Evaluation d'architectures parallèles à mémoire virtuelle partagée distribuée : étude et réalisation d'un émulateur

Olivier Jacquiot

THÈSE

Présentée par

Olivier Jacquiot

pour obtenir le titre de Docteur

de l'Institut National Polytechnique de Grenoble

(arrêté ministériel du 30 mars 1992)

Spécialité: Informatique

Evaluation d'architectures parallèles à mémoire virtuelle partagée distribuée: étude et réalisation d'un émulateur.

Date de soutenance: 27 septembre 1996

Composition du jury:

Paul Jacquet	Président
Jean-Paul Sansonnet	Rapporteur
Traïan Muntean	Rapporteur
Jacques Briat	Examinateur
Gilles Berger-Sabbatel	Examinateur
Guy Mazaré	Directeur de thèse

Thèse préparée au sein du

laboratoire Logiciels Systèmes Réseaux

(institut d'Informatique et de Mathématiques Appliquées de Grenoble)

A mes parents.

Remerciements.

Je tiens à remercier très vivement M. Paul Jacquet, Professeur à l'Institut National Polytechnique de Grenoble et directeur du laboratoire Logiciels Systèmes Réseaux, pour m'avoir fait l'honneur de présider mon jury de soutenance et de m'avoir hébergé au sein du L.S.R..

J'aimerais exprimer toute ma reconnaissance à M. Jean-Paul Sansonnet, directeur de recherche au CNRS et à M. Traïan Muntean, Professeur à l'université Marseille II pour avoir accepter de rapporter sur mon travail. Je les remercie pour le temps qu'ils ont passé à l'étudier et pour leurs remarques judicieuses qu'ils m'ont faites.

Mes remerciements vont aussi à M. Jacques Briat, Maître de conférence, à l'université Joseph Fourier et à M. Gilles Berger-Sabbatel, chargé de recherche du CNRS, non seulement pour avoir accepté d'être examinateur lors de mon jury de soutenance, mais aussi pour toutes les remarques qu'ils ont pu me faire tout au long de mon travail.

Je voudrais exprimer toute ma gratitude à M. Guy Mazaré, Professeur à l'Institut National Polytechnique de Grenoble et Directeur de l'ENSIMAG, de m'avoir accueilli, il y a déjà quelques années lors d'une recherche de stage de magistère et de m'avoir par la suite intégré à son équipe de recherche, lors de mon DEA et pour cette thèse. Je le remercie très sincèrement de l'opportunité qu'il m'a donné en travaillant dans son équipe et pour tous ses conseils qu'il m'a prodigués.

Mes remerciements ne seraient pas complets, si je ne remerciais pas tous les membres du LGI puis du LSR permanents et non permanents (en particulier Ridha Djemal, Olivier Ondoa, Daniel Conil et Jean-François Guillaud). Leur aide et leur sympathie m'ont permis de passer quatre années agréables au sein du laboratoire.

Pour finir, mes plus profonds sentiments vont à ma famille, qui m'a toujours soutenue et encouragée dans ce que je faisais.

Table des matières

Introduction

Jusqu'à ces dernières années, le domaine des systèmes répartis et celui des machines parallèles cohabitaient sans trop se rencontrer. Or, depuis quelque temps, leurs relations ont sensiblement évolué. Les machines individuelles sont devenues, grâce à des progrès dans les technologies d'intégration, de plus en plus puissantes. Parallèlement à ceci, les possibilités de relier les ordinateurs sont devenues très intéressantes, du fait de l'apparition de réseaux très rapides et de nouveaux protocoles. Ces deux évolutions renforcent la similarité que l'on peut trouver entre une machine parallèle centralisée et un réseau de micro-ordinateurs, ce qui implique que les moyens pour exploiter de telles configurations aient donc tendance à être de plus en plus semblables.

Un mode de transmission à haut débit a été largement étudié au sein de l'équipe GRAM du LSR, il s'agit d'ATM (Asynchronous Transfert Mode). ATM est basé sur la commutation de petits paquets de longueur fixe appelé cellules (53 octets réels, soit une quarantaine d'octets utiles par cellule). Cette méthode de transfert est très attractive car outre les très bonnes performances qu'elle laisse présager, les circuits pour réaliser une telle technologie doivent dans l'avenir devenir très peu chers du fait de leur utilisation en parallèle par le domaine des télécommunications. Pour avoir une présentation détaillée de l'ATM, il faut se référer à [Bou92].

Le groupe GRAM, c'est donc intéressé à ATM et, entre autres, à son utilisation pour interconnecter des machines, ou même pour réaliser des réseaux d'interconnexions à l'intérieur des machines parallèles. Le problème de l'équipe est d'évaluer les différentes topologies d'interconnexions possibles réalisées par ATM, afin de fournir un service optimal aux applications qui s'exécutent sur ces machines. Mais la charge sur le réseau est induite par les applications elles-mêmes. C'est pourquoi il est nécessaire d'évaluer cette charge pour définir une topologie.

Or, dans le domaine des multi-processeurs, deux modèles de programmation cohabitent et influencent différemment la charge induite sur les réseaux d'interconnexion. Il s'agit du passage de message et de la mémoire partagée. Historiquement, chacun est apparu avec un type de machine spécifique, le passage de message avec les machines à mémoire

distribuée, et la mémoire partagée avec les machines en offrant une. L'apparition des machines à mémoire distribuée utilisées comme des machines à mémoire partagée, a poussé le modèle de programmation à mémoire partagée sur des machines qui ne pouvaient pas l'accueillir auparavant. De ce point de vue, le modèle à mémoire partagée tend à devenir un modèle universel et fait de l'ombre au passage de message. Celui-ci contient cependant intrinsèquement une propriété qui peut être vue comme un inconvénient ou un avantage, c'est le fait que chaque communication doit être prévue et traitée par le programmeur pour les deux entités communicantes. C'est un inconvénient par rapport à l'autre modèle où les communications sont transparentes du point de vue du programmeur. Mais, c'est un avantage car cela fournit un moyen simple de synchronisation entre les différents flots d'exécution que ne peut pas fournir simplement le modèle à mémoire partagée qui doit faire appel à des primitives de synchronisation. Une comparaison de ces deux modèles n'est pas facile, car pour chaque problème, il faut écrire deux versions radicalement différentes. C'est ce que montrent Klaiber et Levy dans [KL94]. Comme les deux modèles de programmation ont chacun leurs propres avantages et leurs propres inconvénients, de récentes études essaient de ne plus les opposer, mais de jouer sur leur complémentarité. C'est le cas de [KJA+93] et de [FV93].

Dans la suite de cette étude, seule l'approche des machines utilisant une mémoire partagée est abordée. Les machines à passage de messages sont étudiées dans d'autres travaux de l'équipe.

De la volonté d'unifier les machines parallèles et les réseaux de machines individuelles est apparue la mémoire virtuelle répartie qui semble réaliser l'union, entre les deux approches, de manière simple et efficace. Cependant, sous ce terme, coexistent plusieurs types d'organisation de la mémoire qui sont très différents et qu'il ne faut pas confondre. Toutes ces méthodes, ont pour objet de présenter au programmeur une mémoire partagée, même s'il n'en existe pas réellement dans le système, mais le niveau de fabrication de cette mémoire logiquement partagée peut varier. Ces méthodes ont fait l'objet d'une tentative de classement par Sanjay Raina dans [Rai92] et sont résumées dans la figure 0.1. Bien que ce classement ne soit pas unanimement reconnu [LKBT92], nous l'utilisons tout de même pour définir l'architecture à laquelle nous nous sommes intéressés.

La DSM (Distributed Shared Memory) est un système qui autorise l'accès à distance des données, mais pas leur réplication. De ce fait, aucun problème de cohérence n'est rencontré. C'est le premier système qui soit apparu, on le rencontre déjà dans le projet RP3 d'IBM [PBG+85] et dans Butterfly [LSB88]. Les performances de tels systèmes ne sont pas satisfaisantes, pour des applications qui partagent un grand nombre de données.

La VSM (Virtual shared Memory) crée à partir des mémoires distribuées, un espace

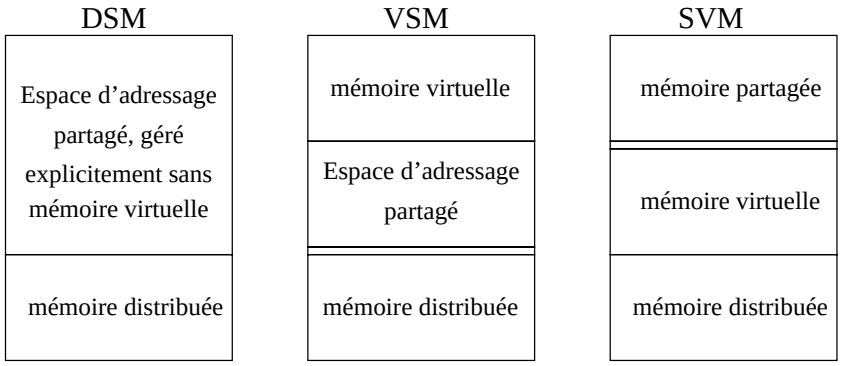


FIG. 0.1 – *Différentes manières d'appréhender une mémoire partagée au-dessus de mémoires physiques distribuées.*

d'adressage commun, partagé par tous les noeuds du système. Cette mémoire partagée est souvent réalisée au moyen de dispositifs matériels et l'unité de partage est généralement de la taille d'une ligne de cache. Une fois l'espace commun réalisé, la gestion de la mémoire virtuelle peut être conçue par-dessus. DASH [LLG+92] et DDM [HLH92] sont des exemples de telles architectures.

La SVM (Shared Virtual Memory) se base sur la gestion de la mémoire virtuelle et crée au-dessus de ce mécanisme une mémoire partagée. La cohérence est ici gérée par des fonctionnalités spéciales du gérant des fautes de pages et l'unité de partage est le plus souvent la page. IVY [LH89], MUNIN [CBZ91], KOAN [LP92] et Treadmarks [AcD+96] en sont des exemples.

Pour avoir une bibliographie très complète sur les problèmes de partage des données dans un environnement distribué, il faut se référer à [Esk96] où l'on peut trouver plus de 400 références classées en diverses catégories sur le sujet.

Des trois systèmes présentés précédemment, un nous a particulièrement intéressés, il s'agit de la SVM. Celle-ci est en effet très attrayante pour permettre de réaliser des tâches en parallèle, avec une grande indépendance vis-à-vis du matériel utilisé (on peut même envisager que celui-ci soit hétérogène), et ceci contrairement aux deux autres approches qui dépendent d'une configuration matérielle spécifique. De plus, il semble possible de réutiliser un bon nombre de techniques des multi-processeurs à mémoire communes car la seule différence fondamentale des deux techniques est le grain d'échanges des données. Celui-ci passe de la taille d'une ligne de cache à la taille d'une page. Il faut donc, pouvoir vérifier que ce changement d'échelle du grain d'échange, n'entraîne pas une perte d'efficacité importante et surtout que le transfert de page de plusieurs Koctets s'effectue correctement et exploite de façon efficace un réseau ATM.

Le but de ce travail est donc de fournir un outil qui permette d'appréhender la charge induite sur le réseau d'interconnexion d'une machine parallèle qui utilise la SVM. Ceci est très difficile à obtenir, en effet, la simulation est trop lente et l'utilisation de traces mémoires n'est pas assez dynamique. Donc, il faut émuler la machine en se basant sur l'exécution directe des programmes et interpréter tous les transferts de pages en tenant compte du fait que le comportement du système est aussi fonction de la nature de la cohérence des données retenue. Le type de cohérence influe sur les besoins en pages de chaque entité du système et donc modifie l'échange des pages sur le réseau.

C'est pour tout cela que nous présentons dans cette thèse, la réalisation d'un système d'évaluation basé sur l'exécution directe (appelé par la suite émulateur). Pour le réaliser, il nous faut donc trouver un moyen simple pour simuler le parallélisme et intercepter les fautes de pages. Il faut aussi que «l'outil» retenu soit flexible à souhait. C'est pourquoi, le micro-noyau Mach [ABB+86] qui est un système très ouvert et dont nous disposons des sources, nous a semblé être un moyen bien approprié. En effet, à l'aide des «threads», il est possible de simuler l'exécution directe de plusieurs flots d'exécution en parallèle sur un mono-processeur, et, grâce à la possibilité de définir des paginateurs externes au noyau, il est possible non seulement d'intercepter les fautes de pages, mais aussi, de leur appliquer le traitement spécifique souhaité. Une fois ce choix fait, nous avons donc développé un prototype d'évaluation d'un système à base de SVM et prouver sa viabilité par des expérimentations.

Cette thèse est divisée en deux parties, la première permet de faire un tour d'horizon sur les hiérarchies mémoires et la cohérence des données et la seconde présente la réalisation d'un émulateur pour un système basé sur la SVM.

La première partie est décomposée en trois chapitres. Le premier fait un rappel sur le caches dans toutes leurs diversités. Il aborde les différents types de cache qui existent, les optimisations que l'on peut leur apporter et discute de leur utilisation dans les multi-processeurs à mémoire partagée. Le second chapitre présente les différentes manières d'aborder le partage des données dans un système de SVM. Il expose aussi diverses techniques utilisées dans les caches, mais plutôt pour un multi-processeur. Le troisième chapitre est lui entièrement consacré à la cohérence des données. Il expose les voies qui permettent de présenter des cohérences de plus en plus faibles.

La seconde partie décrit la réalisation d'une émulateur pour SVM. Il expose brièvement dans le premier chapitre les différentes sortes de simulateurs possibles et présente le choix retenu. Le second chapitre est une présentation en détail de l'émulateur. Cette présentation est décomposée en une exposition des principes généraux, puis nous abordons point par

point les différentes parties de l'émulateur. Le troisième chapitre est une présentation succincte des premiers résultats qui n'ont pas d'autre ambition que de valider l'émulateur construit.

Première partie

Les hiérarchies mémoires et la cohérence des données.

Le constant accroissement ces dernières années de la différence de vitesse entre le processeur et la mémoire principale que celui-ci adresse a créé un déséquilibre de puissance de traitement entre les divers composants d'une architecture interne. Ce déséquilibre implique que le besoin en données du processeur vis-à-vis de la mémoire centrale s'est considérablement accru. En effet, celui-ci a changé d'ordre de grandeur conséquemment à l'accroissement de la fréquence de fonctionnement des unités centrales d'une dizaine de mégahertz à plusieurs centaines de mégahertz. Par contre, dans le même temps, il n'y avait guère de progrès quant à la vitesse de service des mémoires dynamiques (mémoires qui constituent l'essentiel de la mémoire principale des ordinateurs).

La solution adoptée pour pallier cet écart de vitesse est la constitution d'une hiérarchie de mémoires. Cette hiérarchie est composée d'au moins deux niveaux de mémoires, à l'intérieur de laquelle, le niveau supérieur (celui qui est le plus proche du processeur) est plus petit, plus rapide, mais aussi plus cher (à taille égale) que le niveau inférieur.

Le but utopique d'une hiérarchie de mémoires est de fournir au processeur une mémoire dont la taille serait celle du niveau le plus inférieur, et dont la rapidité serait celle du niveau le plus supérieur. Or, les coûts des composants mémoires servant à réaliser ces mémoires sont prohibitifs dans le cas de mémoires de grandes tailles. A l'inverse, les composants mémoires servant à réaliser les mémoires de grandes tailles sont trop lents par rapport aux microprocesseurs actuels. Donc, il semble exclu de réaliser une mémoire à la fois rapide et de grande taille.

Il est cependant possible d'exploiter le principe de localité des programmes qui stipule que la prochaine référence mémoire n'est pas indépendante de celles effectuées précédemment. En effet, l'accès à un mot mémoire implique, que dans le futur, celui-ci ait de fortes chances d'être de nouveau référencé (localité temporelle) ainsi que les mots proches de lui (localité spatiale). C'est grâce à ce principe qu'une hiérarchie mémoire peut donner l'illusion d'avoir une quantité de mémoire importante et rapide, si un grand nombre d'accès aux données sont résolus dans les niveaux rapides proches du processeur. Donc, une hiérarchie mémoire est d'autant plus performantes que la résolution des accès mémoires s'effectue dans un niveau proche de l'unité de traitement, ce qui permet de tirer profit du principe de localité. Le but de la recherche dans le domaine des hiérarchies mémoires et de leurs implications dans l'architecture des machines est de minimiser les temps d'attente du processeur lors de ses accès mémoires.

Actuellement, les hiérarchies mémoires s'étendent souvent sur quatre niveaux (voir figure 0.2). Les trois premiers niveaux correspondent à trois technologies différentes de réalisation de la mémoire. Le premier niveau est réalisé sur la même puce que le processeur. Il est par conséquent de taille très limitée, de quelques Ko à quelques dizaines de Ko. C'est

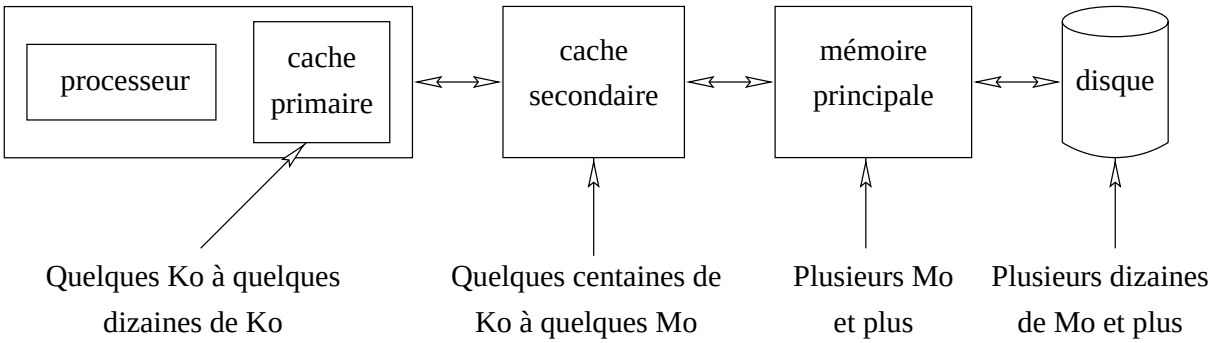


FIG. 0.2 – *Architecture générale d'une hiérarchie mémoire à trois niveaux*

le cache primaire. Le cache secondaire a une taille de quelques centaines de Ko à quelques Mo. Les composants pour le réaliser sont des SRAM. Ils sont plus rapides que la mémoire classique, mais, ils sont aussi moins intégrables donc plus chers. Le troisième niveau est réalisé avec des composants classiques de DRAM. Ceux-ci permettent de concevoir des mémoires de faible coût, mais dont les performances sont moins bonnes. Le dernier niveau est le plus lent de tous, les données sont en effet stockées temporairement sur disque. Il faut alors les rapatrier en mémoire centrale avant de pouvoir les utiliser.

L'objectif du premier chapitre est de rappeler les principes des hiérarchies mémoires. Il va essentiellement détailler le fonctionnement des caches, pour en rappeler les principes et les variantes. En effet, l'intégration de ceux-ci dans des machines dotées de plusieurs processeurs a été le départ du développement des mécanismes de gestion de cohérence. Or, la gestion des caches d'un système multi-processeurs a beaucoup d'analogies avec la gestion des pages d'une SVM. C'est pourquoi, pour chaque type de technique de gestion de cache introduit, nous essaierons de montrer son adaptabilité ou sa non-adaptabilité au contexte de SVM.

Le second chapitre présente les différentes méthodes d'accès aux pages dans une SVM: accès distant, migration et duplication. C'est justement dans ce dernier cas que se pose le problème de gestion de la cohérence des données. Il expose également des techniques d'optimisations ou des voies de recherche pour améliorer le partage des données dans un contexte multi-processeurs.

Enfin, le troisième chapitre expose différentes manières de gérer la cohérence. Ces méthodes diverses ont été construites, afin d'améliorer le parallélisme des accès aux données, de diminuer la charge du réseau d'interconnexion et donc finalement, d'augmenter l'efficacité.

Chapitre 1

Rappels sur les caches et la gestion de leur cohérence.

Le grand problème à résoudre pour les hiérarchies mémoires, c'est de positionner les données le plus près possible du processeur, afin de pouvoir les accéder le plus rapidement possible. Or, plus l'on se rapproche du processeur, plus la place se fait rare. Pour améliorer ceci, un grand éventail de techniques existent. Dans la suite du chapitre, ces techniques sont exposées d'abord dans un contexte mono-processeur, puis dans un contexte multi-processeur. Pour chacune d'elles, nous étudierons leur adéquation avec la SVM.

Avant de présenter les différentes sortes de hiérarchies mémoires limitées aux caches, ainsi que les divers chemins d'investigations pour en améliorer les performances, il faut définir quelques termes.

Les différents niveaux d'une hiérarchie mémoire sont divisés en blocs, aussi appelés lignes. Ces blocs sont indivisibles. C'est à dire que si un mot appartenant à un bloc est présent dans le cache, tous les autres mots le sont aussi. Inversement, si un mot d'un bloc n'est pas présent dans le cache, tous les autres mots de ce bloc ne le sont pas non plus.

La taille des données échangées entre deux niveaux de la hiérarchie est appelée l'unité d'échange. Cette unité d'échange correspond le plus généralement à la taille du bloc du niveau supérieur (le plus proche du processeur).

Lorsqu'un accès à la mémoire n'est pas résolu pour un niveau, il se produit une faute de cache qui va initier le processus de recouvrement de la donnée au niveau inférieur. Il existe deux sortes de fautes de cache, la faute en lecture et la faute en écriture, ce qui entraîne des traitements différents (pour de plus amples renseignements sur le fonctionnement interne des caches voir [Smi87]).

1.1 Rappels sur les caches.

Les caches ne sont pas tous organisés de la même manière. Des différences existent dans la façon de placer les blocs [Sta88], dans la différenciation ou non des instructions et des données et dans quelques organisations atypiques.

1.1.1 Les diverses possibilités de placement d'un bloc.

Cette partie rappelle les trois principales possibilités de placement de blocs dans les caches. Il en existe d'autres qui sont présentées dans les organisations atypiques des caches.

1.1.1.1 La correspondance directe.

Cette méthode de placement est la plus restrictive. Pour un bloc donné, il n'existe qu'un seul emplacement dans le cache possible (figure1.1). Donc, si cet emplacement est déjà occupé, il faut remplacer l'ancien bloc par le nouveau. Cette méthode à l'avantage d'être très simple et donc la décision d'éviction d'un bloc est vite prise, elle est aussi celle qui prend le moins de place. Cependant, le taux de fautes de caches (rapport entre le nombre de fautes de caches et le nombre total d'accès à la mémoire) n'est pas avantagé par cette façon de faire.

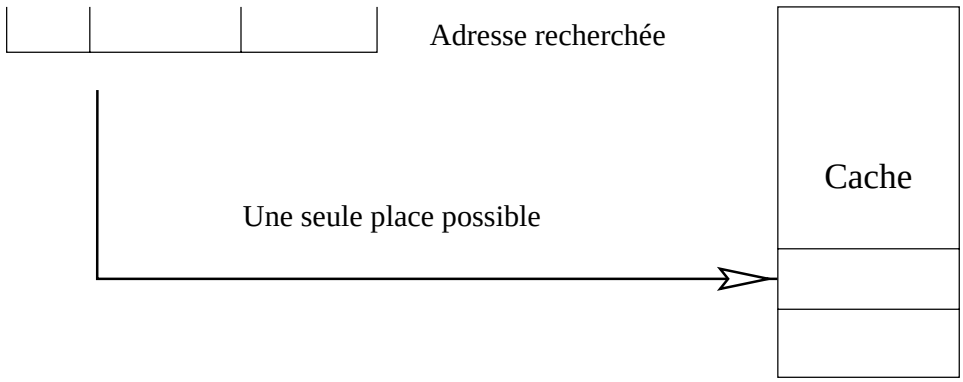


FIG. 1.1 – *Cache à correspondance directe*

1.1.1.2 L'associativité totale.

Cette organisation des placements des blocs est totalement l'inverse de la précédente (figure1.2). Un bloc peut être placé à n'importe quel endroit du cache. Cette méthode permet de choisir parmi plusieurs politiques de remplacement de blocs, afin d'utiliser

au mieux la place disponible dans le cache. Les politiques les plus connues sont le LRU (Last Recently Use) qui évince le bloc le moins récemment utilisé, le FIFO (First In First Out) qui déplace le bloc le plus vieux du cache et le RANDOM qui choisit le bloc à rapatrier au niveau inférieur de manière aléatoire. Les politiques qui permettent une meilleure utilisation de l'espace fourni par le cache en comparaison avec la correspondance directe sont le LRU ou celles qui s'en rapprochent le plus. En effet, le LRU est très coûteux à réaliser, c'est pourquoi, on lui préfère souvent des politiques plus simples à mettre en œuvre, mais qui ne réalisent pas tout à fait le LRU. Les caches à associativité totale, pour peu qu'ils appliquent une bonne politique d'éviction des blocs, ont des taux d'échecs d'accès au cache inférieurs à un cache à correspondance directe de même taille. Cependant, l'utilisation d'une mémoire associative implique qu'un cache à associativité totale est beaucoup plus volumineux.

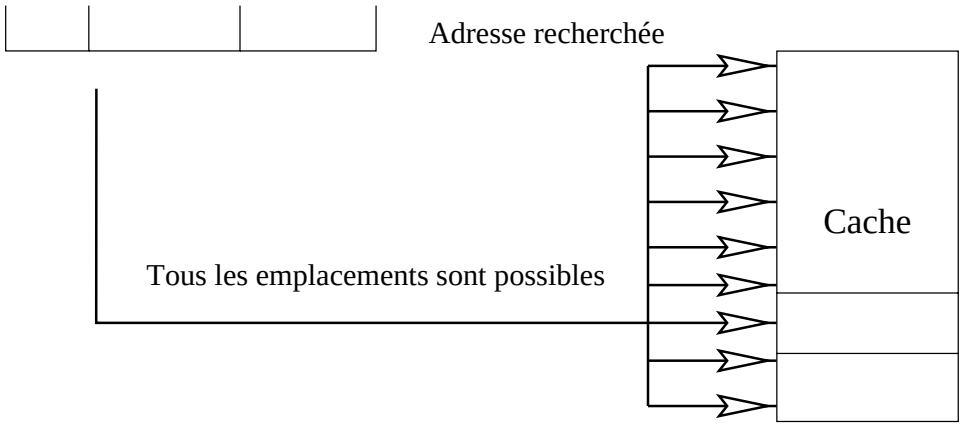


FIG. 1.2 – *Cache dont l'associativité est totale*

1.1.1.3 L'associativité par ensemble.

Ce type de cache est le fruit du mélange des deux méthodes précédentes. Un cache qui est 2-associatif peut être vu comme une juxtaposition de deux caches à correspondance directe, le choix doit se faire entre ces deux caches (figure1.3).

Une autre manière de voir est de considérer un cache 2-associatif comme la juxtaposition de caches à associativité totale dont la taille serait de deux emplacements (figure1.4).

Dans les deux cas, pour passer de l'adresse recherchée à l'adresse dans le cache, on utilise donc une fonction de hash-code (réalisée le plus souvent par troncature de l'adresse) avec traitement des collisions sur un ensemble de blocs de caches de cardinalité 2, 4, ... D'un point de vue comme de l'autre, le résultat est le même et cherche un compromis

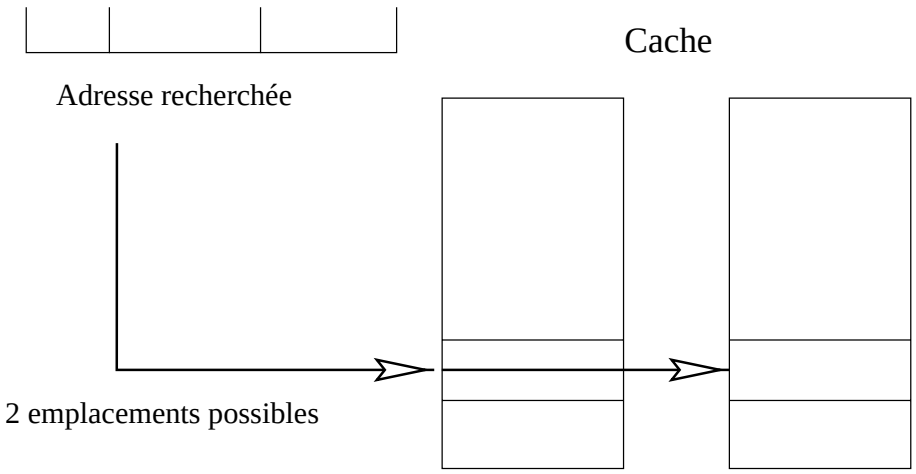


FIG. 1.3 – *Première vision d'un cache 2-associatif.*

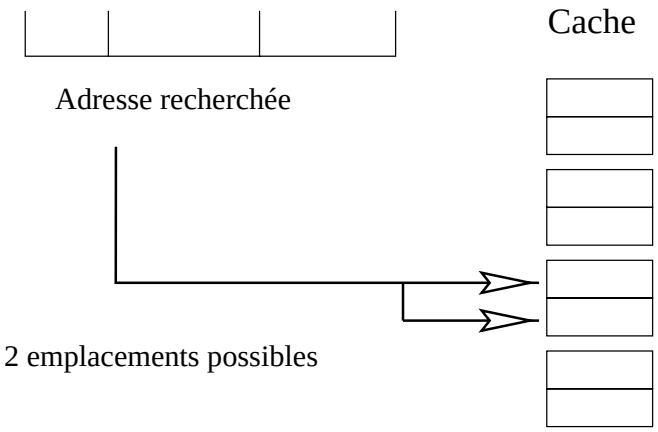


FIG. 1.4 – *Autre vision d'un cache 2-associatif.*

entre la rapidité d'accès en cas de présence du mot voulu et le taux de fautes de caches. Le degré d'associativité peut varier (2, 4, 8, ...), mais en réalité, le degré d'associativité choisi est de 2 ou de 4. Aller au-delà, augmente la complexité de la recherche des mots, sans pour autant accroître de façon significative l'utilisation de l'espace fourni par le cache [HP92].

1.1.1.4 L'associativité et la SVM.

La mémoire virtuelle utilise l'associativité totale pour le placement des pages en mémoire physique. En effet, une page virtuelle peut se retrouver à n'importe quelle adresse dans l'espace dont l'accès lui est autorisé. L'utilisation de cette méthode, est possible à ce niveau car le besoin de rapidité d'accès à l'information, est beaucoup moins critique

que pour les caches. Comme la SVM se base sur la mémoire virtuelle, elle utilisera donc l'associativité totale.

1.1.2 La différenciation entre cache d'instructions et cache de données.

Comme l'efficacité de la notion de hiérarchie mémoire est liée au principe de localité, il semble intéressant de différencier l'accès aux données et l'accès aux instructions [Wyl87]. Ces flux de données sont indispensables à l'exécution des programmes, mais leur traitement et leur nature sont bien distincts. Le flux des instructions est intéressant à étudier, car il n'y a que des accès en lecture, et l'ordre dans lequel les instructions sont adressées est plus régulier que les données classiques. De ce fait, il semble intéressant de séparer les instructions des données en raison de leurs natures différentes, mais aussi pour donner la possibilité d'exploiter les propriétés particulières du flot des instructions.

En ce qui concerne la SVM, Cette différenciation est déjà réalisée dans les processus eux-mêmes. Une zone de mémoire est réservée aux instructions. C'est pourquoi, ici l'adaptation de cette technique n'est pas adéquate.

1.1.3 Les caches atypiques.

Sous cette dénomination de caches atypiques, sont recensées les organisations de caches qui sortent de l'ordinaire.

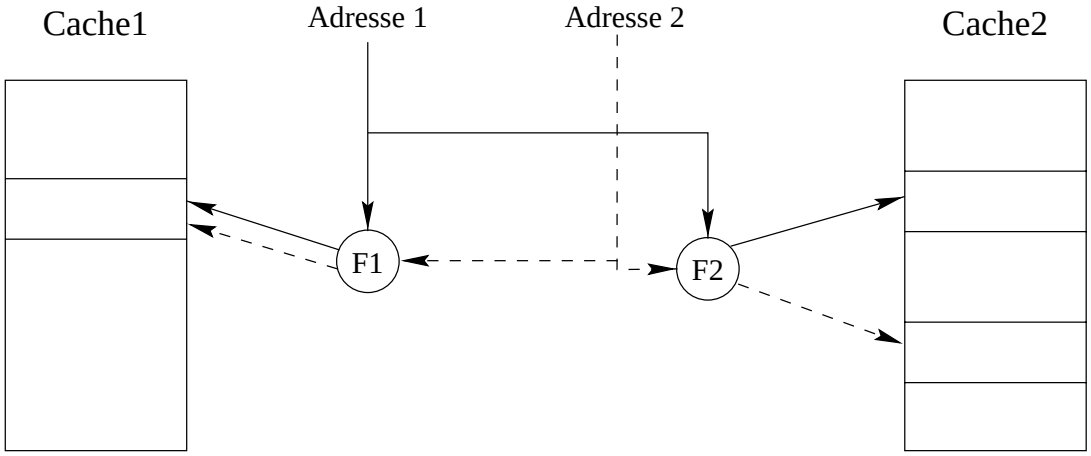


FIG. 1.5 – *Exemple d'un cache «skewed-associatif».*

1.1.3.1 Les caches associatifs brouillés («skewed-associatif»).

Ces caches partent du principe que dans un cache associatif par ensemble, si un bloc rentre en concurrence avec un autre bloc, sur un ensemble, il en sera de même pour tous les autres ensembles [SB93]. L'idée est donc de ne pas avoir les mêmes fonctions de répartitions des blocs dans les différents ensembles qui forment le cache. De cette façon, s'il existe une concurrence entre deux blocs sur un ensemble, il devient possible que ces deux blocs ne soient plus en concurrence pour les autres ensembles. C'est ce que montre la figure 1.5, où deux adresses différentes (Adresse 1 et Adresse2) passent par deux fonctions de hashage différentes (F1 et F2). Le résultat est que pour l'ensemble (cache 1), les adresses sont en concurrences, mais pour l'autre ensemble (Cache 2) elles ne le sont pas. Ceci doit permettre d'utiliser au mieux tout le cache.

