RAPPORT DE RECHERCHE

L R I

RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION D'UNE HABILITATION A DIRIGER DES RECHERCHES

Nathalie DRACH-TEMAM

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud-LRI

11/2002

Rapport de Recherche N° 1340

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Université de Paris-Sud XI

Habilitation à diriger des recherches

Présentée et soutenue publiquement

par

Nathalie Drach-Temam

Le vendredi 8 novembre 2002

Analyse et Optimisation des Architectures de Processeurs : Vers des Modèles, Applications et Métriques Plus Réalistes

Jury

Présidente :		
Mme	Marie-Claude Gaudel	Université de Paris-Sud - Paris XI
Rapporteurs :		
M.	Jean-Loup Baer	Université de Washington (USA)
M.	Daniel Litaize	Université Paul Sabatier - Toulouse III
M.	Eduardo Sanchez	Ecole Polytechnique Fédérale de Lausanne (Suisse)
Examinateurs :		
M.	Pascal Sainrat	Université Paul Sabatier - Toulouse III
M.	André Seznec	IRISA - INRIA Rennes
M.	Claude Timsit	Université de Versailles - Saint-Quentin-en-Yvelines

Remerciements

Un grand MERCI :

- à Marie-Claude Gaudel pour m'avoir fait l'honneur de présider le jury de mon habilitation.

- à Jean-Loup Baer, Daniel Litaize et Eduardo Sanchez pour avoir accepté de rapporter sur mon travail et pour avoir assisté à ma soutenance.

- à Pascal Sainrat, André Seznec et Claude Timsit pour avoir accepté de faire partie de ce jury.

- à tous les membres du groupe architecture du LRI, Olivier, Frédéric, David, Gilles, Michaël, Sami, Sylvain, Yves, pour les échanges fructueux et les moments bien agréables. Egalement aux anciens du groupe, Alexandre, Claude, Jean-Luc, Julien et Grégory et par avance, à nos futurs collègues Christine et Albert.

- aux étudiants dont j'ai dirigé les recherches et qui ont contribué à ces travaux, en particulier Claude, Julien et Alexis.

- à l'ensemble des personnes du LRI qui rendent ce lieu très agréable et intéressant.

- à tous mes collègues du département enseignement (LRI, comme LIMSI) et notamment à Catherine, Martine, Denyse et Dominique pour leur disponibilité, leur efficacité et leur gentillesse.

- à mes compagnons d'habilitation, Khaldoun et Philippe.

- à deux super collègues-copines, Delia et Laurence.

- à mes parents et beaux-parents pour leur aide et leur soutien constants.

- à ma « petite famille », Olivier, Lisa et Nina, pour tellement de choses...

Contenu du mémoire

CHAPITRE 1

Introduction

Ce mémoire présente mes activités de recherche menées depuis fin 1994 au sein de l'équipe Architecture du Laboratoire de Recherche en Informatique de l'Université de Paris-Sud. Si le thème général de mes travaux de recherche est la conception de processeurs performants, mon effort de recherche s'est également porté sur les problèmes de méthodologie de mon domaine et leurs conséquences sur l'utilité et la pertinence des résultats de recherche.

Les systèmes informatiques sont généralement répartis en trois catégories : les systèmes informatiques hautes-performances (multiprocesseurs), les systèmes informatiques généralistes (les PCs ou stations de travail) et les systèmes embarqués (téléphonie mobile, ...). La plupart de ces systèmes utilisent des architectures de processeurs de plus en plus similaires parce que les besoins en performance des applications embarquées ou généralistes sont aussi importants que celles des applications de calcul. Maintenir l'augmentation de la performance des processeurs est primordial pour la quasi-totalité de ces systèmes.

Les performances des processeurs augmentent très rapidement avec l'évolution de la technologie[1] et comprendre comment adapter l'architecture des processeurs aux progrès et aux performances attendues est un défi constant. Avec un milliard de transistors à l'horizon 2010, la complexité des processeurs augmente très rapidement et cette complexité rend la tâche de l'architecte de plus en plus difficile.

En raison de cette complexité, l'évaluation d'une architecture de processeurs est réalisée avant tout prototypage à l'aide de la simulation logicielle. Cette simulation consiste à exécuter un programme qui modélise l'architecture à évaluer. Par ailleurs, afin de simuler l'exécution d'un programme sur l'architecture, une suite d'instructions doit être soumise au simulateur (la charge de travail). Mais il faut avant tout définir précisément le cadre de cette évaluation :

- définir le comportement fonctionnel de l'architecture évaluée, puis la modéliser le plus finement possible,
- définir le contexte d'utilisation de cette machine en déterminant notamment le type d'applications exécutées,
- définir les critères de performance de la machine liés à ses contraintes de fonctionnement et à son utilisation[2] : vitesse d'exécution, consommation, coût de développement, traitement temps réel, etc.

A partir de ces différentes définitions (modèles, applications, métriques), une analyse de l'architecture proposée peut être réalisée. Cette analyse consiste à identifier les facteurs limitatifs de la performance et à

[1] Nous avons choisi volontairement le terme évolution pour introduire ce phénomène, car la progression des performances des processeurs n'est pas uniquement liée aux progrès technologiques, elle dépend également de facteurs sociaux et économiques. Le succès de l'informatique et la concurrence entre firmes déterminent les bénéfices de l'industrie des semi-conducteurs, bénéfices qui fixent l'investissement en recherche de cette industrie. La loi de Moore a également joué un rôle important dans l'évolution de l'informatique. Formulée en 1965 par Gordon Moore, elle postule le doublement annuel des performances des circuits intégrés. Cette loi a contribué à accélérer le rythme de l'innovation et l'industrie s'est fondée sur elle pour fixer ses objectifs et ses plans d'investissements (prophétie "auto-réalisatrice").

[2] Ces deux derniers points notamment distinguent le type de système informatique évalué (haute-performance, généraliste, embarqué).

proposer des optimisations matérielles ou logicielles. Ensuite, l'architecture optimisée est de nouveau analysée. Ce procédé peut être itéré un grand nombre de fois. La description de cette méthodologie est illustrée par la figure 1.1.

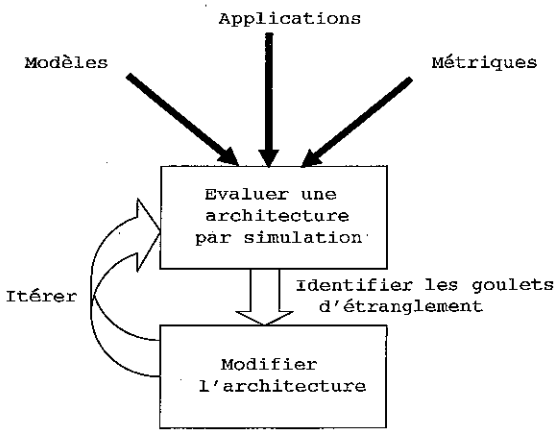

FIG. 1.1: Analyse et conception d'une architecture.

Pour que cette évaluation et ces optimisations apportent une réelle contribution, il faut considérer des modèles de l'architecture, des applications cibles et des métriques les plus réalistes possibles. Il est inutile d'évaluer la vitesse de calcul d'une architecture pour des applications scientifiques lorsque cette architecture est destinée à être utilisée dans un appareil embarqué traitant quasi-exclusivement des applications de type traitement du signal. Par ailleurs, ce souci de considérer des paramètres les plus réalistes possibles est motivé par le fait que la recherche en architecture à court et moyen termes est fortement liée à l'industrie, il faut donc élaborer des solutions réellement utiles et pertinentes.

Plus de réalisme dans les différentes phases d'analyse et de conception

Intégrer plus de précision dans les modèles Une fois défini le comportement fonctionnel des différents composants du processeur, il faut les modéliser le plus précisément possible. Par ailleurs, il faut intégrer dans cette modélisation les interactions temporelles et fonctionnelles entre ces différents composants. Cette tâche n'est pas aisée notamment du fait de la complexité des architectures actuelles. Et pourtant négliger certains comportements peut aboutir à des résultats erronés. Il est, par exemple, important lorsque l'on propose un mécanisme d'exécution spéculatif de le modéliser le plus correctement possible afin de mesurer non seulement son apport en terme de performance, mais également les perturbations générées sur le comportement normal (sans ce mécanisme) du processeur.

Considérer des applications réelles La plupart des études en microarchitecture utilisent pour les tests des programmes standards appelés *benchmarks*. Ces suites de programmes sont composées de programmes censés représenter un large spectre d'applications. Pour les processeurs généralistes, la suite la plus communément utilisée est la suite des SPEC CPU remise à jour régulièrement (SPEC92, SPEC95, SPEC2000). Elle a été conçue afin de mesurer les performances de calculs dits intensifs sur une grande variété de machines. Les applications proviennent de programmes réels (par opposition à des programmes tests ou synthétiques) et permettent de mesurer la performance des processeurs, de la mémoire et des compilateurs. Mais ces

programmes ne comportent pas ou peu d'applications multimédia, alors qu'elles sont devenues la principale charge de travail exécutée par un ordinateur personnel et représentent un marché en pleine croissance depuis 1995. Cette croissance des applications multimédia est due notamment aux évolutions suivantes :

- la performance des machines - elle permet depuis quelques années d'envisager de nombreuses applications multimédia, la compression/décompression en temps réel de son ou de vidéo, les jeux et autres applications en 3D, etc. L'intégration de nouvelles fonctionnalités sur les systèmes à base de processeurs généralistes, notamment les extensions multimédia ou les cartes graphiques performantes dédiées au traitement d'images et de vidéo, illustre bien ce phénomène.
- le développement d'Internet et la diffusion de grandes quantités d'informations - les applications multimédia sont promises à un déploiement rapide dans les années à venir. Consommant aujourd'hui de l'ordre de 2% de la bande passante de l'Internet, elles devraient compter pour plus de 6% d'ici à 2003 et intégrer des applications nécessitant des capacités temps réel comme la transmission de la voix, la musique ou la vidéo.
- la normalisation des applications - cette normalisation facilite leur diffusion et leur utilisation. La plupart des applications multimédia utilisent des formats de description normalisés (exemples : JPEG pour les images, MPEG2 pour la vidéo), même si ce n'est pas encore le cas pour les applications 3D (exemples : VRML, OpenGL et Direct3D pour le rendu 3D polygonal). Cette normalisation permet une plus large diffusion/utilisation des optimisations logicielles et matérielles proposées.

La croissance des applications multimédia et leur besoin en performance nécessitent de mettre particulièrement l'accent sur les applications multimédia lors de la conception de processeurs généralistes et embarqués. A l'heure actuelle, la plupart des travaux en architecture continuent à considérer les *benchmarks* SPEC entiers et flottants comme une charge représentative.

Redéfinir la performance La rapidité d'exécution n'est plus la seule contrainte à considérer même si elle l'a été très longtemps dans l'évaluation des processeurs généralistes et haute-performance. En effet, la dissipation de ces machines est de plus en plus importante (exemples : l'Itanium d'Intel consomme 130W à 800Mhz et l'Athlon XP d'AMD consomme 60W à 1.6GHz). Même si les différentes évolutions technologiques qui ont conduit à une diminution de la taille de gravure et du voltage permettent *a priori* une réduction de la dissipation, elles mènent également à une augmentation de la dissipation. En effet, l'augmentation du nombre de transistors entraîne une augmentation du nombre d'instructions exécutées en parallèle, une exécution dans le désordre, plus d'unités fonctionnelles, une spéculation des données, des techniques de prédiction de branchement sophistiquées, une augmentation de la longueur du pipeline, etc. Ces diverses techniques accroîssent le taux d'utilisation des transistors et augmentent également la dissipation.

Cette augmentation de la dissipation des machines a un impact sur leur utilisation. Dans le cadre des processeurs généralistes, l'augmentation de la dissipation augmente la dissipation ambiante, limite l'utilisation de ces processeurs sur batterie, diminue la durée de vie de ces machines et engendre un coût supplémentaire pour les dispositifs de refroidissement. Dans le cadre des processeurs embarqués, ces mêmes phénomènes se produisent, mais leur importance est accentuée (importance prépondérante de l'utilisation de batterie, des coûts de conception, ...), alors que la demande d'architectures performantes est très forte pour des systèmes de traitement de l'information et de communication. Il est par conséquent de plus en plus important de considérer la consommation induite par les optimisations proposées ce qui est encore relativement peu fréquent dans la recherche académique.

Plan du mémoire

Le mémoire est constitué de trois chapitres principaux et d'une annexe. Ces trois chapitres reprennent les trois points développés précédemment : plus de réalisme dans les modèles (chapitre 2), les applications (chapitre 3) et les métriques (chapitre 4).

Le chapitre 2 concerne les travaux de recherche sur les techniques de préchargement visant à réduire les temps d'accès aux données. Ils ont été réalisés essentiellement de 1994 à 1997. Le but de ces travaux n'est pas d'évaluer l'efficacité des techniques de préchargement en terme de prédiction, thème largement traité dans la littérature, mais de mettre en évidence les perturbations du fonctionnement normal du processeur engendrées par les techniques de préchargement et de proposer des implémentations du préchargement qui permettent d'en limiter l'impact et d'exhiber de bonnes performances. Ces perturbations ne peuvent être mises en évidence qu'en intégrant une modélisation plus précise des mécanismes de préchargement. Nous avons proposé plusieurs stratégies permettant de limiter les perturbations induites par les techniques de préchargement, et nous avons proposé le préchargement sur le second niveau de cache qui permet d'éliminer la plupart des défauts du préchargement habituellement implémenté sur le premier niveau de cache. Ces stratégies permettent d'améliorer les performances des processeurs en améliorant l'efficacité du préchargement.

Dans le chapitre 3, nous nous sommes focalisés sur l'étude et l'optimisation des applications multimédia. Les principales caractéristiques de ces applications multimédia sont le traitement d'une grande quantité de données, des calculs identiques sur un ensemble important de données, le traitement par flux des données et le traitement de tâches indépendantes. Ces applications se prêtent particulièrement bien à l'exploitation du parallélisme d'instructions, du parallélisme de données et du parallélisme de flots. Dans ce chapitre, nous avons exploité ces différentes formes de parallélisme afin d'améliorer les performances des applications multimédia. Nous avons exploité les extensions multimédia rajoutées aux jeux d'instructions des processeurs généralistes permettant d'exploiter du parallélisme de données. Nous avons également exploité le support d'exécution multiflot simultané et la possibilité d'utiliser en parallèle des composants standards, ces deux approches exploitant le parallélisme de flots. Dans ces travaux, nous avons proposé des mécanismes permettant une meilleure adaptation des applications multimédia à ces mécanismes émergents et avons particulièrement insisté sur les problèmes liés à la hiérarchie mémoire, tous ces mécanismes induisant une forte pression sur le bus mémoire.

Le chapitre 4 intègre la mesure de la consommation d'énergie dans les évaluations de performance afin de proposer des architectures performantes, mais n'induisant pas de surcoût de consommation. Dans ce cadre, nous avons considéré des applications multimédia et nous avons évalué l'impact du parallélisme de données sur la consommation d'énergie et la possibilité de transférer les apports en vitesse d'exécution en apports énergétiques.

Le chapitre 5 conclut ce mémoire en y présentant mes perspectives de recherche.

Enfin, en annexe sont présentés la méthodologie et les outils utilisés pour ces différents travaux de recherche.

CHAPITRE 2

Préchargement de données : importance de l'implémentation

La conjecture suivante est bien connue en microarchitecture et explique le nombre de travaux sur la hiérarchie mémoire : *le temps de cycle des processeurs décroissant plus rapidement que les temps d'accès à la mémoire principale, la performance effective du processeur est de plus en plus dépendante du comportement de sa hiérarchie mémoire.*

Par conséquent, pour augmenter la performance et donc le rendement du processeur, il faut tenter de combler cette différence notamment en masquant les latences d'accès à la mémoire. De nombreuses méthodes [43] permettent de masquer ces latences mémoire, notamment les caches non bloquants et l'exécution dans le désordre. Une autre méthode intéressante est le préchargement.

Définition. Le préchargement consiste à initier des chargements de la mémoire avant la demande du processeur en espérant que les informations chargées de manière anticipée soient plus tard effectivement demandées par le processeur. Le préchargement peut s'appliquer aux données et aux instructions d'un programme.

Dans ce travail, nous nous sommes intéressés uniquement aux préchargements des données et non d'instructions ; pour les instructions, l'anticipation dépend essentiellement de la prédiction de branchement.

Le but de nos travaux n'a pas été d'évaluer l'efficacité des techniques de préchargement en terme de qualité de la prédiction, thème largement traité dans la littérature, mais de mettre d'abord en évidence les perturbations du fonctionnement normal du processeur engendrées par les techniques de préchargement. Et ensuite, de proposer des implémentations du préchargement qui permettent de limiter l'impact de ces perturbations et d'exhiber alors de bonne performance pour le préchargement. Ces perturbations ne peuvent être mises en évidence qu'en intégrant une modélisation plus précise des mécanismes de préchargement et en utilisant des mesures prenant en compte l'évaluation complète de ces mécanismes notamment en terme de temps et pas uniquement en terme de succès dans la prédiction.

Dans la section 2.1, nous décrivons brièvement les techniques de préchargement proposées dans la littérature et introduisons le problème d'évaluation de ces techniques. Ensuite, dans la section 2.2, nous présentons une analyse détaillée des perturbations engendrées par le préchargement : pollution du cache, pression sur la bande passante du cache de premier niveau, augmentation et modification du trafic mémoire. Dans les sections suivantes, nous présentons nos travaux de recherche relatifs à l'amélioration des techniques de préchargement.

Dans la section 2.3, nous proposons diverses stratégies qui permettent de limiter certaines perturbations des techniques de préchargement et donc d'améliorer l'efficacité du préchargement. Ensuite, nous proposons dans la section 2.4 le préchargement sur le second niveau de cache qui, associé à un contrôle sur le bus mémoire, permet de limiter la plupart des perturbations engendrées par le préchargement proposé habituellement sur le premier niveau de cache et qui est une solution de préchargement performante.

2.1 Préchargement : principe et fonctionnement

2.1.1 Hiérarchie mémoire : un premier algorithme de prédiction

Des hiérarchies mémoire complexes incluant plusieurs niveaux de caches ont été introduites afin de masquer la latence d'accès aux données et aux instructions. L'utilisation de ces caches est basée sur les propriétés de localités spatiale et temporelle des instructions et des données [24, 43].

Définition. Une donnée référencée exhibe de la *localité spatiale* si une donnée située à une adresse voisine est référencée dans un futur proche.

Définition. Une donnée référencée exhibe de la *localité temporelle* si elle est de nouveau référencée dans un futur proche. On parle également de réutilisation.

Les programmes ont en général des propriétés de localités spatiale et temporelle. Pour les instructions, la localité spatiale est due à la séquentialité des codes et la localité temporelle aux branchements (relativement nombreux) en arrière, d'où une réutilisation des mêmes instructions. La localité temporelle des instructions est associée au principe selon lequel un programme utilise 90% de son temps d'exécution sur seulement 10% des instructions [43].

Pour les données, la localité spatiale est due au stockage contigu des données (variables) d'un programme et est particulièrement présente dans les codes numériques et les codes multimédia (en raison des structures de données utilisées : matrices ou vecteurs) et la localité temporelle est due aux réutilisations, accumulations de données.

Le point commun entre toutes les techniques exploitant la localité des références mémoire est l'utilisation de la prédiction mise en oeuvre au travers d'algorithmes plus ou moins sophistiqués. L'algorithme le plus simple consiste à prévoir que chaque donnée exhibe de la localité spatiale et temporelle dans les mêmes proportions. Cet algorithme est mis en œuvre dans les caches où l'on charge à chaque échec une ligne de cache (plusieurs références contiguës) sans savoir si les données chargées exhibent de la localité spatiale. De même, la ligne chargée est stockée dans le cache de données sans savoir si ces données exhibent de la localité temporelle.

2.1.2 Mécanismes de préchargement de données

L'autre manière d'exploiter la localité spatiale des données et ainsi de masquer les latences mémoire est le préchargement, technique basée sur la prédiction des accès mémoire. Le préchargement peut être matériel ou logiciel. Le préchargement matériel consiste à précharger les données dynamiquement lors de l'exécution du programme (décision de préchargement basée sur une analyse simple du comportement du programme), tandis que le préchargement logiciel consiste en une analyse statique du programme afin d'insérer des instructions de préchargement à la compilation. Le préchargement peut également être hybride, i.e., matériel et logiciel.

Un mécanisme de préchargement doit être capable de déterminer quelle donnée sera bientôt référencée et doit être capable d'initier une requête de préchargement suffisament tôt pour masquer la latence mémoire.

Différentes techniques matérielles ont été proposées dans la littérature afin d'exploiter la localité des données et certaines techniques sont mises en oeuvre dans des processeurs récents (exemples : Pentium 4, Athlon, Power4, ...). Pour décrire les techniques de préchargement, Smith [79] a proposé de répondre à trois questions : 1) quand un préchargement est-il initié ? 2) quelles données sont préchargées ? 3) où sont rangées les données préchargées ? Cette description peut nous permettre d'introduire les principales techniques de

préchargement proposées, mais ne suffit pas à définir parfaitement un mécanisme de préchargement notamment son interaction avec les autres composants du processeur (cache de premier niveau, bus mémoire, ...) comme nous le verrons dans le section suivante.

Les premiers algorithmes de prédiction de chargement ont été introduits par Smith [79]. Leur description est simple. Lorsqu'une ligne est référencée seule la ligne suivante est succeptible d'être préchargée selon une certaine stratégie : préchargement systématique (dès qu'une ligne est référencée), préchargement lors d'un échec dans le cache et, préchargement lors d'un échec dans le cache et lorsqu'une donnée préchargée est référencée (préchargement marqué, *tagged prefetch*). Les données préchargées peuvent être stockées dans le cache ou dans un tampon spécifique.

Les tampons de flux proposés par Jouppi [47] permettent de précharger non plus une seule ligne, mais plusieurs lignes de données consécutivement à partir de l'adresse de la ligne ayant provoqué un défaut de cache. Les données préchargées sont placées dans un tampon et ne sont copiées dans le cache que lorsqu'elles sont accédées. Palacharla *et al.* [65] ont étudié les tampons de flux en remplacement du cache de second niveau. Les techniques décrites précédemment sont appelées techniques de préchargement séquentiel.

Des techniques plus complexes [44, 3, 18, 45] ont été proposées pour précharger les données et sont basées sur la détection d'un pas constant sur le patron d'accès aux données (références spatiales, mais non forcément séquentielles). Cette détection est réalisée par comparaison des chargements/rangements successifs dans une table spécifique, appelée table de prédiction des références (RPT : *Reference Prediction Table*). D'autres travaux [38] ont proposé une extension de la table de références afin de permettre la détection de pas dans des références indirectes (pointeurs). Enfin, des méthodes de prédiction plus complexes basées sur un modèle de Markov [46] ont été proposées pour prédire les références mémoire. Toutes ces techniques qui visent à déterminer un patron des références peuvent utiliser comme espace de stockage le cache ou une autre structure.

Un certain nombre de techniques logicielles sont également mises en œuvre. Elles consistent en l'ajout d'instructions de préchargement destinées à anticiper les besoins du processeur. Les patrons des accès aux données sont détectés lors d'une analyse statique du programme, puis des instructions de préchargement sont insérées plusieurs cycles avant leur instruction mémoire correspondante. Des instructions de préchargement existent dans la plupart des processeurs [6, 70, 101]. Le préchargement logiciel porte l'effort sur l'écriture des compilateurs qui doivent être capables d'utiliser efficacement ces instructions de préchargement. La plupart des préchargements proposés se limitent au cas des nids de boucles avec des limites de boucles connues statiquement [97, 9, 32, 59] (généralement des éléments de tableaux). Par ailleurs, le compilateur doit déterminer statiquement la distance de préchargement ce qui n'est pas aisé du fait du comportement non déterministe de certains composants de l'architecture, notamment les caches. Un préchargement survenant trop tôt peut écraser des données encore utiles au processeur ou lui-même être écrasé avant son utilisation, un préchargement survenant trop tard engendrera des pénalités (latence partiellement recouverte).

Des techniques de préchargement combinant préchargements matériel et logiciel ont également été proposées ; la principale idée est d'assister le matériel par des informations obtenues statiquement [19, 36], ...

Nous ne détaillons pas tous les mécanismes de préchargement proposés dans la littérature, l'objet de ce travail n'étant pas de proposer une nouvelle technique de préchargement, ni de comparer l'efficacité de différentes techniques. Des descriptions récentes plus détaillées peuvent être trouvées dans [86, 85].

2.1.3 Performance des techniques de préchargement

Pour évaluer les performances des techniques de préchargement, la mesure de la réduction du nombre d'échecs sur le cache ou la mesure de la quantité de préchargements efficaces sont souvent utilisées. Ces mesures donnent l'efficacité absolue de la prédiction, mais ne permettent pas d'évaluer précisément l'impact des techniques de préchargement sur les performances notamment dans des environnements superscalaires, avec cache non bloquant et exécution dans le désordre. Ainsi, considérer uniquement la réduction du nombre d'échecs sur le cache peut conduire à des résultats trop optimistes (on constate même des cas extrêmes [57] où l'échec d'une donnée sur le cache génère de meilleure performance que le succès de cette même donnée).

Mais de même que la réduction du taux d'échecs n'est pas une mesure fiable des performances du préchargement, ne pas tenir compte de l'interaction du mécanisme de préchargement avec les différents composants du processeur concernés peut fausser partiellement les résultats. En effet, le préchargement interfère avec les composants de base du processeur en occupant le cache, le bus mémoire, les bancs mémoire, etc. Beaucoup d'études sur le préchargement ne tiennent pas compte des détails d'implémentation des mécanismes proposés et de leur intéraction avec les autres composants du processeur, négligeant ainsi un certain nombre d'effets temporels (exemple : une requête de préchargement peut ralentir une requête normale parce qu'elle a monopolisé une ressource (cache, bus), ...) nécessaire à la requête normale).

