N° d'ordre: 2040

THÈSE

Présentée devant

l'Université de Rennes 1
Institut de Formation Supérieure en Informatique et Communication

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE
École Doctorale : Informatique, Traitement du Signal et Télécommunications

par

Dan Nam TRUONG

Équipe d'accueil : IRISA
Composante universitaire : IFSIC

Optimisations logicielles de la localité :
le placement précis des données en mémoire

soutenue le 21 Septembre 1998 devant la commission d'examen

M. :	Jean-Pierre	BANÂTRE	Président
MM. :	Daniel	ETIEMBLE	Rapporteurs
	Daniel	LITAIZE	
MM. :	Olivier	SENTIEYS	Examinateurs
	François	BODIN	
	André	SEZNEC	

Remerciements

Je remercie Jean-Pierre BANÂTRE, Directeur de l'Institut de Recherche en Informatique et Systèmes Aléatoires, qui me fait l'honneur de présider ce jury.

Je remercie Daniel ETIEMBLE, Professeur à l'Université Paul Sabatier de Toulouse, et Daniel LITAIZE, Professeur à l'Université d'Orsay Paris-Sud, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Olivier SENTIEYS, Chargé de Recherche au Laboratoire d'Analyse des Signaux et de Traitement de l'Information de Lannion, d'avoir bien voulu juger ce travail.

Je remercie enfin André SEZNEC, Directeur de Recherche à l'Institut de Recherche en Informatique et Systèmes Aléatoires, et François BODIN, Professeur à l'Université de Rennes 1 qui ont dirigé ma thèse.

Chapitre 1

Introduction générale

La vitesse des processeurs croît plus vite que la vitesse des mémoires dynamiques, et aujourd'hui les processeurs sont déjà nettement plus rapides que les mémoires dynamiques. En effet, les contraintes économiques poussent les architectes à privilegier la vitesse des processeurs, alors que c'est la capacité de stockage qu'ils doivent privilegier pour les mémoires. Pour exploiter la puissance de calcul du processeur malgrès cette disparité de performance, les architectes ajoutent des caches à la hiérarchie mémoire. Il est étudié ici comment améliorer la performances des machines en améliorant le comportement des caches, à l'aide de techniques logicielles basées sur la réorganisation des données en mémoire.

Un cache compense la latence de la mémoire en conservant les données les plus utilisées. Il évite ainsi au processeur la plupart des accès mémoire. Afin de déterminer automatiquement quelles sont les données à conserver, le cache utilise les propriétés de localité des applications : si une zone mémoire est référencée, alors il est probable qu'elle même ou qu'une zone mémoire proche sera référencée dans un futur très proche.

Cette propriété de localité découle de la manière dont les programmes sont conçus et compilés. Par exemple, un programmeur aura tendance à déclarer ses variables en fonction de leur utilisation. Cependant, avec l'augmentation de l'occupation mémoire des applications et la nécessité de tirer le meilleur parti de la mémoire cache, il devient nécessaire d'extraire plus de localité des programmes.

Les méthodes matérielles ou architecturales consistent à modifier les mémoires cache pour les rendre plus performantes en détectant mieux la localité. Bien que nécessaire, cette approche est coûteuse et reste insuffisante.

Les méthodes logicielles consistent à améliorer la localité des références mémoires. Deux solutions sont possibles.

La première est de transformer le code de l'application pour changer l'ordre des références mémoire. Cette solution est très largement utilisée pour améliorer la performance des applications scientifiques manipulant des matrices numériques. Il s'agit des transformations de boucles ou *loop transformations*.

La seconde solution qui sera étudiée ici, consiste à modifier la position des données en mémoire. Cela change indirectement l'ordre des accès mémoire et donc la localité des accès. Il s'agit du placement de données ou *data layout*.

Historiquement, avec le passage des machines vectorielles aux machines à mémoire distribuées, les études visant à améliorer la localité des programmes se sont surtout concentrées sur les programmes scientifiques. Cependant, pour les matrices, les possibilités de placement des données sont limitées, et offrent un intérêt bien moindre que les transformations de boucles. Cette approche a donc été très peu étudiée, au profit des techniques de transformation de code.

Les programmes généraux rencontrés couramment dans le monde Unix ou PC sont le plus souvent écrits en langages de haut niveau (C, C++, Pascal, Ada...). Ces langages de programmation offrent de nombreux types de données, pas nécessairement scalaires, comme les structures de données. Les éléments des structures de données ne sont pas accédés avec des calculs d'indices. Les transformations de code développées pour les matrices ne s'appliquent donc pas à ces types de données.

Le placement des données en mémoire semble prometteur pour les données structurées de manière non matricielle, car ces données bénéficient d'une plus grande souplesse de placement en mémoire dont on peut tirer profit (organisation des données en listes chaînées, en arbres, etc.).

L'étude évalue en premier lieu les gains de localité qu'il est possible d'espérer en réorganisant les données en mémoire, puis propose dans un deuxième temps des solutions de placement de données pour améliorer la performance des programmes. Le cas des structures de données allouées dynamiquement est étudié plus particulièrement. Ces dernières sont fréquemment utilisées dans les programmes généralistes.

1.1 Etat de l'art

De nombreux travaux ont été proposés ces dernières années pour améliorer la performance des caches [Smi86]. Certains de ces travaux proposent des solutions matérielles pour améliorer le taux de réussite des caches, et d'autres pour réduire la pénalité d'un accès à la mémoire. Le préchargement des données est envisagé comme une solution prometteuse, mais il est nécessaire pour cela d'avoir des programmes avec de bonnes propriétés de localité pour que le préchargement soit efficace. Il faudra donc avoir recours lors de la compilation des programmes à des techniques de transformation de code et de placement des données en mémoire pour garantir une bonne localité des références mémoire des programmes.

Pour augmenter le taux de réussite des caches, c'est à dire le nombre de références servies par le cache, des solutions sont proposées pour améliorer leur gestion à moindre coût [AP93, Jou90, Sez94, BS95] ou empêcher le chargement de données qui ne sont pas réutilisées, avec des instructions de bipasse[1] [TFMP95].

Pour réduire le coût d'un accès mémoire, de nouveaux types de mémoires apparaissent, plus performantes sans pour autant sacrifier leur capacité de stockage. Les mémoires SDRAM [Prz93, BD94], en pipelinant les accès permettent de réduire la

1. Instructions de chargement de données dans le processeur sans la mémoriser dans le cache.

latence apparente d'un accès mémoire et augmenter la bande passante mémoire. Cependant elles s'appuient sur les caches et ont besoin de la localité des accès pour fonctionner [KOI95]. Pour réduire la latence des mémoires, [AK96] propose dans chaque puce mémoire d'exploiter la localité des références afin de prédire les données qui seront utilisées, et de les précharger dans les tampons de sortie. Cela évite de payer le coût d'un accès aux bancs mémoire (mais pas le coût de l'accès au bus).

Quantités de solutions sont proposées pour résoudre le problème de la latence mémoire. Il est possible d'ajouter plusieurs niveaux de cache [JW94], d'utiliser des caches non bloquants qui permettent au processeur de continuer alors que le cache attend le retour d'une donnée [CB92, FJ94] ou de faire du préchargement (*prefetch*).

L'organisation des caches en lignes cause automatiquement le préchargement d'un bloc mémoire de plusieurs mots [Prz90][2]. Pour aller au-delà, le préchargement peut être fait par le matériel en préchargeant des blocs mémoire consécutifs avec des *Stream Buffers* [Jou90, PK94], en prédisant le pas des accès mémoire avec une *Stride Prediction Table* [FPJ92] ou de manière plus complexe en combinant l'utilisation d'une table de prédiction, et d'un compteur ordinal de préchargement (le *Look-Ahead PC*) qui devance le compteur ordinal [BC91, CB92, CB95, Che95]

De nombreuses solutions de préchargement logiciel ont aussi été proposées pour gérer la latence mémoire au niveau du cache.

Elles sont plus souples, car elles bénéficient d'une analyse statique du programme et n'ont pas à être systématiques. De nombreux processeurs ont des instructions de préchargement implantées, et certains compilateurs les utilisent. Les études sur le préchargement logiciel restent cependant majoritairement liées au calcul des pas d'accès à la mémoire et donc au chargement de vecteurs ou de matrices dans le cache [Por89, CKP91, TE95]. Quelques études s'intéressent au préchargement de structures de données allouées dynamiquement, soit en préchargeant les pointeurs passés en paramètres aux fonctions [LSKR95], soit en suivant les pointeurs dans les instances de structures allouées pour les précharger [ZT95, LM96]. Ces solutions se heurtent au problème du *pointer chasing*, c'est-à-dire qu'il faut lire les pointeurs en mémoire pour les précharger. Cela rend difficile le préchargement de structures récursives (structures s'auto-référençant, comme les noeuds de listes chaînées ou d'arbres) [LM96].

Le problème du préchargement est qu'il ne modifie pas la localité des références. Si un programme a une mauvaise localité, il faut beaucoup de préchargements pour compenser le manque de localité, au risque de générer des interférences dans le cache et de saturer la bande passante mémoire [SL95].

La limitation du nombre de broches des processeurs amènera un problème de bande passante vers la mémoire [BGK96] appelé *memory wall*. Cela pose un problème pour l'utilisation du préchargement, qui augmente en général la bande passante utilisée. Il a donc été proposé de placer sur le même composant la mémoire vive et le processeur [WM95, SPN96], mais cela, à long terme ne fera que déplacer le problème.

La difficulté que rencontrent toutes ces solutions est que les programmes ont des comportements très variables qui modifient considérablement leurs propriétés de loca-

2. Section 4.3 page 67 "*Block size - Memory Speed Tradeoffs*"

lité. Les politiques de gestion qui sont implantées dans les caches ne peuvent donc pas exploiter tout le potentiel de localité d'un programme et sont donc très peu efficaces, comparées à la politique de gestion optimale qui pourrait être faite si l'on connaissait à l'avance les motifs d'accès mémoire [Bel66, Den70, Sug93]. Le résultat est que la capacité des caches est largement sous exploitée car peu de données stockées sont en fait réutilisées [BGK95]. Si l'on arrivait à améliorer l'exploitation de la localité par les caches, les caches actuels suffiraient pour garantir la performance des systèmes [HS95], et la bande passante utilisée serait beaucoup plus faible [HS96].

Une solution pour améliorer l'utilisation des caches est de modifier les programmes, et d'adapter leurs propriétés de localité à l'organisation matérielle de la hiérarchie mémoire.

Pour les programmes scientifiques, souvent écrit en Fortran77, les accès aux matrices se font par le biais de boucles de calcul. Pour améliorer la localité des accès mémoires, des techniques de transformations de boucle permettent de changer l'ordre des références mémoire pour en améliorer la localité [BGS94, CM95, CKT94, McK92, Por89]. Cela se fait en modélisant les accès générés par les boucles [WL91] et en analysant l'impact qu'ils ont sur les défauts de cache [Tem93].

Plus récemment, des techniques de placement mémoire des matrices, appelées *array padding* ont été étudiées [BCJ94, BS95, SSGC95]. Ces techniques complètent les techniques de transformations de boucles et peuvent être développées en utilisant les mêmes outils d'analyse [JD91, CL95, Shi96, GMM97]. Le *padding* reste une technique limitée, à cause des contraintes de placement mémoire. Comme l'accès aux éléments des matrices se fait avec un index, la position relative des éléments reste figée.

Pour les données ayant des contraintes de placement mémoire moins sévères, le placement mémoire semble par contre plus prometteur que les transformations de code. Les transformations de boucles modifient le calcul des indices des éléments de matrices pour modifier l'ordre des accès. Ces techniques ne s'appliquent donc pas aux de types de données qui ne sont pas indexées.

C'est pour cette raison qu'il y a encore peu de solutions de placement proposées pour les données qui ne sont pas organisées en matrices. Il faut d'abord développer de nouvelles méthodes d'analyse et de placement mémoire.

Des solutions de placement mémoire ont déjà été proposées pour les instructions du programme, dans le but de rapprocher la performance de caches instructions à celle de caches ayant une gestion optimale [McF89]. Les techniques de placement consistent à déplacer les blocs de base (*basic blocs*) du programme, pour améliorer la localité du programme dans le cache instruction. Cela peut apporter un gain significatif, car le compilateur place souvent à la suite en mémoire des morceaux de codes ayant une exécution mutuellement exclusive. Il est même parfois avantageux de charger les lignes de cache du coté du branchement non pris [PM96], car cela peut précharger le code vers lequel le programme doit sauter après une zone d'exécution mutuellement exclusive (*if/else*), ou encore le code de retour d'un appel à une procédure.

Les techniques proposées permettent de réduire les interférences entre les procédures qui s'appellent en les plaçant correctement en mémoire [Gos93, HKC97]. On peut réduire l'espace qu'occupent les procédures dans le cache en séparant les blocs inutilisés (le *fluff code* [PH90]) des blocs fréquemment utilisés [HC89, MPS94, TXD95]. Cela permet en plus d'apporter des optimisations supplémentaires sur le code fréquemment utilisé [CL96], et d'améliorer la prédiction de branchement statique (du type *backward taken, forward not taken*), en plaçant physiquement les blocs contenant des sauts conditionnels en fin de boucle [PH90, Hei94]. Une fois le code placé correctement en mémoire, il est même possible de faire automatiquement du préchargement d'instructions sans gaspiller la bande passante mémoire [XT96]. On peut ainsi envisager de découpler le chargement des instructions de leur utilisation, afin de supprimer l'impact de la latence mémoire sur le flot d'instruction et de tolérer la variation de l'occupation du bus mémoire.

Les résultats obtenus pour le placement des instructions laissent penser qu'un gain significatif est possible pour le flot de données. En effet, les optimisations combinées sur les instructions apportent très souvent un gain de vitesse d'exécution de 10%, alors que le flot d'instruction a de bien meilleures propriétés de localité que le flot de donnée. La performance du cache de données a aussi un impact très important sur le temps d'exécution.

La présente étude se porte plus particulièrement sur les programmes "généralistes", qui ne font pas spécifiquement un usage intensif de matrices numériques. Pour ces programmes, peu d'optimisations sont disponibles pour améliorer la localité spatiale des données, mis à part l'allocation de registre [ASU86] qui élimine déjà une grande partie des accès mémoires aux variables scalaires.

Une catégorie importante de types de données sont les structures de données. Le plus souvent allouées dynamiquement, elles sont souvent utilisées dans les programmes écrits dans des langages de haut niveau.

L'optimisation de la performance et de la localité des librairies d'allocation dynamique a été étudiée pour les programmes écrits en C et C++ [GZH93, ZG92b, GZ93, ZG92a, BZ93, CGZ94], et aussi pour les programmes écrits dans des langages utilisant des ramasses miettes (le *garbage collection*) [Zor93, CG96] mais la localité des données allouées dynamiquement n'a jamais été analysée à notre connaissance.

Il parait donc important de trouver des solutions de placement pour ces données allouées dynamiquement si l'on veut que les programmes "généralistes" profitent aussi de l'évolution des performances des processeurs.

1.2 Organisation du document

Cette thèse montre l'impact du placement des données sur la performance de la hiérarchie mémoire, plus particulièrement pour les programmes généralistes. Elle propose une librairie d'allocation optimisée et des techniques de placement mémoire pour les structures de données allouées dynamiquement, afin d'améliorer la localité spatiale des programmes.

Dans le chapitre 2, le problème de la latence mémoire est exposé, et l'utilité et le fonctionnement des mémoires caches est détaillé. Le temps d'accès aux mémoires dynamiques croit très lentement par rapport à la fréquence des processeurs. Ceux-ci de plus deviennent de plus en plus sophistiqués, et les processeurs actuels permettent d'exécuter jusqu'à 4 instructions par cycles, alors qu'il y a encore quelques années, les processeurs CISC ne pouvaient exécuter qu'une instruction tous les 2 ou 3 cycles. L'évolution de l'interface des mémoires dynamiques (DRAM) permet d'améliorer la bande passante mémoire, mais cela reste insuffisant pour compenser le retard accumulé vis à vis des processeurs. Il faut donc avoir recours aux caches. Les caches réduisent la bande passante mémoire nécessaire, et cachent une partie de la latence mémoire au processeur.

Dans le chapitre 3, une nouvelle classification des défauts de cache est exposée. Elle permet d'analyser l'efficacité des techniques de placement de données en mémoire. Pour mieux évaluer la performance des caches, les défauts de caches sont classifiés en fonction de leur causes ; un défaut de cache a lieu lorsque le cache ne possède pas la donnée que veut accéder le processeur. La classification la plus utilisée, communément appelée la classification des *3C* a été proposée par M. D. Hill [Hil87]. Elle distingue trois types de défauts. Un défaut peut avoir lieu lorsque la donnée n'a pas encore été référencée. Il est classifié comme un défaut de démarrage. Il peut aussi avoir lieu parce que le cache est trop petit pour contenir toutes les données. C'est alors un défaut de capacité. Lorsque le défaut est causé par une gestion non-optimale du cache, il est classé comme un défaut de conflit. Cette classification a été raffinée et complétée par Sugumar [Sug93]. Il décompose les défauts de conflit en défauts de placement et en défauts de remplacement. Cependant, ces classifications ne permettent pas de caractériser les défauts de cache générés par une mauvaise distribution des données en mémoire. Ces classifications sont donc complétées en différenciant les défauts de capacité en défauts fondamentaux et en défauts de distribution. Les défauts fondamentaux sont définis comme les défauts de cache que l'on ne peut pas supprimer, ni en optimisant le placement des données du programme, ni en optimisant la gestion du cache. Les défauts de distribution sont des défauts qui ne peuvent pas être supprimés en améliorant l'architecture du cache mais qui peuvent l'être en améliorant le placement des données du programme, de manière à augmenter la localité des références. Compléter ainsi les classifications permet d'étudier l'impact du placement mémoire sur la performance des caches.

Le chapitre 4 essaye de caractériser le gain potentiel que pourraient apporter des techniques de placement de données. Pour cela, le cas idéal où les données n'ont pas de contraintes de placement en mémoire est étudié, par l'intermédiaire de traces de programmes. Des mesures pour évaluer la localité des références du programme sont développées, afin de déterminer quelles sont les données qui doivent être conservées ensemble dans le cache. A l'aide de ces mesures, une heuristique est développée pour regrouper les scalaires en blocs de manière à augmenter la localité spatiale. Ces mesures sont aussi utilisées pour placer ces blocs à des adresses mémoire pour maximiser l'utilisation de la localité temporelle. Il est montré ainsi qu'il est possible d'améliorer le taux de défauts de cache en éliminant une partie des défauts de conflits ainsi qu'une partie des défauts de distribution. Les résultats indiquent qu'un cache à correspondance directe peut parfois être plus performant avec un programme pour lequel le placement des données a été optimisé qu'un cache géré de manière optimale accédé par le même programme, mais sans optimisation du placement des données.

Ces résultats encouragent le développement de solutions de placement mémoire pour chaque type de donnée. Comme un certain nombre de travaux ont déjà été fait sur le *array padding*, l'étude se porte surtout sur l'optimisation de la localité de programmes généralistes. Les plus gros de ces programmes utilisent de façon systématique l'allocation dynamique pour gérer la mémoire. Une grande partie d'entre eux utilisent intensivement l'allocation de structures de données hétérogènes. L'étude se penche donc sur l'optimisation de la localité des structures de données allouées dynamiquement.

Le chapitre 5 propose des techniques de placement mémoire et une librairie d'allocation optimisée pour les structures de données hétérogènes. Dans une première partie, l'impact de la librairie d'allocation dynamique sur la performance des programmes est mis en évidence, et *ialloc*, une librairie d'allocation optimisée est proposé. Celle-ci gère la mémoire allouée dynamiquement sans détruire la localité spatiale qui peut apparaître entre des instances contiguës en mémoire. De plus, elle est plus performante qu'une librairie standard. Elle fait moins d'appels systèmes, nécessite moins d'instructions pour gérer la mémoire et gaspille moins de mémoire qu'une librairie d'allocation standard. Dans une seconde partie, deux techniques pour améliorer la localité spatiale des données sont proposées. La réorganisation des champs d'une structure permet de mettre côte à côte en mémoire les champs de la structure souvent référencés ensemble. L'entrelacement des instances d'une structure permet d'améliorer la localité spatiale entre plusieurs instances d'une structure, en regroupant en mémoire les champs fréquemment utilisés de plusieurs instances. La combinaison de ces améliorations permet d'augmenter la performance des applications limitées par l'efficacité de la hiérarchie mémoire.

Chapitre 2

Principe de la hiérarchie mémoire

2.1 Introduction

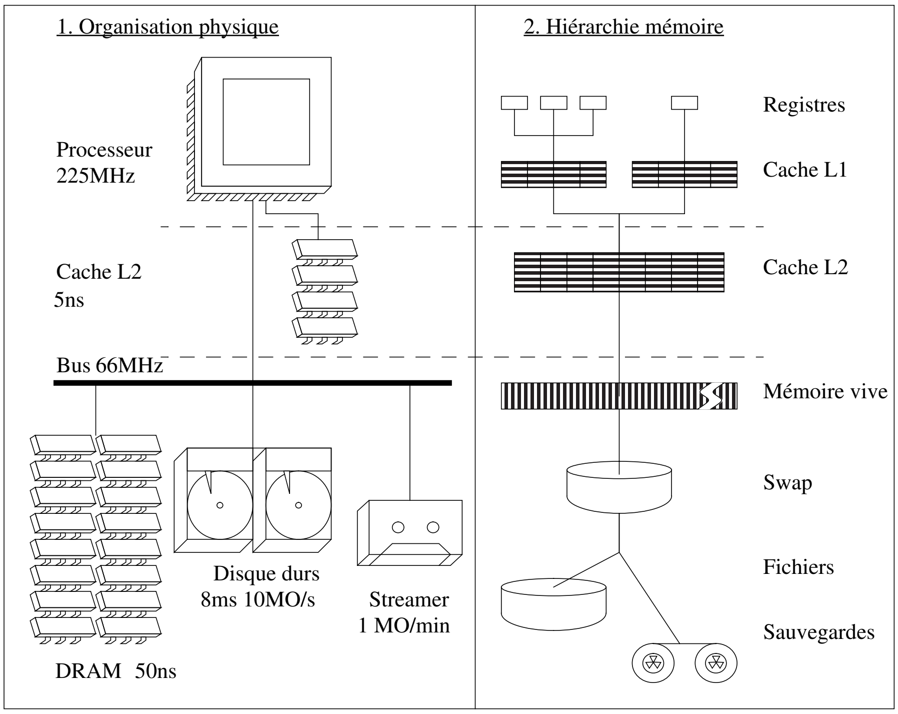


FIG. 2.1 – *Organisation logique et physique de la hiérarchie mémoire.*

Pour compenser la différence de vitesse entre processeur et mémoire, on utilise des caches. Cependant, le temps de cycle des processeurs diminue plus vite que le temps d'accès à la mémoire vive. Pour compenser cette disparité, les caches doivent devenir de plus en plus efficaces.

La hiérarchie mémoire d'un ordinateur s'organise comme en figure 2.1, du niveau le plus rapide et le plus coûteux au niveau le plus lent et le plus économique, afin d'offrir la meilleure performance et la plus grande capacité de stockage au meilleur prix [JCSM96]. Par exemple, le tableau 2.1 récapitule les caractéristiques[1] de la hiérarchie mémoire d'un PC à base de Pentium P54CS à 225MHz.

Pour être performant, un programme doit accéder le moins souvent aux niveaux inférieurs de la hiérarchie mémoire. Pour cela, la gestion de chaque niveau de la hiérarchie doit être la meilleure possible, afin d'éviter de gaspiller la place. En conservant le maximum de données utiles à chaque niveau, on minimise les transferts de données entre chaque niveau [HS96].

TAB. 2.1 – *Organisation de la hiérarchie mémoire.*

	Modèle	Latence	Taille de transfert	B. Pass. MO/s	Taille Octets	Coût par MO
registre	P54C	4,3ns	4/8O	3600	16 x 4	(3000F)
D-Cache L1	P54C	4,3ns	32O	950	8K	(3000F)
Cache L2	PB-SRAM	8ns	128O	76	512K	400F
Mémoire vive	S-DRAM	50ns	4KO	56	32M	35F
Swap	Disque dur	8ms	4KO	10	100M	50c
Fichiers	Disque dur	8ms	-	10	4G	50c
Sauvegardes	Bande	min.	-	0,2	∞	5c

En haut de la hiérarchie mémoire se trouvent les registres du processeur et les caches.

La gestion des registres du processeur est critique. Pour que le processeur reste performant, la gestion des registres est confiée au compilateur (allocation de registre [ASU86]), avec le support du jeu d'instructions (opérations registre à registre, l'adresse des registre est codée explicitement dans les instructions).

Les caches sont aussi sollicités très fréquemment, mais ils sont plus gros. C'est pour cette raison qu'ils sont gérés automatiquement par le matériel. Plus la gestion matérielle est complexe, plus elle est efficace, mais plus il faut de couches logiques pour l'implanter et plus le cache devient lent [Prz90, WJ93]. Pour cette raison, les caches de premier niveau restent le plus souvent de conception simple. Ils sont soit à correspondance directe ou avec une associativité faible.

Notre approche, le placement de données, consiste à améliorer l'utilisation des caches par des solutions logicielles. Elle revient à faire, tout comme pour les registres, une gestion statique[2] du placement des données dans le cache. En déchargeant le matériel d'une partie de la gestion, il est possible de combiner gestion performante du cache et implantation matérielle simple, donc très rapide.

Afin de comprendre l'utilité et le fonctionnement des caches, le problème de la latence mémoire est brièvement exposé section 2.2, avant une explication sommaire du

1. La bande passante est calculée avec *ctcm*.
2. pendant la compilation.

fonctionnement global des caches et de l'influence de la localité sur leur conception en section 2.3.

2.2 Latence mémoire

2.2.1 Les choix technologiques

La latence apparente de la mémoire croit de façon exponentielle, car la vitesse des processeurs croit plus vite que le temps d'accès des mémoires vives (60% à 80% par ans contre 7% [PH96]).

Cette tendance est due à des choix technologiques différents, liés à des contraintes économiques divergentes. Pour les DRAMs[3] on favorise la densité, qui croit de 60% par an, alors que pour les processeurs on favorise la vitesse. Ce choix semble irréversible actuellement, car parallèlement la consommation mémoire des applications augmente de 50% à 100% par an [PH96].

Pour maximiser la densité des DRAMs, on ne conserve chaque bit qu'à l'aide d'un condensateur et un transistor [AG92]. Les bits sont organisés en matrices 2D pour ne pas gaspiller la surface du composant.

TAB. 2.2 – *Décomposition du temps d'accès d'une DRAM.*

décodage	accès rangée	accès colonne	sortie	t_{RAC}
7ns	23ns	16ns	7ns	53ns

L'accès à un bit est lent. Il faut activer une rangée de la matrice et détecter le bit de chaque colonne. Plus la matrice est grande, plus l'effet capacitif est important. La table 2.2 montre les 4 composantes incompressibles du temps d'accès t_{RAC} pour une mémoire 64 mégabits, selon Tanabe [Tan94].

A ce temps d'accès il faut ajouter le temps nécessaire pour faire transiter les données sur le bus, entre la mémoire et le processeur, ce qui peut correspondre à 300-400ns, comme l'indique le tableau 2.3 tiré de [MS95]. Par exemple, sur une *Origin 2000* il faut en moyenne[4] 75,5 cycles, soit 387ns.

TAB. 2.3 – *La latence mémoire réelle est beaucoup plus élevée que le temps d'accès d'une mémoire DRAM.*

t_{RAC}	t_{Cycle}	pin-to-pin	back-to-back
60ns	110ns	265ns	**350ns**

3. DRAM : acronyme pour *Dynamic Random Access Memory.*

4. Chiffre empirique utilisé par l'outil d'analyse de performance *Perfex* pour estimer le coût d'un accès mémoire. La latence de la mémoire locale est de 310ns pour ce système, soit 61 cycles, alors que les SDRAM utilisées ont un temps d'accès t_{RAC} de 125ns, et peuvent fonctionner à 83MHz (t_{CK} à 12ns).

Les SRAMs[5] utilisées dans les caches arrivent à suivre la performance des processeurs et n'ont pas besoin d'être rafraîchies ni préchargées. Cependant, elles sont 4 fois plus volumineuses (elles utilisent 6 transistors par bits stockés contre 1,5 transistors), et consomment beaucoup plus. Le tableau 2.4[6] compilé à partir de fiches techniques constructeurs montre une SRAM 16 fois moins volumineuse et consommant 30 fois plus que les DRAMs. L'utilisation des SRAM reste donc cantonnée à la conception des caches. A titre indicatif, les temps d'accès t_{RAC}, t_{CAC}, t_{CK}, définis en 2.2.2 sont aussi fournis dans le tableau.

TAB. 2.4 – *Comparatif entre DRAM et SRAM.*

	SDRAM	EDO DRAM	PB-SRAM
Fabriquant	Fujitsu	Fujitsu	IBM
Modèle	SDC16UV6484-125	ESA8UN3242B-50	IBM041841
Format	4x1Mx16	8Mx32	256Kx18
temps de cycle	77ns	80ns	5ns
temps d'accès	45ns / 21ns / 8ns	50ns / 15ns	2,5ns
fréquence d'utilisation	125MHz	66MHz	200MHz
taille	64Mbit	64Mbit	4Mbit
consommation	1W	700mW	2,25W

2.2.2 Des solutions à court terme

La latence des DRAMs devient trop importante, et les constructeurs exploitent les propriétés physiques de leur DRAMs pour augmenter leur débit.

Une DRAM est composée de plusieurs matrices. Chaque matrice ne fournit qu'un bit à chaque accès, bien que tous les bits d'une rangée soient disponibles. Les constructeurs ont donc développé les modes page pour pouvoir accéder successivement à plusieurs bits d'une rangée sans avoir à recharger la matrice. Le temps d'accès à un bit n'est alors plus t_{RAC} (50ns), mais t_{CAC} (de l'ordre de 25 à 15ns).

La figure 2.2 montre les chronogrammes de trois modes page. L'adresse rangée est fournie juste une fois sur le front descendant du signal RAS. Plusieurs colonnes de la rangée peuvent être accédées en changeant l'adresse colonne.

- Le *Fast Page Mode* (FPM-DRAM) : la donnée est disponible tant que CAS est bas.
- L'*Extended Data Out* (EDO-DRAM) : la donnée est latchée, donc valable jusqu'au prochain front descendant de CAS.
- Le *Pipeline Burst* (PBEDO-DRAM) : l'adresse colonne n'est fournie qu'une fois. A chaque front descendant de CAS, l'adresse est incrémentée automatiquement par la mémoire.

5. SRAM : Acronyme pour *Static Random Access Memory.*
6. PB-SRAM : Acronyme pour *Pipeline Burst Static Random Access Memory.*

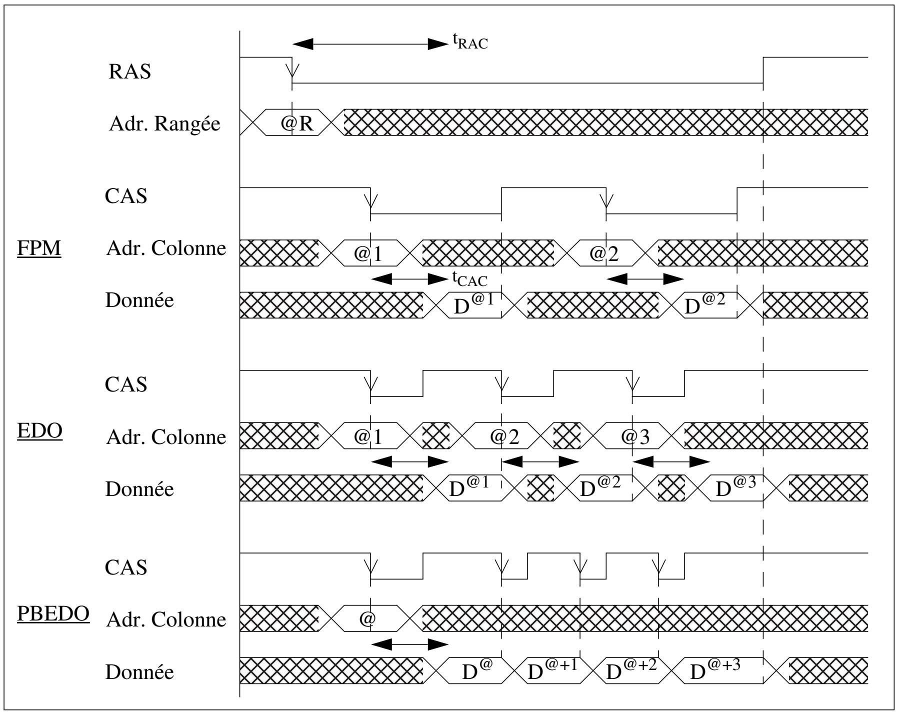


FIG. 2.2 – *Chronogrammes des accès mémoires en mode page.*

En regardant la fugure 2.4, on voit que l'évolution de t_{CAC} suit celle de t_{RAC}. Les modes page sont donc des solutions à court terme.

Les EDO-DRAMs sont supplantées par les SDRAMs (DRAMs synchrones). Le bus est synchronisé par une horloge, de cycle t_{CK}, figure 2.3. Une synchronisation sur l'horloge permet d'éliminer les signaux de synchronisations qui occupent le bus avec les mémoires conventionnelles et ralentissent les opérations. De plus, l'horloge permet de pipeliner les accès mémoire. Pour exploiter le pipeline du bus sans temps morts, les SDRAMs sont organisées en bancs accédés de façon cyclique. Le tableau 2.4 montre les temps d'accès t_{RAC}, t_{CAC} et t_{CK} d'une SDRAM et t_{RAC} et t_{CAC} d'une EDO-DRAM. Lorsque le système peut prédire les pages mémoires qui sont accédées, le temps d'accès devient t_{CAC}. Avec une mémoire SDRAM, lorsque le système arrive en plus à pipeliner [7] les accès mémoire, le temps d'accès apparent devient t_{CK}.

D'autres solutions sont proposées, comme les ERAM, CRAM, RamBUS, afin de rattraper la différence de performance entre DRAMs et processeur [KOI95, Prz93].

Les solutions proposées tentent d'accélérer les accès en exploitant la localité des accès sans sacrifier de la capacité de stockage. La technologie sous-jacente ne change donc pas, la latence d'un accès aléatoire n'est donc pas améliorée. Les solutions proposées n'évolueront donc pas non plus au même rythme que les processeurs, et bénéficieront toujours des optimisations de localité.

7. La profondeur du pipeline est de $\frac{t_{CAC}}{t_{CK}}$, soit de 1 à 4 cycles.

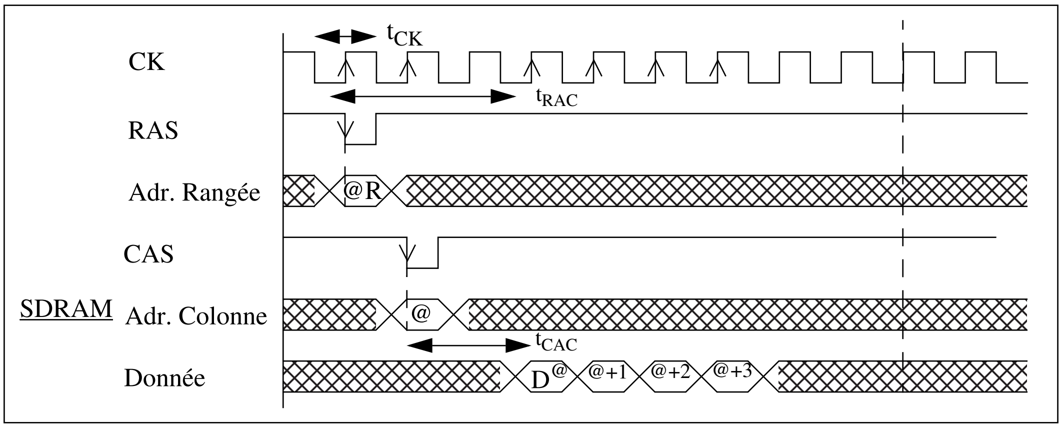


FIG. 2.3 – *Chronogrammes des accès mémoires pour les SDRAM.*

2.2.3 Conclusion

Le tableau 2.4 récapitule l'évolution des temps d'accès des mémoires et l'évolution des temps de cycle des gammes de processeurs DEC AXP et Intel i80x86. On voit que la courbe de latence mémoire apparente diverge des courbes de temps de cycle processeurs. La latence apparente[8] de la mémoire va donc en augmentant exponentiellement, même avec les nouvelles interfaces proposées pour les DRAMs.

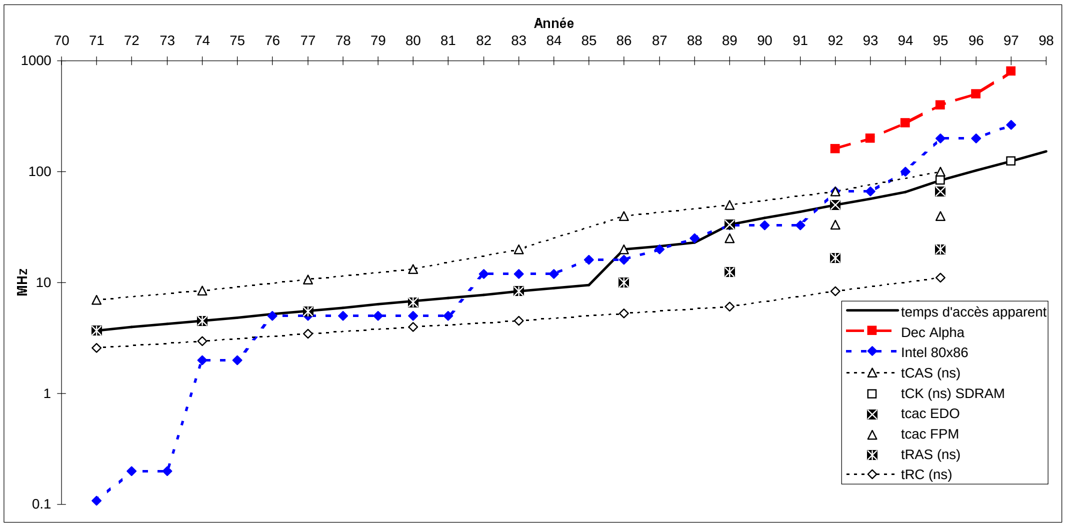


FIG. 2.4 – *Evolution exponentielle de la latence mémoire au cours des années.*

Les processeurs RISC actuels peuvent exécuter simultanément jusqu'à 4 instructions, dont en moyenne 25% à 40% sont des accès mémoire [CB96]. Pour que le processeur ne soit jamais en attente, il faut donc que la hiérarchie mémoire soit capable

8. Le nombre moyen de cycles d'attente des processeurs lors d'un accès mémoire croît.

de satisfaire cinq références, pour fournir à chaque cycle du processeur jusqu'à quatre instructions et une donnée. Cela est en partie possible grâce à une bonne utilisation des mémoires cache.

2.3 Les caches

Un cache permet de réduire l'impact de la latence mémoire sur les performances du processeur. Il conserve une copie des informations accédées le plus récemment en mémoire. Lorsque le processeur réutilise une information déjà présente dans le cache, elle est disponible plus rapidement. On s'affranchit ainsi de la latence mémoire pour la plupart des accès mémoire. Pour fonctionner automatiquement, un cache utilise les propriétés de localité des accès mémoire.

2.3.1 La localité dans les programmes

Il a été observé que les programmes ont tendance à répéter souvent les mêmes opérations. Ainsi, la règle empirique du "90/10" qui stipule qu'un programme passe 90% de son temps à exécuter 10% du programme est souvent vérifiée.

Pour les accès à la mémoire, cette propriété se traduit par la localité des accès:

- **La localité temporelle**: un élément mémoire référencé aura de fortes chances d'être de nouveau référencé prochainement.
- **La localité spatiale**: lorsqu'un élément est référencé, il y a de fortes chances que des éléments contigus en mémoire soient aussi référencés prochainement.

L'organisation matérielle d'un cache est conçue pour exploiter ces deux propriétés.

2.3.2 Fonctionnement

2.3.2.1 Organisation

L'organisation d'un cache associatif par ensemble est décrit très sommairement ici. Pour avoir des exposés plus complets il est préférable de se référer à d'autres études spécialisées sur le sujet [PH96, AG92, Prz90, Jou93].

Un cache est composé de lignes, pouvant stocker chacune un bloc mémoire de plusieurs mots. Pour conserver un bloc mémoire, une ligne de cache doit aussi conserver l'adresse du bloc en mémoire et des bits d'informations indiquant le statut du bloc. L'adresse d'un élément contenu dans une ligne mémoire est calculée en additionnant l'adresse du bloc et le déplacement de l'élément à l'intérieur du bloc.

Les lignes du cache sont organisées en ensembles[9]. On dit que le cache est associatif par ensembles de N lorsque chaque ensemble contient N lignes. La figure 2.5 est un schéma simplifié du fonctionnement d'un cache associatif par ensemble de deux en train de charger un bloc mémoire.

Le bloc provenant de la mémoire est assigné à un ensemble qui est choisi en faisant la correspondance directe entre son adresse mémoire et numéro d'ensemble (On fait

9. en anglais *set*

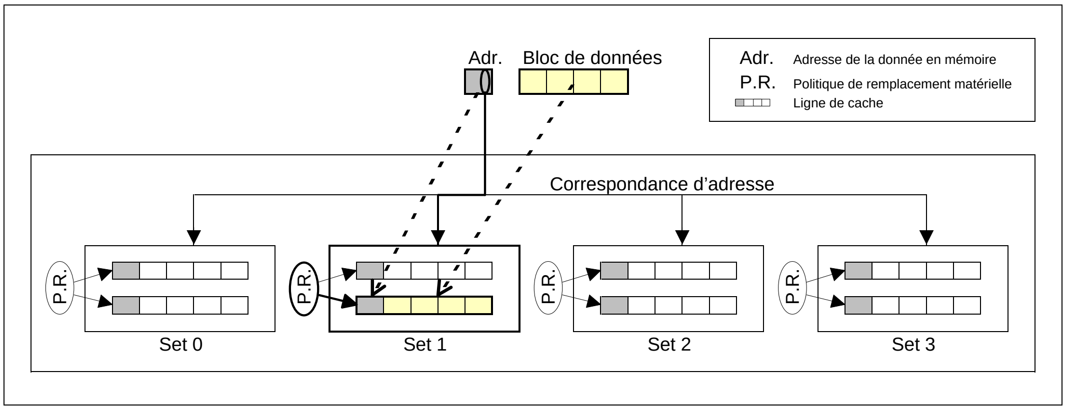


FIG. 2.5 – *Organisation d'un cache associatif par ensemble de deux.*

un modulo de l'adresse sur le nombre de sets). Cette méthode permet de choisir très rapidement l'ensemble dans lequel le bloc doit être conservé.

Le bloc est stocké dans une des lignes de l'ensemble, choisie par la politique de remplacement matérielle du cache (voir section 2.3.2.4). Plus il y a de lignes par ensemble, plus l'implantation de la politique de remplacement est coûteuse. Il faut plus de surface pour l'implanter, et surtout les accès au cache deviennent plus lent [Prz90, WJ93].

Il y a deux cas particuliers aux caches associatifs par ensemble :

Les **caches à correspondance directe** n'utilisent pas de politique de remplacement matérielle. Les ensembles contiennent une seule ligne, et seule l'adresse du bloc à charger détermine la ligne qu'il peut occuper. Ces caches sont très rapides, et sont souvent utilisé comme cache de premier niveau.

Les **caches entièrement associatifs** ne tiennent pas compte de l'adresse du bloc. Toutes les lignes appartiennent à un seul ensemble. Seule la politique de remplacement matérielle choisit la ligne dans laquelle se charge un bloc. Ce type de cache n'est jamais utilisé en pratique, sauf parfois pour de petites structures comme des TLBs, car il est impossible à implanter à un coût raisonnable. Cependant, ce type de cache sert de référence dans les études sur la performance des caches (voir section 3).

La politique de remplacement permet de mieux gérer le cache en capturant mieux la localité temporelle qu'avec une simple correspondance mémoire. Le plus souvent, les architectes tentent donc d'implanter des caches de faible associativité. Par exemple, sur le DEC 21164, le cache de donnée de premier niveau est à correspondance directe, car il conditionne la vitesse d'horloge du processeur. Pour réduire les accès mémoire cependant, un cache de second niveau associatif par ensembles de trois est ajouté dans le processeur pour minimiser les accès mémoire.

L'organisation des caches est conçue pour tirer automatiquement parti de la localité des références des programmes afin de réduire le nombre d'accès à la mémoire.

2.3.2.2 Localité spatiale et découpage en lignes

La localité spatiale permet d'organiser les données du cache en lignes, car lorsqu'un élément d'un bloc est référencé, il est probable que les autres éléments du bloc le seront aussi.

Pour chaque bloc de donnée stocké dans le cache, il faut conserver des informations annexes, comme l'adresse du bloc et son état. En utilisant de plus grandes lignes, on diminue le pourcentage des bits utilisés à stocker des informations annexes.

De plus, comme tous les octets d'un bloc sont chargés dans le cache ensemble, on peut élargir la taille du bus au-delà de 8 pour augmenter la bande passante. En pratique, les blocs sont souvent de 32 octets, et le bus de 64 bits. Il faut donc quatre accès mémoire pour charger une ligne. Pour améliorer quand même la bande passante en tirant partie de l'organisation en lignes, l'interface mémoire a été améliorée, de façon à ce que la latence des 3 derniers accès soit plus courte (modes page et SDRAM, voir section précédente). Par exemple, la notation 3-1-1-1 indique que l'accès au premier 64 bits coûte trois cycles de bus, alors que les autres n'en coûtent qu'un.

Avec de la localité spatiale, l'organisation en lignes compense la latence mémoire par de la bande passante, car les lignes préchargent automatiquement des données qui ont de grandes chances d'être utilisées.

2.3.2.3 Localité spatiale et correspondance mémoire directe

Quand une application exhibe de la localité spatiale, si un bloc est référencé, ceux qui sont proches de lui seront aussi souvent référencés.

En rangeant les blocs dans les ensembles de lignes en fonction du modulo de leur adresse avec la taille du cache, on s'assure que des blocs contigus en mémoire peuvent cohabiter dans le cache sans interférer, car ils ne sont jamais en compétition dans le même ensemble de lignes.

2.3.2.4 Localité temporelle et politique de remplacement

Les politiques de remplacement matérielles utilisent des informations dynamiques pour tenter d'exploiter la localité temporelle des blocs du cache afin de conserver automatiquement dans le cache les blocs les plus utilisés.

Différentes politiques de remplacement sont connues :

- La politique de remplacement optimale **Min** consiste à remplacer le bloc dont la prochaine référence est la plus lointaine dans le futur [Bel66, Den70]. Cependant il n'est pas possible d'implanter cette politique dans un cache, car il faut connaître le futur pour faire ce choix.
- La politique de remplacement **LRU** consiste à remplacer le bloc dont la dernière référence est la plus ancienne parmi tous les blocs stockés dans le cache. On implante des compteurs dans chaque ligne pour exploiter la connaissance du passé en utilisant à "l'envers" le principe de localité temporelle.

– Des politiques **Pseudo-LRU** (NRU[10], LRA[11]...) sont plus souvent utilisées. elles sont une approximation à moindre coût de la politique LRU.
– La politique de remplacement aléatoire **Random** est aussi très utilisée. Elle donne des performances souvent proches du LRU pour un coût bien moindre, car il n'y a pas d'historique à conserver.

2.3.2.5 Gain de performances avec les caches

Le cache est inséré entre le processeur et la mémoire dynamique, comme cela a été montré en introduction du chapitre dans la figure 2.1(2). En satisfaisant directement une partie des requêtes mémoires du processeur, le cache améliore automatiquement la latence apparente de la mémoire.

Le cache intercepte automatiquement toutes les références mémoires du processeur. Les références mémoires à un élément déjà présent dans le cache sont servies tout de suite. On dit qu'il y a un *hit*. Dans le cas contraire, cet élément doit être chargé à partir de la mémoire, et le processeur doit attendre son retour avant de continuer les calculs. On parle alors de *miss*, c'est à dire de *défaut* de cache ou d'*échec*.