1.1.3.2 Les caches semi-unifiés.

Les caches semi-unifiés [DS93] ont pour objet de concilier un temps d'accès réussi au cache très court et un taux d'échecs bas. Pour cela, les caches d'instructions et de données se trouvant à même la puce sont utilisés quand cela est bénéfique comme cache de second niveau l'un vis-à-vis de l'autre. Le cache de données pourra donc servir de cache de second niveau au cache d'instructions et inversement, celui-ci pourra être utilisé de la même façon pour le cache de données (voir la figure1.6).

Du point de vue de l'extérieur, tout se passe comme si le cache sur la puce était un cache 2-associatif unifié. Il a donc un taux de référence extérieure à la puce se rapprochant du taux d'un cache 2-associatif, en gardant une complexité interne et un temps d'accès réussi proche de deux caches à correspondance directe distincts. De plus, cette organisation, garde la possibilité de pouvoir accéder aux deux caches de manière indépendante. Elle permet aussi de moduler la taille des caches d'instructions et de données, de manière totalement automatique et simple.

1.1.3.3 Les caches atypiques et la SVM.

L'organisation de ces caches est très spécifique et repose sur la présence de caches dissociés entre instructions et données pour l'un et sur le principe d'associativité par ensemble pour l'autre. Or, nous avons déjà vu en 1.1.1.4 et 1.1.2 que ces notions n'étaient pas transposables aux niveaux de la SVM. Les caches atypiques ne le sont donc pas non plus.

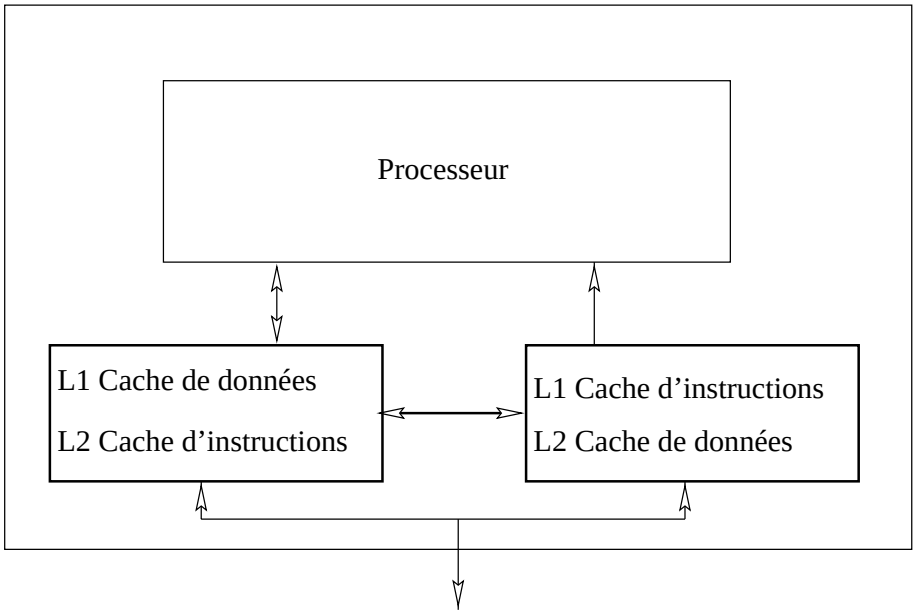


FIG. 1.6 – *Principe des caches semi-unifiés.*

1.2 Les différentes manières de gérer les écritures.

1.2.1 Cohérence cache-mémoire: écriture transparente, écriture différée ou paresseuse.

Lorsqu'une écriture survient, trois stratégies de propagation de cette écriture vers la mémoire centrale sont possibles. Soit l'écriture est immédiatement répercutée sur la mémoire centrale, c'est le cas de l'écriture transparente, soit elle est gardée dans le cache et ne sera réécrite que lors de l'éviction du bloc correspondant du cache, pour l'écriture différée [Wyl87], soit elle est gardée dans le cache et n'est réécrite qu'en utilisant des instants inutilisés du médium de communication, c'est l'écriture paresseuse.

La première solution a l'avantage d'être simple, la valeur de référence est toujours en mémoire et à jour. Ceci a un intérêt éventuel en terme de fiabilité, mais génère un trafic important entre le cache et la mémoire (figure 1.7). La seconde solution est plus rapide car elle n'entraîne que peu de trafic avec la mémoire. Il semble qu'elle soit la solution du futur car l'accent est de plus en plus mis sur la rapidité. La troisième solution cumule les avantages de l'une et de l'autre des solutions, mais il faut être capable de détecter les moments où le réseau d'interconnexion est au repos.

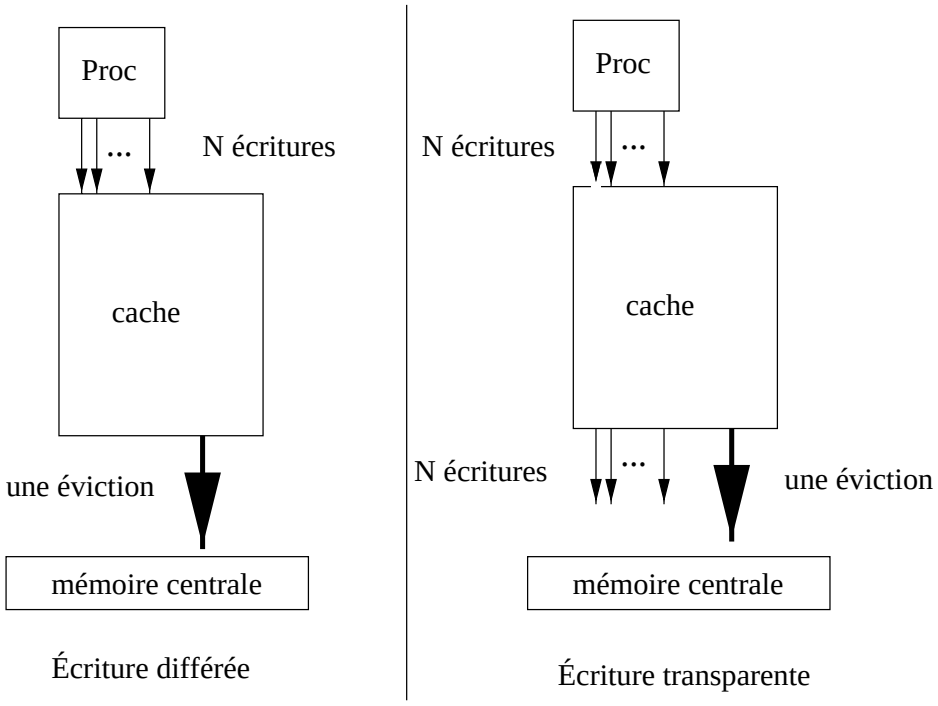


FIG. 1.7 – *Suivi de N écritures et d'une éviction pour des écritures différées et des écritures transparentes.*

1.2.2 La mise en file d'attente des requêtes.

Lors du fonctionnement d'un programme, deux sortes de requêtes sont adressées au cache, les lectures et les écritures. Les premières sont les plus pénalisantes en cas de fautes de caches. Il est en effet impossible de passer outre et il faut attendre que le niveau inférieur de la hiérarchie mémoire fournisse la donnée pour que le programme puisse continuer. A l'inverse, les écritures ne sont pas bloquantes pour le processeur. Il suffit de stocker la requête et de la réaliser, pendant que le processeur continue son exécution. Dans le cas d'une politique d'écritures différées, si le bloc est dans le cache, la requête est simplement écrite dans le cache et la mémoire principale sera mise à jour lors de l'éviction du bloc correspondant. Par contre, pour une politique d'écritures transparentes, les requêtes sont mises dans une file d'attente ce qui implique tout un mécanisme de gestion de celle-ci. En effet, il faut gérer entre autres, le cas de plusieurs écritures sur le même bloc qui peuvent être regroupées en une seule écriture, et le cas d'une lecture qui s'effectue avant qu'une écriture précédente et présente dans la file d'attente ne soit faite. Dans ce cas, il faut relire les données depuis la file d'attente, mais sans arrêter le déroulement des écritures. Le fait de pouvoir mettre les données dans une file d'attente permet donc au processeur, dans

le cas de plusieurs écritures, de continuer son exécution sans attendre la fin effective des écritures précédemment lancées.

1.2.3 La gestion des écritures et la SVM.

L'idée de regrouper les écritures, afin de mettre à jour le niveau inférieur, permet de diminuer le trafic entre les deux niveaux concernés de la hiérarchie mémoire. Cette idée a été reprise pour améliorer la gestion de la cohérence des données (voir 3.1.2) car de la même façon, elle diminue le trafic nécessaire au maintien de la cohérence des données.

1.3 Améliorations de la rapidité d'accès aux données.

1.3.1 Le «multi-threading».

Une manière originale de contourner le problème de la latence mémoire est de prévoir un mécanisme qui permette de maintenir plusieurs flots d'exécutions prêts à l'intérieur d'un processeur et de pouvoir passer de l'un à l'autre très rapidement lorsque surgit une faute de cache [BR92] [NPA92]. On peut faire l'analogie avec la commutation des processus lorsque ceux-ci font une entrée-sortie. L'idée est identique, mais l'échelle change ainsi que l'ampleur du changement de contexte à effectuer. Cette méthode est avantageuse si le temps pris pour basculer d'un flot d'exécutions à un autre est bien inférieur au temps de latence de l'accès mémoire et si le nombre de flots d'exécution est suffisamment important pour faire en sorte qu'ils ne se retrouvent pas tous en attente d'un accès mémoire.

Bien entendu, comme cette méthode est inspirée de la commutation de processus, il est naturel de pouvoir la retrouver lors de la réalisation d'un système de SVM. Donc, dans un tel système, si un processus accède une page qu'il ne peut pas avoir immédiatement, soit par dépassement de la capacité de la mémoire, soit pour des raisons de maintien de la cohérence, il se suspendra afin de laisser la place à un processus qui puisse s'exécuter.

1.3.2 L'anticipation des chargements ou préchargement.

L'anticipation des chargements [Smi87] a pour objet de diminuer le taux d'échecs du cache. En fonctionnement normal, le chargement d'un nouveau bloc ne se déroule qu'au moment où l'absence de celui-ci est constatée. L'anticipation des chargements essaie de prévoir les besoins futurs. Une manière simple de le faire est de raisonner de manière séquentielle et donc de charger à l'avance les données qui suivent celles utilisées. C'est la technique du bloc d'avance (« one block lookahead »). De cette manière, si les données

chargées sont utilisées, le cache aura fait une économie d'une faute de cache. Par contre, le revers de la médaille, est que les données chargées par anticipation prennent la place d'autres données. Cette méthode est donc rentable seulement si le gain obtenu par l'anticipation des chargements compense les fautes de caches qui ne se seraient pas produites autrement.

La reprise de cette technique au niveau de la SVM, ne semble pas pouvoir être une réalité. En effet, la SVM manipule des pages et la taille de celles-ci est beaucoup plus grande que celle des blocs de caches. De cette façon, si l'on charge par anticipation une page, c'est un grand nombre de données qui seront chargées, ce qui aura une influence sur la charge du réseau. De plus, il y aura de plus fortes chances de charger des données inutiles que lors du chargement par anticipation d'un bloc de cache. La charge supplémentaire demandée au réseau, aura peu de chance d'être compensée par un chargement anticipé valable. Par contre, cela n'empêche nullement le préchargement des pages par des systèmes essayant d'évaluer le «working set»; mais cela se déroule surtout en cas de redémarrage d'un processus.

1.3.3 L'adressage du cache par des adresses virtuelles.

Pour un cache de conception classique, la recherche dans le cache s'effectue au moyen d'une adresse réelle. Celle-ci a été obtenue à partir de l'adresse virtuelle fournie par le processeur. Cette translation prend du temps et ce temps est perdu pour tous les types d'accès (réussi ou non). Il semble donc intéressant de supprimer ce temps perdu en adressant directement le cache par les adresses virtuelles [Sta88]. Cependant, il peut apparaître en procédant de la sorte, des problèmes de synonymie entre deux adresses virtuelles qui représentent la même adresse réelle [IKWS92]. C'est le cas, par exemple, d'un programme qui partage des données avec le système ou d'un programme qui utilise deux parties de son espace virtuel pour adresser la même zone de mémoire réelle. Ce problème de synonymie peut être résolu en rajoutant une partie matérielle qui fait la correspondance entre les adresses réelles et les adresses virtuelles. Cet ajout de matériel augmente la surface, et n'est donc justifié que si le gain de temps est appréciable.

En ce qui concerne la SVM, comme elle est basée sur le mécanisme de mémoire virtuelle, toutes les adresses utilisées, sont virtuelles, il n'y a donc aucune amélioration à attendre à ce niveau là. Bien entendu, si l'amélioration est utilisée au niveau du cache, elle est transparente pour la gestion de la mémoire virtuelle et donc pour la SVM.

1.3.4 Les méthodes logicielles en amont de l'exécution.

Comme nous l'avons vu précédemment, le fait qu'une hiérarchie mémoire soit performante, provient du principe de localité. Ce principe a été découvert alors que les programmes qui s'exécutaient sur les machines n'avaient pas été conçus dans le but de l'exploiter. Lors de la conception des programmes ou lors de leur compilation, rien n'était fait pour améliorer la localité des données. Comme toutes les machines modernes sont équipées d'une hiérarchie mémoire, il est évident que l'on peut dès la conception des compilateurs, ou dans une moindre mesure lors de la conception des programmes, améliorer le principe de localité, en répartissant les données de la meilleure façon possible.

La recherche du meilleur emplacement des données n'est pas un problème spécifique aux hiérarchies mémoires d'un mono-processeur. Il est omniprésent dans un environnement multi-processeur. C'est pour cela que cette recherche du meilleur emplacement pour les données peut très bien se transposer dans un système de SVM.

1.4 Les caches dans les systèmes multi-processeurs à mémoire partagée.

Dans cette partie, nous présentons les systèmes multi-processeurs à mémoire partagée. L'idée de départ a été de dupliquer le couple processeur-cache d'une architecture classique, afin d'augmenter la puissance de calcul. Il en a donc résulté des machines du type présenté par la figure 1.8. Cependant, avec la multiplication des caches, un problème de maintien de la cohérence des données est apparu.

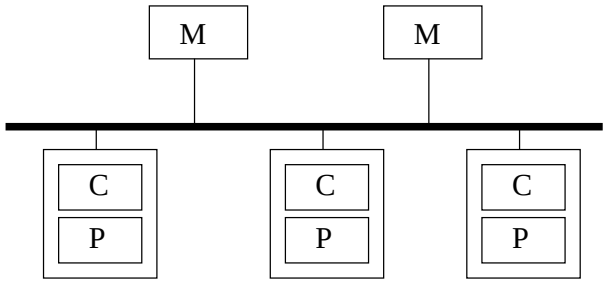


FIG. 1.8 – *Multi-processeur à bus commun. C: cache, P: Processeur, M: Mémoire*

1.4.1 Le problème du maintien de la cohérence dans les caches.

Ce problème apparaît lorsqu'un processeur accède une donnée en écriture: il faut alors disposer d'un moyen pour permettre aux autres processeurs d'accéder la valeur correcte de chaque donnée. Un exemple est donné dans la figure 1.9. Initialement, les processeurs A et B ainsi que la mémoire centrale avait pour la variable X la valeur 100. Après, la modification de celle-ci par A, la valeur pour X est de 200 pour A et la mémoire et de 100 pour B. Dans cette situation, si le processeur B fait un accès en lecture à la variable X, il va recevoir l'ancienne valeur de X, c'est à dire 100; on pourra dire alors que la cohérence n'est plus assurée. (Toutefois il est possible que B puisse se contenter de l'ancienne valeur, on dira alors que la cohérence n'est pas stricte et on parlera d'autres types de cohérence: voir le chapitre 3.)

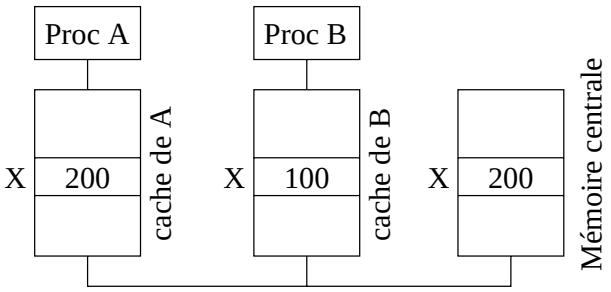


FIG. 1.9 – *Exemple d'un conflit de cohérence.*

1.4.2 Les solutions en cas de bus unique.

Dans le cas d'une architecture à bus unique, il est très facile de résoudre le problème en utilisant la propriété naturelle de diffusion du bus. Celui-ci est en effet un passage obligé de tous les transferts d'un cache vers la mémoire et devient donc un endroit privilégié à observer si l'on veut avoir un point de vue global sur l'évolution du contenu des caches. En espionnant le bus, il est alors facile de se rendre compte d'une modification d'une variable que l'on détient. Deux solutions sont envisageables, soit l'on marque la donnée comme invalide, il faudra donc la recharger lors d'un accès ultérieur, soit l'on met directement la variable à jour en récupérant à notre compte la valeur circulant sur le bus. Ainsi, en reprenant l'exemple de la figure 1.9, l'entrée pour la variable X se trouverais invalidée si l'on utilisait la première méthode ou bien, elle prendrait la valeur 200 issue du bus si l'on utilisait la seconde.

La solution proposée précédemment, admet des améliorations. Il est en effet possible de baisser le trafic sur le bus pour permettre un nombre plus grand de processeurs. Pour

se faire, il suffit de ne pas prolonger automatiquement les écritures jusqu'à la mémoire principale. Ces écritures sont faites uniquement dans le cache. Cette méthode requiert l'envoi d'invalidations à travers le bus pour prévenir les autres caches que leur copie de la donnée spécifiée n'est plus valable. La mise à jour de la mémoire principale, s'effectue lors d'un retour de la donnée du cache qui la détenait ou bien, lors d'une demande d'un autre cache pour cette donnée.

Une autre amélioration est possible en effectuant des transferts cache à cache. Lors d'une faute de cache de cohérence d'un processeur, la donnée était jusqu'a maintenant reçue de la mémoire principale. Or, cette mémoire est plus lente que la mémoire utilisée dans les caches. De ce fait, il semble judicieux de faire le transfert de la donnée directement du cache qui la détient au cache qui la réclame. Il est possible de mettre à jour au passage la mémoire centrale si l'opération n'en est pas ralentie, mais ce n'est pas indispensable, car elle sera de toutes les façons mise à jour lors de l'éviction de la donnée de tous les caches. Toutes ces solutions sont détaillées dans l'annexe A et présentées dans [Rai92] et [Ste90].

1.4.3 Les solutions par répertoires.

Ce qui fait la force d'un bus unique, c'est l'obligation d'avoir recours à lui pour mettre à jour les données. Cette obligation fournit un moyen simple de diffusion, mais à le grand inconvénient de devenir rapidement un goulet d'étranglement.

C'est pour éviter cela que des solutions au problème de la cohérence des caches ont été développés à l'aide de répertoires. Ceux-ci ont pour objectif de permettre l'envoi des invalidations ou la diffusion des écritures, selon le cas, uniquement aux destinataires concernés. En effet, pour casser le goulet d'étranglement que constitue le bus, il faut adopter un autre type de réseau (point à point, multi-étages, «cross-bar» ou autre ...). Or, sur ces réseaux la diffusion est une opération coûteuse, il n'est donc plus question de l'utiliser à chaque instant. C'est pourquoi, le recours à des répertoires qui indiquent pour chaque donnée les autres caches à prévenir en cas de modification de celle-ci permet de limiter ou de supprimer l'utilisation de la diffusion.

Il faut cependant faire attention aux données contenues par ces répertoires. En effet, il est très facile de construire des répertoires contenant toutes les informations nécessaires pour éviter les diffusions. Mais de tels répertoires contiennent alors une trop grande quantité de données qui devient beaucoup trop importante si le nombre de caches augmente grandement. Ainsi, le problème des répertoires est de gérer un nombre suffisant de données afin de beaucoup ou de complètement réduire le recours à la diffusion, tout en garantissant

une structure qui permette un accroissement du nombre de processeurs sans entraîner une augmentation disproportionnée des données stockées dans ces répertoires.

Pour cela, différentes solutions sont possibles. Il s'agit de compromis entre placer les informations de partage en mémoire centrale ou dans les caches, limiter la taille des répertoires et faire appel à des traitements spéciaux lors des dépassements de capacité (diffusion, traitement logiciel) ou bien adopter une structure qui permette de garder toutes les informations utiles. La recherche du compromis idéal à permis l'élaboration d'un grand nombre de solutions qui sont présentées en détail dans l'annexe A ainsi que dans [CFKA90].

La façon dont sont rangées les informations pour la gestion de la cohérence n'est guère critique en ce qui concerne la SVM. En effet, les traitements s'effectuent par logiciel ce qui implique que la taille des données manipulées n'ait donc pas beaucoup d'importance. Par contre, il faut limiter au maximum l'échange des pages sur le réseau d'interconnexion du fait de la taille de celle-ci. Une politique par invalidations semble donc mieux appropriée que la diffusion des écritures. Il semble aussi préférable d'éviter absolument les diffusions et donc de garder toutes les informations utiles pour l'éviter.

1.5 Résumé.

Pour résumer l'étude des différentes organisations de caches et leurs techniques d'exploitation en vue de leur implantation pour la SVM, il s'avère qu'un bon nombre de choix soit fixé par l'adoption de la mémoire virtuelle comme base de départ de la SVM. Ainsi, la SVM utilise une associativité totale, distingue, à l'intérieur des processus, les instructions et les données et utilise les possibilités de changement de contexte du système ainsi que les adresses virtuelles. Par contre, du fait de l'associativité totale, il n'est pas possible de fournir des techniques basées sur l'associativité par ensemble (comme le brouillage); de même, il ne semble pas souhaitable de faire de l'anticipation de chargement en se basant sur la localité spatiale. En revanche, le regroupement des écritures est la base de toute une manière de gérer la cohérence des données qui est exposée en 3.1.2. Enfin, la façon de placer les données prend une toute autre ampleur dans le contexte multi-processeur, car il ne s'agit plus seulement de bien positionner les données dans la hiérarchie mémoire, mais il faut tenir compte du besoin de partage des données par les applications parallèles. C'est en grande partie le sujet du chapitre suivant.

Par la suite, nous ne parlerons plus de toutes ces notions introduites pour gérer les caches dans un mono-processeur ou dans un multi-processeur. Nous nous intéresserons uniquement au maintien de cohérence pour une SVM, même si, les mécanismes sous-jacents à ce maintien de la cohérence sont inspirés de la cohérence des caches.

Chapitre 2

Hiérarchies mémoires et la SVM.

Cette partie s'intéresse principalement aux changements dans l'organisation d'une hiérarchie mémoire qu'apporte le passage à une architecture à plusieurs processeurs et surtout à plusieurs mémoires. Elle n'aborde toutefois pas les difficultés qui sont introduites par le maintien de la cohérence des données qui est exposé dans le chapitre suivant. Le changement le plus fondamental, provient de l'importance accrue du placement des données. C'est pourquoi ce sujet est abordé en priorité. La fin de cette partie présente diverses techniques ou problème des multi-processeurs, pas forcément liés aux hiérarchies mémoire.

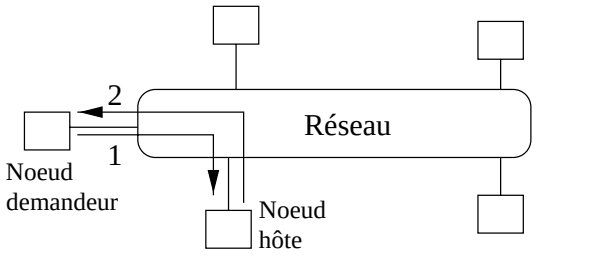
2.1 Les différentes méthodes d'accès aux données.

La manière dont le partage des données s'effectue influe directement sur les performances du système. Le but dans ce domaine pour un système multi-processeur est le même que pour un seul processeur, la donnée que l'on veut traiter doit être la plus proche possible de l'endroit où elle va être traitée. Il faut donc répartir les données le mieux possible; mais ceci étant réalisé de manière imparfaite (car certaines données peuvent être utilisées par plusieurs processeurs) il faudra toujours déplacer des données. Le réseau d'interconnexion devient souvent dans ces cas là un goulet d'étranglement. Il faut donc réduire la quantité de données véhiculées par le réseau. Pour cela, on peut agir sur la granularité des échanges de données et sur la périodicité de ces échanges. En ce qui concerne la granularité des accès aux données, elle peut être très diverse selon les choix retenus. Cela peut aller du mot si la solution retenue est une mise à jour des données au fur et à mesure des modifications, jusqu'à la page si le système multi-processeur utilisé est plutôt un système multi-ordinateur. La granularité est donc surtout fonction du niveau auquel se fait le partage (cache, mémoire physique ou mémoire virtuelle). L'accès aux données

dans un multi-processeur peut être effectuée de différentes manières, selon la possibilité de faire migrer les données ou non et selon la possibilité de dupliquer les données ou non. Ces méthodes sont détaillées ci-après ainsi que dans [SZ90].

2.1.1 L'accès à distance.

La première méthode d'accès aux données dans un multi-processeur, est la plus proche de ce qu'étaient les requêtes mémoires dans un mono-processeur. En effet, elle ne fait intervenir aucune recopie de données (voir la figure 2.1).



Trafic réseau pour chaque lecture:
1: demande du mot voulu
2: recopie de la valeur du mot demandé dans un registre

Trafic réseau pour chaque écriture:
1: valeur et emplacement du mot à écrire
2: éventuellement acquittement

FIG. 2.1 – *L'accès à distance ne recopie aucune donnée en dehors de la hiérarchie mémoire du site hôte. Donc à chaque accès, il y a un échange avec le noeud hôte.*

Lors de la résolution d'une requête en lecture, s'il se trouve que la donnée demandée n'est pas présente dans la hiérarchie mémoire du processeur demandeur, la requête est envoyée sur le réseau d'interconnexion des processeurs. Celle-ci est alors traitée par le noeud qui contient la donnée demandée et cette dernière est renvoyée en réponse. La dernière phase de cette demande est la réception par le processeur demandeur de la donnée et ainsi, la poursuite de l'exécution peut avoir lieu.

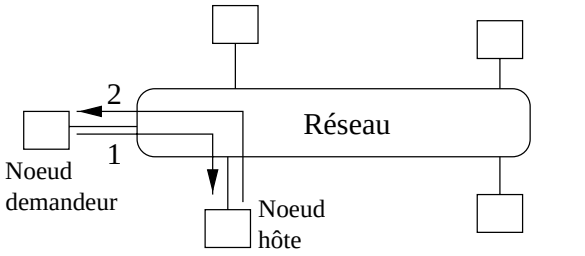
Dans le cas d'une écriture, les mêmes mécanismes se mettent en route, les seules différences sont, d'une part que le processeur n'est pas irrémédiablement bloqué comme il l'est lors d'une lecture, et d'autre part, que l'envoi de la donnée s'effectue à l'aller de la requête et que le retour peut ne pas exister, s'il n'y a pas d'acquittement.

Cette méthode instaure une différence de durée de traitement des requêtes, selon les positions respectives de la donnée et du processeur qui l'accède. C'est ce qui crée le groupe

d'architectures NUMA (Non Uniform Access Memory) où l'accès à une donnée ne se fait pas toujours avec le même temps selon la localisation de celle-ci. Cette façon de faire n'est intéressante que si les accès mémoire distants sont rares. En effet, l'accès au réseau ralentit le processeur et trop y recourir nuirait aux performances.

2.1.2 La migration des données.

Pour éviter ces accès trop répétés au réseau, la migration des données permet à un processeur de faire déplacer une donnée d'un emplacement lointain, vers sa mémoire locale (voir figure 2.2).



Trafic réseau pour le premier accès au bloc voulu :
1: demande du bloc voulu
2: recopie du bloc demandé dans une mémoire tampon

Pour les autres accès, il n'y a pas de trafic réseau
si le bloc n'a pas été demandé par ailleurs.

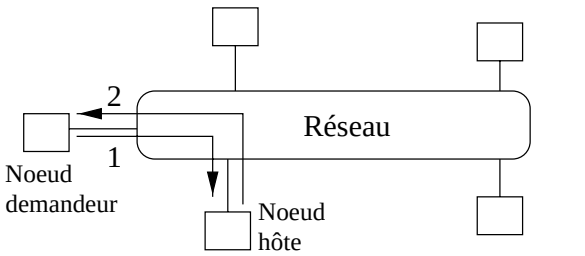
FIG. 2.2 – *Exemple d'un accès mémoire qui provoque une migration.*

Lors d'une lecture ou d'une écriture, si la donnée demandée ne se trouve pas dans la mémoire locale, la requête est, comme dans le cas précédent, dirigée vers le réseau. Le noeud distant qui possède la donnée ne fait pas que la transmettre, mais il effectue la migration de la donnée du noeud distant au noeud demandeur. De cette manière, la donnée se trouve copiée dans la mémoire du processeur demandeur et ne se trouve plus dans la hiérarchie mémoire du processeur distant. Ainsi, dans l'avenir, si un autre accès à la même donnée survient, il pourra se faire localement. En comparaison, avec la méthode précédente, les performances sont meilleures si le partage des données entre les processeurs devient important. Par contre, un effet indésirable apparaît, l'effet ping-pong entre deux processeurs. Il concerne une donnée partagée entre deux processeurs ou plus et qui est sans cesse renvoyée de l'un à l'autre. Cet effet contribue donc à la hausse de l'utilisation du réseau d'interconnexion et peut donc être dommageable pour les performances. Un autre problème voisin du précédent, et avec des conséquences identiques est celui du faux

partage [BS93]. Il apparaît quand deux processeurs ou plus se partagent le même bloc de données, mais pas les mêmes données. Ici, il n'y a pas concurrence sur les données, mais sur les blocs. Il existe alors une solution pour éviter le ping-pong induit par le faux partage, il faut redistribuer les données disputées sur des blocs différents.

2.1.3 La réplication des données.

Pour éviter un certain nombre d'effets ping-pong (ceux se cantonnant aux lectures), il est possible de dupliquer les données. De cette manière, quand un processeur veut travailler sur une donnée, il en charge une copie dans sa mémoire locale et peut l'utiliser comme il l'entend (voir figure 2.3).



Trafic réseau pour le premier accès au bloc voulu :
1: demande du bloc voulu
2: recopie du bloc demandé dans une mémoire tampon

Pour les autres accès, il n'y a pas de trafic réseau
à l'exception du maintien de la cohérence.

FIG. 2.3 – *Exemple d'un accès mémoire qui provoque une réplication.*

Cette façon de faire est idéale lorsqu'il ne s'agit que de manipuler des données en lecture. Si l'on veut manipuler des données en écriture, des problèmes de cohérence vont apparaître ainsi qu'un retour de l'effet ping-pong si plusieurs processeurs écrivent régulièrement sur la même donnée. Ces problèmes de cohérence de la mémoire sont présentés dans le chapitre suivant.

2.1.4 La combinaison de plusieurs méthodes.

Les trois méthodes exposées précédemment, ont toutes leurs avantages et leurs inconvénients spécifiques. La méthode de l'accès à distance ne génère pas d'effets ping-pong, ni de problèmes de cohérence. Par contre, elle est très limitée dans le nombre d'accès, car elle génère beaucoup de trafic sur le réseau d'interconnexion. La migration des données, améliore le niveau de trafic en cas d'accès répétés aux mêmes données. Mais, elle introduit

dans certaines situations l'effet ping-pong. La duplication des données permet de limiter l'effet ping-pong et de conserver un trafic acceptable. Cependant, il instaure un nouveau problème, la cohérence des données.

Ces trois méthodes, sont le reflet de trois types de partages des données. L'idéal serait d'avoir une cohabitation des trois, et chaque partage de données, choisirait la méthode qui lui convient. Cela pose deux problèmes, faire cohabiter les trois méthodes prend de la place, et surtout, il faut des informations, pour savoir laquelle choisir. Ces informations, nous montrant les besoins de chaque partage de données, ne peuvent venir que de la compilation ou du programmeur.

2.1.5 Les mémoires attractives.

Une manière originale de faire coopérer la réplication et la migration des données provient du concept de mémoires attractives [HLH92]. La mémoire principale d'un processeur, est utilisée comme un niveau supplémentaire de cache (voir figure 2.4).

