

Nous donnons ici les différentes étapes du processus d'expansion automatique des états. À gauche figure l'ancien état, à droite l'état résultant de la transition.

$(Inv^{+2}, PCK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Inv^{+2}, PCK^2) \rightarrow Lec\ Inv \rightarrow (Par^1, Inv^{+1}, PCK^2)$

$(Inv^{+2}, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+2}, PCK^2) \rightarrow Lec\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+2}, PCK^2) \rightarrow EPR1\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+2}, PCK^2) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+2}, PCK^2) \rightarrow Inj\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow Lec\ Inv \rightarrow (Par^1, Inv^*, MP^1, ICK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow Ecr\ Exc \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow Lec\ Exc \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow EPR1\ Exc \rightarrow (Inv^*, ICK^2, PR2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow RPR\ Exc \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow Inj\ Exc \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, Exc^1, ICK^2) \rightarrow Inj\ ICK \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow Ecr\ Par \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow Lec\ Par \rightarrow (Par^1, Inv^{+1}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow Remp\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow Lec\ Inv \rightarrow (Par^2, Inv^*, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow Lec\ PCK \rightarrow (Par^1, Inv^{+1}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow EPR1\ PCK \rightarrow (Par^1, Inv^{+1}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow Inj\ PCK \rightarrow (Par^1, Inv^{+1}, PCK^2)$
$(Inv^{+2}, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+2}, PCK^2) \rightarrow EPR2\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+2}, PCK^2) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Ecr\ Par \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Lec\ Par \rightarrow (Par^1, Inv^*, MP^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Remp\ Par \rightarrow (Inv^{+1}, MP^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Lec\ Inv \rightarrow (Par^2, Inv^*, MP^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Ecr\ MP \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Lec\ MP \rightarrow (Par^1, Inv^*, MP^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow EPR1\ MP \rightarrow (Inv^*, ICK^2, PR2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow RPR\ MP \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Inj\ MP \rightarrow (Inv^{+1}, MP^1, ICK^2)$
$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^2)$

$(Par^1, Inv^*, MP^1, ICK^2) \rightarrow Inj\ ICK \rightarrow (Inv^{+1}, MP^1, ICK^2)$
$(Inv^*, ICK^2, PR2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^*, ICK^2, PR2) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^*, ICK^2, PR2) \rightarrow EPR2\ PRE \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^*, ICK^2, PR2) \rightarrow RPR\ PRE \rightarrow (Inv^{+2}, PCK^2)$
$(Par^2, Inv^*, PCK^2) \rightarrow Ecr\ Par \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^2, Inv^*, PCK^2) \rightarrow Lec\ Par \rightarrow (Par^2, Inv^*, PCK^2)$
$(Par^2, Inv^*, PCK^2) \rightarrow Remp\ Par \rightarrow (Par^1, Inv^{+1}, PCK^2)$
$(Par^2, Inv^*, PCK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^2, Inv^*, PCK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^2, Inv^*, PCK^2) \rightarrow Lec\ Inv \rightarrow (Par^{+2}, Inv^*, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow EPR2\ PCK \rightarrow (Par^1, Inv^{+1}, PCK^2)$
$(Par^1, Inv^{+1}, PCK^2) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow Lec\ Inv \rightarrow (Par^1, Inv^*, MP^1, ICK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow Ecr\ MP \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow Lec\ MP \rightarrow (Inv^{+1}, MP^1, ICK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow EPR1\ MP \rightarrow (Inv^*, ICK^2, PR2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow RPR\ MP \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow Inj\ MP \rightarrow (Inv^{+1}, MP^1, ICK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^2)$
$(Inv^{+1}, MP^1, ICK^2) \rightarrow Inj\ ICK \rightarrow (Inv^{+1}, MP^1, ICK^2)$
$(Par^2, Inv^*, MP^1, ICK^2) \rightarrow Ecr\ Par \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^2, Inv^*, MP^1, ICK^2) \rightarrow Lec\ Par \rightarrow (Par^2, Inv^*, MP^1, ICK^2)$
$(Par^2, Inv^*, MP^1, ICK^2) \rightarrow Remp\ Par \rightarrow (Par^1, Inv^{+1}, MP^1, ICK^2)$
$(Par^2, Inv^*, MP^1, ICK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^2, Inv^*, MP^1, ICK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^2, Inv^*, MP^1, ICK^2) \rightarrow Lec\ Inv \rightarrow (Par^{+2}, Inv^*, MP^1, ICK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow Ecr\ Par \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow Lec\ Par \rightarrow (Par^{+2}, Inv^*, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow Remp\ Par \rightarrow (Par^{+1}, Inv^{+1}, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow Lec\ Inv \rightarrow (Par^{+2}, Inv^*, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow Lec\ PCK \rightarrow (Par^{+2}, Inv^*, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow EPR1\ PCK \rightarrow (Par^{+2}, Inv^*, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+2}, Inv^*, PCK^2) \rightarrow Inj\ PCK \rightarrow (Par^{+1}, Inv^{+1}, PCK^2)$