La latence mémoire apparente au processeur dépends du nombre de défauts, et peut être calculée par la formule suivante pour chaque niveau de cache.

$$Lat_{apparente} = Hit_{\%} \times Lat_{cache} + Miss_{\%} \times Lat_{memoire}$$

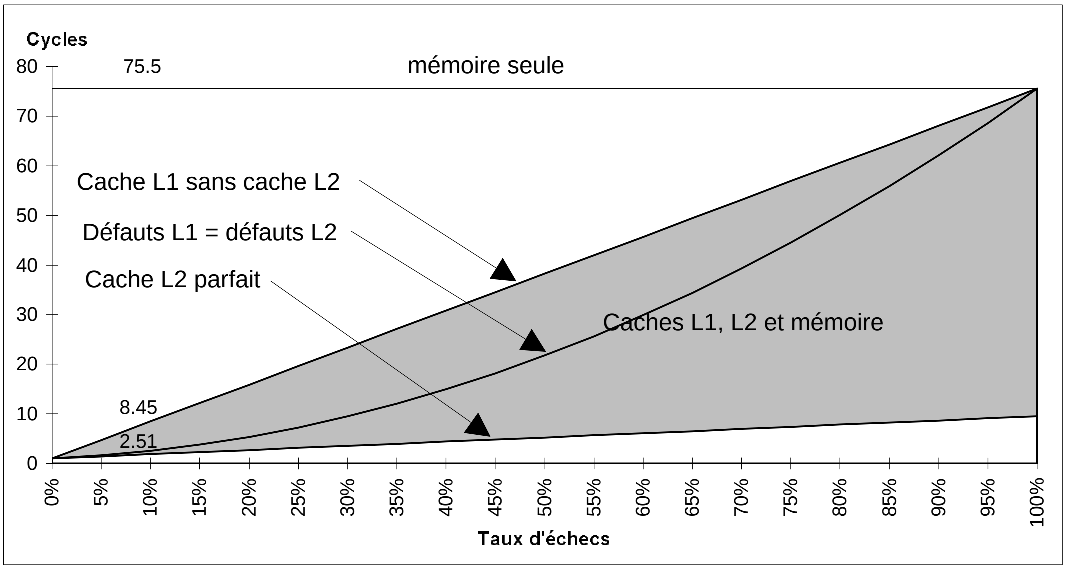


FIG. 2.6 – *Les caches réduisent la latence mémoire apparente.*

La courbe 2.6 montre comme exemple la latence mémoire apparente au processeur en fonction du taux d'échecs du cache de premier niveau L1 pour une hiérarchie mémoire

10. “Not Recently Used”.
11. “Least Recently Accessed”

composée de deux niveaux de cache, plus particulierement quand le cache L2 fait tout le temps des défauts, quand il ne fait aucun défauts, ou quand il en fait autant que le cache L1. La latence mémoire apparente est comprise dans la zone grisée, entre la courbe obtenue sans cache L2 (100% de défauts) et la courbe obtenue avec un cache L2 parfait (0% de défauts). La courbe de latence obtenue quand les caches L1 et L2 génèrent le même nombre de défauts est aussi représentée. La courbe est tracée avec les latences moyennes observées sur les machines SGI Origin 2000 [12]. Avec un niveau de cache et 10% d'échecs, la latence mémoire apparente est réduite de 89% (soit une moyenne de 8,5 cycles contre 75,5 cycles sans cache). Avec deux niveaux de cache, la performance peut encore être accrue.

Lorsque le taux d'échecs des caches est faible, la latence moyenne des accès aux données est réduite de façon significative. Ainsi, l'exécution du flot d'instructions est gelée moins souvent par l'attente du retour des données de la mémoire.

12. La latence des accès est estimée par SGI dans le logiciel perfex à 1 cycle pour L1, 9,01 cycles pour L2 et 75,5 cycles pour la mémoire.

Chapitre 3

Une nouvelle classification des défauts de cache

3.1 Introduction

Un fort taux d'échecs indique un problème de fonctionnement du cache. Pour analyser la performance des organisations matérielles des caches, les défauts ont été classifiés en fonction de la cause qui leur donnent naissance. Pour aller plus loin, les classifications disponibles sont adaptées ici à l'analyse des optimisations aussi bien logicielles que matérielles. Il devient ainsi possible de détecter si un dysfonctionnement du cache est lié à un mauvais placement des données en mémoire.

Une première classification des causes des défauts de cache, connue sous le nom des 3Cs, a été proposée par M. D. Hill [Hil87]. Elle différencie principalement les défauts causés par le manque de capacité du cache des défauts causés par une gestion sous-optimal des remplacements du cache. Cette classification prend comme référence un cache entièrement associatif avec la même politique de remplacement [1] que le cache étudié.

Cette classification a été corrigée par Sugumar [SA93, Sug93]. Il a remarqué que pour certaines traces, des caches de conception simples sont capables de meilleures performances que les caches entièrement associatifs. Dans ce cas, les résultats de la classification de Hill deviennent inconsistants. Pour corriger cela, Sugumar prend comme référence un cache utilisant Min, une politique de remplacement optimale, décrite par Belady [Bel66, Den70] au lieu de la politique de remplacement d'origine.

Comme les politiques de remplacement qui peuvent être implantées sont nettement moins performantes que Min, même les caches LRU entièrement associatifs génèrent des défauts de conflit.

Sugumar affine donc la classification des défauts de conflit. Il distingue les défauts de remplacement, qui sont causés par l'inefficacité de la politique de remplacement matérielle, des défauts de placement, qui apparaissent à cause d'interférences générées

1. Le plus souvent on utilise comme référence une politique LRU, dans l'hypothèse que l'on ne sait pas implanter par matériel de meilleure heuristique.

par la correspondance directe entre les ensembles de lignes du cache et l'adresse des blocs. Sugumar nomme sa nouvelle classification OPT car elle utilise comme référence un cache entièrement associatif avec une politique de remplacement optimale.

Une telle classification permet de faire une analyse fine du comportement d'un cache associatif par ensembles sur une trace d'exécution, mais ne permet pas d'expliquer le comportement d'une application pour une configuration de cache donnée.

Un paramètre primordial n'est pas pris en compte par cette classification : l'influence du placement des données en mémoire. Elle est donc insuffisante pour analyser l'impact des techniques de placements des données étudiés ici.

Pour analyser l'efficacité des optimisations de placement des données, la classification de Sugumar doit être complétée afin de prendre en compte l'influence de la localité spatiale. Pour cela, la mesure des défauts de capacité est affinée en différenciant les défauts fondamentaux des défauts de distribution [TBS96]. Les défauts fondamentaux représentent les défauts de capacité qui apparaissent lorsque la localité spatiale du programme est optimale. Les défauts de distribution sont les défauts de capacité qui apparaissent quand le placement mémoire des données réduit la localité spatiale que peut exploiter le cache.

Le chapitre est organisé de la façon suivante : la classification de M. D. Hill est exposée section 3.2, puis la méthode de mesure des défauts de capacité utilisée par Sugumar dans la section 3.3, et enfin la nouvelle classification des défauts de conflit qu'il propose est détaillée section 3.4. Dans la section 3.5, la nouvelle classification des défauts de capacité proposée est définie. Les expériences, section 3.6, montrent le potentiel des techniques de placement mémoire : Les défauts de démarrage et les défauts fondamentaux ne représentent souvent qu'un petit pourcentage des défauts des programmes.

3.2 Les 3Cs

La classification de M. D. Hill, communément appelée classification des 3C, a pour vocation d'analyser la performance des caches afin de rechercher des causes de dysfonctionnements éventuels.

Hill décompose les défauts de cache en 3 catégories [Hil87]:

- Les **Défauts de démarrage**[2] : Nombre minimum de défauts de cache qu'il faut faire pour accéder au moins une fois à chaque élément mémoire référencé par le programme.
- Les **Défauts de capacité**: Défauts de cache qui surviennent parce que le cache à une capacité trop faible pour contenir toutes les données manipulées par le programme.
- Les **Défauts de conflits**: Défauts de cache causé par une “mauvaise[3]” gestion de la capacité de stockage disponible.

2. En anglais, *compulsory miss* ou *cold start miss*

3. Pour M. D. Hill, la gestion est mauvaise si le cache est moins performant qu'un cache entierement associatif. Nous introduiront dans le chapitre suivant une notion moins empirique de gestion optimale.

Pour évaluer le nombre de défauts de capacité, Hill mesure le nombre de défauts que génère un cache identique au cache étudié, mais entièrement associatif.

En pratique, les défauts de capacité sont le plus souvent mesurés en utilisant un cache LRU entièrement associatif. Le taux de défauts de cette configuration est donc noté *FA.Lru*[4]. Cette approche pour évaluer les défauts de capacité est justifiée par le constat qu'il n'est en pratique pas possible de fabriquer de cache plus performant qu'un cache LRU entièrement associatif.

Dès lors, les 3C se mesurent comme suit :

- Le taux de défauts de démarrage, noté *Cold*, est calculé en divisant le nombre de blocs mémoire distincts chargés dans le cache par le nombre R de références mémoire faites par le programme. Le nombre de blocs peut être obtenu obtenu en divisant la quantité de mémoire M consommée par le programme par la taille des lignes du cache L. $Cold = \frac{\frac{M}{L}}{R}$
- Le taux de défauts de capacité vaut *FA.Lru - Cold*
- T représente le taux de défauts total du cache étudié. Le taux de conflit vaut donc T - *FA.Lru*.

La figure 3.1 montre les différentes décompositions des défauts de cache étudiées, pour le programme *Go*, avec un cache à correspondance directe de 32KO et 32 octets par ligne[5]. La figure 3.1(a) montre la décomposition obtenue avec la classification de Hill.

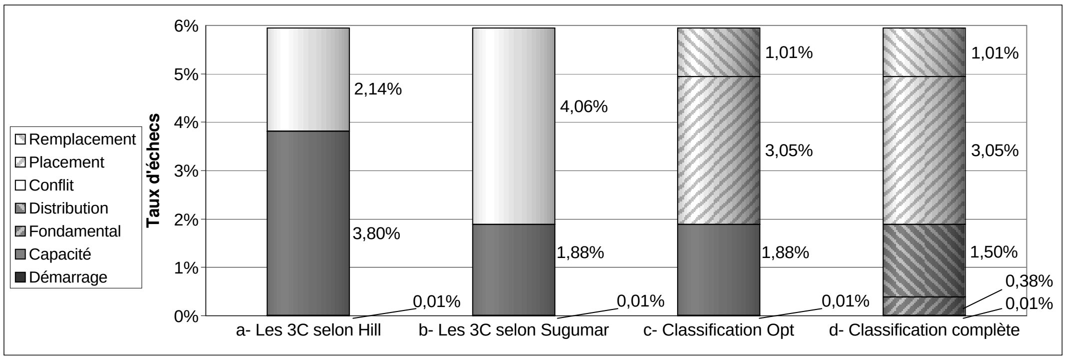


FIG. 3.1 – *Décomposition des défauts de cache pour Go avec plusieurs classifications.*

Le classement de M. D. Hill permet d'évaluer la performance d'un cache par rapport à un cache entièrement associatif de même caractéristique. Plus le taux de conflits est faible, plus la configuration étudiée se comporte comme un cache entièrement associatif et donc plus elle est performante.

4. *FA.Lru* pour *Fully-Associative LRU cache.*
5. Le protocole expérimental est détaillé section 3.6.

3.3 Mesurer précisément les 3Cs

La méthode de mesure des défauts de capacité de M. D. Hill n'est pas tout à fait satisfaisante. En effet, comme la politique LRU n'est pas optimale, les défauts d'un cache LRU entièrement associatif ne sont pas le reflet exact des défauts causés par la capacité limitée d'un cache.

Dans certaines expériences, Sugumar a obtenu avec la méthode de mesure de Hill des taux de conflits négatifs [Sug93] [6]. Cela peut en effet se produire avec certains programmes pour lesquels la politique de remplacement LRU n'est pas performante.

Par exemple, si le cache a une capacité de B blocs mémoire, cela se produit pour un programme qui accède $B+1$ blocs de façon cyclique. En remplaçant l'élément avec la référence la plus ancienne, on remplace en fait le prochain élément référencé. Dans ce cas, même une politique de remplacement aléatoire peut s'avérer plus efficace.

Sugumar a donc affiné la méthode de mesure des défauts de capacité en utilisant Min, la politique de remplacement optimale décrite par Belady [Bel66] qui consiste à remplacer le bloc de l'ensemble qui sera référencé le plus loin dans le futur (voir section 2.3.2.4). Le taux d'échecs de cette configuration est noté *FA.Min*.

Cette méthode de mesure est devenue possible car Sugumar a aussi mis au point *Cheetah*, un simulateur de cache capable de simuler une politique de remplacement optimale avec un temps de calcul relativement proche d'un simulateur LRU.

On peut donc mesurer les taux d'échecs comme suit:

- Le taux de défauts de démarrage est toujours égal à *Cold*.
- Le taux de défauts de capacité vaut *FA.Min* - *Cold*.
- Le taux de défauts de conflits vaut T - *FA.Min*

La figure 3.1(b) montre cette classification des défauts pour *Go*. Il y a une réduction des défauts de capacité par rapport à la figure 3.1(a). Sugumar note qu'avec la méthode de mesure de Hill, le taux de défauts de capacité peut parfois être surestimé de plus de 200%.

Avec la méthode de calcul des défauts de capacité de Sugumar, un cache associatif LRU génère aussi des défauts de conflit, ce qui peut paraître surprenant au premier abord pour un architecte. Sugumar a donc proposé le modèle *OPT* décrit dans la section suivante.

3.4 Le modèle OPT

En mesurant les défauts de capacité avec une politique de remplacement optimale, même les caches LRU entièrement associatifs génèrent des défauts de conflit. Ces défauts de conflit n'ont pas la même origine que les défauts de conflit apparaissant dans les caches à correspondance directe. Il faut donc les différencier.

Sugumar définit deux types de défauts de conflit:

- **Les défauts de placement** [7]: L'adresse du bloc chargé dans le cache ne corres-

6. Voir section 4.1, page 72 de sa thèse.
7. En anglais *Mapping miss*

pond pas à l'ensemble de lignes où se trouve le bloc à remplacer avec la politique optimale.

- **Les défauts de remplacement**: L'adresse du bloc chargé correspond au bon ensemble, mais la politique de remplacement matérielle ne le range pas dans la ligne de cache contenant le bloc à remplacer.

Il faut noter qu'un cache entièrement associatif ne génère pas de défaut de placement. Un cache à correspondance directe peut générer quelques défauts de remplacement, car les blocs mémoire les moins utilisés ne doivent pas être systématiquement chargés dans le cache (bipasse du cache [TFMP95]).

Afin d'évaluer ces deux taux de conflits, Sugumar simule un cache identique au cache à étudier, mais change la politique de remplacement originale pour Min. Le taux de défauts de cette configuration est noté *SA.Min*. T représente aussi le taux de défauts total du cache étudié.

les différents taux de conflits peuvent être mesurés comme suit:

- Le taux de défauts de placement vaut *SA.Min - FA.Min*
- Le taux de défauts de remplacement vaut T - *SA.Min*.

Un exemple de la décomposition des défauts de cache avec la classification OPT est représenté figure 3.1(c) pour le benchmark *go*. Comme le cache étudié est à correspondance directe, les défauts de remplacement qui apparaissent sont dus aux blocs chargés inutilement dans le cache (bipasse du cache).

Cette classification, baptisée *Opt* par Sugumar, permet d'évaluer la performance d'un cache lorsqu'on exécute un programme. Cependant, elle ne suffit pas pour évaluer la qualité des optimisations logicielles du placement des données en mémoire, car changer la localité d'un programme peut non seulement modifier le taux de défauts de placement mais aussi le taux de défaut de capacité.

3.5 Affiner le taux de défauts de capacité

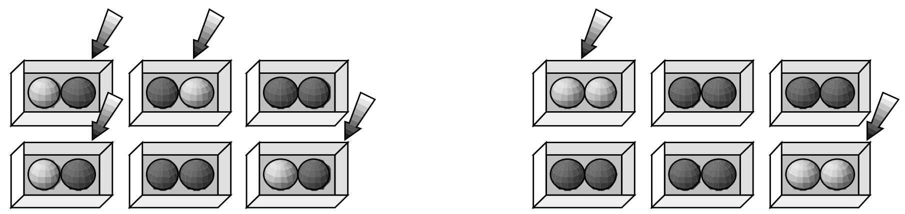
Avec la classification *Opt*, le taux de défauts de capacité représente le taux d'échecs minimum que l'on peut obtenir avec un cache géré de façon optimale, pour une taille de cache donnée avec une taille de ligne donnée et une trace d'adresses donnée. Cependant, pour une même application, il est possible d'améliorer les taux d'échecs du cache, car si l'on modifie le placement des données en mémoire, on change la trace d'adresses générée par le programme.

Il est donc intéressant d'affiner la mesure des défauts de capacité afin de tenir compte de l'impact des optimisations logicielles.

3.5.1 Limites de la classification de Sugumar

Pour une même application et une configuration de cache donnée (C et L fixés), le taux de défauts de capacité peut être modifié en rangeant les données différemment en mémoire.

Pour visualiser simplement cet effet, considérons, figure 3.2, comment organiser des billes dans un nombre quelconque de boîtes. Le but est de transporter les 4 billes claires.



(a) Distribution aléatoire (b) Billes triées

FIG. 3.2 – *Comment distribuer des billes dans des boîtes.*

- Si les billes sont distribuées en aveugle, les billes claires et foncées sont mélangées, figure 3.2(a) : pour transporter toutes les billes claires il faut manipuler au moins 4 boîtes (cela correspond aux défauts de capacité).
- Si les billes sont distribuées dans les boîtes en fonction de leur couleur, figure 3.2(b), il est possible de ne manipuler que deux boîtes (cela correspond aussi aux défauts de capacité, mais pour une meilleure distribution).

Pour un cache, la problématique est similaire[8] : les données sont placées dans les blocs de mémoire selon des critères statiques. Ce sont le programmeur et le compilateur qui influent sur la distribution des éléments dans les blocs mémoire. Les placements se font donc "en aveugle". Ils n'obéissent pas à des contraintes de localité mais à des contraintes de gestion mémoire. Cependant, une distribution avec une mauvaise localité diminue la performance du cache, car le cache doit manipuler plus de blocs pour satisfaire le même nombre de références.

Comme un cache ne manipule que des blocs mémoire, il n'est pas possible au niveau matériel de changer la distribution des données parmi les blocs mémoire, ce qui fixe le taux de défauts de capacité comme le minimum atteignable.

Par contre, avec des optimisations logicielles, on modifie la distribution des données parmi les blocs mémoire. Pour cette nouvelle distribution, le taux de défauts de capacité peut être inférieur à celui de la distribution initiale ; ceci montre les limites des classifications de Sugumar et de Hill.

Ces classifications ont été définies par des architectes de machines pour essayer de comprendre l'impact de paramètres tels qu'associativité, taille de cache ou politique de remplacement sur la performance des caches, mais ne permettent pas de capturer des comportements liés à la distribution des données en mémoire. La classification que proposée ci-dessous à pour but de capturer ces comportements.

8. La simplicité de l'exemple figé choisi est trompeuse. En fait, dans un programme, l'ordre des accès aux données évolue dans le temps. Lorsque le temps entre en compte, il devient difficile de déterminer quelle distribution (statique) offrira la meilleure localité au cours de toute l'exécution du programme (dynamique) : Il faudrait en fait modéliser le problème avec des billes qui changent de couleur au cours du temps.

3.5.2 Défauts fondamentaux et défauts de distribution

Afin de tenir compte de l'impact du placement des données en mémoire sur les défauts de capacité, ceux-ci sont décomposés en défauts fondamentaux et en défauts de distribution.

3.5.2.1 Définitions

Deux types de défauts de capacité peuvent être distingués :

- **Les défauts fondamentaux** : Défauts de cache survenant parce que le cache a une capacité trop faible pour contenir toutes les données manipulées, même lorsqu'elles sont rangées en mémoire de manière optimale.
- **Les défauts de distribution** : Défauts de capacité survenant parce que les donnés sont mal distribuées parmi les blocs mémoire. Cela réduit le nombre de données manipulées qui peuvent être contenues dans le cache simultanément.

Le taux de défauts fondamentaux est influencé par les motifs d'accès mémoires du programme, la taille du cache et la taille des blocs mémoire. Les défauts de distribution sont eux directement dus au placement des données en mémoire.

3.5.2.2 Estimation du taux de défauts fondamentaux

Il n'existe pas d'algorithme simple permettant d'évaluer le nombre exact de défauts fondamentaux d'un programme, car il n'y a pas de manière raisonnable de trouver une distribution optimale des données en mémoire. Cependant il est possible de faire une estimation du nombre de défauts fondamentaux en utilisant les outils de simulation disponibles.

Remarquons que les caches avec des lignes ne contenant qu'un seul élément ne génèrent pas de défauts de distribution. Dans ce cas, les défauts de capacité sont tous des défauts fondamentaux.

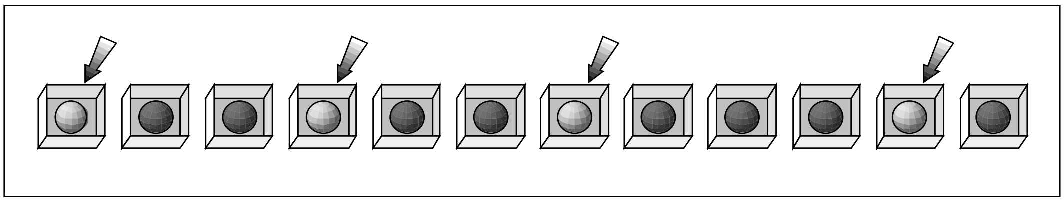


FIG. 3.3 – *Boîtes contenant une seule bille : pas de défaut de distribution.*

En reprenant l'exemple des billes à transporter, on voit bien figure 3.3 que si les boîtes ne contiennent qu'une seule bille, quelle que soit la distribution des billes le nombre de boîtes à transporter, quatre, reste le même. C'est cette propriété que l'on utilise pour estimer le taux de défauts fondamentaux. Si l'on double la capacité des boîtes, il faut porter deux fois moins de boîtes. On retrouve le résultat obtenu figure 3.2(b) lorsque les billes sont triées.

Il faut donc mesurer le taux de défauts de capacité d'un cache de même taille que le cache étudié, mais ayant des lignes larges d'un seul octet. Le taux de défauts de capacité de ce cache est noté *FA.Min.1*.

Les taux de défauts fondamentaux et de distributions sont estimés comme suit :

- On note L la taille de ligne du cache étudié. Le taux de défauts fondamentaux est estimé [9] par $\frac{FA.Min.1}{L}$.
- Le taux de défauts de distribution vaut alors $FA.Min$ - $\frac{FA.Min.1}{L}$.

La mesure qui est faite donne une borne inférieure du taux de défauts fondamentaux réel.

En effet, à taille de cache constante, augmenter la taille des lignes réduit le nombre de lignes disponibles et donc les possibilités de remplacement de blocs. Effectuer une simple division pour estimer le taux de défauts fondamentaux revient à considérer que la réduction du nombre de lignes n'influe pas sur les choix de placement à faire. Plus le nombre de lignes du cache est élevé, plus le taux de défauts fondamentaux réel s'approche de la borne calculée.

Dans les chapitres qui suivent, nous assimilerons (abusivement) le taux de défauts fondamentaux à l'estimation qui en est faire.

Classer les défauts de capacité permet d'évaluer la localité spatiale générée par le programme que le cache parvient à exploiter. Lorsqu'un programme génère peu de défauts de distribution, il est certain que le programme présente une bonne localité spatiale.

3.5.3 Calcul des défauts de démarrage

Afin de rester cohérent avec la définition des défauts fondamentaux, les défauts de démarrage sont redéfinis. Ils doivent être définis en fonction des octets mémoire réellement référencés par le programme plutôt qu'en fonction des blocs de mémoire chargés dans le cache.

- **Les défauts de démarrage** : Ils sont obtenus en divisant le nombre d'octets distincts réellement référencés par le programme par la taille des lignes de cache.

La mesure directe du nombre de blocs mémoire distincts chargés peut donner un taux de défauts de démarrage plus élevé qu'avec notre méthode de mesure, dans le cas où certains octets des blocs chargés dans les lignes de cache ne seraient jamais référencés.

Dans une optique d'optimisation logicielle, on peut envisager de déplacer les octets inutilisés, ce qui modifierait le nombre de défauts de démarrage. Comme les défauts de démarrage sont censés représenter les défauts que l'on ne peut éviter, même avec un cache infini, l'ancienne méthode de mesure devient trop imprécise.

9. On aurait pu évaluer le taux de défauts fondamentaux de façon moins précise par $\frac{FA.Min}{L}$, voir *Cold*, mais on sous estimerait encore plus le taux de défauts fondamentaux.

3.5.4 Une classification Complète des défauts

En complétant la classification OPT de Sugumar, elle-même adaptée des 3C de Hill, la classification des défauts de cache obtenue permet d'évaluer à la fois l'impact des optimisations logicielles et des optimisations matérielles sur la performance des caches.

- Les **défauts de démarrage**[10] : Nombre minimum de blocs qu'il faut utiliser pour exécuter le programme.
- Les **défauts de capacité**: Défauts de cache survenant lorsque le cache est géré de façon optimale, parce qu'il a une capacité trop faible pour contenir tous les blocs manipulés par le programme.
  - **Les défauts fondamentaux** : Défauts de capacité survenant lorsque les données sont rangées dans les blocs mémoire de manière optimale.
  - **Les défauts de distribution** : Défauts de capacité causés par une mauvaise distribution des données en mémoire, empêchant d'exploiter pleinement toute la capacité de stockage des lignes de cache.
- Les **défauts de conflits**: Défauts de cache causé par une mauvaise gestion des lignes de cache disponibles.
  - **Les défauts de placement**[11]: L'adresse du bloc chargé dans le cache ne correspond pas à l'ensemble où se trouve la ligne contenant le bloc à remplacer.
  - **Les défauts de remplacement**: L'adresse du bloc chargé correspond au bon ensemble mais la politique de remplacement matérielle ne le range pas dans la ligne contenant le bloc à remplacer.

3.5.5 Mesure des défauts

Le tableau 3.2 récapitule comment mesurer les défauts de caches de la classification complète. Les abréviations utilisées sont expliquées tableau 3.1.

La redistribution des données en mémoire influe sur le taux de défauts de distribution, car les motifs d'accès aux blocs changent. Elle influe aussi sur les défauts de placement, car les blocs sont rangés à des adresses différentes. Le taux de défauts de remplacement est aussi influencé indirectement, car si les taux de défauts de capacité et de placement diminuent, on fait moins souvent appel à la politique de remplacement matérielle. Elle a donc moins souvent l'occasion de faire des choix de placement erronés. De plus, les blocs ne sont plus rangés aux même adresses, et la politique de remplacement doit manipuler dans chaque ensemble des blocs n'ayant plus les mêmes propriétés d'accès.

Le tableau 3.3 résume le potentiel d'optimisation qu'offrent le placement des données en mémoire. Le tableau montre notamment que, contrairement aux améliorations matérielles, le placement mémoire influe aussi sur les défauts de capacité.

10. En anglais, *compulsory miss*
11. En anglais *Mapping miss*

TAB. 3.1 – *Notations pour le calcul des taux de défauts de cache.*

Notation	Mesure
$Total$	Taux de défauts du cache étudié (taille C, taille de ligne L, associativité A)
$Bytes$	Nombre d'octets mémoire distincts référencés
$LSize$	Nombre d'octets par lignes
$NRef$	Nombre de références mémoire
FA.Min.1	Taux de défauts d'un cache de taille C, avec des lignes de 1 octet, entièrement associatif, avec une politique de remplacement optimale Min
FA.Min.L	Taux de défauts d'un cache de taille C, avec des lignes de L octets, entièrement associatif, avec une politique de remplacement optimale Min
SA.Min.L	Taux de défauts d'un cache de taille C, avec des lignes de L octets, associatif par ensembles de A, avec une politique de remplacement optimale Min

TAB. 3.2 – *Calcul des taux de défauts de cache.*

Type de défaut		Valeur
Démarrage		$\frac{Bytes}{LSize \times NRef}$
Capacité		*FA.Min.L* - $\frac{Bytes}{LSize \times NRef}$
	Fondamentaux	$\frac{FA.Min.1}{LSize} - \frac{Bytes}{LSize \times NRef}$
	Distribution	*FA.Min.L* - $\frac{FA.Min.1}{LSize}$
Conflit		*Total - FA.Min.L*
	Placement	*SA.Min.L - FA.Min.L*
	Remplacement	*Total - SA.Min.L*

La classification permet d'évaluer le potentiel qu'il y a pour optimiser le placement des données des programmes. La classification complète pour *Go* peut être comparée aux autres classifications sur la figure 3.1(d). Les défauts de capacité sont composés en majorité de défauts de distribution. Un placement optimal des données en mémoire peut donc éliminer la majorité des défauts de capacité. Cette information n'apparaît pas avec les autres classifications.

Le taux de défauts total d'une application est souvent très supérieur au taux de défauts fondamentaux. Les techniques de placement des données permettent de réduire les types de défauts prépondérants (défauts de distribution, de placement et de remplacement), et peuvent donc en théorie réduire le taux de défauts du cache au niveau du taux de défauts fondamentaux.

3.6 Expériences

Afin d'étudier la décomposition des taux d'échecs avec la nouvelle classification des défauts de cache, les traces d'exécutions de plusieurs applications ont été utilisées dans

TAB. 3.3 – *Potentiel de la réorganisation des données.*

Type de défaut		placement	matériel
Démarrage		-	-
Capacité	Fondamentaux	-	-
	Distribution	+	-
Conflit	Placement	+	+
	Remplacement	+	+

des simulateurs de caches.

3.6.1 Protocole expérimental

Les traces ont été obtenues sur une machine SGI Onyx, avec des processeurs R10000 (jeu d'instruction mips4), et le système d'exploitation Irix 6.2, en utilisant *Pixie*, *GetCold* et *Cheetah*.

Les programmes ont été compilés avec le compilateur standard (*C MipsPro Compiler*), avec la ligne de commande "cc -Ofast=IP25 -LNO -IPA". Les options de compilation les plus poussées ont été sélectionnées pour maximiser la performance des applications testées (transformations de boucles et analyses interprocédurales). Le code généré est 32 bits (de type *n32*), bien que le processeur supporte du code 64 bits.

Les exécutables ont été tracés avec *Pixie* [Smi91], un outil d'instrumentation de programme fourni avec les outils d'analyse de performance *SpeedShop*. La ligne de commande utilisée est "pixie -idtrace -idtrace_file 9". La trace générée a été stockée sur disque pour garantir que les six simulations que nous effectuons utilisent la même trace.

Les cinq configurations de cache nécessaires pour générer le graphique ont été obtenues avec le simulateur de cache *Cheetah*.

Pour les défauts fondamentaux, il faut simuler un cache avec des lignes de un octet. Pour cela l'interface de *Cheetah* avec *Pixie* a été réécrite pour qu'elle puisse générer des références octet par octet [12]. Les références mémoires sont converties en des références octet par octet, pour les lignes de cache de un octet. Cela entraîne une erreur dans le nombre de références mémoires vues par le simulateur, qui devient égal au nombre de références mémoires multipliées par leurs tailles. Les résultats sont donc corrigés en calculant le taux d'échecs avec le compte des références mémoire obtenus avec d'autres simulations de caches ayant des tailles de lignes supérieures à la taille des données. Comme *Cheetah* retourne un taux d'échecs, le taux d'échecs réel se calcule comme suit :

$$FA.Min.1 = \frac{Taux\ d'echecs \times Nb\ ref.\ 1\ octet}{L \times Nb.Ref.\ reel}$$

GetCold, un petit programme a aussi été mis au point pour obtenir le nombre de défauts de démarrage, car si la trace référence trop d'éléments en mémoire, *Cheetah*

12. *Cheetah*, le simulateur de cache développé par Sugumar à l'origine ne gère pas correctement les caches avec des lignes de taille plus petite que les données référencées.

retourne un nombre de blocs mémoire touchés surestimé. Le calcul des défauts de démarrage est alors faussé.

TAB. 3.4 – *Caractéristiques des applications utilisées.*

nom	Provenance	ligne de commande	$\overline{M}$ d'instr.	$\overline{M}$ de ref.	Trace (MO)
	description				
Go	SpecInt'95	5 21 5stone21.in	335	136	907
	Moteur de jeu de GO.				
M88ksim	SpecInt'95	test	312	116	256
	Simulation du processeur 88100.				
TreeAdd	Olden	20	42	31	350
	Calcul récursif de la somme de valeurs dans un arbre binaire.				
Health	Olden	5 250	31	26	150
	Simulation de la gestion des hôpitaux Colombiens.				
Tsp	Olden	32768	153	40	362
	Résolution du problème du démarcheur.				
Perimeter	Olden	1000 60000	531	114	856
	Calcul du périmètre d'une image rasterisée.				
Raytrace	Splash-2	balls4.env (rés. 8x8)	80	43	209
	Synthèse d'image par lancer de rayon.				
Jigsaw	WPI	random.in, 64x64	101	22	180
	Résolution mathématique d'un puzzle 2D.				

3.6.2 Programmes

Les principales caractéristiques des différents programmes simulés sont décrites table 3.4.

Les programmes sont tirés de différentes suites de *benchmarks* : la suite SpecInt'95 [GHPS93, Dix92], les suites Splash-1 et Splash-2 [SWG94, WOT+95], la suite Olden [RCRH95, CR95, LM96], et la suite WPI [FKJ+90, FKLC92]. Une partie des applications testées ici sont aussi utilisées dans les chapitres suivants. Il est donc intéressant de connaître leur comportement mémoire pour voir l'apport des techniques de placement qui sont utilisées.

3.6.3 Résultats

3.6.3.1 Décomposition du taux d'échecs

La figure 3.4 montre le découpage des défauts de cache de plusieurs applications avec la classification de Hill et la classification complète, pour un cache de 32KO associatif par ensemble de deux et des lignes de 32 octets.

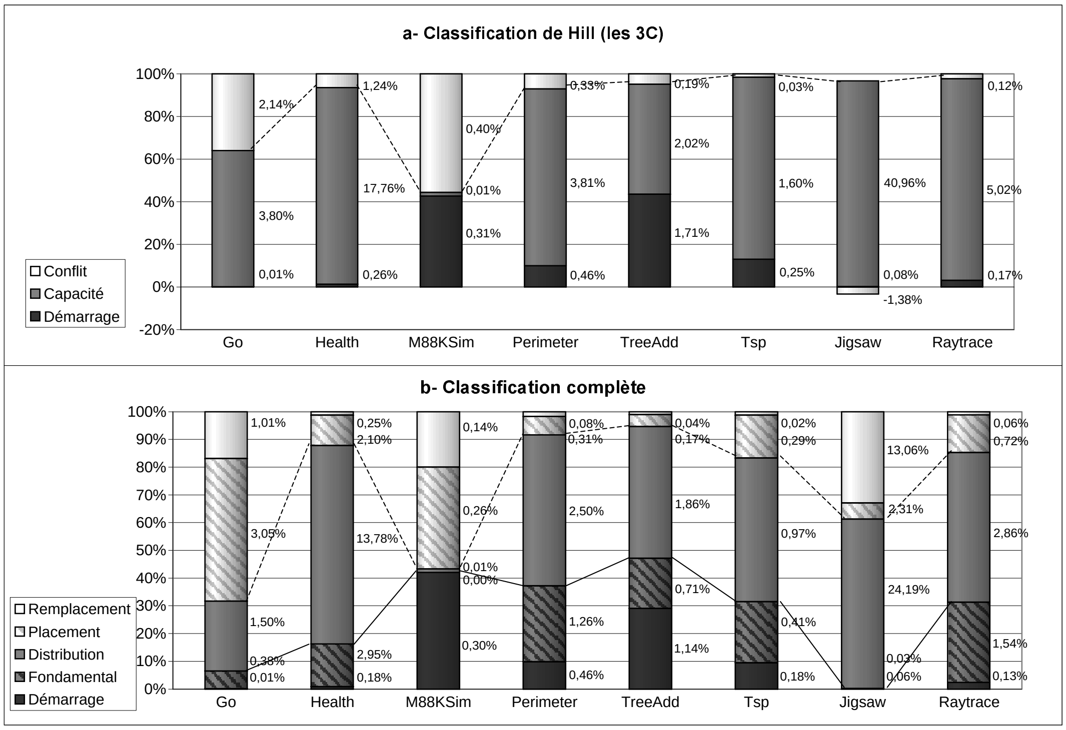


FIG. 3.4 – *Classification des défauts de cache des programmes.*

La classification de Hill surestime très nettement les défauts de capacité par rapport à la classification complète pour *Go, Health, Jigsaw* ou *Raytrace*. Cela peut induire, comme pour *Jigsaw* des taux de défauts de conflits négatif, comme l'avait déjà remarqué Sugumar [Sug93].

Cependant, même avec la classification complète le taux de défauts de capacité reste élevé. Il représente de 30% à 80% de tous les défauts de cache. Quelle que soit la gestion matérielle du cache, idéale ou non, le taux d'échecs des applications testées ne peut pas être réduit plus sans avoir recours à des techniques d'optimisations logicielles. La limite est matérialisée sur la figure par une ligne en pointillés.

Pour la plupart des applications, les défauts de capacité sont en majorité des défauts de distribution (ils représentent jusqu'à 60% des défauts) qui peuvent être éliminés avec un placement optimisé des données dans les blocs mémoires. En utilisant des optimisations logicielles ou en combinant des optimisations matérielles et logicielles, on peut alors envisager d'éliminer la majorité des échecs. La limite est représentée sur la figure par une ligne continue.

La classification complète permet d'affiner l'analyse des défauts de cache. Il devient ainsi possible d'évaluer le potentiel de réduction des défauts de cache avec des techniques d'optimisation matérielles et logicielles.

3.6.4 Influence des paramètres du cache sur les taux d'échecs

La nouvelle classification mise au point permet d'affiner l'étude des défauts de capacité. De plus les méthodes de calcul des défauts de conflit et de démarrage ne correspondent plus aux méthodes utilisées par Hill. Il convient donc d'étudier à nouveau l'influence des paramètres de cache sur le taux d'échecs des programmes.

L'influence sur le taux d'échecs de trois paramètres architecturaux du cache, l'associativité, la taille du cache et la taille des lignes du cache est étudiée. La plus grande variété de comportements est observée lorsque la taille des lignes du cache est modifiée. Cela indique que la localité spatiale et temporelle des programmes est très variable. Il y a donc un fort potentiel de gain à ce niveau avec des techniques logicielles d'optimisation de la localité.

3.6.4.1 Taux d'échecs en fonction de l'associativité

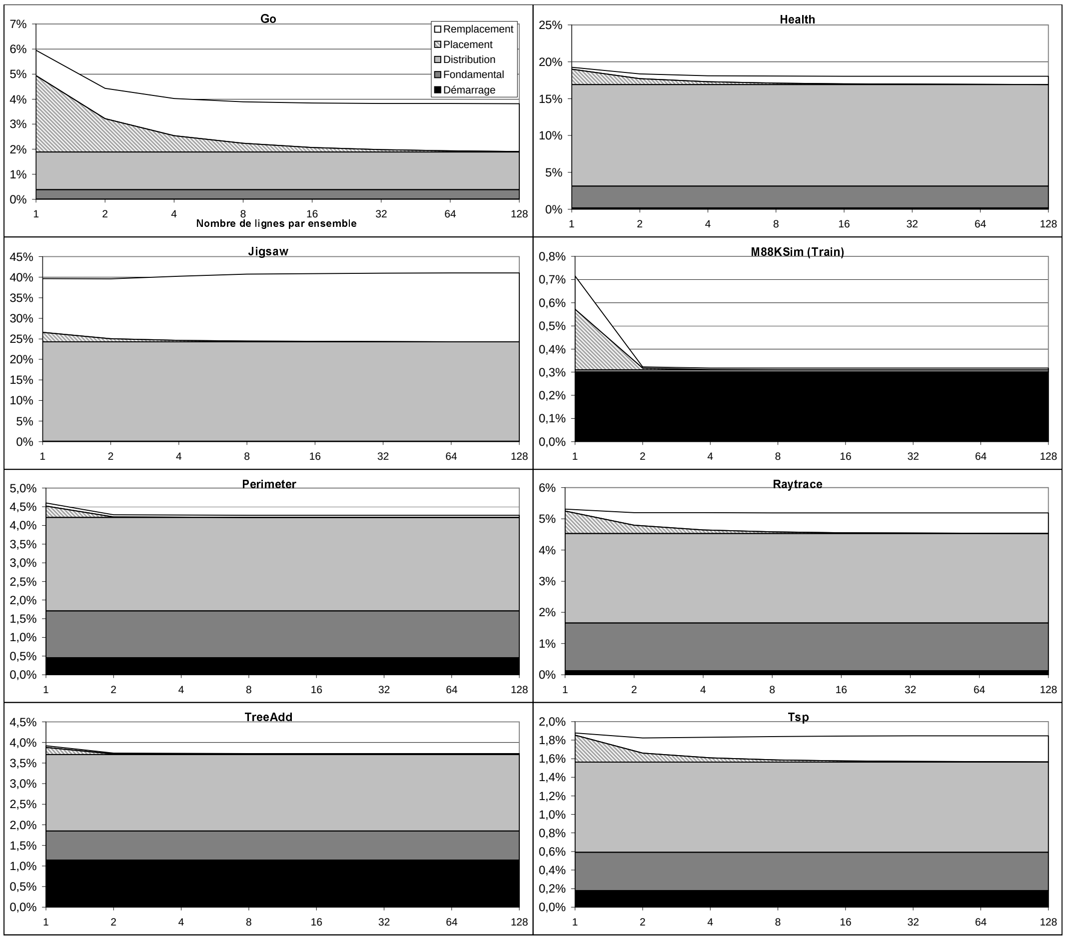


FIG. 3.5 – *L'associativité influe peu sur la performance d'un cache.*

La figure 3.5 montrant les résultats pour un cache de 32KO avec des lignes de 32 octets, indique que l'associativité n'influe quasiment pas sur le taux d'échec total du cache avec les applications testées. On note juste une dégradation lorsque les caches ont une associativité très faible.

Cela s'explique en partie pour les applications testées par le faible pourcentage de défauts de conflit. L'associativité n'influe pas sur les défauts de capacité. Cependant, *Go, Jigsaw, Raytrace* et *Tsp* ont quand même un pourcentage de défauts de conflit non négligeable. En augmentant l'associativité, leurs défauts de placement se convertissent juste en défauts de remplacement. La politique de remplacement LRU n'arrive pas ici à trouver les blocs qu'il faut conserver dans le cache.

Avec un cache de 32KO, augmenter l'associativité au-delà de 2 ou 4 semble ne pas apporter de gain pour les programmes généralistes testés, car les défauts de conflit représentent une petite partie des défauts du cache. De plus, les conflits restant sont difficiles à éliminer avec une politique de remplacement LRU.

3.6.4.2 Influence de la taille de ligne

La taille des lignes du cache influence beaucoup le nombre de défauts du cache. Selon la taille des lignes et selon le programme testé, le cache arrive plus ou moins bien à tirer parti de la localité spatiale du programme. Lorsque les lignes sont trop grandes, les interférences entre les blocs chargés augmentent les défauts de conflits. Lorsque les lignes sont trop petites, le cache tire moins parti de la localité spatiale, ce qui augmente le nombre de défauts.

Il y a une taille de ligne idéale pour chaque programme pour laquelle le taux d'échecs est minimal. A cette taille, il y a à la fois peu d'interférences entre les blocs chargés (donc peu de conflits) et une exploitation maximale de la localité spatiale du programme (donc peu de défauts de distribution). Elle varie de 128 octets à 1KO selon l'application pour un cache de 32KO associatif par ensembles de quatre, comme on peut le voir figure 3.6. Le meilleur compromis semble être une taille de ligne de 128 octets avec les programmes testés. Plus la taille des lignes augmente, plus le cache peut exploiter la localité spatiale du programme. Cependant, quand la taille des lignes augmente, le nombre de lignes du cache diminue. Lorsque la courbe est en forme de cuvette, c'est que le programme n'a pas assez de localité à exploiter, et à un moment donné le gain de localité spatiale ne suffit pas à éviter l'augmentation des interférences dues à la réduction du nombre de lignes.

Charger de petites lignes de cache nécessite moins de bande passante mémoire, ce qui évite de saturer le bus mémoire lorsque des conflits de cache apparaissent. Pour que la bande passante utilisée n'augmente pas quand on augmente la taille des lignes, il faut que le taux d'échecs diminue au moins de moitié lorsque la taille des lignes est doublée. Si par exemple le placement des données permettait de garantir cela, il deviendrait avantageux d'utiliser des grandes lignes de cache. En proportion, il faut moins de bits pour stocker les tags dans le cache. De plus, avoir des grandes lignes de cache permet de mieux exploiter les nouvelles interfaces mémoires comme les SDRAM qui permettent le chargement pipeliné de mots consécutifs en mémoire.

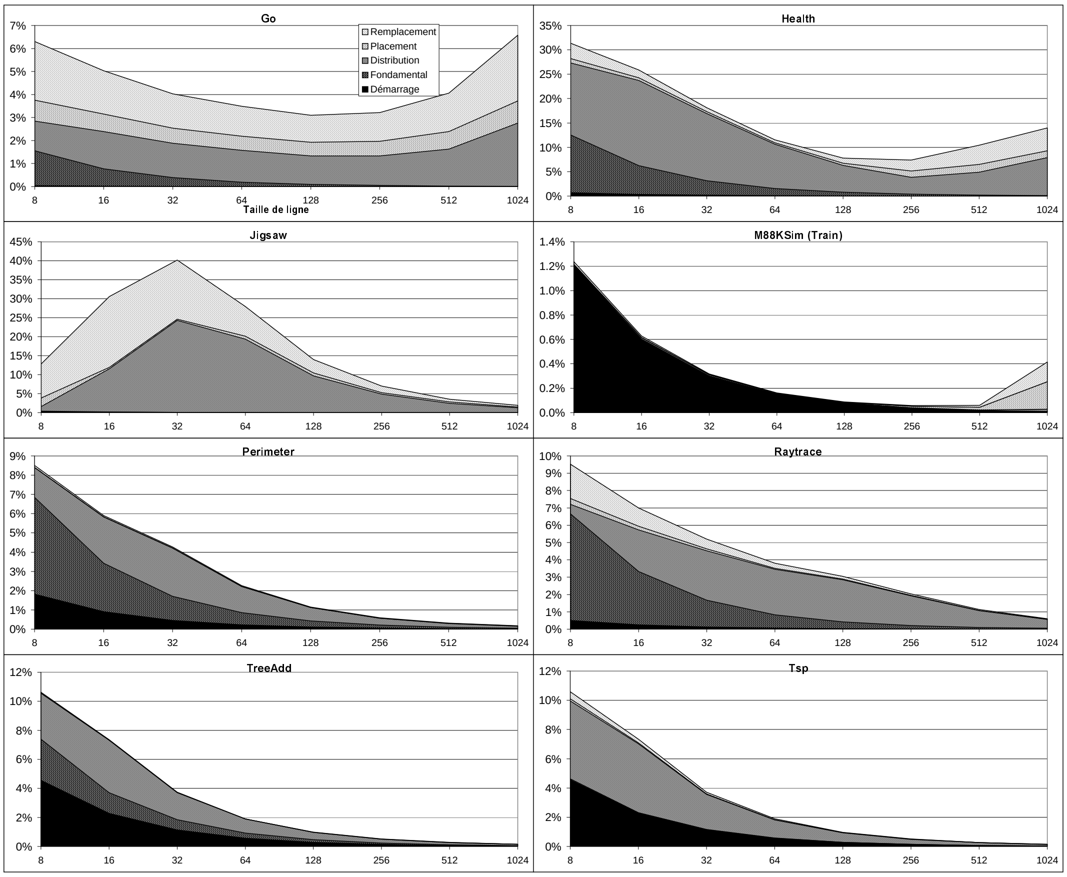


FIG. 3.6 – *Changer la taille des lignes de cache permet de mieux exploiter la localité des programmes.*

Quand le cache n'arrive pas à exploiter la localité, augmenter la taille des lignes peut déteriorer le taux d'échecs. Par exemple, on remarque pour *Jigsaw* que le nombre de défauts augmente quand la taille de ligne passe de 8 à 32 octets, avant de décroître pour les tailles de lignes supérieures. Ici, augmenter la taille des lignes n'apporte aucun gain de localité et augmente les interférences dans le cache. Cet effet est d'autant plus désastreux pour la bande passante mémoire qu'il faut charger des blocs plus grands. Ce problème s'explique au niveau du code source : *Jigsaw* gère une liste chaînée de noeud de 44 octets. Lorsque la liste est parcourue, peu de champs sont utilisés dans chaque instance. Augmenter la taille des lignes de huit à 32 octets permet de charger simultanément plus de champs de chaque noeud, ce qui est inutile s'ils ne sont pas référencés. Avec une taille de plus de 32 octets, les lignes peuvent précharger simultanément plusieurs noeuds. Dans ce cas, le cache peut exploiter la localité spatiale de *Jigsaw* et le taux d'échecs diminue.