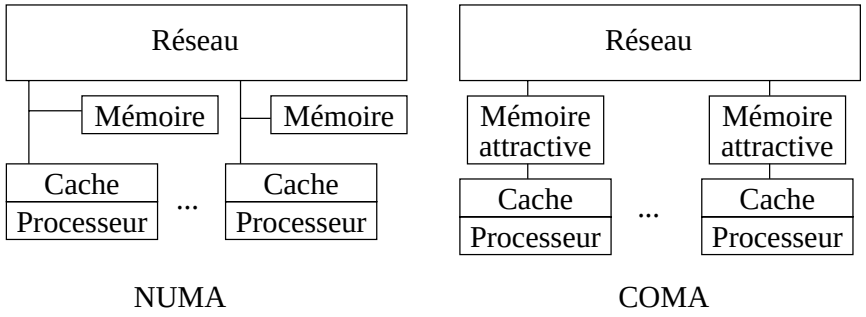


FIG. 2.4 – *Comparaison entre les architectures Non Uniform Memory Architecture et Cache Only Memory Architecture.*

Une donnée n'a donc plus d'emplacement fixe dans la mémoire principale, ce qui permet au partage des données de ne plus être obligatoirement statique. Si nous la comparons à la réplication classique, elle ne diffère que par l'absence de cette position fixe de référence. Cette méthode permet d'attirer les données vers les endroits où elles sont le plus utilisées de façon dynamique. De cette manière, elle est aussi performante qu'une architecture NUMA, dont l'optimisation du placement des données aurait été fait statiquement. Par contre, comme elle perd la notion d'emplacement fixe pour une donnée, lors d'une requête, il faut soit diffuser celle-ci ce qui n'est pas forcément aisé ou performant sur certains réseaux, soit maintenir une structure de données de l'emplacement de chaque donnée ce qui peut être volumineux.

2.1.6 L'accès aux données et la SVM.

La SVM, se base essentiellement sur la réplication des données. C'est pourquoi, la cohérence des données est un problème à résoudre pour la mettre en œuvre (voir le chapitre 3). Il n'est pas exclu cependant d'utiliser en parallèle la migration. En effet, dans la réplication, chaque donnée à un site de référence auquel il faut s'adresser si l'on veut avoir une copie de celle-ci. Si l'on envisage un mécanisme où ce site de référence puisse bouger d'un site à l'autre, la migration et la réplication sont satisfaites toutes les deux. Toutefois, cette possibilité n'a pas été implantée dans la première version de l'émulateur qui n'utilise que la réplication.

2.2 Diverses techniques utilisées ou à venir pour les multi-processeurs.

Dans cette partie, est cité un ensemble de recherches qui abordent des sujets très différents. Ces sujets évoquent des aspects de très bas niveau matériel comme la taille des blocs de caches et d'autres de haut niveau comme la synchronisation. Tous ces sujets ont toutefois toujours un point commun, ils influencent plus ou moins l'accès aux données et donc la hiérarchie mémoire constituée. Pour chaque, nous présenterons les relations qu'ils peuvent avoir ou ne pas avoir avec la SVM.

2.2.1 La mise en file d'attente des requêtes en lecture.

Dans un contexte uniprocesseur, les requêtes de lectures ne pouvaient pas être mises en file d'attente car la continuation des opérations du processeur était dépendante du résultat de la lecture et qu'il n'existait qu'un processeur donc, il n'y avait pas de compétition. Avec plusieurs processeurs c'est différent, car les lectures peuvent être mises en file d'attente sur les lieux où la lecture doit se faire. Ce stockage des lectures, fait même partie intégrante du protocole de communication. Il permet d'ordonner les demandes de tous les processeurs et dans certains cas, de soulager le réseau. En effet, certaines requêtes émanant de processeurs différents peuvent être regroupées si elles concernent le même bloc mémoire et tirer parti d'éventuels moyens de diffusion du réseau pour réduire le trafic. De ce fait, l'adjonction de files d'attente en entrée et en sortie du réseau permet de mieux harmoniser le dialogue entre les différents noeuds de calcul et le réseau d'interconnexion (voir figure 2.5).

Cette mise en file d'attente des requêtes ne posent pas de problèmes pour être réalisée dans une SVM. Une demande de page ou de ligne de cache sont très similaires. L'émulateur

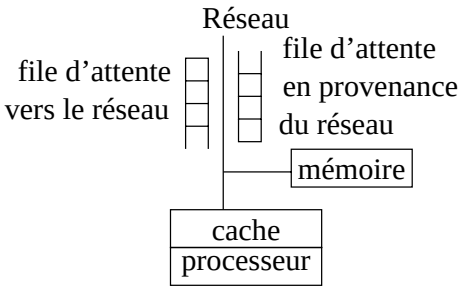


FIG. 2.5 – *Représentation d'un noeud de calculs avec des files d'attente en entrée et en sortie.*

que nous avons réalisé assure la mise en attente des requêtes de pages.

2.2.2 Les caches à taille de blocs variables.

Le fait que, quelque soit l'application à exécuter par le processeur, la taille des blocs de cache soit fixe peut être un inconvénient. Une taille trop grande peut entraîner un chargement de données qui n'auraient pas dû l'être. Inversement, une taille de blocs trop petite peut être à l'origine d'un nombre de fautes de caches plus important car le principe de localité spatiale n'est pas assez utilisé. Ceci n'est pas réservé aux multi-processeurs, et aurait donc pu très bien figurer dans les axes de recherches dans le domaine des mono-processeurs. Mais, avec plusieurs processeurs, le problème prend une dimension nouvelle. Il ne s'agit plus d'optimiser les échanges entre différents niveaux d'une hiérarchie mémoire, mais aussi de réduire le trafic d'un réseau d'interconnexion. Avec un seul processeur, une taille de blocs inadaptée influait uniquement sur les performances des accès à la mémoire. Mais avec plusieurs processeurs, une taille de blocs inadaptée a pour conséquence une augmentation du trafic réseau et donc une baisse de performance d'une toute autre ampleur. Il s'agit donc ici, d'adapter au maximum la taille des blocs du cache en fonction de l'application. Deux blocs de cache contigus sont réunis s'ils sont utilisés par le même processeur ensemble et un bloc est coupé en deux si les accès effectués sur la partie haute n'ont aucun rapport avec ceux de la partie basse [Dub93].

Si l'on transpose cette étude dans un contexte de SVM, il faut discuter de la taille des pages utilisées. Cette taille est celle utilisée par le gérant de la mémoire virtuelle. Si l'on veut rester strict, et ne pas intervenir dans la mémoire virtuelle, il est impossible de modifier cette taille et encore moins de la faire varier dynamiquement.

2.2.3 La synchronisation.

La synchronisation reste un important sujet de recherche à l'intérieur des systèmes multi-processeurs à mémoire partagée [Rai92]. Indépendamment du trafic lié au maintien de cohérence, le trafic lié à la synchronisation inhérente aux programmes à mémoire partagée reste présent et peut même être très volumineux. Par exemple, l'utilisation par plusieurs flots d'exécution de l'instruction Test-and-set sur le même emplacement mémoire engendre un trafic incessant sur le réseau d'interconnexion, car la valeur de l'emplacement est à chaque fois réécrite même si le verrou n'est pas disponible. Pour réduire ce trafic réseau, l'instruction Test-and-test-and-set est apparue. Elle réduit le trafic des processus en attente du verrou car elle supprime la modification de la valeur de celui-ci, en testant sa valeur avant de faire le Test-and-set.

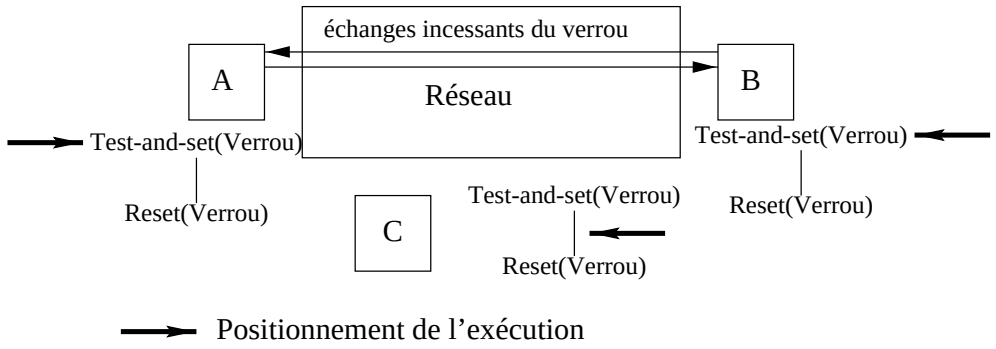


FIG. 2.6 – *Illustration du problème de saturation du réseau avec le Test-and-set.*

La figure 2.6 montre le problème du test-and-set. Si le site C a acquis le verrou, les deux autres sites qui sont en compétition pour ce verrou vont continuellement s'échanger la variable convoitée du fait de l'écriture réalisée par le test-and-set. Tandis que le test-and-test-and-set charge la donnée et la teste localement tant que le site détenant le verrou ne le lâche pas. Lorsque celui-ci modifie le verrou pour le remettre à zéro, il s'agit d'une écriture, donc les copies existantes sur d'autres sites sont altérées et le premier test du test-and-test-and-set peut être franchi pour exécuter le test-and-set. Cependant, le test-and-test-and-set ne peut pas empêcher l'assaut des processus concurrents lors de la libération du verrou. De plus, le concept de Test-and-set a un inconvénient, il est trop simple, et il ne permet pas une grande souplesse dans la réalisation des sections critiques. Par exemple, il permet l'entrée en section critique d'un seul processeur alors que le programmeur peut vouloir créer des sections critiques où un nombre limité de processus peuvent être présents. C'est pourquoi, plusieurs autres primitives de synchronisation sont apparues, comme compare-and-swap et Load-linked/Store-conditionnal [MS94].

Basés sur ces primitives de bas-niveau, il existe des primitives de haut-niveau qui permettent de réaliser des sections critiques pour les verrous ou des rendez-vous entre plusieurs flots d'exécution pour les barrières. Ces deux primitives de haut niveau sont les plus répandues et leur implantation sur différentes machines parallèles devient indispensable. Cependant, ceci n'exclut pas la recherche d'autres formes de synchronisation de haut-niveau et surtout des implantations plus performantes de celles existantes.

L'exploitation de la synchronisation pour le maintien de la cohérence, est une idée qui est très bien adaptée à la SVM. Elle est étudiée en détail en 3.1.2 et reprise dans notre émulateur.

2.2.4 L'adressage 64 bits.

Le passage d'un système d'adressage sur 32 bits à un système d'adressage sur 64 bits risque de remettre en cause certaines notions de conception des hiérarchies mémoires. Il faut déjà bien se rendre compte de ce que représente un tel espace d'adressage. Si l'on devait en lire tous les mots au rythme de 100 megaoctets par seconde, il faudrait environ 5000 ans pour arriver au bout de cette tâche. Autant dire que la disponibilité d'une adresse virtuelle est assurée [CLLBH92]. Ceci permet de ne pas réutiliser les adresses déjà utilisées et d'avoir donc un adressage unique non seulement dans l'espace d'adressage, mais aussi dans le temps. Cet adressage unique permet de résoudre les problèmes de synonymie des caches adressés virtuellement [CLBHL92]. En effet, une adresse représente toujours la même chose quel que soit l'entité qui l'utilise. Il n'y a donc plus de problème de synonymie possible car chacun utilise directement l'adresse virtuelle qui désigne l'objet référencé. Cet espace unique à 64 bits d'adresse permet un partage aisé de structures de données complexes. Dans un système classique, le partage de données contenant des pointeurs se heurtait au fait que ces données pouvait être référencées à des endroits différents d'espaces virtuels distincts. Ce problème ne se pose plus dans un espace d'adressage unique à 64 bits.

Par contre, le fait de donner à chaque utilisateur un espace virtuel propre dans lequel on «couplait» des segments partagés, permettait d'assurer facilement des mécanismes de protection. S'il n'existe plus qu'un seul espace virtuel totalement partagé, un système de protection complet doit être construit par ailleurs.

La mise en œuvre d'un tel système d'adressage peut être très intéressant pour la SVM. Il améliore grandement la portabilité des programmes en supprimant le problème des pointeurs à l'intérieur des structures de données. Cela aurait été très utile pour l'adaptation de certains programmes à notre émulateur.

2.3 Résumé.

Jusqu'à maintenant, nous avons présenté les hiérarchies mémoires et les différentes manières de les gérer. Pour chaque technique présentée, nous avons essayé de justifier leur utilisation ou non dans un système à base de SVM. De toute cette étude, il ressort que le placement des données est un problème crucial, si l'on veut avoir un accès rapide à celles-ci. On peut d'ores et déjà dire que la SVM doit utiliser pour être efficace la réplication des données, du fait des limites de performance des réseaux actuels. L'utilisation de la réplication implique forcément un problème de maintien de cohérence des données. Or, nous avons entrevu dans ce chapitre que la gestion de la cohérence pouvait être couplée avec la synchronisation afin de permettre le regroupement des requêtes d'écritures sur une même page dans certaines conditions.

Il manque donc à notre tour d'horizon des techniques utilisées dans les hiérarchies mémoires, tous ce qui concerne l'affaiblissement de la cohérence des données, afin de limiter la charge du réseau. Ce sujet est abordé au chapitre suivant.

Chapitre 3

La cohérence des données.

Avant de présenter les différents modèles de mémoires qui existent, il faut définir le terme de modèle de mémoire. Un modèle de mémoire au sens de la cohérence des données communes accédées par des exécutions parallèles est l'ensemble des moyens mis en jeu pour fournir au programmeur des accès à la mémoire dotés d'une sémantique bien définie. Dans un tel environnement, il arrive souvent que l'on se pose des questions sur ce que va nous renvoyer un accès à la mémoire. Définir un modèle mémoire, c'est donc pouvoir répondre à ces questions à tout instant.

3.1 Présentation des différents modèles de mémoires.

Il existe deux grandes familles de modèles de mémoires, les modèles uniformes et les modèles hybrides. Ces deux familles ainsi que les modèles qui les composent sont présentés dans [Mos93]. Pour les modèles uniformes, tous les accès mémoires sont de même type (lecture ou écriture) et obéissent au même protocole de cohérence. Par contre, pour les modèles hybrides, il faut distinguer les accès qui sont astreints à certaines règles et les autres. Les premiers permettent de rétablir la cohérence des données de temps en temps, alors que les seconds permettent d'accéder aux données sans se préoccuper de l'intégrité de celles-ci.

3.1.1 Les modèles uniformes.

Dans les sections qui suivent, cinq types de cohérences uniformes sont définies. Elles sont présentées ci-dessous dans l'ordre décroissant des contraintes:

- la cohérence atomique,
- la cohérence séquentielle,

- la cohérence causale,
- la cohérence processeur et
- la cohérence à «mémoire lente».

Pour expliquer les différences de ces divers types de mémoires, plusieurs schémas sont utilisés par la suite. Ils respectent les conventions suivantes. Ces schémas sont une juxtaposition de traces non-exhaustives de ce qui se déroule sur le processeur correspondant ligne par ligne. En ordonnée, on trouve donc les processeurs prenant part au problème, et en abscisse, le temps absolu, qui se déroule de gauche à droite. Les événements présentés par ces traces sont de deux types (écritures et lectures). L'écriture de la valeur 1 dans la variable x sera notée Wx(1) et la lecture de la variable y rendant la valeur 2 sera notée Ry(2). Cette façon de noter les traces est inspirée de [Mos93]. Pour des raisons de clarté des explications, les différentes sortes de mémoires ne sont pas présentées tout à fait dans l'ordre décroissant des contraintes. Cependant, la première présentée reste quand même la cohérence atomique.

3.1.1.1 La cohérence atomique.

Le modèle mémoire supportant la cohérence atomique est le plus strict de tous. Il faut que chaque lecture, effectuée sur un certain emplacement mémoire, retourne la valeur écrite, à ce même emplacement, par la dernière écriture ayant eu lieu dans l'ensemble du système. Il autorise donc l'existence, à un instant donné (comme le montre la figure 3.1), soit d'un rédacteur, soit d'un ou plusieurs lecteurs.

P1	Wx(1)	Rx(1)						
P2		Rx(1)		Rx(2)				
P3		Rx(1)	Wx(2)				Wx(4)	
P4		Rx(1) Rx(1)		Rx(2)	Wx(3)			

FIG. 3.1 – *Exemple de cohérence atomique, chaque zone contient soit un écrivain, soit un ou plusieurs lecteurs.*

Chaque opération, doit avant de se dérouler, s'assurer qu'aucune autre opération n'est en compétition avec elle. Si c'est le cas, une des deux seulement devra s'exécuter, et la seconde ne pourra le faire qu'une fois toutes les modifications sur la mémoire (introduites par la première opération) ne soient propagées par le réseau. En faisant l'analogie avec le problème bien connu des lecteurs rédacteurs, ce modèle autorise plusieurs lecteurs à lire

les données ensembles, mais ne permet aux écrivains ni d'écrire ensemble, ni d'écrire en parallèle avec des lectures. Ces contraintes obligent donc à définir un ordre total sur les opérations d'accès à la mémoire.

3.1.1.2 La cohérence séquentielle.

La cohérence séquentielle est un peu plus souple que la cohérence atomique. Elle n'exige plus que toutes les lectures renvoient la valeur dernièrement écrite. Elle remplace cette contrainte par une obligation pour tous les processeurs à percevoir les écritures dans un ordre total et unique. Cet ordre doit respecter l'ordre des écritures qui se déroulent sur un même processeur. Ce n'est que lors de l'entrelacement des différents ordres des différents processeurs qu'un certain degré de liberté est donné.

P1	Wx(1)							Wx(3)		Rx(4)					
P2			Wx(2)		Wx(4)										
P3						Rx(1)			Rx(2)		Rx(3)		Rx(4)		
P4				Rx(1)						Rx(2)				Rx(3)	Rx(4)

FIG. 3.2 – *Exemple de cohérence séquentielle.*

La figure 3.2 nous montre un exemple d'exécution qui obéit aux critères de la cohérence séquentielle, mais pas à ceux de la cohérence atomique. En effet, la première lecture de la variable X par le processeur 3 se déroule après l'écriture de cette même variable par le processeur 2. Or, le résultat retourné n'est pas ce qui a été dernièrement écrit. Donc, la cohérence de cet enchaînement n'est pas atomique. Par contre, chaque processeur est d'accord sur l'ordre des écritures affectant la variable X (Wx(1), Wx(2), Wx(3) et Wx(4)), tout se passe donc comme si les écritures étaient réalisées de manière séquentielle selon cet ordre. De plus, comme celui-ci respecte l'ordre de chaque processeur (Wx(1), Wx(3) pour P1 et Wx(2), Wx(4) pour P2) l'exemple de la figure 3.2 est bien de cohérence séquentielle.

Une autre manière de décrire la cohérence séquentielle est de faire référence à l'entrelacement des processus dans un système séquentiel. Dans une exécution parallèle d'un programme à mémoire partagée, pour que le modèle mémoire suivi puisse être un modèle séquentiel, il faut pouvoir trouver un entrelacement des différents flots d'exécution de telle manière que si ceux-ci avaient été exécutés de manière séquentielle sur un seul processeur, le résultat sur la mémoire soit identique à l'exécution parallèle. Si cet entrelacement est possible, le modèle de mémoire est au moins séquentiel (soit atomique, soit séquentiel),

par contre, si ce n'est pas le cas, le modèle mémoire respecté est moins strict que le modèle séquentiel.

3.1.1.3 La cohérence processeur.

La cohérence séquentielle demande que tous les processeurs observent le même entrelacement des écritures et que cet entrelacement respecte l'ordre d'écriture de chaque processeur. Si l'on retire la première contrainte, il ne reste plus que l'obligation de garder l'ordre fournit par chaque processeur pour les écritures qu'il a lui-même exécuté. Cette seule condition suffit à la cohérence processeur qui est donc plus souple que la cohérence séquentielle.

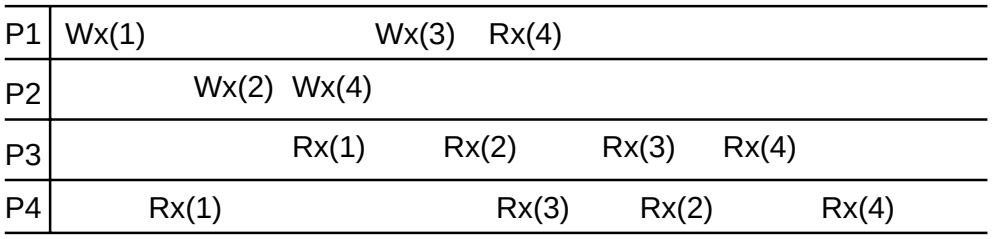


FIG. 3.3 – *Exemple de cohérence processeur.*

Dans la figure 3.3, les processeurs P3 et P4 ne sont pas d'accord sur l'ordre des écritures pour la variable x, l'un considère l'ordre Rx(1), Rx(2), Rx(3) et Rx(4) et l'autre l'ordre Rx(1), Rx(3), Rx(2) et Rx(4). De plus, ces deux ordres diffèrent de l'ordre absolu des écritures Wx(1), Wx(2), Wx(4) et Wx(3). Cependant, P3 et P4, malgré leurs différences, respectent l'ordre imposé par P1 (Wx(1) puis Wx(3)) et l'ordre imposé par P2 (Wx(2) puis Wx(4)). Ceci suffit à respecter la cohérence processeur.

3.1.1.4 La cohérence causale.

La mémoire à cohérence causale se classe du point de vue des contraintes entre la mémoire à cohérence séquentielle et la mémoire à cohérence processeur. Elle est plus souple que la première car elle n'impose pas une vision unique de l'entrelacement des écritures; elle est plus stricte que la seconde, car elle rajoute des dépendances entre les écritures autres que celles simplement imposées par l'ordre d'exécution sur un processeur. Ces dépendances de causalité permettent d'ordonner pour tout le système des écritures survenues sur deux processeurs différents.

La figure 3.4 montre par une flèche en pointillé une dépendance de causalité induite par cet exemple (il en existe d'autres, mais elles ne sont pas représentées). Les flèches pleines

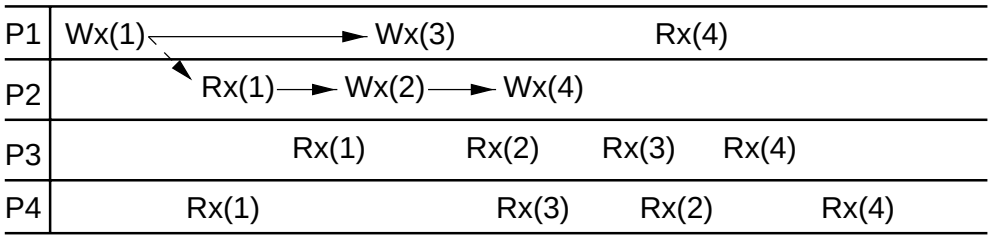


FIG. 3.4 – *Exemple de cohérence causale.*

désignent les dépendances induites par le processeur. La lecture de la valeur 1 par P2 montre que P2 a été informé de l'écriture dans la variable X de 1, sinon, c'est 0 qui aurait été lu et non 1. Comme la lecture de 1 s'effectue avant l'écriture de 2, cela instaure un ordre à respecter entre Wx(1) et Wx(2). De ce fait, tous les processeurs, doivent percevoir l'écriture de 1 avant celle de 2. Cette contrainte est respectée par P3 et P4. Elle le serait aussi, si un processeur faisait simplement une lecture de 2 mais pas de 1 ou s'il ne faisait aucune lecture des deux. Par contre, la lecture de 2 avant une lecture de 1 instaurerait une violation de la cohérence causale.

3.1.1.5 La cohérence à «mémoire lente».

La cohérence à «mémoire lente» est encore plus souple que la cohérence processeur. En effet, contrairement à cette dernière qui impose un respect de l'ordre du processeur quel que soit l'emplacement mémoire accédé, la «mémoire lente» n'impose un respect de l'ordre du processeur que pour chaque emplacement pris un à un. Cela revient exactement à dire que chaque écriture se diffuse lentement dans le système et que deux écritures s'effectuant sur le même processeur peuvent parvenir à un autre processeur dans l'ordre inverse de leur exécution, si elles ne concernent pas les mêmes emplacements mémoire.

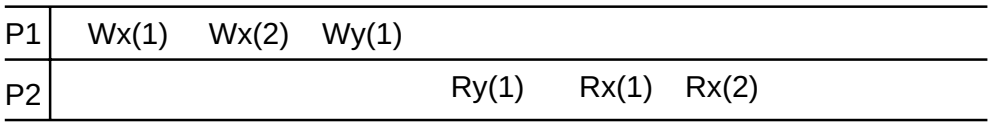


FIG. 3.5 – *Exemple de cohérence à «mémoire lente».*

Selon la cohérence processeur, la première lecture de X par le processeur P2 de la figure 3.5 devrait rendre la valeur 2, comme l'impose l'ordre du processeur P1 (Wx(1), Wx(2) et Wy(1)). Il y a donc violation de cette cohérence du fait que l'ordre entre Wx(2) et Wy(1) n'est pas respecté. Par contre, si l'on prend les deux variables X et Y séparément, l'ordre

du processeur P1 pour chaque variable (1 puis 2 pour X et 1 pour Y) est respecté. Cet exemple est donc cohérent du point de vue de la cohérence à «mémoire lente».

3.1.2 Les modèles hybrides.

Les modèles de mémoires hybrides font la distinction entre deux sortes d'accès à la mémoire. Il y a les accès libres qui se font sans contraintes en temps normal et les accès contraints qui permettent une synchronisation des accès libres. L'idée générale est de se servir de la nécessité des synchronisations, afin de réduire le trafic induit par le maintien de la cohérence. Les synchronisations permettent de créer des sections critiques à l'intérieur desquelles les accès aux variables partagées sont libres. Le maintien de la cohérence est moins coûteux car elle n'est rétablie avec l'extérieur de la section critique qu'à la fin de celle-ci ou au début de la suivante. Ceci permet de regrouper les communications dont l'objet est le maintien de la cohérence. Dans la partie qui suit, les différents modèles mémoires hybrides sont expliqués.

3.1.2.1 La cohérence faible.

La cohérence faible définit des points de synchronisations qui sont représentés par des accès privilégiés [DSB86]. Les accès normaux à la mémoire sont libres en dehors de ces points de contrôle. C'est au voisinage des points de contrôle que des restrictions interviennent. Lors de la survenue d'un accès privilégié, celui-ci ne peut s'effectuer qu'a la fin de tous les accès normaux en cours. En attendant de pouvoir se réaliser, l'accès privilégié bloque tout démarrage d'accès nouveaux car il ne faut pas qu'un nouvel accès débute alors qu'un accès privilégié est en cours (figure 3.6).

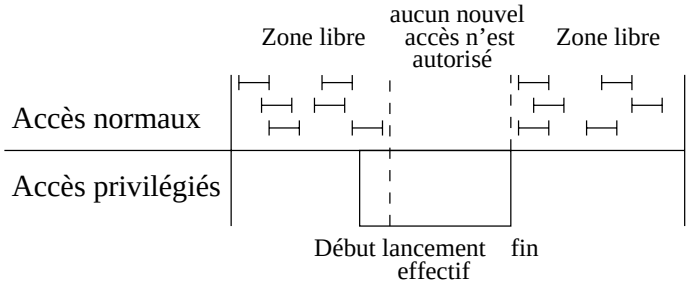


FIG. 3.6 – *Représentation des contraintes imposées par un accès privilégié.*

En plus de ces règles qui régissent les rapports entre accès normaux et accès privilégiés, il faut que tous les accès privilégiés obéissent aux règles de la cohérence séquentielle de manière à être séquentiellement consistants entre eux.

3.1.2.2 La cohérence à la libération.

La cohérence à la libération [GLL+90] est un peu plus souple que la cohérence faible. Elle introduit deux points différents de synchronisations. Le premier permet d'acquérir un verrou et le second de le relâcher. L'acquisition du verrou garanti qu'aucun accès normal à la mémoire ne sera plus effectué par les autres processeurs avant que le verrou ne soit relâché. L'accès au verrou suspend donc les accès normaux. Ceux-ci ne seront autorisés de nouveau que lors du relâchement du verrou. Le fonctionnement est synthétisé dans la figure 3.7.

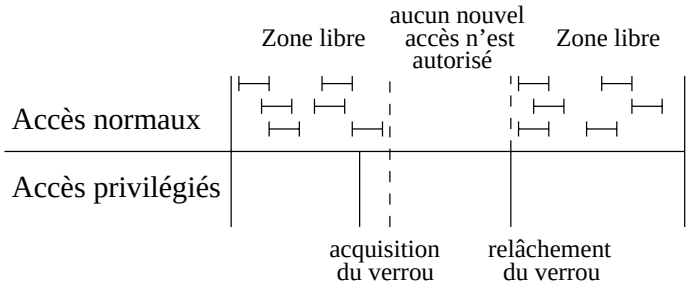


FIG. 3.7 – *Influence de l'acquisition et du relâchement d'un verrou sur les accès libres.*

Dans ce type de maintien de la cohérence, les accès privilégiés doivent se plier aux contraintes de la cohérence processeur.

3.1.2.3 La cohérence «à l'entrée» (entry consistency).

Sur le principe, la cohérence à l'entrée [BZ91] est identique à la cohérence à la libération. Pourtant, elle est plus permissive. Elle oblige l'association des mémoires partagées accédées à l'intérieur des sections critiques à être liées aux variables de synchronisation de celles-ci. Ceci permet de rendre indépendants les accès à des sections critiques qui ne concernent pas les mêmes variables. Par exemple, si une section critique manipule les variables partagées A,B,C et que pour une autre section critique, ce soit D,E,F. Les ensembles sont disjoints donc les deux sections critiques ne sont liées par aucune contrainte ce qui n'est pas le cas avec une cohérence à la libération. L'association entre section critique et variables partagées est dynamique. La cohérence à l'entrée, distingue les accès exclusifs à une section critique et les non-exclusifs. Ceci permet d'assouplir encore plus les contraintes entre les sections critiques, car deux sections critiques concernant au moins une variable partagée en commun peuvent s'exécuter sans contrainte, si celle-ci est accédée de façon non-exclusive.

3.2 Influence des modèles mémoires sur l'architecture.

La compatibilité entre un modèle de cohérence mémoire et une architecture de machine n'est pas toujours assurée. C'est-à-dire qu'une architecture donnée ne peut pas forcément héberger n'importe quel modèle de mémoire, à moins d'utiliser de coûteux protocoles. La cohérence de la mémoire exige donc un minimum de propriétés de la part du matériel. Par exemple, il faut un réseau capable d'ordonner les requêtes de manière à pouvoir fournir une vue séquentielle d'une exécution, sinon, l'implantation de certains modèles, risque d'être très inefficace.

En plus de cette contrainte sur les réseaux, l'adjonction de tampons et surtout l'optimisation de la gestion de ces tampons peuvent être remises en cause par l'adoption d'un modèle de mémoire uniforme. C'est pourquoi ceux-ci avec leurs multiples contraintes sont en perte de vitesse. Il faut en effet supprimer trop d'optimisations dans la gestion des tampons ce qui ne rend plus rentable l'opération.

Il faut donc trouver pour une relation optimale entre le matériel et le choix d'un modèle mémoire, un compromis entre les contraintes du modèle qui simplifie l'utilisation du modèle mémoire et entre les optimisations possibles du matériel.

3.3 Influence des modèles mémoires sur la programmation.

Le choix d'un modèle mémoire précis influe sur le cadre de travail d'un programmeur. Le plus difficile, est de réaliser les primitives de synchronisation pour un modèle donné. Ces synchronisations ne sont cependant réalisées qu'une seule fois par un programmeur système. Le programmeur d'application, doit connaître la sémantique des accès à la mémoire pour réaliser des programmes corrects. La grande tendance de la relation entre les modèles de mémoire et la programmation, réside dans l'unification du maintien de la cohérence avec la synchronisation. C'est pourquoi, les modèles hybrides qui réalisent en partie cette union sont promis à un bel avenir. Ils permettent en effet de fournir à chacun la donnée qu'il veut sans ce soucier du maintien de la cohérence, celui-ci est réalisé par les primitives de synchronisation. C'est ce que montre la figure 3.8 avec un exemple de barrière entre deux flots d'exécution. Dans cette figure, les modifications apportées aux données ne sont diffusées que lors des barrières de synchronisations. Ainsi, une fois une barrière passée, tous les processeurs peuvent utiliser la valeur d'une variable définie

avant la barrière par n'importe lequel des processeurs. Autre avantage, si une donnée est modifiée à plusieurs reprises avant une barrière de synchronisation, seule la dernière modification sera diffusée. C'est le cas dans la figure 3.8 pour le second flot d'exécution et la variable B entre les deux barrières.

Jusqu'a ces dernières années, le nombre de modèles mémoires n'a fait que croître. Il semble que l'inflation soit terminée et que la tendance soit à l'unification et donc à la simplification des modèles. En effet, Adve et Hill dans [AH93] présentent un nouveau modèle de mémoire qui résulte de l'unification de quatre autres modèles. Cette tendance à l'unification qui permet de garder ce qu'il y a de mieux dans chaque modèle, tout en conservant une bonne rapidité et en gagnant en simplification, semble être une bonne voie pour réaliser des modèles simples et efficaces.