2.2 Considérer les perturbations dues au préchargement

Dans ce travail, nous nous sommes focalisés sur les perturbations occasionnées par les requêtes de préchargement sur les requêtes normales du processeur [1] afin d'en limiter l'impact et de rendre plus performant le préchargement.

Nous présentons dans cette section les principales perturbations engendrées par les techniques classiques de préchargement : perturbations du cache de premier niveau (pollution, accès) et perturbations du trafic mémoire[2] et nous présentons dans les sections suivantes, les solutions que nous avons proposées pour limiter ces perturbations.

2.2.1 Perturbations dues au stockage des données préchargées

Les données préchargées peuvent être stockées soit directement dans le cache de données, soit dans une autre structure.

Les techniques utilisant le cache pour stocker les données préchargées peuvent polluer le cache en générant des conflits entre les données préchargées et les données encore utilisées par le processeur. Cette pollution est d'autant plus pénalisante que la prédiction est mauvaise. Une donnée préchargée peut écraser une donnée utile au processeur et générer un échec qui ne se serait pas produit sans préchargement.

Les techniques utilisant un tampon pour stocker les données préchargées se heurtent notamment à des problèmes de cohérence de données. Détaillons ces problèmes en considérant le code suivant et l'architecture de la figure 2.1 et un préchargement séquentiel (préchargement sur défaut).

[1] On appelle requête normale du processeur les requêtes mémoire spécifiées dans les instructions de chargement/rangement mémoire du programme exécuté.

[2] Nous avons fait le choix de travailler à politique de préchargement fixée et d'évaluer uniquement les variations de performance engendrées par les perturbations du préchargement sur les requêtes normales du processeur.

Lecture de la donnée située à l'adresse	A - Ls	:	échec.
Ecriture de la donnée située à l'adresse	A	:	succès.
Lecture de la donnée située à l'adresse	B	:	échec.

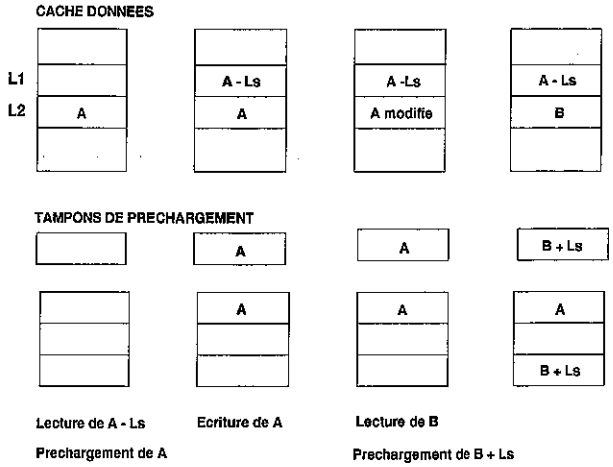

FIG. 2.1: Cohérence dans des tampons de préchargement à 1 ligne et à X lignes.

Le processeur demande la donnée située à l'adresse A-Ls (où Ls est la taille de la ligne de cache). Le cache ne contient pas cette donnée ; on a donc un défaut de cache. La donnée située à l'adresse A-Ls est chargée de la mémoire principale dans le cache, ainsi que la donnée située à l'adresse A (technique de préchargement sur défaut). Dans le cas de tampons de préchargement à une ligne et à X lignes (X = 3 dans la figure 2.1), la donnée située à l'adresse A est copiée dans le tampon. Ensuite, il faut modifier (pour réaliser l'écriture) la donnée située à l'adresse A. Cette donnée appartient déjà au cache de données et est donc modifiée dans ce cache. On constate qu'à cet instant, le tampon de préchargement et le cache ne sont pas cohérents (valeurs différentes de A dans les deux espaces de stockage). Puis la donnée B est lue. Elle provoque un défaut sur le cache de données et est donc chargée de la mémoire principale dans le cache. La donnée située à l'adresse B+Ls est préchargée dans le tampon de préchargement.

Dans le cas d'un tampon de préchargement à une ligne, la donnée située à l'adresse B+Ls écrase celle située à l'adresse A contrairement au cas d'un tampon à plusieurs lignes de cache. Ainsi la gestion de la cohérence est automatiquement préservée dans le cas d'un tampon à une seule ligne de cache, mais pas dans l'autre cas.

L'utilisation d'un tampon de préchargement engendre d'autres problèmes que les problèmes de cohérence, elle peut notamment augmenter le temps de cycle du processeur si le tampon se situe sur un chemin critique (c'est à dire si l'accès au tampon augmente le temps de cycle du processeur).

2.2.2 Perturbations dues aux accès supplémentaires au cache

Un programme comprend en moyenne 30% d'instructions d'accès à la mémoire. Si l'on dispose d'un cache à un seul port, il est impossible d'excéder 3,3 instructions exécutées par cycle [98] (même avec 100% de succès et un nombre illimité d'unités fonctionnelles). D'autres études ont donné le nombre de ports optimal compte tenu du dégré de superscalaire considéré (exemple : d'après [48] avec un superscalaire de degré 4, le nombre optimal de ports est 2). Ainsi, étant donnée la bande passante mémoire nécessaire sur les processeurs

superscalaires, les caches ont été modifiés afin de supporter deux requêtes par cycle, en général (exemples : utilisation de cache multi-bancs, e.g., Intel Pentium, AMD K7, MIPS R10000 ou en duplicant le premier niveau de cache, e.g. Alpha 21264).

Par ailleurs, certaines techniques visant à réduire la latence mémoire, notamment le préchargement, augmentent le nombre d'accès au cache en générant des accès supplémentaires. En ce qui concerne le préchargement, les accès supplémentaires sont nécessaires afin de :

- tester si la donnée à précharger appartient ou non au cache (avant d'envoyer la requête de préchargement aux niveaux mémoire supérieurs),
- copier la donnée préchargée dans le cache (après son chargement des niveaux mémoire supérieurs),
- tester la cohérence entre le cache et un espace de stockage annexe pour les requêtes de préchargement (par exemple, en cas d'écriture avec succès sur le cache), etc.

Beaucoup d'études réalisées sur les techniques de préchargement ignorent l'impact des accès supplémentaires au cache qui peuvent avoir un coût non négligeable notamment sur les processeurs superscalaires pour lesquels la fréquence d'accès au cache est élevée.

2.2.3 Perturbations dues à la modification du trafic mémoire

Le préchargement augmente le trafic entre la mémoire principale et le cache lorsque la prédiction n'est pas parfaite, en augmentant le volume de données chargées de la mémoire par rapport au volume de données réellement nécessaires. Il est donc particulièrement limitatif lorsque le trafic mémoire est important. D'autres phénomènes dus au préchargement perturbent le trafic mémoire. En effet, précharger des données à certain moment perturbe le chargement normal des données ; les défauts de cache semblent se réaliser par groupe [93] (les défauts de cache arrivent souvent en même temps suivis par une plus longue période avec peu de défauts de cache). Pendant ces rafales de défauts, augmenter le trafic mémoire peut être particulièrement pénalisant. Par ailleurs, le préchargement peut conduire à charger de la mémoire un ensemble de données dans un intervalle de temps plus court augmentant localement le trafic mémoire.

Ding *et al.* [27] ont montré que la bande passante mémoire est un sérieux goulet d'étranglement pour les processeurs superscalaires actuels et ont proposé des techniques logicielles permettant de réduire le nombre de transferts mémoire d'un programme. De plus, D. Burger et *al.* [15] ont montré que les techniques proposées pour réduire la latence mémoire augmentaient le nombre de cycles d'attente à cause de la limitation de la bande passante. Dans la même idée, Seznec et *al.* [78] ont montré que le cache de second niveau (L2) est une ressource très utilisée, car la taille des caches de premier niveau (L1) induit un taux d'échecs L1 relativement élevé et qu'il est alors important de réduire la pression sur le cache L2, ce qui n'est pas forcément compatible avec le préchargement.

Ainsi, le fonctionnement des bus de communication entre les différents niveaux mémoire est perturbé par les techniques de préchargement. Les requêtes normales du processeur peuvent être décalées à cause de requêtes de préchargement, phénomène bien connu, mais que seule une implémentation détaillée peut permettre de quantifier précisément.

Illustrons ainsi avec un exemple simple comment la performance peut être mal évaluée lorsque le modèle de bus entre le cache L1 et le cache L2 (hors de la puce du processeur) est trop optimiste et comment le préchargement peut affecter les requêtes normales du processeur. Considérons l'exécution des instructions suivantes sur un processeur superscalaire (les adresses A et B sont telles que $B \neq A$ et $\neq A + L$) :

Lecture de la donnée (D1) située à l'adresse A : échec.
Lecture de la donnée (D2) située à l'adresse B : échec.

Nous supposons que les deux instructions de chargement font un échec sur le cache de données. Après chaque échec, nous supposons qu'une requête de préchargement est envoyée à la mémoire (nous n'étudions que la première requête de préchargement (P1) induite par le premier échec de donnée (D1)). Considérons maintenant deux modèles de bus :

Modélisation réaliste du bus Cette modélisation consiste à considérer l'occupation du bus adresses et du bus données et donc à intégrer les latences supplémentaires induites par d'éventuelles contentions sur le bus (les données circulant sur le bus par paquets, une donnée ne peut circuler si le bus est déjà occupé et devra donc être retardée le temps que le bus se libère). Si l'on considère une taille de ligne pour le cache L1 de N octets et une largeur de bus de données de n bits, il faudra faire circuler sur le bus $\frac{N}{n}$ paquets de n bits pour charger une ligne complète de cache. L'instant où le défaut de cache L1 est déterminé et l'instant où la ligne de cache est complètement chargée du cache L2 vers le cache L1 dépend : du temps nécessaire pour gérer la demande sur le bus et du temps de transfert de l'adresse sur le bus, du temps d'accès au cache L2 et du temps de retour des données sur le bus données. Pour illustrer le comportement du bus ainsi défini, nous considérons les valeurs numériques suivantes : taille de la ligne de cache L1 N = 32 octets, largeur du bus de données n = 64 bits, fréquence du bus à la moitié de la fréquence du processeur, temps pour gérer l'adresse de 2 cycles bus et temps d'accès au cache L2 (L) de 2 cycles processeur (cette valeur est volontairement petite afin de visualiser sur une même figure le déroulement de notre exemple). Ainsi comme le montre la figure 2.2, les premiers 64 bits arrivent 4 cycles après que l'adresse ait été reçue par le cache L2, ensuite, les 3×64 bits arrivent tous les 2 cycles.

Un modèle optimiste de bus Dans un certain nombre d'études, seule la latence du cache L2 (L) est considérée, mais l'occupation du bus et les conflits potentiels ne sont pas modélisés. Lorsqu'une adresse est envoyée à la date t, la donnée est supposée revenir à la date $t + L$. Parmi ces études, certaines intègrent dans la valeur de la latence la gestion de l'adresse et le temps de transfert de la donnée sur le bus. Dans ce cas, la latence totale pour récupérer une donnée est, en considérant l'exemple numérique précédent, $2+2+2 = 6$ cycles. L'occupation temporelle du bus est généralement ignorée et les requêtes mémoire sont envoyées/récupérées à chaque cycle.

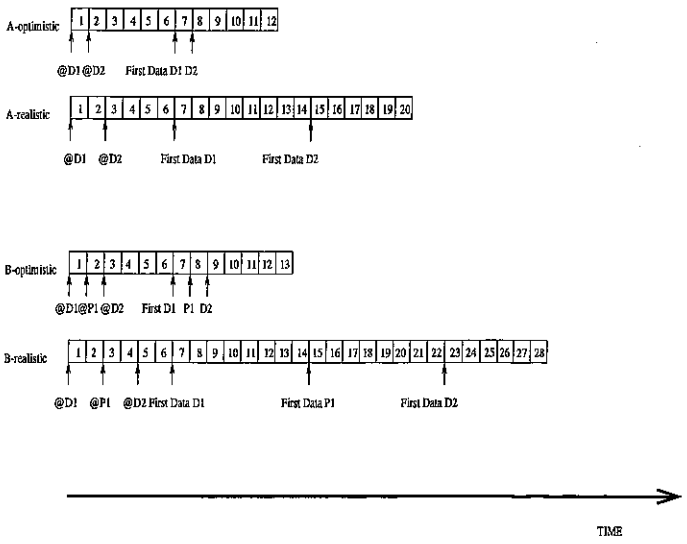

FIG. 2.2: Impact du modèle de bus choisi et du préchargement sur les requêtes normales.

Maintenant détaillons le comportement des requêtes normales et des requêtes de préchargement associées sur les modèles de bus précédemment décrits :

Modèle réaliste Considérons qu'une requête faisant un échec n'induise pas de requête de préchargement (cas A). Comme montré sur la figure 2.2, les adresses D1 et D2 sont envoyées en séquence aux cycles 1 et 3. Les premiers 64 bits de D1 sont reçus au cycle 6 et les 32 octets de D1 sont complètement reçus au cycle 12. Au début du cycle 10, le cache L2 a terminé la requête D1 et le chargement de la requête D2 commence. Les 64 premiers bits sont envoyés sur le bus au cycle 12 et arrivent au processeur au cycle 14. La requête D2 est terminée au cycle 20.

Considérons maintenant qu'une requête qui fait un échec soit suivie par une requête de préchargement (cas B). L'adresse de P1 est envoyée après D1, et l'envoi de l'adresse D2 est alors décalée et envoyée au cycle 5 et ainsi terminée au cycle 28.

Modèle optimiste La requête D2 est terminée au cycle 10 dans le cas A et au cycle 11 dans le cas B. Par conséquent, l'impact du préchargement sur la latence d'échecs du cache L1 est fortement sous-estimé.

Ce simple exemple montre que la latence des accès normaux du processeur peut être augmentée du fait de requêtes de préchargement et que les performances peuvent varier de façon significative en fonction du modèle de bus utilisé. Ce phénomène est particulièrement vrai entre le cache L1 et le cache L2 où il y a beaucoup de communications, mais intervient également sur le bus mémoire.

Dans ce travail, nous n'avons considéré que les perturbations liées aux transferts des données sur les bus entre les différents niveaux mémoire, mais d'autres paramètres ont un impact sur les performances du système mémoire [22], notamment les caractéristiques de la mémoire (exemple : nombre de bancs de la mémoire) et le fonctionnement de la mémoire (exemple : tous les accès à la mémoire n'ont pas la même latence).

2.3 Limiter les perturbations liées au préchargement

Pour limiter les perturbations engendrées par les requêtes de préchargement et ainsi exhiber de meilleures performances, nous avons étudié différentes mises en oeuvre du préchargement.

Les deux premières perturbations introduites précédemment (stockage dans le cache versus tampon de préchargement, accès au cache) seront étudiées dans le cadre du préchargement L1, la dernière perturbation (trafic mémoire) dans le cadre du préchargement L2. Cet ordre correspond à la chronologie des travaux.

2.3.1 Réduire la pollution du cache

Les propriétés de localités spatiale et temporelle des données sont différentes selon les codes exécutés. Par exemple sur les codes numériques, les accès aux différents tableaux exhibent essentiellement des propriétés de localité spatiale ce qui entraîne de nombreux échecs de démarrage et une occupation du cache inutile (pollution potentielle). Malheureusement, l'organisation des caches n'est pas flexible quand à l'exploitation des localités spatiale et temporelle. Quelle que soit la localité spatiale, la même taille de ligne de cache est utilisée. Quelle que soit la localité temporelle, une donnée est toujours chargée dans le cache qu'elle soit ou non réutilisée. Pourtant, il est possible de déterminer statiquement (à la compilation) pour certains algorithmes (par exemple, les nids de boucle) si une référence exhibe de localité spatiale et/ou temporelle [97]. Par conséquent, le but de ce travail a été de proposer une organisation de cache utilisant de simples informations statiques pour mieux exploiter la localité spatiale tout en limitant la pollution dans le cache. Les mécanismes matériels utilisés consistent à précharger simplement les données via une ligne de cache plus large et d'éviter la pollution du cache en court-circuitant le stockage dans le cache pour les données n'exhibant pas de localité

temporelle, ces deux mécanismes étant couplés aux informations statiques sur les propriétés de localité des références extraites à la compilation.

Exploiter la localité spatiale en limitant la pollution du cache La plus simple façon d'exploiter la localité spatiale est d'utiliser un cache avec une taille de ligne large. Cette solution pose notamment deux problèmes : un cache performant doit exploiter correctement à la fois la localité spatiale et la localité temporelle et pour exploiter la localité temporelle, il est nécessaire d'avoir un cache possédant beaucoup d'entrées, entrées dont le nombre est réduit lorsque la taille de la ligne est importante (nombre d'entrées = $\frac{Taille\ du\ cache}{Taille\ de\ la\ ligne\ de\ cache}$). Par ailleurs, une ligne large génére un important trafic mémoire.

Pour exploiter cette solution tout en limitant les problèmes précédemment décrits, il est possible d'utiliser le mécanisme de ligne de cache virtuelle [82]. Ce mécanisme consiste à utiliser une petite taille de ligne physique (32 octets comme dans la plupart des caches des processeurs actuels) et de charger une ligne de cache virtuelle large correspondant à n lignes physiques lorsque les références exhibent de la localité spatiale. Seule la ligne de cache contenant la requête processeur est chargée dans le cache L1, les autres lignes (n-1) sont chargées dans un deuxième niveau de cache d'accès rapide (de taille petite et sur la puce du processeur). Ce deuxième niveau de cache est utilisé pour stocker les lignes supplémentaires chargées, mais également comme *victim cache* [47] (un *victim cache* permet de réduire le taux d'échecs du cache L1 en réduisant les conflits sur le cache et ceci sans affecter le temps de cycle processeur). Les performances de cette solution sont relativement bonnes, mais il subsiste quelques problèmes.

Tout d'abord, l'utilisation de la ligne de cache virtuelle ne permet pas de limiter suffisamment les effets de la pollution dans le cache, i.e., l'éviction de données exhibant de la localité temporelle par des données n'exhibant pas de localité temporelle. Le *victim cache* utilisé dans le mécanisme de ligne virtuelle ne permet pas non plus de réduire la pollution du cache qui est un problème de capacité plutôt qu'un problème de conflits. De plus, l'utilisation de la ligne de cache virtuelle peut augmenter sensiblement le trafic lors du traitement de références n'exhibant pas de localité spatiale.

Pour limiter ces effets, on peut court-circuiter le cache pour les données n'exhibant pas de localité temporelle et les stocker dans le *victim cache*. Mais si toutes les références qui n'exhibent pas de localité temporelle court-circutent le cache, leur localité spatiale peut ne pas être exploitée (à cause de la taille limitée du *victim cache*) et les performances peuvent alors être moins bonnes que dans une configuration de cache classique.

Assistance du compilateur Une solution pour limiter les problèmes de pollution et de trafic mémoire est de pouvoir, quand c'est possible, avoir une information sur les propriétés des références mémoire. Si une référence n'exhibe pas de localité spatiale, seule une ligne de cache est chargée et ainsi, le trafic mémoire n'est pas inutilement augmenté. Si une référence n'exhibe pas de la localité temporelle, mais de la localité spatiale, il faut que les références contiguës (appartenant à la même ligne de cache) puissent être stockées jusqu'à leur utilisation.

Nous avons donc proposé de collecter des informations sur les propriétés de localités spatiale et temporelle des tableaux à la compilation et de les transmettre au matériel. Divers travaux [9, 32, 59, 97] ont été menés sur la détermination à la compilation des propriétés de localité des références mémoire. Dans ce travail, nous avons utilisé des techniques élémentaires présentées dans [97] (analyse de la localité temporelle) et dans [59] (analyse de la localité spatiale).

Une référence $A[j+a]$ est marquée *spatiale* avec j incrément de la boucle la plus interne (voir figure 2.3), si a est inférieur à la taille de la ligne de cache. Si le coefficient a est une variable (non connue à la compilation), la référence n'est pas marquée *spatiale*.

```
for (i = 0; i < N; i++) {
  reg = Y(i);
  for (j = 0; i < N; j++) {
    reg = reg + (A(i,j) + A (i,j+1) + A(i,j+a)) * X(j);
  }
  Y(i) = reg;
}
```

FIG. 2.3: Exemple de références pour lesquelles l'information de localité temporelle est extraite à la compilation.

Une référence est marquée *temporelle* si elle est réutilisée dans une même itération (exemple de $X(j)$) ou si elle est réutilisée entre différentes itérations de manière simple (exemple de $A(i,j)$ et $A(i,j+1)$).

Ces types de références sont relativement faciles à extraire et représentent une portion importante des dépendances généralement présentes dans les codes numériques [97].

Pour collecter les adresses manipulées et les étiquettes de localités spatiale et temporelle, nous avons travaillé au niveau du code source. Pour ce faire, nous avons inséré des appels à une procédure avant chaque référence à un tableau dans le code source permettant d'analyser la référence et d'annoter le code avec l'étiquette de localité. Le compilateur Sage++ [34] a été utilisé pour instrumenter toutes les références aux tableaux.

Par ailleurs, il faut stocker pour chaque instruction mémoire les informations de localité spatiale et temporelle (2 bits). Ceci nécessite une modification dans le jeu d'instructions. Nous avons utilisé les bits réservés au codage des déplacements pour stocker ces 2 bits ce qui a pour conséquence de limiter la taille des déplacements pour le calcul de l'adresse et donc de complexifier le calcul des adresses surtout quand elles sont très largement distribuées.

Exploitation des propriétés de localité par le matériel Nous avons donc implémenté le mécanisme de ligne virtuelle associé au *victim cache* proposé dans [82] et modifié le mécanisme afin d'exploiter les propriétés de localité déterminées statiquement et de court-circuiter le cache en fonction (stockage directement dans le *victim cache*).

Dans ce travail, chaque étiquette du cache contient deux bits supplémentaires : un bit de localité spatiale et un bit de localité temporelle stockés lors de l'analyse à la compilation dans les bits de déplacement. Le comportement du *victim cache* est également modifié, il va servir en plus à court-circuiter le cache en exploitant les informations statiques de localité (données marquées uniquement spatiales).

Cependant, s'il y a beaucoup de données marquées temporelles placées dans le cache, il y a des riques que ces lignes polluent le cache et perturbent l'exploitation des bits de localités temporelle et spatiale d'une nouvelle ligne. Pour éviter ce problème, à chaque fois qu'une ligne de cache est ramenée du *victim cache* dans le cache L1, le bit de localité temporelle est remis à 0. Il est réactivé si une instruction de chargement/rangement marquée temporelle accède de nouveau à la ligne. Autrement, le bit de localité temporelle reste inactif et la donnée ne sera pas stockée dans le *victim cache* lors de son éviction du cache.

Performance Les performances de la ligne virtuelle et du court-circuit du cache assisté par le logiciel sont données dans la figure 2.4. La mesure utilisée est le temps moyen d'accès à la mémoire AMAT (*Average memory access time*) : Taux de succès du cache L1 + Taux d'échecs du cache L1 × Pénalité d'échecs du cache L1. Les programmes testés sont des programmes numériques : ADM, MDG, BDN, DYF, ARC, FLO, TRF extraits de la suite *Perfect Club*, boucles de Livermore LIV, NAS et Slalom, et deux primitives numériques, multiplication matrice-vecteur MV et multiplication matrice creuse-vecteur SpMV.

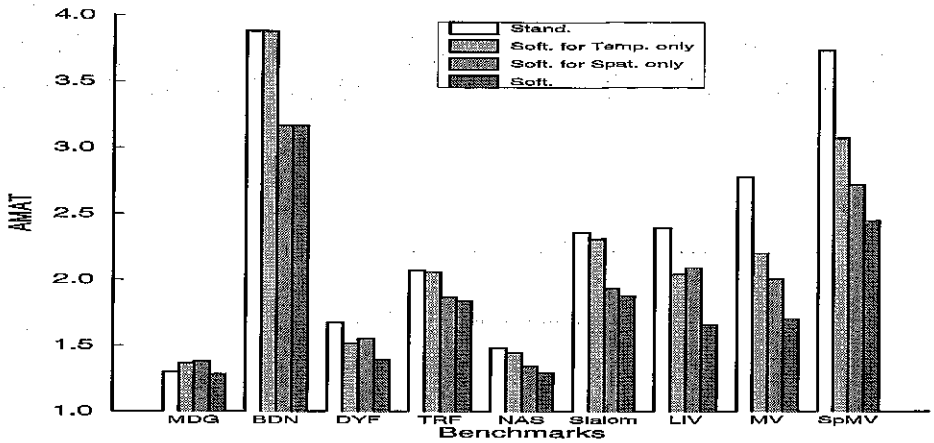

FIG. 2.4: Performance de la ligne virtuelle et du court-circuit du cache assistés par logiciel.

La configuration de base utilisée pour le cache correspond notamment à celle du DEC Alpha et du MIPS R4000 : taille du cache de 8 Koctets, taille de la ligne de cache de 32 octets, associativité de 1 (à correspondance directe). Le *victim cache* est de 64 octets, la taille de ligne virtuelle de 64 octets, la latence mémoire de 20 cycles et enfin, la largeur du bus de 16 octets. Ces différents paramètres correspondent à l'état des architectures en 1994. La taille du cache L1 est limitée compte tenu de la place disponible sur la puce du processeur, l'associativité est également limitée afin de maintenir un temps de cycle faible (pipeline très court).

Si seule l'exploitation de la localité temporelle est activée (voir `Soft. for Temp. only` sur la figure) ou seule l'exploitation de la localité spatiale est activée (voir `Soft. for Spat. only` sur la figure), la performance est très peu améliorée. Mais combiner les deux exploitations (voir `Soft.` sur la figure) permet d'augmenter sensiblement les performances. De plus, cette approche permet de préserver un trafic mémoire proche du trafic mémoire normal (voir figure 2.5). Ceci est du notamment à une utilisation sélective de la ligne de cache virtuelle.