En améliorant le placement des données en mémoire, la localité spatiale des programmes augmente, et le préchargement fait par les lignes de cache devient plus efficace.

Utiliser des optimisations logicielles de placement de données permettrait donc de tirer un meilleur parti des grandes lignes de cache, et éviterait les contre performances observées pour les programmes comme *Jigsaw* quand la taille de ligne est augmentée.

3.6.4.3 Taux d'échecs en fonction de la taille du cache

Augmenter la taille du cache n'offre des gains de performance notables que lorsque cela permet de faire tenir le working-set du programme à l'intérieur du cache. Le working-set dépend à la fois de l'application et de son jeu de données. Recourir à un cache plus grand ne garantit donc pas un gain de performance systématique.

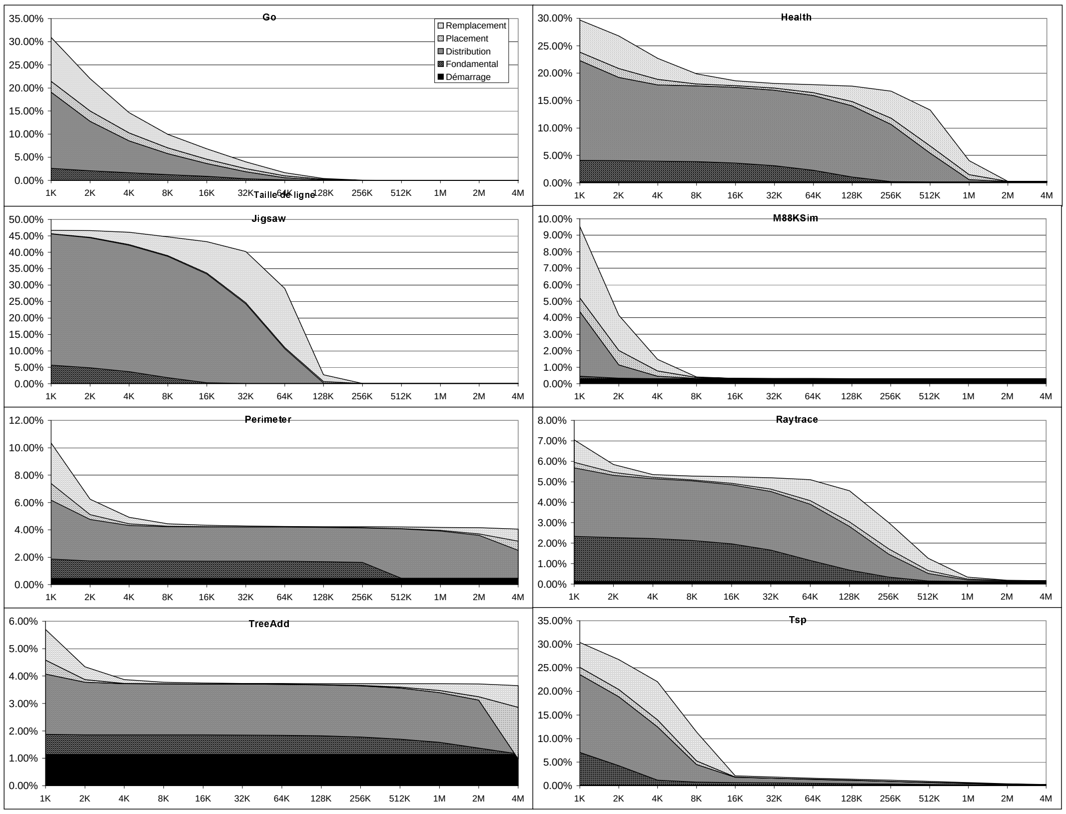


FIG. 3.7 – *Augmenter la taille du cache permet de contenir le working-set de certaines applications.*

Pour *Health, Jigsaw* et *Raytrace* le working-set est très marqué et se situe entre 128KO et 2MO. La caractéristique en est un plateau où le taux d'échecs varie peu, suivi d'une chute rapide en escalier du nombre de défauts quand le cache atteint une taille proche du working-set. On obtient ainsi une courbe en forme d'escalier, comme le montre la figure 3.7 avec les applications citées pour un cache associatif par ensemble de quatre avec des lignes de 32 octets. Une application peut avoir plusieurs niveaux de working-sets se traduisant par plusieurs paliers, comme pour *Health.*

Les autres applications semblent avoir un comportement similaire, bien que moins marqué. *Perimeter* et *TreeAdd* semblent avoir un working-set allant au-delà des tailles de cache testées. *Go, M88KSim* et *Tsp* semblent avoir des working-sets plus petits que les tailles de caches simulées, de sorte qu'il n'est possible d'observer que le bout de "l'escalier".

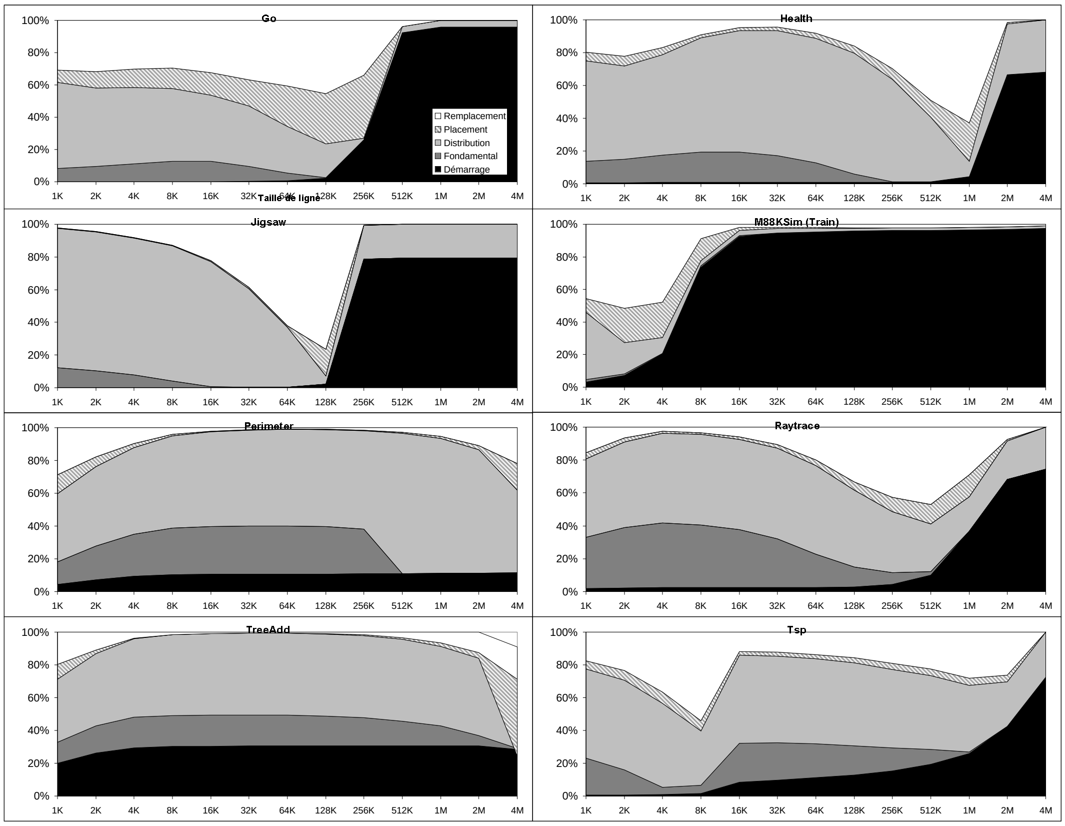


FIG. 3.8 – *La décomposition des défauts de cache change rapidement lorsque le cache atteint la taille du working-set.*

On peut remarquer que les défauts de conflits augmentent lorsque la taille du cache est proche de la taille du working-set et restent faibles sur les plateaux. Cela est très marqué lorsqu'on étudie la décomposition des types de défauts du cache, figure 3.8. Le changement de plateau se voit avec l'augmentation subite de la part des défauts de démarrage (le nombre des défauts de démarrage ne change pas quelle que soit la taille du cache). L'augmentation de la part des défauts de conflit est très nette. Par exemple, pour *Health*, à la transition, quand le cache atteint une taille de 512K à 1MO, jusqu'à 50% des défauts deviennent des défauts de conflit. Les défauts de conflits décalent les changements de paliers. Augmenter l'associativité des caches permet de réduire ces décalages en évitant les conflits.

Les techniques de placement des données en mémoire ont un fort potentiel car elles influent sur les défauts de distribution. Les défauts de distribution comptent pour une grande partie des défauts de cache sur les plateaux. Les éliminer réduirait donc le taux d'échec des caches même lorsque le cache est plus petit que le working-set. De plus, les transitions entre plateaux de plusieurs applications démarrent à des tailles de cache plus faibles avec les défauts fondamentaux qu'avec les défauts de distribution. Eliminer les défauts de distribution réduirait de plusieurs ordres de grandeurs la taille des working-sets d'une partie des programmes. Les optimisations logicielles permettent donc de faire tenir dans un cache plus petit le working-set des applications. Par exemple pour *Health*, les défauts fondamentaux diminuent sensiblement avec un cache de 256KO, alors qu'en ajoutant les défauts de distribution, il faut 1MO de cache. En incluant aussi les conflits, il faut alors non moins de 2MO de cache pour faire tenir le working-set. Cette caractéristique se retrouve aussi avec les autres programmes.

Le working-set des applications est souvent très marqué. Augmenter la taille du cache n'apporte alors un gain de performance significatif que si cela permet de contenir le working-set, sinon le taux d'échecs n'évolue pas car on reste sur le plateau de la caractéristique. Les défauts de conflits augmente lorsque le cache a la même taille que le working-set. Augmenter l'associativité permet de faire plus rapidement la transition vers le palier suivant en les éliminant. Les défauts de distribution sont très nombreux à la fois sur les paliers et lors des transitions entre paliers. Les éliminer avec des optimisations logicielles peut à la fois réduire la taille du working-set, et réduire le taux d'échec du cache quand il ne peut pas contenir le working-set.

3.6.4.4 Conclusion

La nouvelle classification des défauts de cache montre qu'en faisant varier les paramètres architecturaux du cache, on ne change pas nécessairement la performance du cache.

Un cache associatif par ensemble de deux ou de quatre est souvent plus performant qu'un cache à correspondance directe. Cependant, augmenter l'associativité au-delà à peu d'influence, sauf lorsque le cache a une taille proche du working-set du programme.

La taille des lignes influe beaucoup sur la performance du cache. Cependant il est difficile de déterminer la taille de ligne idéale pour tous les programmes. Optimiser le placement des données dans les blocs mémoires permet de générer de la localité spatiale et donc de tirer avantage de lignes de cache plus grandes.

La taille du cache a beaucoup d'influence. Si le cache peut contenir le working-set, il y a peu de défauts, sinon il y a plus de défauts de conflit et de distribution. Lorsque le cache ne peut contenir le working-set, changer sa taille influe peu sur le taux d'échecs. Dans ce cas, il faut essayer d'éliminer les défauts de distribution, par exemple en réorganisant les données en mémoire. Eliminer les défauts de distribution réduit aussi la taille du working-set.

Les paramètres architecturaux du cache influent sur le taux d'échecs. Cependant, une amélioration du matériel a un impact plus aléatoire que n'aurait une optimisation logicielle de la localité spatiale des programmes.

3.7 Conclusion

La nouvelle classification des défauts de cache exposée permet de visualiser l'impact de la localité des références sur la performance du cache.

La classification des *3C* de Hill ou la classification *Opt* de Sugumar visent à analyser la performance de l'organisation physique des caches. La nouvelle classification reste compatible avec celles-ci mais permet en plus d'analyser l'influence du placement des données en mémoire sur le comportement du cache.

Hill a défini dans sa classification les défauts de démarrage, de capacité et de conflit. Sugumar a affiné la méthode de mesure des défauts de capacité proposée par Hill, et subdivisé les défauts de conflits des caches associatifs en défauts de placement et défauts de remplacement.

La nouvelle classification proposée ici subdivise à son tour les défauts de capacité en défauts fondamentaux et en défauts de distribution. Les défauts fondamentaux représentent les défauts qui ne peuvent pas être supprimés en améliorant le placement des données en mémoire. Les défauts de distribution sont des défauts de capacités qui surviennent parce que les données du programme sont mal distribuées dans les blocs mémoire, ce qui réduit la localité spatiale que le cache peut exploiter.

Pour les traces de programmes testées, le taux de défauts de capacité d'un cache associatif par ensemble de deux est souvent du même ordre de grandeur que le taux de défauts de conflit. Pour améliorer la performance des caches, il est donc tout aussi important de réduire les défauts de capacité que de réduire les défauts de conflit.

D'après les expériences, les défauts de capacité sont le plus souvent composés d'une majorité de défauts de distribution. Ceux-ci peuvent être supprimés en adaptant mieux le placement des données en mémoire à l'organisation des caches.

L'étude a aussi évalué l'impact des paramètres du cache sur le taux d'échecs. Il apparaît pour les programmes généralistes testés qu'augmenter l'associativité au-delà de quatre ne réduit pas le taux d'échecs du cache de façon significative. De plus, bien qu'augmenter la taille du cache soit la seule solution matérielle pour réduire les défauts de capacité, cela ne permet de réduire de façon significative le taux d'échecs que s'il devient possible de contenir le working-set dans le cache. Cette approche est donc utile pour les programmes avec de petits jeux de données, mais reste peu efficace pour les programmes manipulant beaucoup de données. Augmenter la taille de ligne permet de tirer parti de la localité des programmes mais peut augmenter les interférences car le nombre de lignes est réduit d'autant. Il est difficile de définir la taille de ligne idéale car les programmes ont des propriétés de localité très différentes.

Les résultats obtenus indiquent qu'il est difficile de trouver une optimisation matérielle qui améliore à coup sûr le taux d'échecs de toutes les applications. Les optimisations logicielles peuvent compléter les optimisations matérielles. En éliminant les défauts de distribution, il devient possible d'exploiter des lignes de cache plus grandes, de réduire le nombre de défauts de cache même lorsque celui-ci ne peut contenir le working-set. De plus, en écartant les données rarement référencées, les optimisations logicielles permettent de réduire le working-set des programmes.

On voit l'importance que pourraient avoir les optimisations logicielles sur l'organisation et la performance des caches. Le chapitre suivant étudie donc plus précisément le potentiel qu'offrent les optimisations logicielles du placement des données, en analysant le cas particulier du placement en mémoire de variables scalaires sans contraintes de placement.

Chapitre 4

Evaluation du potentiel de la redistribution des données en mémoire

4.1 Introduction

Le chapitre précédent a mis en place une classification des défauts de cache pour évaluer l'influence du placement des données en mémoire. Il a été montré qu'un placement optimal des données en mémoire permet d'éliminer tous les types de défauts de cache sauf les défauts de démarrage et les défauts fondamentaux, qui sont en général peu nombreux. Ce chapitre montre par l'exemple qu'il est possible de développer des techniques de placement des données en mémoire qui permettent effectivement d'améliorer le taux d'échecs des caches en améliorant la localité des programmes.

Afin d'éluder le problème posé par les contraintes de placement mémoire des données que génèrent les techniques de gestion mémoire utilisées par les compilateurs (allocation sur la pile de variables locales, allocation dynamique, structuration des données, etc.), l'étude se restreint à une solution de placement automatique de données scalaires sans aucune contrainte de placement [TBS96]. Pour cela, l'expérimentation est faite avec des traces de programmes.

Le but est de montrer qu'il est possible avec un algorithme implémentable automatiquement d'améliorer la localité d'un programme sans aucune aide extérieure du programmeur, afin de réduire à la fois les défauts de capacité et les défauts de conflit.

4.1.1 La redistribution des scalaires

Pour mettre en place un algorithme de redistribution de scalaires indépendants en mémoire qui maximise la localité spatiale du programme exploitée par le cache, il faut définir des critères pour mesurer la localité lors de l'exécution de l'algorithme.

Il faut être capable d'évaluer quels sont les éléments le plus souvent utilisés ensemble. Pour cela une mesure de corrélation est définie. Elle quantifie la proximité temporelle des utilisations de plusieurs données en analysant les motifs d'accès mémoire. Ainsi il est possible d'évaluer le gain de localité que l'on peut obtenir en regroupant ces données.

A l'aide de cette mesure, une heuristique automatique de placement mémoire de scalaires est mise en place. Elle est mise en pratique dans un cadre idéal où le compilateur ne générerait pas de contraintes de placement mémoire pour les données. Pour cela, l'expérimentation utilise des traces de programme, en assimilant tous les accès à la mémoire à des références à des scalaires indépendants.

La redistribution des scalaires se fait en deux étapes.

- Il faut d'abord améliorer la localité spatiale pour améliorer la réutilisation de chaque ligne de cache. Cette optimisation permet de réduire le taux de défauts de capacité en éliminant les défauts de distribution.
- Il faut ensuite améliorer la localité temporelle et évitant les interférences entre les blocs mémoires. Pour cela, il faut que les blocs utilisés en même temps se chargent dans des lignes distinctes. On utilise pour cela la correspondance directe entre *set* et adresse mémoire. Cette optimisation réduit le taux de défauts de conflit en éliminant les défauts de placement.

En pratique un programme référence peu de scalaires en mémoire, car la majorité de ceux-ci sont alloués directement dans les registres. De plus, le compilateur génère des contraintes de placement fortes pour les scalaires restants, qui sont souvent alloués dans la pile. Les résultats présentés ici ne permettent donc qu'une évaluation du gain potentiel de performances que pourraient apporter des techniques de redistribution de données en mémoire.

La section 4.1.2 présente l'état de l'art. La section 4.2 met l'accent sur la difficulté qu'il y a de trouver une distribution optimale des données. La méthode utilisée pour la mesure de la corrélation est décrite section 4.3. La corrélation permet de développer une heuristique de placement des données scalaires indépendantes en mémoire pour des caches à correspondance directe. Celle-ci est décrite section 4.4. Les résultats expérimentaux sont détaillés dans la section 4.5. Ils montrent l'efficacité de cette technique de placement mémoire. Afin de tirer parti de cette technique, la section 4.6 envisage quelques solutions qui pourraient lever les contraintes de placement mémoire des scalaires, ceci avant de conclure le chapitre par la section 4.7.

4.1.2 Etat de l'art

Il n'y a pas eu à notre connaissance, d'autre étude portant sur le placement mémoire des scalaires pour réduire le taux d'échecs du cache.

Afin d'éviter d'accéder à la mémoire lors de l'utilisation de scalaires, les compilateurs modernes s'appuient sur l'allocation des registres du processeur plutôt que sur les caches. Comme un processeur a peu de registres, les efforts portent sur l'amélioration des heuristiques d'allocation de registres [ASU86].

Il s'agit d'exploiter les registres sans temps morts. Pour cela, on définit la vivacité [1] d'une donnée stockée dans un registre. Une donnée est vivante entre une écriture et

1. En anglais *liveness*

une lecture ou entre deux lectures. La donnée est morte si elle est remise à jour à sa prochaine référence. Dans ce cas, elle peut être écrasée sans modifier la validité du programme car la valeur qu'elle contient ne sera plus réutilisée. Le registre la contenant peut donc être réutilisé pour stocker une autre donnée.

Bien que parfois utilisée pour analyser les performances des caches [BGK95, HS95], cette approche n'est pas valable pour les caches. Que la référence soit une écriture ou une lecture, le cache doit éviter les accès mémoires, et un bloc est d'abord chargé dans le cache même si c'est juste pour être modifié. La définition de vivacité n'est donc pas appropriée.

Plusieurs études ont montré que même si l'utilisation de caches améliore les performances des machines, les caches restent extrêmement mal exploités. Burger et al. ont montré que jusqu'à 90% de la capacité de stockage du cache peut être gaspillée, et qu'il est impératif d'améliorer cette utilisation pour avoir des machines performantes [BGK95, BGK96]. Ceci est causé par une mauvaise gestion du cache et par le manque de localité des programmes. Dans une autre étude, Huang et Shen ont montré qu'avec une gestion optimale de la hiérarchie mémoire, il suffit d'une petite capacité de stockage pour obtenir des performances équivalentes à des caches de taille conventionnelle [HS95].

La présente étude sur les scalaires est un premier pas pour tirer mieux parti des mécanismes qui régissent les caches.

4.2 Trouver un placement optimal

Il a été exposé précédemment qu'une distribution optimale des données dans les blocs mémoire permet de réduire à la fois les taux de défauts de capacité et de conflits. Ce chapitre propose de voir dans quelle mesure il est possible de trouver un placement mémoire optimal pour un cache à correspondance directe.

4.2.1 Cas général

Pour trouver un placement optimal des données en mémoire, c'est-à-dire générant le taux de défauts de cache le plus faible, il suffit d'essayer tous les placements mémoire possibles et de mesurer le taux de défauts qu'ils génèrent.

Le nombre de positions à tester en mémoire est limité par la taille du cache, et le nombre de données manipulées est limité. Le nombre de solutions à tester est donc fini.

Bien que théoriquement possible, une telle approche ne peut pas être envisagée concrètement car trop longue.

4.2.2 Utiliser les défauts de capacité

Idéalement, le but de la redistribution des données en mémoire est d'avoir un cache qui ne génère que des défauts de démarrage et des défauts fondamentaux. Est-il possible de trouver les placements mémoires optimum pour lesquels un cache à correspondance directe ne génère que des défauts fondamentaux et des défauts de démarrage?

L'heuristique proposée figure 4.1, permet de trouver pour un cache à correspondance directe toutes les distributions de données qui ne génèrent que des défauts de capacité ou de démarrage, si une telle distribution existe. Cet algorithme n'analyse que les placements mémoires ne générant pas de défauts de conflit, et retourne la distribution optimale qui génère le moins de défauts de capacité. Par définition, ces défauts de capacité sont tous des défauts fondamentaux.

Cet algorithme place les données au fur-et-à-mesure dans les blocs mémoire, de façon à ce que les remplacements faits par le cache correspondent aux remplacements que générerait la politique de remplacement optimale Min [Bel66]. Une donnée ne peut donc être placée que dans un bloc déjà chargé dans le cache ou dans un bloc correspondant au set choisi par Min.

4.2.2.1 L'algorithme

L'algorithme, décrit figure 4.2 teste toutes les possibilités de placement des données dans les blocs mémoires, et de placement de ces blocs dans les lignes de cache, de sorte que les motifs d'accès respectent une politique de remplacement optimale. En cas d'échec d'une configuration donnée ou lorsque la fin de la trace du programme est atteinte, les autres configurations sont testées les unes après les autres (par *backtracking*). Les taux d'échecs des configurations retenues sont comparés, et seule la configuration générant le plus faible taux d'échecs est retenue. Pour cette configuration, le placement mémoire est optimal pour le cache testé.

- Lorsqu'une nouvelle donnée doit être placée, l'heuristique tente d'abord les configurations qui évitent un défaut de cache. La donnée est donc placée successivement dans tous les blocs présent dans le cache, ayant de la place.
- Ensuite, les configurations causant un défaut sont testées. Pour suivre la politique de remplacement optimale, la donnée ne peut être stockée que dans un bloc se chargeant dans la ligne de cache contenant le bloc référencé le plus loin dans le temps. La donnée est donc placée successivement dans tous les blocs se chargeant dans cette ligne ayant de la place pour l'accueillir.
- Finalement, on créé un nouveau bloc pour cette ligne de cache, dans laquelle la donnée est placée.
- Lorsque la donnée référencée a déjà été placée auparavant, on vérifie qu'elle est déjà présente dans le cache ou qu'elle se charge dans la ligne de cache contenant le bloc référencé le plus loin dans le temps. Dans le cas contraire, le placement choisi ne génère pas que des défauts de capacité ; il est abandonné. Une nouvelle configuration est testée en modifiant le placement de la dernière donnée placée.

4.2.2.2 Nécessité de développer une autre méthode de placement

Cet algorithme permet de trouver toutes les configurations ne générant que des défauts de capacité et retourne la configuration avec le taux de défauts le plus bas. On peut ainsi trouver, si elle existe, la distribution qui ne génère que des défauts fondamentaux et des défauts de démarrage pour un cache à correspondance directe.

Il est peu probable en fait qu'une telle configuration des données existe pour un cache à correspondance directe. La présence de défauts de placement et de remplacement est probable. Dans ce cas, cet algorithme ne retourne pas de solution. Cette approche de placement trop ponctuelle n'offre pas de critère évident pour choisir les compromis de placement qui minimiserait les défauts de conflit dans un cache à correspondance directe. Ce n'est donc pas une bonne approche pour distribuer des scalaires en mémoire.

Pour trouver une méthode de placement plus généralement exploitable, il est préférable de prendre du recul par rapport à la mesure des défauts de cache, et d'étudier plutôt les motifs d'accès mémoire, qui sont la cause des défauts de cache.

4.3 Quantification de la localité

Pour avoir un cache performant, il faut que les motifs d'accès mémoires aient de très bonnes propriétés de localité tant spatiales que temporelles.

Les données ayant des références entrelacées doivent si possible être regroupées ensemble dans les mêmes blocs mémoire pour réduire les défauts de distribution. De même, afin de ne pas entrer en conflit, les blocs mémoires ayant des références entrelacées doivent être placés en mémoire de façon à correspondre à des lignes de cache distinctes.

Les notions qui vont permettre de formaliser et de quantifier ces aspects de la localité des références sont introduites ici.

4.3.1 Mesure du temps

Les caches exploitent la localité spatiale et temporelle pour tenter de conserver simultanément toutes les données "*utilisées en même temps*" (données étant référencées répétitivement, dont les références sont entrelacées). Pour cela, il faut définir une notion de temps qui permette de mesurer si deux données sont *utilisées en même temps*.

La politique de remplacement optimale *Min* mesure le temps comme étant le nombre de références séparant la référence mémoire présente et la prochaine référence à la donnée étudiée. Si cette distance est la plus grande, alors *Min* choisit de remplacer la donnée.

Belady remarque que remplacer par erreur un bloc est moins pénalisant si sa prochaine référence est loin dans le futur que si elle est proche [Bel66], car jusqu'à sa prochaine référence, le bloc remplacé ne risque plus d'interférer dans le cache. Faire une analyse à une distance quelconque dans le futur n'est pas pratique pour définir une mesure globale. Notre approche ne consiste pas à trouver un placement générant les remplacements optimaux, mais ceux générant le meilleur placement au meilleur coût.

Il est donc intéressant d'introduire la notion de localité temporelle dans une fenêtre de références W.

Définition : Deux références seront dites W-proches, si l'intervalle de temps les séparant est inférieur à $||W||$, sinon elles seront dites éloignées.

```
ElementSuivant(): rends le scalaire de référencé au temps donné ou ∅ à la fin.
Bloc(): rends le bloc du Layout contenant de ou ∅ sinon.
Ligne(): rends l'ensemble de lignes (le set) auquel un bloc appartient.
Cache(): rends la liste des blocs du Layout présents dans le cache au temps donné.
GetBelady(): rends l'ensemble des blocs du Layout référencés le plus loin dans le futur.
SizeOf(): rends le nombre d'octets qu'occupent le(s) de(s).
NewBloc(): crée un nouveau bloc apparteant à l'ensemble donnée et contenant le de donné.
Layout: Liste des blocs en cours d'utilisation, chaque bloc contient une liste de de(s)

BestLayout = PLACE(0, ∅)
PLACE(t, Layout) {
    de = ElmentSuivant(t, ProgramTrace)
    Si de == ∅                                  // fin, layout conforme à Min
    Sinon Si Bloc(Layout, de) ∈ Cache(Layout, t)  // de déjà alloué, Hit
        Layout =PLACE(t + 1, Layout)
    Sinon Si Bloc(Layout, de) ≠ ∅                 // de déjà alloué, défaut de cache
        Si Ligne({Bloc(Layout, de)}) ⊂ Ligne(GetBelady(Layout, t))
            Cache(Layout, t) = Cache(Layout, t) ∪ Bloc(Layout, de)−
                {b | (b ∈ Cache(Layout, t)) ∧ (Ligne({b}) == Ligne({Bloc(Layout, de)})}
            Layout =PLACE(t + 1, Layout)
        Sinon                                   // Conflit : annuler, essayer un autre layout
            Layout = ∅
        Fin Si
    Sinon                                       // de non alloué: tenter tous les layouts
        BetterLayout = ∅
        Pour Tous B ∈ Cache(Layout, t)          // Essayer les blocs dans le cache (Hit)
            Si SizeOf(B) + SizeOf(de) ≤ BlocSize
                B = B ∪ {de}            // note: l'offset du de dans B n'influe pas
                TempLayout =PLACE(t + 1, Layout)
                SiMissCount(TempLayout) < MissCount(BetterLayout)
                    BetterLayout = TempLayout
                B = B − {de}
            Fin Si
        Repéter
        // Essayer les blocs des lignes pouvant faire des défauts (Miss)
        Pour Tous B ∈ {b | (b ∈ Layout) ∧ (b ∉ Cache(Layout, t))∧
            (Ligne({b}) ⊂ Ligne(GetBelady(Layout, t))}
            Si SizeOf(B) + SizeOf(de) ≤ BlocSize
                B = B ∪ {de}
                TempLayout =PLACE(t + 1, Layout)
                SiMissCount(TempLayout) < MissCount(BetterLayout)
                    BetterLayout = TempLayout
                B = B − {de}
            Fin Si
        Repéter
        Pour Tous L ∈ Ligne(GetBelady(Layout, t)) // Créer nouveau bloc (Miss)
            B = NewBloc(de, L)
            TempLayout =PLACE(t + 1, Layout ∪ {B})
            SiMissCount(TempLayout) < MissCount(BetterLayout)
                BetterLayout = TempLayout
        Repéter
        Layout = BetterLayout
    Fin Si
return Layout }                                 // renvoyer un layout conforme à Min ou ∅
```

FIG. 4.1 – *Algorithme de recherche d'un placement optimal*

Choisir la taille de la fenêtre est un problème de discrétisation du temps. Avec une fenêtre trop étroite, certains entrelacements ne seront pas repérés. Cela n'est important que s'ils influent de façon significative sur la performance du cache. Si la fenêtre est trop grande elle ne rend plus compte de façon efficace de la proximité des références : pour évaluer correctement la localité temporelle, il faudrait alors avoir recours à une mesure de type Min à l'intérieur même de la fenêtre, mais ceci ne sera pas mis en pratique ici.

Lorsqu'un bloc est chargé dans le cache et que certains de ses éléments ne sont pas référencés, il y a gaspillage de l'espace utilisé pour les stocker, et gaspillage de la bande passante mémoire utilisée pour les charger.

Si des éléments avec des références W-proches sont placés dans le même bloc, ils ont plus de chance d'être préchargés utilement dans le cache que si leurs références sont lointaines. Il y a moins de risque que le bloc soit remplacé avant que tous les éléments soient utilisés. Dans ce cas, le cache bénéficie mieux de la localité spatiale entre les données du bloc.

4.3.2 Activité d'une donnée

Pour décrire qu'un élément est ou non référencé dans l'intervalle de temps W, il faut définir l'activité de cette donnée.

Définition : un élément X est actif à un temps t, si l'instant t est compris entre deux références à X qui sont W-proches l'une de l'autre.

l'**activité** de X au temps t est notée $A^W(X,t)$. Elle vaut 1 si X est actif, et 0 sinon.

Si une donnée est active à un instant t, alors il est souhaitable que celle-ci soit stockée dans le cache, pour réduire le nombre de défauts de cache.

Définition : Le prédicat **Ref(X,t)** est défini tel que

$Ref(X,t) = 1$ si X est référencé au temps t, et

$Ref(X,t) = 0$ si X n'est pas référencé au temps t.

Cela permet donc de noter:

$A^W(X,t) = 1$ *ssi*

$\exists(t', t")|(|t" - t'| \leq ||W||) \wedge (t \in [t', t"]) \wedge (Ref(X,t') = 1) \wedge (Ref(X,t") = 1)$

$A^W(X,t) = 0$ sinon.

Pour exploiter au maximum la localité spatiale des lignes de cache, il est préférable de regrouper dans les blocs des données qui sont souvent actives ensemble, et rarement actives séparément. Dans ce cas, si l'une des données est utilisée, l'autre a de fortes chances d'être référencée aussi.

4.3.3 Corrélation entre deux données

La mesure de la corrélation entre deux données définie ici doit mesurer la localité spatiale qui est obtenue en regroupant deux données dans le même bloc mémoire.

Définition : la **corrélation** entre deux données X et Y, notée $C^W(X,Y)$, est égale au total des références faites à ces données lorsqu'elles sont toutes les deux actives, moins le compte des références à ces données lorsque l'une d'elles est inactive.

En utilisant les notations précédentes, cela peut s'écrire:

$$C^W(X,Y) = \sum_{t=0}^{t=t_{fin}}((A^W(X,t) \times Ref(Y,t)) + (A^W(Y,t) \times Ref(X,t)))-$$
$$((1 - A^W(X,t)) \times Ref(Y,t) + (1 - A^W(Y,t)) \times Ref(X,t))$$

Un placement optimal des données est donc une distribution pour laquelle les données les plus corrélées sont stockées dans le même bloc mémoire, et pour laquelle les blocs mémoires les plus corrélés sont placés de façon à correspondre à des sets distincts.

Choix de $||W||$: Il influe sur les mesures d'activité et de corrélation.
Si $||W||$ est petite, les données seront actives sur un grand nombre de petits segments de temps. Ils auront donc tendance à moins se recouvrir. La corrélation entre deux éléments sera donc plus faible, car on négligera les effets de références plus lointaines. Si $||W||$ est trop petite, les éléments ne seront plus corrélés.
Si $||W||$ est grande, l'activité des éléments s'étend sur quelques grands segments de temps. Les données seront donc plus fortement corrélées, car il sera plus facile de trouver des intersections entre segments. Si $||W||$ est trop grand, la corrélation entre deux éléments se simplifie en la somme des références aux deux éléments, on perd les informations de localité.
Avec la mesure de corrélation, on ne choisit pas les éléments à conserver ensemble en regardant lequel est référencé le plus tôt (contraposé de la méthode utilisée pour la politique de remplacement Min), mais on élimine les éléments référencés trop loin dans le temps, et on sélectionne l'élément référencé le plus souvent pour maximiser la localité temporelle de la ligne.

4.4 Heuristique de placement mémoire

A l'aide des mesures de localité définies dans le chapitre précédent, il devient possible de développer une heuristique pour optimiser le placement mémoire de scalaires qui tire parti des informations fournies par un *profil* d'exécution des programmes.

L'heuristique exposée ici permet de redistribuer des scalaires n'ayant aucune contrainte de placement. Cette heuristique suppose un cache à correspondance directe, mais peut facilement être adaptée pour des caches associatifs par ensemble.

$NewBloc(), NewLine(), NewLineList()$: créent un nouveau bloc, ligne ou liste de lignes et réinitialise leur *Activité*.
$SizeOf()$: donne l'espace d'un élément qui est utilisé pour stocker le(s) de(s).
$Correlate()$: calcule la Corrélation de 2 éléments ou $\int_t A^W dt$ d'un élément si le second terme est $\emptyset$.

```
BlocList DISTRIBUTION_DES_SCALAIRES_DANS_LES_BLOCS(deList)
    BlocList = ∅                                  // Liste des blocs contenant des scalaires
    While deList ≠ ∅
        B = NewBloc()
        BlocList = BlocList ∪ {B}
        While SizeOf(B) < BlocSize
            Bestde = ∅
            For all de ∈ deList
                If Correlate(de, B) ≥ Correlate(Bestde, B)
                    Bestde = de
                End If
            Loop
            B = B ∪ {Bestde}                      // Cela modifie l'Activité pour B
            deList = deList − {Bestde}
        Loop
    Loop
return BlocList

LineList DISTRIBUTION_DES_BLOCS_DANS_LES_LIGNES(BlocList)
    LineList = NewLineList()                      // Liste de listes de blocs
    For i = 1 to CacheLineCount
        L = NewLine()                             // Crée la liste de blocs d'une ligne
        LineList = LineList ∪ {L}
    Loop
    While BlocList ≠ ∅
        BestBloc = ∅
        For all B ∈ BlocList
            If Correlate(B, LineList) ≥ Correlate(BestBloc, LineList)
                BestBloc = B
            End If
        Loop
        BestLine = ∅
        For all L ∈ LineList
            If Correlate(L, BestBloc) ≤ Correlate(BestLine, BestBloc)
                BestLine = L
            End If
        Loop
        BestLine = BestLine ∪ {BestBloc}          // Change l'Activité
        BlocList = BlocList − {BestBloc}
    Loop
return LineList
```

Le calcul d'adresse des scalaires se fait simplement en utilisant les contraintes de placement exprimé par $LineList$.

FIG. 4.2 – *Heuristique de placement mémoire des scalaires*

4.4.1 L'algorithme

L'algorithme, décrit figure 4.2 se décompose en quatre étapes :

- Calcul de l'activité de chaque donnée pour toute la durée de la trace.
- Redistribution des éléments les plus corrélés en blocs. Pour cela, pour chaque bloc, on insère de façon répétitive le scalaire le plus corrélé[2] avec le bloc. On recalcule l'activité du bloc jusqu'à ce qu'il soit complet. On recommence en créant un nouveau bloc jusqu'à ce qu'il n'y ait plus de scalaires à placer.
- Distribution des blocs les plus corrélés dans des lignes de cache distinctes. On recherche le bloc le plus corrélé aux blocs déjà placés dans le cache. Le bloc choisi est placé dans la ligne de cache à laquelle il est le moins corrélé. On met à jour l'activité de la ligne et du cache puis on recherche un autre bloc.
- Le placement des blocs en mémoire est simplement fait en assignant les blocs dans des zones libres correspondant à la ligne dans laquelle ils doivent se charger (on suppose une correspondance directe). On fixe aussi de façon arbitraire le placement des scalaires à l'intérieur des blocs, car leur position n'influe pas sur le taux d'échecs du cache.

Le placement mémoire obtenu utilise un nombre de blocs mémoire minimal pour satisfaire les motifs d'accès mémoire du programme. Ce placement réduit le risque d'avoir deux blocs utilisés simultanément en conflit pour la même ligne de cache.

Cette heuristique trouve toujours un placement mémoire efficace. Cependant elle reste coûteuse à mettre en oeuvre. Elle ne peut donc pas être utilisée telle quelle dans un compilateur s'il faut calculer la position relative d'un grand nombre de données en mémoire.

- Durant l'analyse, il faut stocker l'activité de chaque scalaire à corréler. Cela est d'autant plus coûteux que la trace est longue. Si la trace dure T références, et qu'il faut placer n données, il faut stocker $n \times T$ bits d'activité. Un calcul de corrélation est en $o(T)$. Il coûte T calculs élémentaires décrits en section 4.3.3.
- Le placement des scalaires dans les blocs est en $o(n^2)$. Il nécessite de faire $\frac{n^2}{2k}$ calculs de corrélation entre les éléments à placer et les blocs à remplir, où k est le nombre d'éléments par lignes de cache (environs 8 pour une ligne de 32 octets).
- Le placement des blocs en mémoire est en $o(n^2)$. Il nécessite de faire $\frac{n^2}{2k^2}$ calculs de corrélation pour corréler les blocs au cache, plus $l \times \frac{n}{k}$ pour corréler les blocs choisis aux l lignes de cache (256 lignes si le cache fait 8KO).
- L'heuristique ne prend pas en compte les contraintes de placement du compilateur (variables locales, statiques...) ni la structuration des données en types plus complexes (elle ne s'applique qu'aux scalaires).

Pour tenir compte des différentes contraintes de placement, il faudrait faire une analyse plus complexe afin d'adapter l'heuristique.

2. Cependant, si l'élément ne tient pas dans le bloc, il est ignoré.

L'ajout de contraintes de placement externe réduit le potentiel qu'offrent les techniques de redistribution.

Cependant, l'ajout de contraintes peut aussi faciliter la redistribution. Avec moins de choix possible, on élimine le calcul d'un grand nombre de corrélations (par exemple, dans une structure, il n'y a que quelques champs qui peuvent être redistribués les uns par rapport aux autres, dans ce cas n est très petit).

Les caches actuels sont déjà relativement grands (32KO pour un cache L1, 1MO pour le cache L2 en moyenne). Ils ont donc une grande tolérance de placement.

De plus les contraintes de placement du compilateur ou de typage reflètent souvent des propriétés de localité sur l'utilisation des données. C'est bien ces contraintes de placement qui sont déjà exploitées par les caches pour capturer la localité des références (organisation des données en structures, pile, zone de variables locales). La faible performance des caches pourrait en fait venir d'une faiblesse de ces contraintes lorsque le nombre de données devient élevé, lacune que des solutions de placement combleraient.

Pour montrer que des heuristiques de placement de données peuvent être implémentées (sur des traces), un programme utilisant une version simplifiée de l'algorithme a été écrit. Il a été testé sur des traces de programmes pour montrer la viabilité de cette approche.

4.5 Expérimentation

Une version simplifiée de l'heuristique de placement des scalaires a été mise en oeuvre. Elle a été expérimentée sur les traces de plusieurs programmes.

4.5.1 Mise en place

Les traces ont été obtenues à partir d'un exécutable avec *spy* [Irl92], un outil de génération de traces fonctionnant sous SunOS4.

La classification complète des défauts de cache a été calculée pour des caches à correspondance directe, ainsi que des caches LRU associatifs par ensembles de deux et par ensembles de quatre, de 16KO, les lignes étant de 32 octets.

Pour cela, plusieurs outils ont été utilisés.

- L'outil *Belady* a été écrit pour donner le taux d'échecs d'un cache parfait (un cache entièrement associatif, utilisant la politique de remplacement optimale *Min*). Cet outil fournit les taux de défauts de capacité, ainsi que le taux de défauts fondamentaux en simulant des lignes d'un octet.
- *CacheSkew* et *NewSim* sont deux simulateurs de cache écrits pour fournir les taux d'échecs des caches à correspondance directe, ainsi que des caches 2 et 4-associatif par ensemble.
- *DataLayout* est l'outil qui optimise le placement des données de la trace.
- *SpannerFork* est un outil qui permet de remplacer les adresses dans la trace par les adresses sélectionnées par *DataLayout*.

L'approche basée sur la simulation de trace corresponds aux choix simplificateurs qui ont été fait : cela permet de faire abstraction des contraintes de placement mémoire du compilateur. Dans une trace, toutes les références sont perçues comme des références à des scalaires indépendants. Les adresses du flot de références mémoire de la trace peuvent donc être modifiées à loisir par l'heuristique.

Les résultats ne sont donc pas une projection réaliste de l'efficacité du placement de données qui pourrait être fait par un compilateur. Cependant, le but ici n'est pas de fournir un outil à insérer dans un compilateur, ni de voir quelle est l'efficacité réelle des techniques de placement qui peuvent être implantés dans un compilateur, mais bien de voir le potentiel qu'offre la distribution de données.

La lenteur de simulation a limité la taille des traces à quelques millions de références mémoire.

4.5.2 Simplification de l'heuristique

Le programme *DataLayout* utilise une version simplifiée de l'heuristique de manière à réduire la consommation mémoire et le temps d'exécution.

- Une fenêtre de temps fixe est utilisée au lieu d'une fenêtre glissante. Cela revient à discrétiser le temps en unités de $||W||$ références. Les références appartenant à deux fenêtres différentes sont considérées comme éloignées. Cependant, dans certain cas avec une fenêtre mobile, elles auraient été considérées comme proches (par exemple, si $t_i \in W_1\ et\ t_j \in W_2\ et\ |t_i - t_j| < ||W||$).
- Un seul bit par fenêtre est utilisé pour dire si un scalaire est actif/référence dans la fenêtre de temps.
  $A^W(de,t) = Ref(de,t) = A^W(de,W_i) = 1\ si\ (t \in W_i) \wedge (\exists t' \in W_i | Ref(de,t') = 1)$
  On réduit ainsi la consommation mémoire à 1 bit par scalaire par fenêtre. Par contre, parmi 2 scalaires actifs dans une fenêtre, le programme n'est pas en mesure de dire lequel est le plus référencé (il aurait fallu plus de bits pour compter le nombre de références dans la fenêtre et calculer plus précisément la corrélation). A cause de cette approximation, si la fenêtre est trop large on ne peut pas distinguer quels sont les éléments les plus corrélé entre eux. Ils paraissent tous autant corrélés. *Notre programme considère comme non corrélés des éléments lointains, mais n'utilise pas la mesure de temps de Min pour calculer la corrélation entre éléments proches.*
- Le calcul de corrélation est réduit à un ET logique entre les bits d'activité.
  $C^W(X,Y) = \sum_{W_i=0}^{W_i=W_{end}} (A^W(X,W_i) \wedge A^W(Y,W_i))$
  Cela revient à regrouper les scalaires souvent référencés ensemble, sans tenir compte du cas où les scalaires sont aussi souvent référencés séparément.
- Pour éviter de recalculer l'activité des blocs lorsqu'on distribue les scalaires dans les blocs, l'activité d'un bloc est assimilée à l'activité du premier élément qui a été mis dans le bloc. Le premier élément mis dans un bloc est toujours l'élément le plus actif et le plus référencé disponible à la création du bloc, ce qui réduit l'erreur induite.

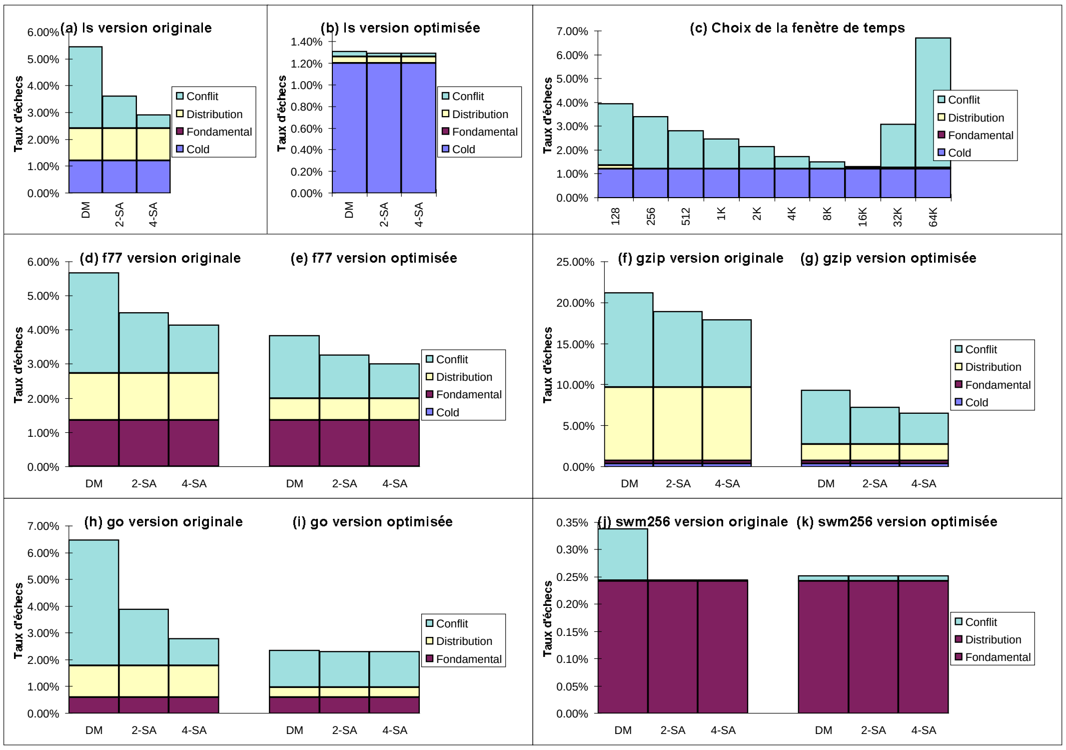


FIG. 4.3 – *Résultats expérimentaux du placement de scalaires.*

4.5.3 Trace de ls

Notre première trace optimisée est celle de la commande Unix *ls*. La trace est composée de 102 614 références mémoire correspondant à 16 149 éléments distincts, pour un total de 39 471 octets (soit 1,20% de défauts de démarrage).