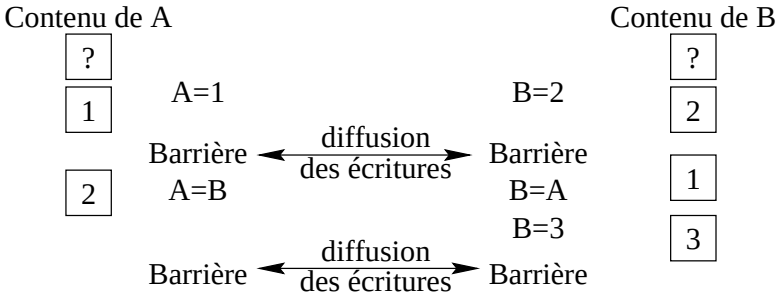


FIG. 3.8 – *Exemple d'interactions entre le modèle de mémoire à la libération et une barrière de synchronisation.*

3.4 Importance de l'évaluation de ces méthodes.

Dans l'optique de la réalisation d'un système de SVM, il est essentiel de définir quelle cohérence choisir. Pour cela, il faut pouvoir évaluer ces systèmes en faisant varier un bon nombre de paramètres. Ces paramètres peuvent être le nombre de processeurs employés, la taille des données manipulées ou le type de maintien de la cohérence utilisée. L'application qui doit être exécutée sur un tel système influe aussi sur le choix de ces différents paramètres. C'est pourquoi, il faut être en mesure d'utiliser des jeux d'essais variés et significatifs.

Pour ce qui concerne la validation de l'émulateur réalisé, nous avons choisi l'ensemble de programmes SPLASH [SWG92] et SPLASH2 [WOT+95] pour nos expérimentations. Ces programmes ont été conçus à l'université de Stanford. Ils regroupent une grande variété de programmes réels qui sont spécialement écrits pour des machines parallèles à

mémoire partagée. Le but des concepteurs de ces programmes est de mettre à la disposition des concepteurs de machines des applications variées, réelles et qui permettent des évaluations conséquentes, afin de permettre les simulations de nouvelles architectures. C'est pourquoi un bon nombre d'études sur l'architecture des systèmes parallèles ont repris ces programmes pour leurs expérimentations. Ces programmes ont donc au moins deux bonnes raisons de nous convenir. Ils sont écrits pour des machines à mémoire partagée, or notre émulateur doit permettre d'évaluer des systèmes dont la mémoire est virtuellement partagée, ce qui assure une adaptation minimum. Enfin, ils ont toutes les qualités requises pour permettre une bonne évaluation.

Maintenant que nous avons choisi le type de programmes qui vont nous servir à évaluer un système à base de SVM, il nous faut présenter le type et les fonctionnalités de l'émulateur que nous avons construit. C'est le but de la seconde partie de cette thèse.

Deuxième partie

La réalisation d'un émulateur pour SVM.

Chapitre 1

Les différents types de simulateurs.

La seule manière d'éviter des simulations lors de l'étude de nouvelles machines, est de construire réellement celles-ci. Cependant, pour construire une machine efficace, il faut régler certains paramètres de façon optimale. Ce réglage ne peut se faire que par des simulations, ou par la construction d'un nombre important de machines réelles qui ne serviront qu'à se rendre compte qu'elles sont mal dimensionnées. Or, la construction d'une machine réelle est d'un coût importante. C'est pourquoi, la simulation est utilisée, afin de valider certaines innovations avant de construire des machines les supportant.

Le simulateur idéal d'une machine est celui qui permet d'obtenir des résultats de simulation identiques à ceux que l'on aurait obtenu par une exécution sur une machine réelle, avec des temps de réponse proches de celle-ci, et qui permet de faire varier facilement les paramètres qui constituent la machine. Ce simulateur idéal n'existe pas. Le but est donc de s'en rapprocher le plus possible. Il faut donc concilier la vitesse, la précision et la souplesse de la simulation. Ce qui se représente dans le graphe 1.1 par la volonté de se rapprocher le plus possible de l'origine où se situent les performances d'une machine réelle.

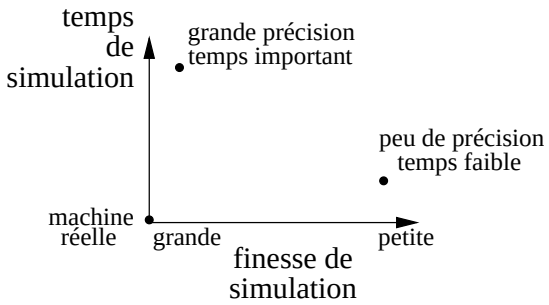


FIG. 1.1 – *A la recherche d'un simulateur idéal.*

Le graphe 1.1 ne représente pas la notion de facilité de variation des paramètres constituants la machine. Il faut voir celle-ci comme une troisième dimension du problème. Dans cette dimension, à l'inverse des autres dimensions, il faut, pour obtenir un simulateur performant où l'on puisse faire varier beaucoup de paramètres, s'éloigner le plus possible d'une machine réelle, car celle-ci est par construction figée.

Pour réaliser des simulateurs, il existe plusieurs manières de faire qui diffèrent selon leur aptitude à approcher le simulateur idéal.

1.1 La simulation complète.

La première méthode de simulation consiste à tout simuler. Il faut interpréter les instructions, simuler le contenu des registres et de la mémoire. Cette méthode n'est pas satisfaisante car, si la simulation est précise, le temps pour simuler devient énorme. Un exemple de simulateur fonctionnant de la sorte, peut être trouvé en [Lar90].

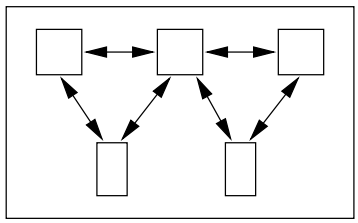


FIG. 1.2 – *simulation d'un système composé de 5 sous-systèmes.*

La simulation complète du système de la figure 1.2 nécessite la simulation des 5 sous-systèmes et de leurs interactions. Pour améliorer la rapidité de cette simulation, il est possible de regrouper des sous-systèmes. Mais, cela entraîne une perte d'information sur les données échangées par ces sous-systèmes. Ainsi qu'une possibilité de perte de précision dans les échanges restants.

1.2 La simulation guidée par les traces.

Ce procédé permet de ne pas tout simuler et donc, d'obtenir une simulation assez rapide. Ici, le simulateur reçoit en entrée des traces qu'il interprète pour pouvoir fournir des résultats. Ces traces sont constituées par l'enregistrement d'une suite d'événements qui adviennent sur une partie de la machine. Cet enregistrement nécessite le plus souvent un dispositif matériel afin de perturber le moins possible l'exécution tracée. Les traces dépendent beaucoup de la partie étudiée de la machine simulée. Si l'on s'intéresse à la

hiérarchie mémoire, il faudra avoir des traces d'accès à la mémoire; pour la communication avec d'autres machines, il faudra disposer des traces des communications avec le réseau, etc. Les traces d'accès à la mémoire doivent être constituées au minimum de la localisation de l'accès mémoire, de son type (écriture ou lecture) et d'une manière d'intégrer le temps.

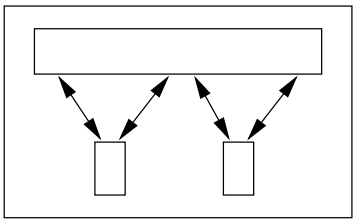


FIG. 1.3 – *simulation d'un système avec la méthode des traces.*

Par exemple, la figure 1.3 reprend le système précédent, mais remplace les 3 sous-systèmes du haut par un seul élément qui fourni les traces des échanges entre les sous-systèmes remplacés et leurs homologues restants. Cette façon de procéder permet de ne simuler entièrement que la partie étudiée de la machine (ici les sous-systèmes du bas). Ce fait explique qu'elle soit plus rapide que la simulation complète. Mais la précision de la simulation dépend en grande partie de la qualité des traces utilisées. C'est pourquoi, au fil du temps se sont développées plusieurs méthodes de génération de traces.

1.2.1 Les méthodes statistiques ou aléatoires.

Cette classe de méthodes regroupe des traces qui sont déterminées par le calcul. Le mode de calcul peut-être une génération de traces aléatoires (plutôt pseudo-aléatoires) ou bien une fonction mathématique qui est censée reproduire le comportement de ce qui, dans la réalité, génère les traces Cette méthode est présentée par [Spi77] et est illustrée par la figure 1.4.

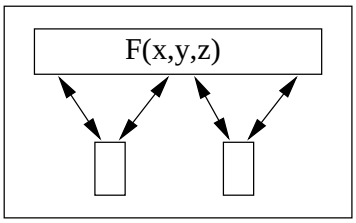


FIG. 1.4 – *simulation d'un système avec des traces générées par une fonction mathématique.*

La difficulté est ici de bien reproduire le comportement réel et de ne pas escamoter les attitudes qui sortent de l'ordinaire en se concentrant trop sur une moyenne. Quel que soit la fonction retenue, elle ne sera qu'une approximation du comportement réel, ce qui peut ne pas convenir dans certaines situations, c'est pourquoi la méthode suivante a été explorée.

1.2.2 La prise de traces réelles.

Pour prendre des traces réelles, il faut disposer d'une machine réelle dont le comportement de la partie à l'origine des traces doit être assez proche de la partie qui doit être remplacée par ces mêmes traces dans la machine à étudier. La figure 1.5 montre les deux machines, celle sur laquelle les traces sont prises et celle sur laquelle celles-ci sont utilisées. La prise des traces apporte certaines contraintes quant au choix des modules simulés et des modules remplacés par des traces. Les interactions entre ces deux types de modules doivent permettre l'utilisation des traces. C'est à dire que la partie simulée ne doit pas influencer la partie remplacée par les traces, car celles-ci sont figées. Il est possible d'illustrer ceci en prenant l'exemple d'un gérant de cache.

Lorsque le sous-système simulé de la machine étudiée n'influence pas les données qui lui sont fournies en entrée, il est possible de générer des traces à partir de machines réelles (un exemple parmi d'autres est donné dans [Cla83]). Le sous-ensemble de cette machine qui fourni les traces devra correspondre au sous-ensemble de la machine simulée en matière de comportement. La figure 1.5 montre que les traces doivent être unidirectionnelles et que la machine servant à générer les traces doit être proche de la machine simulée, car les traces doivent pouvoir s'adapter au module simulé et elles doivent correspondre au module qui n'est pas simulé, mais qui est remplacé par les traces.

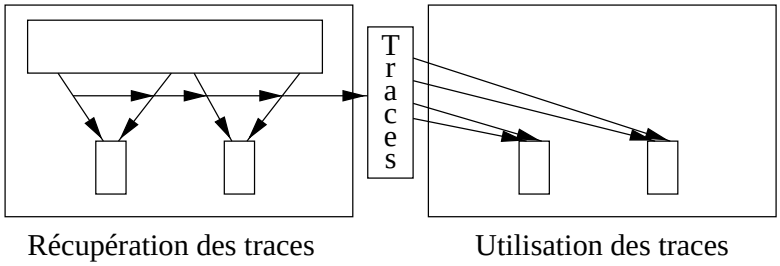


FIG. 1.5 – *simulation d'un système avec des traces générées par une exécution antérieure.*

L'utilisation des traces implique une grande contrainte. La coupure entre les modules simulés et ceux remplacés par les traces, n'autorise pas les interactions dans le sens des modules simulés vers les modules remplacés par les traces. Ceci limite grandement les

possibilités de simulation par cette méthode. C'est pourquoi, la méthode de simulation guidée par l'exécution est apparue.

1.3 La méthode de simulation guidée par l'exécution.

Il faut dans ce cas séparer la machine à simuler en deux parties. Une partie est exécutée et une autre partie est simulée. Cette séparation à un double avantage, d'une part, la simulation est concentrée sur la partie intéressante de la machine, c'est à dire la partie étudiée, et d'autre part la partie non-étudiée est exécutée, de manière à accroître la rapidité de l'ensemble de la simulation (un exemple de ce procédé peut être trouvé dans [Boo94]). Dans notre exemple, cela se traduit par une coupure du système en deux, d'une part les modules du haut, qui ne sont pas étudiés, sont exécutés, et d'autre part, les modules du bas sont simulés. La circulation des informations dans le système reste entière. C'est à dire qu'il n'y a plus de limitations comme dans la manière de simuler précédente. Tout ceci est illustré par la figure 1.6.

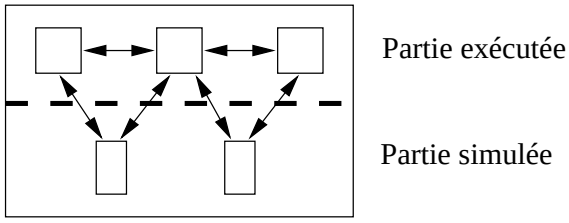


FIG. 1.6 – *simulation d'un système exécuté en partie.*

1.4 Solution retenue.

Le simulateur qui a été développé par notre équipe vise à étudier le comportement d'une machine parallèle à mémoire distribuée dans le cadre d'une utilisation impliquant une unique mémoire virtuelle partagée. Une simulation complète n'est pas envisageable, car elle prendrait trop de temps vu la taille des problèmes visés. Une simulation à l'aide d'une récupération de traces, n'est pas possible non plus car le problème que l'on veut étudier ne le permet pas. En effet, le but du simulateur étant de comparer l'efficacité des différentes formes de maintien de cohérence de la mémoire, il n'est pas possible d'utiliser les traces des accès à la mémoire car la façon dont sont gérés ceux-ci peut influer sur les accès futurs. Il ne reste donc comme solution que d'opter pour la simulation guidée par l'exécution. Dans ce cas, les applications qui servent de base à la simulation, sont

exécutées, elles ne font appel au simulateur que lors des fautes de pages. L'acheminement des pages à travers un réseau et le maintien de la cohérence de la mémoire sont alors simulés. C'est parce que cette technique mélange l'exécution directe et la simulation (des fautes de pages et de leur acheminement) qu'on l'appellera souvent «émulation». Nous avons donc défini et réalisé cet émulateur.

Chapitre 2

Description du simulateur.

2.1 Principes généraux.

Comme nous l'avons vu dans le chapitre précédent, il faut, pour répondre du mieux possible aux critères de rapidité, de précision et de polyvalence, un émulateur qui exécute les applications et qui simule le dialogue d'un processeur avec ses voisins lors d'une faute de page due au partage des données. Pour cela, nous avons construit un émulateur qui est schématisé dans la figure 2.1.

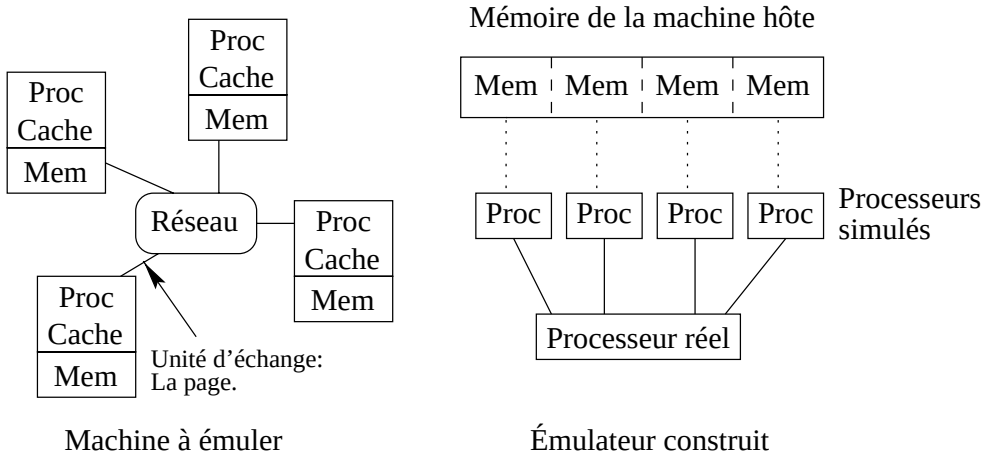


FIG. 2.1 – *Schémas du système à émuler et de l'émulateur construit.*

Cet émulateur doit s'exécuter sur une seule machine. C'est pour cela qu'il faut partager le processeur et la mémoire. La façon dont ceux-ci sont partagés est décrite dans les deux parties suivantes.

2.1.1 Le partage du processeur.

L'émulateur s'exécute sur un seul processeur, il faut donc simuler l'exécution parallèle de plusieurs processeurs sur un seul. Pour cela, nous faisons appel au «partage de temps»: les processeurs simulés utilisent chacun leur tour le processeur réel. Durant le temps mis à leur disposition, ces processeurs exécutent réellement la partie de l'application qui leur incombe.

Pour gérer l'entrelacement des exécutions des différents processeurs, il faut connaître le temps que chaque processeur simulé à déjà passé à utiliser le processeur réel. C'est pourquoi il faut gérer pour chaque processeur un temps virtuel qui nous permette de diriger l'ordonnancement des processeurs par un échéancier. Cet échéancier classe donc les processeurs dans l'ordre croissant de leur temps virtuel avec celui qui possède le plus petit temps virtuel en tête de liste. Cette simulation par échéancier se place donc dans la catégorie des simulations «dirigées par les événements» (event-driven).

L'exécution de chaque processeur est donc comme dans tous les systèmes partagés une suite de périodes d'exécution et de changements de contexte. Toutes ces exécutions et ces changements de contexte, s'ils sont mis bout à bout, reconstituent l'exécution du processeur et permettent comme le montre la figure 2.2, de suivre l'évolution du temps local à chaque processeur ou temps simulé.

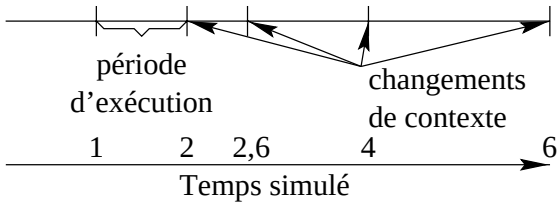


FIG. 2.2 – *Détail de la représentation d'un processeur et suivi du temps simulé.*

Le temps simulé évolue de manière croissante, en fonction du temps passé à s'exécuter sur le processeur réel par le processeur correspondant. Dans l'exemple de la figure 2.2, les périodes d'exécution ne sont pas constantes, ce qui est reflété par un accroissement du temps simulé en conséquence. Dans le cas précis de notre émulateur, les périodes d'exécution sont à peu près constantes, en l'absence de requêtes de pages et de synchronisations. Elles sont de plus courte durée, si le processeur fait une requête de page ou a recours à des synchronisations car dans cette situation, l'exécution repasse à l'émulateur. Cette irrégularité dans les durées d'exécution des processeurs, ne gêne pas l'ordonnancement, car celui-ci réagit toujours en fonction du temps simulé propre à chaque processeur (voir la figure 2.3).

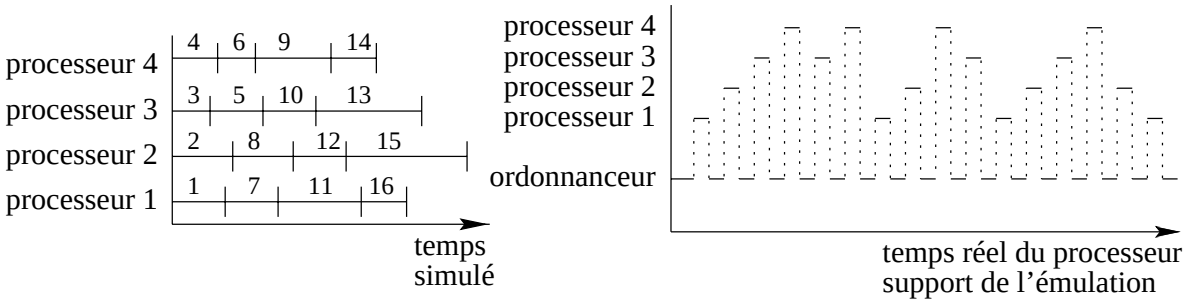


FIG. 2.3 – *Exemple d'un ordonnancement possible entre 4 processeurs en temps simulé et réel.*

L'émulateur choisit toujours comme futur processeur à s'exécuter celui qui possède le temps simulé le plus faible. La figure 2.3 montre le déroulement de l'exécution en numérotant toutes les périodes d'exécution dans l'ordre chronologique de leur exécution réelle. Lorsqu'un processeur arrive à la fin du temps alloué pour sa période d'exécution courante ou si celui-ci fait une requête, l'émulateur reprend alors la main, pour élire le prochain processeur devant s'exécuter. La figure 2.3 permet aussi de voir le rapport qui existe entre le temps simulé et le temps réel lors d'une exécution.

L'élection du prochain processeur à s'exécuter et le lancement effectif de son exécution sur le processeur réel sont réalisés par une partie de l'émulateur que nous appellerons par la suite l'ordonnanceur.

Y a-t-il un risque de mauvais ordonnancement et de ce fait, une obligation de retour arrière ?

Tant que les processeurs ne travaillent pas sur des données partagées, il n'y a pas de risque d'erreur. En revanche, l'introduction du partage de données peut introduire de mauvais séquencements d'exécutions qui font que la simulation n'est pas fonctionnellement équivalente à ce que serait l'exécution parallèle réelle. C'est le cas lorsqu'un processeur lit une donnée avant que celle-ci ne soit écrite par un autre processeur, alors que le programmeur pensait lire la nouvelle valeur et non l'ancienne. Pour contourner ce type de problème nous nous limitons dans un premier temps à des applications programmées à l'aide de primitives de synchronisation comme lock, unlock et barrier. C'est toujours le cas avec les applications tirées de «SPLASH». De ce fait, le bon déroulement du programme est toujours garanti. En effet, les variables partagées, ne changent pas de valeur et ne sont pas lues par des processeurs différents à l'intérieur d'une même zone de programme, à moins que cela n'ait pas d'importance. Le processeur qui lit une variable partagée le fait après avoir franchi une barrière de synchronisation; et en ce dernier point, son exécution n'est lancée que s'il est le moins avancé, c'est à dire, si les autres ont déjà réalisé les

écritures.

2.1.2 Le partage de la mémoire.

En ce qui concerne la mémoire, il faut que chaque processeur simulé dispose d'une mémoire qui lui soit propre pour l'exécution des programmes. Cette mémoire est disponible aisément avec les systèmes d'exploitation actuels par l'intermédiaire de mémoires virtuelles et de la pagination. Les processeurs doivent signaler leurs besoins en données par l'intermédiaire de fautes de pages sur leur zone mémoire. Ces fautes de pages sont récupérées par le système et traitées. Leur traitement dans un système réel consiste en un échange de messages sur le réseau d'interconnexion. Dans l'un de ces messages, se trouvera la page demandée. Cependant, comme tout l'émulateur réside sur une seule machine, il faut simuler cette circulation des messages à travers le réseau d'interconnexion. Pour cela, une partie de l'émulateur que nous appellerons par la suite le paginateur, est chargée de cette circulation. Mais comme il faut maintenir une certaine cohérence des données, le paginateur est aussi chargé de recevoir et de satisfaire les différentes demandes de pages émanant des processeurs simulés, et de gérer la circulation des messages d'invalidations et d'acquittements nécessaires au maintien de la cohérence. Les processeurs doivent signaler leurs besoins en données par l'intermédiaire de fautes de pages.

2.2 Le micro-noyau Mach.

Pour assurer l'exécution des programmes de «SPLASH», qui sont des programmes réalistes et parfois importants, il faut disposer d'un noyau de système d'exploitation. Mais comme nous souhaitons pouvoir intervenir au niveau du traitement des fautes de pages et de l'ordonnancement, il nous fallait choisir un système ouvert, et auquel l'accès est raisonnablement facile.

Nous avons choisi Mach.

Le micro-noyau Mach a été développé au départ à l'université de Carnegie-Mellon [ABB+86]. Il a été conçu pour exploiter aussi bien des architectures mono-processeurs que des architectures multi-processeurs. C'est un micro-noyau, il fournit donc un petit ensemble de briques de base qui permettent la réalisation efficace d'une grande variété de systèmes d'exploitation. Les buts principaux de Mach sont d'exploiter le parallélisme aussi bien dans le système d'exploitation que dans les applications des utilisateurs, d'utiliser au mieux un espace d'adressage étendu et potentiellement dispersé avec un partage de mémoire souple, d'autoriser un accès transparent au réseau et d'être portable.

Un autre but du projet Mach est d'extraire toujours plus de fonctionnalités du noyau. Cette externalisation des fonctionnalités a une limite. Il n'est pas possible d'externaliser la gestion des flots d'exécution, la distribution de la mémoire physique, des processeurs ou des périphériques. Par contre, il est possible de définir des paginateurs externes au noyau qui peuvent prendre en charge la gestion d'une partie de la mémoire virtuelle.

Pour réaliser tous ces buts, le micro-noyau Mach fournit un certain nombre d'abstractions qui sont détaillées dans ce qui suit.

En ce qui concerne l'exécution, il faut distinguer deux abstractions différentes: la tâche et le «thread». Le «thread», appelé aussi processus léger, représente une unité d'exécution minimale. Il appartient à une tâche et une seule. Une tâche contient un ou plusieurs «threads», ainsi que tout ce qui est nécessaire à l'exécution de ceux-ci. Il s'agit principalement d'un espace d'adressage partagé et de ports de communications. Dans ces conditions, pour obtenir un processus UNIX, il faut créer une tâche qui ne contienne qu'un «thread».

La communication entre entités est assurée par des canaux appelés ports. Ces ports sont accessibles selon les droits d'accès dont on dispose. Ainsi, dans une communication classique entre deux entités, l'une aura les droits en écriture, et l'autre en lecture.

La mémoire est gérée par objets. Ainsi, des pages qui disposent de propriétés identiques peuvent être regroupées dans un même objet. A partir de ces objets sont construits des messages pour servir dans les communications.

Pour avoir une présentation plus détaillée du noyau Mach, il faut se reporter à [Ope92a].

Pour accueillir notre émulateur, nous avons donc choisi le micro-noyau Mach [Ope92b], d'une part en raison des fonctionnalités qu'il offre (tâches, «threads», paginateurs externes) et surtout en raison des possibilités qu'il offre pour étendre notre étude sur un réseau de plusieurs ordinateurs. La suite de ce chapitre présente la façon dont l'émulateur a été intégré à Mach.

2.3 Tâches ou «threads»?

Dans l'optique de la réalisation du simulateur, plusieurs possibilités d'implantations étaient possibles. Les deux solutions extrêmes sont présentées par les figures 2.4 et 2.5. Par ailleurs, il existe des solutions intermédiaires qui sont un compromis des deux possibilités précédentes.

La première possibilité consiste à placer un «thread» par tâche, afin de permettre une extension aisée sur des sites distants. Par contre, comme l'espace d'adressage d'une tâche n'est pas partagé, les communications entre les «threads» sont moins simples car il faut utiliser des messages.

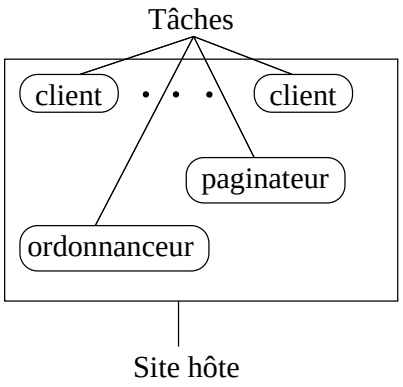


FIG. 2.4 – *Première implantation possible du simulateur.*

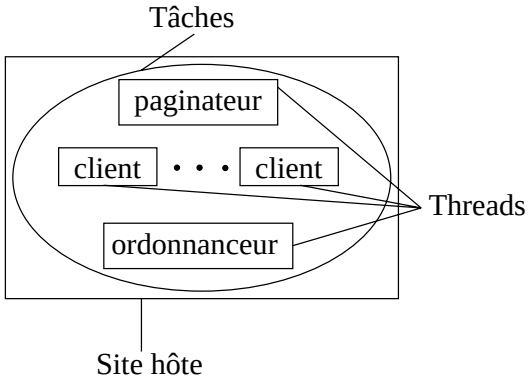


FIG. 2.5 – *Seconde implantation possible du simulateur.*

La seconde façon de procéder privilégie la facilité de communication. Elle repose sur l'utilisation de l'espace d'adressage unique des tâches. C'est pourquoi, tous les «threads» sont réunis à l'intérieur d'une seule tâche. Ici, la difficulté est de communiquer avec d'éventuels sites distants. Cette option a été préférée à la précédente, car l'interconnexion de sites, n'était pas au départ prioritaire. Pour réaliser, celle-ci, il faudra vraisemblablement sortir le paginateur et l'ordonnanceur de la tâche, pour les mettre dans une tâche spécifique. Ceci entraînera donc une modification du mode de communication entre les différentes entités de la simulation. Mais, en gardant un certain nombre de «threads» clients à l'intérieur d'une tâche unique par site, cela permettra de simuler des machines interconnectées par groupe en simulant un groupe ou plus par machine réelle. Bien entendu, la question de la duplication de l'ordonnanceur et surtout du paginateur devra être résolue.

La figure 2.6, montre l'intégration du simulateur dans l'empilement des couches qui

constituent tout le système. On retrouve à la base de cet empilement un PC 486DX33 qui constitue la couche physique de l'émulateur. Au-dessus, la première couche logicielle est constituée par le micro-noyau Mach, celui-ci fournit des services de bases de la gestion système. Le système à proprement parler est celui qui nous a été fourni par l'OSF, en même temps que Mach. Il s'agit de la version 1.3 du système OSF/1. L'émulateur se place entre les applications destinées à être exécutées et le système OSF/1. Les applications utilisées ont été prises dans l'ensemble des applications destinées aux architectures à mémoires partagées SPLASH, puis SPLASH2.

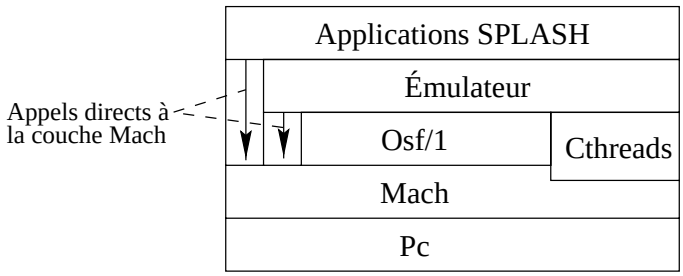


FIG. 2.6 – *Représentation en couche de tout le système.*

Ce modèle en couches comporte cependant quelques irrégularités. Il est en effet possible à certains «threads» de l'émulateur, voire même de l'application simulée, de faire des appels à certaines fonctions du micro-noyau. Le cas de la fonction clock_get_time, qui intervient dans la gestion du temps et qui appartient au noyau, permet d'obtenir la date avec une précision théorique d'une microseconde. Cette fonction est souvent appelée par l'émulateur ou les applications clientes de celui-ci, c'est un exemple parmi d'autre de violation du modèle en couches du système. Une autre violation du système des couches est l'utilisation de la librairie de «cthread». Celle-ci est basée sur l'utilisation des «threads» Mach, et permet de simplifier certaines tâches, comme le lancement d'un «thread» ou la synchronisation de ceux-ci, par l'utilisation de fonctions adéquates. L'utilisation de cette librairie, ne suffit pas, il faut pour d'autres travaux, utiliser, des fonctions, du noyau Mach. C'est particulièrement le cas pour toutes les fonctions qui permettent l'ordonnancement des «threads». L'usage de deux groupes de fonctions, ne s'appuyant pas sur une même couche, peut être un inconvénient, car du point de vue de l'utilisateur de ces fonctions, il faut gérer deux sortes de «threads» qui sont censés représenter une entité unique. De plus, ce couplage n'est pas garanti par défaut. Il faut donc le garantir par une fonction adéquate de la librairie de «cthreads», qui lie le «cthread» courant et le «thread» Mach sous-jacent, jusqu'à la mort de ceux-ci. Ce lien à pour effet, d'étendre la couche Mach, avec les fonctions de la librairie de «cthread», c'est pour cela que celle-ci est représentée

incrustée dans la couche Mach.

En conclusion, les processeurs à émuler sont représentés dans notre émulateur par des «threads» appartenant tous à la même tâche. Ces «threads» sont exécutées en tenant compte pour chacun du temps virtuel qui leur est associé. Une fois qu'un «thread» est choisi, il s'exécute pour une durée qui correspond à la valeur du quantum de temps défini pour l'ordonnancement des opérations par l'ordonnanceur du système. Ce quantum est de 10 ms. De tous ces «threads» utiles pour l'émulation, ceux qui correspondent à l'exécution de l'application choisie sont parfois appelés par la suite les «threads» clients.

2.4 Les différents types de mémoires utilisées.

Dans une machine à mémoire virtuellement partagée, la mémoire d'un nœud est le plus souvent divisée en 2 zones, une première zone pour les données privées et une seconde pour les données virtuellement partagées. Nous avons vu en 2.3, que le simulateur est composé d'une tâche unique. Cette tâche dispose d'une zone de mémoire qui est partagée entre tous les «threads». Il faut donc organiser la mémoire de manière à faire le lien entre les deux modèles.