Nous avons également étudié l'association du mécanisme de ligne virtuelle et du préchargement matériel. Tout d'abord, le *victim cache* peut être utilisé comme un tampon de préchargement afin d'éviter de perturber le cache et donc le processeur par les requêtes de préchargement. Ensuite, les informations sur la localité spatiale extraite statiquement peuvent être utilisées pour assister le préchargement. Le mécanisme est mis en oeuvre de la manière suivante : lors d'un échec, une ligne virtuelle est chargée, ainsi que la ligne physique suivante. Cette seconde ligne est marquée (préchargement marqué) et stockée dans le *victim cache*. Lors d'un succès dans le *victim cache* sur la ligne marquée préchargée, le transfert de cette ligne dans le cache et le préchargement de la ligne suivante sont initiés. Ainsi, le préchargement est progressif et les requêtes en rafale sont limitées.

Ce mécanisme exhibe de meilleure performance (voir figure 2.6) notamment parce qu'il permet de réduire les latences dues aux échecs de démarrage et de capacité dans les vecteurs.

Références avec résultats détaillés : [80, 81].

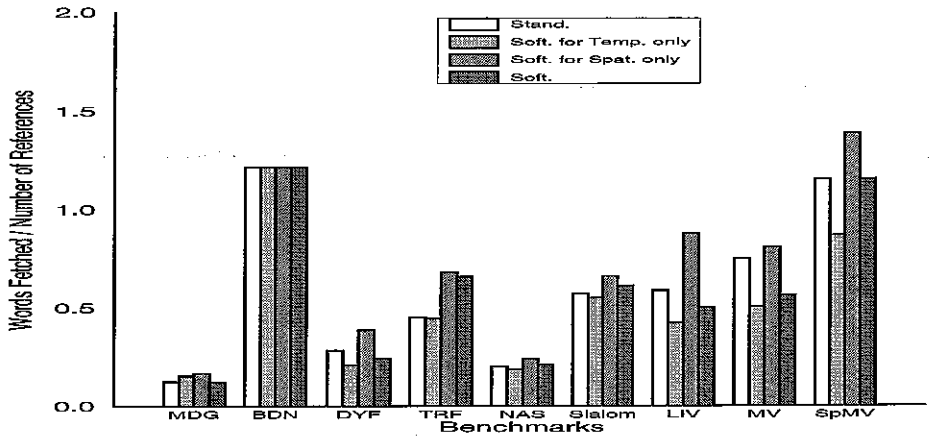

FIG. 2.5: Impact sur le trafic mémoire.

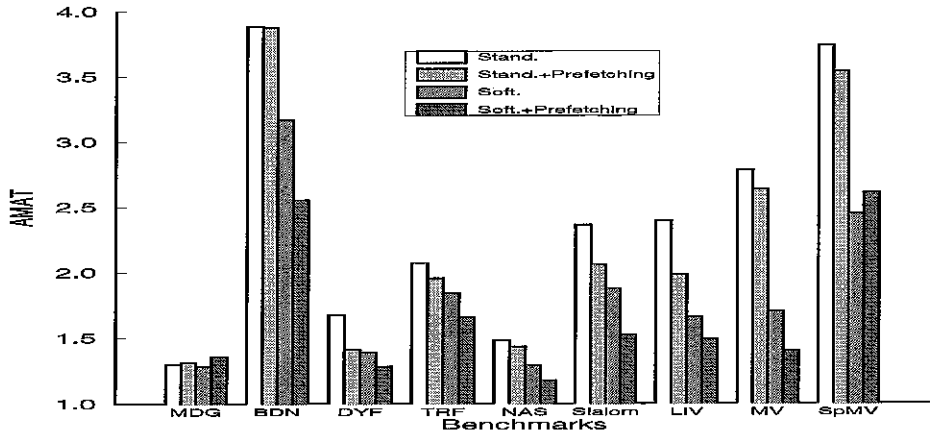

FIG. 2.6: Performance du préchargement assisté par logiciel.

2.3.2 Réduire les accès au cache

Les requêtes de préchargement peuvent pertuber l'accès au cache pour les requêtes normales en occupant le ou les ports d'accès au cache à des moments inopportuns. Pour limiter ces accès, une solution simple est d'utiliser une structure de stockage spécifique, par exemple un tampon, pour les données préchargées. Mais l'utilisation d'un tampon de préchargement afin de réduire les accès au cache (notamment les accès d'écriture) et les accès de tests (appartenance, cohérence) pose des problèmes de cohérence des données (cohérence entre le cache et le tampon de préchargement) comme nous l'avons décrit dans la section précédente. L'utilisation d'un tampon à une ligne permet de résoudre automatiquement ces problèmes de cohérence, mais les performances sont nettement moins bonnes que celles obtenues avec un tampon à plusieurs lignes. Nous avons donc proposé l'intégration d'un tampon de préchargement à plusieurs lignes qui permette de maintenir la cohérence des données et qui limite les pénalités engendrées par la gestion de ce tampon. Dans cette étude, nous considérons le préchargement marqué[3].

Comparaison des performances des tampons de préchargement à 1 ligne et à X lignes Nous avons comparé les performances d'un tampon de préchargement à 1 ligne avec un tampon à X lignes (en faisant varier X). La mesure utilisée est le pourcentage de réduction d'échecs donné par :

$$\%\ \text{réduction d'échecs} = \frac{\text{nombre de succès sur le tampon}}{\text{nombre d'échecs sur le cache}} \times 100.$$

Les simulations montrent qu'avec une technique de préchargement séquentiel (préchargement marqué), le taux d'échecs avec un tampon de 8 lignes peut être 2 à 3 fois moins important qu'avec un tampon de 1 ligne (nous avons choisi une taille de tampon de 8, car le gain en performance moyen obtenu en augmentant la taille du tampon tend à se réduire, l'objectif étant par ailleurs d'introduire une structure de taille limitée). Les simulations ont été réalisées sur des programmes extraits de la suite SPEC92 (3 programmes entiers : eqntott, compress, li et 5 programmes flottants : mdljdp2, wave5, tomcatv, alvinn, swm256) et pour différentes caractéristiques de cache. Nous présentons dans la figure 2.7 les résultats obtenus pour un cache de 16 Koctets, associatif par ensemble de 2 blocs et une taille de ligne de 32 octets.

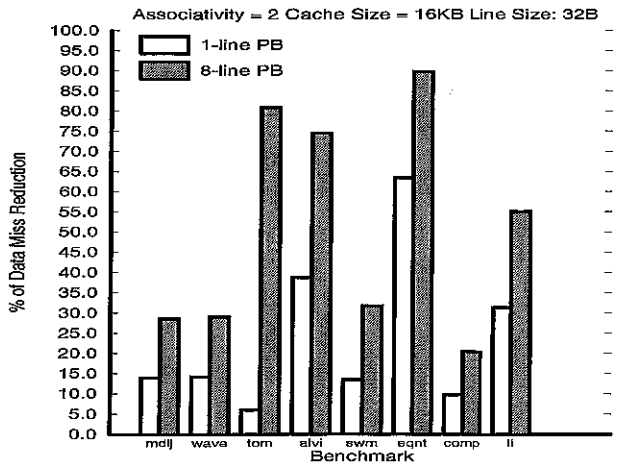

FIG. 2.7: Réduction du taux d'échecs avec des tampons de préchargement de 1 ligne et de 8 lignes.

[3] Comme nous l'avons dit au début de ce chapitre, nous ne nous sommes pas intéressés au algorithme de préchargement, mais aux perturbations engendrées par les requêtes de préchargement. L'algorithme de préchargement choisi est volontairement simple, mais a priori le plus performant des algorithmes séquentiels.

Gérer la cohérence sur les tampons de préchargement à plusieurs lignes Les performances d'un tampon à plusieurs lignes étant très largement supérieures aux performances d'un tampon à une ligne, il semble judicieux d'essayer de mettre en place une stratégie de cohérence indépendante du chemin critique pour les tampons de préchargement à plusieurs lignes. La stratégie de cohérence proposée dépend de la politique d'écriture des caches.

Politique d'écriture simultanée A la suite d'une requête de préchargement, une ligne dans le tampon de préchargement est allouée. Un bit supplémentaire est nécessaire pour spécifier que la ligne est allouée, mais non disponible. Ce bit est mis à 0 lorsque le contenu de la ligne allouée est chargé. Considérons le cas où la requête traitée n'appartient pas au cache, mais appartient au tampon de préchargement. Si cette requête est une lecture, la ligne correspondante est simplement copiée dans le cache. La double copie de la donnée (tampon de préchargement et cache) peut être utile pour minimiser les phénomènes de *ping-pong* [47]. Si cette requête est une écriture, la ligne est copiée dans le cache et mise à jour, la ligne correspondante est invalidée dans le tampon de préchargement. Cette stratégie diffère quelque peu de la politique d'écriture non allouée souvent combinée avec la politique d'écriture différée. Une autre stratégie possible est de mettre à jour la donnée présente dans le tampon de préchargement. Si la ligne est allouée, mais que la donnée est toujours en cours de chargement dans le tampon de préchargement, que ce soit pour une requête en lecture ou une requête en écriture, la demande est inhibée jusqu'au retour de la donnée.

Politique d'écriture différée Pour cette politique, on considère le même traitement que pour la politique d'écriture simultanée précédemment décrite. Par contre, il faut également considérer le problème de cohérence entre le cache et le tampon de préchargement. Lorsque l'on écrase une donnée modifiée dans le cache et que cette donnée appartient également au tampon de préchargement, on invalide la ligne correspondante dans le tampon de préchargement. S'il existe un tampon d'écriture, un autre test est nécessaire. En effet, il est possible qu'un mot modifié à l'adresse A soit stocké dans le tampon d'écriture avant que la donnée située à l'adresse A n'ait été préchargée. Une solution est de tester simplement chaque requête de préchargement avec le contenu du tampon d'écriture avant qu'elle soit envoyée à la mémoire.

Nous avons vu dans cette partie comment concevoir un tampon de préchargement à plusieurs lignes et maintenir pour toutes les politiques d'écriture la cohérence entre le cache et le tampon de préchargement.

Recouvrir les latences d'accès au tampon de préchargement Le couple cache et tampon de préchargement fonctionne comme une hiérarchie mémoire. Dans un premier cycle, le cache est accédé. Si la donnée n'appartient pas au cache, le tampon est accédé dans un deuxième cycle, puis s'il y a succès dans le tampon, la donnée est copiée dans le cache lors d'un cycle suivant. Par conséquent, un échec sur le cache coûte 2 cycles de pénalité supplémentaire. Pour masquer ces cycles, nous avons proposé d'ajouter au pipeline un étage supplémentaire dédié à l'accès au tampon de préchargement afin de recouvrir les cycles d'accès à ce tampon. Les deux politiques d'écriture diffèrent essentiellement par le fait que le cache et le tampon de préchargement sont toujours cohérents dans la politique d'écriture simultanée, alors que ce n'est pas le cas dans la politique d'écriture différée et requièrent donc deux implémentations différentes :

Politique d'écriture simultanée L'étage supplémentaire appelé PB est rajouté avant l'étage d'accès à la mémoire pour les données (pipeline RISC classique, voir figure 2.8). Lors d'un accès mémoire, le tampon de préchargement est accédé pendant l'étage PB. Si c'est un succès, la donnée est copiée dans le cache durant l'étage MEM (avec une écriture simultanée, le tampon de préchargement et le cache sont cohérents). La solution de placer l'étage MEM après l'étage PB est utilisée pour masquer les

opérations de copie dans le cache. Normalement, deux accès au cache sont nécessaires lors d'un succès sur le tampon de préchargement (un accès pour tester la présence de la donnée et un accès pour copier la donnée du tampon vers le cache), mais s'il y a un succès dans le tampon de préchargement (à l'étage PB), alors il n'est pas nécessaire d'accéder au cache pour le test et le tampon de préchargement peut envoyer la donnée au processeur.

Mais insérer l'étage PB dans le pipeline augmente le nombre de délais de chargement. Considérons la séquence d'instructions de la figure 2.8, sans étage PB; l'étage ALU pour l'instruction I doit attendre 1 cycle, i.e., la fin de l'étage MEM pour l'instruction L. Avec un étage PB, s'il y a un échec sur le tampon de préchargement et un succès sur le cache, l'instruction I doit attendre 2 cycles i.e. la fin de MEM pour l'instruction L. Ces délais peuvent être partiellement masqués en réordonnant le code en fonction de l'architecture cible et notamment de la structure du pipeline utilisé.

Politique d'écriture différée Dans un cache à écriture différée, une ligne de cache peut être modifiée et donc non cohérente avec le reste du système mémoire. Par conséquent, il n'est pas possible de copier une ligne du tampon de préchargement dans le cache avant de savoir si la ligne écrasée n'est pas modifiée. Les deux structures (cache et tampon de préchargement) sont donc accédées en parallèle durant l'étage MEM, mais afin de ne pas pénaliser le temps de cycle du processeur, l'accès au tampon n'est exploité qu'au cycle suivant si l'accès au cache est un échec. Afin de masquer ce second cycle (échec sur le cache, succès sur le tampon), un étage de pipeline supplémentaire est également utilisé, mais placé après l'étage MEM (voir figure 2.9).

Sur ce pipeline, un échec sur le cache suivi d'un succès sur le tampon prend 2 cycles ce qui est identique au temps requis pour une écriture avec succès dans le cache (la comparaison d'étiquettes ne peut être réalisée en parallèle de l'écriture). Lors d'une écriture, la ligne victime est lue (et préservée) au premier cycle. Si l'accès correspond à un échec sur le cache, le transfert entre le tampon et le cache a lieu dans le deuxième cycle avant même que soit déterminé le succès ou l'échec sur le tampon (déterminé à la fin du deuxième cycle). En parallèle, la ligne victime est envoyée au tampon d'écriture. A la fin de ce cycle, soit la donnée appartient au tampon et le transfert du tampon vers le cache était nécessaire, soit la donnée n'appartient pas au tampon et le transfert n'était pas nécessaire, mais la donnée écrasée le sera de toute façon. Ainsi, une écriture faisant un échec sur le cache, mais un succès sur le tampon prend 2 cycles i.e. le même temps qu'une écriture avec succès sur le cache. Si le cache n'est pas accédé par l'instruction suivante, l'étage PB masque le second cycle d'écriture avec succès sur le cache et le second cycle d'écriture avec échec sur le cache et succès sur le tampon.

Les délais supplémentaires dus à l'étage PB sont moins nombreux pour le pipeline avec écriture différée que pour le pipeline avec écriture simultanée. En effet, considérons une séquence d'instructions causant des délais de chargement (voir figure 2.9). Pour l'organisation de pipeline avec écriture différée, 2 cycles de délai supplémentaires sont comptabilisés uniquement dans le cas d'un échec sur le cache suivi d'un succès sur le tampon de préchargement (le cas de l'écriture avec succès n'est pas propre à la nouvelle structure de pipeline proposée). Ce cas arrive moins souvent que le cas d'un échec sur le tampon de préchargement suivi d'un succès sur le cache de l'organisation de pipeline avec écriture simultanée. Comme pour l'écriture simultanée, le code doit être réordonné afin de masquer les délais générés par cette nouvelle organisation de pipeline.

Performance Dans cette étude, nous avons utilisé l'outil Spa développé par Gordon Irlam [41] afin de générer les traces des programmes exécutés sur une station de travail de type SPARC. Les traces sont ensuite analysées par notre simulateur. Dans les simulations, nous avons considéré une configuration performante

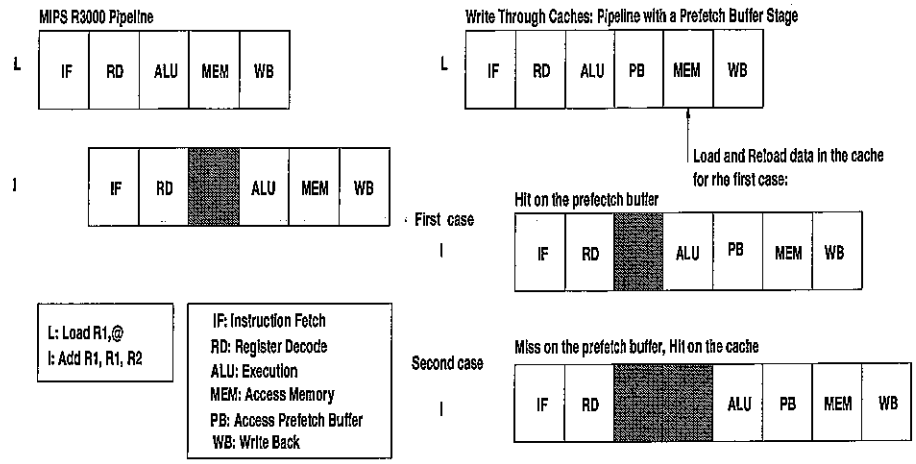

FIG. 2.8: Délai de chargement avec un étage de préchargement dédié - Cas de l'écriture simultanée.

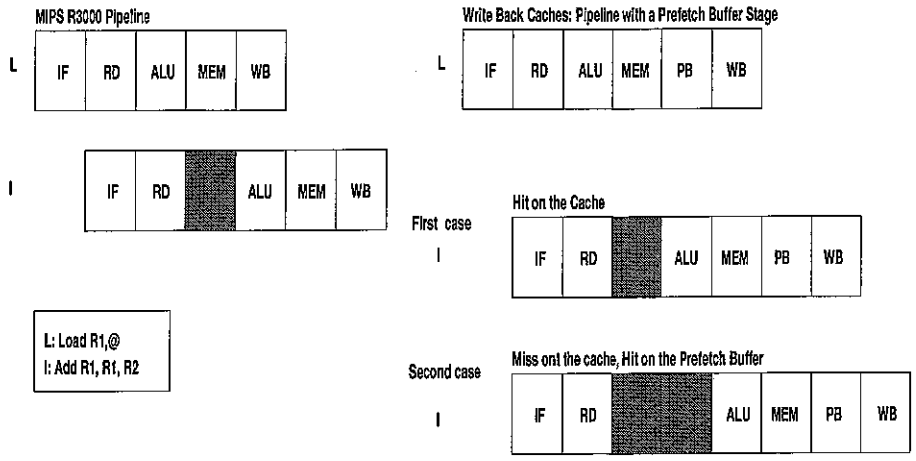

FIG. 2.9: Délai de chargement avec un étage de préchargement dédié - Cas de l'écriture différée.

(assimilable à un superscalaire large) traitant une requête mémoire à chaque cycle. La conséquence est que les seconds cycles d'une écriture avec succès sur le cache et d'une lecture ou d'une écriture avec échec sur le cache, mais avec succès sur le tampon, sont systématiquement comptabilisés. Dans notre modèle simplifié, les instructions flottantes sont exécutées en 3 cycles et toutes les autres, en 1 cycle. Pour le réordonnancement de code, nous avons utilisé un optimiseur de code objet appelé OCO [10]. Tous les programmes sont réordonnés en fonction de l'organisation de pipeline utilisée (5 ou 6 étages avec ajout de l'étage PB).

Dans cette analyse, nous avons voulu mesurer comment les cycles de délais générés par l'utilisation d'un tampon de préchargement pouvaient être masqués par l'utilisation des organisations de pipeline précédemment décrites. Pour cela, nous avons utilisé la pénalité d'échecs moyenne donnée par :

$$\bar{P} = \frac{((\text{nombre d'échecs}) - (\text{nombre de succès sur le tampon})) \times (\text{pénalité d'échecs}) + \Delta(\text{délais chargement}) + BRH}{\text{nombre d'échecs}}$$

où :

Δ(délais chargement) correspond à la différence entre le nombre de délais de chargement sur un pipeline normal et le nombre de délais sur une organisation de pipeline avec un étage supplémentaire. Ce nombre est différent selon que l'on considère une écriture simultanée ou une écriture différée. Cette quantité permet de mesurer les compromis entre un pipeline standard et le préchargement et le pipeline proposé.

BRH pour l'écriture simultanée, $BRH = 0$. Pour les caches à écriture différée, BRH est égal au nombre de succès en lecture sur le tampon de préchargement (l'opération de copie du tampon vers le cache en cas de succès sur le tampon ne peut être masquée).

Les performances de la technique de pipeline combinée avec un tampon de préchargement à plusieurs lignes sont présentées dans la figure 2.10 pour une configuration de cache fixée (taille de cache de 16 Koctets, taille de la ligne de 32 octets, associativité par ensemble de 2 blocs) et une latence mémoire de 10 cycles. Globalement, en comparant les figures 2.7 et 2.10, les réductions en nombre d'échecs conduisent à des réductions similaires en terme de pénalité moyenne d'échecs. Pour les codes qui profitent de façon modérée du préchargement, la réduction de la pénalité moyenne d'échecs est à peu près de 1 cycle (mdljdp2, wave5, li, swm256) ou de 2 cycles (tomcatv, alvinn). Pour les codes qui profitent de façon importante du préchargement, les réductions varient entre 4 cycles (eqntott) et 7 cycles (tomcatv). Il faut noter que plus le nombre de succès dans le tampon est grand, plus la performance du préchargement assisté de la technique du pipeline est importante. Par ailleurs, la politique d'écriture différée donne de moins bonnes performances que la politique d'écriture simultanée, car un succès sur le tampon en lecture conduit à un blocage du cache pour le transfert du tampon vers le cache. Avec cette technique de pipeline, les délais supplémentaires dus au chargement et à la gestion du tampon de préchargement à plusieurs lignes sont largement compensés.

Référence avec résultats détaillés : [28].

2.4 Choisir une implémentation adaptée au préchargement

Dans les études précédentes, nous avons proposé des solutions pour réduire certaines perturbations dues au préchargement, mais nous n'avons pas considéré le problème plus globalement, ni intégrer une mesure de performance permettant de mieux appréhender les intéractions entre les différentes composants du processeur et le préchargement. Nous allons décrire maintenant une stratégie de préchargement sur le second niveau de cache, qui peut être une réponse plus globale aux problèmes de pollution, bande passante et trafic mémoire précédemment étudiés.

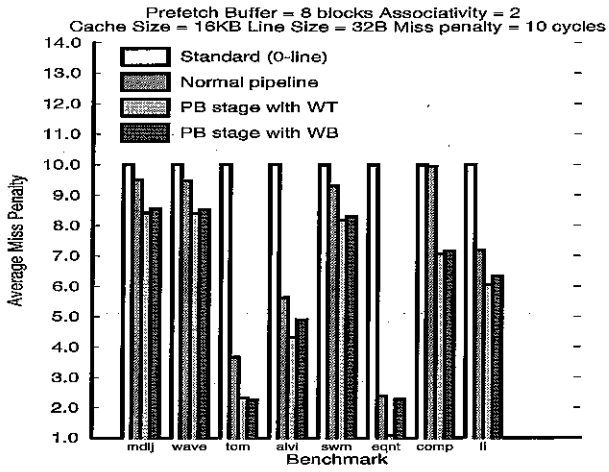

FIG. 2.10: Performance avec tampon de préchargement et étage de pipeline supplémentaire.

2.4.1 Comportement du préchargement sur le premier niveau de cache

Les études réalisées sur le préchargement matériel concernent essentiellement le premier niveau de cache. Mais les problèmes de trafic sur le bus décrits précédemment ne plaident pas pour ce type de préchargement qui augmente significativement le trafic entre le cache L1 et le cache L2, et entre le cache L2 et la mémoire : le taux auquel le processeur consomme et produit ses opérandes augmente, les mauvaises prédictions augmentent le nombre de données chargées, etc. De même, notamment pour le cache L1 et entre les caches L1 et L2, certaines requêtes normales peuvent être retardées pénalisant les performances. Enfin, un certain nombre d'études montrent que les processeurs superscalaires à exécution dans le désordre peuvent masquer des latences faibles ce qui remet en cause l'intérêt du préchargement sur le cache L1 par rapport à l'intérêt qu'il pouvait avoir sur un processeur à exécution dans l'ordre. Seznec *et al.* [78] montre que l'ordonnancement dynamique des instructions dans les processeurs superscalaires à exécution dans le désordre permet de masquer une partie de la latence mémoire des instructions. Ainsi, le principal effet du préchargement L1 étant de masquer la latence d'échec L1, le préchargement L1 est redondant avec la capacité intrinsèque des processeurs superscalaires à exécution dans le désordre de masquer les courtes latences mémoire.

2.4.2 Une solution : le préchargement sur le second niveau de cache

Dans ce travail, nous montrons que le cache L2 peut bénéficier davantage du préchargement que le cache L1 et que ce préchargement L2 n'est pas affecté par la plupart des perturbations précédemment décrites.

Plus de localité spatiale sur le cache L2 Le cache L1 voit passer toutes les requêtes mémoire. Le cache L2 voit uniquement les requêtes d'échecs du cache L1. Ainsi, le cache L1 fonctionne comme un filtre de requêtes mémoire : la plupart de l'exploitation de la localité temporelle est réalisée par le cache L1 plutôt que par le cache L2 et le rôle du cache L2 est essentiellement de charger les nouvelles données requises par le cache L1. Ainsi, le cache L2 exploite davantage les propriétés de localité spatiale.

Si le cache L2 exhibe plus de localité spatiale, il peut potentiellement être plus efficace pour le préchargement que le cache L1. Pour vérifier et quantifier nos hypothèses, nous avons mesuré la localité spatiale sur le cache L1 et sur le cache L2. Pour cela, nous avons collecté la trace L1 (la trace des accès mémoire) et la trace L2 (la trace des échecs sur le cache L1). Nous avons défini un tampon de 1000 entrées qui fonctionne comme

Programmes	Cache L1	Cache L2
gcc	54	59
ijpeg	39	90
perl	3	20
vortex	36	41
applu	67	82
hydro2d	33	66
mgrid	83	93
swim	67	99

TAB. 2.1: Pourcentage de localité spatiale (taux de succès sur les tampons de localité) pour le cache L1 et le cache L2.

un cache complètement associatif (1000 entrées = 1000 références à des données pour la trace L1 et 1000 références à des lignes de cache L1 pour la trace L2). Pour chaque trace, nous avons mesuré le nombre de requêtes adjacentes à des données déjà contenues dans le tampon. Ainsi, notre métrique de localité spatiale est réaliste dans le sens où elle tient compte du fait que les requêtes doivent être spatialement adjacentes dans un intervalle de temps raisonnable (lié à la taille du tampon).