Cette première expérience donne déjà quelques informations :

– L'étude sur cette petite trace a permis de déterminer empiriquement la taille de fenêtre $||W||$ à utiliser. La taille de la fenêtre influe sur le placement mémoire obtenu, comme le montre la figure 4.3(c).
  Modifier la taille de la fenêtre a peu d'effet sur les défauts de capacité. Il semble que la fenêtre n'a pas besoin d'être large (256 références) pour permettre de choisir comment placer les données dans les blocs. Elle doit rester suffisamment petite ($<$ 8K références) pour différencier quelles données doivent être en même temps dans le cache. Comme le cache est relativement grand (16KO), il y a de la tolérance pour placer les données de façon à ce qu'elles tiennent dans le cache simultanément.
  La taille des fenêtres a plus d'impact sur le placement des blocs en mémoire. Si la fenêtre est petite, des conflits apparaissent parce que l'heuristique n'arrive pas à voir que des blocs sont corrélés (il suffit de références répétitives séparées de plus de $||W||$). Avec une fenêtre trop grande, l'approximation de l'heuristique qui est

faite rends impossible la discrimination d'une faible et d'une forte corrélation à l'intérieur d'un même segment de temps.
Une taille de $||W|| = 16K$ références semble être le meilleur compromis, car la fenêtre utilisée est la même pour le placement des scalaires et des blocs. Cette valeur sera utilisée pour toutes les expériences[3].

- Ici, le taux de défauts de démarrage est égal au taux de défauts fondamentaux (1,20%). Avec cette petite trace on peut donc espérer trouver une distribution mémoire ne générant quasiment que des défauts de démarrage, alors que le taux de défauts de capacité de la version originale est de 2,41%.

- En optimisant le placement mémoire, on améliore de 48% le taux de défauts de capacité (somme des défauts fondamentaux et des défauts de distribution). Il passe de 2,41% à 1,26%, figure 4.3(a,b). Cela est dû à la réduction de 95% des défauts de distribution qui passent de 1,21% à 0,06%.

- Au total, le taux d'échec du cache à correspondance directe est amélioré de 76%. Il passe de 5,46% à 1,31%, figure 4.3(a,b). La plupart des défauts sont en effet supprimés et on se rapproche effectivement très près du nombre de défauts de démarrage. Le taux d'échec du cache à correspondance directe avec le placement optimisé est bien plus faible que le taux de défauts de capacité obtenu avec le placement original (2,41%). Dans ce cas utiliser une optimisation logicielle s'avère bien plus efficace que toute optimisation matérielle, comme l'utilisation d'un cache entièrement associatif avec une politique de remplacement optimale.
Contrairement aux optimisations matérielles, on ne peut pas borner le gain qu'il est possible d'obtenir avec des optimisations logicielles, par le taux de défauts de capacité. La limite inférieure est réduite au taux de défauts fondamentaux.

- Les caches associatifs bénéficient aussi de l'amélioration du placement des données. Le cache 4-associatif par ensemble gagne 56%, son taux d'échec passe de 2,92% à 1,29%, figure 4.3(a,b). On remarque que pour cette trace référençant peu de données, la distribution est suffisamment efficace pour rendre inutile l'utilisation d'un cache associatif. Le taux d'échec du cache à correspondance directe (1,31%) est en effet déjà proche du taux de défauts fondamentaux (1,20%), lui-même égal au taux de défauts de démarrage.

L'heuristique réussit ici à réduire le taux de défauts du cache à correspondance directe de 76%. Un cache parfait avec une distribution optimale éliminerait tous les défauts de conflit et de distribution réduirait de 78% le taux d'échecs. Le résultat obtenu pour cette trace est donc proche de la perfection, même avec un cache à correspondance directe. Ce résultat est bien meilleur que toute autre solution matérielle qui ne peut au mieux qu'éliminer tous les défauts de conflit.

3. D'autres tailles de fenêtres peuvent peut-être fournir de meilleurs résultats pour les autres applications, mais les résultats obtenus sont déjà suffisamment probants avec des fenêtres de 16K références.

4.5.4 Trace de swm256

Une trace de 5 millions de références a été générée pour le benchmark 078.swm256 de la suite Spec92. Il est compilé avec gcc -O4 sous SunOS4. Les résultats sont exposés figure 4.3(j,k).

La trace de ce programme scientifique exploite pleinement la hiérarchie mémoire. Le taux d'échec est quasiment égal au taux de défauts fondamentaux (0,242%). Bien que les accès aux matrices ne génèrent pas d'interférences dans cette trace, quelques conflits (0,095%) apparaissent sur le cache à correspondance directe.

Notre heuristique se comporte bien : elle ne dégrade pas de façon significative la performance du cache, ce qui indique qu'elle trouve un placement mémoire à peu près équivalent au placement mémoire original que génère automatiquement l'organisation des éléments en matrices. Elle arrive même à éliminer les quelques conflits qui apparaissent dans le cache à correspondance directe.

4.5.5 Trace de f77

Une trace de l'exécution du compilateur f77 disponible sous SunOS4 de deux millions de références a été générée, alors qu'il compilait swm256, figure 4.3(d,e). Les deux premiers millions de références ont été au préalable éliminés pour éliminer l'effet de démarrage du programme.

Les défauts se répartissent équitablement en défauts de capacité et en défauts de conflits. Les défauts de capacité sont pour 51% des défauts de distribution qui peuvent être éliminés avec un placement mémoire optimal.

L'heuristique arrive à éliminer 53% des défauts de distribution, représentant 27% des défauts de capacité. Pour le cache à correspondance directe, l'heuristique réduit aussi les défauts de conflit de 38% (de 2,95% à 1,83%).

Au total, en jouant sur les défauts de capacité et de conflits, la redistribution élimine 33% des défauts du cache à correspondance directe, et 27% pour le cache 4-associatif (de 4,15% à 3,01%). Avec l'amélioration du placement des données, les défauts générés par les caches étudiés deviennent proches du taux de défauts de capacité obtenu avec le placement original des données (ce taux représente le taux minimum d'échecs qu'il est possible d'avoir avec des optimisations purement matérielles du cache).

4.5.6 Trace de gzip

Une trace de 2 millions de références a été obtenue pour gzip pendant qu'il compressait une autre trace. Les deux premiers millions de références ont été supprimés au préalable pour éliminer l'effet de démarrage du programme. La figure 4.3(f,g) montre les résultats expérimentaux obtenus.

Les défauts sont équitablement répartis entre défauts de capacité et défauts de conflits. L'heuristique élimine 78% des défauts de distribution, soit au total 72% des défauts de capacité. Elle élimine aussi 43% des conflits pour le cache à correspondance directe (de 11,54% à 6,63%).

Au total la réduction du taux d'échecs est de 56% pour le cache à correspondance directe (de 21,21% à 9,34%) et de 64% pour le cache associatif par ensemble de 4 (de 17,94% à 6,49%). Avec la trace optimisée les taux d'échecs des caches sont inférieurs au taux de défauts de capacité de la trace originale (9,67%). Il n'est donc pas possible avec des solutions purement matérielles d'obtenir de telles performances. De plus, le taux d'échecs est encore réduit avec l'utilisation de caches associatifs. Il est donc possible de combiner efficacement les optimisations logicielles et matérielles.

4.5.7 Trace de go

Pour finir, le programme go, un benchmark de la suite Spec'95, a été tracé de bout en bout. La trace contient 47 millions de références mémoires. Le programme a été compilé avec gcc -O sous SunOS4, et exécuté avec la commande “go 10 10” avec le paramètre FixPlayLevel=20.

Pour le cache à correspondance directe, le taux de défauts de conflit est élevé, mais il décroît sensiblement avec l'associativité. Les optimisations matérielles sont donc très utiles.

Notre heuristique réduit les défauts de distribution de 69%, soit une réduction de 46% des défauts de capacité (de 1.18% à 0,37%). Les défauts de conflit sont réduits de 79% avec le cache à correspondance directe (de 6,47% à 1,39%), de 66% pour le cache associatif par ensembles de deux et de 52% pour le cache associatif par ensembles de quatre.

L'heuristique apporte une réduction des défauts de cache totale allant de 56% à 64%. Les taux d'échecs de la version optimisée de la trace sont là aussi très proches du taux de défauts de capacité de la trace originale. Ici, l'associativité du cache influe peu sur le taux d'échecs de la trace optimisée, même si les défauts de cache restant sont pour la majorité des défauts de conflit.

4.6 Eliminer les contraintes de placement

Les résultats exposés dans la section précédente présentent l'extraordinaire potentiel de gain associé à une optimisation de la distribution des données en mémoire. Cependant, les résultats ne tiennent pas compte des contraintes imposées par le programmeur et le compilateur, le plus souvent implicitement, sur le placement mémoire des données. Il n'est donc pas possible d'appliquer directement une telle heuristique de placement mémoire sur les données d'un programme.

L'heuristique de placement n'est utilisable sur un groupe de scalaires que s'il est possible d'éliminer les contraintes de placement relatives entre les scalaires de ce groupe.

Dans certains cas, les contraintes de placement mémoire des données sont trop fortes. Elles peuvent venir d'une organisation logique des informations faites par le programmeur qui ne correspond pas à des contraintes de gestion mémoire ou de contraintes de placement trop conservatrices imposées par le compilateur pour gérer la mémoire.

4.6.1 Suppression des contraintes générées par le programmeur

Le programmeur peut regrouper les données en des structures hétérogènes. Cependant, pour des variables statiques ou locales, le nombre d'instances de ces structures généré est connu.

Il est donc souvent possible d'éliminer cette contrainte de placement sans perte de fonctionnalité si elle est superflue, en supprimant la structuration des données (cette structuration peut parfois s'avérer difficile à supprimer, par exemple lorsqu'il y a des assignations entre instances allouées dynamiquement et instances statiques).

4.6.2 Suppression des contraintes générées par le compilateur

La pile est conçue pour gérer le cas de routines appelées récursivement. Lorsque les procédures ne sont pas récursives, l'allocation des variables locales dans la pile ne s'impose pas.

Dans ce cas, elles peuvent être allouées comme variables statiques sans modifier le comportement du programme. Les contraintes de placement des variables locales des procédures non récursives peuvent donc être supprimées en les déclarant comme variables statiques.

Pour faire cette modification, il faut faire lors de la compilation une analyse interprocédurale du graphe d'appel des fonctions pour repérer les routines non récursives.

4.6.3 Réduction de l'occupation mémoire

Un ensemble de scalaires alloués statiquement peut être généré sans modifier la validité du programme, en cassant les structures hétérogènes qui ne sont pas allouées dynamiquement, et en déclarant comme variables statiques les variables locales de routines non récursives.

Il est ainsi envisageable d'obtenir un ensemble de scalaires indépendants pouvant être redistribués relativement les uns par rapport aux autres en mémoire pour améliorer la localité spatiale.

Cependant, une variable locale n'est utilisée que lorsque sa routine est exécutée. Elle peut donc être longtemps inactive, ce qui réduit fortement la localité temporelle et qui gaspille de la mémoire, car son espace aurait été réutilisé automatiquement si elle avait été allouée dans la pile.

Avant de redistribuer les scalaires en mémoire, il est possible de faire d'abord une optimisation équivalente à l'allocation de registres, mais en mémoire:

- Les scalaires sont démultipliés en fonction de leur vivacité, de sorte à ce que les nouveaux scalaires ne soient vivants que sur un seul segment de code.
- Les scalaires vivants qui ne se chevauchent pas dans le temps sont regroupés pour ne former qu'un scalaire, de façon à créer un plus petit nombre de scalaires, vivant et très référencés sur la plus grande période de temps possible, de façon à maximiser la localité temporelle.

– Ces scalaires sont ensuite redistribués en mémoire avec l'heuristique pour maximiser la localité spatiale. Le code source est adapté en conséquence.

Il est possible de créer au niveau du compilateur des opportunités pour accroître le nombre de scalaires indépendants qui peuvent être redistribués en mémoire. Une fois qu'une partie des données est convertie en scalaires alloués statiquement, une heuristique de placement de scalaires comme celle qui est proposée peut être appliquée. Le placement peut être basé sur un *profil* du programme ou sur une analyse statique interprocédurale des sources du programme afin d'estimer quelles sont les routines les plus utilisées et intraprocédurale pour déterminer les motifs d'accès aux données.

4.7 Conclusion

Aucune optimisation logicielle de la localité n'a été proposée jusqu'à présent à notre connaissance, autrement que pour des matrices, car il est admis que les motifs d'accès aux données autres que les matrices sont trop aléatoires pour être analysés.

Il a cependant été montré une heuristique capable de placer automatiquement des scalaires indépendants en mémoire, de façon à réduire les défauts de capacité et de conflit des caches.

Cette heuristique utilise des informations de *profil* qui sont fournies par une trace du programme. L'heuristique trouve en une passe une distribution améliorée des données. Elle a été testée sur des traces de programmes. Ces résultats montrent qu'il existe des distributions qui améliorent suffisamment la localité du programme, pour que parfois un cache à correspondance directe puisse avoir un taux d'échec plus faible que le taux de défauts de capacité du programme original. Il est donc possible d'améliorer la localité des programmes de façon significative, si les contraintes de placement en mémoire ne vont pas à l'encontre de la localité spatiale.

L'heuristique développée ne peut être implantée directement dans un compilateur, car les données manipulées par le programme sont rarement des scalaires, et les problèmes de gestion mémoire apportent beaucoup de contraintes de placement.

Pour pouvoir redistribuer les données en mémoire, il faut néanmoins trouver des solutions de distribution adaptées à chaque type de données. Des solutions pour générer des scalaires indépendants sont proposées dans ce chapitre. Le *padding*, présenté dans l'annexe A a aussi déjà été étudié pour les matrices, mais offre des possibilités limitées. Seules, ces techniques seront cependant insuffisantes pour améliorer la performance des applications généralistes. C'est pourquoi, dans le chapitre suivant des nouvelles techniques de placement sont proposées pour les structures de données hétérogènes allouées dynamiquement, très fréquemment utilisées par les programmes généralistes.

Chapitre 5

Allocation dynamique de structures de données hétérogènes

5.1 Introduction

Les langages de haut niveau permettent de déclarer des structures de données hétérogènes. Celles-ci autorisent les programmeurs à définir des types de données spécifiques à leur application. Elles se présentent soit sous la forme de structures en C, soit sous la forme de classes dans les langages objet comme le C++.

Les structures de données hétérogènes sont utilisées en général pour allouer dynamiquement un nombre variable de données. Une instance d'une structure de donnée sert souvent à mémoriser toutes les propriétés relatives à un même élément d'information. Pour gérer un nombre variable de tels éléments, on crée pendant l'exécution du programme autant d'instances de la structure de donnée qu'il est nécessaire. Cette opération se fait à l'aide d'une bibliothèque d'allocation dynamique.

Les accès à ces instances peuvent pour certains programmes représenter la majorité des accès mémoires. Il est donc important de chercher à optimiser la localité des accès aux instances de structures de données hétérogènes allouées dynamiquement. Bien que des travaux étudient et cherchent à optimiser la localité à l'intérieur des bibliothèques d'allocation dynamique [GZH93, GZ93, DTM94, WW88], à notre connaissance aucun ne s'est penché spécifiquement sur la localité spatiale des références faites par le programme sur des données allouées dynamiquement. Notre étude tente donc d'améliorer la localité des références faites par le programme aux données allouées dynamiquement. Pour cela, deux techniques sont proposées pour modifier le placement mémoire des instances de structures allouées dynamiquement et ainsi améliorer la localité de leur références. Des améliorations de la bibliothèque d'allocation sont aussi étudiées, afin que la gestion dynamique de la mémoire reste efficace sans pour autant entraver la localité spatiale des références du programme aux données allouées.

5.1.1 Contributions

Pour améliorer la localité des données allouées dynamiquement, une technique de placement interne à chaque instance est d'abord proposée, puis ensuite une technique permettant de redistribuer les champs entre plusieurs instances [TBS98]. Pour faciliter cette seconde optimisation, une bibliothèque d'allocation dynamique supportant cette optimisation a été développée. Afin de rendre cette bibliothèque performante, l'influence sur la performance du système et sur la localité des données des techniques de gestion mémoire de la bibliothèque a été aussi étudiée.

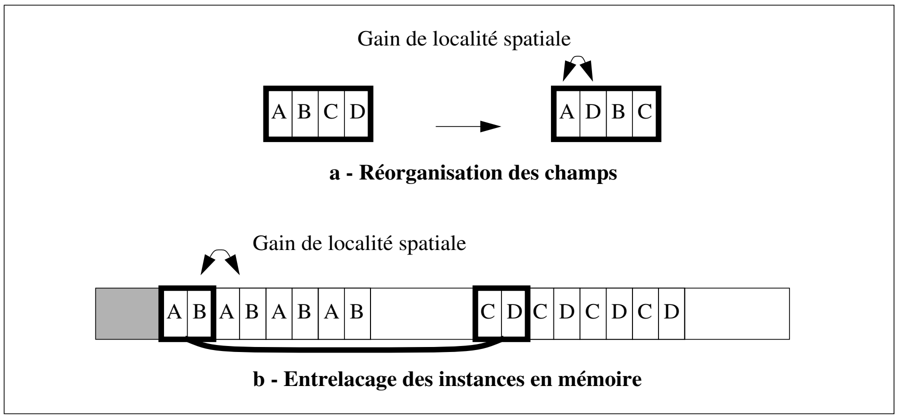


FIG. 5.1 – *La redistribution et l'entrelacement des champs augmentent la localité spatiale.*

Dans une première étape, il est montré que redistribuer les champs des structures de données entre eux dans la déclaration de la structure peut modifier la localité des données. La redistribution des champs consiste à placer côte à côte les champs les plus corrélés pour améliorer la localité spatiale des instances de la structure. Cela correspond à une approche équivalente à celle utilisée pour les scalaires au chapitre 4. Les scalaires les plus corrélés avaient alors été redistribués dans les mêmes blocs mémoires. Pour les structures de données cela se fait à une échelle plus réduite : ne sont redistribués entre eux que les champs d'une même structure. Toutes les instances suivent cette distribution. L'avantage d'une telle méthode est que sa mise en oeuvre est immédiate.

La figure 5.1(a) illustre cette optimisation en présentant l'exemple d'une instance d'une structure composée de quatre champs. Si pour la majorité des instances de la structure, les champs A et D sont très corrélés, ils sont regroupés dans la déclaration de la structure. Les champs corrélés de l'instance deviennent donc contigus en mémoire.

Il n'est possible de redistribuer les champs qu'à l'intérieur d'une même structure de donnée. Cette solution bien qu'efficace dans certains cas, reste donc insuffisante. Souvent la corrélation est plus grande entre les champs de différentes instances qu'entre les champs d'une même instance. De plus les structures ont rarement suffisamment de champs pour occuper plusieurs lignes de cache. La redistribution dans ce cas ne permet pas de séparer les champs les moins corrélés dans des lignes différentes de ceux souvent référencés.

Il est donc proposé dans une seconde étape l'entrelacement des instances, une solution pour accroître plus spécifiquement la localité spatiale entre plusieurs instances d'une même structure de donnée. Certains champs dans une structure sont utilisés plus fréquemment que d'autres. La méthode consiste donc à regrouper dans le même bloc mémoire les champs les plus corréles de plusieurs instances. Les champs faiblement corrélés sont isolés dans d'autres blocs mémoires. Ils ne sont donc pas chargés inutilement dans le cache dès que l'instance est référencée, et ce quelle que soit la taille de la structure.

La figure 5.1(b) montre un exemple d'entrelacement pour quatre instances. Dans cet exemple, l'entrelacement a pour but d'accroître la localité spatiale entre les champs *(A,B)* des quatre instances. Ce type d'optimisation s'avère efficace, par exemple dans le cas d'un parcours de liste.

Pour mettre en oeuvre l'entrelacement dans des programmes écrits en langage C, *ialloc*, une bibliothèque d'allocation prenant en charge la gestion mémoire nécessaire pour allouer des instances entrelacées a été mise au point. *Ialloc* est un *front-end allocator*, c'est-à-dire qu'elle s'installe au-dessus de la bibliothèque d'allocation standard *malloc* pour la compléter.

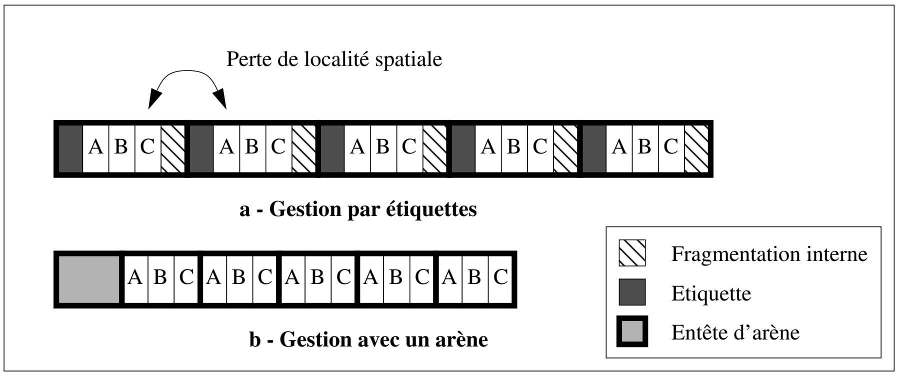


FIG. 5.2 – *Les arènes améliorent la localité spatiale.*

En développant cette bibliothèque d'allocation, il a fallu d'abord prendre en compte les résultats des travaux de Grunwald et al. [GZH93], qui remarquent que lorsqu'elles sont mal conçues, les bibliothèques d'allocation accèdent à la mémoire avec de mauvaises propriétés de localité. De plus, il est remarqué ici qu'une conception standard de la bibliothèque d'allocation engendrerait aussi des contre-performances notables en vitesse et en gaspillage mémoire, et réduit surtout la localité spatiale entre les instances contiguës en mémoire.

Dans une troisième étape, une bibliothèque performante prenant en compte tous ces problèmes est développée, afin de ne pas perdre les bénéfices des optimisations de placement. Une section est donc consacrée à l'étude de la bibliothèque d'allocation optimisée pour l'allocation de structures de données hétérogènes.

Une bibliothèque standard accole une étiquette à chaque instance allouée, afin de connaître la taille et l'état du bloc alloué. Comme cette étiquette n'est utilisée que par la bibliothèque d'allocation, elle n'est pas corrélée aux champs de l'instance, et le programme perd de la localité spatiale. Pour éviter cela, la bibliothèque *ialloc* alloue des arènes dans le tas. Une arène sert à allouer plusieurs instances d'une structure. La gestion de l'utilisation de l'arène ne se fait qu'avec une petite entête. Il n'est plus nécessaire d'accoler des étiquettes aux blocs mémoire alloués, ce qui améliore la localité spatiale, comme le schématise la figure 5.2.

En combinant l'utilisation d'une bibliothèque optimisée qui n'entrave pas la localité spatiale des données, et les techniques de réorganisation et d'entrelacement des structures de données, il a été possible d'améliorer le temps d'exécution de la plupart des programmes étudiés.

5.2 Les bibliothèques d'allocation standard

5.2.1 Problématique des bibliothèques d'allocation standard

Les bibliothèques d'allocation dynamique ont été mises au point pour permettre aux programmeurs d'allouer dynamiquement des données en fonction des besoins du programme. Une zone mémoire, appelée *tas* est réservé pour l'allocation dynamique. Sa taille peut évoluer en fonction des besoins du programme, en faisant des appels système. La bibliothèque d'allocation a pour rôle de gérer le tas. Lorsque le programme demande un bloc mémoire, la libraire doit lui fournir un pointeur sur un bloc de mémoire libre suffisamment large. Lorsque le programme se débarrasse d'un bloc mémoire, la bibliothèque doit s'occuper de recycler ce bloc pour ne pas le gaspiller.

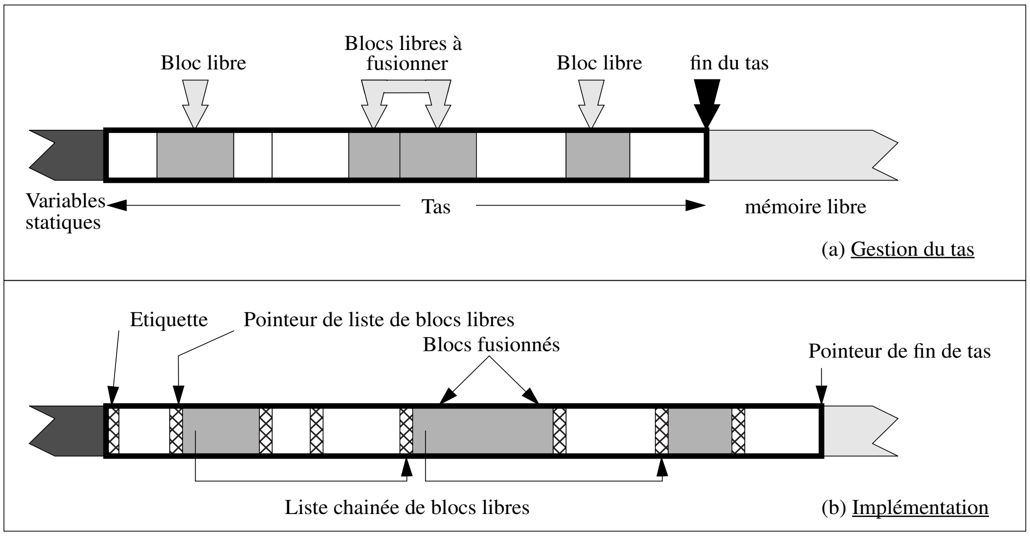


FIG. 5.3 – *La fragmentation du tas.*

Lorsque le programme alloue et libère des blocs de mémoire, le tas se fragmente : il devient composé de blocs occupés et de blocs libres non contigus, figure 5.3(a). Il faut donc que la bibliothèque d'allocation gère les blocs mémoire du tas qui sont libérés par le programme afin de les réutiliser.

La bibliothèque d'allocation effectue principalement trois tâches représentées figure 5.3(b) :

– **Reconnaître la taille des blocs** libérés : pour cela, la bibliothèque ajoute le plus souvent une **étiquette** aux blocs alloués indiquant la taille et l'état libre ou occupé du bloc.
– **Réallouer les blocs libres** : il faut garder la liste des blocs libérés. La technique *first-fit* [Knu73, Tan95] consiste à organiser les blocs libres en une liste chaînée circulaire. Lorsque le programme demande un bloc, on parcoure cette liste pour trouver le premier bloc libre suffisamment large pour satisfaire la requête. Le bloc est coupé à la bonne taille et le résidu est remis dans la liste chaînée. Si aucun bloc ne convient, on demande au système d'agrandir le tas.
– **Eviter la fragmentation externe** : Lorsque le programme libère un bloc et que celui-ci est contigu à un autre bloc libre, on **fusionne**[1] ces blocs de façon à obtenir un plus gros bloc. Lorsque le programme demande un gros bloc mémoire, il y a ainsi plus de chance de recycler un des blocs libres. Il n'y a pas à redemander de la mémoire au système.

La gestion mémoire peut être très coûteuse en temps et en mémoire. De nombreuses optimisations sont possibles.

Pour accélérer la fusion, on tire parti des informations contenues dans les étiquettes des blocs. Lorsqu'un bloc est libéré, la bibliothèque d'allocation n'a accès qu'à l'étiquette. Celle-ci doit donc contenir la taille du bloc. De plus, il n'est pas envisageable de faire le tour de la liste de blocs libres lorsqu'un bloc est libéré pour savoir si un autre bloc peut être fusionné avec celui-ci. Lorsqu'on libère un bloc, on connaît sa taille. On sait donc où chercher l'entête du bloc suivant. Pour fusionner un bloc avec son successeur, il suffit donc que chaque bloc conserve dans son étiquette son état occupé ou libre. Fusionner deux blocs devient alors simple si la liste est doublement chaînée. Il suffit de changer la taille dans l'entête du bloc libéré, et de mettre à jour les pointeurs dans la liste chaînée. Pour aller plus loin, on peut ajouter une étiquette à la fin de chaque bloc alloué. Cela permet lorsqu'on libère un bloc de le fusionner aussi avec son prédécesseur. D'autres approches, comme le *buddy-system* permettent de calculer rapidement l'adresse de l'entête des blocs contigus, afin de faire rapidement la fusion. Toutes les méthodes échangent du gaspillage mémoire pour une économie de temps, alors que le but même de la fusion est d'éviter le gaspillage mémoire qu'engendre la fragmentation du tas.

La fusion coûte cher. Pour l'éviter, d'autres algorithmes de parcours de liste ont été envisagés afin de rechercher des blocs libres de taille aussi proche que possible de celle requise par le programme. Ces algorithmes (*worst-fit, best-fit...*) n'apportent aucun gain décisif par rapport au *first-fit* [Knu73, Tan95]. Pour réduire le temps de parcours, la bibliothèque d'allocation de type *better-fit* de Sun-OS V4.1.1 gère un arbre binaire trié par adresses [GZ93, Ste83], au lieu d'une liste.

1. En anglais, le terme *coalescing* est utilisé.

Afin d'éviter de fusionner les blocs libres et de réduire les recherches dans la liste de blocs libres, de nombreuses solutions ont été mises au point, souvent au détriment de l'économie de la mémoire. Il a été remarqué que la majorité des programmes ont tendance à allouer principalement quelques tailles de bloc. Ces blocs sont de plus souvent de petite taille. On tire parti de ces propriétés avec le *segregated storage*. Le *segregated storage* consiste à gérer plusieurs listes triées par tailles de blocs. C'est ce que fait la bibliothèque d'allocation du *Gnu C++* [GZH93]. Pour réallouer ces blocs, il n'est pas besoin de les fusionner, car le programme a une forte probabilité de redemander cette taille fréquemment. Pour éliminer complètement les parcours de listes, il suffit que ces listes contiennent des blocs toujours de même taille. Pour cela, la taille des blocs alloués est arrondie [GZ93, OA85, Boz84, LH82] à des multiples, voir à des puissances de 2 comme dans la bibliothèque d'allocation du noyaux d'Unix *BSD*[2] [MBKQ96]. Arrondir la taille des blocs alloués génère un type de gaspillage mémoire appelé **fragmentation interne** [Tan95]. Bien que le but d'une bibliothèque d'allocation soit de gérer le tas au mieux, le gaspillage mémoire n'est en fait pas en général le critère le plus important retenu pour évaluer les bibliothèques d'allocation. Un gaspillage de 50% est souvent jugé satisfaisant [KV85].

A l'origine, le but de la bibliothèque d'allocation dynamique était de gérer le tas de façon à limiter le gaspillage mémoire. A mesure que les machines disposaient de plus de mémoire, les bibliothèques conçues se sont mises à gaspiller de plus en plus de mémoire, privilégiant la rapidité de gestion sur l'économie mémoire.

5.2.2 La localité des bibliothèques d'allocation

Afin d'améliorer la performance des bibliothèques d'allocation, les études se sont penchées sur la localité des références mémoires faites par les bibliothèques d'allocation lorsqu'elles gèrent le tas.

Les méthodes classiques pour améliorer la gestion mémoire sont souvent contre performantes [GZ93, GZH93, ZG92a]. Il faut par exemple éviter de parcourir des listes chaînées de blocs libres ou de fusionner les blocs. Pour eux, il n'est pas non plus nécessaire de générer une fragmentation interne excessive pour obtenir une gestion du tas plus simple et plus rapide, comme cela est fait par exemple dans la bibliothèque BSD [MBKQ96, MK88].

Parfois conscients de l'inadéquation des bibliothèques d'allocation aux propriétés de leurs programmes, les programmeurs recherchant la performance gèrent eux même la mémoire qu'ils allouent dynamiquement [Zor93]. Ils savent que les bibliothèques d'allocations, trop généralistes, ne sont pas suffisamment performantes pour leurs besoins. Malheureusement, ces optimisations logicielles s'avèrent en réalité le plus souvent relativement inefficaces, car la gestion ne prend pas toujours en compte la localité des accès mémoires. Une bibliothèque d'allocation optimisée peut en général remplacer avantageusement ces optimisations manuelles [Zor93].

2. Si une page de mémoire virtuelle fait 4KO sur la machine, toutes les requêtes de moins de 8KO sont gérées de cette manière.

Les bibliothèques à base de *segregated storage* offrent de bonnes propriétés de localité [GZ93, GZH93, ZG92a], surtout si les listes contiennent qu'une seule taille de bloc : lorsqu'il faut réallouer un petit bloc, il n'y a pas besoin de parcourir la liste chaînée, il suffit de prendre le premier élément de la liste. Par exemple, la bibliothèque *QuickFit* [WW88] utilise pour les petits blocs un tableau de listes de blocs libres, indexé par tailles (les tailles sont arrondies au mot près). Chaque liste contient une seule taille de bloc. Aucune fusion n'est tentée pour les petits blocs dans ces listes. Cette bibliothèque a une bonne localité pour les tailles de blocs gérés, mais comme les blocs de tailles différentes sont séparés et jamais fusionnés, elle peut gaspiller de la mémoire pour les applications qui allouent certaines tailles de façon cyclique.

Il n'est pas nécessaire d'utiliser des listes de blocs libres pour gérer le tas. La bibliothèque d'allocation fournie avec le Gnu C, développée par Mike Haertel [GZH93] gère les gros blocs de plus de 4KO avec un algorithme *FirstFit* [Tan95]. Les petits blocs sont arrondis à une puissance de 2 et alloués dans des **arènes**. Une arène est une zone du tas gérée de manière autonome. Elle a une entête qui contient les informations nécessaires pour allouer les blocs de sa zone. L'utilisation d'une entête unique pour gérer l'allocation améliore la localité de la routine d'allocation, qui n'a plus à parcourir la mémoire pour accéder à une liste chaînée de blocs libres. Une arène n'est libérée que quand tous ses blocs sont libérés.

Afin d'adapter la gestion du tas aux motifs d'allocation/libération des programmes, Grunwald et Zorn proposent *CustoMalloc* [GZ93], une bibliothèque paramétrée avec un *profil*. Ainsi les tailles de blocs gérées par les listes correspondent précisément aux tailles utilisées par le programme. Les tailles de blocs alloués le plus fréquemment sont gérées très rapidement par des listes de blocs libres. Certaines listes peuvent être fusionnées si leur *profil* indique qu'elles sont utilisées de façon cyclique, afin d'économiser la mémoire et améliorer la localité temporelle. Ils notent que pour la majorité des programmes, il suffit de pouvoir gérer 4 tailles de blocs distincts pour satisfaire 95% des allocations. Leur bibliothèque est 2 à 10 fois plus rapide que les bibliothèques standard, et ne gaspille pas plus de 25% de l'espace alloué en fragmentation interne par rapport à *FirstFit*, l'algorithme le plus économique.

Grunwald et Zorn concluent que le but premier des bibliothèques d'allocations d'économiser la mémoire n'est plus d'actualité depuis que les machines utilisent la mémoire virtuelle. Il devient plus important de garantir de bonnes propriétés de localité au programme que d'améliorer la réutilisation des blocs mémoire libres. En effet, une page mémoire allouée mais inutilisée est *swappée* sur le disque dur par le système d'exploitation et influe peu sur la quantité de mémoire dynamique disponible, alors que la mauvaise localité des références pénalise les caches, voir le système de *swap* des pages mémoires sur le dur (si les éléments référencés sont répartis dans un trop grand nombre de pages, celles-ci ne tiennent plus dans la mémoire disponible). Par exemple, améliorer la réutilisation des blocs libérés peut réduire la performance du programme si cela distribue les nouveaux blocs alloués dans toutes les pages mémoires (perte de localité spatiale pour le système de *swap*).

La bibliothèque d'allocation proposée ici est conçue autour de l'allocation d'arènes pour avoir une bonne localité dans la phase d'allocation. Cependant, le but est non seulement d'améliorer la localité de la bibliothèque, mais aussi d'améliorer la localité de toutes les références du programme aux données allouées dynamiquement.

5.3 Améliorer la localité spatiale des données allouées dynamiquement

Deux techniques sont proposées ici pour améliorer la localité spatiale de structures allouées dynamiquement, la réorganisation des champs d'une structure, et l'entrelacement des champs de plusieurs instances d'une même structure.

La réorganisation des champs permet d'améliorer la localité spatiale entre les champs d'une même instance alors que l'entrelacement des instances permet d'exploiter la localité spatiale qu'il peut y avoir entre plusieurs instances d'une structure.

Ces deux techniques sont d'abord décrites.

Il est ensuite expliqué quand l'utilisation de chacune de ces approches est appropriée. Une méthodologie d'optimisation à base de *profiling* est esquissée avant de conclure. Les techniques exposées sont appliquées à des programmes en C, bien qu'elles soient utilisables avec tous les langages, du moment qu'ils définissent un type de donnée équivalent aux structures de données hétérogènes.

5.3.1 Réorganisation des champs d'une structure

5.3.1.1 Méthode

Une structure de donnée est composée d'un ensemble de champs qui peuvent être redistribués arbitrairement les uns par rapport aux autres.

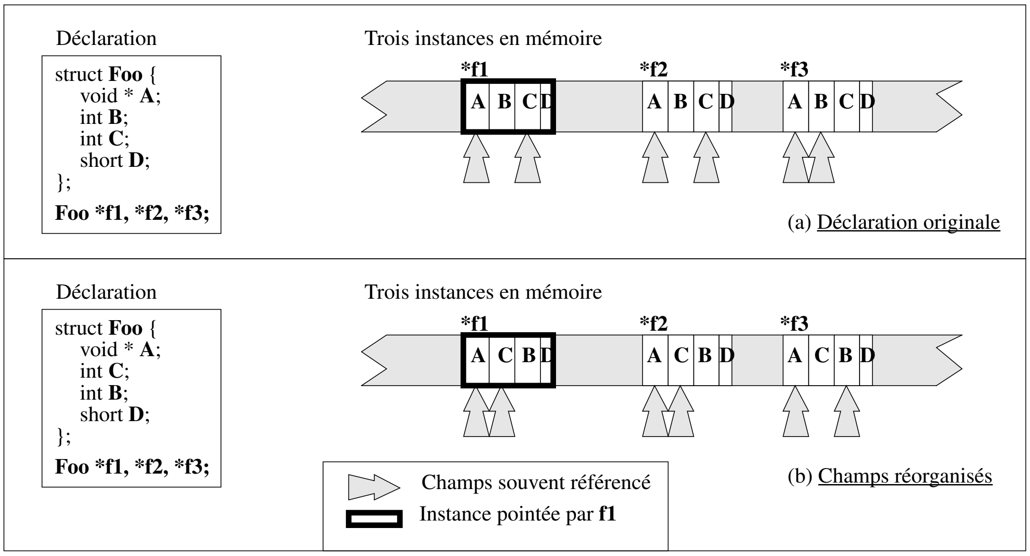


FIG. 5.4 – *Réorganisation des champs d'une structure.*

Contrairement aux cellules d'une matrice, les champs d'une structure de donnée n'ont pas de contraintes de placement[3], car chaque champ est nommé, et l'accès au champ ne se fait qu'avec ce nom. Pour l'accès à un champ, le compilateur calcule statiquement sa position relative par rapport à l'adresse de base de l'instance en s'aidant de la déclaration de la structure de donnée.

Cela permet entre autres de composer la structure avec des champs de n'importe quel type et n'importe quelle taille. Par exemple, la structure figure 5.4(a) contient à la fois des champs de type *short*, *int* et *void **.

Cette propriété permet en plus de redistribuer les champs sans contrainte les uns avec les autres à l'intérieur de la structure. Si par exemple les champs A et C de la figure 5.4(a) sont plus fréquemment référencés ensemble[4] que les champs A et B, la structure *Foo* est réorganisée de telle sorte que A et C soient contigus en mémoire, comme dans la figure 5.4(b).

En fait, comme les compilateurs C n'ont pas de contraintes de placement pour les champs des structures, ceux-ci sont placés en mémoire dans l'ordre de déclaration fait par l'utilisateur dans le code source.

Ce placement n'est pas toujours optimal. Le programmeur n'est en général pas conscient de l'impact du placement des champs dans la structure sur la performance du programme. De plus, il ne lui est fourni aucun outil pour décider de l'ordre de déclaration des champs.

La réorganisation des champs proposée peut améliorer la localité des programmes. Il a été vu au chapitre 4 qu'il existe des placements mémoires meilleurs que d'autres pour des scalaires, et qu'il est possible de trouver des placements meilleurs que ceux générés par le compilateur. La même approche peut être utilisée pour améliorer automatiquement le placement des champs des structures de données, car ceux-ci n'ont pas de contraintes de placement entre eux.

Bien qu'une structure donnée génère beaucoup d'instances, il ne peut y avoir qu'une seule déclaration pour la structure, et donc une seule façon d'organiser les champs entre eux. La distribution des champs doit donc être choisie de façon à améliorer la localité spatiale de la majorité des instances allouées. Heureusement, la majorité des accès aux instances se font par une petite partie du code, car souvent à peine 10% du code est responsable de 90% des accès mémoire [PH96]. Pour cette raison, la même distribution sera probablement efficace pour la majorité des instances de la structure.

La réorganisation des champs des structures permet d'améliorer la localité spatiale lorsque le programme ne référence le plus souvent que quelques champs dans les instances d'une même structure de donnée. La distribution doit être faite de manière à améliorer la localité des références des parties les plus actives du programme.

TAB. 5.1 – *Impact de la réorganisation des champs sur le temps d'exécution de BigStruct.*

(secondes)	Original	Réorganisée
	Malloc	
initialisation	50	31
recherche 1	87	68
recherche 2	91	71
total	**229,89**	**172,25**
utilisateur	181,36	141,90
système	46,77	29,16
	Ialloc	
initialisation	40	39
recherche 1	84	71
recherche 2	85	71
total	**210,59**	**182,54**
utilisateur	184,15	156,38
système	25,02	24,88

5.3.1.2 Influence sur le temps d'exécution

La réorganisation des champs d'une structure permet de réduire l'empreinte utile d'une instance d'une structure dans le cache. Pour les grosses structures de données, qui s'étendent sur plusieurs lignes de cache, réorganiser les champs permet de réduire le nombre de lignes à charger dans le cache lorsqu'une instance est utilisée.

Pour mettre en évidence l'impact que peut avoir la réorganisation des champs, *BigStruct* a été développé. Ce petit programme alloue un grand nombre d'instances d'une grosse structure récursive organisées en une liste chaînée. Le corps du programme effectue 2 séries de recherches utilisant des champs différents. Pour chaque instance, peu de champs sont référencés à chaque recherche. Les champs sont donc réorganisés pour que ceux utilisés lors de la première recherche soient regroupés ensemble, de même pour les champs utilisés dans la seconde recherche. Ainsi, lors de la recherche, dès qu'un champ d'une instance est référencé, les autres champs utiles sont préchargés dans la même ligne de cache. Les autres champs de l'instance ne sont pas chargés dans le cache, ainsi ils ne le polluent pas.

Pour une petite structure par contre, la réorganisation des champs n'apporte rien, car les champs inutilisés seront quand même préchargés automatiquement par la ligne de cache.

La table 5.1 montre les gains obtenus en réorganisant les champs avec la bibliothèque *malloc* standard et la bibliothèque *ialloc*. La bibliothèque *ialloc* sera décrite au chapitre 5.4. Elle est juste utilisée en remplacement de malloc afin montrer l'influence de la

3. Il arrive cependant parfois qu'en C certains programmeurs utilisent des calculs de pointeurs pour référencer les champs, même si cela n'est pas conseillé dans le standard.

4. Symbolisés par des flèches grises dans la figure.

réorganisation sur les deux bibliothèques. Avec *malloc* la réorganisation des champs apporte un speedup de 1,33 et avec *ialloc* un speedup de 1,15.

5.3.2 Entrelacement des instances d'une structure

La réorganisation des champs permet de regrouper ensemble les champs souvent référencés ensemble. Cette optimisation n'est utile que si cela réduit le nombre de blocs chargés dans le cache. Lorsque la structure de donnée est de taille inférieure ou égale à la taille des lignes du cache, il faut entrelacer les instances pour séparer les champs rarement utilisés des champs fréquemment utilisés. Dans un bloc mémoire, les champs rarement utilisés sont ainsi remplacés par les champs d'autres instances ayant une plus grande probabilité d'être référencés.

5.3.2.1 Méthode

Les programmes allouant un grand nombre d'instances d'une structure ont besoin de manipuler ces instances à l'aide d'un petit nombre de pointeurs. Il faut donc organiser l'accès à ces instances en mémoire. Les structures allouées dynamiquement sont donc le plus souvent soit organisées en tableaux soit en structures récursives, comme des listes chaînées ou des arbres binaires.

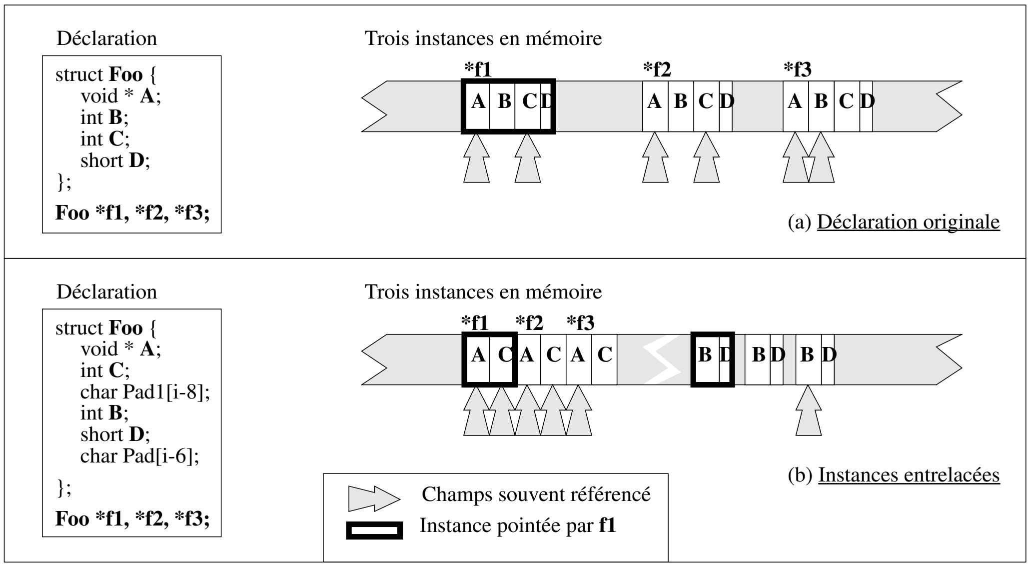


FIG. 5.5 – *Entrelacement des champs de plusieurs instances d'une structure.*

Lorsque le programme parcourt les instances d'une structure récursive, l'organisation en structure génère rarement une bonne localité spatiale, surtout si peu de champs sont accédés à chaque parcours. L'entrelacement des champs a pour but de regrouper en mémoire les champs fréquemment référencés de plusieurs instances, afin d'améliorer la localité spatiale lorsque le programme effectue ce type de parcours.