La solution retenue est décrite dans la figure 2.7. La mémoire de la tâche est divisée en 4 zones distinctes. La première zone correspond à toutes les variables de travail du simulateur. Elle contient toutes les données correspondant aux requêtes de pages en cours, à l'état de chaque page et de chaque «thread» client. Pour recréer les deux zones présentes dans les machines distribuées, il faut réserver deux zones par «thread» client, l'une correspondra aux éventuelles données privées et la seconde aux données partagées. L'accès en lecture ou en écriture à une donnée contenue dans cette dernière, déclenche, si la donnée n'est pas disponible immédiatement une requête pour le paginateur. Ce sont les accès successifs à cette zone pour chaque client qui va représenter le besoin de partage des données, et ainsi selon la politique de maintien de la cohérence, du nombre de requêtes au paginateur.

La dernière zone nécessaire est un peu spéciale, elle sert à stocker les données qui ne peuvent pas se situer dans les autres zones. Ces données sont de deux sortes.

D'une part, il s'agit des données qui interviennent dans la réalisation de «LOCK» et de «UNLOCK» entre les «threads» clients. Ces primitives de synchronisation, sont basées sur leurs homologues fournis dans le paquetage de «cthreads», ce qui facilite grandement leur réalisation au prix d'une petite imprécision dans les simulations (pour plus de détails, voir en 2.9.3).

Le second type de données concernées par cette zone, correspond à l'ensemble des

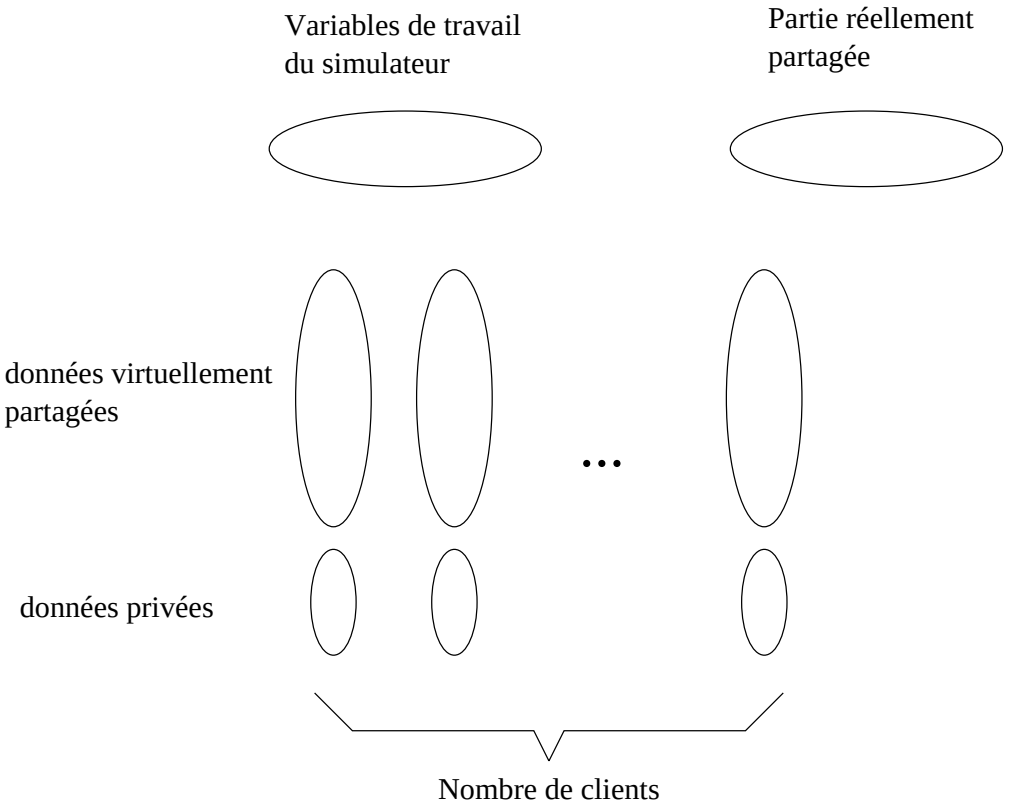


FIG. 2.7 – *Présentation des différentes zones de mémoires du simulateur.*

données qui bien qu'étant en principe partagées, ne peuvent pas être distribuées. Il s'agit par exemple de structures contenant des pointeurs. Dans la réalité, ces pointeurs auraient tous la même valeur, ainsi ils pourraient tous être partagés et distribués. Par contre, dans le simulateur, les zones correspondant à une structure de données partagées ne se trouvent pas à la même adresse (du fait de la juxtaposition des différentes zones de chaque «thread» dans l'unique zone d'adressage de la tâche). Il faut donc pour résoudre le problème extraire les pointeurs de la structure virtuellement partagée et les placer dans une zone spéciale, afin qu'ils soient distribués, mais pas partagés(Un exemple est présenté dans la figure 2.8). Il faut donc ajuster leurs valeurs en conséquence. Cette incartade à la réalité, n'a pas de grande conséquence, car les pointeurs ne représentent en principe qu'une partie minime des structures de données manipulées, et de plus, ils sont très rarement accédés en écriture, en dehors de l'initialisation du programme.

Après avoir, présenté l'organisation de la mémoire, dans notre simulateur, nous allons aborder dans la partie suivante la gestion du temps.

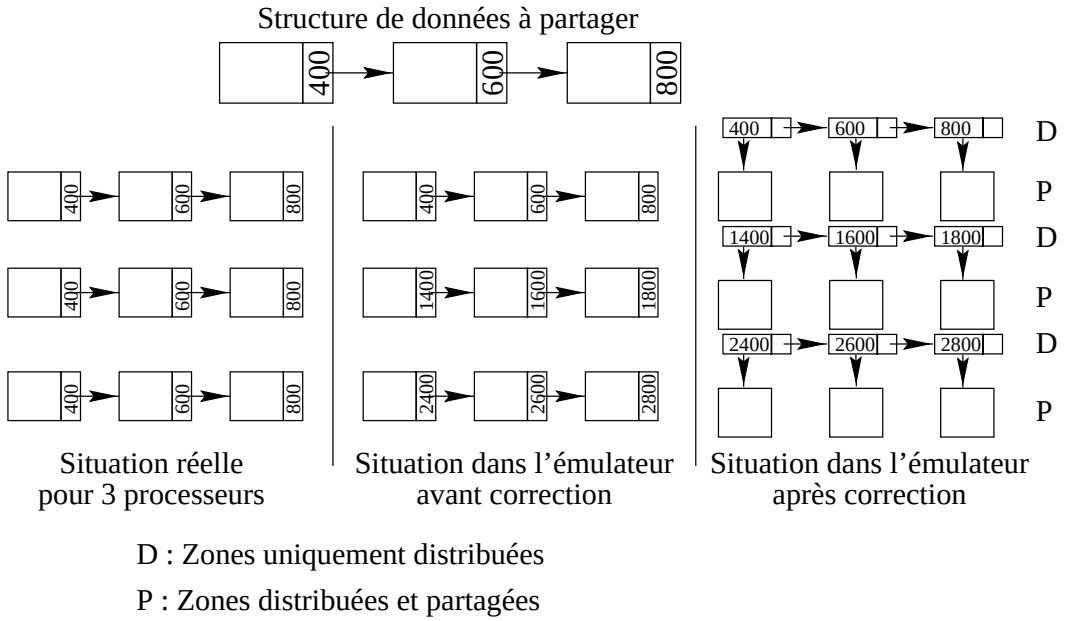


FIG. 2.8 – *Exemple de corrections des pointeurs.*

2.5 La gestion du temps.

La gestion du temps est très importante dans ce type de réalisation. Elle permet d'obtenir une exécution correcte des applications et aussi proche que possible de la réalité. Cependant, étant donné que le but du simulateur est de reproduire une exécution parallèle des applications sur un seul processeur, cette simulation ne peut être qu'imparfaite.

2.5.1 Récupération du temps pour l'ordonnancement.

La prise de temps sert principalement de base à l'ordonnancement. Pour chaque lancement d'un «thread», le temps que celui-ci passe à s'exécuter est comptabilisé et additionné au compteur de temps associé au «thread» concerné. La valeur ainsi constituée est ensuite utilisée par l'ordonnanceur pour réaliser sa tâche. L'élaboration de cette valeur est réalisée par l'intermédiaire d'une fonction particulière. Celle-ci permet d'interroger directement le micro-noyau Mach sur le temps écoulé depuis le lancement du système. Le fait de ne pas rester au niveau du système OSF/1 ou de tous serveurs UNIX équivalents permet d'avoir une très bonne résolution d'horloge. Là, où nous avions une résolution de l'ordre du quantum de temps (typiquement 10 ms), la fonction clock_get_time annonce une résolution d'environ une microseconde soit 10000 fois plus précise. Il est bien évident que cette résolution de ne peut pas être exploitée telle quelle. Un minimum de précautions doit être

pris lors de l'interprétation des résultats qui découleront de ces prises de mesures. Une série de tests a été réalisée pour déterminer la fiabilité de la mesure des prises de temps. Ceux-ci sont présentés en 2.5.3.

2.5.2 Récupération du temps en vue de statistiques sur l'exécution du programme.

L'autre activité du simulateur qui repose sur la prise de temps est celle qui permet la mesure du temps passé dans certaines zones du programme. Cette fonction repose également sur clock_get_time, mais pas uniquement sur celle-ci. Supposons que le programme à simuler se décompose en trois «threads» , tous divisé en trois zones numérotées de 1 à 3, et que l'on veuille mesurer le temps passé par chaque «thread» dans la zone 2. Il faudra alors faire un programme équivalent à celui présenté en figure 2.9.

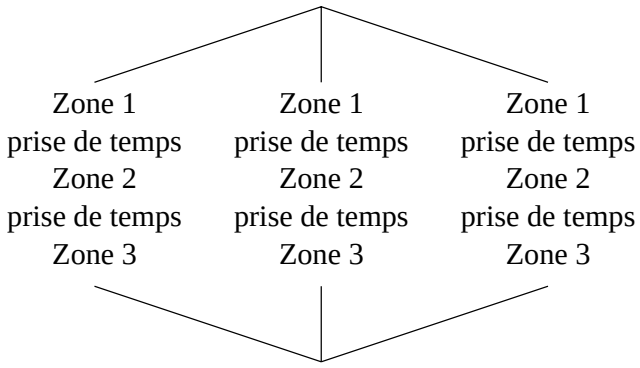


FIG. 2.9 – *Illustration de la prise de temps pour chaque «thread».*

Les prises de temps s'ajoutent à l'exécution normale du programme. Elles entrent donc dans le déroulement de celui-ci et du simulateur, ce qui constitue une source d'imprécision dans la simulation. De plus, elles ne peuvent pas se résumer en de simples appels à la fonction clock_get_time, il faut en plus utiliser le temps local à chaque «thread». C'est pour toutes ces raisons que cette activité ne doit pas être beaucoup sollicitée, afin de ne pas trop intervenir dans l'exécution du programme.

2.5.3 Tests de fiabilité de la prise de temps.

Pour réaliser l'émulateur, nous avons vu en 2.5.1, qu'il fallait une résolution plus précise que le quantum de temps pour la fonction chargée de fournir une datation précise des événements. Le but de ces tests, est de déterminer la précision maximale que l'on

puisse adopter pour l'émulateur et de vérifier que cette précision est bien inférieur au quantum de temps. Dans un deuxième temps, il faut vérifier, que l'ordonnanceur peut bien exploiter les informations que lui fourni la fonction clock_get_time.

Le premier test de fiabilité de la prise de temps consiste à stocker le plus rapidement possible une suite de valeurs résultant de l'usage de clock_get_time. Pour cela, l'appel à clock_get_time est répété un certain nombre de fois, avec entre chaque appel uniquement le stockage de la valeur obtenue et le test de continuité de la boucle. Il faut alors vérifier si la différence entre deux valeurs consécutives est à peu près constante, et dans le cas d'une réponse satisfaisante, utiliser cette valeur comme la résolution maximale utilisable. Les résultats typiques de ce test sont résumés dans le tableau 2.1. Ils se basent sur une série de 12 tests de 10000 valeurs. Ces valeurs sont les différences entre deux prises de temps consécutives. Elles sont ensuite réparties en une vingtaine d'intervalles. De ce fait, pour chaque intervalle, nous disposons de 12 valeurs. Pour calculer la moyenne de ces valeurs présentées dans le tableau 2.1, nous supprimons pour chaque intervalle la valeur minimum et la valeur maximum parmi les douze présentes. La moyenne est donc calculée sur les dix valeurs restantes. Par exemple, si pour l'intervalle > 60 et $<= 100$, on avait eu la suite des valeurs suivantes, 2,3,3,0,3,2,3,2,3,5,2,2 , les valeurs 0 et 5 auraient été éliminées et la moyenne aurait été de 2,5.

La différence entre les valeurs est donc dans presque tous les cas inférieure à 60 microsecondes. Cette valeur est toutefois dépassée dans moins de 0,8% des cas, et de façon très significative, les cas supérieurs à 200 microsecondes, dans moins de 0,4%. Cette dispersion des valeurs est due au non-déterminisme du système hôte. Si un événement prioritaire intervient pendant la requête à clock_get_time, celle-ci est arrêtée et n'est reprise qu'après la fin de la requête prioritaire. Comme ceci est inévitable, il faut prendre comme valeur de la précision de clock_get_time utilisable une valeur assez petite par rapport au quantum de temps qui est de 10 ms, mais une valeur assez grande de façon à ce que les dépassements de cette valeur soient rares. C'est pourquoi, il semble raisonnable de prendre comme précision de la fonction clock_get_time la valeur de 200 microsecondes. Cette valeur est utilisable car les 0,4% d'erreurs résiduels, sont peu fréquents, et sera pour des exécutions importantes, réparties quasi équitablement sur tous les processeurs émulés.

Un autre test à été réalisé. Il consiste à fournir au simulateur une charge de calcul générique qui ne fait aucun accès à la mémoire partagée. Cette charge de travail est obtenue en exécutant une boucle assez longue et de même durée théorique à chaque «thread» lancé, censé représenter un client du simulateur. Ce test a pour objet de vérifier si l'ordonnancement se passe bien entre les «threads» et de mesurer le temps passé à ordonnancer. Ce test montre donc le nombre de fois que chaque «thread» a été élu pour

TAB. 2.1 – *Liste du nombre de cas rencontrés pour chaque intervalle de temps considéré en microsecondes, pour une série de 10000 valeurs*

Intervalle de temps	**Nombre de cas concernés**
$<= 60$	9922,5
> 60 *et* $<= 100$	2,5
> 100 *et* $<= 200$	32
> 200 *et* $<= 300$	2,6
> 300 *et* $<= 400$	6,3
> 400 *et* $<= 500$	16,4
> 500 *et* $<= 600$	3,7
> 600 *et* $<= 700$	5,8
> 700 *et* $<= 800$	1,6
> 800 *et* $<= 900$	0,9
> 900 *et* $<= 1000$	0,1
> 1000 *et* $<= 2000$	3,5
> 2000 *et* $<= 3000$	0
> 3000 *et* $<= 4000$	0
> 4000 *et* $<= 5000$	0
> 5000 *et* $<= 6000$	0
> 6000 *et* $<= 7000$	0,1
> 7000 *et* $<= 8000$	0
> 8000 *et* $<= 9000$	0,1
> 9000 *et* $<= 10000$	0
> 10000	1,9

TAB. 2.2 – *Résumé du test d'ordonnancement.*

Nombre d'élections pour chaque «thread»				**Mesures hors norme**
thread 1	**thread 2**	**thread 3**	**thread 4**	moyenne: 17,5%
261	264	268	259	17,6%
265	252	262	268	17,8%
262	266	259	272	17,2%
269	257	268	255	16,9%
268	257	265	255	18,1%
257	251	268	271	17,7%
263	270	255	262	17,6%
265	258	263	263	17%
266	268	260	255	17,5%
265	264	258	258	18%

s'exécuter et le pourcentage de temps passé à s'exécuter hors-norme. L'anormalité est définie comme un temps d'exécution qui dépasse la durée du quantum d'un processus osf/1. Ceci est a priori impossible. Cependant, quand on liste toutes les mesures, on s'aperçoit que la durée d'exécution d'un «thread» est la plupart du temps comprise entre 7800 et 9000 microsecondes. Ce qui pour un quantum de 10 ms semble tout à fait normal car la différence est expliquée par l'exécution de l'ordonnanceur du simulateur lui-même. Il arrive toutefois que dans 17,5% des cas, la durée soit comprise entre 17800 et 19000 microsecondes et très rarement entre 27800 et 29000 microsecondes. Les résultats de dix tests sont résumés dans le tableau 2.2.

Ceci s'explique par la manière, dont est réalisé le simulateur. Le simulateur ne prend pas la place de l'ordonnanceur du système, mais se superpose à celui-ci. Ce qui veut dire que, si d'autres tâches doivent se dérouler en parallèle avec l'ordonnanceur, elles seront ordonnancées au même titre que l'ordonnanceur lui-même. Il arrive donc que le simulateur perde la main au profit d'autres tâches pour une durée de un ou deux quanta. Comme cette première série de mesures avaient été faites dans un environnement graphique avec fenêtrage, pour voir si cet environnement et tous les processus qu'il induit ont une influence sur les mesures hors-normes, une autre série de mesures identique à la précédente a été faite sur un système où uniquement le strict nécessaire se déroule. Le tableau 2.3 présente les valeurs obtenues.

La différence entre les deux séries de mesures, n'est pas significative. Donc, l'environnement fenétré, ne perturbe pas les simulations en s'accaparant du temps de calculs. Si l'opérateur ne fait pas des activités parallèles aux simulations, celles-ci peuvent donc se dérouler sans problème à partir de l'environnement graphique. Il reste donc à résoudre les

TAB. 2.3 – *Résumé du test d'ordonnancement en environnement dépouillé.*

Nombre d'élections pour chaque «thread»				**Mesures hors norme**
thread 1	**thread 2**	**thread 3**	**thread 4**	moyenne: 17,4%
256	269	259	250	17,4%
258	258	255	271	17%
258	259	255	267	17,6%
271	259	252	259	17,1%
255	269	269	264	16,7%
258	270	264	253	16,8%
260	247	276	264	17,3%
269	271	251	242	18%
259	260	260	258	17,4%
262	265	254	251	18,5%

mesures litigieuses qui dépassent le quantum de temps. Cela peut se faire en leur appliquant à posteriori un modulo au quantum afin de toujours avoir des valeurs inférieures à celui-ci. Ainsi, une valeur qui aurait intégré un ou plusieurs quanta de temps par erreur, se verrait ramener à la bonne valeur.

En conclusion de toutes ces mesures de fiabilité, il faut retenir que la précision acceptable de la fonction clock_get_time est de 200 microsecondes, et que l'ordonnancement basé sur ces prises de mesures est correct.

2.6 L'ordonnancement.

Les candidats pour l'ordonnanceur sont d'une part tous les «threads» qui appartiennent à l'application simulée et qui sont en mesure de s'exécuter une fois lancés et d'autre part, les requêtes de pages qui sont en attente. Celles-ci sont en effet datées comme les flots d'exécution et participent à l'élection pour être servi. La manipulation de ces requêtes est présentée dans la section 2.7. Le choix s'effectue sur la date portée par chaque «thread» et chaque requête. Cette date est élaborée à l'aide de la fonction clock_get_time étudiée précédemment en 2.5. Une prise de temps est réalisée juste avant le passage de l'exécution au «thread» choisi et une seconde s'effectue lors du retour de l'exécution à l'ordonnanceur. Le temps virtuel du «thread» concerné est alors incrémenté de la différence de ces deux mesures. Grâce à ce mécanisme, on connaît le temps exact que chaque «thread» a déjà passé à s'exécuter. En ce qui concerne les requêtes, la date est celle du «thread» demandeur au moment où celui-ci émet la requête.

Pour être complet, il faut évoquer ce qui se passe si un «thread» fait une requête de page et comment et a quel moment celle-ci est traitée. Si un «thread» fait une requête, il est suspendu et passe la main à l'ordonnanceur. Celui-ci lancera le paginateur qui stockera la requête dans une structure de données qui sera présentée plus en détail en 2.7. Cette requête, une fois stockée, sera lancée lorsque la date qui lui est associée devient la plus faible de toutes les requêtes et de tous les «threads» réunis. L'ordonnanceur choisi donc toujours pour la prochaine exécution parmi les requêtes stockées et les «threads» remplissants les conditions celui dont le temps simulé est le plus faible.

En ce qui concerne les requêtes, la même question que pour les «threads» peut se poser. A savoir, est-il possible d'avoir un ordonnancement des requêtes qui ne respecte pas l'ordre de l'exécution dans des conditions réelles et qui par conséquent oblige à un retour arrière?

Les requêtes sont traitées dans l'ordre croissant de leur datation. Cette datation, comme cela a déjà été présenté, repose sur la datation des «threads». De ce fait, si l'ordonnancement des «threads» est correct, la génération des requêtes le sera aussi. Il ne reste donc plus qu'à s'assurer du lancement de ces requêtes au bon moment pour que tout ce passe bien. Ceci est fait par un traitement chronologique des requêtes. Chaque requête est traitée à son tour et certaines peuvent être différées si leur traitement vient se superposer à celui d'une autre requête. Dans ce cas, la requête est repoussée jusqu'à la fin du traitement en cours.

2.7 La simulation de la pagination: le paginateur externe.

En principe, tel que le conçoit le micro-noyau Mach, le traitement des requêtes de pages est réalisé à l'aide d'un dialogue entre le micro-noyau et le paginateur qui a en charge les pages. Ce dialogue est initié par le micro-noyau et est synthétisé dans la figure 2.10.

Lors de l'exécution d'un «thread» qui accède une page absente, le défaut de page est récupéré par le micro-noyau; celui-ci adresse alors la requête au paginateur. Le traitement d'une requête par ce dernier peut être soit immédiat, soit différé. Dans le premier cas, le paginateur peut fournir la page demandée sans avoir de problème de partage. Dans le second cas, le paginateur ne peut pas fournir la page immédiatement, car les droits demandés sur la page s'avèrent incompatibles avec les droits déjà acquis sur celle-ci par un ou plusieurs autres «threads». La fréquence du second cas, par rapport au premier, est

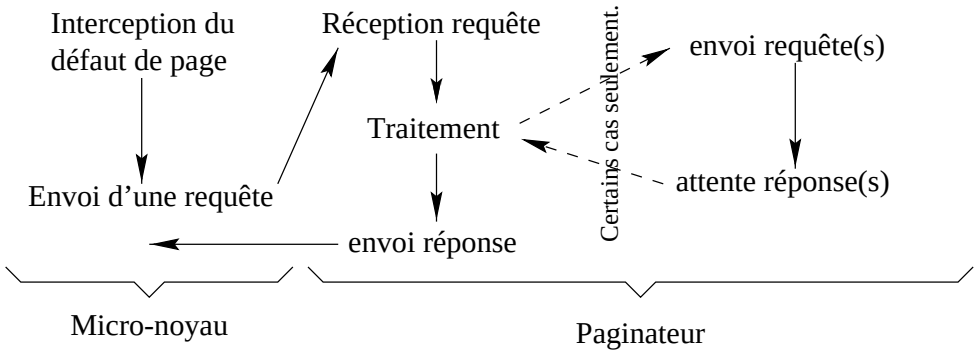


FIG. 2.10 – *Diagramme de principe du dialogue entre le micro-noyau et un paginateur.*

fortement dépendante de la méthode de gestion de la cohérence utilisée.

La gestion des requêtes de pages dans le simulateur est basée sur le dialogue précédent. Il est cependant enrichi pour permettre le stockage de toutes les requêtes et éventuellement, un réarrangement de celles-ci. Cette gestion plus complexe est présentée par la figure 2.11. Celle-ci comporte deux sortes de flèches, les barrées et les non-barrées. Ces dernières représentent un lien entre actions se déroulant forcément sur une même requête. Les flèches barrées quant à elles, représentent un retour à l'ordonnanceur. Les deux actions reliées par celles-ci appartiennent à deux appels différents au paginateur . Ce qui implique que la requête intervenant dans la première action, n'est pas forcément la même que celle intervenant dans la seconde. De ce fait, le traitement d'une requête peut se faire en deux ou trois étapes, entre lesquelles, peuvent s'intercaler des étapes qui concernent d'autres requêtes.

Le paginateur commence le traitement d'une requête par le stockage de celle-ci dans une première file d'attente triée selon les dates d'occurrence des requêtes. Ce stockage permet de différer le traitement de la requête, pour qu'il advienne au bon moment de la simulation. Une fois la requête stockée, son traitement ne pourra avoir lieu que lorsque l'ordonnanceur le décidera. Pour l'élection, celui-ci se base toujours sur la requête présente en tête de la file, c'est à dire celle qui à la date la plus faible, pour décider de la suite de l'émulation. Au cas où la requête est choisie, un appel est effectué au paginateur. Celui-ci traite la requête et stocke la réponse avec une date calculée en fonction des paramètres réseau dans une autre file qui fonctionne à l'identique de la première. Pour pouvoir élaborer la réponse, il est possible qu'il faille construire une ou plusieurs requêtes en direction du micro-noyau et peut-être attendre les réponses. Dans ce cas, ces requêtes, sont elles aussi stockées, avant d'être envoyées au micro-noyau. Elles sont stockées dans la même file que les réponses précédentes comme le montre la figure 2.12.

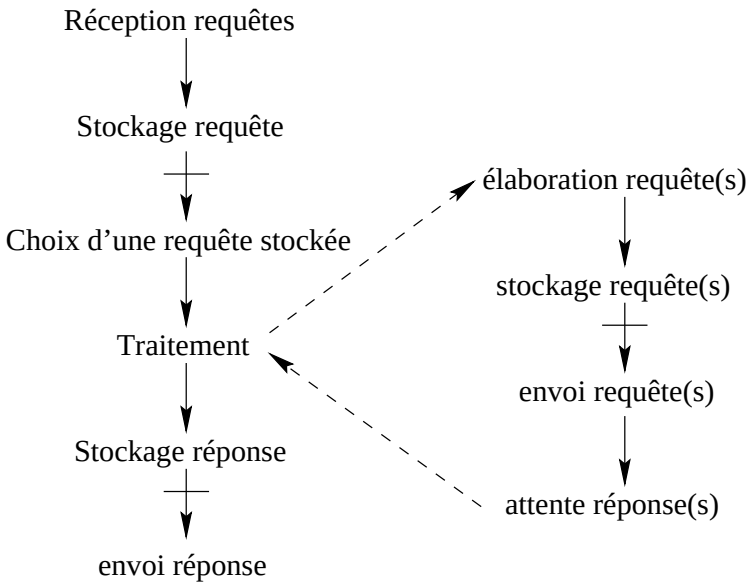
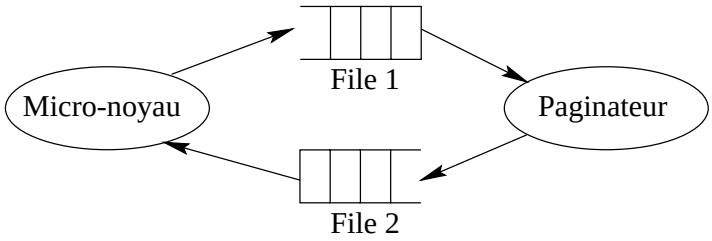


FIG. 2.11 – *Diagramme de principe du traitement d'une requête de page par le paginateur appartenant à l'émulateur.*

Enfin, la dernière partie de la gestion d'une requête, est le déstockage de la file d'attente de la réponse et l'envoi de celle-ci au micro-noyau.

A ce stade, il est important, de faire un retour en arrière sur l'ordonnancement. On a vu en 2.6 que l'ordonnancement se faisait non seulement sur les «threads» des clients, mais aussi sur les requêtes stockées. Or, les requêtes sont stockées dans deux files selon des critères de date, c'est donc en comparant les dates des deux requêtes en tête de files (toujours celles ayant la date la plus faible) avec les dates des «threads» que l'ordonnanceur choisi d'exécuter un «thread» ou de lancer le traitement d'une requête.



File 1: file de stockage des requêtes du micro-noyau
File 2: file de stockage des requêtes du paginateur et des réponses au micro-noyau

FIG. 2.12 – *Positionnement des files entre le micro-noyau et le paginateur.*

Le micro-noyau et le paginateur dialoguent ensemble pour traiter les demandes de pages. Pour cela, ils utilisent un protocole bien précis. L'émulateur que nous avons réalisé utilise le micro-noyau existant tel quel et un paginateur qui lui est spécial et qui est presque entièrement nouveau. Ce paginateur doit néanmoins respecter le protocole de dialogue lors des échanges avec le noyau. C'est ce protocole et les fonctions qui le réalisent que nous présentons dans la partie qui suit.

2.7.1 L'initialisation.

Les interactions qui concernent l'initialisation de la zone mémoire gérée par le paginateur sont représentées dans la figure 2.13. Elles débutent par un appel du client au noyau par l'intermédiaire de la fonction vm_map. Cette dernière permet de définir la taille de la zone mémoire que l'on souhaite utiliser, ainsi que le paginateur par lequel on veut qu'elle soit gérée, et enfin divers autres paramètres tel que, entre autres, les propriétés d'héritage et de protection. Une fois reçu cet appel, le noyau adresse une requête au paginateur par la fonction memory_object_init. En retour, celui-ci répond par la fonction memory_object_change_attributes, ce qui permet de spécifier certaines caractéristiques de la zone mémoire visée. Celles-ci sont, parmi d'autres, la stratégie de copie et de mise dans le cache de la zone mémoire concernée.

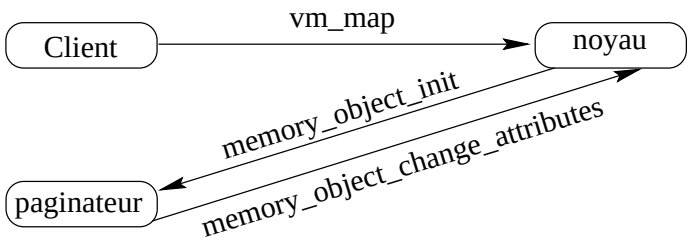


FIG. 2.13 – *Interactions lors de l'initialisation.*

2.7.2 La terminaison.

Lorsqu'un client veut libérer une zone mémoire gérée par un paginateur (cas illustré par la figure 2.14), il s'adresse au noyau par l'intermédiaire de la fonction vm_deallocate qui lui permet de désigner la zone mémoire à libérer. Celui-ci, informe à son tour le paginateur à l'aide de la fonction memory_object_terminate. Dans certains cas, il faut renvoyer au paginateur les parties ou pages de cette zone mémoire qui auraient été modifiées et dont ces modifications sont uniquement présentes dans le cache, donc ignorées du paginateur. Cette transmission s'effectue à l'aide de la fonction memory_object_data_return.

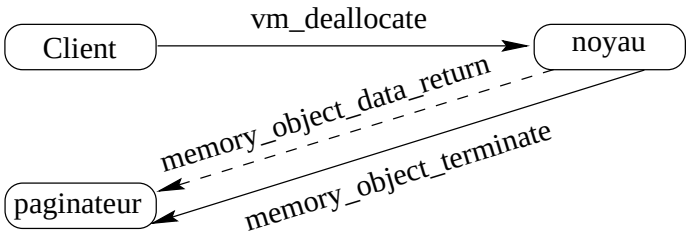


FIG. 2.14 – *Interactions lors de la terminaison.*

2.7.3 Les fautes de pages.

Si un client fait une faute de page, cela provoque une interruption que le noyau traite. Celui-ci contacte le paginateur soit par la fonction memory_object_data_request, soit par la fonction memory_object_data_unlock. Le premier cas correspond à une absence totale de la page souhaitée, il est schématisé par la figure 2.15.

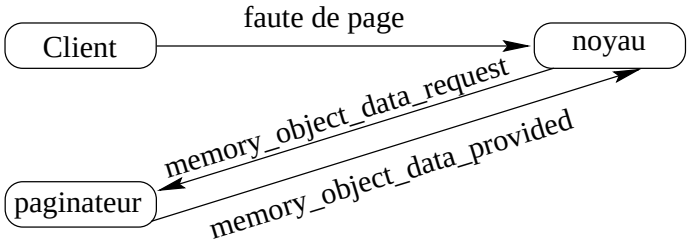


FIG. 2.15 – *Interactions lors d'une faute de page initiale.*

Le second cas survient lors d'une écriture alors que la page souhaitée était déjà possédée, mais seulement en lecture. Il est décrit en figure 2.16.

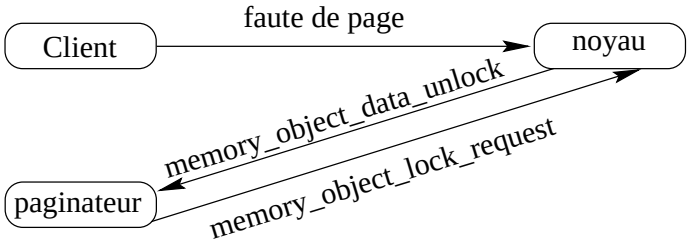


FIG. 2.16 – *Interactions lors d'une faute de page non-initiale.*

Le paginateur peut donc recevoir deux sortes de requêtes de la part du noyau, il répondra donc à celui-ci de deux manières différentes. Dans le cas, où la faute de page ne résulte que d'un problème de droits sur la page, le paginateur répondra par l'intermédiaire

de la fonction memory_object_lock_request. Si c'est réellement une faute de page initiale, il répondra à l'aide de la fonction memory_object_data_provided.