Nous avons collecté et évalué cette métrique pour une partie des programmes extraits de la suite SPEC95. Les résultats présentés dans la table 2.1 montrent que la localité spatiale sur le cache L2 est toujours plus élevée que la localité spatiale sur le cache L1 et le rapport entre les deux varie de 1.1 à 6.6. Ainsi, un cache L2 peut *a priori* bénéficier davantage que le cache L1 du préchargement séquentiel.

Comportement du préchargement sur le cache L2 La plupart des limitations du préchargement sur le cache L1 ont un impact limité sur le préchargement L2. Tout d'abord, la latence d'échecs du cache L2 est généralement plus élevée que la latence d'échecs du cache L1 et a donc moins de chance d'être recouverte par l'ordonnancement dynamique des instructions ce qui rend le préchargement sur le cache L2 potentiellement plus utile. Par ailleurs, le bus entre le cache L2 et la mémoire principale est moins utilisé que le bus entre le cache L1 et le cache L2, les requêtes d'échecs sur la cache L2 arrivant à une fréquence moins élevée que les requêtes d'échecs L1. Enfin, le cache L2 possède généralement un unique port d'accès, mais la fréquence d'accès au cache L2 est bien plus faible que celle d'accès au cache L1 (deux ordres de grandeur). Ainsi, la plupart des accès au cache L2 peuvent tester le cache sans délai. S'ils sont retardés, ils ont un impact limité sur la latence L2 (proportionnellement moins important que sur la latence L1).

Performance Pour cette étude, nous avons utilisé un simulateur de processeur développé dans notre équipe et basé sur ASF [5]. L'architecture simulée (présentée figure 2.11) est un processeur superscalaire cadencé à 800 Mhz, de degré 4 à exécution dans le désordre comportant 3 unités de calcul entier, 1 unité de calcul flottant, 2 unités de chargement/rangement et 1 unité de branchement. Les registres sont renommés (64 registres entiers physiques versus 32 registres entiers logiques, même valeur pour les registres flottants). La hiérarchie mémoire est composée de 2 niveaux de cache, un cache L1 et un cache L2. Le cache L1 est un cache non bloquant à 2 ports, sa taille est de 32 Koctets, la taille de sa ligne est de 32 octets, il est associatif par ensemble de 4 blocs et à écriture différée. Le cache L2 est de 256 Koctets avec une taille de ligne de 32 octets et une associativité de 4. Il est connecté à la puce du processeur via un bus dédié de 64 bits à la moitié de la fréquence du processeur. Le bus mémoire est cadencé à 100 Mhz et est connecté à une SDRAM à 10 ns (5-1-1-1). Le transfert de l'adresse est de 2 cycles et la latence d'accès aux 64 premiers bits est de 7 cycles bus. Les traces sont collectées à l'aide de ATOM [30] sur des stations de travail Alpha.

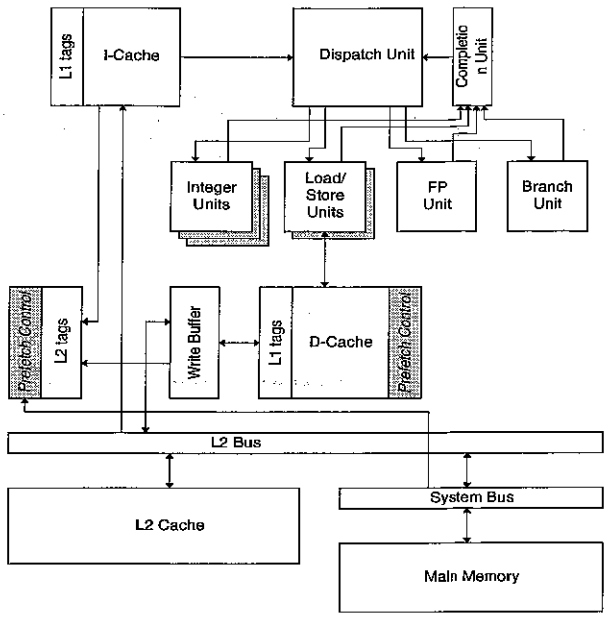

FIG. 2.11: Architecture simulée.

La technique de préchargement implémentée est celle du préchargement marqué. Nous n'autorisons que 4 requêtes de préchargement en cours qui sont stockées dans une table de préchargement à 4 entrées permettant notamment d'éviter les préchargements redondants. Dans la mesure du possible, nous voulons éviter qu'une requête de préchargement retarde une requête normale. Aussi, les requêtes normales ont priorité sur les requêtes de préchargement pour accéder aux ports du cache. Les requêtes de préchargement peuvent être lancées lorsqu'un ou plusieurs ports du cache sont libres. Ces cycles libres peuvent être déterminés à l'avance en contrôlant les étages du pipeline qui précèdent les étages d'accès à la mémoire.

La figure 2.12 donne la réduction du taux d'échecs avec du préchargement L1 et L2. Le préchargement L1 réduit le taux d'échecs, excepté pour Perl et Vortex, deux programmes qui utilisent intensivement la mémoire, ayant peu d'échecs et dont la pollution induit sur le cache L1 par le préchargement dégrade les performances. Avec le préchargement L2, tous les programmes exhibent de 7% à 99% de réduction du taux d'échecs. En terme de taux d'échecs, le préchargement sur les caches L1 et L2 exhibe des performances similaires. Par contre, lorsque l'on considère d'autres paramètres comme le trafic L1-L2, la latence d'échecs L1 et L2, les accès aux caches L1 et L2, l'exécution dans le désordre, les performances du préchargement sur le cache L2 sont meilleures que celles du préchargement sur le cache L1 (voir figure 2.13).

En conséquence et contrairement à un certain nombre d'études, le préchargement devrait être proposé sur le cache L2 plutôt que sur le cache L1. La récente annonce du préchargement sur le cache L2 du Pentium 4 est une validation supplémentaire de cette analyse.

Références avec résultats détaillés : [29, 12, 13].

2.4.3 Réduire les pertubations sur le trafic bus engendrées par le préchargement

Malgré les bonnes performances du préchargement L2, il reste néanmoins le problème de la perte de performance induite par le préchargement, essentiellement sur les programmes entiers (voir figure 2.13). Ceci est du en partie à une sur-utilisation du bus qui entraîne une augmentation de la latence moyenne pour les

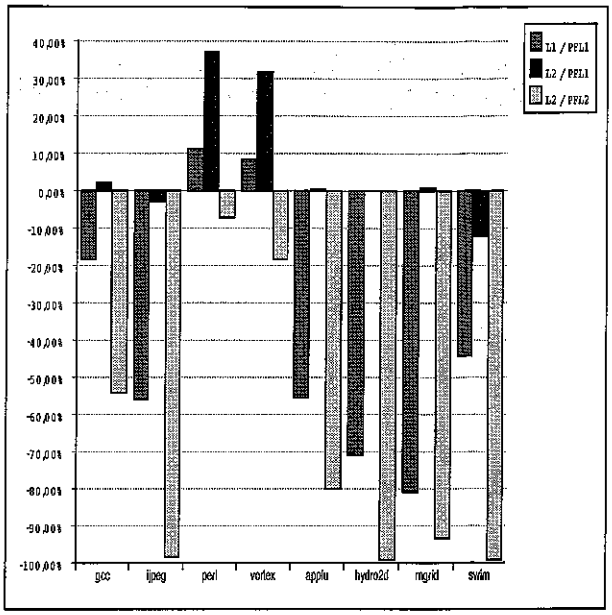

FIG. 2.12: Réduction du nombre d'échecs due au préchargement L1 et au préchargement L2 : taux d'échecs sur L1 avec préchargement L1 (L1 / PFL1), taux d'échecs sur le L2 avec préchargement L1 (L2 / PFL1) et taux d'échecs sur le cache L2 avec préchargement L2 (L2 / PFL2).

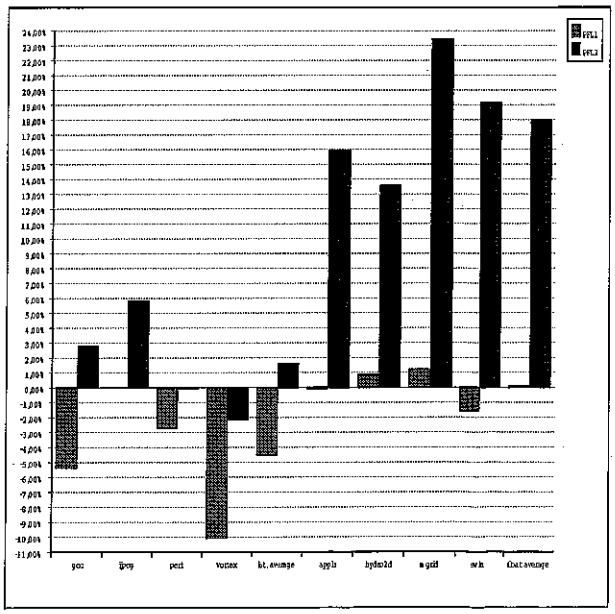

FIG. 2.13: Amélioration en IPC (%) pour le préchargement sur le cache L1 (PFL1) et le préchargement sur le cache L2 (PFL2).

requêtes normales du processeur.

Le but est maintenant de mettre en place un contrôle dynamique des accès au bus permettant de désactiver le préchargement lorsqu'il pénalise le fonctionnement normal de la machine. Ce contrôle repose sur la mesure du trafic sur le bus mémoire et sur les performances du préchargement.

Pénalités engendrées par les requêtes de préchargement Comme nous l'avons déjà précisé précédemment, le problème de bande passante est encore accentué par les techniques de préchargement qui génèrent notamment davantage de transactions sur le bus, et un même nombre d'opérations sur des périodes de temps plus courte. Le premier phénomène pourrait ne pas être pénalisant si les requêtes supplémentaires n'entraient pas en conflit avec les requêtes normales du processeur. Le deuxième phénomène peut pénaliser la latence moyenne d'accès des requêtes normales du fait de conflits sur le bus entre requêtes normales et requêtes de préchargement.

Considérons une latence mémoire L (latence d'accès au premier groupe de mots du bloc de cache incluant le transfert sur le bus), M le rapport entre la taille du bloc de cache et la largeur du bus et l, la latence de retour des données sur le bus (dépend de la fréquence du bus et de sa largeur). Considérons deux accès mémoire lancés à t_1 et t_2:

Date lancement accès mémoire	Date arrivée 1ère donnée	Date fin accès mémoire
t_1	$t_1 + L$	$T_1 = t_1 + L + (M-1) \times l$
t_2	$max(T_1, t_2 + L - l) + l$	$T_2 = max(T_1, t_2 + L - l) + M \times l$

Afin que le premier accès ne pénalise pas le second, il faut que la date de lancement de ce dernier soit telle que : $t_2 + L \geq T_1 + l \quad \rightarrow \quad t_2 - t_1 \geq M \times l$

Nous définissons ainsi les notions de distance d'échec et de distance d'échec optimale.

Définition. La distance d'échec est le nombre de cycles pendant lesquels le bus de données est libre entre 2 échecs consécutifs sur le niveau de cache considéré (deux requêtes normales du processeur).

Définition. La distance d'échec optimale est égale à $M \times l$ et correspond à la distance d'échecs minimum pour laquelle le lancement du deuxième échec n'est pas retardé par le premier.

Théoriquement, une requête de préchargement ne pénalise pas la latence d'accès aux requêtes normales si elle est initiée $M \times l$ cycles avant la requête normale suivante et que la distance d'échecs est supérieur à $2 \times M \times l$.

Contrôle du préchargement L'idée de ce travail a été de gérer dynamiquement le compromis entre la perte liée aux conflits entre requêtes normales et requêtes de préchargement sur le bus (contrôle suivant l'occupation du bus) et le gain lié au préchargement (contrôle suivant les performances). Nous utilisons pour cela un compteur de préchargement. Ce compteur de préchargement oscille dynamiquement entre différentes valeurs pendant l'exécution du programme, il est incrémenté lorsque le préchargement pénalise le fonctionnement normal du processeur (contrôle suivant l'occupation du bus) et est décrémenté lorsque le préchargement profite au fonctionnement du processeur (contrôle suivant les performances).

Contrôle selon l'occupation du bus Le contrôle doit permettre de désamorcer le préchargement en présence de rafales d'échecs sur le cache L2. Un accès spéculatif diminue les performances du processeur s'il n'est pas initié dans un intervalle d'accès suffisament long (2 × M × 1 cycles). Comme des études l'ont

déjà prouvé [11, 92, 94], les intervalles repondant à ce critère sont relativement prévisibles sur le cache de premier niveau. Les échecs arrivent par rafales, c'est-à-dire qu'ils sont initiés par série dans des intervalles de temps courts, suivis d'une longue période avec peu d'échecs. Ce phénomène se produit également à la sortie du cache L2. Ainsi, lorsqu'un préchargement pénalise le fonctionnement normal du processeur (i.e. requête normale retardée à cause d'un accès pour un préchargement), le compteur de préchargement est incrémenté.

Contrôle suivant les performances L'efficacité du préchargement peut s'évaluer par le simple fait que le processeur accède ou non aux données préchargées [23]. Le compteur de préchargement est donc décrémenté lors d'un accès à une donnée préchargée (utilisation du bit introduit pour le préchargement marqué). Lorsque la valeur minimale est atteinte, tout accès supplémentaire à une donnée préchargée ne le modifie plus.

Activation/désactivation du préchargement Le préchargement est désactivé lorsque le compteur atteint la valeur maximale, le compteur est alors remis à 0. La réactivation du préchargement dépend des performances de celui-ci. En fait, plus le préchargement pénalise le processeur, plus le temps d'attente avant la reprise du préchargement augmente et moins le préchargement pénalise le processeur, moins l'attente est grande avant de reprendre le préchargement. Cette règle est mise en oeuvre via un compteur d'attente (incrémenté à chaque fois que le compteur de préchargement atteint sa valeur seuil).

Performance Le mécanisme de préchargement adaptatif alliant contrôle du bus et efficacité du préchargement permet aux programmes qui enregistraient des pertes de performance d'obtenir des performances de l'ordre de la configuration sans préchargement (voir figure 2.13). Ces programmes étaient pénalisés par les requêtes de préchargement, soit parce qu'elles étaient inefficaces et pouvaient perturber globalement (cache, bus, ...) le comportement du processeur (contrôle selon les performances), soit parce qu'elles pénalisaient des requêtes normales du processeur (contrôle du bus). De plus, le mécanisme de préchargement adaptatif ne pénalise pas les programmes flottants pour lesquels le préchargement sur le cache L2 est efficace.

Références avec résultats détaillés : [12].

2.5 Conclusion

Les perturbations engendrées par les requêtes de préchargement sont nombreuses et doivent être considérées lorsque l'on évalue une technique de préchargement. Cette considération passe par une conception plus précise de la stratégie de préchargement, notamment par l'intégration des intéractions avec les autres composants de l'architecture. Nos différents travaux ont permis de mieux appréhender ces perturbations et de proposer des stratégies permettant d'améliorer les performances du préchargement.

Le préchargement sur le cache de second niveau est une réponse simple aux perturbations rencontrées par le préchargement sur le cache de premier niveau et nos expériences ont confirmé qu'associé avec un contrôle du bus, il semble être une solution performante pour les architectures actuelles. Dans le chapitre suivant, lorsque nous introduisons du préchargement matériel, nous considérons du préchargement sur le second niveau de cache.

CHAPITRE 3

Adapter les nouvelles architectures aux nouvelles applications

Les applications multimédia sont devenues la principale charge de travail exécutée par un ordinateur personnel et représentent un marché en pleine croissance grâce au développement d'Internet et à la diffusion possible de grandes quantités de données, grâce à la normalisation d'une partie importante de ces applications, et surtout grâce à l'augmentation des performances des machines. Ces performances augmentent grâce à l'exploitation possible de différentes formes de parallélisme. Mais ces fonctionnalités supplémentaires accroissent la complexité du matériel et les exploiter correctemment devient de plus en plus difficile.

Notre travail s'est focalisé sur l'étude et l'optimisation des applications multimédia pour les architectures émergentes, architectures permettant d'exploiter en plus du parallélisme d'instructions, du parallélisme de données et du parallélisme de flots. Ces différentes adaptations/optimisations sont réalisées au niveau du code source et du matériel.

3.1 Adéquation applications multimédia - architectures émergentes

Les applications multimédia Les applications multimédia requièrent des puissances de calcul importantes. Alors que les systèmes à base de processeurs généralistes n'ont pas été créés pour les applications multimédia, une grande partie de leurs cycles d'horloge sont aujourd'hui utilisés par de telles applications. Les caractéristiques importantes de ces applications sont le traitement d'une grande quantité de données, un traitement identique sur un ensemble important de données, le traitement par flux des données et le traitement de tâches indépendantes.

Ces différentes caractéristiques mettent en avant le parallélisme de données (traitements identiques) et le parallélisme de flots (tâches indépendantes) de ces applications. Par ailleurs, elles mettent l'accent sur le problème particulier de l'accès aux données, donc de la hiérarchie mémoire. En effet, le volume de données traitées est très important et tend encore à s'étendre (augmentation de la complexité des images). Par ailleurs, le traitement des données se fait généralement par flux ce qui implique une forte localité spatiale, mais une localité temporelle presque nulle.

Dans notre travail, nous n'avons pas considéré tous les types de données manipulés par les applications multimédia, mais uniquement le traitement et la génération d'images 2D et 3D. Les catégories d'applications considérées sont :

- la compression-décompression d'une image fixe en deux dimensions et la compression-décompression en temps réel d'une vidéo constituée de plusieurs images en deux dimensions.
- la synthèse d'images, i.e., la génération en temps réel de scènes en trois dimensions.

Ces applications multimédia et notamment celles liées à la compression et au traitement d'images et de vidéos nécessitent des puissances de calculs importantes, notamment du fait de l'aspect temps-réel de ces

traitements. Toutes ces applications sont décrites précisément dans l'annexe de ce mémoire.

Architectures émergentes adaptées La vitesse de calcul nécessaire à ces applications peut être apportée par l'augmentation de la vitesse de traitement des composants électroniques (augmentation de la fréquence de traitement), mais aussi par l'utilisation du parallélisme. Différents types de parallélisme sont exploitables pour augmenter les performances, parallélisme au niveau des instructions (ILP pour *Instruction Level Parallelism*), parallélisme au niveau des données (DLP pour *Data Level Parallelism*) et parallélisme au niveau des flots (TLP pour *Thread Level Parallelism*).

Le *parallélisme d'instructions* est donné par les instructions (assembleur) indépendantes d'un programme. Il peut être extrait soit dynamiquement (à l'exécution), soit statiquement (à la compilation). Les microprocesseurs superscalaires qui dominent actuellement le marché des machines généralistes orientées performance utilisent un ordonnancement dynamique des instructions à l'exécution pour pouvoir en démarrer plusieurs en parallèle au cours du même cycle. Cette extraction dynamique du parallélisme presque entièrement implémentée de manière architecturale (stations de réservation, renommage de registres, ...) est très efficace et ne sollicite pratiquement pas le compilateur. Par contre, elle a un coût matériel important. Les microprocesseurs VLIW (*Very Long Instruction Word*) dépendent tout autant de l'exploitation de ce parallélisme d'instructions pour obtenir des gains de performances, mais ces derniers étant de conception différente, l'ordonnancement des instructions ne peut se faire dynamiquement et est donc réalisé statiquement à la compilation ; il dépend donc fortement des compilateurs et des techniques d'optimisation mises en oeuvre.

Le *parallélisme de données* consiste à effectuer le même traitement sur des données indépendantes. Pour exploiter ce parallélisme de données, les jeux d'instructions des processeurs généralistes ont été étendus avec des instructions ne travaillant plus sur une seule valeur, mais sur un ensemble de données regroupées en vecteurs. Ces instructions sont dites multimédia ou SIMD (*Single Instruction Multiple Data*). Le gros intérêt de telles instructions est que chacune d'elle traite n fois plus de données qu'une instruction classique. Les premières extensions SIMD sont apparues en 1995 et ont été enrichies au cours de ces dernières années (exemple : ajout d'instructions pour les traitements flottants).

Le *parallélisme de flots* peut être obtenu soit en exécutant en parallèle plusieurs programmes, soit en parallélisant une application. La première forme correspond à une activité multi-utilisateurs, tandis que la seconde a pour but d'accélérer l'exécution d'une application, comme le font traditionnellement les calculateurs parallèles. Pour exploiter ce parallélisme de flots, différentes architectures peuvent être considérées. D'une part, des architectures multiflot simultané (SMT pour *Simultaneous MultiThreading*) qui permettent aux flots de partager au sein du même processeur une grande partie des ressources d'exécution. D'autre part, des architectures dîtes multiprocesseurs soit intégrées sur une puce (CMP, pour *Chip MultiProcessor*), soit utilisant plusieurs processeurs en parallèle. Dans cette dernière approche, les ressources d'exécution sont partagées statiquement entre les processeurs élémentaires qui exécutent chacun un flot d'instructions.

Compte tenu de leurs caractéristiques, les applications multimédia peuvent potentiellement profiter des différents types de parallélisme mis en oeuvre au travers des architectures émergentes. Par contre, adapter les applications multimédia à ces architectures posent un certain nombre de problèmes : problème pour exploiter efficacement les unités multimédia (utilisation d'instructions SIMD, section 3.2), problème pour gérer efficacement le partage des ressources (utilisation du SMT, section 3.3) et problème pour obtenir une parallélisation efficace sans dégradation de performance (utilisation de processeurs généralistes en parallèle, section 3.4).

Par ailleurs, les problèmes liés à la hiérarchie mémoire (cache, mémoire, bus) communs à toutes les architectures sont encore accentués. En effet, le système mémoire reste inchangé lorsqu'un processeur généraliste est étendu à d'autres formes de parallélisme. Les caches, composants clés dans les systèmes mémoire des

processeurs généralistes, ne sont pas adaptés en général au calcul multimédia. Les accès mémoire exhibent de la localité spatiale et très peu de localité temporelle (les données d'entrée sont souvent utiles durant une courte période de temps). Par ailleurs, le trafic mémoire augmente et devient un facteur limitatif encore plus important.

3.2 Architecture superscalaire avec extensions multimédia

Dans ce travail, nous sommes intéressés aux performances potentielles des applications multimédia utilisant les extensions SIMD (*Single Instruction Multiple Data*). Dans ce but, nous avons réalisé la vectorisation de certaines applications multimédia et avons intégré des instructions de préchargement adaptées à ce type d'applications (traitement de flots de données). Nous avons d'une part, évalué les performances du SIMD et du préchargement et d'autre part, mesuré l'impact de la bande passante mémoire sur les performances de ces optimisations. Nous avons utilisé comme support d'analyse les extensions SIMD AltiVec qui étend le jeu d'instructions du PowerPC avec 170 nouvelles instructions (multiplication-accumulation, toutes les permutations possibles entre deux registres vectoriels, conversions entières flottantes, préchargement, ...).

3.2.1 Principe

Les instructions SIMD (*Single Instruction Multiple Data*) ont été rajoutées aux architectures afin d'exploiter le parallélisme de données des applications multimédia. Elles permettent, par exemple, d'effectuer une même opération en parallèle sur plusieurs éléments (exemple : pixels) à l'intérieur d'un même registre (vecteur). Elles permettent également de réduire la sous-utilisation des chemins de données et des unités fonctionnelles des processeurs généralistes travaillant généralement sur 32 ou 64 bits, alors que les applications multimédia opèrent sur des données de 8 ou 16 bits.

La plupart des fabricants intègrent actuellement des extensions multimédia à leur jeu d'instructions (citons notamment SSE2 d'Intel, 3DNow! d'AMD et AltiVec de Motorola). Ces extensions présentent un certain nombre de similitudes et d'instructions communes et après avoir traité uniquement les calculs entiers, elles traitent maintenant des calculs flottants nécessaires pour accélérer notamment les jeux 3D. Mais elles se distinguent également par leur complexité : certains jeux SIMD ont plutôt introduit des instructions génériques, alors que d'autres ont plutôt fait le choix d'introduire des instructions très spécialisées. Cette distinction est parfaitement illustrée par les deux jeux d'instructions SIMD les plus utilisées, SSE et AltiVec. Dans AltiVec, toutes les opérations simples ont été implémentées et ce pour tous les types de données, ainsi que les opérations les plus génériques. Dans SSE, par contre, les instructions introduites répondent aux besoins propres d'applications données et sont donc très spécifiques. Par ailleurs, les choix architecturaux diffèrent : utilisation des composants généralistes pour l'exécution des instructions SIMD ou au contraire intégration des composants propres aux traitements SIMD (exemple : AltiVec emploie des registres et des unités fonctionnelles propres).

3.2.2 Exploitation du parallélisme de données

Les applications multimédia disposent massivement de parallélisme de données, mais il faut pouvoir l'extraire afin d'exploiter les atouts des jeux d'instructions SIMD. Ainsi, les problèmes se concentrent essentiellement sur la vectorisation.

Lorsque les applications sont vectorisables, une première approche consiste à utiliser les mêmes techniques de vectorisation que celles traditionnellement utilisées pour paralléliser les applications scientifiques dans les

machines vectorielles [102, 99] (entre les itérations d'une boucle). Mais d'autres caractéristiques propres aux applications multimédia et aux instructions SIMD doivent être exploitées : parallélisme de données à un grain plus fin (vecteurs de quelques éléments), utilisation d'opérations complexes (racine carrée), etc. Des travaux spécifiques ont été réalisés sur la vectorisation automatique [25, 51, 8] et semi-automatique [33] (avec assistance du programmeur qui doit fournir par exemple des informations sur l'alignement des données), les optimisations concernant essentiellement des structures simples (boucles simples, boucles internes, minimum, maximum, alignement des structures sur 128 bits). D'autres travaux concernent l'extraction du parallélisme de données au niveau des blocs de base [50].

Mais beaucoup d'applications multimédia ne sont pas vectorisables simplement et nécessitent des transformations pour bénéficier des instructions SIMD. Il faut réaliser la vectorisation inter-itération, intra-itération ou par opérations similaires. La vectorisation inter-itération peut être appliquée lorsque toutes les itérations sont indépendantes. La vectorisation intra-itération consiste à regrouper dans un vecteur des données adjacentes sur lesquelles des opérations similaires sont effectuées au sein d'une même itération. La vectorisation par opérations similaires consiste à regrouper des variables indépendantes dans un vecteur sur lesquelles sont réalisées des opérations similaires.