Pour cela, en fonction de leur utilisation, on regroupe les champs d'une structure en groupes de champs que l'on entrelace. Supposons par exemple que la majorité des références soient faites par la recherche d'une instance parmi toutes les instances allouées. Les champs référencés le plus fréquemment lors de la recherche sont marqués d'une flèche dans la figure 5.5(a). Comme les champs A et C sont les plus référencés, ils sont rassemblés pour former un premier groupe, les champs B et D restant formant le dernier groupe. Figure 5.5(b), on entrelace ensuite les groupes ainsi formés de façon à ce que les groupes de champs (A,C) de chaque instance, fréquemment utilisés, soient plus proche possible en mémoire. Cette solution est plus efficace qu'une simple réorganisation des champs, comme en figure 5.4(b), car les champs utilisés occupent une zone mémoire encore plus restreinte, les champs inutilisés ayant étés écartés dans une autre zone mémoire.

L'entrelacement des instances pour une structure récursive s'apparente à la transposition ou à l'échange de boucle (*loop interchange*) pour les matrices [BGS94]. Le but est de générer des accès mémoires de pas unitaire pour augmenter la localité spatiale.

En se référant à la règle du 90%/10%, l'entrelacement des champs peut s'avérer très efficace. Cela se vérifie par exemple quand la petite portion du programme responsable de la majorité des accès aux instances de la structure correspond à la partie du programme qui effectue la recherche d'une instance. Il faut donc rassembler dans un même groupe tous les champs de la structure touchés par cette portion de code, puis entrelacer ce groupe de champs pour bénéficier au maximum de l'effet de préchargement des lignes de cache.

L'entrelacement des champs des instances d'une structure permet d'augmenter la localité spatiale du programme lorsqu'il parcourt un grand nombre d'instances allouées en utilisant les même champs à chaque fois.

5.3.2.2 Influence sur le temps d'exécution

L'entrelacement des champs de plusieurs instances d'une structure permet de réduire l'empreinte globale d'un groupe d'instances d'une même structure dans le cache. Cela est possible lorsque le programme accède à un grand nombre d'instances d'une même structure, si pour chaque instance utilisée seuls quelques champs sont référencés.

Dans le cas de listes chaînées, par exemple, les éléments sont souvent alloués dans l'ordre dans lequel ils sont référencés. Les éléments contigus étant référencés dans l'ordre, l'entrelacement des champs apporte un gain immédiat. Les champs utilisés sont regroupés en mémoire, et ce regroupement écarte les champs non référencés, et favorise la localité spatiale entre instances entrelacées consécutivement. Dans le cas d'organisations en arbres, le gain de localité est plus faible, mais la diminution de l'empreinte des instances dans le cache augmente toujours le nombre d'instances chargées simultanément dans le cache et réduit les risques d'interférences.

Pour montrer l'intérêt de l'entrelacement, *MedStruct* a été développé. Ce petit programme effectue un million de recherches dans une liste chaînée. La liste chaînée est composée de quatre millions d'instances d'une structure de donnée allouées dynamiquement. La structure de donnée utilisée est aussi large qu'une ligne de cache. Réorganiser les champs de la structure est inutile car pour référencer une instance il faut la charger, et elle occupe une ligne de cache complète indépendamment de l'organisation des champs à l'intérieur de la structure et du nombre de champs réellement référencés. Il ne peut donc pas y avoir d'effet de préchargement entre deux instances car chaque instance occupe un bloc mémoire distinct.

L'entrelacement des champs permet d'augmenter la localité spatiale entre instances contiguës. Au lieu de charger dans la ligne de cache tous les champs de l'instance référencée, on ne charge que les champs utilisés. L'espace restant est rempli avec les champs de plusieurs autres instances, qui ont de fortes chances d'être référencés ultérieurement.

TAB. 5.2 – *Impact de l'entrelacement des champs sur le temps d'exécution de MedStruct.*

(secondes)	Malloc	Ialloc	Ialloc et entrelacement
initialisation	20	25	2
recherche	48	48	22
Temps système	17,03	17,69	0,97
Temps utilisateur	51,98	56,60	23,47
Temps total	**69,52**	74,84	**24,64**

La table 5.2 montre le gain apporté par l'entrelacement des champs. Comme l'entrelacement des instances se fait en utilisant la bibliothèque optimisée *ialloc*, les résultats obtenus sans entrelacement sont montrés à la fois avec la bibliothèque *ialloc* et la bibliothèque *malloc*. La mise en oeuvre de l'entrelacement à l'aide de la bibliothèque *ialloc* sera expliqué au chapitre 5.4.

Sans entrelacement, le programme est plus lent avec la bibliothèque *ialloc* qu'avec la bibliothèque standard *malloc*, parce qu'il apparaît plus de défauts de cache secondaire et de défauts de TLB. En entrelaçant des champs par contre, lors de la recherche on ne charge plus dans le cache les champs inutilisés. On obtient alors un speedup de 2,82 en entrelaçant simplement la structure champ par champ.

D'une manière générale, si on arrive à déterminer qu'une partie des champs d'une structure sont rarement utilisés, on peut définir plusieurs groupes de champs à entrelacer. En combinant entrelacement des instances et réorganisation des champs, on réduit l'empreinte des champs fréquemment utilisés dans le cache, et on augmente automatiquement la localité spatiale du programme.

5.3.3 Optimisations par profiling

Pour entrelacer ou réorganiser les champs d'une structure, il faut connaître dès la compilation comment le programme utilise les champs des instances de structures allouées. Pour cela, il faut analyser le flot d'instructions.

Afin de déterminer le flot dynamique d'instructions, il est possible de faire une analyse statique du programme et d'inférer à partir de règles d'exécution quelles sont les parties du programme qui seront le plus exécutées. Cela est fait par les compilateurs tels que le C Mipspro Compiler de Silicon Graphics sous Irix 6.4. Cependant cette analyse est aléatoire : elle se base sur des pondérations arbitraires appliquées aux branchements de programme. On peut appliquer par exemple une pondération 100 au branchement d'une boucle, et 1 au branchement sortant de la boucle.

Une autre approche moins aléatoire est d'utiliser un *profil* d'exécution du programme. Le programme est compilé et instrumenté. En l'exécutant, on obtient des informations précises sur son comportement dynamique. En prenant en compte les informations amassées, on peut adapter le placement des données en mémoire afin qu'il s'adapte le mieux à la partie du programme s'exécutant le plus souvent. Le problème d'une telle approche est qu'il faut garantir que l'exécution utilisée pour optimiser le programme est typique de la majorité les exécutions futures.

Les outils de *profiling* ont évolué. Sur la plupart des machines Unix, avec des outils comme prof ou gprof, il est possible d'obtenir la liste des routines représentant la majorité du temps d'exécution d'un programme. Sur les machines modernes, des compteurs d'événements matériels sont implantés dans le processeur. Il est donc maintenant possible avec des outils logiciels tirant partie de ces compteurs, comme perfex et ssrun sous Irix 6.4, d'obtenir un décompte précis du nombre de défauts de cache lors de l'exécution du programme. On peut donc obtenir ligne par ligne le nombre de défauts de cache générés par les références mémoires de la ligne de code source. Des informations aussi précises facilitent la mise au point des programmes.

Nous ne disposons pas de moyens permettant d'implanter automatiquement ces optimisations dans un compilateur. Il n'est donc pas possible de tirer parti de l'analyse de code statique faite par le compilateur. L'expérimentation en section 5.5 se base donc sur une optimisation manuelle des programmes faite à partir d'informations obtenues par un *profil*.

5.4 La bibliothèque d'allocation ialloc

5.4.1 Introduction

Pour faciliter l'allocation de structures de données entrelacées, une bibliothèque d'allocation adaptée appelée *ialloc*, a été développée. Il est remarqué que les méthodes courantes utilisées pour concevoir les bibliothèques d'allocation ont de nombreux défauts de performance. Il a donc fallu se pencher sur ce problème afin de développer des techniques de gestion mémoire performantes complétant les techniques de placement de données qui ont été détaillées dans le chapitre précédent.

5.4.1.1 Vocabulaire

Pour faciliter la compréhension du chapitre, les termes employés par la suite sont récapitulés ici avec leur définition, afin d'éviter toute confusion.

Dans un langage impératif comme le C, les données manipulées sont toutes typées. Le **type** d'une donnée est une abstraction qui informe le compilateur des propriétés de la données (par exemple sa taille en mémoire) et comment gérer la donnée (par exemple comment s'appliquent les opérateurs arithmétiques). On classe les données en fonction de leur types. En C, les *int*, les *char* et les *float* sont des types standards. Un programme en général n'a besoin que d'un petit nombre de types de données différents.

Une **structure de donnée** est un type de donnée hétérogène, c'est à dire composé d'éléments de types plus simples. Ces éléments sont appelés **champs** de la structure. Par exemple, on peut définir une structure composée de champs *int* et de champs *float*. Dans un programme, plusieurs structures de données peuvent être déclarées. Cela veut dire que le programmeur définit plusieurs types de données correspondant à ses besoins. Dans les langages objets, on ne parle pas de structures de données, mais de classes.

Une fois que l'on a déclaré la structure de donnée, on peut créer des données de ce type pour stocker des informations utiles. Ces données sont appelées **instances** de la structure. Dans un langage objet on parlera d'**objets** de la classe. On retrouve dans chaque instance d'une structure les champs qui ont été déclarés dans la structure (les champs de l'instance sont des instances des champs de la structure).

Les programmes étudiés manipulent surtout des **pointeurs** sur des instances, car l'étude porte sur les structures de données dont les instances sont allouées dynamiquement.

Les optimisations que nous proposons pourront nécessiter plusieurs types de modifications du programme. La plus simple est de changer certains appels à la bibliothèque d'allocation pour utiliser la bibliothèque optimisée. La plus complexe nécessitera de modifier aussi les structures de données, afin de changer le placement relatif des champs des instances en mémoire.

5.4.1.2 Problématique

Le but de la bibliothèque *ialloc* est de fournir des routines d'allocation mémoire performantes, capables de supporter les techniques d'entrelacement des champs d'instances de structures de données.

Il est entendu par bibliothèque performante :

- Une bibliothèque rapide, grâce à des méthodes de gestion mémoire rapides.
  Dans les cas les plus fréquents d'allocation, les opérations doivent être faites avec un minimum de code. Ce code doit être compact afin de ne pas polluer non plus le cache d'instructions.
- Une bibliothèque dont les routines ont de bonnes propriétés de localité.
  La gestion doit être faite en référençant un nombre minimal d'éléments mémoire. Ceux-ci doivent de plus être disposés en mémoire de manière à avoir de bonnes propriétés de localité spatiale.
- Une libraire qui ne réduit pas la localité spatiale entre les blocs de mémoire alloués.
  La gestion mémoire ne doit pas être simplifiée au prix d'une réduction de la localité spatiale de toutes les références mémoires du programme, en accolant par exemple des étiquettes aux blocs alloués, ou encore en générant de la fragmentation interne [GZ93, Tan95, GZH93]. Bien que Grunwald et al. notent que la

perte de localité générée par ces deux techniques n'influe pas de façon significative sur la performance [GZH93], il est montré ici que dans certains cas, la perte de performance induite n'est pas négligeable.

– Une libraire d'allocation qui gaspille le moins de mémoire possible pour gérer les blocs alloués.
  Gaspiller la moitié de la mémoire lorsqu'on gère dynamiquement la mémoire est souvent considéré comme acceptable [MBKQ96]. Cependant, ce gaspillage peut être réduit de façon significative, en évitant justement d'avoir recours aux étiquettes et à la fragmentation interne pour gérer la mémoire. La fragmentation externe peut aussi être maîtrisée. Les bibliothèques standard optimisées par exemple gèrent les petits blocs mémoires libres dans des listes distinctes en fonction de leur tailles pour les réutiliser. Cela est possible car les programmes allouent en majorité de petits blocs de même taille [GZH93, GZ93, ZG92a]. Les blocs plus gros sont fusionnés.

5.4.1.3 Propriété des données allouées et des machines

Afin de créer une bibliothèque performante, il faut tirer partie des propriétés connues des machines et des programmes utilisés.

Les machines actuelles permettent en général d'implanter un système d'exploitation utilisant la mémoire virtuelle. La mémoire virtuelle permet d'allouer des blocs mémoires non contiguës à n'importe quelle adresse virtuelle. Le placement physique dans une zone mémoire physique est géré par le système d'exploitation en fonction des besoins. Une zone inutilisée est déchargée (*swappée*) sur le disque dur pour faire de la place et n'influence pas sur la performance du programme. Les bibliothèques d'allocation tirent parti de la mémoire virtuelle, en ne rendant l'espace qui leur a été attribué qu'en fin de programme. Cela simplifie la gestion mémoire. Les zones libérées inutilisées sont justes *swappées* sur le disque dur de façon transparente par le système d'exploitation pour laisser de la mémoire vive aux autres applications concurrentes.

Les programmes écrits en langage impératif comme le langage C utilisent les structures de données allouées dynamiquement en respectant le plus souvent plusieurs propriétés qu'il est possible d'exploiter. Les données d'un même type ont le plus souvent une durée de vie similaire (temps entre l'allocation et la libération). Un programme manipule le plus souvent un nombre très réduit de types de données distinctes. Les données allouées en même temps ont de fortes chances d'être référencées en même temps. Les bibliothèques d'allocation optimisées exploitent ces propriétés pour gérer plusieurs listes de blocs libres afin d'accélérer la réallocation.

Dans la conception de la bibliothèque, il faut exploiter au mieux ces propriétés pour biaiser les choix de gestion mémoire faits par les routines d'allocation, afin d'avoir une gestion aussi performante que possible à moindre coût.

5.4.1.4 Principe de fonctionnement des arènes

Pour faire une bibliothèque d'allocation performante, la bibliothèque *ialloc* est conçue autour de l'allocation d'arènes.

Pour gérer la mémoire, les bibliothèques d'allocation standard accolent à chaque bloc alloué une étiquette contenant l'état (libre ou alloué) et la taille du bloc. Les blocs de mémoire libérés sont organisés en listes. Les bibliothèques les plus performantes gèrent des listes de blocs libres triées par taille afin de ne pas avoir à perdre de temps à chaque requête du programme à rechercher un bloc de taille correspondante. Une telle gestion nécessite parfois d'arrondir la taille des blocs alloués. Ce gaspillage mémoire est appelé fragmentation interne.

Il a été vu que la majorité des allocations et libérations mémoires concernent un petit nombre de structures de données. Cette propriété des programmes est exploitée pour mettre en place *ialloc*, une bibliothèque d'allocation reposant sur l'allocation d'arènes.

Une arène est un bloc mémoire indivisible, possédant sa gestion propre. Dans une arène, il n'est alloué que des instances d'une seule structure de donnée. Comme toutes les instances ont la même taille, la gestion de l'arène reste très simple. Un bloc de l'arène est soit libre, soit allouée. Il ne faut donc qu'un bit d'information pour connaître l'état d'une zone de l'arène. Ce bit d'information est conservé dans une table dans l'entête de l'arène, et non plus dans une étiquette.

Une arène a une taille fixe, et contient donc un nombre limité d'instances. Il est donc nécessaire de pouvoir gérer plusieurs arènes pour chaque structure de donnée. On fabrique donc des listes chaînées d'arènes. Comme la gestion des arènes reste occasionnelle, il est possible de concevoir des routines d'allocation mémoire très rapides.

L'utilisation d'arènes pour les structures de données permet de palier à tous les problèmes de performances liés à la gestion mémoire conventionnelle.

- Les routines d'allocation peuvent être rapides. Dans une nouvelle arène, l'allocation d'une zone mémoire peut se réduire à l'incrémentation d'un pointeur. Les tâches moins fréquentes, comme l'allocation d'une nouvelle arène, influant peu sur la performance de la bibliothèque. Pour trouver une zone libre dans une arène, il suffit d'analyser la table d'allocation dans l'entête de l'arène. Il n'y a pas à gérer de liste de blocs mémoire libres, comme dans les bibliothèques optimisées standard.
- Le gaspillage mémoire est réduit de façon significative. Utiliser quelques bits dans l'entête de l'arène consomme moins de mémoire que les étiquettes accolées par les bibliothèques standard aux blocs mémoires alloués. De plus, comme dans une arène la taille des instances est connue précisément, il n'y a pas besoin d'arrondir la taille des blocs mémoire pour gérer des listes de blocs mémoire libres.
- La localité spatiale entre instances allouées n'est pas entravée.
  - En accolant des étiquettes aux blocs mémoires dont l'utilisation n'est pas corrélée avec le contenu des blocs alloués, les bibliothèques standard éloignent en mémoire les instances allouées consécutivement. Ce problème n'apparaît plus avec les arènes. Ce problème peut se révéler lorsque le programme manipule beaucoup de données en mémoire. Le système d'exploitation commence

à *swapper* beaucoup plus tôt avec une bibliothèque standard qu'avec notre bibliothèque optimisée. En effet, comme données et étiquettes sont conservées dans les même pages de mémoire virtuelle, le système a plus de mal à choisir quelles pages *swapper* (le *working-set* est plus gros). Avec notre bibliothèque, le système *swappe* d'abord les pages contenant les informations liées à l'allocation.
  - Les zones de l'arène sont allouées dans l'ordre, donc les instances d'une même structure allouées successivement ont de fortes chances d'être allouées consécutivement en mémoire. Avec une bibliothèque standard, ces éléments peuvent être n'importe où en mémoire, en fonction des blocs de mémoire disponibles, ce qui réduit les chances de tirer profit de la localité spatiale entre instances d'une structure.
- La localité de la bibliothèque d'allocation est élevée. Pour gérer la mémoire, la bibliothèque n'a besoin de scruter que l'entête des arènes.
- Le nombre d'appels système est réduit. Notre bibliothèque ne demande plus de mémoire au système que lorsque les arènes sont pleines. Dans ce cas, elle alloue une nouvelle arène qui pourra recevoir un grand nombre d'instances. Une bibliothèque standard par contre peut avoir à demander de la mémoire au système à chaque allocation d'une nouvelle instance.

5.4.2 Mise en oeuvre pratique

La bibliothèque *ialloc* a été écrite en langage C. Elle gère le tas avec des arènes. Elle se greffe par-dessus la bibliothèque d'allocation standard. Son interface n'est pas standard, ce qui permet de d'utiliser en parallèle la bibliothèque d'allocation standard pour les données non optimisées, et notre bibliothèque pour gérer l'allocation de structures entrelacées.

Le plus souvent, seul un petit nombre de structures de données nécessitent d'être optimisées. *Ialloc* gère donc pour chacune de ces structures une liste chaînée d'arènes permettant d'accueillir les instances allouées.

Il est donc décrit en premier l'organisation globale de la bibliothèque d'allocation permettant de gérer les listes d'arènes pour chaque structure, avant que soit précisé comment chaque arène est gérée.

5.4.2.1 Les listes d'arènes

Comme une arène contient un nombre limité d'instances d'une même structure, *ialloc* gère pour chaque structure de donnée une liste d'arènes distincte.

L'accès à une liste d'arènes se fait par l'intermédiaire d'un tableau statique de pointeurs sur les différentes listes, comme en figure 5.6. Toutes les structures allouées avec *ialloc* sont assignées un index arbitraire, correspondant à l'index du pointeur dans le tableau.

Lorsqu'une liste contient plusieurs arènes, comme la liste blanche de la figure 5.6, la première arène de la liste est toujours celle qui est en cours d'allocation. Les autres ont

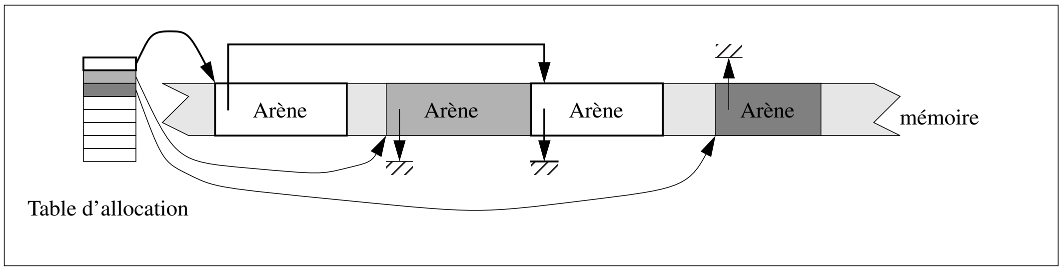


FIG. 5.6 – *Gestion mémoire globale.*

déjà été entièrement allouées (elles peuvent cependant quand même contenir des zones libres si des instances ont été libérées). On peut remarquer que tant qu'il n'y a pas de réallocation dans les arènes, les arènes les plus lointaines dans la liste contiennent les instances les plus anciennes. Elles ont donc plus de chances de contenir un grand nombre de zones libres.

Pour chaque structure de données, le tableau contient le pointeur sur la liste d'arènes, ainsi qu'un pointeur sur la prochaine zone libre dans la première arène. Ainsi l'allocation d'une instance peut être très rapide.

Pour faciliter la gestion mémoire, l'interface des routines de la bibliothèque *ialloc* n'est pas standard. Les routines d'allocation ont plus de paramètres, ce qui permet d'avoir plus d'informations afin de faire une gestion mémoire plus efficace et aussi de pouvoir gérer l'allocation de structures entrelacées. Les informations supplémentaires [5] nécessaires sont des informations connues statiquement : la taille de l'entrelacement si la structure est entrelacée, et le numéro d'index de la structure correspondant à l'indexe dans la table d'allocation de la liste d'arènes.

5.4.2.2 Les arènes

Notre bibliothèque d'allocation s'articule autour de l'allocation et de la gestion d'arènes. L'utilisation d'arènes permet d'obtenir une bibliothèque d'allocation rapide, ayant de bonnes propriétés de localité, gaspillant peu de mémoire et ne dégradant pas la localité spatiale qu'il peut y avoir entre les données allouées consécutivement en mémoire.

La figure 5.7 schématise l'organisation d'une arène. Les arènes sont composées d'une entête et d'une zone d'allocation.

Une arène n'est utilisée que pour allouer les instances d'une même structure de donnée. L'entête contient toutes les informations nécessaires pour gérer la zone d'allocation. Ces informations, indicées figure 5.7, sont décrites ci-dessous :

- _Fill_ est juste une zone qui permet d'aligner la zone d'allocation sur une frontière de ligne de cache.

5. Notre bibliothèque utilise des informations supplémentaires pour allouer des blocs mémoire, mais pas pour les libérer.

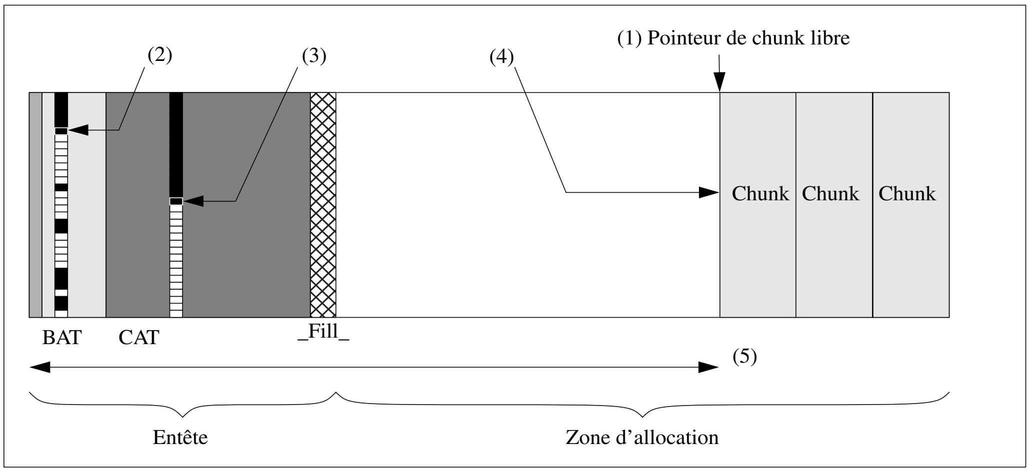


FIG. 5.7 – *Gestion d'une arène.*

- La CAT, *Chunk Allocation Table*, est une table d'allocation contenant un bit (3) par bloc de l'arène (4). Ce bit est à un si le bloc est alloué, à zéro sinon.
  Lorsqu'une arène est créée, les blocs sont systématiquement alloués en partant de la fin de l'arène en remontant jusqu'à l'entête. Ainsi, pour allouer un bloc, il suffit de décrémenter le pointeur de bloc libre (1), et de mettre à jour l'entête de l'arène.
  Lorsque le pointeur atteint le premier bloc de l'arène, pour allouer un nouveau bloc, il faudra trouver de l'espace qui a été libéré. Cette recherche peut être longue s'il faut regarder tous les bits de la CAT pour en trouver un à zéro.
  Pour accélérer ce processus, les blocs consécutifs de l'arène sont regroupés logiquement et la BAT est ajoutée dans l'entête.
- La table d'allocation des groupes ou BAT, acronyme pour *Block Allocation Table*, contient un bit par groupe de blocs. Lorsqu'un groupe est utilisé, son bit dans la BAT est à un. Lorsque tous les blocs du groupe sont libérés, son bit est remis à zéro.
  Lorsqu'on cherche une zone libre où allouer un bloc, on ne cherche pas juste un bloc libre, mais tout un groupe. Dans notre implémentation, un groupe contient 32 blocs. La recherche est donc plus courte, car la BAT est 32 fois plus petite. De plus, une fois qu'on a trouvé un groupe libre, on peut allouer rapidement toute une série de blocs, en ne mettant à jour que la CAT (3).
  De plus, avec cette solution, les blocs alloués en même temps restent contigus en mémoire. Comme les données allouées en même temps ont tendance à être utilisées en même temps, l'utilisation de la BAT peut améliorer la localité spatiale du programme.
- L'entête contient aussi un pointeur sur les arènes suivantes pour la gestion des arènes, deux compteurs (un compteur de blocs occupés et un de groupes de blocs occupés), et la taille des blocs de l'arène, pour un total de 13 octets supplémen-

taires.

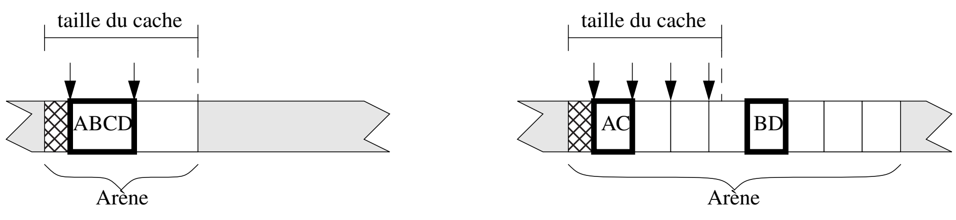
Le pointeur sur le prochain bloc libre (1) n'a pas à être dupliqué dans chaque arène. Il est donc conservé dans le tableau statique d'allocation, avec le pointeur sur la liste des arènes de la structure.

Les arènes sont positionnées sur des frontières de cache en mémoire, et tous les blocs qu'elle contient ont des adresses telles que l'adresse de l'entête de l'arène puisse être calculée en prenant l'adresse du bloc modulo la taille du cache. Un bloc n'est donc jamais éloigné de plus de la taille du cache de l'entête (5), ce qui fixe le nombre d'éléments que l'on peut allouer dans chaque arène. Ainsi, lorsqu'on doit libérer un bloc, on calcule facilement l'adresse de l'entête de l'arène pour mettre à jour les tables d'allocation.

Potentiellement, la nécessité d'avoir les arènes alignées sur des frontières de cache peut causer de gros gaspillages mémoire (jusqu'à 50% de la mémoire utilisée). Notre bibliothèque évite le plus souvent ce problème, car elle alloue des arènes de taille plus petite que l'alignement à faire afin de laisser de la place à la bibliothèque *malloc*. De plus notre bibliothèque peut s'affranchir des contraintes de la bibliothèque *malloc* en allouant la mémoire avec les bibliothèques *mmap* ou *shmat*. D'une manière générale, s'il y a de l'espace mémoire gaspillé pour aligner les arènes, cela n'influe pas sur la performance du programme, car les pages de mémoire virtuelle complètement inutilisées sont *swappés* rapidement sur disque par le système d'exploitation et n'encombrent pas la mémoire vive.

5.4.2.3 Entrelacement et taille des arènes

La taille des arènes est conditionnée par les calculs de pointeurs que doit faire notre bibliothèque d'allocation pour accéder à l'entête de l'arène. Lorsqu'une arène est utilisée pour allouer des structures entrelacées, elle devient plus grande que la taille du cache. Sa taille est conditionnée par la taille des groupes de champs.



(a) Instances non entrelacées (b) 2 groupes entrelacés

FIG. 5.8 – *L'entrelacement change la taille de l'arène.*

La figure 5.8(a) montre le cas d'une arène contenant des instances d'une structure non entrelacée. La taille de l'arène est limitée à la taille du cache, car il faut pouvoir calculer l'adresse de l'entête de l'arène à partir d'un pointeur sur une instance. L'entête de l'arène est alignée sur une frontière de cache pour que son adresse puisse être obtenue en mettant juste les bits de poids faible du pointeur à zéro.

Dans le cas d'une structure entrelacée, cette contrainte s'applique aussi, mais au lieu d'être limité par la taille de la structure, on est limité par la taille des groupes de champs entrelacés. En effet, le pointeur sur une instance pointe sur le premier groupe de champs. Comme les premiers groupes des instances sont regroupés au début de l'arène, ils sont plus proches de l'entête. On peut donc mettre plus d'instances dans l'arène, les groupes de queue contenant (B,D) étant séparés des groupes (A,C), ils n'influent pas sur l'adresse des pointeurs, figure 5.8(b). Dans ce cas où la structure est décomposée en deux groupes de champs, l'arène contient deux fois plus d'instances.

On alloue deux fois plus d'instances dans l'arène, mais l'entête reste de même taille, donc l'arène ne double pas tout à fait de taille. On utilise cette propriété pour faire en sorte que les groupes de champs d'une même instance ne rentrent pas en conflit dans le cache, lorsque le programme accède à l'instance en entier. Pour une instance, les champs appartenant à différents groupes de champs sont séparés en mémoire d'une distance égale à la taille du cache moins la taille de l'entête. Comme l'entête est plus grande qu'une ligne de cache, les groupes de champs ne sont jamais chargés dans le même ensemble de lignes de cache (*set*), ce qui élimine les risques d'interférences.

L'entrelacement des structures modifie légèrement le comportement de la bibliothèque d'allocation, car avec une arène contenant 2 fois plus d'instances, il y a 2 fois moins d'arènes à créer. Il est fait en sorte qu'entrelacer une structure ne nuise pas à la performance du programme, en évitant les interférences entre champs d'une même instance.

5.4.2.4 Fonctions de la bibliothèque

La bibliothèque ialloc propose 5 fonctions pour allouer et libérer la mémoire. Leur mode d'emplois est récapitulé ici afin d'éclaircir le mode d'utilisation de la bibliothèque d'allocation.

```
"ialloc.h"
1. extern (type *) ialloc(type, int chunksize, int typeindex);
2. extern (type *) balloc(type, int typeindex);
3. extern void ifree(void * ptr);
4. extern void iallocinit(void);
5. extern void iallocstats(void);
```

1. alloue une instance d'une structure de données entrelacée composée de blocs de taille *chunksize*, dans une des arènes de la liste d'arènes *structindex* qui a été arbitrairement assignée à la structure de données lors de la phase d'optimisation.
2. alloue une instance d'une structure de donnée non entrelacée dans une arène réservée à la structure *structindex*.
3. libère l'instance sur laquelle pointe le pointeur *ptr*. Elle doit nécessairement avoir été allouée précédemment avec *ialloc()* ou avec *balloc()*.

4. initialise la bibliothèque d'allocation. Il n'est utile d'appeler cette routine au début du programme que si l'initialisation automatique est désactivée.
5. affiche les statistiques d'utilisation des arènes. A utiliser de préférence lorsque la bibliothèque est compilée en mode *debug*, afin d'obtenir le plus d'informations possibles.

5.4.2.5 Paramètres de configuration

En plus de ces routines, l'interface de la bibliothèque contient des paramètres de configuration qui permettent d'adapter la bibliothèque à l'architecture de la machine. La taille de cache pour laquelle l'arène est prévue est fixée à 32KO, avec des lignes de 32 octets.

En théorie, l'arène peut être aussi large que le cache. En pratique, la taille de la zone d'allocation est fixée de manière à ce que l'arène fasse 32KO - 256 octets, pour éviter les problèmes d'alignement des arènes. En effet, la bibliothèque d'allocation standard *malloc* utilisée en *back-end* gère la mémoire à l'aide d'étiquettes. Dans ce cas, lorsqu'on alloue 32KO de mémoire, la bibliothèque *malloc* alloue 32KO + 8 octets. Comme il faut aligner les arènes sur une frontière de 32KO, cela induirait un gaspillage mémoire de 32KO - 8 octets entre chaque arène. Empiriquement, allouer 32KO - 256 octets semble le plus efficace pour éviter le gaspillage mémoire. Ce problème ne se rencontre pas lorsqu'on utilise *mmap* ou *shmat* pour allouer la mémoire, mais ces bibliothèques ne sont pas présentes sur toutes les plates-formes.

La taille de l'entête de l'arène dépend des paramètres de configuration (taille de la CAT et la BAT par exemple).

La taille de la zone d'allocation est contrainte par le calcul d'adresse qu'il faut faire pour retrouver l'entête de l'arène à partir d'un pointeur sur un bloc de l'arène, c'est à dire la taille du cache. Cela borne la taille de la CAT et de la BAT. Cependant, la taille de la CAT dépends de la taille minimum que peut avoir un groupe de champs dans une structure entrelacée. La taille minimum d'un groupe est fixée à quatre octets, la taille d'un mot mémoire, car les structures contiennent en général au moins un pointeur. Le nombre de mots de la CAT est donc calculé approximativement[6] par :

$$\left\lceil \frac{taille\ de\ l'arene}{sizeof(unsigned\ long) \times taille\ min\ des\ blocs} \right\rceil$$

Pour notre implémentation, la CAT fait exactement 1016 octets.

La BAT doit être suffisamment large pour gérer tous les groupes de blocs. Dans notre implémentation, un groupe contient 32 blocs. La BAT fait donc $\frac{1026}{32} = 32$ octets.

Il en résulte que _Fill_ doit faire 27 octets pour aligner le début de la zone d'allocation d'une arène.

Les choix de configuration que nous avons faits pour la bibliothèque permettent la plus grande souplesse avec un gaspillage mémoire minimal.

6. Il faut déduire la taille de l'entête lorsqu'on calcule la taille de la zone d'allocation. La zone d'allocation fait donc un peu moins que la taille de l'arène.

Il est gaspillé [7] 0,78% de mémoire pour éviter les problèmes d'alignement d'arènes. De plus, pour gérer l'arène, l'entête fait 1088 octets, soit 3,3% de gaspillage par arène.

Le gaspillage mémoire induit par la méthode de gestion mémoire qu'utilise *ialloc* est bien moindre qu'avec une bibliothèque standard : Par exemple, avec des structures non entrelacées de 12 octets, le gaspillage mémoire est de 4,12%, soit 3,96 bits par instance allouée. Pour des instances non entrelacées de 64 octets, le gaspillage est de 4,10%, soit 21 bits. A titre comparatif, avec une bibliothèque standard, des étiquettes de 2 mots (64 bits) sont accolées à chaque bloc alloué, soit respectivement 66% et 12,5% de pertes, sans compter l'effet de la fragmentation. Avec l'entrelacement des structures, la taille des arènes augmente, ce qui réduit d'autant le gaspillage mémoire induit par l'entête de l'arène.

5.4.2.6 Organisation interne des routines

Les routines de la bibliothèque sont découpées par fonctionnalité de façon à accélérer l'exécution, d'éviter la pollution du cache d'instruction, et d'autoriser l'inlining lorsque cela est possible [8].

$$\left.\begin{array}{l} \mathit{balloc}()\dagger \\ \mathit{ialloc}()\dagger \end{array}\right\} \rightarrow \text{\_\_ialloc()}\ddagger \rightarrow \left\{\begin{array}{l} \text{iallocinit()} \\ \text{\_\_GetAFreeBlock()} \rightarrow \text{\_\_alloc\_arena()}\ddagger \end{array}\right.$$

$$\begin{array}{l} \mathit{ifree}()\dagger \quad \rightarrow \text{void \_\_ifree()}\ddagger \\ \mathit{iallocstats}() \end{array}$$

†L'interface de la bibliothèque est faite par des *define* qui appellent les fonctions adéquates avec les bons paramètres.

‡Les routines principales de gestion mémoire sont petites. Elles ne font que les opérations essentielles pour pouvoir être *inline* lorsque le compilateur le supporte, sans augmenter la taille du code de façon inutile. De plus, les routines utilisées occasionnellement sont séparées, ce qui évite la pollution du cache d'instruction.

– __ialloc() appelle __GetAFreeBlock() s'il n'y a plus de blocs libres accessibles rapidement. Le plus souvent, elle calcule l'adresse du prochain bloc libre en décrémentant juste le pointeur de blocs libres, puis elle met à jour la CAT et le nombre de blocs libres dans le groupe. Si le compte tombe à zéro, une vérification simple de la BAT est faite pour choisir rapidement un nouveau groupe de blocs libres avant de passer la main à __GetAFreeBlock().

7. Il est entendu par gaspillage mémoire, tout espace mémoire qui n'est pas utilisé pour contenir une donnée du programme. Souvent, dans les bibliothèques d'allocation standard, l'espace utilisé par la bibliothèque pour gérer la mémoire n'est pas considéré comme gaspillé, ce qui réduit les statistiques sur les pertes.

8. Pour bénéficier de l'inlining avec les compilateurs C actuels, il faut non seulement inclure les entêtes de la bibliothèque, mais aussi inclure les sources dans les fichiers source du programme faisant appel aux routines d'allocation. Les compilateurs C ne savent en effet pas faire l'inlining de procédures externes au fichier source.

- __ifree() calcule l'adresse de l'entête afin de mettre à jour la CAT ainsi que la BAT si un groupe de blocs se libère.
- Les actions lentes ou exceptionnelles sont gérées dans des procédures distinctes afin de minimiser la taille des routines principales Cela inclut l'initialisation, gérée par __init_ialloc(), les recherches complexes de blocs disponibles parmi toutes les arènes de la liste faites par __GetAFreeBlock() ou l'allocation d'une nouvelle arène par __alloc_arena().

5.4.3 Utilisation de la libraire

Les modifications à apporter aux sources d'un programme pour exploiter la bibliothèque d'allocation *ialloc* sont décrites brièvement. Deux cas sont possibles : l'allocation d'instances d'une structure normale sans entrelacement, et l'allocation d'instances d'une structure qui a été modifiée pour être entrelacée.

5.4.3.1 Allocation normale

Pour allouer des structures avec la bibliothèque *ialloc*, il suffit de remplacer *malloc()* et *free()* par *balloc()* et *ifree()*, et de définir un index pour chaque structure allouée avec *balloc*.

Par exemple,

```
include <stdlib.h>
typedef struct _Foo {
        struct _Foo * next;
        int titi;
        char toto;
} Foo;

main() {
        Foo * bar = (Foo *) malloc(sizeof(Foo));
        bar->titi = 1;
        free(bar);
}
```

devient:

```
   include <stdlib.h>
1. include "ialloc.h"
2. #define FooIndex 0
   typedef struct _Foo {
           struct _Foo * next;
           int titi;
           char toto;
   } Foo;

   main() {
3.         Foo * bar = balloc(Foo, FooIndex);
4.         bar->titi = 1;
3.         ifree(bar);
   }
```

La transformation des sources comporte trois étapes simples.

1. Il faut tout d'abord inclure la bibliothèque *ialloc*.
2. Il faut définir arbitrairement un index différent pour chaque structure allouée avec la bibliothèque *ialloc*.
3. Il faut modifier les appels à *malloc()* et *free()*. Pour éviter les erreurs lors des modifications, l'interface des routines est conçue de manière à ce qu'il n'y ait plus besoin de faire de *cast* des pointeurs. *balloc()* calcule directement le type de la structure.
4. Les autres références aux instances de la structure ne sont pas modifiées.

Comme le montre l'exemple, l'utilisation de la bibliothèque *ialloc* ne nécessite que très peu de modifications du programme. Ces changements peuvent même être automatisés et mis en place dans un précompilateur.

5.4.3.2 Allocation d'instances entrelacées

La bibliothèque d'allocation optimisée permet d'allouer des instances de structures de données entrelacées, une optimisation de la localité spatiale des données décrite chapitre 5.3.2.

Pour allouer une structure entrelacée, il faut modifier la structure pour ajouter des matrices de remplissage, et remplacer les appels à *malloc()* et a *free()* :

Par exemple,

```
include <stdlib.h>
typedef struct _Foo {
        struct _Foo * A;
        int B;
        int C;
        short D;
} Foo;

main() {
        Foo * bar = (Foo*) malloc(sizeof(Foo));
        bar →B = 1;
        free(bar);
}
```

devient:

```
   include <stdlib.h>
1. include "ialloc.h"
2. #define FooIndex 0
3. #define FooChunk sizeof(struct _Foo *) + sizeof(int)
   typedef struct _Foo {
           struct _Foo * A;
           int C;
4.         char Pad1[_IA_INTERLEAVE - sizeof(struct _Foo *) - sizeof(int)];
           int B;
           short D;
4.         char Pad2[_IA_INTERLEAVE - sizeof(int) - sizeof(short)];
   } Foo;

   main() {
5.         Foo * bar = ialloc(Foo, FooChunk, FooIndex);
6.         bar→B = 1;
5.         ifree(bar);
   }
```

1. En premier lieu, il faut inclure la bibliothèque *ialloc*.
2. Il faut ensuite choisir arbitrairement un index différent pour chaque structure allouée avec *ialloc*, ici l'index choisi est zéro.
3. Les champs fréquemment utilisés, ici A et C, sont regroupés ensemble. Les autres champs sont organisés de manière à former des groupes approximativement de même taille (ou plus petits). Le groupe (A, C) fait 8 octets, alors que le groupe (B, D) en fait 6. *FooChunk* représente la taille du plus gros groupe fabriqué.
4. Il faut aussi insérer dans la structure de donnée des matrices de remplissage entre les groupes de champs (A, C) et (B, D). Elles sont toujours de taille

_IA_INTERLEAVE (prédéfini dans *ialloc.h*) moins la taille du groupe à entrelacer (8 ou 6 octets dans le cas présent). Systématiser ainsi la déclaration des matrices de remplissage évite de faire des erreurs de calcul sur la taille des matrices de remplissage.
Si la taille d'un groupe est plus petite que *FooChunk*, l'espace restant sera gaspillé, et on perdra de la localité spatiale sur ce groupe (par exemple on perd 2 octets après le champ D). Cela n'a pas d'importance si le gaspillage est fait sur un groupe de champs peu référencé comme (B, D).
5. La dernière étape consiste à modifier les appels à *malloc()* et *free()*. Il n'est plus nécessaire de faire de *cast* de pointeur, car ialloc() récupère directement le type. *Ialloc()* prend 3 paramètres en entrée connus à la compilation. Il lui faut le type de la structure, la taille du plus gros bloc entrelacé, et l'index qui a été assigné arbitrairement à la structure.
6. Les nombreuses références aux champs de la structure dans le reste du programme restent inchangées.

Les modifications nécessaires pour utiliser ialloc() sont peu nombreuses et systématiques. Elles peuvent être faites par un outil automatique. Un utilisateur peut aussi facilement adapter son code pour utiliser ialloc, s'il suit l'approche décrite ci-dessus.

5.4.3.3 Contraintes d'utilisation

En C, l'allocation dynamique n'est pas gérée par le compilateur mais se fait avec une bibliothèque d'allocation. Par conséquent nos optimisations ne sont faisables qu'en modifiant à la fois le programme et la bibliothèque d'allocation. Le langage C offre une grande souplesse d'utilisation aux programmeurs, ce qui peut générer des conflits avec nos optimisations.

Les problèmes qui peuvent être rencontrés sont recensés ici. Des solutions sont proposées pour les contourner :

- Allocation de données de type autre que des structures de données.
  Nous n'étudions pas ces cas là, et ces données restent gérées par la bibliothèque d'allocation standard qui reste accessible dans les programmes optimisés.
- Allocation de tableaux de structures avec *calloc()*.
  La bibliothèque limiterait au mieux la taille des tableaux qui peuvent être alloués à la taille des arènes. Il n'a donc pas été inclus de support pour gérer les tableaux de structures.
  Si les données du tableau doivent être entrelacées, il faut les allouer avec *ialloc()*. Dans ce cas, il est possible de transformer le tableau en un tableau de pointeurs sur les instances de structures allouées avec *ialloc()*. Cette solution n'est pas la meilleure, car cela génère une indirection supplémentaire qui ralentit l'exécution. De plus elle nécessite de modifier le code pour convertir les références de tableaux $Tab[i].Field$ en $Tab[i] \rightarrow Field$.
  Parfois, le programmeur utilise un tableau de structures juste pour faciliter l'allocation, mais utilisera les objets alloués séparément. Dans ce cas, l'organisation en

tableau est inutile et peut être supprimée pour utiliser *ialloc()*. Une bibliothèque supportant l'allocation de matrices de structures entrelacées peut être envisagée, mais son utilisation nécessiterait quand même de modifier tous les accès aux champs des instances dans la matrice. Il faut en effet obligatoirement ajouter des *cast* pour obliger le compilateur à faire les bons calculs d'adresses .

- Appels à *realloc()*.
  Agrandir/rétrécir un tableau de structures entrelacées pose le même problème qu'avec *calloc()*. Cela est donc géré de la même manière.
- Assignation d'instance à instance.
  Lorsqu'on assigne un objet à un autre, en C ANSI ou en C++, cela revient à recopier tous les champs d'un objet dans l'autre. Si les objets sont entrelacés, ce n'est pas désirable, car on recopie aussi les champs utilisés pour le *padding* (les champs *Pad[]*, section 5.4.3.2). Il faut donc remplacer cette assignation par une assignation champs par champs. Cela peut entraîner un surcoût d'instructions s'il y a beaucoup de champs (on remplace un *memcpy()* par des copies champs par champs).
  Ce problème n'a pratiquement jamais été rencontré[9], car peu de programmes C exploitent cette propriété qui n'est disponible qu'avec les compilateurs ANSI.
- Allocation mixte.
  Il n'est pas possible d'allouer des structures entrelacées dans la pile ou statiquement, car les matrices de remplissage ajoutées pour l'entrelacement sont très grosses, et gaspilleraient beaucoup de mémoire. Dans ce cas, il faut convertir les instances statiques ou locales en instances allouées dynamiquement. L'allocation étant rapide, cela n'engendre pas de perte de temps notable.
- Allocation de différentes structures par un même site d'appel à *malloc()*.
  Si un site d'appel à *malloc()* correspond à l'allocation de différents types de structures, il n'est pas possible de remplacer simplement l'appel par un appel à *ialloc()*, car il faut fournir des paramètres supplémentaires dépendant de la structure allouée. Cela se rencontre dans les programmes ou les appels à la bibliothèque d'allocation sont encapsulés dans des routines, soit pour gérer les erreurs d'allocation, soit parce que le programmeur gère lui-même l'allocation mémoire. Dans ce cas, il faut passer outre les routines encapsulant l'appel à *malloc()*.
- Libération de différentes structures par un même site d'appel à *free()*. Pour remplacer *free()* par *ifree()*, il faut que toutes les structures libérées par le site d'appel correspondent à des structures allouées par *ialloc()* ou *balloc()*.
- *Cast* entre plusieurs structures ou héritage.
  Un objet héritant d'un parent en C++ peut être référencé juste par un pointeur de la classe du parent. De même, en C on peut définir plusieurs structures avec des champs communs, référencés avec un pointeur générique (c'est le cas par exemple dans le Gnu C). Ceci implique dans ce cas, que les structures doivent avoir le

9. Lorsque les données sont allouées dynamiquement, il est très rare de rencontrer des assignations du type *(*bar1) = (*bar2);*. Ce sont surtout des assignations entre pointeurs *bar1= bar2;* qui sont rencontrées. Elles coûtent moins cher et sont plus appropriées à ce style de programmation.

même entrelacement pour les champs en commun. Cela peut rendre l'analyse à faire plus compliquée.

Les problèmes énumérés ci-dessus se rencontrent lorsqu'une application est convertie pour utiliser la bibliothèque d'allocation *ialloc*. Un programme pensé pour tirer parti de *ialloc* ne rencontre pas ces problèmes, et peut facilement être converti pour utiliser *malloc*.