Les deux cas étudiés auparavant, étaient des cas simples. En effet, le paginateur pouvait résoudre la requête sans information extérieure. Ce n'est pas toujours le cas; il faut pouvoir dans certaines situations, invalider d'autres pages pour pouvoir répondre à la requête. Ceci s'effectue en interrogeant le noyau par la fonction memory_object_lock_request comme le montre la figure 2.17. Le noyau peut dans certains cas avoir besoin de renvoyer des pages vers le paginateur, Il le fait, comme dans le cas d'une terminaison, par la fonction memory_object_data_return. Une fois que toutes les pages concernées ont été renvoyées et que le noyau a fait tout le travail nécessaire au traitement de la demande du paginateur, il fait un appel à la fonction memory_object_lock_completed pour signaler au paginateur qu'il a le champ libre pour terminer le traitement de la requête initiale.

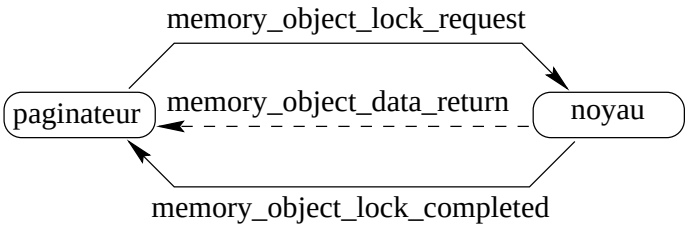


FIG. 2.17 – *Interactions nécessaires lors de certaines requêtes.*

2.8 La simulation du réseau.

Le but prioritaire des simulations effectuées a d'abord été de démontrer la propre faisabilité et l'utilité de l'émulateur. C'est pourquoi le réseau simulé est le plus simple qui soit, ce qui implique qu'il ne peut pas être le reflet d'une machine réelle. Il est considéré comme une boîte noire qui relie tous les processeurs entre eux. Quand une requête se présente à l'entrée du réseau à une certaine date, elle est délivrée au destinataire avec un certain délai. Ce délai est paramétrable non seulement selon les requêtes, mais aussi indépendamment de celles-ci. Ainsi, on peut fixer la latence et le débit du réseau, et se baser sur ces valeurs ainsi que sur la taille des requêtes pour calculer le délai d'acheminement de celles-ci. Dans l'exemple qui suit, la latence est de 1 ms, le débit est de 100 Mbits/s et la taille de la requête est de 4096 octets.

$$TEMPS = LATENCE + TAILLE/DEBIT$$

exemple:

$$temps = 1000 + 4096 * 8/100000000 * 1000000 = 1327\ microsecondes$$

Il est ainsi possible de tester grossièrement, aussi bien un réseau de type Ethernet à 10 Mbits/s, qu'un réseau ATM à 155 Mbits/s et plus. Les réseaux ATM sont par ailleurs étudiés en détail par d'autres membres de l'équipe. Il est donc envisagé de perfectionner la partie simulation de réseau de l'émulateur, pour tenir compte des spécificités du réseau ATM. Il sera ainsi possible d'avoir un outil complet de simulation de machines parallèles, fonctionnant dans un contexte de mémoires distribuées partagées, dotées d'un réseau d'interconnexion basé sur l'ATM (ou sur un tout autre réseau d'interconnexion).

2.9 Approximations et imprécisions du simulateur.

Afin de clore ce chapitre sur la présentation de notre simulateur, il faut revenir sur les approximations et imprécisions introduites dans celui-ci. Ces imperfections ont été présentées tout au long de ce chapitre. Mais, comme elles sont les garantes du bon déroulement des simulations, il est important de revenir sur chacune d'elles et de démontrer que les perturbations engendrées sont acceptables.

2.9.1 La prise de temps.

La prise de temps a déjà été étudiée avec précision en 2.5.3. Il a été conclu, qu'une précision de 200 microsecondes était tout à fait envisageable. Cette précision est à mettre en parallèle avec celle retenue pour les comparaisons des résultats qui est de 1 milliseconde. Ces deux chiffres sont bien sûr compatibles. D'une part parce que le résultat de 200 microsecondes est dans un grand nombre de cas surévalué et d'autre part, parce que les mesures sont faites plusieurs fois pour réduire la part des imprécisions.

Il est donc possible, de ce baser sur les temps recueillis au cours des simulations pour faire des comparaisons et des interprétations des résultats.

2.9.2 Les données partagées, mais non distribuables.

Comme il a été vu en 2.4 et plus particulièrement avec la figure 2.8, il existe des données partagées et distribuées dans la réalité (les structures de données qui contiennent des pointeurs en particuliers), qui du fait de la juxtaposition des mémoires distribuées dans un même espace d'adressage posent des problèmes d'adaptation à notre simulateur.

Cette imprécision est probablement la plus importante et donc la plus gênante de toutes. Elle est très dépendante de la structure des données partagées et de ce fait de l'application simulée. Elle nous dérange à deux niveaux. Le premier est évidemment l'introduction d'une certaine dose d'imperfection par rapport à la réalité. Il faut donc pour chaque application déterminer si cette imprécision est dommageable ou non. Dans l'affirmative, l'application ne peut pas être traitée par notre simulateur, ce qui réduit son champ d'investigation. Le second niveau de dérangement, repose sur le besoin de faire une adaptation pour toutes les applications qui partagent des structures de données complexes. Cette obligation va à l'encontre de la simplicité recherchée pour l'adaptation des applications en vue de leur exécution dans le simulateur.

Il faut donc en présence d'une nouvelle application, étudier la structure de données partagées, afin de déterminer si celle-ci est adaptable au simulateur, avec une précision raisonnable. Cette décision repose sur l'étude de la taille que représentent les pointeurs par rapport à la taille totale des données partagées.

2.9.3 La simulation des sections critiques.

Les sections critiques utilisées dans les applications sont à classer en deux catégories, d'une part les sections critiques qui surviennent pendant les mesures et d'autre part les autres. Nous avons vu en 2.4 que les "LOCK" et "UNLOCK" qui réalisent les sections critiques sont basés sur un paquetage de «cthread». De ce fait, le passage dans une section critique, ne génère pas de trafic de requête. Un processeur qui se bloque sur une section critique, se contente de rendre la main à l'ordonnanceur. Il ne pourra reprendre que lors de la libération de la section critique. Cette manière de faire est simple, mais engendre un risque d'erreur.

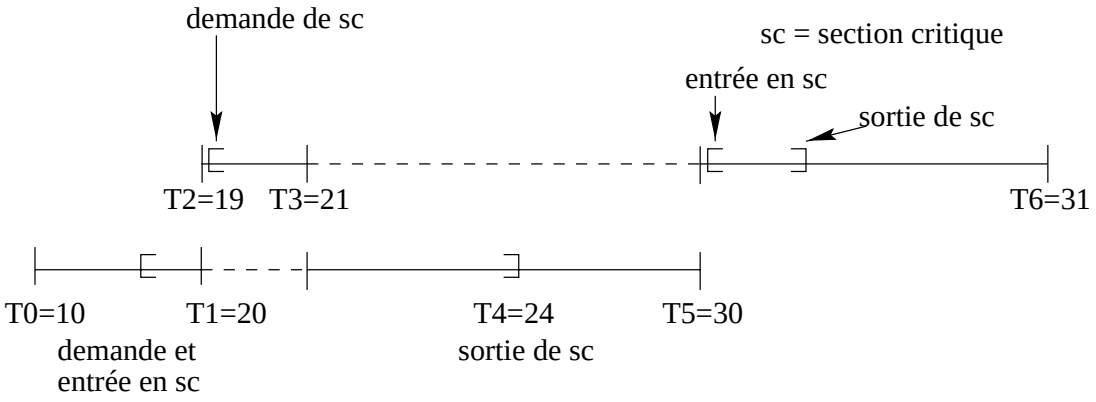


FIG. 2.18 – *Illustration du risque d'erreur dans une section critique.*

Dans la figure 2.18 le temps virtuel se déroule selon l'axe des abscisses de gauche à droite. L'exécution réelle se déroule dans l'ordre des Ti. Donc, l'expression T4=24, indique le quatrième point de l'exécution réelle, au temps virtuel du «thread» concerné 24. Il faut encore préciser que dans cet exemple, l'ordonnancement s'effectue à chaque trait vertical.

Dans cet exemple, chaque «thread» fait appel à une section critique, durant le quantum de 10 à 20 pour le «thread» du bas et au temps 21 pour celui du haut. Celui du bas rentre dans la section critique et ne la libère qu'en T4. L'erreur se produit sur le temps de reprise du second «thread». Il va reprendre son temps exécution au temps 21 alors que dans la réalité, il a dû attendre jusqu'au temps 25 pour rentrer en section critique.

Cette erreur est cependant négligeable, car la durée d'une section critique qui intervient pendant les mesures est très petite. Il s'agit pour la plupart du temps d'un accès à une variable ou un ensemble de variables qui nécessite une mise à jour atomique. L'échelle des temps de l'exemple, afin de bien montrer le problème, n'est pas réaliste, la section critique dure beaucoup trop longtemps par rapport à la durée du quantum. Les seules sections critiques qui peuvent avoir une durée assez longue, sont celles qui par exemple participent aux entrées-sorties. Cependant, elles se déroulent dans des parties du programme qui ne sont pas soumises aux mesures du simulateur et ne posent donc pas de problèmes.

Chapitre 3

Résultats et commentaires.

3.1 Présentation des différents modèles de mémoires utilisés.

3.1.1 La cohérence de type atomique.

Comme cela a déjà été présenté dans la première partie en 3.1.1.1, le modèle atomique exige une vue cohérente à chaque instant de l'exécution et une vue des écritures qui soit uniforme pour tous les processeurs. C'est pour cela que, dans ce modèle, on ne peut avoir qu'un processeur à la fois en écriture sur la page. Par contre, il est possible d'en avoir plusieurs en lecture. Pour garantir ceci, il faut, lors des traitements de requêtes d'écritures, s'assurer que la page sera fournie en exclusivité au processeur demandeur. Ceci est réalisé par l'envoi d'invalidations à chaque processeur possédant la page. Il faut ensuite attendre l'arrivée des acquittements des invalidations pour être sûr de pouvoir fournir la page dans les conditions requises. Le passage d'une page de lecture en écriture, est soumis aux mêmes contraintes. Par contre, une requête en lecture, du fait de la possibilité de cohabitation de plusieurs lecteurs, se déroule sans envoi d'invalidations envers les autres pages en lectures. En revanche, dans le cas de l'existence d'une page en écriture, celle-ci doit être invalidée. Il faut dans ce cas là, attendre le retour de la page modifiée, afin de fournir au processeur demandeur la dernière version à jour de la page et non pas une version antérieure.

Un programme peut faire trois types de requêtes:

- une requête de lecture de la page.
- une requête d'écriture de la page sans l'avoir préalablement en lecture et
- une requête d'écriture de la page en l'ayant auparavant en lecture.

A ces trois cas, correspond l'état possible de la page. Celle-ci peut être accédée en lecture ou en écriture ou ne pas être accédée. L'intersection de tous ces cas, détermine le nombre de possibilités à étudier.

3.1.2 La cohérence de type séquentielle.

La cohérence séquentielle exige uniquement que tous les processeurs soient d'accord sur l'ordre d'exécution des écritures. De ce fait, il est possible de fournir les pages sans attendre l'arrivée de certains acquittements des invalidations.

3.1.3 Les modèles hybrides.

Il existe dans les machines séquentielles tout un arsenal de techniques pour réduire la latence d'un accès mémoire. Malheureusement, ces techniques ne peuvent plus être utilisées dans le cas d'une machine parallèle à mémoire partagée qui utilise un modèle de mémoire uniforme. C'est pourquoi, les modèles hybrides ont été introduits. Ces modèles sont moins stricts que les modèles uniformes et ont pour objet de favoriser l'utilisation de ces techniques d'optimisations des accès mémoires. Ils distinguent deux sortes d'accès à la mémoire, les accès normaux et les accès de synchronisations.

La résolution des premiers, ne pose pas de problèmes particuliers. Ils sont traités au moment où ils se présentent et peuvent bénéficier de toutes les améliorations fournis par le matériel, tel un passage par un tampon des écritures.

En ce qui concerne, les seconds, ils sont soumis à des règles plus strictes. Leur présence influence l'exécution des accès du premier type car deux accès de types différents, ne peuvent pas être traités en parallèle, l'un doit être obligatoirement traité entièrement avant que le second ne débute.

La programmation des modèles s'est faite selon deux axes bien différents l'un de l'autre. L'un est basé sur le regroupement des écritures avant leur diffusion aux points de synchronisations. Il verrouille l'accès aux données tant que celles-ci sont utilisées, il est appelé LOCK par la suite. Le second exploite aussi cette notion de regroupements des écritures, mais autorise des accès à la mémoire qui étaient interdits jusqu'alors. Ainsi, des accès en lecture peuvent être réalisés de manière beaucoup plus large que dans la cohérence processeur en permettant à un écrivain de cohabiter avec un nombre quelconque de lecteurs, cette méthode est appelée 1ECRXLECT. Cette autorisation des accès auparavant interdits peut être poussée plus loin, en permettant à plusieurs écrivains de cohabiter ensemble et avec un nombre quelconque de lecteurs. Cette méthode est dénommée par la suite MULTIECRITURE.

Avant de présenter ces deux méthodes en détail, il faut présenter les moyens utilisés pour réaliser la synchronisation. Dans les programmes SPLASH qui sont présentés en 3.2.1, les primitives de synchronisation sont de deux sortes. Il y a LOCK et UNLOCK qui permettent la réalisation de sections critiques et BARRIER qui permet la réalisation de rendez-vous entre plusieurs flots d'exécution.

3.1.3.1 Présentation de l'adaptation LOCK.

L'idée principale de ce procédé est de regrouper les écritures pour ne faire qu'une seule diffusion par zone inter-synchronisation. Une zone inter-synchronisation correspond à la section critique en ce qui concerne la synchronisation de type LOCK et UNLOCK et à la zone comprise entre deux rendez-vous dans le cas d'une synchronisation par BARRIER. Le regroupement des écritures peut se faire de deux manières: soit on interdit l'accès à la même zone pour les autres «threads», soit on autorise tous les accès mais on s'efforce de rendre les données cohérentes aux points des synchronisations. La seconde méthode est décrite dans la présentation du procédé MULTIECRITURE.

L'implémentation de la méthode LOCK est réalisée de la manière suivante. Quand un flot d'exécution demande une page en lecture, celle-ci lui est donnée sans aucune autre formalité. Il faut remarquer que dans cette situation, un écrivain peut cohabiter au même moment avec un ou plusieurs lecteurs. Par contre, une demande en écriture entraîne un autre comportement.

Si pour la page, c'est la première fois qu'elle est demandée en écriture, la demande d'écriture s'effectuera normalement.

Par contre, si la page demandée est déjà en écriture, alors la demande est mise en attente et différée jusqu'à ce que le «thread» qui la détient en écriture la libère. Ceci interviendra quand celui-ci atteindra le prochain point de synchronisation, c'est à dire la fin d'une section critique ou un rendez-vous.

3.1.3.2 Présentation de l'adaptation 1ECRXLECT.

Cette méthode est en tous points semblable à la précédente en ce qui concerne les lectures. Elle permet la cohabitation d'un écrivain avec un nombre quelconque de lecteurs. Cependant, elle ne permet pas à plusieurs écrivains de cohabiter et a une méthode de gestion des conflits sur l'accès d'une page en écriture différente de la précédente. Ici, un écrivain qui est sollicité par un autre processus pour avoir ces droits d'écritures, les cède et devient un lecteur. Le droit d'écrire sur une page donnée, peut être vu comme un jeton circulant entre les différents processeurs au gré de l'exécution du programme. Ce procédé

peut avoir des conséquences néfastes, comme l'instauration d'un effet ping-pong entre des «threads». Par contre, il n'est plus possible de garder en réserve, alors que l'on n'en a plus besoin, le droit d'écrire sur une page.

3.1.3.3 Présentation de l'adaptation MULTIECRITURE.

Ce procédé est la méthode de maintien de la cohérence la plus souple qui soit étudiée. Elle essaye de tirer profit au maximum des informations qui sont fournies par la synchronisation. Ceci permet, non seulement de faire cohabiter des lecteurs avec un écrivain, mais aussi des lecteurs avec plusieurs écrivains. En observant bien la sémantique des opérateurs de synchronisation, il est en effet possible d'organiser cette cohabitation.

En ce qui concerne les rendez-vous, entre chaque rendez-vous, les flots d'exécution ne modifient pas les mêmes zones de mémoires. Dans le cas contraire, il y aurait incohérence des données, malgré l'instauration de la synchronisation. Cette discontinuité des zones modifiées permet de fournir la même page en écriture à tous les «threads» qui la demandent. Une fois la page distribuée, chaque «thread» en modifie une partie. Il faut donc au prochain point de rendez-vous regrouper toutes les modifications sur une même page afin d'en distribuer une version cohérente pour le futur.

Pour ce qui est des sections critiques, l'approche est différente. Le but de cette synchronisation, est de permettre à plusieurs «threads», de modifier un ensemble de valeurs, d'une façon atomique au regard des autres flots d'exécutions. C'est pourquoi, il n'est pas permis de laisser plusieurs écrivains sur une même page à l'intérieur d'une zone délimitée entre un LOCK et un UNLOCK. Dans ce cas là, on utilise la méthode présentée précédemment en 3.1.3.1.

3.1.4 Résumé des modèles testés.

1. Le modèle atomique autorise à chaque instant, un ou plusieurs lecteurs ou un écrivain.

2. Le modèle séquentiel peut autoriser la cohabitation des lecteurs avec un écrivain durant de courtes périodes transitoires.

3. Le modèle LOCK autorise plus largement cette cohabitation mais un écrivain garde le droit d'écriture sur une page jusqu'à la fin de la zone de synchronisation courante. Une zone de synchronisation est définie par les primitives de synchronisations. Il peut s'agir, soit d'une section critique délimitée par LOCK et UNLOCK, soit d'une section située entre deux barrières de synchronisations.

4. Le modèle 1ECRXLECT, reprend le modèle précédent, mais en permettant la circulation facile des droits d'écriture à l'intérieur des zones critiques.

5. Et enfin, le modèle MULTIECRITURE autorise toutes les sortes de cohabitation possibles, mais ceci dans des zones bien définies par les primitives de synchronisations.

Si l'on effectue, le rapprochement avec les modèles présentés en 3.1.2 de la première partie, le MULTIECRITURE est très proche de la cohérence à la libération. Quant à 1ECRXLECT et LOCK, se sont des modèles faibles, mais pas totalement, ils sont donc inclassables parmi les modèles de cohérences présentées.

3.2 Présentation du protocole de test.

3.2.1 Présentation de l'adaptation des programmes SPLASH à notre émulateur.

Les programmes de SPLASH, sont écrits en langage C et utilisent des macros. Ces macros servent surtout aux synchronisations, aux prises de temps et à la gestion de la mémoire. La présence de ces macros facilite grandement l'adaptation des programmes en vue de leur exécution sur l'émulateur. En effet, il suffit de réécrire les macros utilisées par le programme et elles peuvent être réutilisées pratiquement à l'identique pour tous les autres programmes SPLASH. Ainsi, une macro comme CLOCK qui est chargée de donner la date précise de son appel, peut être réécrite en se basant sur clock_get_time et être réutilisée par tous les programmes SPLASH qui l'utilisent.

Les macros de synchronisation utilisées dans les programmes SPLASH sont BARRIER et LOCK, UNLOCK. BARRIER sert à réaliser des barrières de rendez-vous et LOCK, UNLOCK des sections critiques. Ces macros sont donc redéfinies pour l'adaptation au simulateur. Cette adaptation dépend aussi des politiques de cohérence, il faut donc prévoir les modifications à effectuer pour chaque type de cohérence. Pour certaines cohérences retenues, l'agencement entre les synchronisations doit respecter le schéma présenté en figure 3.1 . Ce schéma peut paraître très contraignant, mais ce n'est pas le cas car, cela respecte le bon usage des synchronisations, à savoir une durée très courte pour les sections critiques et des zones de programmes entiers délimités par les barrières.

La mesure du temps est aussi effectuée par l'intermédiaire des macros. Ainsi, il suffit de modifier celles-ci, afin d'avoir des mesures de temps aux mêmes endroits que pour SPLASH.

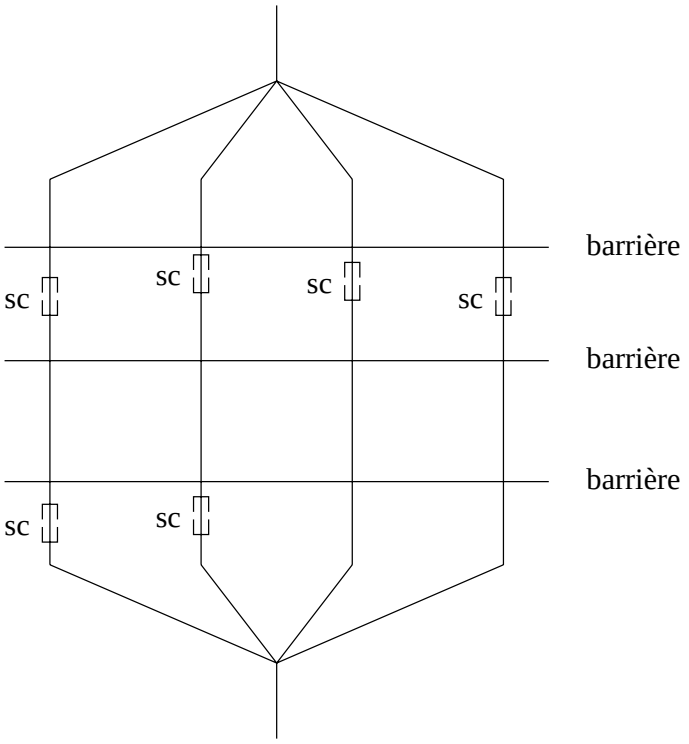


FIG. 3.1 – *Agencement des synchronisations.*

Les macros servent aussi à intégrer le simulateur aux programmes SPLASH. Pour cela, un bon nombre de macros ont été créées, afin de lancer le simulateur et tout ce qui va avec.

3.2.2 Présentation des mesures et choix des programmes.

Pour les résultats, deux types de mesures sont faites. Le premier calcule le temps total que dure la simulation, il est appelé temps global par la suite. Le second permet de savoir le temps passé par le programme SPLASH à l'intérieur de sa phase parallèle, il est appelé temps local par la suite. La différence qui existe entre ces deux durées, est illustré par la figure 3.2.

Le temps local, est pris comme référence pour les comparaisons entre différentes valeurs de paramètres. Ainsi, il est possible de comparer les différentes méthodes de cohérences entre elles, d'étudier l'impact de la répartition des données sur l'efficacité du partage des données ou de montrer l'influence de la taille des données manipulées sur le partage de celles-ci.

Le temps global permet de rendre compte du temps réel pris par les applications pour

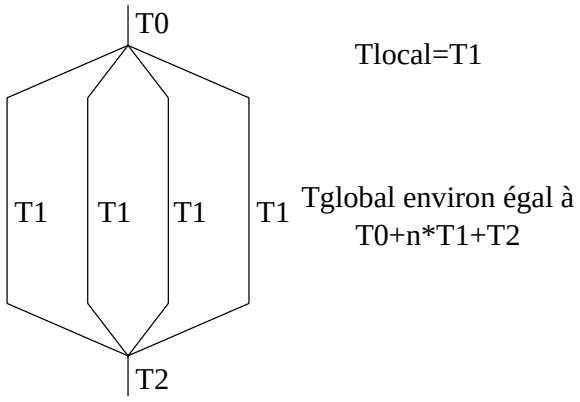


FIG. 3.2 – *Points de mesures du temps.*

être évaluées par notre émulateur. Si on le compare avec le temps local, il est à peu près égal à ce qui est donné par la figure 3.2. A peu près, mais pas tout à fait, car si l'on compare comment est géré le temps local, on s'aperçoit que celui-ci se base sur le temps réel pour tout ce qui touche à l'exécution des «threads», mais que ce temps est modifié par l'émulateur. Ces modifications sont apportées pour tenir compte du réseau et de l'attente sur des synchronisations ou des demandes de pages. Ainsi, cette formule serait juste, si le réseau permettait des temps de service instantanés et si les «threads» n'avaient jamais à s'attendre les uns les autres.

En résumé, le temps local sert à faire des comparaisons entre les diverses situations émulées et le temps global est une indication pour montrer que l'émulateur est efficace.

Les programmes ont été choisis afin d'être les plus variés possible. Le programme LU sert de base à toutes les expérimentations. Il est en effet le premier à avoir été adapté au simulateur. Il manipule une structure à deux dimensions, utilise très peu de synchronisations, mais passe un temps non négligeable à cause de celles-ci à attendre. De plus, il existe en deux versions différentes qui utilisent deux organisations de données différentes.

Un programme de FFT a ensuite été choisi. Il travaille sur des données contenues dans une structure à une dimension, passe très peu de temps dans les synchronisations, qu'il utilise très peu.

3.3 Les résultats.

La plupart des résultats présentés ont été créés en faisant une moyenne de dix temps obtenus, ceci afin de gommer les aspérités dues au non-déterminisme de l'exécution. Cependant, la valeur la plus grande et la valeur la plus petite sont également présentées,

pour se rendre compte des disparités de temps d'exécution.

Les simulations sont présentées paramètre par paramètre et consistent à faire varier le paramètre concerné. Les paramètres peuvent être les paramètres du réseau simulé, la variation de la taille du problème, le nombre de processeurs simulés, la répartition des données en mémoire et le changement de stratégie de maintien de cohérence. Pour toutes ces expériences, une configuration de base est définie, afin de toujours comparer à des résultats connus. Cette configuration de base repose sur l'application LU. Celle-ci manipule une matrice carrée de 128 nombres de côté avec une répartition des données non contiguë. Cette application est émulée sur un système disposant d'un réseau de 100 Mbits/s de débit et de 10 microsecondes de latence. Elle s'exécute avec 4 processeurs et la cohérence utilisée est atomique.

Par la suite en 3.3.6, une série d'expériences fait varier les applications utilisées, selon un certain nombre de configurations.

3.3.1 Le paramètrage du réseau.

Dans cette série de simulations, nous nous intéressons à l'influence des paramètres qui définissent le réseau. Des mesures ont été effectuées dans une configuration irréaliste (un réseau d'interconnexion complet) d'une machines parallèles (100Mbits/s de débit et 10 microsecondes de latence), un réseau local conventionnel (10Mbits/s de débit et une latence de 100 microsecondes), et un réseau à débit plus important comme ATM (155Mbits/s de débit et une latence de 100 microsecondes). La simulation du réseau étant simpliste, il n'est pas question de faire une grande étude pour comparer les différents réseaux simulés. Le but ici, est de montrer que les paramètres réseau que nous rentrons sont bien pris en compte par l'émulation.

TAB. 3.1 – *Évaluation des paramètres du réseau simulé pour le temps local en secondes.*

	Tlocal	**Tlocal min**	**Tlocal max**
machines parallèles	**1,303**	1,215	1,360
réseaux conventionnels	**1,422**	1,337	1,505
réseaux ATM	**1,349**	1,292	1,403

Dans le tableau 3.1, il apparaît que les paramètres que nous définissons pour le réseau ont bien une influence réelle, mais limitée sur le résultat final. En effet, plus les paramètres rentrés définissent un réseau rapide, plus les applications qui l'utilisent peuvent s'exécuter rapidement. La différence n'est pas flagrante car l'application LU ne partage pas un

nombre suffisant de données pour qu'il y ait une grande compétition pour les données. Le tableau 3.2 montre que la formule qui donne le temps global en fonction du temps local de la figure 3.2 est à peu près bien respectée.

TAB. 3.2 – *Évaluation des paramètres du réseau simulé pour le temps global en secondes.*

	Tglobal	**Tglobal min**	**Tglobal max**
machines parallèles	**6,473**	6,213	6,705
réseaux conventionnels	**6,765**	6,455	7,070
réseaux ATM	**6,621**	6,368	6,932

En conclusion, dans les limites de la simulation installée à l'heure actuelle dans l'émulateur, la gestion des paramètres du réseau se comporte bien comme il était attendu qu'elle le fasse. Cela n'empêche pas d'envisager une forme de simulation du réseau plus complexe afin de mieux se rapprocher du comportement réel de celui-ci (en particulier, des problèmes de goulets d'étranglements). Cependant, cette étude sort du cadre étroit de cette thèse.

3.3.2 Les tests sur la variation de la taille du problème.

L'émulateur que nous avons réalisé permet de faire varier la taille des problèmes traités par les applications simulées. Une série d'expérience à ce sujet doit pouvoir nous renseigner sur les limites inférieure et supérieure de cette variabilité, mais aussi sur le comportement de l'émulateur. Il est en effet intéressant de suivre l'évolution des temps d'exécution des applications au fur et à mesure de l'évolution de la taille des données. Le tableau 3.3 répertorie tous les résultats à ce sujet.

TAB. 3.3 – *Présentation des temps locaux obtenus pour l'application LU, pour des tailles de données variables (en secondes).*

	Tlocal	**Tlocal min**	**Tlocal max**	**Tlocal(N)/Tlocal(N/2)**
Matrice de 16 X 16	**0,019**	0,011	0,026	
Matrice de 32 X 32	**0,032**	0,023	0,041	1,68
Matrice de 64 X 64	**0,170**	0,145	0,184	5,31
Matrice de 128 X 128	**1,303**	1,215	1,360	7,66
Matrice de 256 X 256	**6,818**	6,618	7,053	5,23
Matrice de 512 X 512	**58,210**	57,475	58,696	8,54

Bien entendu, la limite minimum du nombre de pages manipulées lors d'une émulation est 1. Elle est définie dans le tableau 3.3 par une matrice 16X16 qui est la plus grande

matrice à pouvoir tenir dans une page dont la taille est 4Ko. La dernière matrice 512X512 représente la plus grande matrice qui puisse être utilisée pour une simulation correcte. En effet, une taille plus grande de matrice entraîne la mise en fonction du mécanisme de pagination. Celui-ci rend impossible une prise de temps correcte. Le «thread» qui a besoin d'une page non-présente en mémoire est suspendu par le noyau, alors que du point de vue de l'émulateur, il reste exécutable. Cette distorsion de vue entre le noyau et l'émulateur entraîne une incohérence dans la prise de mesure du temps. L'émulateur n'est donc pas conçu pour exploiter la pagination. Pour remédier à ce problème, il existe deux solutions, soit augmenter la mémoire disponible sur la machine qui est actuellement de 16 Mo, soit résoudre le problème de prise de temps lors de la pagination. La première solution à l'avantage d'être simple, mais limitée. La seconde est plus avantageuse et permettra de traiter de gros problèmes. Elle demande cependant une intégration de l'émulateur au noyau ce qui n'a pas encore été traité et fait partie des futurs développements possibles.

TAB. 3.4 – *Présentation des résultats pour l'application LU, pour des tailles de données variables (en secondes).*

	Tglobal	**Tglobal min**	**Tglobal max**	**Tglobal(N)/Tglobal(N/2)**
Matrice de 16 X 16	**0,816**	0,626	0,973	
Matrice de 32 X 32	**0,922**	0,819	1,036	1,13
Matrice de 64 X 64	**1,739**	1,670	1,872	1,87
Matrice de 128 X 128	**6,473**	6,213	6,705	3,72
Matrice de 256 X 256	**40,799**	38,585	42,536	6,30
Matrice de 512 X 512	**295,293**	290,939	298,712	7,25

Il reste à étudier le comportement temporel de l'émulateur pour chaque taille utilisée. Pour cela, le tableau 3.4 complète le tableau 3.3 en présentant le temps global de simulation. De ces deux tableaux, il est possible de conclure que le rapport des temps locaux n'est pas significatif car ces temps ne dépendent pas seulement de la taille des données, mais aussi de l'influence de cette taille sur le partage des données. Par contre, le rapport des temps globaux permet de se rendre compte de l'évolution des temps d'émulation. Le rapport théorique est de 4. Mis à part les premières émulations qui sont trop petites et donc rendent un résultat non significatif, le rapport s'établit à plus de 4 et croit légèrement. Ceci est dû au surplus de calculs qu'entraîne l'émulateur vis à vis d'une exécution normale. Ce surplus dépend en partie de la taille des données et donc le léger accroissement est tout à fait normal.

3.3.3 Les tests sur le nombre de processeurs.

Après avoir fait varier la taille des données, revenons à une taille de 128X128, mais avec un nombre de processeurs variables. Les résultats sont présentés dans les tableaux 3.5 et 3.6.

Ce qui nous intéresse ici concerne l'efficacité de l'émulateur, c'est à dire le temps global nécessaire pour émuler N processeurs. Toutefois l'utilisation d'une machine parallèle est d'habitude beaucoup plus concerné par le «speed-up»; c'est à dire le maximum des temps locaux, c'est pourquoi nous présentons les deux.