$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Ecr\ Par \rightarrow (Inv^{+2}, Exc^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Lec\ Par \rightarrow (Par^{+2}, Inv^{*}, MP^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Remp\ Par \rightarrow (Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Ecr\ Inv \rightarrow (Inv^{+2}, Exc^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Lec\ Inv \rightarrow (Par^{+2}, Inv^{*}, MP^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Ecr\ MP \rightarrow (Inv^{+2}, Exc^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Lec\ MP \rightarrow (Par^{+2}, Inv^{*}, MP^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow EPR1\ MP \rightarrow (Par^{+1}, Inv^{*}, ICK^{2}, PR2)$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow RPR\ MP \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Inj\ MP \rightarrow (Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2}) \rightarrow Inj\ ICK \rightarrow (Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow Ecr\ Par \rightarrow (Inv^{+1}, Exc^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow Lec\ Par \rightarrow (Par^{+1}, Inv^{+1}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow Remp\ Par \rightarrow (Par^{*}, Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow Ecr\ Inv \rightarrow (Inv^{+1}, Exc^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow Lec\ Inv \rightarrow (Par^{+2}, Inv^{*}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow Lec\ PCK \rightarrow (Par^{+1}, Inv^{+1}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow EPR1\ PCK \rightarrow (Par^{+1}, Inv^{+1}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2}) \rightarrow Inj\ PCK \rightarrow (Par^{+1}, Inv^{+1}, PCK^{2})$
$(Par^{+2}, Inv^{*}, PCK^{2}) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+2}, Inv^{*}, PCK^{2}) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+2}, Inv^{*}, PCK^{2}) \rightarrow EPR2\ PCK \rightarrow (Par^{+2}, Inv^{*}, PCK^{2})$
$(Par^{+2}, Inv^{*}, PCK^{2}) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Ecr\ Par \rightarrow (Inv^{+2}, Exc^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Lec\ Par \rightarrow (Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Remp\ Par \rightarrow (Par^{*}, Inv^{+2}, MP^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Ecr\ Inv \rightarrow (Inv^{+2}, Exc^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Lec\ Inv \rightarrow (Par^{+2}, Inv^{*}, MP^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Ecr\ MP \rightarrow (Inv^{+2}, Exc^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Lec\ MP \rightarrow (Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow EPR1\ MP \rightarrow (Par^{*}, Inv^{+1}, ICK^{2}, PR2)$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow RPR\ MP \rightarrow (Inv^{+2}, PCK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow Inj\ MP \rightarrow (Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2}) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^{2})$