La transformation des programmes doit également tenir compte des problèmes de placement et d'alignement des données qui empêchent une vectorisation efficace des programmes. Les problèmes de placement sont liés au fait que les éléments d'un vecteur doivent être contigüs en mémoire. Afin d'exploiter les calculs vectoriels, une réorganisation des données peut être nécessaire (exemple : utilisation d'une structure de tableaux au lieu d'un tableau de structures comme utilisé dans les moteurs 3D). Les problèmes d'alignement de données existent dans certains jeux d'instructions qui ne manipulent que des données alignées (exemple : AltiVec opère sur des données alignées sur 128 bits). Une solution consiste à travailler sur des accès alignés dans le corps de la boucle et à exécuter les débuts et fins des tableaux sur des accès non alignés en dehors de la boucle.

Un autre moyen pour optimiser les applications multimédia consiste à utiliser des librairies, des macros ou à écrire directement le code en assembleur. Même si cela représente une tâche fastidieuse, les gains possibles en vitesse d'exécution peuvent être suffisants pour motiver les programmeurs. D'ailleurs, afin de faciliter l'utilisation des instructions SIMD, Motorola tout comme Intel a introduit des instructions spéciales (*intrinsics*) permettant une utilisation simplifiée des instructions SIMD sans avoir recours à l'assembleur. Ce sont des appels de fonctions qui s'utilisent directement en langage haut niveau et qui manipulent de nouveaux types de données (utilisation de variables et non de registres). Cette fonctionnalité simplifie la tâche du programmeur, mais nécessite de connaître le jeu d'instructions SIMD cible.

D'autres techniques peuvent améliorer les performances des applications multimédia vectorisées, notamment les techniques de préchargement exploitant la localité spatiale des données. En effet, les applications multimédia manquent généralement de localité temporelle (traitement par flots de données), ce qui rend les caches présents dans les microprocesseurs génériques relativement inefficaces [68]. En revanche, la localité spatiale dans ces applications est très importante. Motorola a intégré à AltiVec des instructions de préchargement par flot permettant un accès aux données en mémoire sous forme de flots optimisés pour tirer parti au mieux de la bande passante mémoire. Une instruction de préchargement de ce type prend en paramètre l'adresse de départ des données à précharger, le nombre de lignes de cache à précharger et le pas entre deux données.

A l'heure actuelle, les compilateurs ne sont pas réellement capables d'exploiter le parallélisme de données (SIMD) d'un programme écrit en langage haut niveau même si différents travaux de recherche s'y consacrent. Dans nos travaux sur le SIMD, toutes les optimisations SIMD sont spécifiées par le programmeur (*intrinsics*).

3.2.3 Comportement des applications multimédia sur un processeur moderne

Les applications multimédia optimisées avec des instructions SIMD AltiVec mettent à l'épreuve essentiellement deux éléments du processeur : les blocs de calculs SIMD (unités fonctionnelles, registres, chemins de données, etc) et la hiérarchie mémoire. Dans ce travail, nous avons étudié le potentiel des optimisations SIMD AltiVec en mettant particulièrement l'accent sur les problèmes de hiérarchie mémoire et de bande passante. D'autres études ont été réalisées sur l'augmentation des performances liée à l'ajout d'instructions SIMD [100, 7, 63, 68].

Le processeur PowerPC G4 considéré pour notre évaluation de performance est un microprocesseur RISC superscalaire à ordonnancement dynamique à 4 étages de pipeline (*Fetch, Dispatch, Execution, Completion*) pouvant exécuter des instructions SIMD (AltiVec). Il possède 6 unités d'exécution avec une station de réservation pour chacune : 2 unités entières FX1 et FX2, 1 unité flottante FPU, 1 unité de Load/Store, 1 unité vectorielle de permutation, 1 unité vectorielle arithmétique et logique. Cette dernière unité est décomposée en trois sous unités : VSI pour les calculs entiers vectoriels simples, VCI pour les calculs entiers complexes et VFP pour les calculs vectoriels flottant simple précision. Le simulateur de G4 utilisé pour nos évaluations est un simulateur fourni par Motorola. Il simule un PowerPC G4 à 400Mhz avec un cache de données de premier niveau de 32 Koctets et un cache de second niveau de 512 Koctets sur bus dédié à la moitié de la fréquence du processeur. Le bus mémoire a une largeur de 64 bits et fonctionne à 100 Mhz (utilisation de SDRAM).

Les résultats de l'exécution des programmes analysés (voir annexe méthodologie) sont présentés dans la figure 3.1. Dans le travail sur le SIMD, nous avons également étudié quelques microbenchmarks de type DSP et numérique[1]. Les tailles des ensembles de données manipulés varient de 1 à 16 Moctets (excepté pour le produit matriciel). Les programmes flottants emploient des vecteurs de 4 éléments flottants simple précision et les programmes entiers des vecteurs de 16 éléments de 8 bits, 8 éléments de 16 bits, et 4 éléments de 32 bits. Tous ces programmes présentent une forte localité spatiale et une faible localité temporelle.

Pour toutes ces applications, nous avons réalisé des transformations manuelles (à l'aide des instructions spéciales *intrinsics*) afin de les vectoriser (inter et intra itérations) en traitant au préalable les problèmes de placement et d'alignement des données. Nous avons également introduit des instructions de préchargement (la taille maximale du flot préchargé est de 128 Koctets). Pour ces différentes optimisations, nous avons essayé d'utiliser des méthodes *a priori* performantes, mais rien ne prouve l'optimalité de ces optimisations. Le graphe de la figure 3.1 représente pour chaque programme l'accélération des 3 versions (avec préchargement, avec SIMD et avec SIMD + préchargement) par rapport à la version originale.

Nous observons des accélérations variant de 2,3 à 9,8 avec SIMD et préchargement logiciel. L'implémentation SIMD de Motorola permet de traiter des vecteurs de 4 éléments flottants. Les accélérations supérieures à 4 traduisent une meilleure adaptation d'AltiVec aux applications multimédia et DSP que le jeu d'instructions PowerPC. En effet, une des particularités d'AltiVec est de disposer d'une unité SIMD spécialisée dans les permutations (réordonnancement des données) souvent nécessaires pour le passage en SIMD. En AltiVec, les permutations peuvent être réalisées en parallèle de calculs, ce qui donne un avantage important à AltiVec par rapport aux autres jeux d'instructions SIMD. Par ailleurs, il y a deux types de programmes dont l'accélération est inférieure à 4, ceux qui sont limités par la bande passante mémoire (DAXPY, X matrices, 3D Culling) soit parce qu'ils passent moins de temps dans les calculs SIMD, soit qu'ils sont intrinsèquement limités par la bande passante mémoire (DAXPY) et ceux qui sont limités par le parallélisme extractible

[1] 1) Les filtres FIR et IIR qui sont les deux opérations fondamentales dans le traitement de signaux digitaux (DSP). 2) DAXPY qui est employé dans le calcul numérique intensif : $Y_i = \alpha \times X_i + Y_i$. et 3) Le produit matrice-matrice avec des matrices de taille inférieure à 64×64.

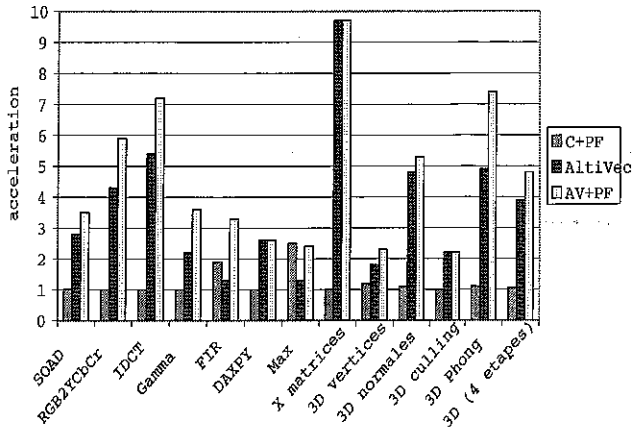

FIG. 3.1: Accélération avec préchargement par flot (C+PF), SIMD (AltiVec) et SIMD + préchargement (AV+PF) de noyaux d'application multimédia par rapport à la version originale (version C).

(FIR, Max).

Nous avons également évalué l'impact de la bande passante mémoire sur les performances des applications multimédia. Nous avons étudié l'impact sur les performances d'une division par 2, d'un doublement et d'un quadruplement de la bande passante mémoire sans changement de la latence (caractéristique non modifiable dans le simulateur de Motorola). Des mémoires proposant une bande passante doublée voire quadruplée sont actuellement disponibles pour un coût assez faible (mémoires DDRSDRAM (*Double Data Rate Synchronous DRAM*) et RAMBUS). Les figures 3.2 et 3.3 présentent l'impact de la bande passante mémoire sur les performances des applications vectorisées (sans préchargement figure 3.2 et avec préchargement figure 3.3).

On constate que l'augmentation de la bande passante mémoire profite davantage aux versions AltiVec des applications qu'aux versions C. En effet, le temps passé dans les opérations mémoire (relativement au temps d'exécution) est bien plus important dans les programmes AltiVec (traitements SIMD). On remarque également que l'augmentation de la bande passante mémoire a plus d'impact sur les versions avec préchargement que sans préchargement, ce qui s'explique par le fait que le préchargement permet de recouvrir la latence mémoire par des calculs.

DAXPY et le moteur 3D sont sensibles à l'augmentation de la bande passante mémoire dans les deux versions (même si AltiVec en bénéficie davantage). SOAD, RGB2- YCbCr, Gamma et Max sont sensibles à la latence mémoire en version C, mais très peu à la bande passante mémoire. En effet, une fois le préchargement activé, l'augmentation de la bande passante mémoire n'a presque plus d'impact sur leurs performances. FIR et IDCT sont essentiellement limités par les performances de calcul du microprocesseur, et ne sont pas sensibles à la bande passante mémoire en version C et très peu en AltiVec.

Avec une bande passante mémoire multipliée par 4, la moitié des programmes dans leur version AltiVec ont des performances comparables à celles obtenues avec une hiérarchie mémoire parfaite (barre L1 PARFAIT

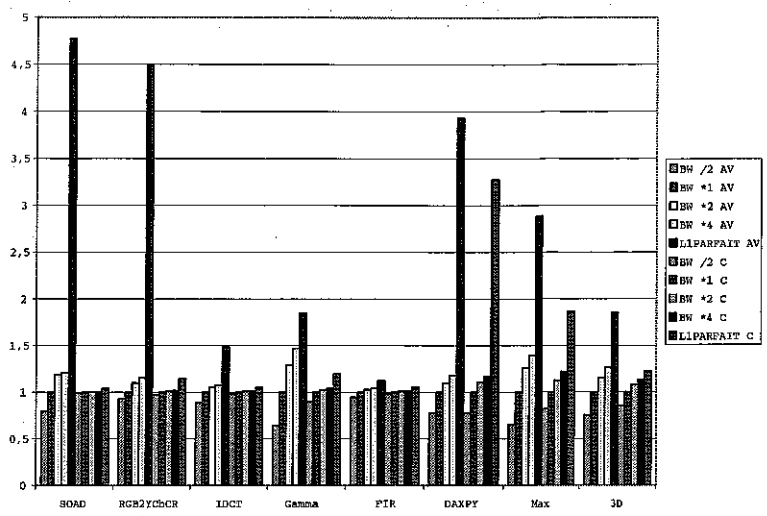

FIG. 3.2: Impact de la bande passante mémoire avec SIMD (BW=bande passante mémoire, AV=AltiVec, C=version originale, L1PERFECT=cache de premier niveau parfait).

C dans la figure 3.2). SOAD, RGB2YCbCr, DAXPY, Max, et le moteur 3D ont des besoins en bande passante mémoire plus importants que ceux étudiés pour pouvoir saturer les unités d'exécution AltiVec.

En utilisant les instructions AltiVec (SIMD) et le préchargement logiciel par flot, les performances exhibées sont importantes notamment grâce à une forte réduction de l'impact de la latence mémoire. En utilisant des mémoires proposant une bande passante quadruplée, disponibles aujourd'hui, les applications multimédia pourraient bénéficier d'une amélioration de performance jusqu'à 40 %. Ces résultats confirment l'intérêt du SIMD combiné avec du préchargement pour les applications multimédia et mettent l'accent sur les problèmes de dimensionnement du système mémoire (dans ce cas précis, du bus).

Ce travail d'optimisation SIMD des applications multimédia va nous servir de base dans l'étude combinée du parallélisme de données et du parallélisme de flots, notamment les études sur le moteur 3D. En effet, pour le parallélisme de flots, nous nous sommes essentiellement concentrés sur les performances du rendu 3D polygonal et plus particulièrement sur Mesa [66]. Le but est maintenant d'obtenir les meilleures performances possibles en exploitant toutes les fonctionnalités (tous les niveaux de parallélisme) des architectures émergentes. Par ailleurs, les résultats obtenus sur le dimensionnement mémoire avec l'utilisation combinée du SIMD et du préchargement vont également nous être utiles pour le travail sur le parallélisme de flots.

Références avec résultats détaillés : [73].

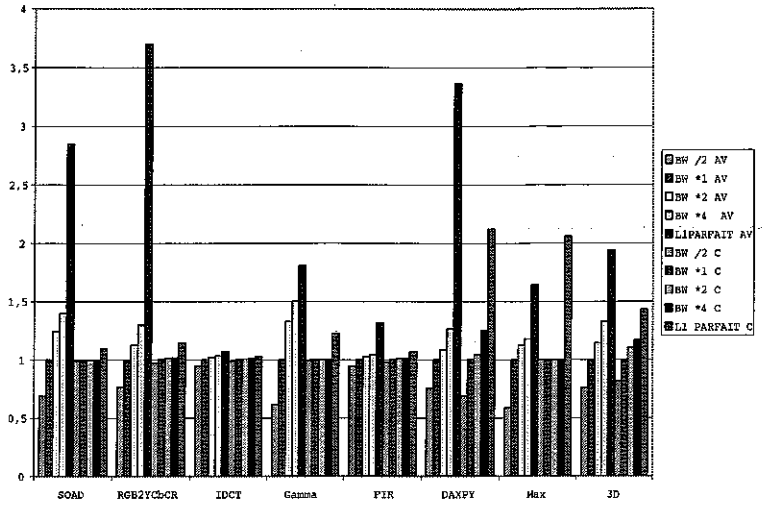

FIG. 3.3: Impact de la bande passante mémoire avec SIMD et préchargement (BW=bande passante mémoire, AV=AltiVec, C=version originale, L1PERFECT=cache de premier niveau parfait).

3.3 Architecture multiflot simultané

Nous nous sommes focalisés dans ce travail sur l'étude des performances des applications multimédia lorsqu'était également exploité le parallélisme de flots. Nous avons au travers des caractéristiques des données dans les applications multimédia, parallélisme et localité spatiale, étudié d'une part les performances de la combinaison du multiflot simultané (SMT pour *Simultaneous MultiThreading*) et d'autre part les performances du préchargement matériel (voir chapitre précédent). Mais ces techniques efficaces en monoflot placent une pression supplémentaire sur le bus mémoire qui s'ajoute à celle déjà exercée par le multiflot simultané. Nous étudions particulièrement ces aspects dans l'étude combinée avec le SIMD et le préchargement. Dans ce travail, nous présentons l'optimisation particulière du multiflot simultané pour la 3D polygonale temps-réel qui peut tirer parti de la structure du pipeline graphique.

3.3.1 Principe

Le matériel disponible (grand nombre de transistors) est de plus en plus difficile à exploiter efficacement par un unique flot d'instructions, la latence mémoire et les dépendances de données entraînant une sous-utilisation du matériel dans les processeurs superscalaires à exécution dans le désordre. Ainsi, le débit d'instructions exécutées est bien inférieur à la largeur du superscalaire considéré. Il est alors possible d'utiliser une partie de ce matériel pour gérer l'exécution de plusieurs flots d'instructions (parallélisme de flots ou TLP *Thread Level Parallelism*) sur une même puce afin de mieux utiliser le matériel disponible.

Plusieurs solutions architecturales existent pour exploiter ce parallélisme de flots (nous ne présentons ici

que les solutions envisagées/introduites par les constructeurs). Une première solution adoptée par IBM dans le Power4 [39] consiste à intégrer sur la puce un multiprocesseur, dit *multiprocesseur monolithique* (CMP, pour *Chip MultiProcessor*). Dans cette approche, les ressources d'exécution sont partagées statiquement entre les processeurs élémentaires qui exécutent chacun un flot d'instructions. Bien que simple à mettre en œuvre puisque nécessitant uniquement la duplication du matériel, sans accroîssement de la complexité du cœur superscalaire (et donc sans impact négatif sur le temps de cycle), ce partitionnement statique ne permet pas de maximiser l'utilisation des ressources.

La seconde solution utilisée par plusieurs constructeurs (Alpha 21464 de Compaq, XStream Logic [35], Pentium4 [58]) est appelée le plus souvent *multiflot simultané* (SMT pour *Simultaneous MultiThreading*) [84]. Dans cette solution, le processeur peut exécuter en parallèle des instructions provenant de différents flots, mélangeant le parallélisme entre instructions à celui entre flots. Les processus (associés aux flots) peuvent soit être extraits via une parallélisation de l'algorithme et aboutir à la création de plusieurs flots d'instructions indépendants, soit provenir de programmes indépendants (exemple : traitements concurrents d'une image, du signal, d'accès à la mémoire et au réseau). Ainsi, la possibilité de choisir des instructions parmi ces différents flots permet de compenser les contraintes dues aux seules dépendances de données d'un même flot, de recouvrir les latences mémoire et permet d'augmenter le taux de remplissage des unités de traitement. Cette approche a l'avantage d'être plus flexible quand à l'exploitation du parallélisme d'instructions et du parallélisme de flots.

3.3.2 Combiner parallélisme de flots, parallélisme de données et préchargement

Dans ce travail, nous avons exploité les différents types de parallélisme possibles sur la puce du processeur : parallélisme d'instructions, de données et de flots. Par ailleurs, nous avons exploité les caractéristiques des applications multimédia, traitement linéaire d'un grand ensemble de données, et étudié le préchargement matériel sur le cache de second niveau (plus performant que le préchargement sur le cache de premier niveau, voir chapitre précédent). Nous avons considéré l'implémentation matérielle du préchargement, car nous n'avions pas les informations permettant d'implémenter le préchargement logiciel par flot disponible sur les architectures PowerPC i.e. de décrire son comportement fonctionnel.

Notre étude porte donc sur l'association du SMT, du SIMD et du préchargement et plus particulièrement sur l'impact d'une telle architecture sur le bus mémoire. Deux autres études ont été réalisées, mais portent uniquement sur l'utilisation combinée du SMT et du SIMD. La première [64] étude concerne l'exécution d'un programme de décompression MPEG2 sur un processeur SMT avec extensions SIMD (extensions MMX d'Intel). La seconde [20] étude, très récente, compare les performances de deux jeux d'instructions SIMD sur un processeur SMT.

Architecture L'architecture considérée pour ce travail est décrite dans la table 3.1. Le processeur avec extensions multimédia (AltiVec) dispose d'autant d'unités de calculs et de permutations que le processeur sans extension (PowerPC) dispose d'unités flottantes. Pour nos simulations, nous avons utilisé un simulateur superscalaire basé sur ASF [5] (voir annexe).

Au cours de cette étude, les seuls paramètres qui varient sont le rapport de la fréquence processeur sur la fréquence du bus mémoire, noté *rat* (plus le rapport est élevé, plus le débit est faible) et la latence mémoire, notée *lat*. Nous appelons respectivement AV.rat.x.lat.y et PPC.rat.x.lat.y les processeurs AltiVec et PowerPC (i.e. sans extension SIMD) avec un rapport de fréquences x et une latence mémoire de y cycles processeur. Les latences mémoire de 56, 112 et 224 cycles que nous considérons correspondent respectivement à des fréquences processeur de 800 Mhz, 1.6 Ghz et 3.2 Ghz. Nous utilisons des rapports fréquences processeur-bus

Largeur de décodage/émission	8
Entrées dans les stations	20
Entrées dans le ROB	80
Registres de renommage INT	80
Registres de renommage FP	80
Unités INT/INTL/LS/FP/BR	4/2/4/4/2
Taille/assoc. du cache L1	64 Ko / 8
Taille/assoc. du cache L2	1 Mo / 8
Latence du cache L2	7 cycles

TAB. 3.1: Principaux paramètres de l'architecture de base.

de 2, 4 et 7.
Dans cette section, nous présentons uniquement les études réalisées sur un moteur de rendu 3D utilisant PMesa [2] (nous avons réalisé ces mêmes études sur un algorithme normalisé de compression vidéo, DivX). La technique de préchargement matérielle implémentée est une technique de préchargement séquentiel sur le second niveau de cache (voir chapitre pécédent). La ligne préchargée est déterminée par un paramètre appelé *distance de préchargement*. Pour une distance de n, si la ligne i provoque un échec, la ligne $i + n$ est préchargée. Une requête de préchargement est non prioritaire par rapport à une requête normale. Nous présentons dans ce mémoire uniquement les principales conclusions de nos analyses.

Evaluation SMT + SIMD Dans sa version SMT AltiVec, PMesa est sensible à la bande passante et à la latence (voir figures 3.5 et 3.4). Lorsque la bande passante n'est pas trop faible et que la latence ne dépasse pas le potentiel de recouvrement du SMT, la sensibilité accrue à la latence d'AltiVec lui permet de mieux profiter de ce recouvrement. Par contre, le processeur SMT PowerPC peut recouvrir de plus grandes latences. En effet, du fait de sa sensibilité plus grande au débit et à la latence, le processeur SMT AltiVec nécessite de disposer de plus de flots pour tirer parti du SMT et recouvre moins bien des latences élevées que le SMT PowerPC.

Evaluation SMT + SIMD + préchargement Le processeur AltiVec profite plus du préchargement que le processeur PowerPC. En effet, le SMT AltiVec recouvre moins bien la latence que le SMT PowerPC ce qui combiné avec la plus grande sensibilité d'AltiVec à la latence permet au préchargement d'apporter un gain plus important à la version SMT AltiVec. Cependant, les performances d'AltiVec étant plus dépendantes de la bande passante que celles du PowerPC, le bénéfice du préchargement diminue plus rapidement en AltiVec lorsque la bande passante est réduite. Ce phénomène est accentué lorsque la latence est faible à cause de la pression conjuguée sur le bus mémoire du SMT et du préchargement.

L'ajout à notre processeur d'unités SIMD et d'un mécanisme de préchargement de données a permis d'obtenir des gains de performance intéressants, lorsque l'équilibre entre la latence, le débit mémoire et le nombre de flots permet un recouvrement de la latence efficace. Il reste cependant encore une différence de performance significative entre le processeur SMT AltiVec utilisant le préchargement et le même processeur avec cache parfait, comme le montre la figure 3.8, la latence jouant un rôle important dans la limitation des performances. Par ailleurs, l'accroissement de la pression exercée sur le bus mémoire à mesure que le nombre de flots exécutés en parallèle augmente. En effet, même si un recouvrement a lieu entre les flots, il semble

[2] PMesa est une bibliothèque parallèle implémentant l'API OpenGL et basé sur la bibliothèque open source Mesa. Cette bibliothèque est décrite plus précisément en annexe.

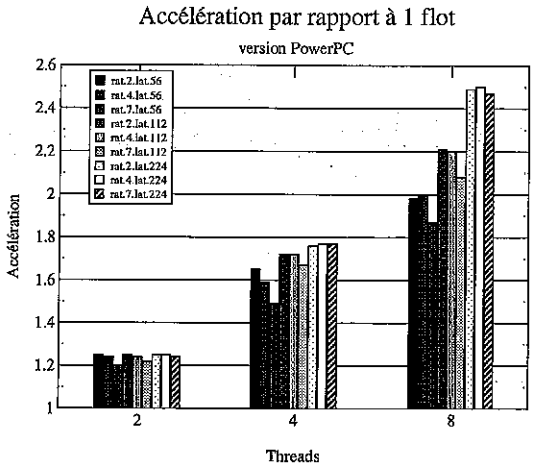

FIG. 3.4: Accélération apportée par le SMT sur le processeur PowerPC (PMesa) par rapport à la version monoprocesseur.

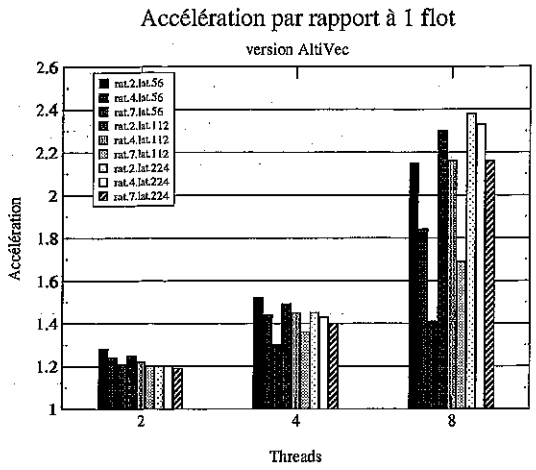

FIG. 3.5: Accélération apportée par le SMT sur le PowerPC avec extensions multimédia (PMesa) par rapport à la version monoprocesseur.

inévitable que certains flots se trouvent simultanément en train d'exécuter des phases d'accès à la mémoire.

Nous allons dans la section suivante proposer une solution matérielle permettant de limiter les effets négatifs des flots en cours d'accès mémoire, solution basée sur la structure de l'application.