TAB. 3.5 – *Présentation des résultats pour l'application LU, pour un nombre de processeurs variables (temps local en secondes).*

	Tlocal	**Tlocal min**	**Tlocal max**	**speed-up basé sur 4 processeurs**
nb processeurs = 4	**1,303**	1,215	1,360	
nb processeurs = 9	**1,906**	1,730	2,018	0,684
nb processeurs = 16	**1,727**	1,656	1,773	0,754
nb processeurs = 25	**1,641**	1,554	1,711	0,794

Le tableau 3.5 montre le temps passé par chaque processeur pour exécuter l'application. Nous pouvons voir que le fait de rajouter des processeurs, ne permet pas forcément d'augmenter l'efficacité. Il semble que le dimensionnement de 128X128 soit très bien adapté pour 4 processeurs mais pas pour un nombre plus élevé. Le passage de 4 à 9 processeurs, entraîne un surcroît de communications qui n'est pas compensé par l'exécution plus rapide. Ce surcroît de communications semble atteindre un plafond avec 9 processeurs. La légère diminution des temps d'exécution pour un nombre de processeurs supérieurs est due à la diminution du nombre de calculs à effectuer du fait d'une plus petite matrice allouée à chaque processeur.

TAB. 3.6 – *Présentation des résultats pour l'application LU, pour un nombre de processeurs variables (temps global en secondes).*

	Tglobal	**Tglobal min**	**Tglobal max**	**Tglobal/nb processeurs**
nb processeurs = 4	**6,473**	6,213	6,705	1,618
nb processeurs = 9	**40,327**	39,109	42,039	4,481
nb processeurs = 16	**65,692**	63,593	67,400	4,106
nb processeurs = 25	**99,519**	96,357	101,241	3,981

Le tableau 3.6 présente le temps passé globalement par l'émulateur pour la simulation de l'application avec un nombre variable de processeurs. Il est donc intéressant de suivre

l'évolution de ce temps en fonction du nombre de processeurs utilisés. Le fait d'augmenter le nombre de processeurs augmente aussi le nombre de changements de contexte à effectuer. De ce fait, le temps passé dans l'ordonnanceur à changer de contexte occasionne une augmentation du temps de simulation. Cependant, bien que le temps augmente, le ratio de celui-ci et du nombre de processeurs semble se stabiliser autour de 4, voir même à baisser un peu.

3.3.4 Les tests sur la répartition des données.

Les applications que nous fournit SPLASH, contiennent deux versions de l'application LU. Ces deux versions diffèrent par la manière dont sont réparties les données. Le détail du stockage de celles-ci en mémoire est présenté par la figure 3.3.

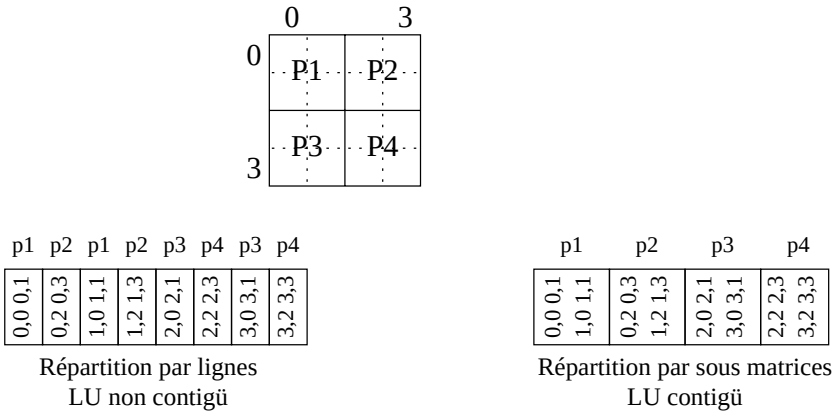


FIG. 3.3 – *Exemple de répartition des données pour un matrice 4X4.*

Pour la première méthode, les données sont rangées en mémoire par lignes. La première ligne est stockée, puis la seconde, etc. Jusqu'à la dernière. Cette façon de faire est classique, mais elle à l'inconvénient de répartir les données par petits blocs. Ainsi, avec cette méthode, il y a de fortes chances que les zones de mémoires soient réparties sur plusieurs processeurs. C'est pourquoi, il semble préférable de répartir la matrice en mémoire non plus par ligne, mais par sous matrice. Ainsi, chaque sous-matrice peut être stockée de manière privilégiée et entièrement auprès du processeur chargé de la traiter. Cette «harmonie» ne sera rompue que par des accès ponctuels des autres processeurs.

Le tableau 3.7 expose brièvement les résultats de la comparaison entre les deux méthodes de répartition des données. Il montre qu'une bonne répartition des données entre les processeurs, peut accroître la rapidité de l'exécution. De plus amples comparaisons sont faites en 3.3.6 où la répartition des données en lignes correspond à l'application LU

TAB. 3.7 – *Présentation des résultats pour l'application LU, pour deux manières de répartir les données en mémoire (en secondes).*

	Tlocal	**Tlocal min**	**Tlocal max**
LU non contigü	**1,303**	1,215	1,360
LU contigü	**1,158**	1,103	1,241

non-contiguë et la répartition des données en sous-matrices correspond à l'application LU contiguë.

3.3.5 Les tests de la cohérence.

Jusqu'à maintenant, tous les tests qui ont été présentés, reposent sur des paramètres de la simulation qui sont très simples à mettre en œuvre. Ces paramètres proviennent pour la plupart des cas des applications fournies. Ces applications intègrent les paramètres concernant la taille des données et le nombre de processeurs utilisés. Nous pouvons considérer que les différentes répartitions possibles des données sont aussi un paramètre des applications quand cette répartition peut varier. Par contre, le paramètrage du réseau et la variation des cohérences utilisées sont réglés par l'émulateur.

TAB. 3.8 – *Tests de la cohérence des données pour l'application LU non-contiguë (en secondes).*

LU non-contiguë	**Tlocal**	**Tlocal min**	**Tlocal max**
Atomique	**1,303**	1,215	1,360
Séquentielle	**1,286**	1,188	1,353
LOCK	**1,505**	1,453	1,592
1ECRXLECT	**1,257**	1,204	1,325
MULTIECRITURE	**1,216**	1,170	1,277

L'émulateur réalisé peut utiliser 5 sortes différentes de gestion de la cohérence. Elles ont été présentées en 3.1. Les résultats des tests effectués pour l'application LU non-contiguë sont présentés dans le tableau 3.8. Comme il est attendu, nous pouvons remarquer qu'excepté l'adaptation LOCK, tous les affaiblissements de la cohérence, sont accompagnés d'une amélioration des performances. Ainsi, nous passons d'un temps de 1,303 secondes pour une cohérence atomique à 1,216 secondes pour une cohérence de MULTIECRITURE. Ce résultat est tout à fait logique car l'affaiblissement de la cohérence s'accompagne d'une diminution des requêtes du paginateur au noyau. Ces requêtes avaient pour objet de diffuser des invalidations totales ou partielles (invalidation en écriture seulement) à l'intention

d'autres processeurs. Par contre, l'idée de la cohérence LOCK à savoir de poser des verrous sur les données que l'on a besoin et ne les relâcher que lorsque l'on est sûr de ne plus en avoir besoin, semble ne pas convenir à cette application. Elle peut peut-être s'adapter de façon plus adéquate à d'autres applications. C'est ce que nous allons vérifier en examinant d'autres applications, toutes tirées de «SPLASH».

3.3.6 Les tests sur certaines applications SPLASH.

TAB. 3.9 – *Tests de la cohérence des données pour l'application LU contiguë (en secondes).*

LU contiguë	**Tlocal**	**Tlocal min**	**Tlocal max**
Atomique	**1,158**	1,103	1,241
Séquentielle	**1,212**	1,159	1,282
LOCK	**1,185**	1,140	1,234
1ECRXLECT	**1,262**	1,173	1,350
MULTIECRITURE	**1,175**	1,124	1,253

Dans le tableau 3.9 les cohérences ne suivent pas la même hiérarchie que pour l'application LU non-contiguë. Ici, l'application LOCK semble avoir un comportement tout aussi correct que les autres. Par contre, des aberrations apparaissent quant au classement des autres cohérences. Ceci est dû au fait que la répartition des données de cette application entraîne très peu de partage de pages par les applications. Ainsi, il est difficile de faire la différence entre toutes les sortes de maintien de la cohérence. Les différences ne sont explicables que par des aléas d'exécutions.

TAB. 3.10 – *Tests de la cohérence des données pour l'application FFT (en secondes).*

FFT	**Tlocal**	**Tlocal min**	**Tlocal max**
Atomique	**0,396**	0,378	0,407
Séquentielle	**0,391**	0,377	0,410
LOCK	**0,391**	0,371	0,417
1ECRXLECT	**0,365**	0,355	0,379
MULTIECRITURE	**0,370**	0,344	0,392

En ce qui concerne l'application FFT, elle ne permet pas une véritable comparaison des cohérences utilisées. Elle aussi, donne des résultats trop proches pour être comparables.

3.4 Interprétation des résultats.

Les résultats qui ont été donnés en 3.3 ne sont qu'une ébauche de ce que permet le simulateur. Ils ont été présentés pour montrer l'étendue des possibilités et les limites de celui-ci. Ainsi, dans l'état actuel de l'implantation de l'émulateur, il faut dans le dimensionnement du problème tenir compte de la formule suivante:

$$Memoiretotaleutilisee \simeq NBprocs * Memoirepourunprocesseur$$

Pour l'application LU, si on utilise 4 processeurs, on peut avoir une matrice de 512X512. Par contre, avec 64 processeurs, la matrice maximum que l'on peut traiter est de 128X128.

Pour ce qui est de la cohérence et de la répartition des données, malgré le petit nombre de résultats, il est possible tout de même de montrer l'importance de bien organiser les données en mémoire et de mettre en évidence que le choix d'une politique de cohérence plutôt qu'une autre influe sur les performances de l'application. Cette influence est plus ou moins visible, selon les besoins de partage en données des applications.

Ces premiers résultats sont encourageants et démontrent que l'émulateur réalisé fournit bien les réponses attendues lors de ces expériences. Cependant, un bon nombre d'améliorations peuvent être envisagées, celles-ci sont abordées dans la conclusion.

Conclusion.

L'objectif de cette thèse était d'étudier et de réaliser un émulateur de systèmes parallèles utilisant une mémoire virtuelle partagée distribuée. Nous avons donc étudié dans un premier temps tout ce qui avait un rapport avec la gestion d'une telle mémoire et dans un second temps leur adaptation pour leur intégration dans l'émulateur.

L'étude préalable à permis de dégager les points qui pouvaient être repris de la gestion des caches et ceux qui ne le pouvaient pas. Ainsi, toutes les améliorations qui tentent de réduire le trafic sur les réseaux d'interconnexions ont été étudiées avec intérêt car le grain d'échange d'une SVM est une page entière. Il est donc facile d'obtenir un goulot d'étranglement sur le réseau si les pages circulent en trop grand nombre. Une de ces améliorations, nous a particulièrement intéressé. Il s'agit des méthodes d'affaiblissement de la cohérence qui en s'appuyant sur les primitives de synchronisations, permettent de regrouper ces deux fonctionnalités en une seule, et donc de réduire le nombre de pages circulant sur le réseau.

En ce qui concerne l'émulateur construit, il fallait prouver le bien fondé des choix faits. Pour cela, nous avons réalisé une série d'expériences qui nous a permis de démontrer l'efficacité de ce type d'émulation et aussi d'en mettre en évidence les limites. L'efficacité (au moins en termes de temps d'exécution) est tout à fait satisfaisante; on émule N processeurs sur 1 processeur réel en multipliant la durée d'exécution par environ C x N, C étant une constante qui dépend de la taille du problème traité (Par exemple, pour une matrice de 128X128, C est à peu près égal à 4).

Les limites peuvent se décomposer en deux catégories: celles imposées par la machine utilisée (Un PC486 DX33 doté de 16 Moctets de mémoire) et celles inhérentes à l'émulateur lui-même.

Le nombre de processeurs et la taille du problème sont deux paramètres qui appartiennent à la première catégorie. L'émulateur n'est pas prévu pour s'exécuter avec la pagination. Il faut donc que toutes ces données puissent prendre place dans la mémoire physique. Comme il faut un exemplaire de la mémoire partagée par processeur pour pouvoir simuler la mémoire partagée distribuée, plus l'on simule de processeurs, moins la

taille des données manipulables est grande et inversement. Cette limitation dépend donc de la mémoire réelle disponible sur la machine. Cependant, il sera possible de passer outre cette limite en intégrant une gestion correcte de la pagination par l'émulateur, tout particulièrement en ce qui concerne le temps passé à attendre une page qui vient du disque.

Dans la seconde catégorie des limites, on pourrait mettre la limitation de l'émulateur en ce qui concerne la façon dont il simule le réseau d'interconnexion. Ce réseau est pour le moment traité comme un réseau point à point avec un débit et une latence paramètrables. Toutefois, comme la simulation du réseau n'était pas notre but primaire, nous nous sommes contentés de cette simulation simpliste. Il est bien évident que celle-ci n'est pas envisageable pour un nombre important de processeurs. Cependant, l'intérêt de l'émulateur réalisé est bien de permettre l'évaluation de l'exécution des programmes parallèles sur des machines disposants de réseaux plus réalistes, et d'étudier l'efficacité de ces derniers. La réalisation de la simulation de ces réseaux est donc du ressort de l'expérimentateur qui se servira de l'émulateur comme support d'exécution.

Une autre limite intrinsèque à l'émulateur est liée au principe même de l'émulateur: le «scénario» joué par l'émulateur n'est pas forcément exactement identique à celui de la machine parallèle; en effet l'exécution sur celle-ci de processus asynchrones peut se passer de façon variable, et l'émulateur choisit un seul ordonnancement parmi ceux possibles. Nous avons montré que lorsque les programmes utilisent des synchronisations pour accéder les variables partagées, il ne subsiste aucun indéterminisme, et le comportement de l'émulateur est exactement celui de la machine parallèle. Mais, si ce n'est pas le cas, l'émulateur ne reproduit qu'une exécution possible du système parallèle, ce qui n'est pas gênant en ce qui concerne les résultats de l'exécution.

Pour contourner ces limitations, deux perspectives proches peuvent être dégagées et une plus lointaine.

La première amélioration possible est d'adapter le matériel utilisé. Avec une mémoire plus conséquente, il serait possible de traiter des problèmes plus importants et donc de fournir des expérimentations plus intéressantes. Cette solution est bien sûr simple à mettre en œuvre mais aussi coûteuse et limitée.

La seconde amélioration possible peut permettre d'augmenter la taille des données traitées d'une manière significative en intégrant le mécanisme de pagination à l'émulateur. Cette solution n'implique pas un changement de matériel, mais oblige à intervenir dans le micro-noyau Mach lui-même. Cette intervention dans le micro-noyau est gênante pour la «philosophie» Mach qui tend à essayer d'extraire au maximum les fonctionnalités du noyau. Il est donc paradoxal, mais indispensable pour traiter de gros problèmes, d'étendre l'émulateur aux couches de pagination du noyau.

Dans une perspective plus lointaine, il est possible d'envisager une évolution de l'émulateur en un véritable système de SVM. Pour cela, il faut faire exécuter un exemplaire de l'émulateur par site. Le plus gros problème à résoudre dans cette optique est la communication entre les différents émulateurs. Les «threads» clients continuerons à s'exécuter et à faire des requêtes au paginateur. L'ordonnanceur lui aussi continuera à s'exécuter comme il le fait maintenant. Les plus gros changements devront donc intervenir au niveau des paginateurs. Il faudra une bonne communication entre eux pour pouvoir gérer des requêtes qui viennent de sites distants. Une fois ceci réalisé, il sera possible d'avoir un système de SVM qui s'exécute sur N processeurs et qui a la possibilité de reproduire le comportement d'un système s'exécutant sur M processeurs, avec bien sûr $M > N$.

Annexe A

Les méthodes de maintien de la cohérence des données.

Cette partie expose les différentes méthodes et systèmes utilisés pour représenter les informations nécessaires au maintien de la cohérence. On en trouve un éventail dans [Rai92], [Ste90] et [CFKA90]. La représentation de ces informations doit permettre une évolutivité du système. Si celui-ci change d'ordre de grandeur, il faut que les données à gérer ne prennent pas des proportions gigantesques. De plus, il faut essayer de ne pas utiliser de diffusions, car avec un grand nombre de processeurs, une diffusion entraîne une hausse du trafic réseau et donc un encombrement plus ou moins rapide de celui-ci. C'est donc à tous ces problèmes que se sont attaquées les solutions qui suivent.

A.1 L'espionnage.

Cette méthode a été utilisée sur les premiers multi-processeurs à bus commun. Elle consiste à espionner tout ce qui passe sur le bus. De ce fait, chaque noeud connaît ce que fait les autres et quelles sont les données échangées sur le bus. Comme, il connaît aussi le contenu de son propre cache, il a en main toutes les informations pour gérer la cohérence des données qui se trouvent dans son cache. L'accès au répertoire des caches peut se faire par le processeur ou par le contrôleur de cohérence. Il peut y avoir conflit d'accès entre les deux. Ceci est résolu par une duplication du répertoire des caches [AB86]. Le fait que l'espionnage se soit surtout développé sur des architectures à base de bus provient du fait qu'elle à besoin d'un moyen peu coûteux et rapide de diffusion atomique. De ce point de vue, le bus est l'outil idéal. Il n'est donc pas exclu de pouvoir implanter cette méthode sur d'autres architectures de multi-processeurs, dans la mesure ou le coût d'une diffusion

n'est pas prohibitif.

A.1.1 L'espionnage sur un seul bus.

C'est la forme la plus simple de l'espionnage de bus. Le multi-processeur type qui se prête bien à ce type de maintien de la cohérence est présenté figure A.1.

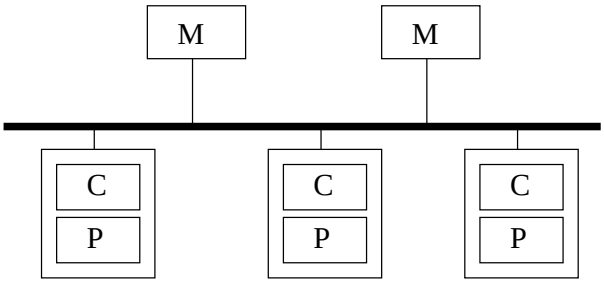


FIG. A.1 – *Multi-processeur à bus commun. C : cache, P : Processeur, M : Mémoire*

Tous les caches, ont accès à tous les échanges qui se déroulent sur le bus. De ce fait, ils espionnent tout ce qui passe sur le bus qui puisse les intéresser, c'est à dire, tous les échanges qui concernent les données présentes chez eux, même si l'échange ne leur est pas destiné. Les différents protocoles de maintien de la cohérence diffèrent à quatre niveaux, le nombre et la signification des états possibles d'une donnée contenue dans un cache, la méthode utilisée en cas d'écriture sur une donnée partagée, la façon de mettre à jour la mémoire principale et la manière de charger un cache avec une donnée déjà partagée.
En ce qui concerne le dernier point, il existe deux solutions, soit l'on charge toujours depuis la mémoire centrale, soit l'on peut charger depuis un cache qui possède déjà la donnée. La seconde solution est souvent la plus rapide car les mémoires servant à faire la mémoire cache sont plus performantes que celles de la mémoire principale.
Pour ce qui est de la manière de mettre à jour la mémoire principale, il existe trois solutions déjà présentées en 1.2.1. L'écriture différée et l'écriture paresseuse permettent une meilleure utilisation de la bande passante du bus.
Lorsqu'un processeur veut écrire dans un bloc qui est partagé entre plusieurs processeurs, deux méthodes sont possibles. Pour la première, il y a diffusion d'une invalidation pour le bloc concerné. Les autres caches qui possèdent la donnée et qui espionnent le bus invalident le bloc contenant la donnée visée lors du passage de l'invalidation sur le bus. Après cela, le processeur peut changer l'état du bloc contenant la donnée et écrire celle-ci. Cette solution permet l'existence à un instant donné de, soit un écrivain, soit un ou plusieurs lecteurs. L'autre solution, autorise plusieurs écrivains et plusieurs lecteurs à cohabiter en

même temps. Il s'agit ici, de diffuser les écritures au lieu d'invalider les autres caches. Un processeur qui veut écrire dans un bloc partagé, le fait et diffuse l'écriture sur le bus. Tous les autres caches qui possédaient ce bloc, mettent à jour la donnée. Toutes ces opérations ce déroulent de manière atomique vue la structure du bus.
Le dernier point à aborder ici, est le nombre et la signification des états des données dans le cache et la mémoire principale. Comme ces états sont intimement liés au protocole utilisé, ils seront présentés par la suite dans la présentation de chaque protocole.

La présentation des protocoles détaillera le premier, mais pour les autres, elle ne fera que souligner les différences.

A.1.1.1 Le protocole "Write-once".

Le premier protocole étudié a été présenté dans [Goo83] et amélioré par [AB86]. Il s'agit d'un protocole avec invalidation. Il est composé de 4 états différents qui sont détaillés ci-après.
- Invalide: La donnée présente dans le cache n'est pas à jour.
- Valide: Le cache contient une copie valide et cohérente avec la mémoire principale. Il peut ne pas être le seul dans cet état.
- Réservé: La donnée a été écrite une seule fois et elle est cohérente avec la mémoire principale qui est la seule hormis le cache concerné à détenir une copie de la donnée écrite.
- Modifié: La donnée a été écrite plus d'une fois. Elle est dans le cache et uniquement dans celui-ci pour tout le système.

Toutes les interactions de ce protocole sont résumées par la figure A.2 [Ste90]. Elles sont expliquées ci-après.

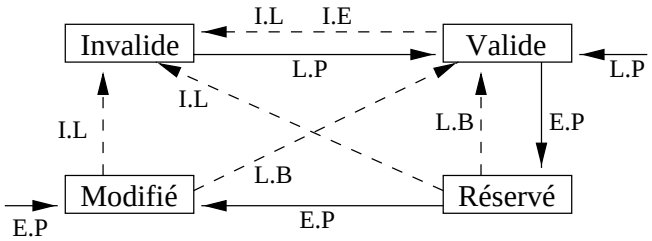


FIG. A.2 – *Interactions entre les différents états.*

Sur un échec en lecture, s'il n'existe pas de copie modifiée, la donnée est chargée depuis la mémoire centrale et l'état de cette copie devient valide. Si une copie existe dans l'état réservé, le cache la contenant aperçoit la requête en lecture à la mémoire et de ce fait, fait passer la copie de la donnée en valide. Si une copie dans l'état modifié existe, le cache contenant la copie fourni la donnée à la place de la mémoire centrale et passe à l'état valide. Au passage, la mémoire principale est mise à jour.
Pour une écriture réussie, si l'état est modifié ou réservé, l'écriture se déroule localement et l'état final est modifié. Si l'état est valide, une requête d'invalidation d'écriture est envoyée qui permet aux autres caches d'invalider leur copie et de mettre à jour la mémoire principale. De cette façon, l'état résultant est réservé.
Sur un échec en écriture, la donnée provient soit d'un cache qui était dans l'état modifié, dans ce cas, la mémoire principale est mise à jour au passage, soit directement de la mémoire centrale. Un échec en écriture est traité par l'envoi d'une invalidation de lecture ce qui initie le processus de rapatriement de la donnée, ainsi que l'invalidation des copies existantes si nécessaire. La copie nouvellement chargée dans le cache se retrouve dans l'état modifié.
En cas de remplacement d'un bloc, celui-ci n'est réécrit dans la mémoire centrale que s'il se trouve dans l'état modifié.

A.1.1.2 Le protocole Synapse.

Dans ce protocole (proposé dans [Fra84]), il ne reste que trois états (Invalide, Valide et modifié). Chaque bloc a un propriétaire. C'est le cache qui contient le bloc dans un état modifié, ou c'est la mémoire centrale dans tous les autres cas. C'est toujours le propriétaire d'un bloc qui répond aux requêtes concernant ce bloc.
En cas d'échec dans un accès mémoire, c'est toujours la mémoire principale qui fourni le bloc. Si ce n'est pas elle qui est le propriétaire, il faut qu'elle le devienne.
Ainsi, sur un échec, si un cache possède le bloc dans l'état modifié, il le recopie en mémoire centrale, en fournissant à celle-ci la propriété du bloc et positionne son état à invalide pour une écriture ou valide pour une lecture. Tous les autres caches dans un état valide passent à l'état invalide et le cache demandeur est obligé de faire une autre demande pour recevoir la donnée de la mémoire principale.
Pour une écriture, si l'état est valide, le cache doit attendre que la mémoire centrale lui ait transféré la propriété sur ce bloc. Ce qui revient à faire un transfert complet de la donnée.

A.1.1.3 Le protocole Berkeley.

Ce mécanisme est proposé dans [KEW+85]. Deux différences le distinguent du système précédent. La copie directe de cache à cache se fait dès qu'elle est possible et la mémoire n'est pas mise à jour lors du partage d'un bloc auparavant dans l'état modifié. Ceci requiert un nouvel état modifié-partagé, qui désigne un bloc qui est partagé, mais qui n'est pas cohérent avec la mémoire centrale et qui devra donc en cas de remplacement être réécrit. La notion de propriétaire est aussi développée ici, un cache est propriétaire d'un bloc s'il se trouve dans l'état modifié ou modifié-partagé.
Pour un échec en lecture, le cache est approvisionné par le propriétaire du bloc qui passe ensuite en modifié-partagé si c'était un cache. Le cache demandeur passe dans tous les cas dans l'état valide.
Pour une écriture dans un bloc présent, il n'y a aucun problème si le cache est dans l'état modifié. Si l'état est valide ou modifié-partagé, il faut invalider les éventuelles autres copies et passer son état à modifié.
Pour un échec en écriture, le bloc provient du propriétaire, toutes les autres copies sont invalidées et l'état du cache demandeur devient modifié.

A.1.1.4 Le protocole Illinois.

Ce système est présenté dans [PP84]. Il comporte un état valide-exclusif qui se rajoute aux états invalide, partagé et modifié. Cela permet de ne pas faire des invalidations inutiles. Ce protocole ne fait appel à la mémoire que si aucun cache ne peut fournir le bloc demandé.
En cas d'échec en lecture, la seule différence avec les protocoles précédents, c'est que l'état valide-exclusif est positionné si aucun autre cache ne fournit la copie. Le cache demandeur doit être capable de savoir d'où lui provient le bloc (mémoire centrale ou autre cache).
En d'écriture sur un bloc présent dans l'état valide-exclusif, l'écriture peut se faire immédiatement et l'état passe en modifié.
En cas d'échec en écriture, le bloc provient d'un cache si possible, les autres copies sont invalidées et l'état devient modifié.

A.1.1.5 Le protocole Firefly.

Dans ce protocole (proposé par Thacker et Stewart [TS87]), comme dans le suivant, il n'y a plus d'invalidation. Les états sont donc ici valide-exclusif (unique copie cachée et pas modifiée), partagé (pas modifiée, mais peut être présente dans d'autres caches) et modifié (unique copie et modifiée). Il n'y a plus d'état valide car il n'y a plus d'invalidation. Une

ligne supplémentaire sur le bus est rajoutée afin de permettre au cache demandeur de savoir si le bloc provient d'un autre cache ou de la mémoire centrale.
En cas d'échec en lecture, si un cache peut fournir le bloc, il le fait en positionnant la ligne partagée et tous les caches passent à l'état partagé. Sinon, la mémoire principale donne le bloc sans positionner la ligne partagée, ce qui permet au site demandeur de faire la différence avec le cas précédent et de passer en valide-exclusif.
Sur un échec en écriture, le cache peut recevoir le bloc de la mémoire principale ou d'un autre cache. Dans le premier cas, le cache prend le bloc et passe à l'état modifié. Dans le second, il passe à l'état partagé et son écriture doit apparaître sur le bus pour mettre à jour les autres copies.
Si l'écriture survient sur un bloc déjà présent, si l'état est modifié ou valide-exclusif, alors l'écriture se déroule en local et l'état passe à modifié. Par contre, si l'état est partagé, l'écriture passe par le bus. Les caches qui possèdent une copie la mettent à jour et positionnent la ligne partagée. En l'absence de positionnement de celle-ci, le cache demandeur peut passer en valide-exclusif et en modifié à la prochaine écriture car il n'existe plus d'autres copies.

A.1.1.6 Le protocole Dragon.

Ce mécanisme ressemble beaucoup au précédent. Il est présenté dans [Cre84]. Il a deux états partagé-non-modifié et partagé-modifié qui remplacent l'unique état partagé du cas précédent. Quand il y a partage, le dernier cache qui a écrit dans le bloc à l'état partagé-modifié et les autres l'état partagé-non-modifié. Cette différenciation permet de ne mettre à jour la mémoire que lors de l'éviction d'un bloc dont l'état est modifié.

A.1.1.7 Amélioration possible des protocoles à invalidations.

Le grand défaut des protocoles à invalidation, est que les blocs invalidés lors d'une écriture sont très souvent réutilisés par la suite. Ceci peut provoquer pour N copies présentes dans les caches, N-1 échecs d'accès. Rudolph and Segall [RS84] ont imaginé la lecture diffusée. Lors d'un accès en écriture, le cache concerné invalide les autres copies, comme dans les systèmes précédents. Mais lors du premier échec en lecture entraîné par cette invalidation, le bloc est chargé dans tous les caches où il se trouve dans l'état invalide. Donc, dans le meilleur des cas, on passe de N-1 échecs en lecture à un seul.

A.1.2 L'espionnage sur plusieurs bus.

A.1.2.1 Architecture hiérarchique des caches et bus.

Wilson a présenté dans [Jr.87] une méthode reposant sur l'architecture décrite dans la figure A.3.

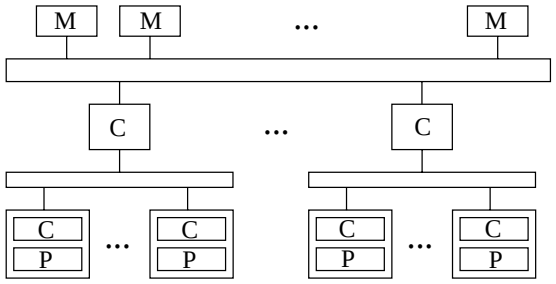


FIG. A.3 – *Hiérarchie de bus et de caches.*

Pour les caches qui sont sur un même niveau, le maintien de la cohérence des copies se déroule comme à l'accoutumé. Ce mécanisme, ne fait appel aux niveaux supérieurs que si c'est nécessaire. C'est à dire si la requête ne peut être résolue à ce niveau. Les caches intermédiaires font office de filtre, ce qui empêche tous les caches sous ce filtre de recevoir des requêtes qui ne les concernent pas.

A.1.2.2 Le multicube du Wisconsin.

Ce système repose sur une grille à deux dimensions de bus. Il a été proposé par Goodman et Woest dans [GW88]. Aux intersections des bus, se trouve un noeud qui contient un processeur, un cache et un contrôleur de cache relié aux deux bus. Un module de mémoire est associé à chaque colonne (figure A.4).

Un bloc peut avoir deux états. Le premier est l'état modifié un seul cache contient le bloc et la mémoire centrale est incohérente. Le second est non-modifié, plusieurs caches peuvent contenir la donnée et la mémoire est cohérente. Une requête est toujours véhiculée par une ligne en premier. Un cache se charge alors de la propager sur une colonne si besoin est. Ce cache appartient à la colonne de la mémoire qui contient la donnée si celle-ci n'est pas modifiée ou à la colonne du cache qui contient la donnée si celle-ci est modifiée. La réponse est véhiculée par une colonne, puis une ligne pour une requête de lecture, mais par une ligne, puis une colonne dans le cas d'une écriture. Cela permet de mettre à jour l'information que le bloc modifié se trouve dans un nouveau cache à toute colonne contenant ce cache.

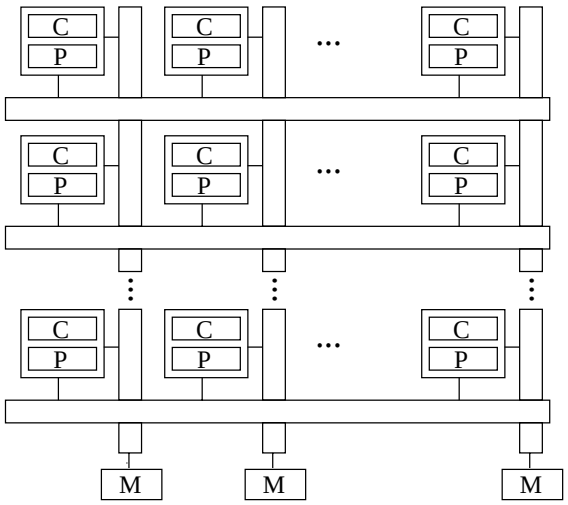


FIG. A.4 – *Le multicube de Wisconsin.*

A.1.2.3 La méthode DDM.

La méthode de la machine DDM (Data Diffusion Machine) est exposée dans [HLH92]. Elle ressemble beaucoup aux bus hiérarchiques de Wilson. Cependant, les noeuds intermédiaires ne contiennent plus les données elles mêmes, mais les informations les concernants. Ceci permet de réduire le surplus de mémoire nécessaire, mais rend l'accès aux données plus soumis à des va-et-vient dans la hiérarchie que la méthode de Wilson. Une autre différence existe, c'est le fait que les mémoires mémoire jouent le rôle de caches. Ceci a déjà été énoncé dans 2.1.5.