Le problème le plus fréquent est l'utilisation de *calloc()*. Une solution élégante à ce problème permettrait d'optimiser plus facilement les applications.

5.4.4 Limitations de la bibliothèque d'allocation

La bibliothèque *ialloc* est conçue dans l'optique d'étudier les techniques de placement mémoire sur les structures de données. Cependant, une telle bibliothèque pourrait être utilisée comme complément à *malloc* pour offrir une alternative aux applications nécessitant plus de performance. Afin de rendre exploitable une telle bibliothèque, il faudrait cependant la reconcevoir afin d'éliminer les défauts de conception qui réduisent son champ d'application.

5.4.4.1 Des contraintes de conception

Lors de l'utilisation de notre bibliothèque, plusieurs contraintes d'utilisation sont apparues, causées par la conception de la bibliothèque.

- *Ialloc* a été conçue pour allouer des instances indépendantes de structures de données. Le cas des tableaux de structures n'a pas été pris en compte. La bibliothèque a été conçue pour utiliser la bibliothèque *malloc* ou *sbrk*, qui ne permettent pas d'allouer de la mémoire à des adresses arbitraires. La prise en charge des tableaux devient alors impossible.
  Lors de cette étude, il a été remarqué que les tableaux de structures sont souvent utilisés dans les programmes manipulant beaucoup de données. Il est donc utile de fournir une bibliothèque capable de les gérer, en proposant des routines équivalentes à *calloc()* et *realloc()*. Un palliatif est proposé, la transformation du tableau de structures en un tableau de pointeurs sur la structure. Cela permet d'allouer des instances de structures entrelacées, mais ajoute le surcoût d'une indirection qui n'est pas nécessairement compensé par les gains de l'entrelacement.
- Les gains obtenus pour les structures de données indiquent que le développement de bibliothèques d'allocations adaptées à chaque type de données pourrait apporter des gains considérables. Parmi les types de données allouées fréquemment, les chaînes de caractères ne bénéficient d'aucune optimisation.

5.4.4.2 Gérer l'allocation des tableaux de structures

Il est possible de modifier la bibliothèque d'allocation de manière à supporter l'allocation de tableaux de structures. Pour cela, il faut créer des arènes suffisamment grandes pour pouvoir accueillir n'importe quel tableau de structures.

Pour cela, on peut tirer parti de l'adressage virtuel, en concevant la bibliothèque d'allocation autour de *mmap()* ou *shmat()*. Contrairement à *malloc*, ces deux bibliothèques permettent de tirer parti de la mémoire virtuelle en choisissant arbitrairement l'adresse en mémoire du bloc que l'on désire allouer. Il n'est plus nécessaire d'allouer des zones mémoires contiguës.

La zone d'adressage virtuelle des nouvelles machines est très grande par rapport à la mémoire disponible (2^{64} contre 2^{30}). Avec l'évolution actuelle de la consommation mémoire, cet espace d'adressage ne sera pas consommé avant une vingtaine d'années. On peut donc effectivement envisager d'exploiter l'espace d'adressage sans se soucier du gaspillage de quelques bits d'adresses. Sur les machines 32 bits, les programmes font le plus souvent moins d'une centaine de Mega-octets, ce qui permet d'exploiter là aussi quelques bits d'adresse pour gérer la mémoire.

Supposons que les programmes à optimiser utilisent moins de 256MO de mémoire et que l'on soit sur une machine 32 bits. On peut alors découper l'espace mémoire en 16 segments de 256MO. En réservant le premier et le dernier segment pour le tas, géré par *malloc*, et la pile, gérée par le CPU, il reste 14 segments disponibles pour accueillir les structures allouées avec la nouvelle bibliothèque. La possibilité d'avoir jusqu'à 14 arènes est suffisante pour couvrir les besoins des programmes.

Telle qu'elle a été conçue, notre bibliothèque ne peut pas être adaptée pour suivre ce modèle de gestion mémoire. Il ne s'agit plus en effet de gérer des listes d'arènes de taille fixe, mais de gérer une seule arène de taille variable pour chaque type de donnée. Il faut donc réécrire la bibliothèque. De plus, les compilateurs ont parfois des limitations quant à la taille que peut avoir une matrice (limitation sur le type de l'index utilisé pour accéder à la matrice). Définir des tableaux de remplissage de plusieurs centaines de Mega-octets peut poser des problèmes de compilation.

5.4.4.3 De nouveaux front-end?

Des techniques sont maintenant disponibles pour améliorer la localité des accès aux structures de données, aux matrices, et aux scalaires (respectivement la réorganisation et l'entrelacement des champs, les transformations de boucles et l'allocation dans les registres). Il existe d'autres types de données.

Les chaînes de caractères forment par exemple un type de donnée avec ses caractéristiques d'accès propre. Pour améliorer la localité des références aux chaînes de caractères, il est possible d'allouer avec chaque chaîne des signatures permettant d'accélérer les manipulations de chaînes les plus fréquentes. En utilisant les propriétés d'allocation et de libération de la mémoire pour les chaînes de caractères, une bibliothèque d'allocation dédiée très rapide pourrait être mise au point.

5.5 Expérimentation

Les techniques d'entrelacement et de réorganisation des champs ont été détaillées. Elles sont mises en pratique grâce à la bibliothèque *ialloc* sur plusieurs programmes qui utilisent des structures de données allouées dynamiquement et qui sont limités par la performance de la hiérarchie mémoire.

Les gains de performance proviennent aussi de l'optimisation de la bibliothèque d'allocation *ialloc*. Les effets de la bibliothèque d'allocation sont donc aussi étudiés.

- Réduction de la consommation mémoire totale et réduction de la partie résidente en mémoire du programme, grâce aux entêtes des arènes.
- Amélioration de la localité spatiale des données allouées car on évite la fragmentation interne, et on n'accole plus d'étiquettes aux blocs alloués.
- La vitesse de la bibliothèque d'allocation grâce à une gestion mémoire plus rapide et une meilleure localité des accès mémoires dans l'entête des arènes.
- La réduction des appels système car la bibliothèque alloue la mémoire arène par arène.

Ces gains s'ajoutent aux gains obtenus avec l'optimisation du placement des données. Il est parfois difficile de déterminer la raison réelle d'un gain de performance. L'impact de chacun de ces facteurs sur la performance des programmes est donc étudié séparément.

5.5.1 Protocole d'expérimentation

5.5.1.1 Mise en oeuvre

Les programmes originaux et optimisés ont été exécutés comme tâches monoprocesseur sur un processeur R10000 d'une *SGI Origin 2000*. L'*Origin 2000* est une machine multiprocesseur à mémoire distribuée. Les noeuds sont connectés par un hypercube. Chaque noeud de la machine est composé d'une carte biprocesseur et de 256 Mega-octets de mémoire vive SDRAM. Chaque processeur a un cache interne de donnée L1 de 32 Kilo-octets, associatif par ensemble de deux ; les lignes ont une largeur de 32 octets. Le cache externe a une configuration identique, mais fait quatre Mega-octets, et les lignes font 128 octets de large. Lorsqu'une application est exécutée, le système d'exploitation la place automatiquement sur le processeur le moins chargé. Les données sont allouées dans la mémoire locale du noeud, si cela est possible, puis dans les noeuds de l'hypercube les plus proches sinon. Lorsque la machine est peu chargée, un programme non parallélisé est principalement exécuté comme sur une machine monotâche monoprocesseur. La machine utilisée contient 64 processeurs, et 8 giga-octets de mémoire vive. Si moins de 64 processus sont exécutés simultanément, chaque processus sera assigné un seul processeur, ce qui réduit l'influence des autres programmes sur nos tests.

Sauf cas particulier noté explicitement, tous les programmes ont été compilés et testés avec le compilateur *C MipsPro Compiler* fourni par défaut sur l'*Origin*, avec toutes les optimisations du compilateur activées (réorganisation de boucle et analyse inter-procédurale inclues) par la commande :

"cc -Ofast=IP27 -LNO -IPA".

Les temps d'exécutions ont été obtenus avec la commande *ssusage*, équivalente à la commande *time* de Unix. La consommation mémoire totale et résidente des programmes est obtenue simplement avec *top*. Les autres statistiques ont été obtenues avec les compteurs matériels du R10000, grâce à *perfex*, un outil de *SpeedShop*, par la commande :

"perfex -y -a"

5.5.1.2 Choix des programmes

Les programmes sélectionnés pour les expérimentations proviennent principalement de suites de test développées pour d'autres recherches.

Pour être sélectionnés, les programmes doivent être écrits en langage C, utiliser des structures de données allouées dynamiquement et souffrir de la mauvaise performance de la hiérarchie mémoire.

Un sous-ensemble de ces programmes est présenté pour l'étude de l'entrelacement et la réorganisation des champs des structures. Ceux-ci ont une mauvaise performance de la hiérarchie mémoire causée par la mauvaise localité des accès mémoires faits par le programme. Certains des programmes ont en effet une mauvaise performance mémoire parce que la bibliothèque d'allocation est utilisée intensivement.

5.5.1.3 Les benchmarks Olden

Ces benchmarks ont été écrits pour étudier la parallélisation de programmes C avec le compilateur Olden développé pour la CM-5 [CR95, RCRH95]. Ils ont aussi été utilisés pour étudier le préchargement logiciel de structures de données récursives [LM96].

- *Bisort* effectue deux tris bitoniques sur des entiers aléatoires stockés dans un arbre binaire. Il y a une structure *node* de trois champs de quatre octets *left, right* et *value*.
  La structure est entrelacée en regroupant *left* et *value*.
- *Em3D* simule la propagation d'ondes électromagnétiques dans un objet en 3D. L'objet est représenté par un graphe biparti. Plusieurs structures sont utilisées.
  La structure *node_t* contient sept champs de quatre octets. L'entrelacement se fait champs par champs. Les deux champs de *graph_t* et *table_t* sont entrelacés de même manière.
- *Health* simule la gestion des hôpitaux colombiens. Il y a quatre structures, *village, hosp, list, patient*, qui servent à représenter les mouvements des malades entre les hôpitaux.
  Les quatre champs de *patient*, la fiche des patients, sont entrelacés. Il en est de même pour les trois champs de *list*, une liste doublement chaînée pointant sur les fiches de patients traités dans chaque hôpital. La structure *hosp* contient sept champs de quatre octets, et la structure *village* neuf champs de quatre octets qui n'ont pas été entrelacés.
- *Mst* calcule l'arbre couvrant d'un graphe. Il y a quatre structures de données utilisées.
  La structure *vertex* comportant trois champs, organisée en liste, est entrelacée champ par champ. Il en est de même pour les structures *Hash* et *HashEntry*.

- *Perimeter* calcule le périmètre d'une image digitalisée. L'image est encodée dans un arbre quaternaire, contenant la couleur du pixel, cinq pointeurs (quatre sur les fils et un sur le parent), et le type des fils.
  La structure est simplement entrelacée champs par champs.
- *Power* calcule le prix de revient de l'électricité d'une centrale pour ces utilisateurs. Le prix est propagé de la centrale vers les utilisateurs, et la consommation des utilisateurs est propagée jusqu'à la centrale. Le problème original est construit avec une arborescence de quatre structures de données différentes, une centrale *root* ayant 10 lignes principales, connectées à 20 lignes latérales *lateral*, connectées à cinq branches *branch* ayant chacune 10 clients *leaf* (soit un total de 10000 clients).
  Pour entrelacer les données, *branch* composée de six doubles et deux pointeurs, est entrelacée en regroupant quatre doubles et un pointeur ensemble, et *leaf* est entrelacée en regroupant les quatre doubles par deux.
- *TreeAdd* calcule récursivement la somme d'éléments stockés dans un arbre binaire composé de trois champs de quatre octets, *left, right, val.*
  La structure est entrelacée champs par champs.
- *Tsp* résout le problème du démarcheur en utilisant l'heuristique du point le plus proche. Le programme utilise un arbre quaternaire. La structure *tree* comporte deux champs de huit octets et cinq champs de quatre octets.
  Les champs sont entrelacés par groupes de huit octets (les pointeurs *next* et *prev* sont regroupés ensemble, de même que *left* et *right*).

TAB. 5.3 – *Paramètres d'entrée des benchmarks Olden.*

Taille du problème	Ligne de commande
$10\overline{M}$ entiers	BiSort 10000000 0
20K noeuds	EM3D 10000 1000 10000 1
1365 villages	Health 6 500 1
2K noeuds	MST 2048
image 90K x 90K	Perimeter 600 90000
20K clients	Power (NUM_FEEDERS 20 ; LATERALS_PER_FEEDER 10 ; BRANCHES_PER_LATERAL 5 ; LEAVES_PER_BRANCH 20)
$8\overline{M}$ noeuds	TreeAdd 23
$3\overline{M}$ villes	TSP 3276800 0

Les benchmarks Olden ont été exécutés avec les lignes de commandes indiquées table 5.3. Ces paramètres d'entrée sont plus grands que les paramètres d'origine utilisés dans [CR95] pour allonger les temps de calcul et stresser plus la hiérarchie mémoire.

5.5.1.4 Les benchmarks WPI

Les benchmarks synthétiques du *Worcester Polytechnic Institute* servent à évaluer la performance des systèmes d'exploitation Unix, et plus spécialement le système d'exploitation Mach [FKJ+90, FKLC92].

- *Jigsaw* est un programme qui résout un modèle mathématique d'un puzzle 2D toroïdal. Il se décompose en une phase d'initialisation, où les pièces sont organisées en liste chaînée, une phase de mélange des pièces et une phase de résolution. La structure de donnée *tile* contient 11 champs de quatre octets (44 octets). Quatre champs contiennent les numéros des pièces connectées sur les quatre côtés et quatre champs contiennent l'identité de chaque côté, les trois champs restant sont l'identité de la pièce, un pointeur sur la pièce suivante et un drapeau indiquant si l'élément est résolu.
  La structure de donnée est entrelacée champ par champ.

Jigsaw est utilisé dans WPI en ajoutant des champs supplémentaires dans la structure pour tester la gestion de la pagination de la mémoire virtuelle. Dans les tests, aucun champ supplémentaire n'est ajouté dans la structure *tile*. Le fichier d'entrée par défaut permet de résoudre un puzzle allant jusqu'à 64x64 pièces. Un puzzle 100x100 a été généré pour allonger le temps de calcul.

Les autres benchmarks n'ont pas été utilisés car ils sont conçus pour stresser d'autres parties de l'OS (I/O, appels systèmes, etc.). Par exemple, l'un des benchmarks représente une base de donnée client/serveur, mais comme il ne stresse que les transactions entre le client et le serveur, il n'a pas pu être exploité ici.

5.5.1.5 Les benchmarks Splash

Les collections d'applications de la suite Splash-1 [SWG92] et Splash-2 [WOT+95] sont des applications comportant entre 1000 et 10000 lignes de code, représentatives des grosses applications que l'on peut rencontrer. Les *Stanford Parallel Applications for Shared-Memory* ont été collectées pour évaluer la performance des machines à mémoire partagées.

Les applications de Splash-1 éligibles pour nos optimisations sont Barnes Hut, MP3D et PTHOR. Les autres manipulent principalement des matrices numériques.

- *PTHOR* est un simulateur *event-driven* de circuit électronique. Il est compilé avec une description comportementale du circuit (3 circuits au choix, un petit multiplieur, un petit circuit RISC composé de 6500 éléments, et le circuit DASH comportant 24611 éléments).

Les applications de Splash-2 éligibles pour nos optimisations sont Barnes Hut, FMM, Raytrace, Radiosity et VolRend.

- *Raytrace* est un programme de rendu d'image 3D par la méthode des lancer de rayons. Ce programme utilise de nombreuses structures de données.
  Les huit champs de la structure *object* ont été entrelacés un à un. La structure *sphere* contient trois champs. Les champs *center* et *rad2* ont été regroupés. *center* est une matrice de quatre doubles de huit octets.

Raytrace a été compilé différemment, avec les drapeaux "-O3 -n32 -R10000 -mips4" car sinon le programme retourne des résultats faux (l'option -Ofast inclus d'autres optimisations, comme l'analyse inter-procédurale et des optimisations au niveau du calcul flottant). Le fichier d'entrée est balls4.env, mais avec *gridsize = 2* pour allonger le temps de calcul.

5.5.1.6 Les autres benchmarks

D'autres applications ont été utilisées. *Radiosity* est une application développée à l'IRISA. Trois benchmarks synthétiques qui permettent de visualiser le gain de localité qu'apportent les différentes techniques d'optimisation ont aussi été écrits.

- *Radiosity* est un programme de rendu d'image par une technique de radiosité hiérarchique, développé à l'IRISA [MBM97]. Bien que conçu autour de la gestion mémoire, ce programme n'a pas été optimisé pour gérer au mieux l'allocation dynamique.
  L'entrelacement est fait sur plusieurs structures de données. Les quatre champs de LIST_SURFACE sont entrelacés un à un (trois champs sont des pointeurs de quatre octets, et un champ est un short de deux octets). Les trois champs de LISTE_PATCHES et de LIST_LINKS sont aussi entrelacés un à un (là encore, il y a deux pointeurs et un short). Les nombreuses autres structures de données ne sont pas modifiées.

Le jeu de données utilisé constitue une scène comportant 213 pièces meublées à rendre. La ligne de commande est *"Radiosity -i tt.gra -o tt -M 1 -T 10 -A 1 -E 0.0001 -N 1500 -s -m 2 -t 1 4"*. C'est à dire que le programme exécute quatre itérations en mode "glouton C".

Trois benchmarks synthétiques ont été créés pour montrer les phénomènes apparaissant avec les différentes optimisations. Ils sont tous trois conçus sur le même modèle. Ils allouent une liste chaînée d'éléments, et effectuent des recherches sur ces éléments. L'algorithme se déroule en trois phases.

1. Allocation et initialisation des éléments de la liste. Cette phase est d'autant plus rapide que la bibliothèque d'allocation est performante.
2. Test de validité de l'allocation. La relecture des valeurs stockées dans la liste chaînée permet de vérifier que les éléments ne se chevauchent pas en mémoire (ce qui peut se produire lors de l'allocation d'éléments entrelacés si la bibliothèque est mal conçue ou mal utilisée).
3. Répétition d'une série de recherches dans la liste. Ces recherches sont d'autant plus rapides que les données allouées ont de bonnes propriétés de localité. La performance de la bibliothèque d'allocation n'influe plus.

La différence entre ces trois applications est la taille des structures allouées.

- *SmallStruct* utilise des instances composées de 3 champs, soit 12 octets. La vitesse de la recherche est perturbée par la faible localité entre les éléments en mémoire, causée par exemple par l'ajout d'étiquettes aux blocs alloués qui réduit la localité spatiale dans les lignes de cache.

- *MedStruct* utilise des éléments composés de 8 champs, soit 32 octets (la taille d'une ligne de cache). Peu de champs sont utilisés pour la recherche. La faible performance est due à la faible localité qu'il y a entre les champs des éléments. Même si les étiquettes des blocs sont supprimées ou si les champs sont réorganisés, il n'y a pas augmentation de la localité dans les lignes de cache. Il faut entrelacer les instances pour accélérer la recherche.
- *BigStruct* utilise des éléments composés de 24 champs, soit 96 octets (3 lignes de cache). Peu de champs sont utilisés pour la recherche. De plus, ils sont dispersés dans la structure. Il faut donc utiliser plusieurs lignes de cache pour manipuler une instance. La réorganisation des champs permet de réduire le nombre de lignes à charger dans le cache pour référencer une instance pendant la recherche. Le programme fait de plus deux séries de recherches en utilisant des champs différents, afin de montrer qu'il est possible de regrouper les champs de façon à améliorer en même temps la vitesse des deux recherches.

SmallStruct et *MedStruct* sont utilisées avec la ligne de commande “4096 1024 1”. Ils effectuent un million de recherches sur quatre millions d'éléments. La ligne de commande de *BigStruct* est “2048 1024 1” soit deux séries d'un million de recherches sur deux millions d'éléments. Le “1” indique que les recherches démarrent au début de la liste. Un “0” indique que les recherches continuent à partir du dernier élément trouvé.

5.5.2 Influence de la bibliothèque d'allocation

Cette section montre les différents types de gains qu'apportent les optimisations de la bibliothèque d'allocation.

5.5.2.1 Performance générale de la bibliothèque d'allocation

Pour évaluer la performance de la bibliothèque, il suffit d'utiliser un petit programme qui alloue et libère une structure de trois champs en mémoire autant de fois qu'on le désire. On peut ainsi faire les expériences suivantes.

- La première expérience consiste à allouer 256K instances et à les libérer, ce 64 fois de suite, afin d'évaluer la performance de la réallocation. Pour minimiser les accès mémoire, le programme se contente juste de relier les instances dans une liste chaînée pour pouvoir les accéder afin de les libérer.
- La seconde expérience consiste à allouer quatre millions d'instances pour tester l'allocation. Comme la mémoire allouée n'est pas libérée, le programme n'a pas à faire pointer les instances les unes sur les autres. Il ne référence donc même pas les instances allouées.
- La troisième expérience alloue quatre millions d'instances puis les libère, pour tester la libération. Comme pour la première expérience, libérer la mémoire né-

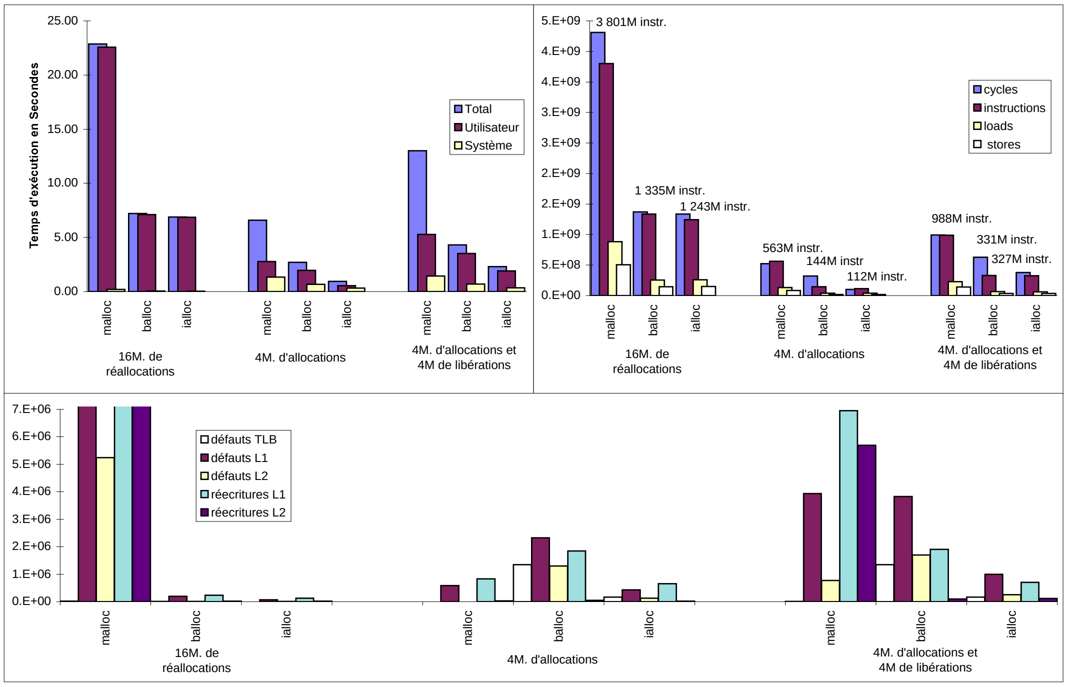


FIG. 5.9 – *Performance de la bibliothèque d'allocation.*

cessite de gérer les instances dans une liste chaînée et donc de lire et d'écrire un pointeur dans chaque instance allouée.

Les résultats de ces expériences sont illustrés figure 5.9 par le temps d'exécution des trois expériences (temps total, utilisateur et système), l'impact de la bibliothèque sur le nombre d'instructions exécutées (nombre d'instructions total, nombre de *load* et de *store*, le nombre de cycles est mis pour référence), et l'impact de la bibliothèque sur la localité du programme (nombre d'échecs et de réécritures pour les caches de données L1 et L2, ainsi que de la TLB).

Ces résultats ont été obtenus avec *time* et *perfex* sur une SGI Onyx avec des processeurs R10000, des caches primaires 2-associatifs par ensemble, non bloquants de 32KO à réécriture, et des caches secondaires de 1MO avec les même propriétés.

Sur la première expérience, on note que *ialloc* est beaucoup plus efficace pour réallouer la mémoire que la bibliothèque *malloc*, elle exécute trois fois moins d'instructions, elle est trois fois plus rapide et elle génère à tous les niveaux de la hiérarchie un nombre de défauts négligeable par rapport à ceux générés par *malloc*.

Pour l'allocation pure, la seconde expérience montre que *ialloc* exécute quatre fois moins d'instructions, mais n'est que deux fois et demi plus rapide, car elle a une bien moins bonne localité que la bibliothèque *malloc*, avec pour *balloc( )*un taux d'échecs sur L1 de 4% et de 56% sur L2.

Sachant que le programme fait 4 million d'allocations, on peut estimer d'après les résultats de la seconde expérience, qu'une allocation avec *malloc( )* coûte 133 instructions, contre 29,6 avec *balloc( )* et 24,6 avec *ialloc( )* (la différence avec *balloc( )* vient de l'utilisation d'arènes trois fois plus grandes). Ces évaluations sont légèrement surestimées, car elles incluent les instructions utilisées pour compter les itérations de la boucle d'allocation du programme.

Pour la troisième expérience, la bibliothèque *ialloc* exécute trois fois moins d'instructions et est à nouveau trois fois plus rapide. On peut donc penser que la bibliothèque standard génère peu de défauts de cache à l'allocation, mais voit sa performance mémoire se dégrader lorsqu'il faut gérer des blocs mémoire qui ont été libérés. Dans la première expérience on voit bien la dégradation qui s'opère. On note que contrairement à la bibliothèque standard, la bibliothèque *ialloc( )* a tendance à générer un nombre de défauts de TLB important lorsqu'il faut allouer des zones mémoires pour la première fois. Ceci est probablement dû à l'éloignement entre l'entête de l'arène et le bloc alloué, ce qui nécessite souvent l'accès à plusieurs pages de mémoire virtuelle.

La bibliothèque *ialloc* qui a été développée est souvent plus rapide que la bibliothèque standard. Elle divise en moyenne par trois le nombre d'instructions exécutées. Elle a aussi une très bonne localité mémoire lors de la réallocation, mais souffre d'une mauvaise localité spatiale lors de la première allocation des blocs.

5.5.2.2 Nombre d'instructions exécutées par la bibliothèque d'allocation

La vitesse de la bibliothèque d'allocation influe sur la performance des programmes qui font beaucoup d'appels à la bibliothèque d'allocations.

La vitesse de la bibliothèque d'allocation dépend de sa conception et du comportement du programme qui influe sur le nombre d'instructions nécessaires pour allouer de la mémoire. La bibliothèque sera plus rapide si elle est amenée à faire des opérations simples, et plus lente si elle doit effectuer la gestion de cas complexes non optimisés.

L'utilisation de notre bibliothèque peut donc se traduire par une réduction ou une augmentation du nombre d'instructions exécutées. La variation va d'une augmentation de 5,89% du nombre d'instructions pour Radiosity à une diminution allant jusqu'à 49,09% pour TreeAdd, comme le montre la figure 5.10 (les applications gagnantes sont encadrées, la variation des *loads* et des *stores* doit être corrélée à leur nombre en (b)). L'ampleur de la variation dépend aussi du pourcentage de temps que le programme passe dans la bibliothèque. Par exemple TreeAdd[10] passe 52,4% de son temps dans *malloc()*.

5.5.2.3 Réduction des appels système

Lorsqu'un programme alloue un grand nombre de petits blocs mémoire, la bibliothèque d'allocation standard peut être amenée à demander souvent de la mémoire au système.

Une partie des programmes testés ont un temps système important. La figure 5.11 montre que la bibliothèque *ialloc* réduit de façon significative le temps système de la

10. Résultat obtenu avec "*gprof -F main TreeAdd*" après avoir exécuté *TreeAdd 22* sur une Ultrasparc avec 120MO de mémoire.

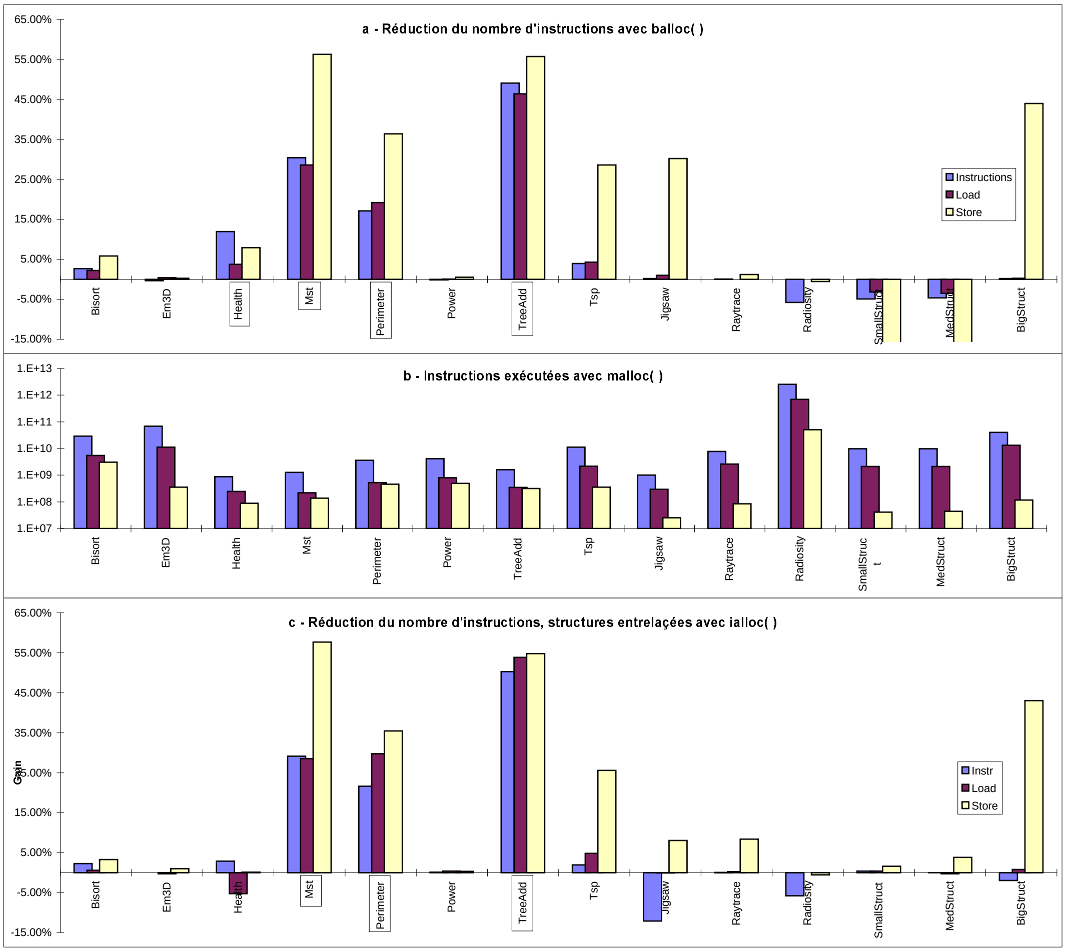


FIG. 5.10 – *Impact de la bibliothèque sur le nombre d'instructions exécutées.*

majorité de ces programmes. Leurs noms sont encadrés sur la figure. Les temps ont été normalisés par rapport au temps d'exécution réel (temps système plus temps utilisateur) du programme avec la bibliothèque *malloc* standard.

Lorsque la bibliothèque *ialloc* demande de la mémoire au système, c'est pour allouer une arène, et non un bloc mémoire. Les appels systèmes sont donc beaucoup moins fréquents qu'avec une bibliothèque conventionnelle.

Pour vérifier que le temps système mesuré est causé directement par la bibliothèque d'allocation, la mémoire utilisée par *TreeAdd*, 195MO, a été préalloué. On voit figure 5.12, que la préallocation élimine la plus grande partie du temps passé dans le système. La préallocation est faite simplement en allouant puis libérant la mémoire en début de programme :

free(**malloc**(*TAILLE*)) ;

L'utilisation des arènes avec notre bibliothèque élimine jusqu'à 75% du temps sys-

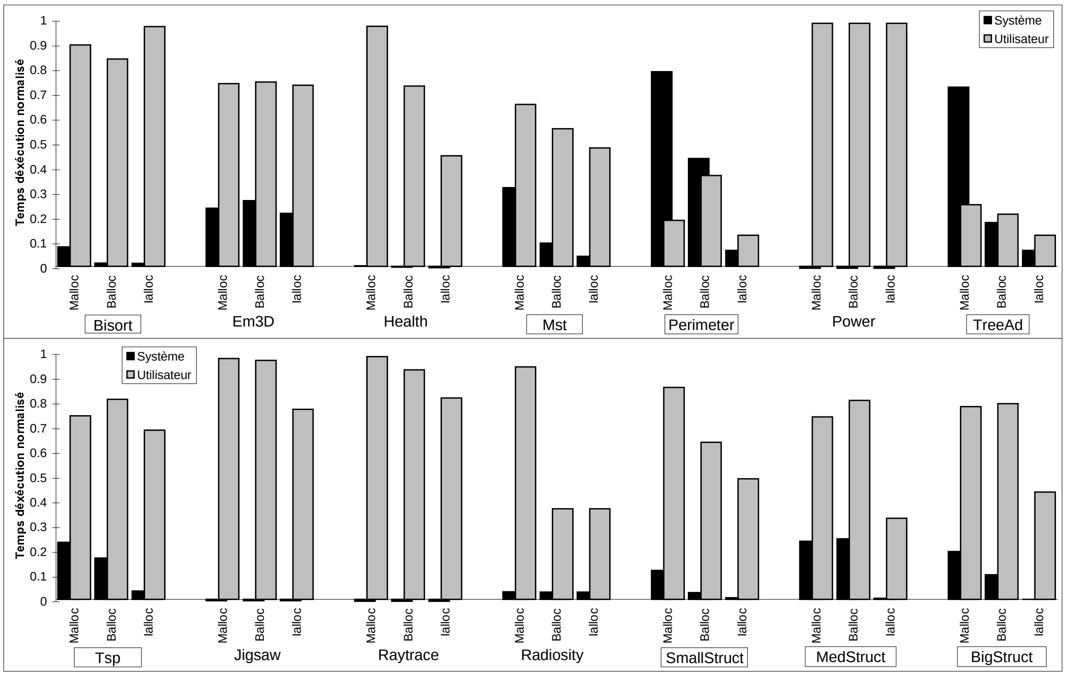


FIG. 5.11 – *Les arènes réduisent le temps système.*

tème lié à l'allocation avec *balloc( )* et 95% avec *ialloc( )*[11]. Cela s'avère important pour certains programmes pour lesquels le surcoût de l'allocation n'est pas négligeable, comme *TreeAdd* ou *Perimeter*.

5.5.2.4 Exploitation de la localité spatiale des données allouées

La localité spatiale des données est améliorée en supprimant les étiquettes de blocs et en évitant toute fragmentation interne. La figure 5.13 montre qu'avec les arènes la localité spatiale est augmentée. Cela s'obtient en supprimant du cache les zones non corrélées par le biais de l'entrelacement.

La figure 5.14 montre le comportement de la hiérarchie mémoire pour les applications testées. Les temps de la figure ont été estimés en utilisant les latences moyennes utilisées dans *perfex*, divisées par 195MHz, la fréquence du processeur.

Le but est de savoir si en éliminant les étiquettes, la bibliothèque *ialloc* améliore la localité spatiale des programmes. Pour cela les versions des programmes optimisées avec *balloc( )* sont étudiées.

L'effet est très variable.

Dans certains cas, la hiérarchie mémoire fait une contre performance. On peut attribuer cet effet à la mauvaise localité de la bibliothèque *ialloc* lorsqu'elle alloue de la mémoire sans avoir recours à la réallocation. On retrouve en effet dans cette catégorie

11. En entrelaçant les structures la taille des arènes augmente, ce qui réduit le nombre d'appels système nécessaires.

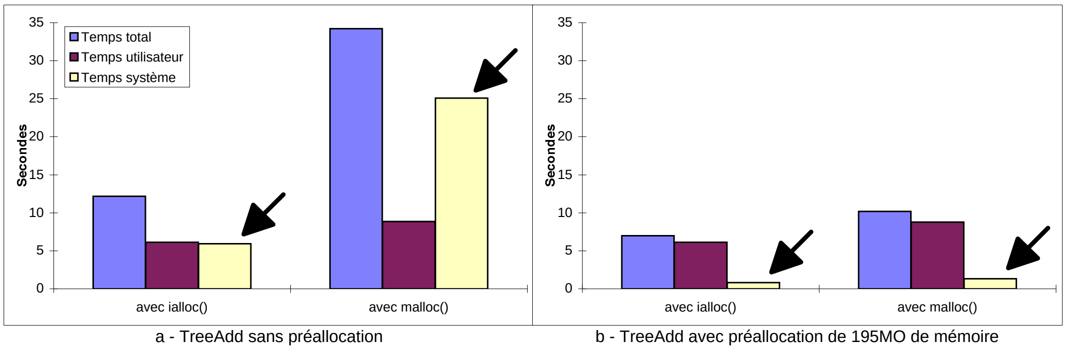


FIG. 5.12 – *Impact de la bibliothèque d'allocation sur le temps système pour TreeAdd.*

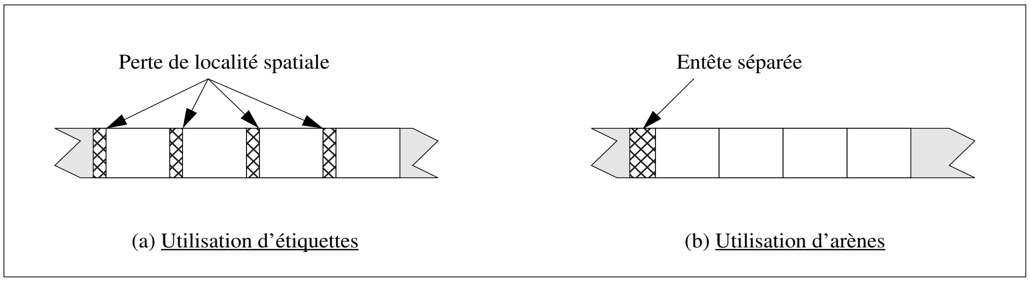


FIG. 5.13 – *Les arènes améliorent la localité spatiale.*

Perimeter et TreeAdd qui ont respectivement une mauvaise performance de la TLB et du cache L2. Ces applications passent la majorité de leur temps d'exécution dans la bibliothèque d'allocation. Les références du programme aux données allouées sont donc masquées par les accès mémoire de la bibliothèque d'allocation qui sont majoritaires.

SmallStruct est un benchmark développé pour montrer que le gain de localité spatiale obtenu en éliminant les étiquettes est tout à fait significatif. En effet, les structures allouées par *SmallStruct* sont de petites tailles, et éliminer les étiquettes augmente sensiblement la localité spatiale entre instances contiguës. Plusieurs applications sont dans ce cas et ont de meilleures performances sans les étiquettes : Bisort et Mst allouent des structures de 12 octets, Health alloue principalement des structures de 12 et 16 octets, Raytrace des structures entre 16 et 32 octets et Radiosity alloue des structures de 24 et 32 octets.

MedStruct, comme prévu, ne bénéficie pas de l'allocation avec *balloc()*. Les instances allouées sont plus grosses, et occupent toute une ligne de cache. Il n'y a donc pas d'augmentation significative de la localité spatiale en éliminant les étiquettes. Jigsaw est dans ce cas là. La principale structure allouée a une taille de 44 octets, bien plus que la taille d'une ligne de cache.

Lorsque le programme ne passe pas beaucoup de temps dans la bibliothèque d'allocation, le fait de ne pas ajouter d'étiquettes augmente sensiblement la localité spatiale du programme, à condition que les structures allouées soient plus petites que la taille des lignes de cache, et qu'elles soient référencées très fréquemment.

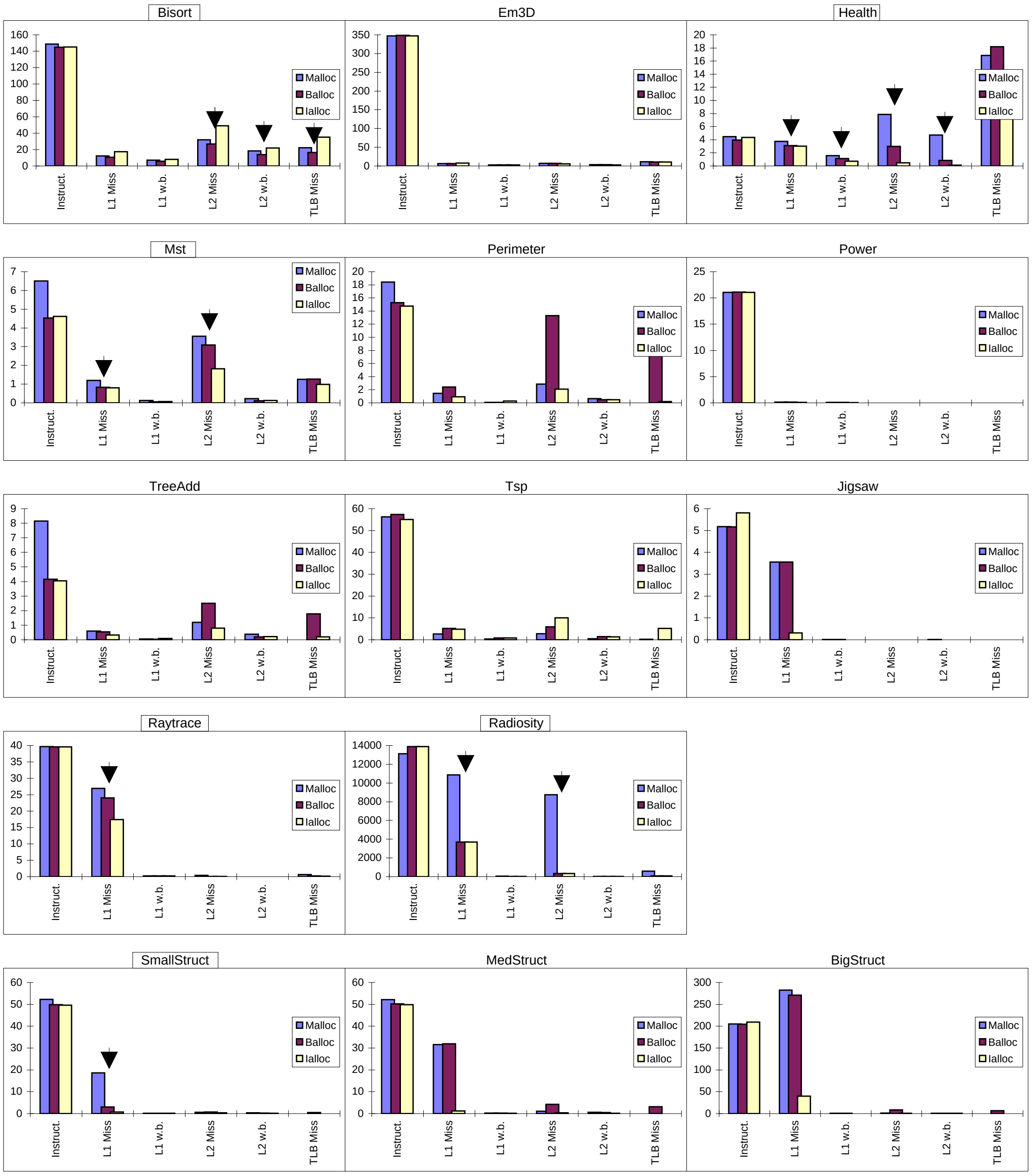


FIG. 5.14 – *Temps utilisateur imputable aux différents niveaux de la hiérarchie mémoire.*

5.5.2.5 Réduction de la consommation mémoire

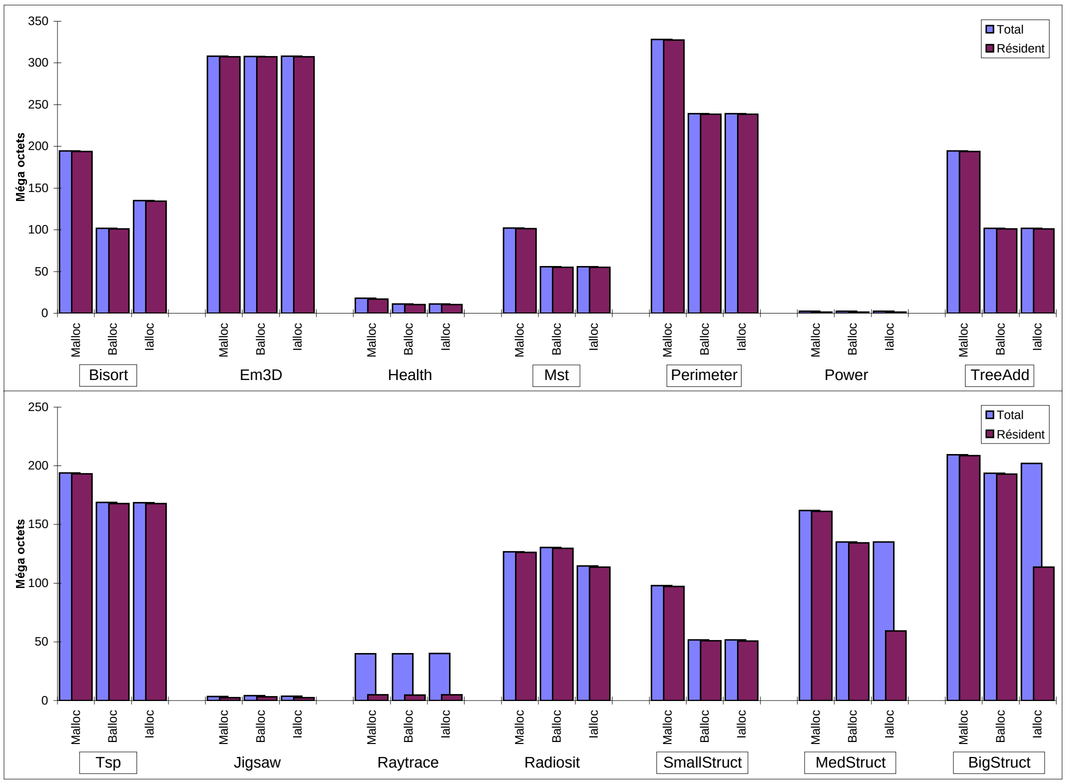


FIG. 5.15 – *La bibliothèque d'allocation ialloc réduit la consommation mémoire.*

Un bloc alloué avec une arène est géré avec quelques bits, alors qu'il faut 8 à 12 octets lorsqu'on utilise des étiquettes, sans compter les pertes induites par la fragmentation interne lorsque la bibliothèque utilisée y a recours.

Le résultat est une économie mémoire pour les applications qui allouent de petites structures de données, qui peut aller jusqu'à 50% comme l'avait remarqué Grunwald et al. [GZH93]. La figure 5.15 montre qu'il est possible de gagner de 30% à 50% de mémoire avec la bibliothèque *ialloc*.

Dans certains cas cependant, on note une dégradation de 2% à 20%. Cette dégradation peut être due au gaspillage mémoire engendré par l'alignement des arènes en mémoire, par la taille de l'entête (elle fait 1088 octets dans nos expériences) ou par l'entrelacement d'une structure avec des groupes de blocs de tailles différentes.

Aucun gaspillage n'est fait pour entrelacer la principale structure allouée dans *Jigsaw*. La structure allouée dans Jigsaw fait 44 octets. Le gaspillage mémoire engendré par la bibliothèque d'allocation standard est de 12 octets, car il faut une étiquette de 8 octets et un gaspillage de 4 octets pour aligner la structure sur une frontière de double mot. Avec l'entête de l'arène de notre bibliothèque, le gaspillage n'est que de 3,45% seulement, soit 1,52 octets par instances, un gain de 19% par rapport à la bibliothèque standard. Les pertes viennent donc de l'alignement des arènes, ce qui donne à notre bibliothèque le même comportement mémoire qu'avec la bibliothèque standard.