$(Par^{+1}, Inv^{+1}, MP^1, ICK^2) \rightarrow Inj\ ICK \rightarrow (Par^{+1}, Inv^{+1}, MP^1, ICK^2)$
$(Par^{+1}, Inv^{*}, ICK^2, PR2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+1}, Inv^{*}, ICK^2, PR2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+1}, Inv^{*}, ICK^2, PR2) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+1}, Inv^{*}, ICK^2, PR2) \rightarrow EPR2\ PRE \rightarrow (Par^{+1}, Inv^{+2}, PCK^2)$
$(Par^{+1}, Inv^{*}, ICK^2, PR2) \rightarrow RPR\ PRE \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow Ecr\ Par \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow Lec\ Par \rightarrow (Par^{+1}, Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow Remp\ Par \rightarrow (Par^{*}, Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+1}, Exc^1, ICK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow Lec\ Inv \rightarrow (Par^{+1}, Inv^{+1}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow Lec\ PCK \rightarrow (Par^{*}, Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow EPR1\ PCK \rightarrow (Par^{*}, Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow Inj\ PCK \rightarrow (Par^{*}, Inv^{+2}, PCK^2)$
$(Par^{+1}, Inv^{+1}, PCK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+1}, Inv^{+1}, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{+1}, Inv^{+1}, PCK^2) \rightarrow EPR2\ PCK \rightarrow (Par^{+1}, Inv^{+1}, PCK^2)$
$(Par^{+1}, Inv^{+1}, PCK^2) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Ecr\ Par \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Lec\ Par \rightarrow (Par^{+1}, Inv^{+2}, MP^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Remp\ Par \rightarrow (Par^{*}, Inv^{+2}, MP^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Ecr\ Inv \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Lec\ Inv \rightarrow (Par^{+1}, Inv^{+1}, MP^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Ecr\ MP \rightarrow (Inv^{+2}, Exc^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Lec\ MP \rightarrow (Par^{*}, Inv^{+2}, MP^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow EPR1\ MP \rightarrow (Par^{*}, Inv^{+1}, ICK^2, PR2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow RPR\ MP \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Inj\ MP \rightarrow (Par^{*}, Inv^{+2}, MP^1, ICK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, MP^1, ICK^2) \rightarrow Inj\ ICK \rightarrow (Par^{*}, Inv^{+2}, MP^1, ICK^2)$
$(Par^{*}, Inv^{+1}, ICK^2, PR2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+1}, ICK^2, PR2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+1}, ICK^2, PR2) \rightarrow RPR\ ICK \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+1}, ICK^2, PR2) \rightarrow EPR2\ PRE \rightarrow (Par^{*}, Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+1}, ICK^2, PR2) \rightarrow RPR\ PRE \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow RPR\ Par \rightarrow (Inv^{+2}, PCK^2)$
$(Par^{*}, Inv^{+2}, PCK^2) \rightarrow RPR\ Inv \rightarrow (Inv^{+2}, PCK^2)$

$(Par^{*}, Inv^{+2}, PCK^{2}) \rightarrow EPR2\ PCK \rightarrow (Par^{*}, Inv^{+2}, PCK^{2})$
$(Par^{*}, Inv^{+2}, PCK^{2}) \rightarrow RPR\ PCK \rightarrow (Inv^{+2}, PCK^{2})$
États essentiels générés
$(Par^{+1}, Inv^{*}, ICK^{2}, PR2)$
$(Par^{+2}, Inv^{*}, PCK^{2})$
$(Par^{+2}, Inv^{*}, MP^{1}, ICK^{2})$
$(Inv^{+1}, MP^{1}, ICK^{2})$
$(Inv^{+1}, Exc^{1}, ICK^{2})$
$(Par^{1}, Inv^{*}, MP^{1}, ICK^{2})$
$(Inv^{*}, ICK^{2}, PR2)$
$(Par^{+1}, Inv^{+1}, MP^{1}, ICK^{2})$
$(Par^{+1}, Inv^{+1}, PCK^{2})$
$(Par^{*}, Inv^{+2}, MP^{1}, ICK^{2})$
$(Par^{*}, Inv^{+1}, ICK^{2}, PR2)$
$(Par^{*}, Inv^{+2}, PCK^{2})$

Annexe C

Glossaire

Ce glossaire rassemble une partie des sigles utilisés dans le reste du document.

CARER : $Cache-AidedRollbackErrorRecovery$. Implémentation d'une technique de récupération arrière utilisant les caches d'un multiprocesseur à mémoire partagée pour conserver les données modifiées.

CC-NUMA : $CacheCoherentNonUniformMemoryAccess$. Caractérise une architecture à accès mémoire non uniforme qui utilise des caches maintenus cohérents par un protocole de cohérence matériel.