3.3.3 Tirer parti de la structure du pipeline graphique

Dans cette section, nous étudions plus particulièrement PMesa et proposons des optimisations permettant d'exploiter la structure de PMesa et d'améliorer les performances du multiflot simultané. En effet, le pipeline graphique (détaillé en annexe) est composé de phases consécutives assez courtes qui ne se comportent pas de la même façon du point de vue des caches. Certaines de ces phases amènent de nouvelles données dans les caches, tandis que d'autres travaillent sur ces données, offrant ainsi de la localité temporelle et un taux de succès élevé dans le cache de premier niveau (voir annexe). Mais cette structure, si elle représente une opportunité de recouvrir les phases d'échecs dans le cache avec les phases de succès, rend PMesa sensible au partage de la fenêtre d'exécution par les flots. En effet, les instructions des flots en échecs peuvent polluer les stations de réservation, nuisant ainsi au bon recouvrement des phases correspondantes. Pour réduire cette pollution, nous avons proposé des mécanismes de contrôle de l'émission des instructions dans les stations de réservation.

Réduire la pollution dans les stations de réservation Dans un processeur SMT, plusieurs flots se partagent un certain nombre de ressources d'exécution et il faut par conséquent déterminer quels flots émettront des instructions vers la fenêtre d'exécution à un instant donné et dans quel ordre. Tullsen *et al.* [83] ont montré que ce choix peut s'avérer déterminant pour les performances et ils concluent que la meilleure heuristique pour assigner les priorités aux flots est de donner une priorité élevée aux flots dont les instructions restent le moins longtemps dans les stations de réservation. Cette politique appelée ICOUNT permet d'éviter qu'un flot ne remplisse les stations de réservation avec des instructions en attente. Bien que ICOUNT réduise effectivement la pollution des stations, elle ne peut éviter que certaines instructions soient émises. En effet, un flot devient peu prioritaire quand il se trouve parmi ceux qui ont le plus d'instructions dans le pipeline, ces instructions restant dans le pipeline jusqu'à la résolution des dépendances de données.

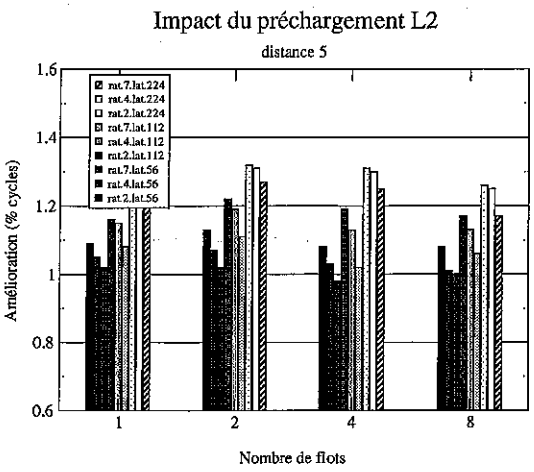

FIG. 3.6: Impact du préchargement sur la version PowerPC (distance de préchargement = 5).

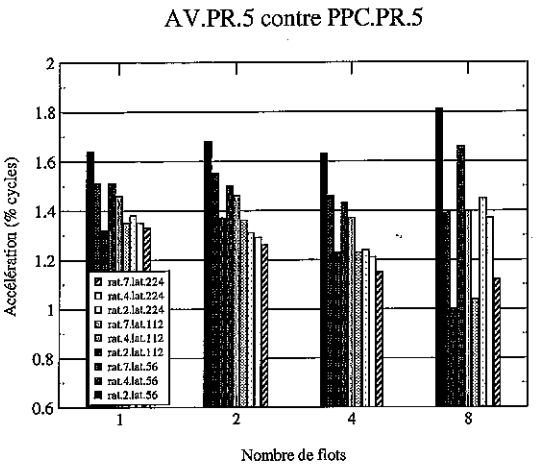

FIG. 3.7: Accélération obtenue par AV.PR.5 sur PPC.PR.5.

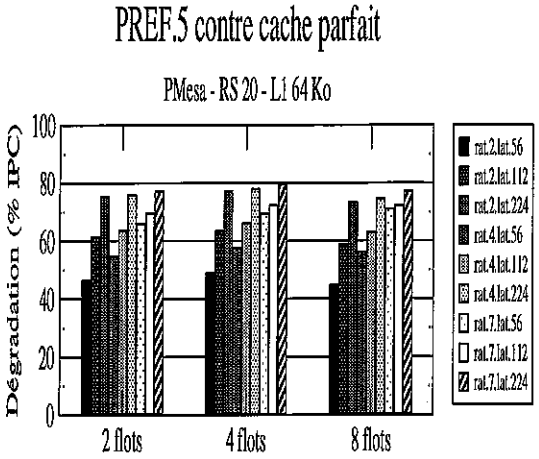

FIG. 3.8: Différence de performance entre une configuration avec un cache parfait et cache + préchargement.

Afin d'éliminer les situations où la fenêtre d'exécution contient une majorité d'instructions provenant de flots en cours d'échecs sur le cache de second niveau notamment (latence élevée), nous proposons une technique permettant de limiter le nombre d'instructions de flots en échec dans les stations de réservation.

La technique de contrôle proposée fonctionne de la manière suivante. Lorsqu'un flot provoque un échec dans le cache de second niveau, il est autorisé à avoir au plus n_i instructions dans la station i (une instruction n'est émise que si le total d'instructions que le flot possède dans la station i ne dépasse pas le seuil n_i de cette station). Cette technique a l'avantage de ne pas bloquer l'émission, mais de la contrôler.

Par contre, si on effectue la détection lorsque l'échec a effectivement lieu, nous risquons de ne pas attaquer complètement le problème de la pollution. En effet, lorsque l'émission des instructions d'un flot sera stoppée ou commencera à être régulée, plusieurs instructions dépendantes auront déjà été émises dans la fenêtre d'exécution de façon non contrôlée. Pour répondre à ce problème d'inertie de la détection déjà présent dans ICOUNT, nous mettons en place *PRED*, un mécanisme de prédiction des échecs L2, qui va permettre de prévoir le comportement d'un chargement et ainsi de pouvoir intervenir sur l'émission du flot en le contrôlant dès la prédiction et avant d'avoir émis d'autres instructions.

Pour mettre en œuvre cette prédiction, nous utilisons une table par contexte. Comme l'adresse de la donnée accédée par un chargement n'est pas connue au moment de la prédiction, chaque table est adressée par les bits de poids faible de l'adresse de l'instruction, à la manière d'un cache. Chaque entrée contient l'étiquette de l'instruction, le nombre de succès dans le cache L2 depuis le dernier échec pour cette instruction et le nombre de succès entre les deux derniers échecs. Pour déterminer la prédiction, on compare les deux nombres stockés. Si ces deux nombres sont égaux, on prédit un échec et le contrôle est activé pour ce flot. Le résultat de la prédiction est placé dans un bit de l'entrée occupée par l'instruction dans le tampon de réordonnancement. Un flot prédit en échec est noté comme tel dans son unité de décodage, et l'adresse de la dernière instruction prédite en échec est conservée. Le flot repassera en mode normal lorsque la donnée de cette instruction sera revenue de la mémoire ou qu'elle aura été lue dans un des caches (on a alors une mauvaise prédiction).

Une modification de l'architecture est nécessaire afin d'effectuer le contrôle des seuils lors de l'étage de sélection qui précède l'étage de décodage. En effet, il faut pouvoir déterminer vers quelle station de réservation l'instruction va être lancée et donc pouvoir effectuer un prédécodage de l'instruction (déterminer le type de l'instruction).

Performance sur le processeur SMT de base Nous évaluons la combinaison de notre mode de contôle avec le mécanisme PRED (notée *dcPRED*) sur le processeur SMT AltiVec de base. Comme dans la section précédente, nous faisons varier la latence et le débit mémoire. Dans les tableaux, r.x.l.y a la même signification que rat.x.lat.y, à savoir un rapport de fréquence processeur-bus de x et une latence mémoire de y. Nous utilisons une table de 64 entrées pour la prédiction. L'amélioration engendrée par *dcPRED* est donnée en pourcentage à la figure 3.9.

dcPRED exhibe des gains de performance quel que soit le nombre de flots. Les gains obtenus augmentent avec la latence, ce qui montre que le potentiel de recouvrement du SMT est accru. Lorsque le débit mémoire est trop faible par rapport à la latence restant à recouvrir, les gains sont quasiment inexistants, comme on peut le constater pour rat.7.lat.56 à 2 et 8 flots. On obtient des gains jusqu'à 14% à 2 flots et 16% à 8 flots, mais les meilleures améliorations enregistrées ont lieu à 4 flots. Comme on l'a vu à la section précédente, dans la version AltiVec à 4 flots, la charge du bus mémoire a plus d'importance que la latence recouverte. Le fait de mieux utiliser le SMT en bridant précisément les flots en cours d'échec L2 permet d'inverser ce rapport. Les gains obtenus atteignent presque 25% pour rat.2.lat.224.

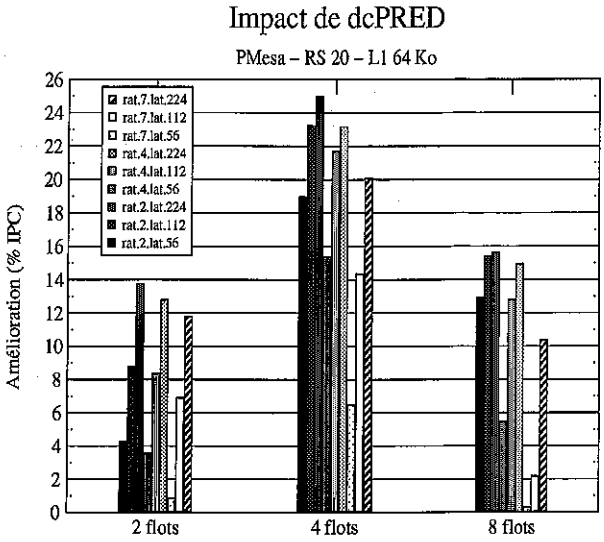

FIG. 3.9: Impact de *dcPRED* sur les performances du processeur SMT AltiVec.

Performance sur le SMT utilisant le préchargement Nous combinons maintenant *dcPRED* et le préchargement L2 avec une distance de préchargement de 5 (ayant donné de bons résultats dans des études préliminaires et permettant d'anticiper davantage les préchargements). L'augmentation de la distance de préchargement n'apporte pas réellement de bénéfices en SMT, car le SMT permet de recouvrir de longues latences. Par contre, le SMT AltiVec recouvre moins bien la latence et est donc plus sensible aux distances de préchargement choisies (ce choix est également dépendant de la latence mémoire considérée). Le processeur correspondant est nommé *dcPRED-PREF.5*. Les performances sont présentées sur les graphes de la figure 3.10.

L'association de *dcPRED* et du préchargement permet d'améliorer les performances obtenues principalement à 2 flots et pour des latences élevées. Pour 4 et 8 flots, l'ajout du préchargement combiné au nombre de flots traités augmente la pression sur le bus et dégrade les performances. Pour ces configurations, il est préférable d'utiliser le mécanisme *dcPRED* sans préchargement.

Références avec résultats détaillés : [52, 55, 54, 53].

3.4 Architecture parallèle

Dans les sections précédentes, nous avons exploité les propriétés des applications multimédia afin d'augmenter les performances (la rapidité d'exécution) de ces applications : traitements identiques (SIMD) et tâches indépendantes (SMT). Dans cette section, nous exploitons également le parallélisme de flots présent dans les applications multimédia et plus particulièrement dans les applications de synthèse d'images, non plus au sein d'un même processeur, mais sur un réseau de processeurs généralistes. Réaliser une machine parallèle à partir de processeurs généralistes n'est pas une idée nouvelle et beaucoup de travaux existent sur la mise en place et l'évaluation de clusters de PC, la spécificité de ce travail est d'étudier et d'optimiser ce type d'architecture pour les applications multimédia et plus précisément pour les applications de synthèse

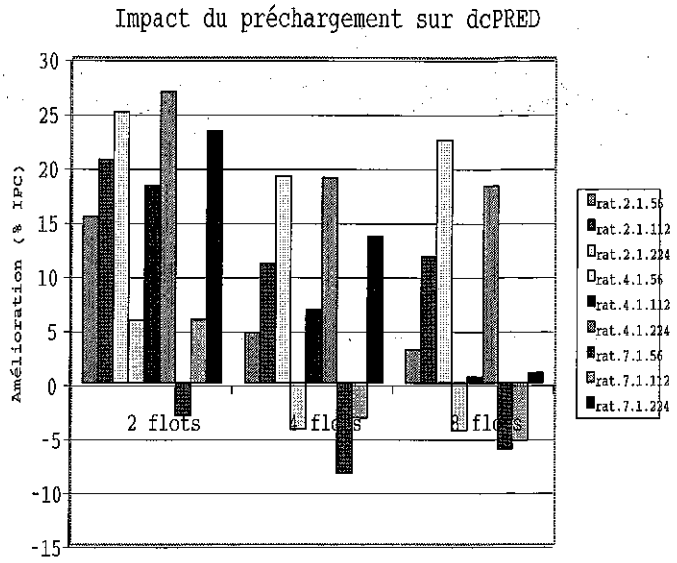

FIG. 3.10: Impact du préchargement sur les performances de *dcPRED*.

d'images.

Un des principaux problèmes des architectures parallèles est que la parallélisation peut se traduire par une dégradation des performances des composants élémentaires. Pour éviter ce problème, il faut notamment éviter une mauvaise utilisation des caches qui se traduirait par une augmentation du débit de données nécessaire. Ainsi, le choix de l'algorithme de distribution des tâches a un impact sur la performance, car il va déterminer l'utilisation et donc l'efficacité des caches. Mais il ne faut pas que l'utilisation des caches pénalise l'équilibre de la charge. En effet, une tâche trop petite peut engendrer des surcoûts liés aux communications et aux partages de blocs de données entre processeurs, une tâche trop grande peut engendrer un mauvais équilibre de la charge. Ces problèmes de parallélisation apparaissent aux différentes étapes du pipeline graphique : étape de géométrie et étape de rasterisation.

Dans cette section, nous nous sommes intéressés à l'étape de rasterisation et plus particulièrement à l'application de textures et aux problèmes posés par la parallélisation sur l'utilisation des caches de textures.

3.4.1 Principe

Plusieurs types d'architectures parallèles existent : réseau de PC (Ethernet, haut de gamme, spécialisé), multiprocesseurs à mémoire partagée et multiprocesseur *on-chip* et diffèrent notamment par leur performance et leur coût. Elles posent un certain nombre de problèmes classiques : problème de communications (partage des données entre sous-tâches distribuées sur différents processeurs), problème de dimensionnement (choix du nombre de processeurs élémentaires), problème de programmation (disponibilité du logiciel) et problème de hiérarchie mémoire (ensemble de données important, partage du bus mémoire, partage des données).

Les traitements en synthèse d'image sont réalisés suivant un pipeline, appelé pipeline graphique et composé de deux étapes principales, une étape réalisant les transformations géomètriques ayant pour but de déterminer diverses informations sur chaque sommet afin de permettre l'affichage du triangle correspondant par l'étape de rasterisation. Au cours de cette deuxième étape, on va dessiner le contenu de chaque triangle et calculer ce qu'il faut afficher à l'écran pour chaque pixel. Les données à distribuer pour les transformations

géométriques sont les triangles, les données à distribuer pour la rasterisation sont les pixels. Les différents traitements réalisés dans ces étapes sont détaillés en annexe.

Ainsi, le parallélisme de données diffère d'une étape à l'autre et nécessite un convertisseur entre les deux ensembles de données afin de trier les triangles selon leur position dans l'espace. Cette étape de tri peut être insérée à différents endroits du pipeline et détermine la classification proposée par Molnar [61] (voir figure 3.11).

- Architecture *Sort-first* : le tri est réalisé durant l'étape de géométrie. Tout le pipeline est traité par la même unité responsable de toutes les étapes du dessin pour une partie donnée de l'image. Ceci implique de connaître *a priori* la position exacte d'un point à l'écran avant toute transformation. Pour mettre en oeuvre une telle architecture, il faut soit rajouter une étape de prétraitement pour déterminer sur quel processeur doit s'exécuter le pipeline, soit réaliser un certain tri pour les transformations géométriques, puis redistribuer avant la rasterisation les triangles dont la position a été mal évaluée.
- Architecture *Sort-middle* : le tri est réalisé entre l'étape de géométrie et l'étape de rasterisation. Les triangles sont distribués aux différents processeurs responsables des transformations géométriques, puis les données résultantes sont entièrement redistribuées aux différents processeurs faisant la rasterisation exclusive d'une partie de l'écran. Cette architecture est la plus utilisée, on la retrouve notamment dans les machines haut de gamme de Silicon Graphics [1, 62].
- Architecture *Sort-last* : le tri est réalisé durant l'étape de rasterisation. Les triangles sont distribués de façon totalement dynamique aux pipelines graphiques et sont transformés et dessinés indépendamment sur leur pipeline dans une mémoire écran locale. Ensuite, les mémoires écran des différents pipelines sont fusionnées. La machine PixelFlow [31] utilise cette forme de parallélisme.

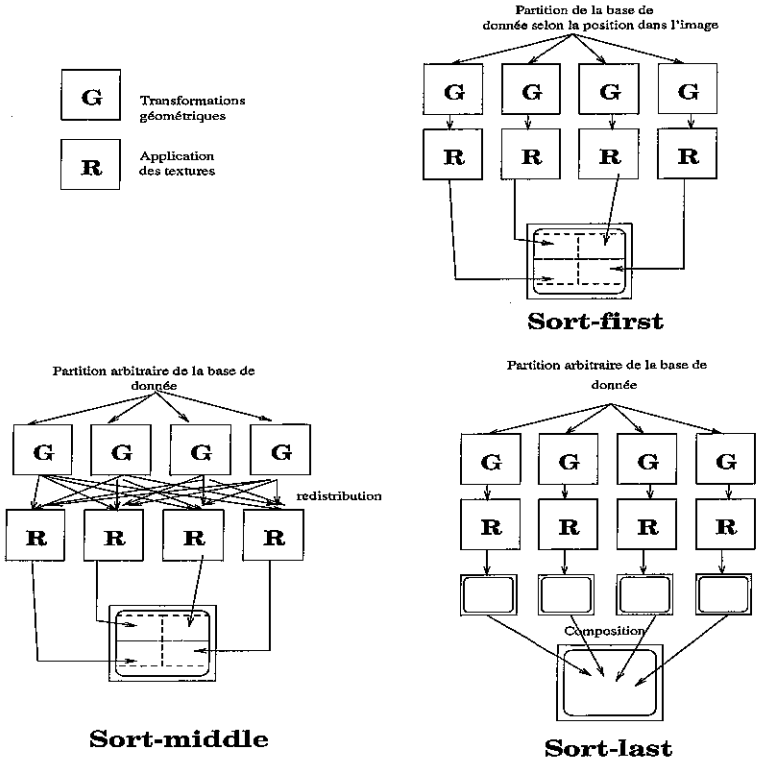

FIG. 3.11: Classification de Molnar.

Le choix du tri affecte donc la distribution des données sur les processeurs réalisant la géométrie et sur ceux

réalisant la rasterisation.

Parallélisme et géométrie. J. Sebot et A. Vartanian ont parallélisé une bibliothèque effectuant les transformations géométriques en OpenGL. Ce prototype, appelé PMesa[3] est basé sur la bibliothèque Open Source Mesa[4] et permet d'accélérer un certain nombres d'applications OpenGL. D'autres travaux sur la parallélisation de Mesa ont été réalisés ces dernières années. Ces différents travaux montrent qu'il est possible d'implémenter efficacement un moteur de transformations géométriques sur un multiprocesseur à base de processeurs standards et ont permis d'identifier les principaux goulets d'étranglement liés au coût de l'ordonnancement des flots et au comportement de la hiérarchie mémoire.

Parallélisme et rasterisation. Du point de vue de l'application de texture, la classification des architectures parallèles de Molnar se ramène à deux cas, répartition par image (*sort-first* ou *sort-middle*) et répartition par triangle (*sort-last*). La répartition par image consiste à distribuer différentes parties de l'image à chaque processeur qui essaient de dessiner de façon synchrone l'intersection des triangles avec les pixels dont ils sont responsables. La répartition par triangle consiste à répartir les triangles sur tous les processeurs qui peuvent dessiner sur toute l'image. La répartition par image est plus simple à implémenter et il n'y a pas de problèmes de cohérence entre les processeurs pour la visibilité puisque chacun gère son propre *z-buffer*[5]. Cette répartition se fait en découpant l'image en blocs. La répartition par triangle a *a priori* l'avantage d'utiliser un équilibrage dynamique de la charge, mais elle est plus lourde à implémenter, car il faut gérer le problème de la visibilité, soit en triant les triangles avant l'application des textures, soit en réalisant le test de visibilité dans une phase postérieure.

3.4.2 Caractéristiques de l'application de textures

Pour comprendre l'impact de la distribution sur la localité des données dans l'application de texture, nous détaillons les algorithmes utilisés et leur comportement vis à vis des accès aux données.

Les textures permettent d'améliorer le réalisme des images sans augmenter la complexité géométrique des scènes. Une texture est une image ou un morceau d'image plaquée sur un polygone. Le pipeline d'application de texture effectue le calcul des pentes des triangles et l'application des textures sur chaque pixel. Un certain nombre d'algorithmes sont mis en oeuvre afin d'appliquer les textures et d'en améliorer le résultat visuel.

Algorithmes L'application de textures consistant à dessiner une image sur le triangle, il suffit de connaître la position de chaque sommet dans l'image pour interpoler les positions de chaque point du triangle. Un des problèmes pour appliquer une texture en un point est celui de la différence de résolution entre l'image de la texture et celle de l'écran. Si l'écran a une résolution plus fine que la texture, on applique à plusieurs points de l'écran le même point dans la texture. Cependant, si la résolution de l'écran est moins fine, on est obligé de calculer la moyenne des couleurs de tous les points de la texture qui devraient s'afficher en ce point. Afin d'éviter d'avoir à faire ce calcul à chaque fois, une pyramide de textures de tailles de plus en plus petites obtenues à partir de la texture originale est conservée en mémoire (uniquement les puissances de 2). Cette technique est appelée *Mip Mapping* [96] (*Multum In Parvo*, i.e., beaucoup de choses dans un espace très petit). Les textures sont alors précalculées au départ et occupent 33% d'espace supplémentaire.

Par ailleurs, lorsque l'on veut afficher une texture sur une surface déformée, il se produit un phénomène de crénelage [67]. C'est le cas lorsque l'on affiche une image sur un écran constitué de pixels qui ne coïncident

[3] http://pmesa.sourceforge.net

[4] http://www.mesa3d.org

[5] Ce tampon permet de mémoriser la profondeur de chaque pixel à afficher. Lorsque l'on veut afficher 1 pixel, on s'assure qu'il se trouve devant celui préalablement affiché.

pas avec ceux de l'image initiale. Pour limiter ce phénomène, on applique le filtrage bilinéaire qui consiste à calculer pour chaque point la moyenne des 4 points dans la texture les plus proches du point affiché.

D'autre part, l'utilisation du *Mip Mapping* sur les objets en mouvement peut poser problème. En effet, si un objet se rapproche de l'observateur, le rapport entre sa résolution à l'écran et celle de la texture originale peut changer et imposer un changement de *map*. Pour éviter que ce changement soit visible, les processeurs utilisent le filtrage trilinéaire qui consiste à utiliser les 4 pixels des deux *mip map* ayant la plus proche résolution ce qui requiert 8 texels [6] pour le rendu d'un pixel.

Ces techniques impliquent donc lors de l'affichage d'un point à l'écran, l'accès à 8 points de la mémoire contenant la texture (à multiplier par le nombre de textures qu'on affiche en ce point). Le débit d'accès aux textures est ainsi 8 fois supérieur au débit des pixels ce qui en fait un goulet d'étranglement pour la réalisation des circuits accélérateurs 3D.

Accès aux données Nous allons maintenant décrire ces algorithmes vis à vis des accès aux données.

Le *mip-mapping* permet de diminuer le nombre de calculs lors de l'application des textures en utilisant des textures à la bonne résolution, ainsi on utilise les points voisins dans la texture pour dessiner les points voisins dans l'image. Ceci augmente la localité spatiale pour l'accès aux points de la texture. Lors du filtrage, le fait d'utiliser pour chaque point plusieurs pixels dans la texture a un impact important sur la localité des textures puisque entre deux points voisins, on partage plusieurs points dans la texture. Quand un pixel est affiché, les 4 texels du précédent pixel dessiné sont réutilisés, ainsi que lors de l'affichage de la ligne suivante. Si une *map* a la bonne résolution (par rapport à l'écran), un texel est réutilisé 4 fois. Le taux de réutilisation est différent selon la résolution de la *map* utilisée. Si elle est plus large que la résolution de l'écran, les texels de la même ligne de cache ne sont pas réutilisés, si elles est plus petite, les texels seront réutilisés plus de 4 fois. Par ailleurs, la localité temporelle peut être amplifiée lorsque l'on utilise pour certains objets de petites textures répétées (rendu d'un objet simple comme un matériau).

Comme il existe un décalage entre le débit interne d'un circuit de traitement et celui de la mémoire externe et que les accès à la mémoire exhibent des propriétés de localité, il est judicieux d'utiliser une hiérarchie mémoire et des caches de textures pour conserver les texels utilisés le plus souvent[7]. Différentes études sur les caches de texture ont été réalisées. Hakura et Gupta ont montré dans [37] que le cache de texture était très performant en monoprocesseur. On trouve également dans [40] une étude qui montre que le préchargement des textures permet de recouvrir la latence des accès à la mémoire lors des défauts de cache. Enfin, l'étude de Cox dans [21] montre qu'un second niveau de cache permet d'utiliser la localité des textures entre les images successives.

3.4.3 Caractéristiques du pipeline d'application de textures

La rasterisation est prise en charge par un composant/circuit spécialisé (standard ; carte accélératrice 3D). Le modèle de pipeline considéré est présenté à la figure 3.12. Une première partie effectue le calcul des pentes des triangles et est capable de préparer un triangle tous les 25 cycles. Une seconde partie effectue l'application des textures sur chaque pixel et est capable de dessiner un pixel par cycle (calculs en entier 32 bits). Le pipeline contient un tampon entre l'accès aux étiquettes du cache et le retour des données qui permet de recouvrir la latence des échecs aux caches (masquer les irrégularités de débit). Un tel composant peut débiter un pixel par cycle quelle que soit la latence de la mémoire. Cependant, il est nécessaire que le débit du bus puisse absorber les échecs aux caches. L'impact de la hiérarchie mémoire sur les performances

[6] Elément de base d'une texture, analogue au pixel d'une image.