Pour *Radiosity*, on gaspille 2 à 3 octets (l'équivalent d'une étiquette) pour entrelacer des structures de 12 à 40 octets de large, ce qui limite le gain mémoire à 10%. Avec *balloc( )* les pertes ne peuvent venir que de l'alignement mémoire.

Pour *Raytrace*, on gaspille pour une des structures 16 octets sur 24 pour l'entrelacer, ce qui explique la perte obtenue avec *ialloc( )*.

Lorsque les structures allouées sont de petites tailles, et que de la mémoire n'est pas gaspillée lors de l'entrelacement des instances, notre bibliothèque permet d'économiser jusqu'à 50% de mémoire.

5.5.3 Influence des techniques de placement mémoire

L'effet des techniques de placement mémoire est étudié sur un sous-ensemble des applications décrites section 5.5.1, pour lesquelles le temps passé dans la bibliothèque d'allocation à allouer la mémoire reste faible. Parmi celles choisies, TSP est l'application passant le plus de temps à allouer de la mémoire.

5.5.3.1 Optimisations par profiling

La technique d'optimisation est basée sur le profiling. Pour faire du *profiling*, les outils logiciels d'optimisation de programme *SpeedShop* ont été utilisés. Ils sont disponibles sur les machines *SGI* à base de microprocesseur R10000. Deux outils ont plus particulièrement été utilisés.

Perfex exploite les deux compteurs d'événements matériels implantés dans le microprocesseur R10000 pour obtenir le décompte d'un grand nombre d'événements dans le CPU, comme les défauts des différents niveaux de cache. Il permet de voir très facilement si la mémoire est un goulet d'étranglement pour le programme étudié.

Ssrun exploite les deux compteurs d'événements matériels en instrumentant le code avec *pixie* [Smi91]. Il peut calculer entres autres le taux d'échecs du cache généré par les références mémoires de chaque ligne source du programme.

Des outils équivalents existent aussi sur d'autres plates-formes, car la plupart des processeurs RISC modernes ont maintenant des compteurs matériels implantés [CB96, ABL97].

Méthodologie Le processus d'optimisation se décompose en trois étapes. Tout d'abord, il faut s'assurer que les accès mémoires sont bien un goulet d'étranglement pour le programme. Si c'est le cas, il faut rechercher les lignes de code responsables des défauts de cache. A partir de là, le placement des champs des structures est déterminé en tenant compte de l'organisation du programme dans les zones critiques trouvées.

- "perfex -y -a *Programme*"
  Perfex donne des statistiques globales sur le programme. Si le programme perd une grande partie de son temps d'exécution dans la hiérarchie mémoire (caches

de données L1, L2 ou TLB) alors le placement des données peut s'avérer utile pour accélérer le programme.

– "ssrun -dc_hwc *Programme* ; prof -h *Programme*.dc_hwc.*nnn*"
Ssrun repère les lignes du programme responsable de la majorité des défauts de cache. On obtient la liste des lignes sources du programme causant le plus de défauts de cache. Comme nos optimisations ne portent que sur les structures de données, il faut que ces lignes là référencent des champs d'instances de structures allouées dynamiquement.
– Analyse et modification du programme
Afin de choisir comment placer et entrelacer les champs, on recherche dans la routine concernée les autres références aux structures incriminées susceptibles d'être exécutées en même temps qu'une des lignes causant les défauts de cache. Ces accès là ne causent pas forcément de défauts de cache si le champ est préchargé par les défauts de cache de la ligne de code incriminée. On obtient ainsi une liste de champs fréquemment utilisés et fortement corrélés pour chaque structure. Les champs sont réorganisés et la structure est entrelacée, de manière à regrouper en mémoire ces champs. Les appels à la bibliothèque d'allocation standard sont remplacés par des appels à la bibliothèque *ialloc* pour ces structures.
– Vérification de l'optimisation
Une fois les optimisations de code faites, le programme est recompilé et exécuté afin de voir s'il est plus performant. Si les gains ne sont pas suffisants, on recommence un cycle d'optimisation.

Cette méthode rapide d'optimisation manuelle est possible grâce aux outils permettant d'utiliser les compteurs d'événements matériels du microprocesseur. Au lieu d'analyser les motifs d'accès de tout le programme, il est possible de focaliser les optimisations sur les routines générant le plus de défauts de cache, ce qui limite le champ d'investigation.

Cette méthode d'optimisation peut s'avérer longue pour certains programmes. En général, cela vient de la difficulté à compiler le code, à générer des jeux de données ou à optimiser le code [12], et non du temps passé à faire le profil ou à choisir le placement des champs en mémoire. L'optimisation par profil peut prendre d'un quart d'heure pour les cas les plus favorables à plusieurs jours pour les cas les plus complexes à traiter.

Exemple Ft est un petit programme de deux mille lignes de code écrit en C. Il a été compilé sur une SGI Onyx avec toutes les options de compilation activées ("cc -oFAST=IP25 -LNO -IPA"). Il est exécuté avec la ligne de commande "ft 20000 30000".

Perfex indique que 9,6 milliards de cycles sont nécessaires à son exécution, soit 49 secondes de temps utilisateur, alors que 2,2 milliards d'instructions sont exécutées. Il génère 300 millions de défauts de cache L1, et 25 millions de défauts de cache L2. Il est donc possible d'envisager une optimisation du placement des données.

12. Les difficultés majeures sont les codes complexes, utilisant des tableaux de structure ou les codes gérant l'allocation à travers des routines d'allocation mémoire spécifiques.

Ssrun indique que 76% des défauts de cache L1 apparaissent à la ligne 169 du fichier *graph.c*, dans la routine *PickVertex*. Le code à cet endroit est le suivant :

```
Vertices * PickVertex(Vertices * graph, int whichVertex) {
  int i;

  for(i = 0; i < whichVertex; i++) {
    graph = NEXT_VERTEX(graph);
  }
  return(graph);
}
```

En lisant *graph.h* il est possible d'obtenir les informations comlémentaires:

```
#define NEXT_VERTEX(V) ((*(V)).next)

typedef struct _Vertices {
  int id;
  Edges * edges;
  struct _Vertices * next;
  int key;
  Edges * chosenEdge;
} Vertices;
```

Pour optimiser la localité du programme, il faut que la boucle de code ait la meilleure localité spatiale possible. Ici, la boucle n'accède qu'au champ *next*. On l'entrelace donc tout seul en modifiant la structure de données comme suit, sachant qu'un pointeur occupe un mot de quatre octets:

```
typedef struct _Vertices
{
  struct _Vertices * next;
  char Pad5[_IA_INTERLEAVE - 4];
  int id;
  char Pad4[_IA_INTERLEAVE - 4];
  Edges * edges;
  char Pad3[_IA_INTERLEAVE - 4];
  int key;
  char Pad2[_IA_INTERLEAVE - 4];
  Edges * chosenEdge;
  char Pad1[_IA_INTERLEAVE - 4];
} Vertices;
```

Bien sûr, il faut changer les appels à *malloc* en appels à *ialloc*, donc

```
  vertex = (Vertices *)malloc(sizeof(Vertices));
```

est converti en:

vertex = ialloc(Vertices,4,0);

Avec Perfex, on voit que le programme ne prend plus que 1,6 milliards de cycles à s'exécuter pour un peu plus de deux milliards d'instructions, soit 8,5 secondes de temps utilisateur. Il ne cause maintenant plus que 30 millions de défauts sur L1 et 171 milles sur L2, une réduction d'un facteur 10. Les résultats indiquent donc que l'optimisation faite est efficace. Elle apporte un speedup de 5,6.

5.5.3.2 Apport de la localité spatiale

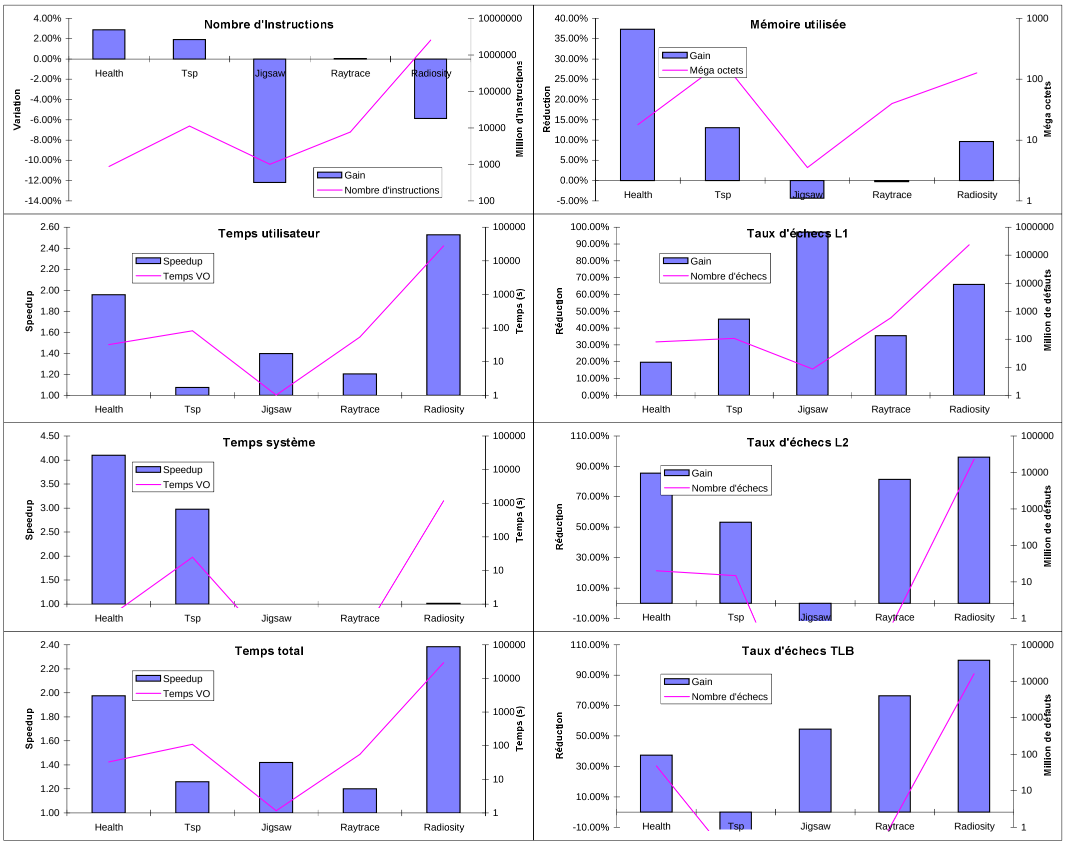


FIG. 5.16 – *Résultats expérimentaux des optimisations de localité.*

Seuls les programmes qui obtiennent leur speedup grâce à l'amélioration de la localité des données allouées sont montrés ici. Ni le temps d'allocation[13], ni le temps passé dans le système n'influent, sauf pour TSP qui voit son temps système passer de

13. La variation du nombre d'instructions reste non significative

25 secondes à huit secondes (à titre de comparaison, son temps utilisateur est de 100 secondes).

Tous les benchmarks, figure 5.16, réduisent le taux de défauts du cache primaire de 20% à 95%, du cache secondaire de 50% à 90% (sauf *Jigsaw* qui a un taux de défauts très faible au départ), et de la TLB de 35% à 90% (sauf *TSP* qui a déjà un taux de défauts de TLB très faible au départ).

Le résultat des optimisations de la localité spatiale des données est un gain non négligeable sur le temps utilisateur avec des speedup de 1,07 à 2,5, figure 5.16.

5.5.3.3 Indépendance du jeu de données en entrée

Les résultats précédents ont été obtenus avec le même jeu de données que celui qui est utilisé pour optimiser le programme. Si le jeu de données utilisé pour le *profil* permet d'exploiter toutes les fonctionnalités du programme, les optimisations devraient se répercuter sur l'exécution du programme, quel que soit le jeu de donné utilisé.

Les optimisations de la localité utilisées modifient le placement des données en fonction de leur utilisation. Lorsque les données sont organisées en structures récursives, indépendamment du jeu de données il est probable que le programme passe une grande partie de son temps à rechercher des données dans la liste ou l'arbre. Comme la recherche est faite le plus souvent par une petite partie du code, les propriétés globales de la recherche resteront les mêmes quel que soit le jeu de données utilisé. Celui-ci n'influe alors que sur le nombre de données manipulées.

Pour vérifier cela, les programmes ont été testés avec différents jeux de données. Pour les programmes *Olden* (*Health* et *Tsp*), il s'agit de modifier les paramètres en lignes. Pour *Puzzle* trois tailles de puzzle ont été utilisées (les puzzles sont générés aléatoirement par un petit programme). Pour *Radiosity*, en plus de la scène *floor* contenant 256 pièces légèrement meublées et reliées par un couloir, une scène *class* contenant 4 classes d'école meublées de chaises et de tables entourant un patio a été utilisée. Cette scène est très différente de la première. Il y a peu de pièces, et celles-ci contiennent un grand nombre d'objets (chaises, tables, lampes...). Le programme a aussi été exécuté en deux et quatre itérations. Pour *Raytrace*, une nouvelle scène a été conçue, représentant en 2 plans parallèles constitués de 100x100 sphères (20003 sphères au total). La scène d'origine représente une forme fractale composée de 7381 sphères de tailles variables et d'un plan servant de fond.

Les résultats figure 5.17 montrent que des gains importants sont obtenus avec des jeux de données différents.

5.5.3.4 Préchargement et placement des structures de données

Le préchargement des structures de données dites récursives a été étudié comme une technique pour compenser la latence des accès mémoire.

Méthode

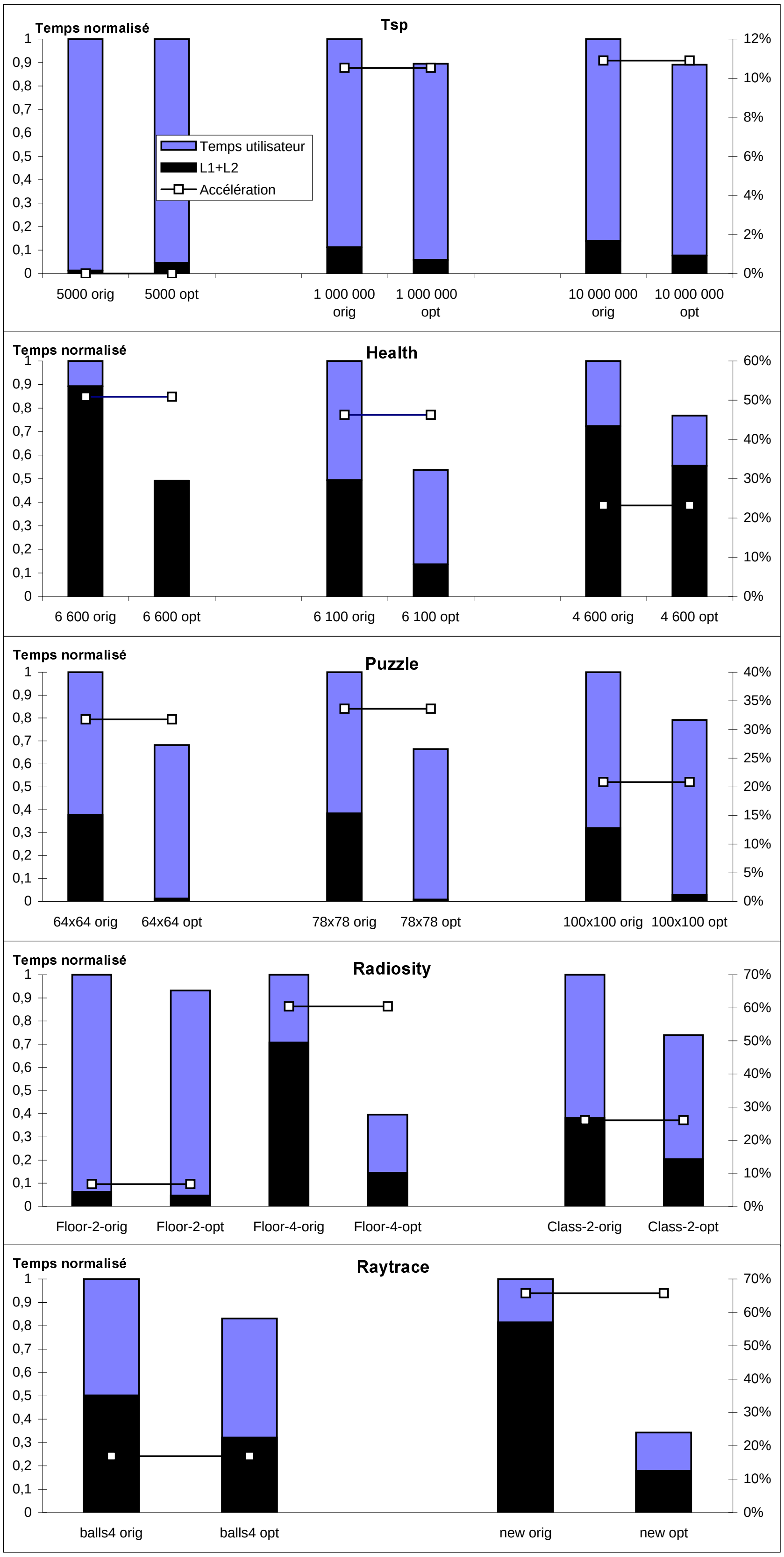


FIG. 5.17 – *Influence du jeu de données sur les programmes.*

Lorsqu'une instance d'une structure est accédée, on peut précharger les pointeurs qu'elle contient afin d'anticiper le chargement des prochaines instances [LSKR95]. Malheureusement, les pointeurs ne sont disponibles que peu de temps avant l'utilisation des instances à précharger (problème du *pointer chasing*). Afin d'augmenter la distance de préchargement, ce qui devient nécessaire lorsque la latence mémoire augmente, [LM96] propose de rajouter dans la structure des pointeurs dédiés au préchargement. Ces pointeurs sont mis à jour au fur et à mesure que les instances sont allouées et insérées dans l'arborescence des instances en mémoire.

Une technique plus simple est proposée ici. *ialloc* utilise des arènes pour allouer les instances en mémoire. Après avoir entrelacé les instances en mémoire dans les arènes, la position des instances en mémoire est connue plus précisément. Il est donc possible de faire automatiquement du préchargement logiciel sans avoir recours aux techniques de [LM96, LSKR95], en faisant juste de l'arithmétique de pointeur.

Pour faire le préchargement, le pointeur sur l'instance courante est incrémenté pour pointer à une distance d'une ou deux lignes de cache plus loin. Il est possible de ne lancer qu'un préchargement par ligne de cache touchée pour éviter de démultiplier les préchargements d'un même bloc mémoire. Si une ligne de cache contient N instances, il suffit de lancer un préchargement toutes les N itérations. Le préchargement de la ligne suivante précharge N instances. De plus, avec *ialloc* la position des instances en mémoire est très régulière, il devient plus rare de lancer des préchargements inutilement.

Mise en oeuvre

Le compilateur C MipsPro Compiler de SGI disponible sous Irix 6.3 est utilisé pour l'expérimentation. Il est capable de générer les instructions de préchargement logiciel qui pilotent les caches de données du Mips R10000.

Le compilateur peut générer automatiquement des instructions de préchargement. Malheureusement, pour les applications non scientifiques, on observe parfois une dégradation des performances lorsque le compilateur génère les préchargements selon ses propres critères. Il faut donc désactiver la génération automatique d'instructions de préchargement avant toute optimisation. La ligne de commande utilisée pour la compilation est donc:

"*cc -Ofast=IP25 -LNO -IPA -LNO:prefetch=0 -LNO:prefetch_manual=ON*
-CLIST:=TRUE, emit_pfetch=TRUE"

Les profils obtenus avec *ssrun* pour entrelacer les instances de structures indiquent quelles parties du code causent des défauts de cache. C'est dans ces parties du code que les instructions de préchargement doivent être insérées. Les instructions de préchargement sont insérées dans le code source à l'aide de *pragma*. Il faut préciser dans le *pragma* l'adresse à charger, éventuellement à l'aide de pointeurs ou d'indices. Le *pragma* est conçu pour faire du préchargement de matrice, il faut utiliser une notation matricielle. La ligne suivante est utilisée pour *ft*:
"*#pragma prefetch_ref=((***char***∗)&(graph→next))[32], stride=8, kind=rd*" Cela veut dire qu'il faut utiliser comme adresse de référence l'adresse du champ *next* de l'instance

courante, et ajouter 32 (la taille d'une ligne de cache). Le préchargement est requis au moins une fois toutes les huit itérations de boucle (cela suppose que la structure est entrelacée, et que la taille de l''entrelacement est de quatre octets).

Lorsque le compilateur fait du pipeline logiciel sur une boucle, il peut générer trop d'instructions de préchargement pour un *pragma* et dégrader la performance de la mémoire. Il faut alors générer le code en assembleur pour les fichiers source concernés par le préchargement et éliminer manuellement tous les préchargements qui ne sont pas dans la partie répétitive du pipeline.

Résultats

Le préchargement logiciel est testé avec *malloc* pour le placement original et avec *ialloc* pour les placement optimisés (entrelacement des instances de structures).

TAB. 5.4 – *Efficacité du préchargement.*

Programme	malloc	malloc + prefetch	ialloc	ialloc + prefetch
ft 20000 30000	47,7s	49,2s	7,61s	6,45s
health 6 500	49,9s	37,4s	20,0s	22,5s
raytrace balls.env	84,0s	83,4s	52,8s	53,33s
SmallStruct 512 512 0	138s	137s	57,7s	42,2s
SmallStruct 512 512 1	29,1s	29,9s	21,8s	21,6s

Le tableau 5.4 montre les résultats obtenus avec quelques programmes. Les résultats obtenus avec *ialloc* correspondent aux programmes dont les structures sont entrelacées.

Ft : Les préchargements sont insérés dans la boucle d'accès aux instances. Cette boucle est pipelinée par le compilateur. Il a donc fallu supprimer les préchargements superflus. On note que le préchargement est innéfficace avec le placement original, mais apporte un gain supplémentaire de vitesse de 15% avec le placement optimisé.

Health : des *pragmas* de préchargement des patients et des listes de patients sont insérées dans une routine. Cela améliore de 25% la performance de la version non optimisée du programme, mais dégrade la performance de la version optimisée.

Raytrace : Il y a trois structures distinctes à précharger, les "*sphere*", les "*object*" et les "*ray*". Après le test de plusieurs combinaisons infructueuses, il semble que le préchargement n'apporte pas de gain de performance pour cette application.

SmallStruct : Un préchargement est inséré dans la boucle de recherche. On note que le préchargement apporte un gain de 27% supplémentaire pour la version entrelacée lorsque les recherches démarrent à partir du dernier élément trouvé dans la liste (SmallStruct 512 512 0), mais n'apporte quasiment aucun gain lorsque la recherche redémarre en début de liste (SmallStruct 512 512 1). Cela s'explique car il y a plus de chances que les données soient encore dans le cache.

Le préchargement des données dans le cache permet d'anticiper d'éventuel défauts de cache. Cependant, si les données préchargées ne sont pas utilisées ou génèrent des interférences dans le cache, la performance peut se dégrader. Les expériences montrent qu'il est parfois possible de faire du préchargement simplement en chargeant des blocs distants de quelques lignes de cache en mémoire. De plus il semble possible de combiner l'entrelacement des structures, l'utilisation de *ialloc* et le préchargement pour minimiser la latence mémoire.

5.5.3.5 Impact de l'entrelacement sur des programmes en C++ utilisant la bibliothèque STL

L'étude porte principalement sur des programmes écrits en C. Cette section tente d'évaluer l'impact que peut avoir l'entrelacement sur les performances de programmes écrits avec des langages objets offrant des fonctionnalités de programmation plus puissantes. Jusqu'à quel point est-il possible de cacher au programmeur les techniques de gestion mémoire (notamment l'entrelacement des structures), et quelles performances peut-on espérer en fonction du niveau d'implication du programmeur dans les techniques de gestion mémoire. Pour étudier cela un petit programme écrit en C++ utilisant la bibliothèque STL est étudié avec différentes techniques d'optimisation. Les résultats indiquent qu'avec la bibliothèque STL, d'autres problèmes occultent l'influence de la hiérarchie mémoire sur la performance du programme.

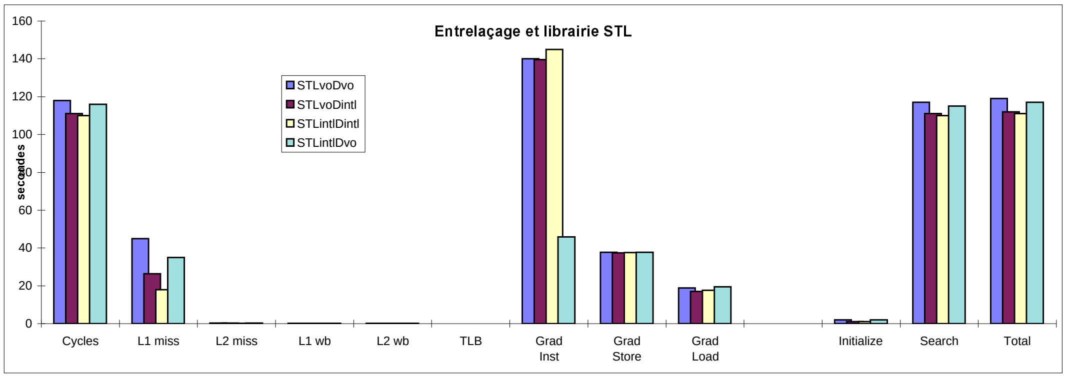


FIG. 5.18 – *L'entrelacement peut augmenter la performance des programmes écrits en STL.*

En modifiant juste la bibliothèque STL, il est possible d'exploiter d'une manière totalement transparente la bibliothèque d'allocation *ialloc* et les techniques d'entrelacement associées. La bibliothèque STL utilise alors des containers entrelacés en mémoire, ce qui permet d'améliorer la localité spatiale des références lorsque l'itérateur parcourt les containers. Avec une telle approche, les containers sont entrelacés, mais les objets rangés dans les containers par le programmeur ne le sont pas.

Lorsqu'un itérateur recherche un objet, il doit référencer à la fois des champs du container (les pointeurs), et des champs de l'objet (le critère de recherche). Pour optimiser complètement la localité spatiale du programme, il faut donc que le programmeur entrelace lui aussi les objets qu'il range dans les containers. Dans ce cas l'optimisation de la localité n'est plus transparente pour lui.

Pour l'expérience, le programme *SmallStruct* est réécrit en C++ avec la bibliothèque STL. La liste chaînée est gérée entièrement par le container STL et la recherche par l'itérateur par défaut. *SmallStruct* exécute un million de recherches sur une liste chaînée d'un million d'éléments.

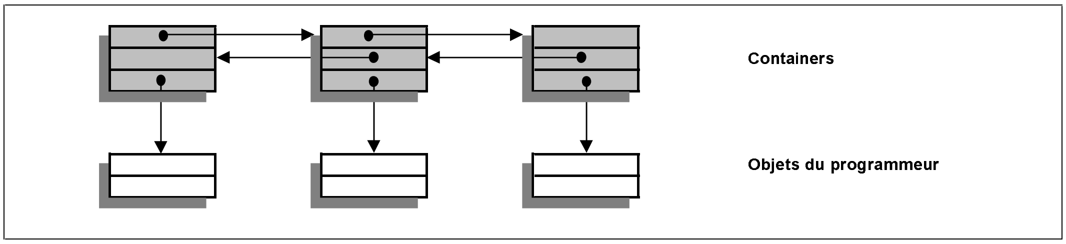


FIG. 5.19 – *Le container STL contient les pointeurs de la liste.*

Chaque noeud de la liste devient donc implicitement composé de deux objets comme le montre la figure 5.19. Le premier, gérée par la bibliothèque STL est le container. Il contient les pointeurs permettant de parcourir la liste de containers et d'accéder aux objets de l'utilisateur. Le second, défini par l'application, contient les données utiles.

Il est possible d'entrelacer séparément le container et l'objet déclaré par le programmeur. Au total, cela génère quatre possibilités d'optimisation, que nous avons testées. Cela correspond à une optimisation de la bibliothèque STL seule, à l'utilisation d'une bibliothèque STL standard par un programmeur qui optimise le placement de ses objets, et à la combinaison d'une bibliothèque optimisée et de l'optimisation des objets du programmeur.

La figure 5.18 montre la performance des quatre programmes en fonction du mode d'optimisation choisi. A gauche de la figure, le nombre de cycles estimé par *perfex* imputable à chaque événement est affiché. Le coût en temps d'un événement (échec sur L1, L2 ou la TLB) est estimé en fonction du nombre de fois où l'événement apparaît, et du coût moyen observé de cet événement sur une Origin 2000 avec des processeurs R10000. Les trois graphiques à droite dans la figure donnent les temps réellement passés dans chacune des trois étapes du programme.

Les résultats indiquent que l'utilisation seule d'une bibliothèque STL entrelacée n'apporte pas beaucoup de gains (1,7% d'accélération). Il n'est donc pas intéressant d'utiliser une bibliothèque STL entrelacée si le programmeur n'entrelace pas lui aussi ses objets.

En entrelaçant les données, le gain de performance passe à 5,9%, ce qui reste modeste, d'autant plus que la configuration des données de *SmallStruct* devrait montrer des gains importants lorsque la bibliothèque *ialloc* est utilisée.

Même si l'entrelacement permet d'améliorer de façon significative la performance de la hiérarchie mémoire (jusqu'à 60% de réduction des défauts de cache), le surcoût en nombre d'instructions qu'implique l'utilisation du C++ et de la bibliothèque STL à une plus grosse influence sur le temps d'exécution que la performance de la hiérarchie mémoire. A titre de comparaison, la version C de *SmallStruct* ne prend que 37,5 secondes, contre 118 secondes pour la version écrite en C++ avec STL. Lorsque la mémoire n'est pas le goulot d'étranglement, l'utilisation de techniques de placement de données en mémoire n'est pas cruciale.

Avec les programmes écrits en C++, l'entrelacement des objets est possible et réduit de façon significative les taux d'échecs des caches. Cependant, ces programmes sont surtout limités par la performance du processeur. Une telle optimisation n'est utile pour les programmes écrits en C++ que si le goulet d'étranglement est effectivement la mémoire.

5.6 Conclusion

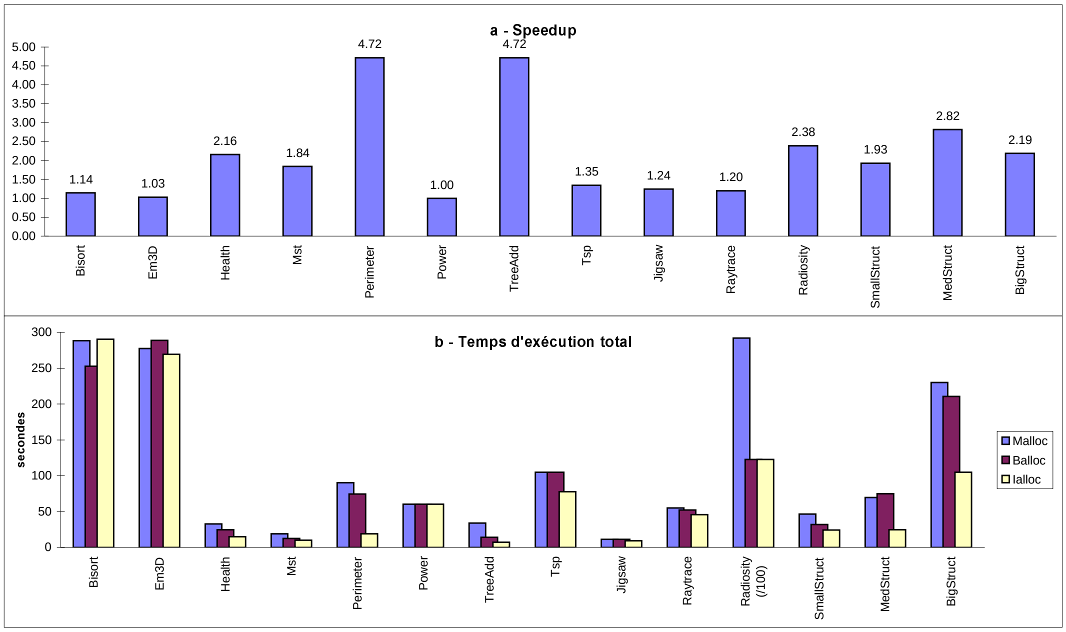


FIG. 5.20 – *Accélération du temps total d'exécution avec nos optimisations.*

Une bibliothèque d'allocation performante est proposée pour gérer les techniques de placement de structures de données qui ont été mises au point.

Cette bibliothèque est performante. Elle permet de gérer la mémoire en exécutant un petit nombre d'instructions et en limitant le nombre des appels système. De plus, elle n'entrave pas la localité qu'il peut y avoir entre données allouées de manière contiguë en mémoire, grâce à l'utilisation d'arènes pour gérer la mémoire au lieu d'étiquettes.

Grâce au support de cette bibliothèque, il a été possible de mettre en place des techniques de placement mémoire des structures de données allouées dynamiquement. La réorganisation des champs d'une structure permet d'améliorer la localité spatiale au sein de chaque instance allouée en regroupant ensemble en mémoire les champs les plus utilisés. L'entrelacement des instances allouées permet d'exploiter la localité spatiale qu'il peut y avoir entre les champs de différentes instances d'une même structure. Cette optimisation s'avère utile pour les programmes utilisant des structures de données récursives (ou structures de données s'auto-référençant), comme les programmes gérant leurs données avec des listes chaînées ou avec des arbres.

La figure 5.20 récapitule les accélérations obtenues avec nos optimisations et le temps total d'exécution[14]. La combinaison de ces optimisations permet d'obtenir des gains de performance considérable pour une grande partie des programmes utilisant des structures de données et qui sont limités par la performance de la hiérarchie mémoire. De tels gains sont d'autant plus importants qu'aucune autre technique logicielle n'est disponible pour améliorer la localité des données allouées dynamiquement.

Telle qu'elle est conçue, la bibliothèque *ialloc* peut être incluse dans une boîte à outils pour optimiser la performance des programmes. Il devient ainsi possible pour les programmeurs d'améliorer la performance de leurs programmes en apportant quelques modifications mineures à leurs programmes.

14. Le temps total d'exécution de *Radiosity* est tellement élevé qu'il est divisé par 100 dans la figure.

Conclusion

Avec l'augmentation exponentielle de la latence mémoire, il devient nécessaire d'exploiter au mieux les mémoires cache.

Pour cela, il est proposé d'augmenter la localité spatiale et temporelle des programmes, en modifiant le placement mémoire des données.

La classification des défauts de cache proposée par Sugumar [SA93] a été complétée afin de prendre en compte l'impact de la localité spatiale sur le taux d'échecs des caches. La classification proposée décompose les défauts de capacité en défauts fondamentaux et en défauts de distribution : les défauts de distribution peuvent être éliminés si l'on redistribue les données en mémoire de façon à améliorer la localité spatiale. La classification complète des défauts de cache a été simulée pour plusieurs applications. Les défauts de capacité sont le plus souvent composés d'une majorité de défauts de distribution. Les techniques de placement mémoire ont donc un fort potentiel d'optimisation car elles peuvent éliminer à la fois les défauts de conflit et les défauts de distribution.

Pour vérifier l'impact du placement mémoire sur la performance des caches, une heuristique de placement mémoire pour des scalaires indépendants a été mise au point. Pour cela une méthode de mesure de la localité des références a été établie. Cet algorithme a été expérimenté pour redistribuer les adresses des références mémoires sur des traces. Les résultats indiquent qu'il est effectivement possible d'exploiter le potentiel qu'offre le placement mémoire de données. Il est même possible d'obtenir avec un cache à correspondance directe des taux d'échecs plus bas que le taux de défauts de capacité obtenus avec le placement mémoire original. Il est donc parfois possible d'obtenir un taux d'échecs plus faible avec un meilleur placement des données qu'avec une meilleure architecture de cache.

Le placement de données en mémoire doit être fait en tenant compte des contraintes de placement imposées par le langage de programmation et le compilateur. Ceci nécessite de développer des solutions de placement pour chaque type de données, surtout celles susceptibles de causer un nombre important de défauts de cache.

L'étude s'est donc orientée vers l'optimisation logicielle du placement mémoire des structures de données allouées dynamiquement. Ce type de donnée est fréquemment utilisé dans les programmes généralistes, mais ne bénéficie pas encore de techniques logicielles d'optimisation de la localité. Deux techniques de placement mémoire sont donc proposées. La redistribution des champs d'une structure et l'entrelacement des instances d'une structure. Ces techniques permettent d'améliorer la localité des instances allouées et génèrent des *speedups* pour les applications limitées par la performance de la hiérarchie mémoire.

Une bibliothèque d'allocation optimisée a été développée pour faciliter l'entrelacement des champs d'instances de structures de données. Une telle bibliothèque pourrait être un complément important d'une boîte à outil d'optimisation logicielle. Elle peut être utilisée par des programmeurs pour optimiser les structures d'un programme en modifiant très peu le code du programme.

Les techniques de padding de matrices sont exposées en appendice. Le placement mémoire est limité pour les matrices, car les accès aux données se font avec des indexes. Le placement en mémoire des éléments des matrices ne peut être modifié que si cela ne modifie pas les calculs d'adresses que doit faire le compilateur en utilisant les indexes de matrice. Les études publiées montrent que le padding est un bon complément des techniques de transformations de boucles qui sont plus généralement employées.

Le placement de données en mémoire offre un potentiel de gain très important. En améliorant la localité spatiale et temporelle, il est possible de mieux exploiter les caches et d'apporter une accélération aux programmes limités par la latence de la hiérarchie mémoire. Certaines des solutions proposées peuvent être utilisées statiquement à la compilation mais il reste encore à les automatiser et à les insérer dans des compilateurs.

Annexe A

Les matrices : padding et transformations de boucles

A.1 Introduction

Les programmes de calcul numérique traditionnellement réservés aux machines vectorielles sont de plus en plus souvent portés sur des machines ayant une hiérarchie mémoire plus économique à base de mémoires cache. Pour exploiter pleinement sur ces machines la puissance de calcul des processeurs, il devient critique d'adapter les applications scientifiques afin d'avoir une plus grande localité spatiale et temporelle.

Les calculs numériques se font en manipulant de nombreuses matrices, généralement plus grandes que la taille des caches installés sur les machines. Afin d'exploiter au mieux la capacité de stockage des caches, de nombreuses techniques logicielles ont été mises au point pour améliorer la localité des accès mémoires. Deux approches sont possibles, les transformations de code et le placement de données.

Les techniques de transformations de boucles sont les plus utilisées. Elles permettent d'améliorer la localité spatiale et temporelle des programmes en réordonnançant les accès mémoires.

Les études sur le placement des données en mémoire sont plus rares. Cette approche, orthogonale aux techniques de transformations de boucles, est plus limitée dans son champ d'application. L'adresse d'un élément d'une matrice est calculée à l'aide des indexes de cet élément. Dès lors les techniques de placement ne peuvent déplacer les éléments en mémoire que si les calculs d'adresses mémoires restent possibles à partir des indexes.

Nous décrivons ici les techniques de placement de données applicables aux matrices, les techniques de *padding*. Bien que limitées, elles peuvent quand même compléter utilement les transformations de boucles, car elles s'appliquent même quand les transformations de boucles ne peuvent pas être utilisées.

La classification des échecs faite pour l'étude des matrices est expliquée section A.1.1 et les transformations de boucles sont brièvement introduites section A.1.2. Les techniques de placement mémoire sont ensuite décrites section A.1.3. Les calculs permettant de faire du padding sont présentés section A.2, et des solutions sont énnoncées pour des cas simples d'interférences, avant de conclure section A.3.

A.1.1 Self et cross interférences

Les optimisations logicielles permettent d'éliminer indifféremment défauts de conflits et de capacité. La classification des 3C étant donc trop imprécise pour évaluer l'impact des transformations de boucles, une autre classification des interférences a été mise au point pour décrire les accès aux matrices.

Cette classification permet de qualifier les différents défauts de caches qui peuvent survenir lorsque des matrices sont référencées dans un nid de boucle.

- Les *self-interférences* correspondent aux défauts de cache générés lorsque des éléments d'une matrice se chassent mutuellement du cache.
- Les *cross-interférences* sont les défauts de cache générés par les accès à deux matrices dont les éléments se chassent du cache.

Il faut noter que ces interférences recouvrent les défauts de conflits et les défauts de distribution sans distinction.

Par exemple, lorsqu'une matrice est accédée avec un pas égal à la taille des lignes de cache, un seul élément est utilisé par ligne. Il ne peut alors pas y avoir plus d'éléments référencés dans le cache que le nombre de lignes. Si une optimisation logicielle ramène le pas des accès mémoires à l'unité, alors les lignes de cache peuvent exploiter la localité spatiale des références du programme. En éliminant les défauts de distribution, on a diminué les self-interférences.

Si maintenant le pas des accès est égal à deux fois la taille des lignes, ce ne sont plus des défauts de distribution mais des défauts de placement qu'il faut éliminer, car les sauts d'adresse sont tels que seulement la moitié des lignes du cache est utilisée. C'est le même type d'optimisation logicielle que précédemment qui pourra éliminer ces défauts. Ils sont donc tout naturellement aussi classés dans la catégorie des self-interférences.

Cette classification nécessite de connaître l'origine d'une référence mémoire pour savoir s'il y a interférence entre une seule ou plusieurs matrices. Les mesures ne se font donc pas avec un simulateur de cache, mais par une analyse mathématique des références générées par le nid de boucle. Pour cela un formalisme basé sur les vecteurs de réutilisation (*reuse vector*) des éléments d'une matrice est utilisé.

A.1.2 Transformations de boucles

Les programmes scientifiques manipulent les matrices par l'intermédiaire de boucles de calculs imbriquées (nids de boucles). Les indices de boucles servent à calculer les indexes pour accéder aux matrices. En modifiant l'ordre d'incrémentation des indices de boucle, on modifie les motifs d'accès aux matrices. Il s'agit d'améliorer la localité en réutilisant le plus possible les éléments de matrice contenus dans une même ligne de cache avant qu'ils ne soient remplacés par d'autres éléments. Cela se fait par exemple en générant des pas d'accès (*stride*) unitaires aux matrices.

Il y a un très grand nombre d'optimisations de nids de boucles possibles (*loop interchange, fusion, blocking, copying,* etc.). Nous ne tenterons pas de les énoncer ici, car elles ne correspondent pas à des techniques de placement de données. Pour une description de ces méthodes, nous vous renvoyons à [BCJ94, PH96, Por89].

Les transformations de boucles s'avèrent efficaces dans le cas de nids de boucles simples. Cependant, elles sont très vites contrariées lorsque le nid de boucle n'est plus parfait. Le cas de boucles dont les bornes des indexes sont variables peut encore être géré, mais lorsque des opérations de la boucle sont soumises à des conditions d'exécution ou lorsqu'il y a des appels de fonctions à l'intérieur de la boucle, l'analyse de dépendance devient impossible et les optimisations ne sont plus applicables.

Un autre problème est le cas d'interférences générées par des accès aux matrices avec des indexes identiques à une constante près (par exemple les accès à deux matrices M_1 et M_2 comme $M_1[i,j]$ et $M_1[i,j+1]$ ou $M_2[i,j]$). Si ces références ne peuvent pas être séparées dans des nids de boucles différents ou par du *blocking*, il devient important dans certains cas de modifier le placement des données en mémoire.

Les transformations de boucles sont les optimisations logicielles des caches qui ont reçu le plus d'attention. Elles permettent d'exécuter très efficacement du code scientifique vectoriel sur des machines ayant une hiérarchie mémoire bas de gamme conçue autour des mémoires caches. Cependant, ces solutions ne résolvent pas tous les problèmes, et peuvent bénéficier du mariage avec le *padding*.

A.1.3 Le placement mémoire des matrices

A.1.3.1 Le padding

Le *padding* (rembourrage)consiste à modifier l'alignement des colonnes des matrices en mémoire, en déclarant des éléments supplémentaires qui servent de rembourrage. Ainsi, des éléments utilisés en même temps dans un nid de boucle subissent un décalage en mémoire et n'interfèrent plus dans le cache.

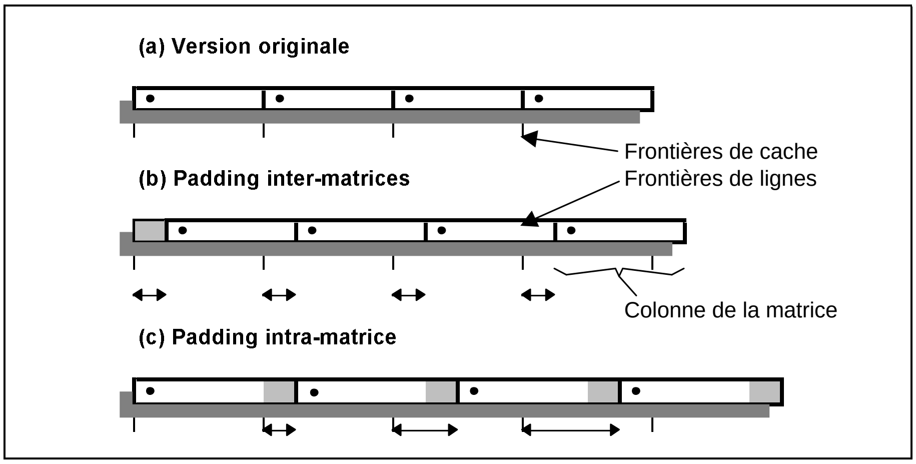


FIG. A.1 – *Les deux types de padding.*

Il y a deux types de *padding*, comme le montre la figure A.1.

- Le *padding* inter-matrice : On intercale des éléments en mémoire entre deux matrices qui interférent dans un nid de boucles. Cela évite les conflits en décalant l'adresse d'une des matrices d'une distance multiple de la taille des lignes de cache.
  Soient deux matrices accédées comme en figure A.1(a), en compétition pour les même lignes de cache. Le padding permet de décaler l'une des deux matrices comme en figure A.1(b), et les deux matrices n'interfèrent plus.
- Le *padding* intra-matrice : On rembourre la dimension principale de la matrice en la déclarant un peu plus grande que nécessaire. Les derniers éléments de chaque colonne sont inutilisés mais servent à décaler en mémoire les colonnes suivantes d'une distance multiple de la taille des lignes du cache. Cela permet d'éliminer les interférences qu'il peut y avoir entre des éléments de plusieurs colonnes d'une même matrice.
  Si les colonnes d'une matrice sont accédées comme dans la figure A.1(a), elles interfèrent. Le *padding* décale les colonnes comme dans la figure A.1(c) pour éliminer les interférences.

A.1.3.2 Les autres techniques de placement

D'autres techniques de placement mémoire sont possibles avec les matrices, mais sont peu étudiées, car elles offrent moins de possibilités d'optimisation. Nous décrivons brièvement l'entrelacement de matrices et le padding intra-ligne.

Pour entrelacer deux matrices, il faut déclarer une matrice d'ordre supérieur, dont chaque colonne contient tous les éléments de même indice des autres matrices. Ainsi, lorsque des éléments de même indice de plusieurs matrices sont référencés, ceux-ci n'interférent pas car ils sont contigus en mémoire. L'inconvénient de cette approche est qu'il faut modifier tout le code pour l'appliquer, car le nom des matrices référencées change. De plus elle ne s'applique que si les matrices utilisées sont de même dimension.

Le padding intra-ligne consiste à régler plus finement le padding intra-matrice afin de distribuer uniformément entre toutes les itérations du nid de boucles les défauts de cache qui apparaissent. En décalant les colonnes en mémoire d'une distance inférieure à la ligne de cache, on change l'itération de la boucle contenant la référence qui causera le défaut de cache. Cette technique permet d'éviter qu'un cache non bloquant reçoive trop de défauts de cache à gérer simultanément [BGS94]. Un cache non bloquant peut gérer simultanément un petit nombre de défauts de cache (souvent quatre) sans interrompre le flot d'instruction.

A.1.3.3 Etat de l'art

Les solutions de placement mémoire des éléments de matrice sont peu nombreuses. Les études tente donc souvent de les utiliser en complément aux techniques de transformation de boucles.