COMA : $CacheOnlyMemoryArchitecture$. Architecture où les mémoires font office de caches de grande dimension.

DASH : Architecture extensible à mémoire partagée proposée par l'université de Stanford.

DDM : $DataDiffusionMachine$. Architecture COMA utilisant une hiérarchie de bus, proposée par le *Swedish Institute of Computer Science*.

Flat-COMA : architecture COMA non-hiérarchique.

FLASH : $FlexibleArchitectureforSHaredmemory$. Multiprocesseur extensible à mémoire partagée développé à l'université de Stanford.

KSR1 : Architecture COMA utilisant une hiérarchie d'anneaux proposée par Kendall Square Researh.

MA : $MmoireAttractive$. Nom d'une mémoire utilisée dans une architecture COMA.

NUMA : $NonUniformMemoryAccess$. Caractérise une architecture où les accès mémoire ne sont pas uniformes.

SCI : $ScalableCoherentInterface$. Norme IEEE définissant des liens de communication ainsi qu'un protocole de cohérence à base de répertoires chaînés.

SPLASH : $StanfordParallelApplicationsforShared-Memory$. Suite d'applications scientifiques développées par l'université de Stanford pour l'évaluation des multiprocesseurs

à mémoire partagée.

SRAM : $StaticRAM$. Mémoire vive statique à accès rapide. Utilisée notamment dans les caches.

UMA : $UniformMemoryAccess$. Caractérise une architecture où les accès mémoire sont uniformes pour tous les processeurs, par exemple un multiprocesseur à mémoire partagée utilisant un bus.

Table des Matières

Table des Figures

Table des Tables

Résumé

Les architectures extensibles à mémoire partagée sont une réponse aux besoins croissants des applications en puissance de calcul. Elles allient la facilité de programmation offerte par l'utilisation d'un espace d'adressage unique, à la puissance de calcul d'un nombre élevé de processeurs. Malgré l'augmentation importante de la fiabilité des composants matériels, le nombre de composants de ces architectures leur confère une probabilité de défaillance élevée. Il devient donc nécessaire de leur fournir des mécanismes de tolérance aux fautes qui leur permettent d'assurer la continuité du service qu'elles rendent malgré la défaillance de certains de leurs composants.

Pour améliorer leur efficacité, les architectures extensibles à mémoire partagée utilisent des caches leur permettant de répliquer localement des données accédées à distance. Associés à ces caches, des protocoles de cohérence se chargent d'assurer la cohérence des données répliquées. Les architectures COMA (Cache Only Memory Architectures), où les mémoires font office de caches de grande dimension de l'espace partagé, sont l'aboutissement de cette politique. D'un autre côté, la récupération arrière est une technique de tolérance aux fautes qui nécessite la conservation et la réplication de données de récupération. L'approche suivie dans cette thèse vise à tirer parti des mécanismes de réplication de données offerts par les architectures COMA pour assurer la redondance nécessaire à l'implémentation d'une stratégie de récupération arrière de façon à limiter le développement matériel et la dégradation de performance engendrée. Dans ce but, nous proposons une extension du protocole de cohérence utilisé par les mémoires de l'architecture afin qu'elles intègrent de façon transparente la gestion des données de récupération à celle des données courantes. Ce protocole de cohérence étendu permet ainsi de traiter les défaillances des nœuds de l'architecture en limitant le matériel spécifique, simplement en utilisant les mémoires et les mécanismes de réplication de données d'une architecture COMA pour conserver et répliquer les données de récupération.

Une évaluation du protocole de cohérence proposé, effectuée par simulation, montre que la dégradation de performance engendrée par son utilisation est faible par rapport à une architecture utilisant un protocole de cohérence standard et ceci même lorsque les fréquences de sauvegarde des points de récupération sont élevées. L'utilisation des mécanismes de réplication de données et des mémoires, pour stocker les données de récupération, assure en effet des sauvegardes de points de récupération efficaces. Il est également montré que l'utilisation du protocole de cohérence étendu n'entrave pas l'extensibilité de l'architecture.

Mots clés

Multiprocesseurs extensibles à mémoire partagée, récupération arrière, réplication de données, protocoles de cohérence, évaluation de performances, simulation.