[7] Les accélérateurs 3D de PC disponibles sur le marché sont équipés de caches de texture.

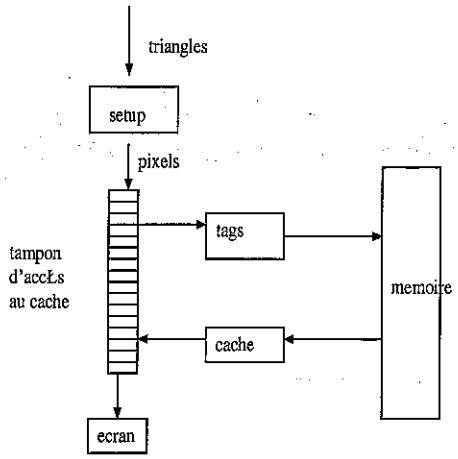

FIG. 3.12: Modèle du pipeline d'application de textures.

Débit en texels par pixel	1/2	1	2	4
Largeur du bus	64	128	128	256
Fréquence interne / fréquence externe	4	4	2	2

TAB. 3.2: Caractéristiques des bus considérés.

dépend de la saturation du bus, et donc de l'efficacité du cache et de la largeur du bus. Les largeurs des bus utilisées sont présentées dans le tableau 3.2.

3.4.4 Améliorer les performances du parallélisme images pour l'application de textures

Dans cette sous-section, nous évaluons l'impact de l'utilisation de caches de textures dans une machine parallèle pour la synthèse d'images utilisant le parallélisme image pour l'étape d'application des textures (architectures *sort-first* et *sort-middle*). Nous nous sommes particulièrement intéressés à l'impact de la distribution sur l'équilibre de la charge et sur la localité des données, impact en terme de performance.

Architecture Les transformations géométriques sont effectuées par différents processeurs utilisant soit le parallélisme image, soit le parallélisme triangle. A la fin de cette étape, les informations pour déterminer à quels processeurs de textures le triangle est destiné sont disponibles. Les triangles transformés sont alors envoyés aux différents processeurs responsables de l'application des textures. Chaque composant possède son cache de texture et sa mémoire de texture privée. Les textures sont donc répliquées dans toutes les mémoires. Dans le cas de l'architecture totalement *sort-first*, les triangles sont déjà sur des processeurs de transformations géométriques associés à une partie de l'image et comme il est logique de conserver la même distribution pour l'application des textures, chaque processeur de géométrie enverra les triangles sur un bus privé au composant associé réalisant l'application de texture. Dans le cas de l'architecture *sort-middle*, les triangles peuvent être tous diffusés sur un bus commun comme dans l'Infinite Reality [62] ou bien n'être communiqués que du processeur producteur aux différents processeurs consommateurs à travers un réseau, option qui n'est intéressante que si un triangle à dessiner est partagé par peu de processeurs. Ensuite, le

processeur d'application de texture reçoit soit l'ensemble des triangles de la scène (dans ce cas, il possède un mécanisme pour éliminer rapidement tous ceux qui ne tombent pas dans la zone écran qu'il doit traiter), soit uniquement les triangles le concernant et il les parcourt (par ligne ou par bloc). A la fin du dessin de la scène, les processeurs envoient leurs zones traitées au circuit de commande de l'écran.

Il est à noter que ce pipeline dessine les triangles rigoureusement dans l'ordre de soumission ce qui est indispensable pour respecter les spécifications d'OpenGL et gérer correctement la transparence. Ainsi, on retrouve ce type d'architecture dans la plupart des machines parallèles actuelles comme les machines haut de gamme de SGI (Reality Engine et Infinite Reality [1, 62]) ou encore dans des produits grand public[8].

Distribution Différentes distributions des zones de dessin sur les processeurs sont *a priori* envisageables, mais nous n'avons considéré que les distributions implémentables sur des composants standards, i.e., faibles coûts. Nous avons considéré des distributions fixes pour lesquelles chaque processeur dessine toujours la même zone, les zones étant constituées de blocs entrelacés composés soit de suites de lignes, soit de suites de colonnes, soit de blocs carrés et avons fait varier la taille des blocs. Nous avons par ailleurs considéré le coût induit par la préparation de triangles supplémentaires. Un pixel n'est dessiné que par le processeur exclusivement responsable de la zone à laquelle il appartient. En revanche, certains triangles correspondent à plusieurs zones et vont donc devoir être préparés sur plusieurs processeurs. Le surcoût associé à cette préparation est plus élevé que dans un monoprocesseur, car le nombre de pixels effectivement dessinés sur le processeur pour un même triangle est plus petit.

Distribution et équilibrage de charge Afin d'évaluer l'impact de la taille et de la forme de la distribution sur l'équilibre de la charge, nous avons simulé une architecture parallèle avec des caches de textures parfaits.

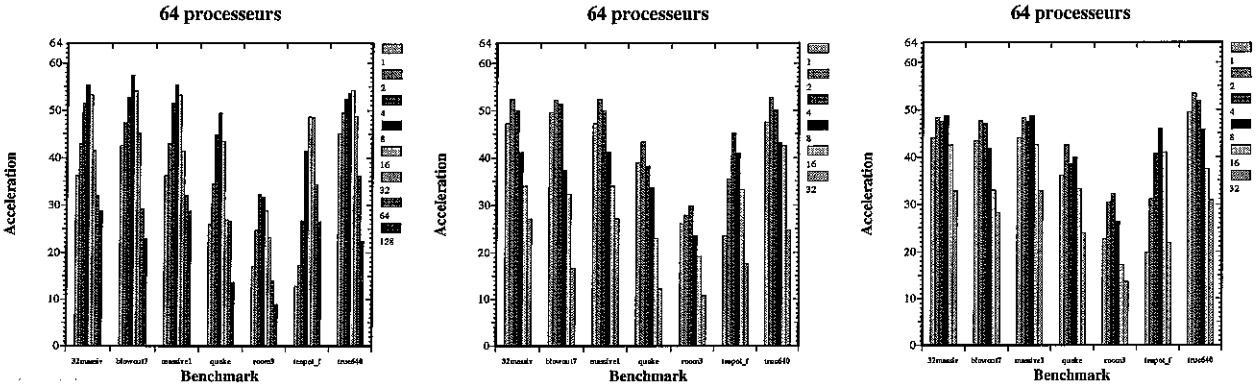

FIG. 3.13: Accélération en fonction de la taille du bloc, ou du nombre de lignes ou de colonnes dans le bloc avec un cache parfait.

Les courbes de la figure 3.13 présentent l'accélération avec un cache parfait des distributions en fonction de la taille des blocs ou du nombre de lignes ou colonnes (pour 64 processeurs). Si les blocs sont trop petits, le surcoût de traitement des triangles supplémentaires dégrade les performances. Par ailleurs, si les blocs sont trop grands, la charge est mal équilibrée.

Pour les différents programmes, l'accélération est relativement importante (3,7 pour 4 processeurs, 14 pour 16 processeurs et 50 pour 64 processeurs) si la taille du bloc est convenablement choisie. Pour un

[8]La 3dfx Voodoo2 fut le premier système 3D bon marché pouvant être utilisé en multiprocesseur. On peut utiliser deux cartes dans un PC pour doubler la résolution, chaque carte se chargeant de dessiner une ligne sur 2.

placement par bloc, il faut considérer des blocs d'une largeur de 4 à 16 pixels. Pour un placement par ligne ou par colonne, il faut considérer des blocs de 2 ou 4 lignes ou colonnes. Pour ces deux types de distribution, le maximum de performance n'est pas obtenu pour la même taille de bloc lorsque l'on fait varier le nombre de processeurs. En effet, pour un placement par ligne, le maximum pour 4 processeurs est à 16 alors que celui pour 64 processeurs est à 2.

Le placement par bloc offre la meilleure accélération (entre 50 et 60 pour 64 processeurs), suivi du placement par colonne et du placement par ligne qui est légèrement moins bon (entre 40 et 50 pour 64 processeurs). Une seule scène (*room3*) exhibe de mauvaises performances, car elle comporte énormément de petits triangles et est donc pénalisée par le surcoût lié au partage de ces petits triangles.

Malgré les problèmes d'équilibre de charge et de surcoût de traitement des triangles partagés, en choisissant correctement la taille des blocs, on arrive à réaliser une machine parallèle avec de bonnes performances. Nous allons maintenant étudier ces mêmes architectures avec des caches de textures non parfaits (réalistes) afin de déterminer quel est l'impact de la parallélisation sur la localité des données.

Distribution et localité Nous avons simulé une machine avec des caches privés dont les paramètres sont ceux qui donnent les meilleures performances en monoprocesseur (déterminés lors d'études préliminaires), soit des caches de taille 16 Koctets, d'associativité 4 et un placement des textures par bloc de 4 x 4 texels.

Nous avons simulé notre machine parallèle avec 4 débits de bus mémoire (1/2, 1, 2 et 4 texels par pixel), avec différentes tailles et formes de blocs. Pour chaque débit de bus, nous avons mesuré les accélérations des machines multiprocesseurs par rapport aux configurations monoprocesseurs avec le même bus, ceci afin de savoir s'il est intéressant de bâtir une machine parallèle pour la 3D à partir de ce type de composants.

L'accélération d'une telle machine dépend, pour une taille et une forme de bloc données, d'une part du ralentissement induit par la perte de localité du cache de textures et d'autre part par l'équilibre de la charge. Le paragraphe précédent a montré que la performance de la machine dépendait de la forme et de la taille du bloc. En intégrant l'impact de la localité, plus les blocs sont petits (mais pas trop pour éviter les limitations liées au traitement des triangles supplémentaires), plus l'équilibre de la charge est *a priori* bon, mais moins bonne est la localité. Par ailleurs, la distribution des parties de l'image qui ont beaucoup de défauts de caches peut être déséquilibrée, et peut aggraver ou au contraire réduire le déséquilibre de charge lié à la distribution des pixels.

Performance Nos simulations permettent d'analyser ces phénomènes et de savoir quels sont les compromis à réaliser en fonction du programme, du nombre de processeurs et du débit du bus. On constate que toutes les simulations présentent un maximum d'accélération pour une valeur de taille de bloc. Les tailles de blocs les plus grandes et les plus petites ont toujours de moins bonnes performances. Pour certaines courbes, une légère variation des valeurs optimales génère une dégradation importante alors que d'autres sont plus tolérantes à une variation autour de l'optimal. Les configurations avec 4 processeurs ont cependant un comportement différent des autres : le maximum est presque toujours obtenu avec la taille de bloc la plus grande, car l'impact du déséquilibre de charge sur les performances est très faible par rapport au problème du cache qui survient même avec peu de processeurs. Pour les autres configurations, le maximum est une valeur intermédiaire :

- *Bus 1/2 et 1 :* pour une répartition par bloc carré, le maximum est toujours obtenu pour une largeur de 16 pixels. Pour des blocs de lignes (colonnes), le maximum varie avec le nombre de processeurs et passe respectivement de 16 à 8 et à 4 (32, 16, 8) quand le nombre de processeurs est de 4, 16 et 64.
- *Bus 2 et 4 :* pour une répartition par bloc carré, le maximum est de 16 pixels pour 4 et 16 processeurs et de 8 pour 64 processeurs. Pour des blocs de lignes (colonnes), le maximum varie avec le nombre de processeurs et passe de 8, 8 à 4 (16, 8, 8) quand le nombre de processeurs est de 4, 16 et 64.

La localité variant peu avec le nombre de processeurs, les mêmes comportements qu'avec des caches parfaits se retrouvent, soit un maximum stable pour les blocs carrés et un maximum qui va en diminuant pour les placements par ligne et colonne. Cependant, pour chacune des distributions, ce maximum est décalé vers des tailles 2 à 4 fois supérieures à cause des caches de textures. Ce décalage est d'autant plus important que le bus est encombré. Ainsi, pour le placement par bloc, la maximum est autour de 16 pour les bus 1/2 et 1 et de 8 pour les bus 2 et 4 qui souffrent moins des pertes de localité et ont donc un comportement plus proche de celui du cache parfait. On constate que les blocs doivent être 2 fois supérieurs pour la distribution colonne par rapport à la distribution ligne. En effet, les blocs colonnes comportant moins de pixels, ils sont moins sensibles au déséquilibre de charge à largeur égale et le placement par colonne est plus sensible que le placement par ligne aux pertes de localité surtout dans les périodes où le bus est très encombré.

Dans cette section, nous avons donc évalué de façon systématique les performances de machines parallèles pour la synthèse d'images utilisant le parallélisme image avec des caches de textures. Nous avons évalué séparément l'impact du déséquilibre de charge et celui des caches de textures afin d'avoir une idée précise des deux phénomènes qui risquent d'avoir un impact sur l'accélération de telles machines.

Nous pouvons conclure que construire ce type de machine permet d'obtenir de bonnes performances : au minimum des accélérations de 3,5, 12 et 30 pour 4, 16 et 64 processeurs, ceci à condition de choisir judicieusement la taille de blocs. Cependant, il reste une marge d'amélioration pour ces architectures (voir figure 3.13), et trouver un système d'équilibrage de charge suffisamment performant qui n'augmente pas trop le coût de l'architecture est un problème ouvert.

3.4.5 Améliorer les performances du parallélisme triangle pour l'application de textures

Après avoir exploré les architectures utilisant le parallélisme image pour l'application des textures, nous allons maintenant étudier l'autre classe de parallélisme pour cette étape, le parallélisme objet.

Architecture Les objets à dessiner sont d'abord distribués de façon totalement dynamique aux pipelines avec n'importe quel algorithme d'équilibrage de charge, généralement le premier pipeline libre dessine le triangle. Les objets sont alors transformés et dessinés indépendamment sur leur pipeline dans une mémoire écran locale. Quand tous les objets de la scène ont été dessinés, les mémoires écran des différents pipelines sont alors toutes fusionnées de façon à déterminer quel pixel est effectivement visible.

Ces architectures présentent deux avantages importants par rapport à celles utilisant le parallélisme image. Premièrement, lorsque l'on augmente le nombre de processeurs des autres architectures afin d'avoir une machine dessinant plus d'objets, il faut pouvoir disposer d'un bus capable de diffuser tous ces triangles. Pour cette machine, le nombre de processeurs n'est pas limité par un tel réseau. Deuxièmement, l'équilibrage dynamique de la charge garantit une utilisation optimale de la machine.

Mais ces architectures sont aussi reconnues pour avoir un certain nombre d'inconvénients, notamment le fait que les triangles soient dessinés indépendamment par les processeurs ne garantit plus l'ordre total, ce qui en particulier rend difficile la fabrication de telles machines respectant rigoureusement l'API OpenGL.

La figure 3.14 présente les performances d'une telle architecture[9]. Nous avons simulé l'algorithme de distribution qui consiste, à chaque fois qu'un processeur est libre, à demander au système de distribution le triangle suivant. Avec un tel algorithme, on peut espérer obtenir un bon équilibre de la charge. On constate que les performances sont excellentes avec 4 processeurs, bonnes sauf pour un benchmark (*blowout775*)

[9] Nous n'avons pas simulé le système effectuant la fusion des images, car le fonctionnement de celui-ci est indépendant de ce qui se passe lors du dessin des triangles, il suffit d'avoir un bus avec un débit suffisant.

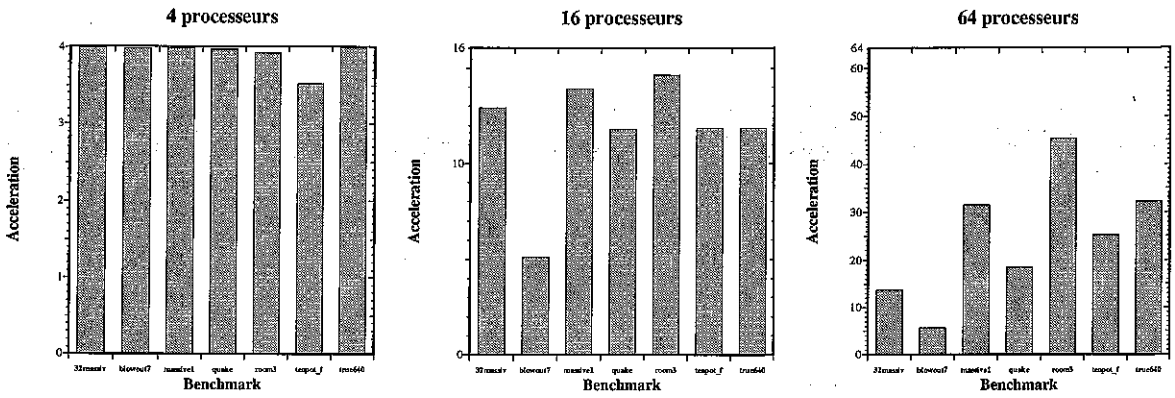

FIG. 3.14: Accélération de l'architecture *sort-last* avec un cache de 16Koctets et un bus débitant 1 texel/pixel.

avec 16 processeurs, mais très mauvaises avec 64 processeurs, en particulier 3 benchmarks sur 7 ont une accélération inférieure à 20. Ces mauvaises performances sont dues d'une part à la diminution de la localité et d'autre part à un déséquilibre de la charge.

Distribution et localité Dans cette architecture, les triangles sont distribués, indépendamment de leur position à l'écran, aux pipelines qui peuvent alors dessiner n'importe où dans l'écran. Ainsi, si deux triangles sont traités sur différents pipelines et si des texels sont réutilisés entre les deux triangles, il y aura moins de localité sur la machine parallèle que sur la machine séquentielle. Nous avons donc proposé deux mécanismes pour améliorer les performances de ces architectures : l'envoi par rafales et le découpage des triangles.

Envoi par rafales L'idée de notre mécanisme est basée sur la constation que les triangles appartenant au même objet sont très souvent envoyés les uns à la suite des autres. Cette constation est directement liée au fonctionnemment de l'application :

- lorsqu'un objet courbe est découpé en facettes, les triangles résultants sont naturellement envoyés les uns après les autres au système,
- lorsque l'application utilise des tableaux (*vertex arrays*) stockant les arêtes de l'objet, le tableau est envoyé tel quel pour être transformé afin que les arêtes partagées entre différents triangles ne soient transformées qu'une seule fois,
- lorsque l'application essaie d'améliorer la localité et de diminuer le nombre de changements de modes, il y a regroupement des objets utilisant les mêmes modes et en particulier les mêmes textures.

Nous allons exploiter ces propriétés afin d'améliorer la localité en envoyant une rafale de triangles au lieu d'un seul triangle, la taille des rafales étant un élément important de cette étude.

Un algorithme simple consiste à fixer la taille de la rafale à un nombre de triangles constant. Ainsi, plus la rafale est longue, plus un triangle a de chances de trouver ses voisins dans une même rafale et ainsi moins on aura de lignes de caches distribuées sur différents processeurs.

La figure 3.15 illustre l'efficacité de ce mécanisme pour un benchmark (*blowout775*). On voit qu'en choisissant une taille optimale de rafale, on arrive à réduire le phénomène de perte de localité. Il faut cependant éviter d'avoir des rafales trop longues, car on voit que la localité moyenne diminue de nouveau si la taille de la rafale dépasse 100. La raison de cette augmentation est le déséquilibre de la répartition des défauts de caches apporté par ces rafales.

Par ailleurs, ce mécanisme peut aggraver le déséquilibre de la charge. En effet, les processeurs recevant de très grands triangles risquent de prendre plus de temps que tous les autres pour les dessiner. Le mécanisme des

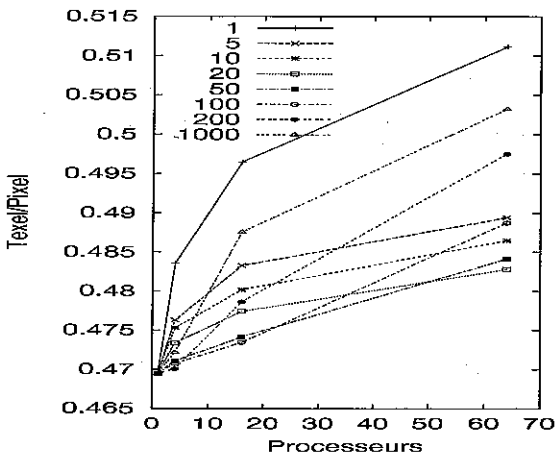

FIG. 3.15: Evolution de la localité en fonction de la taille du paquet de triangles (en triangles, puis en pixels) et du nombre de processeurs.

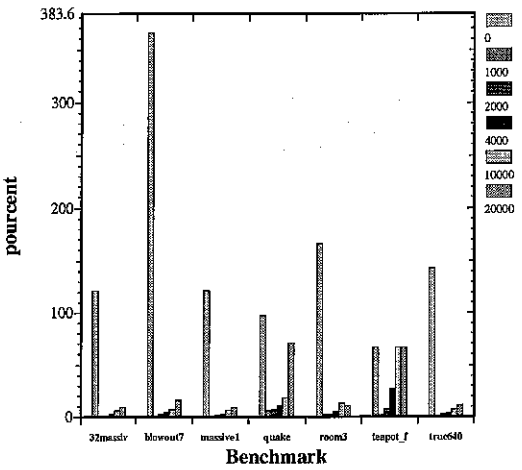

FIG. 3.16: Evolution de la charge supplémentaire en fonction du seuil de découpage (0=pas de découpage), pour 64 processeurs.

rafales de taille fixe risque donc d'aggraver ce phénomène, car on peut avoir dans une rafale plusieurs grands triangles. Le processeur qui aura une telle rafale va donc faire attendre la machine encore plus longtemps. Pour éviter que ce phénomène ne se produise, nous avons testé un mécanisme de "vol de tâches" : quand tous les triangles de la scène ont été envoyés, un processeur qui a fini le dessin de ses triangles va récupérer un triangle à dessiner à un processeur qui a plusieurs triangles en attente. Mais même avec ce mécanisme le problème de l'équilibre de la charge est aggravé par l'utilisation des rafales.

Nous avons étudié un deuxième algorithme de génération de rafales qui utilise des rafales de taille variable en nombre de triangles, mais cet algorithme n'a pas permis de résoudre le problème de la distribution.

Découpage des triangles L'origine du déséquilibre de charge vient de la taille des triangles. En effet, la granularité de la tâche qui est associée aux grands triangles est trop importante pour qu'il soit possible d'équilibrer la charge du système. Il faut donc découper ces tâches en découpant les triangles en triangles plus petits avant de les distribuer aux pipelines. Le mécanisme de découpage que nous avons implémenté est le suivant. Au moment de la distribution, on calcule la taille approximative de chaque triangle. On définit une taille maximale des triangles autorisée dans le pipeline. Tout triangle dont la taille est supérieure à ce seuil est découpé en deux. Ceci se fait en rajoutant un sommet au milieu d'un des côtés et en calculant ses paramètres (texture, luminosité) par interpolation linéaire. On verra plus tard différents algorithmes de choix des côtés. On se retrouve ainsi avec deux triangles. Si l'un des deux triangles est toujours trop grand, on itère le processus.

La figure 3.16 montre l'impact du seuil de découpage sur le déséquilibre de la charge. On voit que ce mécanisme est très efficace pour résoudre le problème. Pour tous les benchmarks, sauf *quake* et *teapot.full*, limiter les triangles à 20000 points suffit à résoudre le problème. Pour les deux autres benchmarks, il est nécessaire de limiter les triangles à une plus petite taille. Dans tous les cas, un seuil de 1000 triangles génère un équilibre de charge parfait à 5% près.

Il faut maintenant mesurer l'impact de ce découpage sur la hiérarchie mémoire monoprocesseur qui est l'autre facteur de performance considéré. Deux phénomènes peuvent alors se produire. On peut avoir un gain de localité lié au fait que dans un triangle plus petit, les lignes peuvent être plus courtes et donc le temps entre

deux utilisations de texels communs à deux lignes peut être plus petit. Ainsi, on diminue l'effet des conflits et des problèmes de capacités qui peuvent enlever du cache des données entre deux utilisations. Cependant, on peut avoir une perte de localité liée au fait que le partage des lignes de caches se fait maintenant entre deux triangles (au lieu d'un seul). On aura besoin de parcourir entièrement le premier triangle avant de réutiliser la ligne ce qui peut entraîner davantage de conflits.

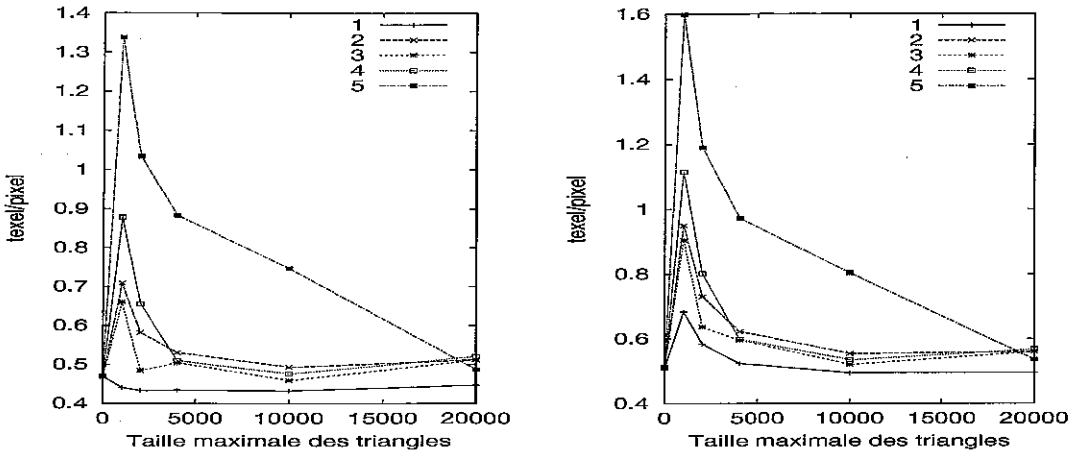

FIG. 3.17: Evolution de la localité en fonction du seuil de découpage et de l'algorithme de découpage, en monoprocesseur et avec 64 processeurs.