Lorsque les matrices ont des dimensions multiples de puissances de deux ($k \times 2^n$), il peut arriver des phénomènes d'interférences dans les caches, car les caches ont aussi une taille multiple de deux. Dans ce cas, les transformations de boucles sont insuffisantes. Augmenter l'associativité du cache n'est pas toujours suffisant, et [BS95] propose par exemple d'utiliser des caches brouillés. Une autre solution est d'utiliser un cache conventionnel et d'avoir recours au padding:

Bacon et al. [BGS94] proposent une solution pour supprimer les self- et les cross-interférences qui sont générées dans la boucle interne d'un nid de boucles, en combinant *padding* inter- et intra-matrice. Ils proposent aussi d'utiliser le padding intra-ligne pour améliorer la performance du cache non bloquant. Leur étude porte sur des accès de pas unitaire aux matrices (*stride 1*).

Dans sa thèse, Shieh [Shi96] propose une étude du *padding* pour des matrices bloquées (*tiling*). Il utilise les propriétés des notations de congruence pour calculer le padding à effectuer.

Gosh et al. [GMM97] proposent une approche plus complète du padding, qui tient compte de tous les paramètres de boucle. Ils peuvent ainsi en théorie avoir une connaissance exacte des interférences ayant lieu entre les matrices afin de déterminer le padding optimum à appliquer. Ils utilisent les équations d'interférences et les vecteurs de réutilisation des éléments des matrices, afin de fournir une solution de *padding* minimisant le nombre d'interférences pour plusieurs niveaux d'un nid de boucle. Bien que cette approche soit très complète, elle est difficile à exploiter, car elle nécessite de résoudre des équations diophantiennes.

A.2 Résolution de l'équation d'interférences

Le *padding* consiste à déplacer les éléments de matrice afin qu'ils n'interfèrent pas dans le cache. Pour que deux éléments interfèrent, ils faut qu'ils se chargent dans la même ligne de cache. L'équation d'interférence exprime ce phénomène.

A.2.1 L'équation d'interférences

Nous étudions le cas d'interférences entre deux références à des matrices dans un nid de boucles. Nous allons voir comment appliquer le *padding* pour réduire les interférences afin de mieux exploiter la localité spatiale des lignes de cache.

Nous étudions le cas général de deux références à des matrices dans un nid de boucles. On peut l'écrire comme suit :

for($il = il_{min}$; $il < il_{max}$; il++)
  for($jl = jl_{min}$; $jl < jl_{max}$; jl++)
    for($kl = kl_{min}$; $kl < kl_{max}$; kl++) {
      M1[i_1, j_1]...;
      M2[i_2, j_2]...;
}

Les indexes de matrice sont fonctions des indices de boucle. Le plus souvent la relation est linéaire, mais on peut écrire plus généralement :

$i_1 = \Phi_1^i(il, jl, kl)$
$j_1 = \Phi_1^j(il, jl, kl)$
$i_2 = \Phi_2^i(il, jl, kl)$
$j_2 = \Phi_2^j(il, jl, kl)$

On note pour chaque matrice α :

M_α^0	Adresse de base de la matrice M_α
N_α	Dimension des colonnes de la matrice M_α
WS_α	la taille d'un élément de la matrice α
i_α	l'index de colonne de la matrice M_α (éléments contigus en mémoire)
j_α	l'index de rangée de la matrice M_α
Δ	$= M_1^0 - M_2^0$, distance séparant les matrices M_1 et M_2 en mémoire
o_i	$= i_2 - i_1$ différence d'index dans la dimension i entre deux éléments de M_1 et M_2.

On supposera un cache à correspondance directe ayant les paramètres suivant :

CS	taille du cache en octets
LS	taille d'une ligne du cache en octets

On utilise les notations suivantes :

$\lceil X \rceil$	partie entière de X
$XmodY$	X modulo Y, reste de la division entière de X par Y

Pour qu'il n'y ait pas d'interférences entre deux références $M_1[i_1, j_1]$ et $M_2[i_2, j_2]$ dans les boucles internes d'un nid de boucles, il faut s'assurer que les blocs mémoires qu'elles chargent dans le cache ne sont pas remplacés avant d'être complètement réutilisés par les boucles internes. L'équation d'interférence indique quand deux blocs de chacune des références se chargent dans la même ligne de cache. Il y a risque d'interférence s'il existe deux itérations des boucles internes telles que les références correspondent au même numéro de ligne de cache:

$$\left\lceil \frac{adresse(M_1[i_1, j_1])}{LS} \right\rceil = \left\lceil \frac{adresse(M_2[i_2, j_2])}{LS} \right\rceil mod[\frac{CS}{LS}]$$

Il faut s'assurer que cette équation n'est jamais vérifiée. L'équation s'exprime en fonction des indices de boucle $il, jl...kl$ en remplaçant les indexes de matrice. Si cette équation est vérifiée, pour qu'il y ait réellement une interférence il faut de plus que les bornes des indices de boucles permettent à la configuration d'apparaître de façon répétitive, sinon un élément en remplace juste un autre dans la ligne de cache, sans pour autant gêner l'utilisation de ce premier. Une telle analyse complète est difficile à exploiter [GMM97] car il faut étudier toutes les équations de réutilisation des éléments de chaque matrice en plus des équations d'interférence. En modifiant le placement mémoire des matrices, on s'assure qu'il n'y a pas de risque d'interférence.

On étudie les interférences dans les boucles internes du nid de boucles. Si on considère par exemple les boucles il et jl, en fixant $kl = kl_0$, on obtient un sous ensemble d'indices i_1, j_1, i_2, j_2 pour lequel l'équation d'interférence ne doit pas être validée:

On note $E_{i_1} \subset [0, N_1 - 1]$ l'ensemble des valeurs que peut prendre i_1 dans les boucles considérées. E_{i_1} est l'ensemble des i_1 touchés par l'une des itérations de l'ensemble de boucles considéré :

$$E_{i_1} = \{\forall i \in [0, N_1 - 1], \exists il \in [il_{min}, il_{max}], \exists jl \in [jl_{min}, jl_{max}] | i = \Phi(il, jl, kl_0)\}$$

On définit de manière similaire E_{j_1}, E_{i_2}, E_{j_2}. Il y a interférence si :

$$\forall i_1 \in E_{i_1}, j_1 \in E_{j_1}, i_2 \in E_{i_2}, j_2 \in E_{j_2}$$

$$\exists k \in I\!N^* | \quad \left\lceil \frac{adresse(M_1[i_1, j_1])}{LS} \right\rceil - \left\lceil \frac{adresse(M_2[i_2, j_2])}{LS} \right\rceil = \frac{k \times CS}{LS}$$

soit, en explicitant les adresses:

$$\exists k \in I\!N^* | \quad \left\lceil \frac{(M_1^0 + WS_1 \times (i_1 + N_1 \times j_1))}{LS} \right\rceil - \left\lceil \frac{(M_2^0 + WS_2 \times (i_2 + N_2 \times j_2))}{LS} \right\rceil = \frac{k \times CS}{LS}$$

En multipliant l'équation par LS et en simplifiant les parties entières[1], on fait apparaître quatre termes (Δ est la distance séparant les deux matrices en mémoire).

$$\underbrace{\Delta}_{(1)} + \underbrace{WS_1 \times (i_1 + N_1 \times j_1) - WS_2 \times (i_2 + N_2 \times j_2)}_{(2)} = \underbrace{k \times CS}_{(3)}$$

$$+ \underbrace{\Big(M_1^0 + WS_1 \times (i_1 + N_1 \times j_1)\Big) modLS - \Big(M_2^0 + WS_2 \times (i_2 + N_2 \times j_2)\Big) modLS}_{(4)}$$

Il est possible d'influer sur l'un de ces quatre termes pour s'assurer que l'équation n'est jamais vérifiée.

- Les techniques de transformations de boucles changent le domaine d'itération et l'ordre des boucles. En modifiant $\Phi_1^i, \Phi_1^j, \Phi_2^i, \Phi_2^j$, elles influent sur le second terme[2].
- Le *padding* permet de modifier M_1^0, M_2^0, N_1, N_2 et influe donc sur le premier et le second terme.
- Le troisième terme dépends de la configuration matérielle du cache.

1. On a $\lceil \frac{A}{B} \rceil \times B = A - AmodB$.

2. Les transformations de boucles influent aussi sur l'ordre de réutilisation des lignes. Cela n'est pas pris en compte par l'équation d'interférence.

- Le quatrième terme fait intervenir la taille des lignes du cache et doit être considéré à part. Il représente l'influence de la position de chaque élément en mémoire par rapport aux frontières de lignes de cache. Le padding intra-ligne utilise ce terme pour distribuer les défauts de cache uniformément sur les itérations de boucles.

Note : Le terme (4) indique si les matrices ne sont pas exactement alignées dans les lignes de cache (décalage possible jusqu'à $LS - WS$). On peut le réecrire en exprimant l'adresse de $M_1[i_1, j_1]$ en fonction de celle de $M_2[i_2, j_2]$:

$$(\Delta + WS_1 \times (i_1 + N_1 \times j_1) - WS_2 \times (i_2 + N_2 \times j_2) + M_2[i_2, j_2]) modLS - (M_2[i_2, j_2]) modLS$$

Si le terme (4) s'annule pour un $M_2[i_2, j_2]$ donné, il s'annulera aussi pour un $M_2[i'_2, j'_2]$ tel que $M_2[i'_2, j'_2] modCS = 0$. On peut donc étudier l'équation d'interférence en simplifiant le terme (4). Pour trouver des séries d'éléments pouvant interférer, on remplace $M_2[i_2, j_2]$ par zéro.
Cependant, il n'y a en fait des risques d'interférences que pour les itérations où $M_1[i, j]$ et $M_2[i, j]$ annulent le terme (4). Le pourcentage des itérations trouvées pouvant effectivement causer des interférences est donné par :

$$\frac{(\Delta + WS \times (o_i + N \times o_j)) modLS}{LS} \times 100$$

Pour savoir si une configuration risque de causer des interférences, le terme (4) de l'équation d'interférence peut donc être simplifié.

Le quatrième terme est simplifié en remplaçant $M_2[i_2, j_2]$ par zéro. L'équation d'interférence devient :

$$\Delta + WS_1 \times (i_1 + N_1 \times j_1) - WS_2 \times (i_2 + N_2 \times j_2) = k \times CS +$$

$$(\Delta + WS_1 \times (i_1 + N_1 \times j_1) - WS_2 \times (i_2 + N_2 \times j_2)) modLS$$

Pour voir si l'équation est vérifiée, il faut remplacer i_1, j_1, i_2 et j_2 par il, jl et kl_0. On obtient une équation avec les indices de boucle. Trois cas se présentent.

- L'équation ne peut jamais être vérifiée. Il n'y a donc pas d'interférences.
- L'équation se vérifie pour quelques valeurs spécifiques de il et jl. Dans ce cas, il y a ponctuellement un risque d'interférence. Ces interférences occasionnelles influent peu sur la performance du cache car elles ne sont pas systématiques.
- L'équation est vérifiée indépendament de la valeur de kl_0. Dans ce cas, il y a des interférences systématiques à chaque itération de la boucle. Il est alors important d'intervenir soit par padding, soit avec des transformations de boucles, surtout si les indices de boucles qui se simplifient sont ceux des boucles les plus internes, car on est dans une situation d'effet ping-pong.

- Un ou plusieurs indices des boucles internes se simplifient et l'équation est vérifiée. Il peut y avoir interférence a l'intérieur d'une itération de la boucle dont l'indice est simplifié, ou entre plusieurs itérations. Dans le premier cas, on se ramène à la situation précédente en étudiant directement la boucle incriminée. On est aussi dans une situation d'effet ping-pong. Il est donc là aussi important d'améliorer la localité spatiale du programme.

Pour analyser un nid de boucles, il est préférable de commencer l'optimisation du placement en commencant par la boucle la plus interne et en remontant vers la boucle la plus externe.

Les cas les plus fréquemment étudiés sont ceux causant l'effet ping-pong, car il dégradent très nettement la performance des caches.

A.2.2 Effet de ping-pong

Le phénomène de ping-pong apparaît lorsqu'une boucle du programme accède systématiquement à des éléments mémoires qui se chargent dans la même ligne de cache et se chassent, empêchant le cache de tirer parti de la localité spatiale ou temporelle.

Lorsque ce phénomène est généré par les références d'une même itération de la boucle interne, les transformations de boucles ne peuvent pas les éliminer. Le *padding* devient alors un complément indispensable pour éliminer l'effet ping-pong.

Les techniques de transformation de boucle agissent sur le terme (2) de l'équation d'interférence, par l'intermédiaire des indices de boucle qui agissent sur i_i et j_i. Cependant, l'équation peut être indépendante de ces indexes.

Si on étudie les interférences de la boucle interne, il suffit que :

- Les matrices utilisent des données de même taille, $WS = WS_1 = WS_2$,
- Les matrices aient la même dimension, $N = N_1 = N_2$,
- Les fonctions Φ soient telles que les indexes de matrice sont identiques à une constante près, $o_i = i_2 - i_1$ et $o_j = j_2 - j_1$.

Note : Pour éliminer les interférences entre plusieurs niveaux de boucles internes, il faut garantir que l'équation d'interférence n'est vérifiée pour aucun des éléments référencés. Cela génère un système d'équations d'interférence, une équation par paire d'éléments référencés.

Les trois conditions ci-dessus sont fréquemment rencontrées, car les programmes font souvent des calculs sur des matrices carrées contenant le même type de donnée, et les indexes sont souvent égaux aux indices de boucle à une constante près.

L'équation d'interférence s'écrit :

$$\underbrace{(\Delta + WS \times (o_i + N \times o_j))}_{(A)} modCS = (\Delta + WS \times (o_i + N \times o_j))modLS$$

Il y aura des risques d'interférences qui ne peuvent être corrigées par transformations de boucles si l'équation d'interférence est vérifiée pour certaines itérations de la boucle interne:

- $WS = WS_1 = WS_2$,
- $N = N_1 = N_2$,
- $\underbrace{(\Delta + WS \times (o_i + N \times o_j))}_{(A)} modCS \in [0, LS - 1]$

(A) correspond à la distance séparant en mémoire deux éléments de matrice référencés dans le même nid de boucle. Pour qu'il y ait interférence, il faut aussi qu'il y ait de la localité à exploiter entre les blocs mémoires qui se chassent d'une même ligne de cache. C'est à dire qu'il faut que l'équation d'interférence soit vérifiée pour des indexes du domaines d'itérations, et que l'ordre des accès génère une réutilisation de ces blocs mémoire. Cette dernière condition est plus complexe à vérifier, il faut tenir compte des vecteurs de réutilisation. L'étude se contente donc de donner des méthodes pour que l'équation d'interférence ne soit pas vérifiée.

A.2.3 Eliminer les cross-interférences

Les cross-interférences peuvent apparaître quand deux matrices sont référencées dans un nid de boucles.

Cas où $o_i = 0$ et $o_j = 0$

Soient deux matrices carrées accédées par le nid de boucle suivant :

```
for(i=0; i < i_max; i++)
  for(j=0; j < j_max; j++)
    M_1[i,j] += M_2[i,j];
}
```

Pour qu'il n'y ait pas d'interférences dans la boucle interne, il suffit de s'assurer que $\Delta modCS \notin [0, LS - 1]$. Si les matrices sont côte-à-côte et que M_2 est rangée avant M_1 on peut écrire :

$$\Delta = M_1^0 - M_2^0 = (M_2^0 + taille(M_2)) - M_2^0 = N^2 \times WS$$

On a donc des interférences si :

$$(N^2 \times WS) modCS \in [0, LS - 1]$$

CS est une puissance de deux. Si l'on note $CS = 2^c$, il faut pour qu'il y ait interférence que $\exists k \in N | \Delta = 2^c \times k$. En notant $WS = 2^w$, il suffit que $N = 2^{\frac{c-w}{2}} \times k$ pour avoir des interférences.

C'est le cas par exemple quand N est une puissance de deux, ce que l'on rencontre dans certains programmes, comme *swm256* qui utilise des matrices 256x256. Pour avoir des interférences avec un cache de 16KO avec des matrices de *float*, il suffit que $N = k \times 64$

Il y a deux solutions pour palier à ce problème :

- le padding intra-matrice (LSP [3]) : On modifie la dimension interne des matrices en rajoutant μ éléments, $\mu \in [\frac{LS}{WS} - \frac{CS-LS}{WS}]$ de façon à ce que l'équation d'interférence ne soit plus vérifiée.
  $(N^2 \times WS + \mu \times WS) mod CS \notin [0, LS-1]$
- le padding inter-matrice (IAP [4]) : On intercale entre les deux matrices un élément de taille Λ, $\Lambda \in [LS, CS-LS]$ tel que l'équation d'interférence ne soit plus vérifiée.
  $(N^2 \times WS + \Lambda)\ mod\ CS \notin [0, LS-1]$

Par exemple pour *swm256* avec des lignes de cache de 32 octets, on peut donc soit déclarer des matrices 257x256, soit intercaler 32 octets de données entre deux matrices qui interfèrent. Cela se fait avec des modifications mineures au niveau de la déclaration des variables, sans modifier le reste du code.

Bien que le cas des matrices dont les dimensions sont des puissances de deux ait été stigmatisé, les calculs précédents montrent qu'il y a des interférences dès que la dimension des matrices vaut $N = k \times \sqrt{CS/WS}, k \in N$.

Ce résultat est vérifié par l'expérience. La figure A.2 est obtenue pour un nid de boucles référençant deux matrices telles que $o_i = o_j = 0$. Elle montre notamment des interférences pour $N = 192 = 3 \times 64$ ou $N = 320 = 5 \times 64$. La figure montre aussi les interférences qui apparaissent pour des matrices rectangulaires $N \times M$ où $M = 256$. Dans ce cas il y a interférence pour $N = k \times \frac{CS}{M \times WS}, \forall k \in N$, soit pour l'exemple choisi $N = k \times 32$.

Cas où $o_i \neq 0$

Dans ce cas, l'équation d'interférence devient :

$$(N^2 \times WS + o_i \times WS)\ mod\ CS \in [0, LS-1]$$

Souvent, o_i est petit, compris entre $[-2, 2]$. Un décalage sur l'indice i a le même effet que le *padding* inter-matrices. Par exemple, si $o_i = \frac{LS}{WS}$ l'équation d'interférence n'est plus vérifiée (par exemple, $o_i = 8$ suffit si $LS = 32$ et $WS = 4$). Lorsqu'on fait du padding inter-matrices, il faut donc prendre en compte le décalage généré par o_i.

Cas où $o_j \neq 0$

Dans ce cas, l'équation d'interférence devient :

$$(N^2 \times WS + o_j \times N \times WS)\ mod\ CS \in [0, LS-1]$$

3. *Leading Size Padding*
4. *Inter Array Padding*

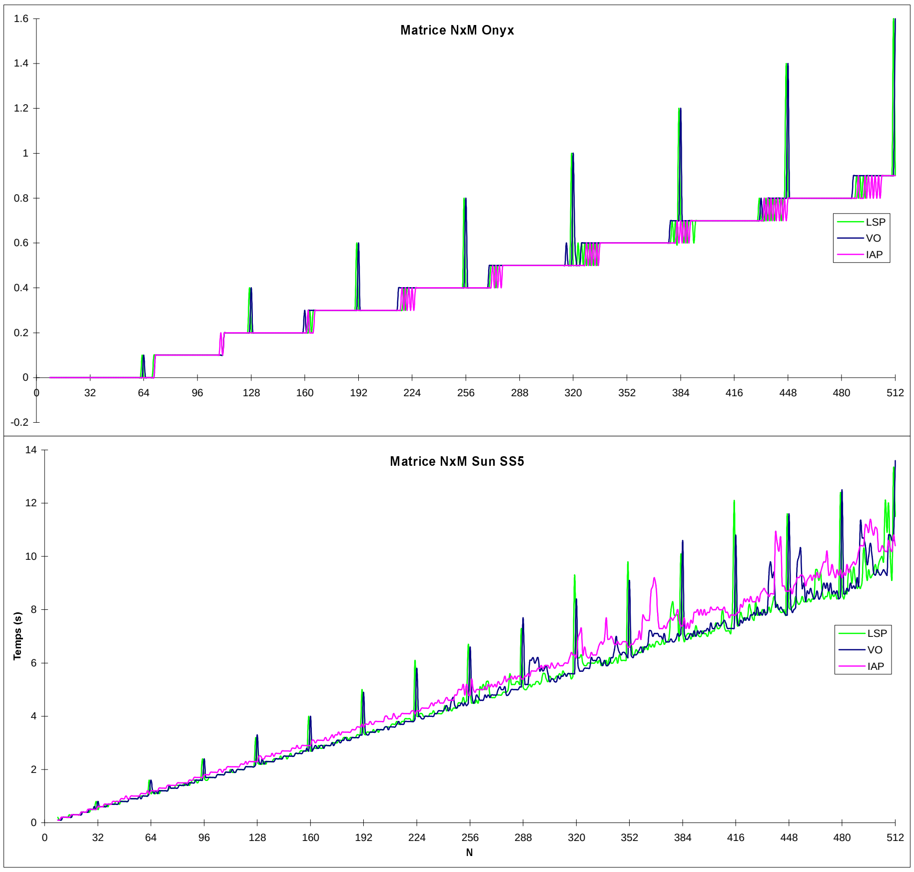


FIG. A.2 – *Interférences pour N multiple de la taille du cache*

Comme précédemment, o_j agit comme du padding inter-matrices. Si $o_j \times N \times WS\ mod\ CS \neq 0$ il y a peu de risque que l'équation soit vérifiée. Pour qu'il y ait interférence, il faut que la taille de o_j colonnes corresponds à la taille du cache. Le padding intra- ou inter-matrice peut être utilisé pour éviter les interférence.

A.2.4 Eliminer les self-interférences

Pour étudier les self-interférences, il suffit de reprendre l'équation d'interférence en considérant $\Delta = 0$. Comme il n'est pas possible de faire du padding inter-matrice, il ne reste que le padding intra-matrice pour éviter les interférences.

Le cas de la boucle ci-dessous illustre bien le type d'interférence qui peut être généré.

for(j=0; $j < j_{max} - o_j$; j++)
  for(i=0; $i < i_{max} - o_i$; i++)
    $M_2[i, j]$ += $M_2[i + o_i, j + o_j]$;
}

Comme $\Delta = 0$, o_i aura plutôt tendance à générer de la localité spatiale que des interférences. Par contre o_j nécessite de charger plusieurs colonnes de la matrice. Pour éviter qu'elles ne se chargent dans la même ligne de cache, il suffit que :

$(WS \times N \times o_j) modCS \notin [0, LS - 1]$

Pour cela, il suffit que N ne soit pas un multiple de $\frac{CS}{WS}$. Comme CS est une puissance de deux, il suffit que N n'en soit pas une.

A.2.5 Interférences entre plusieurs itérations d'une boucle

Lorsque les matrices ont des colonnes plus petites que la taille du cache, il est possible les conserver dans le cache sur plusieurs itérations. Pour exploiter la localité spatiale sur toute une colonne de matrice, il faut placer judicieusement les autres éléments accédés afin d'éviter les interférences tant que cette rangée n'est pas réutilisée complètement.

Prenons le cas de la boucle suivante où $o_j \neq 0$:

for(j=0; $j < j_{max} - o_j$; j++)
  for(i=0; $i < i_{max} - o_i$; i++)
    $M_1[i, j]$ += $M_2[i, j]$+$M_2[i + o_i, j + o_j]$;
}

Il y a risque d'interférence entre deux itérations de jl si $\exists o_i \in [0, i_{max}]$ tel que l'une des équations suivante est vérifiée :
(1) $(WS \times (o_i + N \times o_j)) modCS \in [0, LS - 1]$ (self-interférence de M_2),
(2) $(\Delta + WS \times o_i) modCS \in [0, LS - 1]$ (cross-interférence entre $M_1[i, j]$ et $M_2[i, j]$) ou
(3) $(\Delta + WS \times (o_i + N \times o_j)) modCS \in [0, LS - 1]$ (cross-interférence entre $M_1[i, j]$ et $M_2[i, j + o_j]$)

Comme o_i balaye son champ de définition, cela correspond au recouvrement dans le cache des colonnes des matrices. En pratique, garantir que les colonnes ne se recouvrent pas dans le cache est une contrainte trop forte, surtout si beaucoup de matrices sont référencées. En analysant la réutilisation des champs, on voit que $M_2[i, j]$ est réutilisé après o_j itérations de la boucle interne. Ce sont les seules colonnes à conserver entre plusieurs itérations.

Pour éviter les interférences sur la boucle j, il faut éviter les interférences sur la boucle i en garantissant ces équations :
(1) $(WS \times N \times o_j) modCS \in [LS, CS]$ (self-interférence de M_2),
(2) $(\Delta) modCS \in [LS, CS]$ (cross-interférence entre $M_1[i, j]$ et $M_2[i, j]$) ou
(3) $(\Delta + WS \times N \times o_j) modCS \in [LS, CS]$ (cross-interférence entre $M_1[i, j]$ et $M_2[i, j + o_j]$)

et en garantissant que $M_2[i, j]$ n'est pas recouvert dans le cache :
(4) $(WS \times N \times o_j) modCS \in [LS, CS - N \times WS]$ (self-interférence de M_2 sur l'itération j),
(5) $(\Delta) modCS \in [LS, CS - N \times WS]$ (cross-interférence sur l'itération j).

A.3 Conclusion

Les transformations de boucles sont souvent utilisées pour éliminer les interférences entre matrices dans le cache. Le *padding* est un bon complément, car il permet de lutter contre l'effet ping-pong.

Pour les appliquer, il faut déterminer la quantité de *padding* à faire en utilisant l'équation d'interférence. Les modifications de code à apporter sont minimes car il suffit de modifier la déclaration des matrices, sans changer le reste du code. Nous avons vu comment ces modifications permettent d'éliminer les self- et les cross-interférences.

Les études sur le padding sont encore peu nombreuses. En effet, les calculs d'adresse pour déterminer le padding à appliquer sont soit relativement simples, soit complexes si l'on fait intervenir les vecteurs de réutilisation des matrices.

Bibliographie

[ABL97] Ammons (G.), Ball (T.) et Larus (J. R.). – Exploiting hardware performance counters with flow and context sensitive profiling. *In : ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pp. 171–182. – 1997.

[AG92] Almasi et Gottlieb. – *Highly Parallel Computing, 2nd ed.*, chap. 3.2 Memory technology (Dennard's folly). – Benjamin Cummings, 1992.

[AK96] Alexander (T.) et Kedem (G.). – Distributed prefetch-buffer/cache design for high performance memory systems. *In : Second High-Performance Computing Architecture*, p. 254. – 1996.

[AP93] Agarwal (A.) et Pudar (S. D.). – Column associative caches: A technique for reducing the miss rate of dm caches. *Proceedings of the 20th International Symposium on Computer Architecture*, 1993.

[ASU86] Aho (A. V.), Sethi (R.) et Ullman (J. D.). – *Compilers principles, Techniques and Tools.* – Addison Wesley, 86.

[BC91] Baer (J. L.) et Chen (T. F.). – An effective on chip preloading scheme to reduce data access penalty. *Proceedings of Supercomputing 91*, 1991.

[BCJ94] Bacon (D. F.), Chow (J-H.) et Ju (D. R.). – A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. *Proceedings of CASCON'94*, novembre 1994.

[BD94] Boland (K.) et Dollas (A.). – Predicting and precluding problems with memory latency. *IEEE Micro*, août 1994, pp. 59–67.

[Bel66] Belady (L. A.). – A study of replacement algorithms for a virtual-storage computer. *IBM System Journal*, 1966, p. 24.

[BGK95] Burger (D. C.), Goodman (J. R.) et Kägi (A.). – *The Declining Effectiveness of Dynamic Caching for General Purpose Multiprocessor.* – Rapport technique n° 1261, http://www.cs.wisc.edu/g̃alileo, University of Winsconsin, 1995.

[BGK96] Burger (D.), Goodman (J. R.) et Kägi (A.). – Quantifying memory bandwidth limitations of current and future microprocessors. *23rd International Symposium on Computer Architecture*, 1996.

[BGS94] Bacon (D. F.), Graham (S. L.) et Sharp (O. J.). – Compiler transformations for high-performance computing. *ACM Computing Surveys*, vol. 26, n° 4, décembre 1994, pp. 345–420.

[Boz84] Bozman (G.). – The software lookaside buffer reduces search overhead with linked lists. *Communications of the ACM*, vol. 27, 1984, pp. 222–227.

[BS95] Bodin (F.) et Seznec (A.). – Skewed associativity enhances performance predictability. *Proceedings of the 22nd International Symposium on Computer Architecture*, vol. 23, n° 2, mai 1995, p. 265.

[BZ93] Barrett (D. A.) et Zorn (B. G.). – Using lifetime predictors to improve memory allocation performance. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, vol. 28, n° 6, juillet 1993, pp. 187–196.

[CB92] Chen (T. F.) et Baer (J. L.). – Reducing memory latency via non blocking and prefetching caches. *5th symposium on Architectural Support for Programming Languages and Operating Systems*, 1992.

[CB95] Chen (T. F.) et Baer (J. L.). – Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, vol. 44, n° 5, mai 1995, pp. 609–623.

[CB96] Cvetanovic (Z.) et Bhandarkar (D.). – Performance characterization of the alpha 21164 microprocessor using tp and spec workloads. *In : Seconde High-Performance Computing Architecture*, p. 270. – 1996.

[CG96] Chang (J. M.) et Gehringer (E. F.). – A high-performance memory allocator for object-oriented systems. *IEEE Transactions on Computers*, vol. 45, n° 3, mars 1996, pp. 357–367.

[CGZ94] Calder (B.), Grunwald (D.) et Zorn (B.). – Quantifying behavioural differences between c and c++ programs. *Programming Languages*, vol. -, n° 2, 1994, pp. 313–351.

[Che95] Chen (T. F.). – An effective programmable prefetch engine for on-chip caches. *28th Annual International Symposium on Microarchitecture*, 1995.

[CKP91] Callahan (D.), Kennedy (K.) et Porterfield (A.). – Software prefetching. *4th symposium on Architectural Support for Programming Languages and Operating Systems*, 1991.

[CKT94] Carr (S.), Kinley (K. S. Mc) et Tseng (C-W.). – Compiler optimisation for improving data locality. *proceedings of the 4th symposium on Architectural Support for Programming Languages and Operating Systems*, 94, p. 11.

[CL95] Cierniak (M.) et Li (W.). – Unifying data and control transformations for distributed shared-memory machines. *In : Conference on Programming Language Design and Implementation*, pp. 205–217. – 1995.

[CL96] Cohn (R.) et Lowney (P. G.). – Hot cold optimization of large windows/nt applications. *In : 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 80–89. – décembre 1996.

[CM95] Coleman (S.) et McKinley (K. S.). – Tile size selection using cache organization and data layout. *In : Conference on Programming Language Design and Implementation*, p. 279. – 1995.

[CR95] Carlisle (M. C.) et Rogers (A.). – *Software Caching and Computation Migration in Olden.* – Rapport technique n° TR-483-95, Princeton University, 1995.

[Den70] Denning (P. J.). – Virtual memory. *Computer Surveys*, vol. 2, n° 3, septembre 1970, pp. 153–190.

[Dix92] Dixit (K.). – New cpu benchmark suites from spec. *In : COMPCON.* – 1992.

[DTM94] Diwan (A.), Tarditi (D.) et Moss (E.). – Memory subsystem performance of programs using copying garbage collection. *In : 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* – janvier 1994.

[FJ94] Farkas (K. I.) et Jouppi (N. P.). – Complexity/performance tradeoffs with non-blocking loads. *Proceedings of the 21st International Symposium on Computer Architecture*, 1994, p. 12.

[FKJ+90] Finkel (D.), Kinicki (R. E.), John (A.), Nichols (B.) et Rao (S.). – Developing benchmarks to measure the performance of the mach operating system. *In : USENIX Mach Workshop*, pp. 83–100. – 1990.

[FKLC92] Finkel (D.), Kinicki (R. E.), Lehmann (J. A.) et CaraDonna (J.). – *Comparison of Distributed Operating System Performance Using the WPI Benchmark Suite.* – Rapport technique n° WPI-CS-TR-92-2, Worcester Polytechnic Institute, 1992.

[FPJ92] Fu (J. W. C.), Patel (J. H.) et Janssens (B. L.). – Stride directed prefetching in scalar processors. *Proceedings of Micro-25*, 1992.

[GHPS93] Gee (J. D.), Hill (M. D.), Pnevmatikatos (D. N.) et Smith (A. J.). – Cache performance of the spec92 benchmark suite. *IEEE Micro*, août 1993, pp. 17–27.

[GMM97] Ghosh (S.), Martonosi (M.) et Malik (S.). – Cache miss equations: An analytical representation of cache misses. *In : Workshop on Interaction between Compilers and Computer Architectures.* – février 1997. published in IEEE TCCA Newsletter, June 1997.

[Gos93] Gossmann. – N.a. *In : N.A.* – 1993.

[GZ93] Grunwald (D.) et Zorn (B.). – Customalloc: Efficient synthetized memory allocators. *Software Practice & Experience*, vol. 23, n° 8, août 1993, pp. 851–869.

[GZH93] Grunwald (D.), Zorn (B.) et Henderson (R.). – Improving the cache locality of memory allocation. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, vol. 28, n° 6, juillet 1993, p. 177.

[HC89] Hwu (W-M) et Chang (P. P.). – Achieving high instruction cache performance with an optimizing compiler. *In : 16th Annual International Symposium on Computer Architecture*, pp. 242–251. – mai 1989.

[Hei94] Heisch (R. R.). – Trace-directed program restructuring for aix executables. *IBM Journal of Research and Development*, septembre 1994, pp. 595–603.

[Hil87] Hill (M. D.). – *Aspects of cache memory and instruction buffer performance.* – Rapport technique n° UCB//CSD-87-381, University of California, Berkeley, 1987.

[HKC97] Hashemi (A. H.), Kaeli (D. R.) et Calder (B.). – Efficient procedure mapping using cache line coloring. *In : ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pp. 171–182. – 1997.

[HS95] Huang (A. S.) et Shen (J. P.). – A limit study of local memory requirements using value reuse profiles. *28th Annual International Symposium on Microarchitecture*, 1995.

[HS96] Huang (A. S.) et Shen (J. P.). – The intrinsic bandwidth requirements of ordinary programs. *In : 7th symposium on Architectural Support for Programming Languages and Operating Systems*, p. 105. – 1996.

[Irl92] Irlam (G.). – "spa" tracing and cache simulation package for sunos-4. – The package is available from gordoni@cs.adelaide.edu.au, 1992.

[JCSM96] Jacob (B. L.), Chen (P. M.), Silverman (S. R.) et Mudge (T. N.). – An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, vol. 45, n° 10, octobre 1996.

[JD91] Ju (Y-J.) et Dietz (H.). – Reduction of cache coherence overhead by compiler data layout and loop transformation. *In : 4th International Workshop on Languages and Compilers for Parallel Computing*, éd. par Springer-Verlag. – août 1991.

[Jou90] Jouppi (N. P.). – Improving direct mapped cache performance by the addition of a small fa cache and prefetch buffers. *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[Jou93] Jouppi (N. P.). – Cache write policies and performance. *Proceedings of the 20th International Symposium on Computer Architecture*, 1993.

[JW94] Jouppi (N. P.) et Wilton (S. E.). – Tradeoffs in two level on-chip caching. *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[Knu73] Knuth (D. E.). – *The Art of Computer Programming 2nd ed.*, chap. 2, pp. 435–451. – Addison Wesley, 1973.

[KOI95] Kumanoya (M.), Ogawa (T.) et Inoue (K.). – Advances in dram interfaces. *IEEE Micro*, décembre 1995, pp. 30–36.

[KV85] Korn (D.) et Vo (K.). – In search of a better malloc. *In : USENIX Association Conference*, pp. 489–506. – juin 1985.

[LH82] Leverett (B. W.) et Hibbard (P. G.). – An adaptative system for dynamic storage allocation. *Software Practice and Experience*, vol. 12, 1982, pp. 543–555.

[LM96] Luk (C-K.) et Mowry (T. C.). – Compiler-based prefetching for recursive data structures. *In : 7th symposium on Architectural Support for Programming Languages and Operating Systems.* – 1996.

[LSKR95] Lipasti (M. H.), Scmidt (W. J.), Kunkel (S. R.) et Roediger (R. R.). – Spaid: Software prefetching in pointer and call-intensive environments. *28th Annual International Symposium on Microarchitecture*, 1995.

[MBKQ96] McKusick (M. K.), Bostic (K.), Karels (M. J.) et Quarterman (J. S.). – *The Design and Implementation of the 4.4BSD Operating System*, chap. 5, Memory management. – Addison-Wesley, 1996.

[MBM97] Meneveaux (D.), Bouatouch (K.) et Maisel (E.). – *Memory Management Schemes for Radiosity Computation in Complex Environment.* – PI n° 1097, ftp.irisa.fr/techreports/1996/PI-1097.ps.gz, IRISA, 1997.

[McF89] McFarling (S.). – Program optimization for instruction caches. *3rd symposium on Architectural Support for Programming Languages and Operating Systems*, 1989.

[McK92] McKinley (K. S.). – *Automatic and Interactive Parallelisation.* – Thèse de PhD, Rice University, 1992.

[MK88] McKusick (M. K.) et Karels (M. J.). – Design of a general purpose memory allocator for the 4.3bsd unix kernel. *In : USENIX Association Conference*, pp. 295–403. – juin 1988.

[MPS94] Mendlson (A.), Pinter (S.) et Shtokhamer (R.). – Compile time instruction cache optimization. *In : Computer Architecture News*, pp. 44–51. – mars 1994.

[MS95] McVoy (L.) et Staelin (C.). – *Lmbench: Portable Tools for Performance Analysis.* – Rapport technique, http://www.sgi.com/employees/lm_engr/lmbench.tgz, Hewlett-Packard Laboratories, 1995.

[OA85] Oldehoeft (R. R.) et Allan (S. J.). – Adaptative exact-fit storage management. *Communications of the ACM*, vol. 28, 1985, pp. 506–511.

[PH90] Pettis (K.) et Hansen (R. C.). – Profile guided code positionning. *In : ACM SIGPLAN'90 Conference on Programming Languages and Design Implementation*, pp. 16–27. – juin 1990.

[PH96] Patterson (D.) et Hennessy (J.). – *Computer Architecture: A Quantitative Approach, 2nd ed.*, chap. 5, Memory hierarchy design. – Morgam-Kaufman, 1996.

[PK94] Palacharla (S.) et Kessler (R. E.). – Evaluating stream buffers as a secondary cache replacement. *Proceedings of the 21st International Symposium on Computer Architecture*, 1994, p. 10.

[PM96] Pierce (J.) et Mudge (T.). – Wrong-path instruction prefetching. *In : 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 165–175. – décembre 1996.

[Por89] Porterfield (A. K.). – *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* – Crpc-tr89009, Rice University - Center for Research on Parallel Computation, mai 1989.

[Prz90] Przybylski (S. A.). – *Cache and Memory Hierarchy Design.* – Morgan Kaufmann, 1990.

[Prz93] Przybylski (S.). – Drams for new memory systems. *Microprocessor Report*, mars 1993, pp. 8–23.

[RCRH95] Rogers (A.), Carlisle (M.), Reppy (J.) et Hendren (L.). – Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, vol. 17, n° 2, mars 1995.

[SA93] Sugumar (R. A.) et Abraham (S. G.). – Efficient simulation of caches under optimal replacement with application to miss characterization. *In : SIGMETRICS'93*. – 1993.

[Sez94] Seznec (A.). – Decoupled sectored caches: Consciliating low tag implementation cost and low miss ratio. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, p. 10.

[Shi96] Shih (Tien-Pao). – *Goal-Directed Performance Tuning for Scientific Applications*. – Thèse de PhD, University of Michigan, 1996.

[SL95] Seznec (A.) et Lloansi (F.). – *About effective cache miss penalty on out-of-order superscalar processors*. – Rapport technique n° PI-970, IRISA/INRIA, 1995.

[Smi86] Smith (A. J.). – Bibliography and readings on cpu cache memories and related topics. *Computer Architecture News*, janvier 1986, pp. 22–42.

[Smi91] Smith (M. D.). – *Tracing with Pixie*. – Rapport technique n° CSL-TR-91-497, ftp://steward.harvard.edu/users/smith, Stanford University, novembre 1991.

[SPN96] Saulsbury (A.), Pong (F.) et Nowatzyk (A.). – Missing the memory wall: The case for processor/memory integration. *In : 23rd International Symposium on Computer Architecture*. – 1996.

[SSGC95] Sheffler (Thomas J.), Schreiber (Robert), Gilbert (John R.) et Chatterjee (Siddhartha). – Aligning parallel arrays to reduce communication. *In : fifth Symposium on the Frontiers of Massively Parallel Computations*. – juin 1995.

[Ste83] Stephenson (C. J.). – Fast fits: New methods for dynamic storage allocation. *In : Ninth ACM Symposium on Operating System Principles*, pp. 30–32. – octobre 1983.

[Sug93] Sugumar (R. A.). – *Multi Configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs*. – Thèse de PhD, University of Michigan, 1993.

[SWG92] Singh (J. P.), Weber (W-D) et Gupta (A.). – *SPLASH: Stanford Parallel Applications for Shared-Memory*. – Rapport technique, Stanford University, 1992.

[SWG94] Singh (J. P.), Weber (W-D) et Gupta (A.). – Splash: Stanford parallel applications for shared memory. *Computer Architecture News*, vol. 20, n° 1, juin 1994, pp. 5–44.

[Tan94] Tanabe. – A 64mbit dram. *In : ICSSS*. – 1994.

[Tan95] Tanenbaum (A. S.). – *Modern Operating Systems*, chap. 3. – Morgan Kauffmann, 1995.

[TBS96] Truong (D. N.), Bodin (F.) et Seznec (A.). – *Accurate Data Layout Into Blocks May Boost Cache Performance.* – Rapport technique n° 1000, ftp.irisa.fr /techreports/1996/PI-1000.ps.gz, IRISA/INRIA, 1996. presented at the *Second Workshop on Interraction between Compilers and Computer Architecture*, San Antonio, TX, Feb. 1997.

[TBS98] Truong (D. N.), Bodin (F.) et Seznec (A.). – Improving locality of dynamically allocated data structures. *In : IEEE Conference on Parallel Architecture and Compilation Techniques (PACT'98).* – octobre 1998.

[TE95] Tullsen (D. M.) et Eggers (S. J.). – Effective cache prefetching on bus-based multiprocessor. *ACM Transactions on Computer Systems*, février 1995, pp. 57–88.

[Tem93] Temam (O.). – *Study and Optimisation of Numerical Codes Cache Behaviour.* – Thèse de PhD, University of Rennes 1, 1993.

[TFMP95] Tyson (G.), Farrens (M.), Matthews (J.) et Pleszkun (A. R.). – A modified approach to data cache management. *28th Annual International Symposium on Microarchitecture*, 1995.

[TXD95] Torrellas (J.), Xia (Chun) et Daigle (Russel). – Optimizing instruction cache performance for operating system intensive workloads. *In : First Symposium on High-Performance Computer Architecture*, pp. 360–369. – janvier 1995.

[WJ93] WiltonNorman (S. J. E.) et Jouppi (N. P.). – *An Enhanced Access and Cycle Time Model for On-Chip Caches.* – Rapport technique n° 93/5, Digital WRL, 1993.

[WL91] Wolf (M. E.) et Lam (M. S.). – A data locality optimizing algorithm. *In : Conference on Programming Language Design and Implementation.* – 1991.

[WM95] Wulf (W. A.) et McKee (S.A.). – Hitting the memory wall : Implication of the obvious. *Computer Architecture News*, vol. 23, n° 1, mars 1995, pp. 20–24. – Tech Report CS-94-48.

[WOT+95] Woo (S. C.), Ohara (M.), Torrie (E.), Singh (J. P.) et Gupta (A.). – The splash-2 programs: Characterization and methodological considerations. *In : 22nd Annual International Symposium on Computer Architecture*, éd. par ACM, pp. 24–36. – juin 1995.

[WW88] Weinstock (C. B.) et Wulf (W. A.). – Quickfit: An efficient algorithm for heap storage allocation. *ACM Sigplan Notices*, vol. 23, n° 10, octobre 1988, pp. 141–144.

[XT96] Xia (C.) et Torrellas (J.). – Instruction prefetching of systems codes with layout optimized for reduced cache misses. *In : 23rd Annual International Symposium on Computer Architecture*, pp. 271–282. – mai 1996.

[ZG92a] Zorn (B.) et Grunwald (D.). – Empirical measurment of six allocation-intensive c programs. *ACM Sigplan Notices*, vol. 27, n° 12, décembre 1992, pp. 71–80.

[ZG92b] Zorn (B.) et Grunwald (D.). – *Evaluating Models of Memory Allocation.* – Rapport technique n° CU-CS-603-92, University of Colorado, Boulder, juillet 1992.

[Zor93] Zorn (B.). – The measured cost of conservative garbage collection. *Software Practice & Experience*, vol. 23, n° 7, juillet 1993, pp. 733–756.

[ZT95] Zhang (Z.) et Torrellas (J.). – Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. *Proceedings of the 22nd International Symposium on Computer Architecture*, vol. 23, n° 2, mai 1995, pp. 188–199.

Table des matières

Table des figures

Résumé

La majorité des ordinateurs modernes est conçue avec une hiérarchie mémoire faite de caches. Cette étude montre l'utilité des techniques logicielles de placement de données en mémoire pour le taux d'échecs des caches. Des techniques de placement sont exposées pour différents types de données couramment utilisés par les programmeurs.

Le placement de données permet de compléter le fonctionnement matériel des caches pour améliorer de façon significative la performance de la hiérarchie mémoire sans générer de surcoût matériel.

Dans une première partie, il est fait une classification des défauts de cache pour évaluer l'impact du du placement des données sur le taux d'échecs des caches. Cette classification garde la philosophie du classement des 3C de M. D. Hill. A l'aide de cette classification, l'influence du placement des données dans les blocs mémoires sur le taux d'échecs des caches est mise en évidence. Un placement optimal des données en mémoire permettrait d'éliminer les défauts de conflit ainsi qu'une partie des défauts de capacité des programmes.

Pour confirmer cette étude, une heuristique a été développée pour placer des scalaires indépendants en mémoire. Pour cela, une méthode de mesure de la localité des références entre deux données a été mise au point. L'heuristique a été appliquée sur des traces de programmes. Les gains de localité obtenus ont confirmé qu'il est possible d'éliminer ainsi une grande partie des défauts de capacité et de conflit. L'heuristique ne peut cependant pas être utilisée comme telle. En effet, il est rare de trouver dans un programme des scalaires sans contraintes de placement. Le compilateur et les programmeurs génèrent de nombreuses contraintes de placement mémoire par le biais du typage des données.

Il faut adapter des techniques de placement mémoire à chaque type de données.

Des techniques de placement pour les structures de données allouées dynamiquement sont donc proposées, car de nombreux programmes utilisent ce type de donnée pour stocker en mémoire la majorité des données manipulées. Les gains de localité sont obtenus en réorganisant les champs d'une structure et en entrelaçant les champs de plusieurs instances en mémoire. Une bibliothèque d'allocation adaptée a été mise au point pour faciliter l'entrelacement. Conçue autour de l'allocation d'arènes, elle est rapide et elle maximise la localité spatiale entre les éléments alloués en mémoire. Utiliser une bibliothèque optimisée et des techniques de placement mémoire permet d'améliorer la performance des applications généralistes.

La performance des programmes scientifiques a toujours été un problème. De nombreuses études ont donc été faites pour améliorer la localité des accès aux matrices numériques. Les transformations de boucles ont été largement étudiées. Elles changent l'ordre des références aux matrices. Le *padding* quant à lui modifie le placement mémoire des éléments de matrices. Il est donc décrit en annexe.

Cette étude montre l'importance du placement des données en mémoire. Elle propose des solutions de placement pour différents types de données. L'utilisation de telles techniques permet de réduire le gaspillage de la capacité de stockage des caches.