A.2 Les répertoires.

Les répertoires ont été introduits pour pallier les inconvénients de la méthode de l'espionnage, c'est à dire le fait d'avoir besoin d'un moyen de diffusion efficace. Cependant, leur défaut est de rajouter des bits supplémentaires pour la gestion des répertoires. Ce surplus peut être supportable pour des petites machines, mais devenir trop volumineux pour des machines avec un grand nombre de processeurs. Il faut donc pour chaque méthode proposée, faire attention au facteur d'échelle. La plupart des méthodes sont très proches dans le principe général. Elles diffèrent sur le type d'information stockée dans les répertoires, dans la manière de répartir cette information entre les caches et la mémoire centrale et sur la possibilité de faire des diffusions ou dans la négative, sur le nombre limite de copies autorisées, si limite il y a. Dans la présentation qui suit, les méthodes de maintien de cohérence par répertoires ont été divisées en quatre parties, les répertoires

complets, les répertoires limités, les répertoires chaînés et les répertoires spéciaux.

A.2.1 Les répertoires complets.

La première méthode de maintien de la cohérence a été proposée dans [Tan76]. Chaque cache doit stocker en plus de l'étiquette de chaque donnée, un bit appelé modifié qui sert à déterminer si la donnée correspondante a été écrite ou non. Parallèlement, dans la mémoire centrale, il existe un répertoire regroupant toutes les étiquettes et tous les bits modifiés de tous les caches. Ces répertoires sont illustrés par la figure A.5.

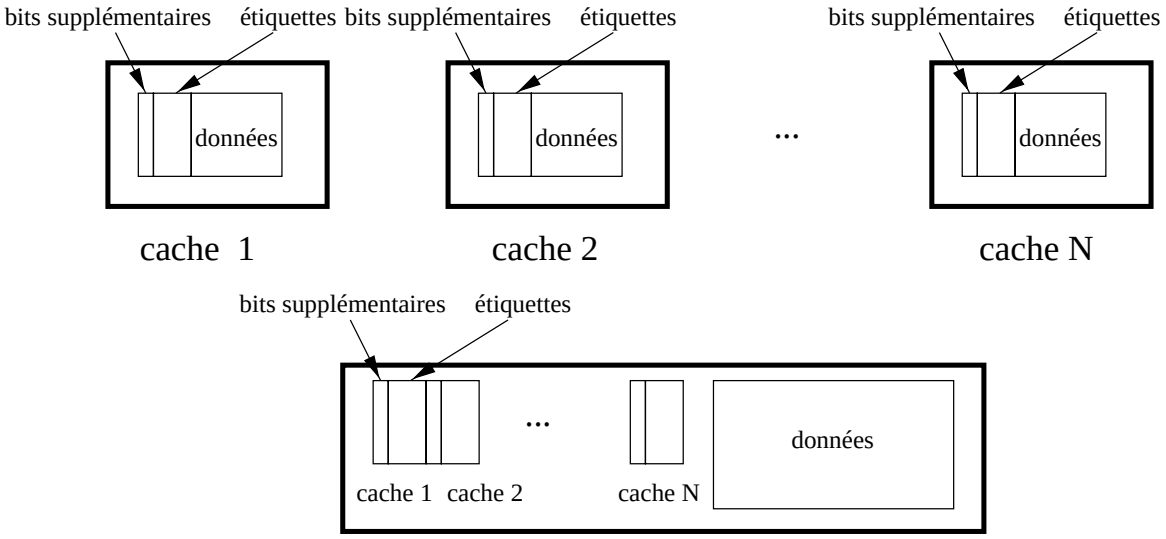


FIG. A.5 – *Les répertoires du système de Tang.*

Lors d'un échec en lecture dans un cache, il faut interroger le répertoire principal, pour savoir si un autre cache contient la donnée et l'a modifiée. Si c'est le cas, le cache en question met à jour le mémoire principale et positionne son bit modifié à zéro; la mémoire centrale peut alors délivrer la donnée au cache demandeur. Si ce n'est pas le cas, la donnée est simplement chargée dans le cache demandeur et le répertoire principal est mis à jour comme il se doit.
L'échec en écriture est traité d'une manière proche de celle en lecture. Si un cache détient la donnée en écriture, celle-ci est réécrite en mémoire principale et le cache perd la donnée. Si un ou plusieurs caches détiennent la donnée en lecture, celle-ci est invalidée. Après les opérations ci-dessus, ou tout simplement pour une donnée qui n'était pas déjà dans un cache, la donnée est chargée dans le cache et les répertoires sont mis à jour de manière à

spécifier que la donnée contenue dans le cache demandeur est modifiée.
Lors d'un accès au cache en écriture, si la donnée est déjà présente dans le cache, il faut s'assurer que le bit modifié est positionné à un. Si c'est le cas, l'écriture peut se dérouler localement. Si ce n'est pas le cas, elle doit dialoguer avec le répertoire principal avant de positionner le bit modifié correctement. Le répertoire principal, ainsi averti, peut invalider les copies qui peuvent éventuellement exister dans d'autres caches.

Le mécanisme décrit ci-dessus à deux inconvénients. Le rassemblement de tous les caches en mémoire principale peut entraîner la création d'un goulot d'étranglement. Et, si l'on veut savoir si un cache contient déjà une donnée, il faut parcourir tous les caches. C'est pourquoi, Censier et Feautrier [CF78] ont imaginé une amélioration à ce système. L'information contenue dans les caches reste la même, ainsi que le déroulement des lectures et des écritures. Cependant, l'organisation de l'information au sein de la mémoire principale est modifiée.

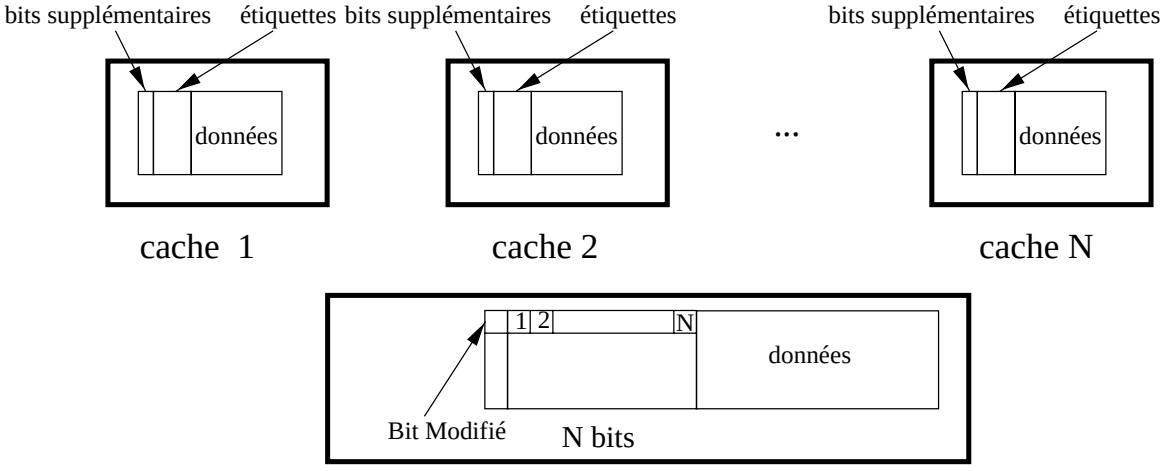


FIG. A.6 – *Les répertoires du système de Censier et de Feautrier.*

Pour chaque bloc de donnée, le répertoire contient un bit pour chaque cache, de manière à savoir si la donnée est présente dans ce cache ou non. En plus, (comme avec le système de Tang) un bit est maintenu pour savoir si la donnée a changé ou non. Cette nouvelle organisation du répertoire principal est présentée dans la figure A.6. Elle a l'avantage d'avoir un accès direct à l'information à partir de l'adresse de la donnée concernée. De plus, la place occupée en mémoire centrale est plus faible qu'avec le mécanisme de Tang.

Yen et Fu, ont présenté une variante du modèle de Censier et Feautrier [YYF85]. L'idée

est ici, de rajouter un bit dans le répertoire des caches pour savoir si le cache est le seul à posséder cette donnée ou non (voir figure A.7). Ce bit permet d'éviter de faire un accès à la mémoire principale, lors d'une écriture, alors que le bit modifié n'est pas positionné. Cependant, pour réaliser cette économie, il faut gérer correctement ce nouveau bit, ce qui entraîne un surplus de l'utilisation du réseau d'interconnexion.

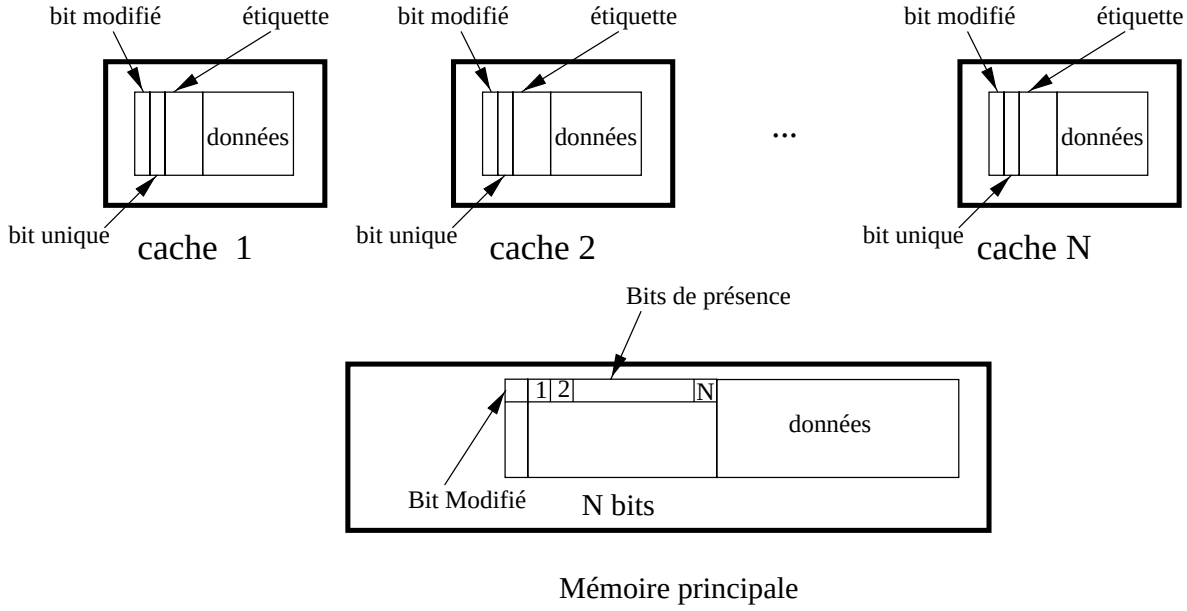


FIG. A.7 – *Les répertoires du système de Yen et de Fu.*

Les méthodes proposées jusqu'à maintenant n'avaient besoin d'aucune diffusion pour fonctionner. Celle présentée par Archibald et Baer nécessite un recours à la diffusion pour retrouver les caches qui possèdent une copie d'un bloc donné [AB84]. Cet inconvénient est compensé par le fait que ce mécanisme n'utilise dans la mémoire principale que deux bits pour chaque cache A.8.

Ces deux bits sont utilisés pour coder quatre états possibles d'un bloc de la mémoire principale. Soit il n'est pas caché, soit il est présent dans exactement un cache et il n'est pas modifié, soit il est présent dans plusieurs cache, mais il n'est modifié dans aucun, enfin, soit il est présent dans un seul cache, mais il est modifié. Le fait d'avoir seulement 2 bits dans le répertoire principal par cache permet une économie de place, mais aussi une bonne extensibilité. Cependant, le recours obligatoire dans certain cas à la diffusion est le grand défaut de cette méthode. C'est pour cela que le cas où la donnée n'est présente que dans un cache et n'est pas modifiée existe, car cela permet de réduire le recours à la diffusion.

Les méthodes de répertoire centralisé ont au moins deux défauts parmi les trois qui

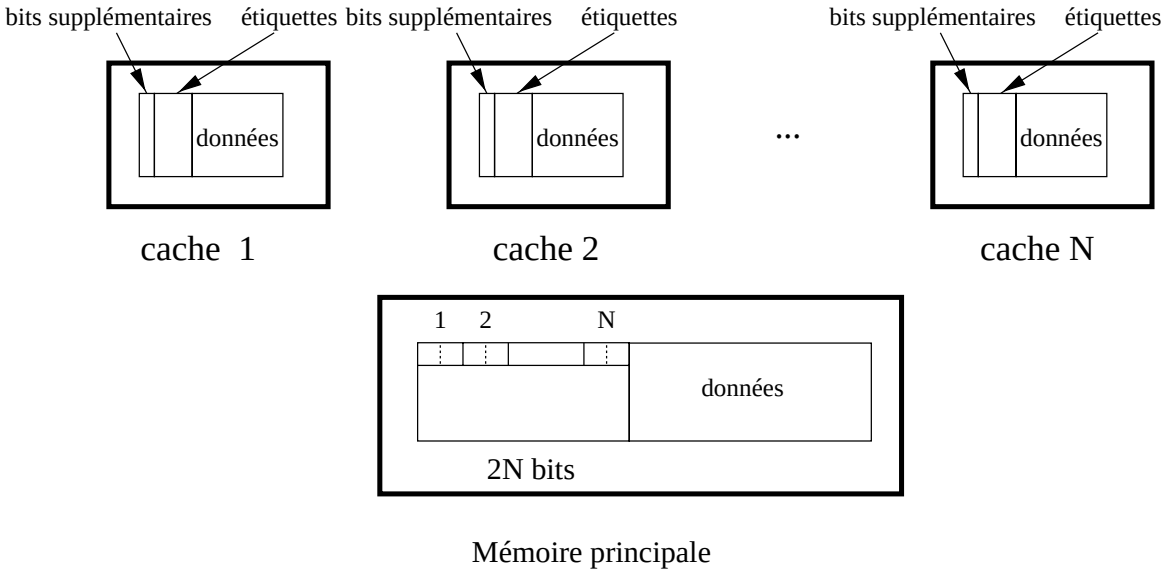


FIG. A.8 – *Les répertoires du système de Archibald et Baer.*

suivent. Premièrement, ils prennent de la place en mémoire centrale et cette place est proportionnelle au produit de la taille de la mémoire par le nombre de caches utilisés. Deuxièmement, le fait de faire référence à un répertoire central peut favoriser l'émergence d'un goulot d'étranglement, surtout avec un nombre important de caches. Troisièmement, il peut être indispensable d'effectuer des diffusions pour certaines méthodes. Pour toutes ces raisons, les mécanismes de maintien de cohérences de caches ne sont pas extensibles. C'est pourquoi d'autres systèmes ont été proposés.

A.2.2 Les répertoires limités.

Une solution assez drastique à ce problème d'extensibilité des répertoires de maintien de la cohérence de caches est de limiter le nombre d'entrées pour les répertoires [ASHH88]. Tout se passe donc comme si dans le mécanisme de Censier et Feautrier (voir figure A.6) le nombre N n'était plus égal au nombre de caches, mais à un nombre inférieur qui n'est pas fonction du nombre de caches. En faisant de cette manière, le nombre de caches n'intervient plus dans la taille des répertoires, ce qui permet de garantir une bonne extensibilité. Cependant, il se pose un problème quand on a besoin de plus de copies que les répertoires peuvent en contenir. Il existe deux approches pour résoudre ce problème. Premièrement, le nombre de copies est fixé strictement et dans aucun cas ne peut être dépassé. Dans ce cas, si un nouveau cache veut une copie alors que les répertoires sont déjà pleins, il faut invalider une copie dans un cache déjà approvisionné. Ce choix peut être

fait en s'inspirant des politiques de remplacement dans les caches. L'invalidation libérera une place dans les répertoires. Ce qui permettra au cache demandeur de pouvoir recevoir une copie. À toutes les étapes de ce dispositif, il n'y a jamais plus de copies que fixé dans les répertoires.
Deuxièmement, le protocole prévoit la possibilité d'avoir plus de copies que le nombre de places disponibles dans les répertoires. Dans ces conditions, le mode de fonctionnement est double, soit le nombre de copies demandé permet d'utiliser les répertoires et tout ce déroule dans ceux-ci, soit ce nombre est trop élevé et l'on fournit quand même le cache demandeur sans enlever une copie à un autre cache, il faut alors avoir recours à la diffusion.

A.2.3 Les répertoires chaînés.

Si l'on refuse d'avoir des répertoires non extensibles, que l'on ne veut aucune diffusion et ne pas limiter le nombre de copie, il existe la solution des répertoires chaînés. Cette solution a été présentée dans [JLGS90] et est illustrée par la figure A.9.

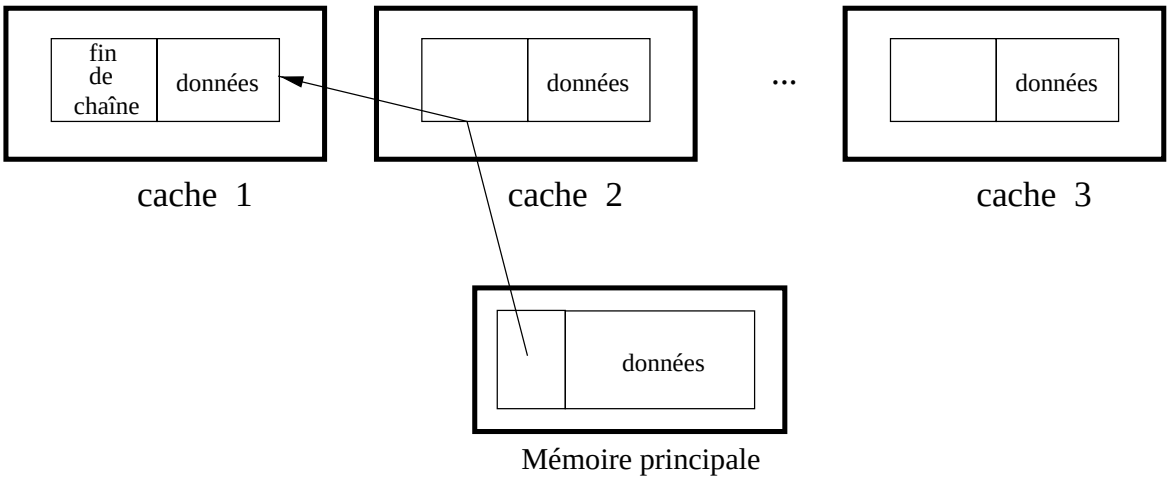


FIG. A.9 – *Principe des répertoires chaînés.*

Dans ce système, les caches qui possèdent la même donnée forment une liste dont le point de départ est le bloc de données correspondant dans la mémoire centrale. Dans la figure A.9, une liste relie la mémoire principale, le cache 2 et le cache 1. Ceci dénote que le cache 1 et le cache 2 ont tous les deux une copie du bloc concerné par cette liste. Cette liste est extensible (à la limite, tous les caches peuvent en faire partie), et prend toujours la même place dans le répertoire principal quelle que soit sa longueur. Cette méthode n'échappe pas à des inconvénients. Le temps d'accès est allongé car le parcours de la liste prend du temps. De plus, la gestion de la liste est complexe, l'insertion est facile, mais

la suppression est plus délicate. C'est pour essayer de remédier à ces inconvénients que des solutions ont été proposées, d'une part, transformer la liste en une liste doublement chaînée pour faciliter la suppression, et d'autre part, créer un arbre de manière à permettre un parcours plus rapide qu'avec une liste ($log_2 n$ au lieu de n). Cependant, le choix de la manière d'effectuer les opérations de gestion de la liste est vaste et correspond à toutes les solutions algorithmiques possibles.

A.2.4 Les répertoires spéciaux.

Stenström a proposé dans [Ste89], une méthode qui permet de ne plus avoir de répertoire centralisé dans la mémoire principale. Elle gère les mêmes informations que le système de Censier et Feautrier, mais celles-ci sont présentes dans les caches (voir figure A.10).

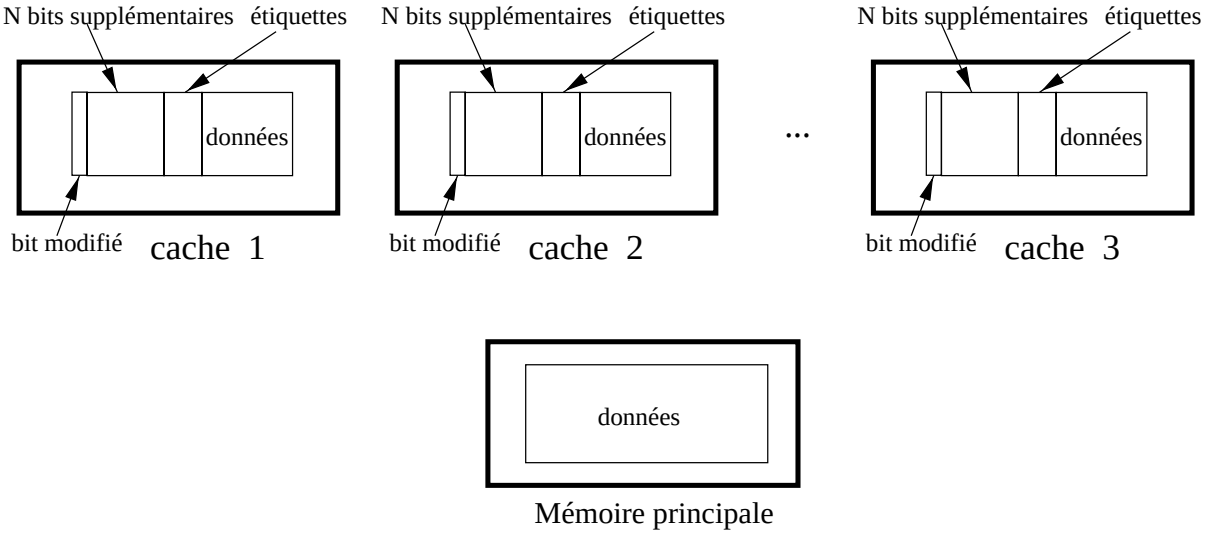


FIG. A.10 – *Les répertoires du système de Stenström.*

Ce mécanisme a l'avantage de restreindre le nombre d'accès à la mémoire principale qui pouvait devenir un goulot d'étranglement pour un grand nombre de processeurs. La taille des répertoires est proportionnelle à la taille des caches alors qu'avec Censier et Feautrier, elle était proportionnelle à la taille de la mémoire.

Pour réduire la taille qu'occupent en mémoire les répertoires, Maa, Pradhan et Thiebaut ont proposé dans [MPT91], deux méthodes de stockage des répertoires.
La première méthode est une extension des répertoires limités. Elle se comporte comme les répertoires limités si le nombre de copies cachées est inférieur à la limite. Dans le cas

d'un dépassement du nombre de copies autorisées, une arborescence est créée comme dans la figure A.11.

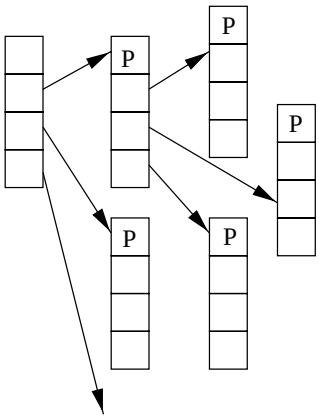


FIG. A.11 – *Les répertoires sont organisés en arbre. (P représente un pointeurs vers le parent et le nombre limite est de 4)*

Si le nombre de copies est inférieur à la limite, l'entrée des répertoires est composée d'un pointeur vers le cache adéquat et d'un bit signifiant que c'est le niveau terminal. Si le nombre de copies dépasse la limite, le bit change d'état et l'entrée du répertoire pointe maintenant vers un autre noeud de l'arbre. Les pointeurs vers les caches, ne se retrouvent que dans les feuilles de l'arbre.

Maa, Pradhan et Thiebaut ont présenté un autre mécanisme qu'ils ont appelé répertoire entier hiérarchique. Ici, le répertoire conserve les mêmes informations que dans le système de Censier et Feautrier. Cependant, le répertoire s'organise comme un arbre. Les caches sont regroupés en sous-ensembles qui forment des ensembles et ainsi de suite. L'ensemble final représente l'information pour tous les caches. Un noeud contient l'information pour tous les noeuds qui se trouvent en dessous de lui. Ainsi, si tous les noeuds contenus dans un sous-arbre ne contiennent pas la copie d'un bloc, ce sous-arbre peut être élagué et l'information ne reste que dans le noeud qui représente ce sous-arbre.

Les répertoires flous sont présentés dans [IH89]. Une fois de plus, il s'agit de trouver le meilleur compromis entre la taille que représente un répertoire en mémoire et surtout son extensibilité et l'utilisation de diffusions. Pour cela, ce mécanisme limite le nombre d'entrées possibles des répertoires et autorise des diffusions restreintes. Pour cela, si le nombre d'entrées allouées est dépassé, un bit est changé et permet de faire en sorte qu'une entrée ne corresponde plus à un cache mais à un ensemble de cache. Ainsi le nombre d'entrées nécessaire est réduit, mais à l'intérieur d'un ensemble de cache, il est impossible de faire la différence entre les caches qui hébergent une copie et les autres.

C'est pourquoi, une diffusion est nécessaire pour l'ensemble des caches concernés.

A.3 Les méthodes logicielles.

Face à l'agrandissement de la complexité des méthodes matérielles, des méthodes logicielles ont été proposées. Elles ont pour objet de restreindre le partage des données à certains moments du programme, de manière à réduire les communications pour le maintien de la cohérence. Une méthode très restrictive est d'interdire la réplication des données partagées. Ceci n'est pas tolérable, car c'est bien trop contraignant. C'est pourquoi des assouplissements peuvent être envisagés. Tout d'abord, on peut autoriser la réplication des données partagées avec uniquement des accès en lecture. Puis, en assouplissant encore, on peut autoriser la duplication des données partagées, même si elles sont quelquefois accédées en écriture, lors de périodes où elles ne sont utilisées qu'en lecture. Un certain nombre de méthodes pour gérer le partage des données en amont de l'exécution ou lors de celle-ci sont proposées dans [Ste90] et [CV90].

Bibliographie

[AB84] James Archibald and Jean-Loup Baer. An economical solution to the cache coherence problem. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 355–362. IEEE, 1984.

[AB86] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ABB+86] Mike Acetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proceedings of summer Usenix*, July 1986.

[AcD+96] Cristiana Amza, Alan L. cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[AH93] Sarite V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on parallel and distributed systems*, 4(6):613–624, June 1993.

[ASHH88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory scheme for cache coherence. In *Proceeding of the 15th Annual International Symposium on Computer Architecture (ISCA'88)*, pages 280–289, May 1988.

[Boo94] Bob Boothe. Fast accurate simulation of large shared memory multiprocessors. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 251–260, 1994.

[Bou92] J. Y. Le Boudec. The asynchronous transfer mode: A tutorial. *Computer Networks and ISDN Systems*, 24(4):279–309, May 1992.

[BR92] Bob Boothe and Abhiram Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 214–223, Gold Coast, Australia, May 1992.

[BS93] William J. Bolosky and Michael L. Scott. False sharing and its effects on shared memory performance. In *Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71, San Diego, CA, September 1993. USENIX.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1991.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–164. Association for Computing Machinery SIGOPS, October 1991.

[CF78] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CFKA90] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–59, June 1990.

[Cla83] Douglas W. Clark. Cache performance in the vax-11/780. *ACM Transactions on computer systems*, 1(1):24–37, February 1983.

[CLBHL92] Jeffrey S. Chase, Henry M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, University of Washington, March 1992.

[CLLBH92] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bits operating system. In Andreas

Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 397–413, Vancouver, BC CD, October 1992. ACM Press , New York, NY , USA.

[Cre84] E. Mac Creight. The dragon computer system: An early overview. Technical report, Xerox Corporation, September 1984.

[CV90] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–47, June 1990.

[DS93] Nathalie Drach and André Seznec. Semi-unified caches. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume I - Architecture, pages I–25–I–28, Boca Raton, FL, August 1993. CRC Press.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[Dub93] Cezary Dubnicki. The effects of block size on the performance of coherent caches in shared-memory multiprocessors. Tr 462 and ph.d. thesis, University of Rochester, Computer Science Department, May 1993.

[Esk96] M. Rasit Eskicioglu. A Comprehensive Bibliography of Distributed Shared Memory. *ACM Operating Systems Review*, 30(1):71–96, January 1996.

[Fra84] S. J. Frank. Tightly coupled multiprocessor systems speed memory access times. *Electronics*, 57(1):164–169, January 1984.

[FV93] Matthew I. Frank and Mary K. Vernon. A hybrid shared memory/message passing parallel machine. In Syracuse University, editor, *Proceedings of the 1993 International Conference on Parallel Processing*, volume I - Architecture, pages I–232–I–236, Boca Raton, FL, August 1993. CRC Press.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Computer Architecture News*, 18(2):15–26, June 1990.

[Goo83] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, New York NY (USA), June 1983. IEEE.

[GW88] James R. Goodman and Philip J. Woest. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, 1988.

[HLH92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[HP92] John L. Hennessy and David A. Patterson. *Architecture des ordinateurs : Une approche quantitative*. Mac Graw-Hill, 1992.

[IH89] E. D. Brooks III and J. E. Hoag. A scalable coherent cache system with fuzzy directory state. Technical report, Lawrence Livermore National Laboratory, University of california, 1989.

[IKWS92] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The effects of virtually addressed caches on virtual memory design and performance. *ACM Operating systems review*, 26(4):14–29, October 1992.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. New directions in scalable shared memory multiprocessor architectures: Scalable coherent interface. *IEEE Computer*, 23(6):74–77, June 1990.

[Jr.87] Andrew W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–252, 1987.

[KEW+85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache-consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, 1985.

[KJA+93] David Kranz, Kirk Johnson, Anant Agarwal, john Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared memory: Early experience. In *Proceedings of th 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*, pages 54–63, July 1993.

[KL94] Alexander C. Klaiber and Henry M. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *Proceedings of th 21st Annual Symposium on Computer Architecture*, pages 94–105, May 1994.

[Lar90] James R. Larus. Spim s20: A mips r2000 simulator. Technical Report TR 966, Computer Sciences Department, University of Wisconsin-Madison, September 1990.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LKBT92] W. G. Levelt, M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaum. A Comparison of Two Paradigms for Distributed Shared Memory. *Software–Practice and Experience*, 22(11):985–1010, November 1992.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford Dash multiprocessor. *Computer*, 25(3):63–79, March 1992.

[LP92] Z. Lahjomri and T. Priol. Koan: A shared virtual memory for the iPSC/2 hypercube. *Lecture Notes in Computer Science*, 634:441–452, 1992.

[LSB88] Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown. Large-scale parallel programming: experience with BBN butterfly parallel processor. *ACM SIGPLAN Notices*, 23(9):161–172, September 1988.

[Mos93] David Mosberger. Memory consistency models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

[MPT91] Yeong-Chang Maa, Dhiraj K. Pradhan, and Dominique Thiebaut. Two economical directory schemes for large-scale cache coherent multiprocessors. *Computer Architecture News*, 19(5):10–18, September 1991.

[MS94] Maged M. Michael and Michael L. Scott. Scalability of atomic primitives on distributed shared memory multiprocessors. Technical Report TR.528, University of Rochester, Computer Science Departement, July 1994.

[NPA92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *t: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, May 1992.

[Ope92a] Open Software Foundation and Carnegie Mellon University. *Mach 3 Kernel Principles*, July 1992.
ftp:mach.cs.cmu.edu doc/osf/kernel_principles.Z.

[Ope92b] Open Software Foundation and Carnegie Mellon University. *Mach 3 Server Writer's Guide*, July 1992.
ftp:mach.cs.cmu.edu doc/osf/server_writer.Z.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weise. The IBM research parallel processor prototype (RP3): Introduction. In *Proc. Int. Conf. on Parallel Processing*, August 1985.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[Rai92] Sanjay Raina. Virtual shared memory: A survey of techniques and systems. technical Report CSTR-92-36, Department of Computer Science, University of Bristol, UK, December 1992.

[RS84] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[SB93] André Seznec and François Bodin. Skewed-associative caches. In *Parallel Architectures and Languages Europe*, pages 305–316, June 1993.

[Smi87] Alan Jay Smith. Cache memory design: an evolving art. *IEEE Spectrum*, 24(12):40–44, December 1987.

[Spi77] Jeffrey R. Spirn. *Program Behavior: Models and Measurments.* American Elsevier, New York, 1 edition, 1977.

[Sta88] Mark Stansberry. Cache memory design in 32-bit microprocessor systems. *VLSI systems design*, 9(3):32–42, March 1988.

[Ste89] P. Stenström. A cache consistency protocol for multiprocessors with multistage networks. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 407–415. IEEE, May 1989.

[Ste90] Per Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[SZ90] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.

[Tan76] C. K. Tang. Cache design in the tightly coupled multiprocessor system. *AFIPS Conference Proceedings, National Computer Conference*, pages 749–753, June 1976.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating systems, ASPLOSII*, pages 164–172, 1987.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[Wyl87] David C. Wyland. Cache tag ram chips boost speed and simplify design. *Computer design*, 26(20):85–90, November 1987.

[YYF85] Wei C. Yen, David W. L. Yen, and King-Sun Fu. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, C-34(1):56–65, January 1985.