Les mesures effectuées ont montré que le second phénomène est prépondérant. Afin d'en réduire l'impact, nous avons évalué différentes façons de découper les triangles. Pour diviser un triangle (de sommets A_1,A_2 et A_3), on choisit un des côtés, on rajoute un point au milieu et il devient le sommet des deux triangles ainsi générés. Mais, il faut choisir le côté à découper parmi les 3 possibles $[A_1A_2], [A_2A_3], [A_3A_1]$ et donc déterminer un critère de choix. Nous avons considéré différents critères :

1. choisir le côté qui a la plus grande hauteur : $\max_{i=1,2,3} \mid Y(A_i) - Y(A_{(i+1) \pmod 3}) \mid$
2. choisir le côté qui a la plus grande longueur (distance approximative) : $\max_{i=1,2,3}(\mid X(A_i) - X(A_{(i+1) \pmod 3}) \mid + \mid Y(A_i) - Y(A_{(i+1) \pmod 3}) \mid)$
3. choisir le côté qui a la plus grande longueur (distance euclidienne plus coûteuse en calcul que la précédente mais plus précise) : $\max_{i=1,2,3} \| \overrightarrow{A_iA_{(i+1) \pmod 3}} \|^2$
4. choisir le côté qui a la plus grande largeur : $\max_{i=1,2,3} \mid X(A_i) - X(A_{(i+1) \pmod 3}) \mid$
5. chosir systématiquement le côté entre les deux premiers sommets dans l'ordre de soumission : $[A_1A_2]$

La première figure 3.17 présente la localité d'un monoprocesseur équipé d'un cache de 16Koctets pour *blowout775* et montre que les découpages peuvent dégrader la localité. Cette dégradation dépend cependant beaucoup de l'algorithme de découpage choisi, elle est systématiquement la pire pour l'algorithme 5. Ceci s'explique par les très grands triangles sur lesquels il faut itérer un grand nombre de fois le découpage.

On observe que la localité d'un monoprocesseur est préservée lors d'un découpage des triangles en coupant systématiquement le plus haut côté. Ceci ne garantit pas cependant que la localité multiprocesseur soit autant préservée, les triangles étant non seulement coupés, mais également distribués sur différents processeurs. Pour évaluer cela, nous avons observé l'impact du découpage sur la localité dans le cas le pire, c'est-à-dire sans utiliser de rafales. La deuxième figure 3.17 montre que le découpage des triangles dégrade de manière importante les performances. De plus, la taille maximale des triangles a un impact sur la localité qu'elle n'avait pas précédemment.

L'impact des algorithmes de découpage est différent en monoprocesseur et en multiprocesseur. En multiprocesseur, dans la plupart des cas, le découpage qui préserve la taille des côtés dans les mêmes ordres de grandeur (découpage du plus grand côté 2 et 3) est celui qui donne les meilleures performances. En effet, lorsque des triangles sont séparés, il vaut mieux qu'ils aient une forme proche du carré car ceci préserve à la fois la localité inter-ligne et inter-colonne alors que les autres formes ne privilégient qu'une des deux localités.

Cela montre que l'algorithme de découpage 1 est meilleur pour préserver la localité inter-triangle et que l'algorithme 2 est meilleur pour la localité intra-triangle. Cependant, ce qui va nous intéresser dans la suite c'est l'utilisation de rafales de triangles pour précisément préserver cette localité inter-triangle. Nous avons donc évalué l'impact de l'algorithme de découpage sur la localité et la taille maximale des triangles. La figure 3.18 montre le comportement des caches en présence de rafale. On voit que ces comportements sont identiques à ceux d'un cache monoprocesseur.

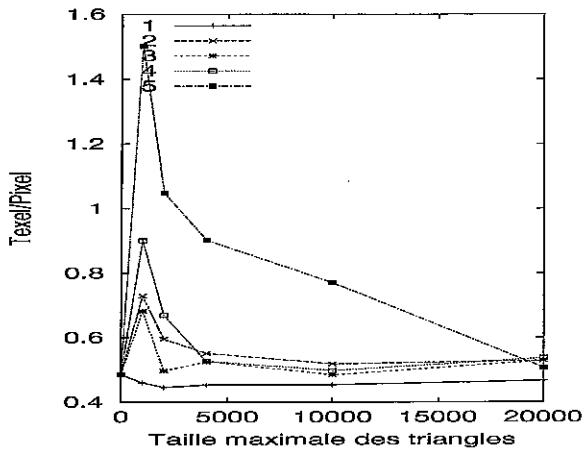

FIG. 3.18: Evolution de la localité en fonction du seuil de découpage et de l'algorithme de découpage, avec 64 processeurs et des rafales de 40 triangles.

Nous avons donc vu que le découpage des triangles diminuait la localité, mais qu'on pouvait limiter cette diminution en choisissant bien l'algorithme de découpage. Nous utiliserons donc dans la suite l'algorithme de découpage qui convient le mieux : limiter les triangles à 1000 pixels avec l'algorithme de découpage 1 (couper systématiquement le côté le plus haut).

Performance Nous avons étudié les améliorations de performances par rapport aux performances présentées au début de cette section et ceci pour différents débits de bus (1/2, 1, 2 et 4 pixels par texel). Sans les mécanismes, l'accélération variait entre 5 et 45 pour 64 processeurs. L'accélération varie maintenant entre 45 et 64. Ceci est principalement du au mécanisme de découpage des triangles qui résout efficacement le problème de l'équilibrage de la charge.

Le reste de l'amélioration concerne la localité des triangles et est apporté par les rafales de triangles. Pour tous les bus et tous les benchmarks, les rafales améliorent les performances. Cette amélioration peut aller jusqu'à 25%. Elle est plus importante sur les petits bus (1/2 et 1 texel par pixel), car ils présentent plus de périodes de saturations et sont donc plus sensibles à des augmentations de bande passante. Le mécanisme de rafales est utile quel que soit le nombre de processeurs, même si l'impact sur les performances est plus important avec un grand nombre de processeurs. Ceci est du au fait que le problème de la localité se produit à partir du moment où des triangles successifs sont dessinés sur des processeurs différents. Par ailleurs, dans

beaucoup de configurations (en particulier pour 64 processeurs), il est difficile de trouver une taille de rafales optimale (valeur réalisant le meilleur compromis entre équilibre de charge du cache et apport de localité) quel que soit l'algorithme utilisé. Ceci fait que l'optimum n'est pas stable et dépend du nombre de processeurs et du benchmark.

Références avec résultats détaillés : [87, 88, 89, 76, 90].

3.5 Conclusion

Les résultats obtenus dans ce chapitre montrent que l'exploitation des différents niveaux de parallélisme disponibles dans les processeurs peut conduire à des gains en performance considérables lors de l'optimisation d'une application multimédia au niveau code source et au niveau matériel. Ils ont également mis en avant le problème majeur de la bande passante mémoire nécessaire pour soutenir ces différentes exploitations du parallélisme.

CHAPITRE 4

Intégrer différents aspects de la performance : vitesse d'exécution et consommation

La rapidité d'exécution n'est plus la seule contrainte à considérer même si elle l'a été très longtemps dans l'évaluation des processeurs généralistes et haute-performance. En effet, la consommation de ces machines est de plus en plus importante (exemples : l'Itanium d'Intel consomme 130W à 800 Mhz et l'Athlon XP d'AMD consomme 60W à 1.6 GHz) et a un impact sur leur utilisation. Dans le cadre des processeurs généralistes, l'augmentation de la puissance dissipée augmente la dissipation ambiante, limite l'utilisation de ces processeurs sur batterie, diminue la durée de vie de ces machines et engendre un coût supplémentaire pour les dispositifs de refroidissement. Dans le cadre des processeurs enfouis, ces mêmes phénomènes se produisent, mais leur importance est accentuée (importance prépondérante de l'utilisation de batterie, des coûts de conception, ...), alors que la demande d'architectures performantes est très forte pour des systèmes de traitement de l'information et de communication (téléphones portables, assistants personnels, ...).

Dans ce travail, nous avons évalué l'impact de l'utilisation des instructions SIMD (impact du parallélisme de données) sur la consommation d'énergie pour des applications multimédia. Ensuite, nous avons évalué comment les améliorations de performance induites par l'utilisation d'instructions SIMD pouvaient engendrer des réductions supplémentaires de la consommation d'énergie.

4.1 Intégrer l'évaluation de la consommation

4.1.1 Mesure de consommation

Les différentes évolutions technologiques ont conduit à une diminution de la taille de gravure et du voltage qui permet *a priori* une réduction de la dissipation. Mais la puissance dissipée augmente à chaque nouvelle génération de machine. Cette augmentation est due à la technologie (augmentation du nombre de transistors), mais aussi à l'utilisation de ce grand nombre de transistors. En effet, l'augmentation du nombre de transistors a entraîné une augmentation du nombre d'instructions exécutées en parallèle, une gestion de l'exécution dans le désordre, l'intégration d'un plus grand nombre d'unités fonctionnelles, la mise en oeuvre de la spéculation sur les données et de techniques de prédiction de branchement sophistiquées, une augmentation de la longueur du pipeline, etc.

Il existe deux types de dissipation d'énergie dans un circuit CMOS, la dissipation statique provoquée par le petit courant de fuite et la dissipation dynamique provoquée par le courant transitoire de court-circuit pendant le basculement des portes logiques, et le chargement et le déchargement des capacités associées aux différents noeuds.

La dissipation statique représente une fraction très faible de l'énergie totale de l'ordre du dixième de nanowatt par porte logique. A l'heure actuelle, la dissipation statique est inférieure d'un ordre de grandeur à la consommation dynamique, mais elle augmente avec la réduction de la finesse de gravure [16]. Dans notre travail, nous avons considéré uniquement la puissance dynamique.

La formule d'estimation de la puissance est donnée par $P_d = a * C * V^2 * f$ où a est le facteur d'activation (nous précisons la signification de ce facteur dans la section suivante). La puissance croît linéairement avec la fréquence f et géométriquement avec le voltage V, C étant la capacité d'une porte. La puissance est définie comme le taux auquel l'énergie est consommée, l'énergie est calculée en intégrant la puissance par rapport au temps.

Dans les technologies CMOS ou BiCMOS, les différents composants (portes logiques, bus) consomment de l'énergie uniquement durant les transitions logiques (si on ignore le courant de fuite ou courant de polarisation inverse). Ainsi, la consommation est très dépendante de l'activité de ces circuits. Autrement dit, plus un circuit commute (change d'état), plus il consomme de l'énergie.

Face à l'augmentation de la dissipation, des techniques telles que le *clock gating* sont intensivement utilisées. Le *clock gating* permet de désactiver le signal d'horloge dans une unité dont les ressources sont momentanément non utilisées. Il permet de limiter, mais pas d'éliminer l'augmentation de la dissipation.

4.1.2 Modélisation

Différents travaux ont été réalisés sur les modèles de consommation au niveau fonctionnel. Les plus récents considèrent les principaux éléments de l'architecture [14, 91] (et pas uniquement les caches) et ont été intégrés à des outils de simulation fonctionnelle [2].

La formule d'estimation de la puissance est donc dépendante d'un facteur d'activation (a). Ce facteur dépend des programmes simulés et il est difficile à estimer (enregistrement de toutes les transitions). Ainsi, les modèles actuellement utilisés dans la simulation fonctionnelle fixent sa valeur en considèrant qu'il n'est dépendent que de l'élément activé. Ainsi, pour les circuits utilisant le préchargement (exemple : blocs de logique dynamique, lignes de bits dans les mémoires SRAM) qui sont chargés et déchargés à chaque cycle, le facteur d'activation sera égal à 1. Pour les autres circuits (dans lesquels il est très difficile d'enregistrer l'activité), le facteur d'activité est égal à 0.5 (changements d'état aléatoires).

Pour notre étude, nous avons considéré la classification et la modélisation de Wattch [14] pour les éléments structuraux de base dont l'assemblage permet la réalisation d'un bloc fonctionnel complexe en CMOS.

Les principaux éléments structuraux que nous avons considérés sont les suivants :

- tableaux de cellules de mémoire SRAM. Cette structure est paramétrée par le nombre de lignes (taille de la mémoire en nombre de mots), le nombre de colonnes (largeur d'une colonne) et par le nombre de ports de lecture/écriture. Les éléments que nous avons modélisés avec cette structure sont les caches d'instructions et de données de premier et second niveau et les prédicteurs de branchement.
- mémoires associatives (CAM). Avec cette structure, nous avons modélisé le TLB d'instructions et de données, le tampon de réordonnancement et les stations de réservation. Par ailleurs, les jeux d'instructions considérés étant de type RISC (instructions de longueur fixe), nous avons considéré que les étapes de chargement et de décodage ont la même consommation quel que soit le type de l'instruction considérée. De même, une entrée dans le tampon de réordonnancement ou dans une station de réservation consomme la même énergie à chaque cycle quel que soit le type de l'instruction.
- blocs logiques combinatoires complexes. Il est trés difficile de modéliser la consommation des blocs logiques complexes notamment parce qu'il n'existe pas une façon standard d'implémenter une fonction logique donnée dans un circuit CMOS (nombre de transistors, disposition spatiale dont dépend

la longueur des fils d'interconnexion, etc.). Etant donné qu'on cherche uniquement une estimation relative de la consommation pour les differents composants d'un processeur, on considère uniquement le nombre de transistors utilisés dans une unité fonctionnelle, la consommation d'un bloc de logique étant supposée proportionnelle au nombre de transistors qu'il contient. Nous avons modélisé les unités entières, flottantes et multimédia, le décodage et la logique pour la fonction de hachage utilisée dans le prédicteur de branchement (*g-share*). La consommation des unités fonctionnelles est dépendante du nombre d'instructions exécutées et du type de l'instruction.

En définitif, une partie des structures dépend uniquement du nombre d'instructions exécutées et de leur type (unités fonctionnelles, stations de réservation, tampon de réordonnancement, etc.) et une autre partie est dépendante de l'addresse des données et de leur taille (caches, TLB, etc.).

Par ailleurs, nous n'avons pas considéré la consommation générée par l'activité des divers bus à l'intérieur du processeur qui doit tenir compte des routages physiques dans le plan du circuit. Nous n'avons pas considéré également la dissipation due au réseau d'horloge du processeur qui peut être une source de dissipation importante, mais qui nécessite là aussi des informations sur la disposition des différents éléments sur le circuit. De plus, nous avons considéré un *clock gating* parfait, i.e., une consommation nulle lorsqu'un bloc fonctionnel est inactif.

4.2 Consommation des applications multimédia

Les applications multimédia utilisent souvent des types de données de 8 ou 16 bits. Si aucun mécanisme spécifique n'est implémenté dans les registres scalaires (registres 32 bits) et les unités fonctionnelles, la consommation pour l'exécution d'une instruction opérant sur 8 bits (utilisation d'un registre 32 bits) est similaire à celle d'une instruction opérant sur 32 bits. Les instructions SIMD permettent une meilleure utilisation des ressources en opérant sur des largeurs de données utiles. Ces extensions SIMD permettent également de réduire la consommation en :

- réduisant le nombre d'accès aux caches grâce à une augmentation de la taille des éléments chargés. Les types de données manipulés en SIMD sont plus larges (exemple de l'AltiVec : 128 bits), ce qui permet d'effectuer quatre fois moins d'accès aux caches (réduction du nombre de lecture des étiquettes des caches et du nombre d'accès au TLB). Par ailleurs, la mise en oeuvre du SIMD est associée à une augmentation du nombre de registres (exemple de l'AltiVec : 32 registres de 128 bits supplémentaires) et qui va permettre d'exploiter plus de localité au niveau des registres, et donc de réduire le nombre d'accès au cache.
- réduisant le nombre d'instructions. Les instructions SIMD permettent d'effectuer un traitement sur plusieurs données, permettent donc de réduire le nombre d'instructions exécutées (exemple : AltiVec permet le traitement de 4 à 16 données en parallèle pour une seule instruction). Elles permettent également d'effectuer plusieurs traitements à l'aide d'une seule instruction (exemple : instruction complexe comme la racine carré).

Evaluation La configuration de la machine simulée est présentée dans la table 4.1. Pour cette étude, nous considérons un cache d'instructions parfait (tout accès est un succès) et un prédicteur de branchement parfait (toutes les prédictions sont bonnes). Ces hypothèses sont valables pour les applications multimédia étudiées, applications très régulières, possédant un code de petite taille et des branchements très prévisibles. Les caches de données de premier et second niveau, ainsi que le TLB, ont une consommation dépendante des données (taille et adresses), tandis que le reste des éléments du processeur est dépendant du nombre d'instructions

	Configuration de la machine
Largeur de chargement	8
Registres de renommage entiers	40
Registres de renommage flottants	40
Taille du tampon de réordonnancement	80
Profondeur des stations de réservation	10
Nombre d'unités de chargement/Rangement	2
Nombre d'unités entières	4
Nombre d'unités flottantes	2
Nombre d'unités de branchement	2
Nombre d'unités de calcul SIMD	2
Nombre d'unités de permutation SIMD	2
Taille cache L1, taille ligne, associativité	64KB/32B/8
Taille cache L2, taille ligne, associativité	1024KB/32B/8
Largeur du bus L2	256b
Ratio CPU/cache L2	1
Largeur du bus mémoire	128
Ratio CPU/mémoire	8

TAB. 4.1: Caractéristiques de la machine simulée.

Benchmark	Nb inst PPC/ Nb inst SIMD
Mesa (flottant)	1.91
FIR (flottant)	3.45
SAD	16.27
IDCT	9.42
RGB2YCbCr	9.83
Image Filter	18.35

TAB. 4.2: Réduction du nombre d'instructions en SIMD.

exécutées et de leur type (voir description de la méthodologie en annexe).

Nous avons utilisé les extensions SIMD AltiVec du PowerPC, elles permettent le traitement de 4 à 16 données en parallèle ce qui permet une réduction du nombre d'instructions exécutées du même ordre. En revanche, la consommation associée à l'unité fonctionnelle exécutant l'instruction est modélisée comme 4 fois celle d'une unité classique PowerPC. La consommation de l'étage d'exécution est donc plus importante par instruction en AltiVec (unités et registres plus larges).

La réduction du nombre d'instructions exécutées est présentée dans la table 4.2. Cette réduction est moins importante pour les programmes flottants notamment parce que la taille des données traitées est plus importante et donc que le parallélisme de données est plus faible.

La réduction de la consommation pour les différents composants de l'architecture est présentée dans la table 4.3. Elle est également moins importante pour les programmes flottants. De même que pour la réduction du nombre d'instructions, ce phénomène est du au fait qu'un vecteur SIMD entier permet de travailler sur 8 à 16 éléments, alors qu'un vecteur flottant permet de travailler sur 4 éléments. Pour Mesa, il faut rajouter le fait que les données doivent être réorganisées et un grand nombre d'instructions SIMD sont donc consacrées à la transposition de ces données. SAD présente la meilleure amélioration de la consommation, car les données traitées sont des données 8 bits et les instructions SIMD min/max permettent de supprimer des branchements (et donc des instructions).

La répartition de la consommation est présentée en moyenne dans les figures 4.1 et 4.2. La consommation

Conso PPC/SIMD	Mesa(fp)	FIR(fp)	SAD	IDCT	RGB2YCbCr	Image Filter
DL1	3.43	8.04	12.12	6.46	8.22	7.71
IL1	2.03	7.30	17.22	9.96	10.40	19.42
DTLB	3.89	8.64	15.72	7.27	13.33	10.63
ITLB	1.91	3.45	16.27	9.42	9.83	18.35
ROB	1.91	3.45	16.27	9.42	9.83	18.35
BPRED	1.91	3.45	16.27	9.42	9.83	18.35
UNITS	1.16	0.78	7.35	3.50	1.93	8.22
REGS	0.67	0.79	7.29	2.86	1.73	8.27
SR	1.91	3.45	16.27	9.42	9.83	18.35
DECODE	1.91	3.45	16.27	9.42	9.83	18.35
Total	1.62	2.44	10.74	5.82	4.29	8.64
Sans cache	1.32	1.60	11.40	5.65	4.33	13.13

TAB. 4.3: Réduction de la consommation pour les différents blocs fonctionnels du processeur.

totale d'énergie est systématiquement réduite en SIMD par rapport au PowerPC et la consommation d'énergie associée à l'exécution prend une part presque deux fois plus importante en SIMD qu'en PowerPC. En effet, un certain nombre de structures sont moins sollicitées (accès aux caches, TLB, décodage, stations de réservation, prédicteurs, ROB, ...). Par contre, toute la partie traitement des calculs génère une consommation similaire.

Les résultats observés en moyenne sur du code PowerPC ont été validés par des comparaisons avec des résultats moyens mesurés par Intel sur des Pentium III.

4.3 Réaliser un compromis performance/consommation

Certaines applications multimédia (applications temps réel) ont un besoin donné (limité) en performance. Par exemple, le rendu d'une scène 3D affiché sur un écran avec une fréquence de rafraîchissement de 75Hz ne nécessite pas d'afficher plus de 75 images par seconde ou encore, la décompression vidéo n'a pas besoin de s'effectuer à plus de 25 images par seconde. Ces besoins en performance fixés peuvent être exploités afin de réduire la consommation d'énergie. L'idée est alors d'ajuster la fréquence et le voltage du microprocesseur afin de fournir le niveau de performance souhaité pour une application donnée. Ainsi, les gains en performance peuvent se traduire par une réduction de la consommation.

Pour cette analyse, nous allons déterminer la fréquence d'utilisation en considérant que la performance atteinte à 1GHz en PowerPC est suffisante. Nous allons donc simuler une architecture utilisant un jeu d'instructions SIMD, faire varier le rapport fréquence bus sur fréquence processeur et choisir la fréquence de fonctionnement telle que le niveau de performance soit supérieur ou égal au niveau de performance atteint en PowerPC. Les voltages choisis sont 1.8V pour 1GHz, 1.5V pour 500MHz, 1.3V pour 250MHz et 1.1V pour 125MHz. Ils sont utilisés sur des processeurs existants gravés en 0.18 microns. Les résultats de ces simulations sont présentés dans la figure 4.3. Pour obtenir ces résultats, on fait varier le rapport fréquence bus sur fréquence du processeur pour les configurations SIMD. Ils permettent ensuite de déterminer à quelle fréquence les programmes doivent s'exécuter en SIMD afin d'atteindre au minimum les mêmes performances que sans SIMD (exemple sur la figure 4.3 : pour Mesa, on peut considérer une fréquence de 500 Mhz avec la configuration SIMD et exhiber les mêmes performances que la configuration sans SIMD et une fréquence de 1 Ghz).

Les résultats de consommation d'énergie sont présentés dans la figure 4.4 et donnent la réduction de la consommation d'énergie totale (relativement à une exécution sans SIMD) liée à l'utilisation du SIMD, et à l'ajustement de la fréquence et du voltage du microprocesseur afin de conserver des performances supérieures

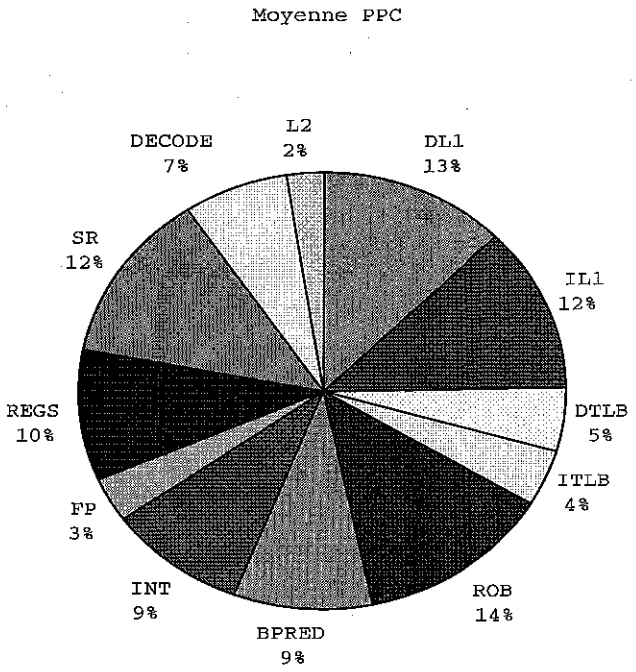

FIG. 4.1: Répartition de la consommation d'énergie sans extensions SIMD.

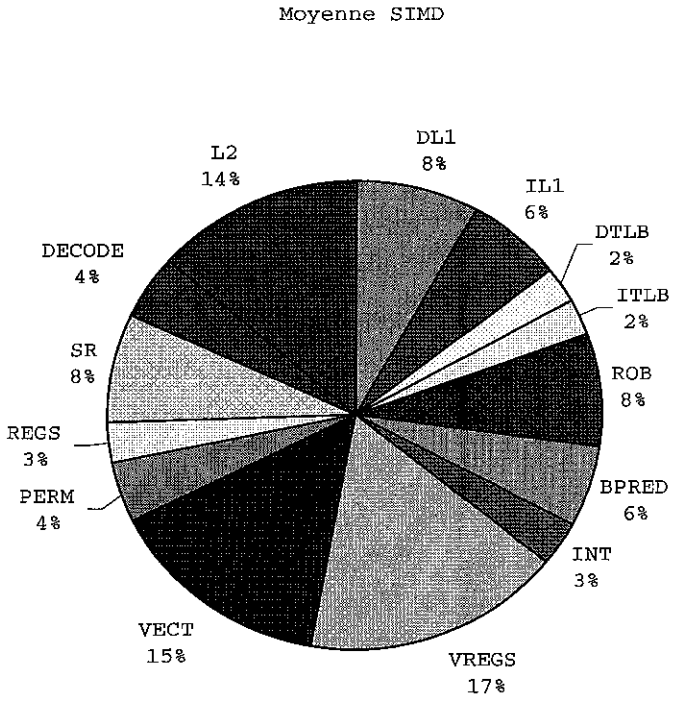

FIG. 4.2: Répartition de la consommation d'énergie avec extensions SIMD.

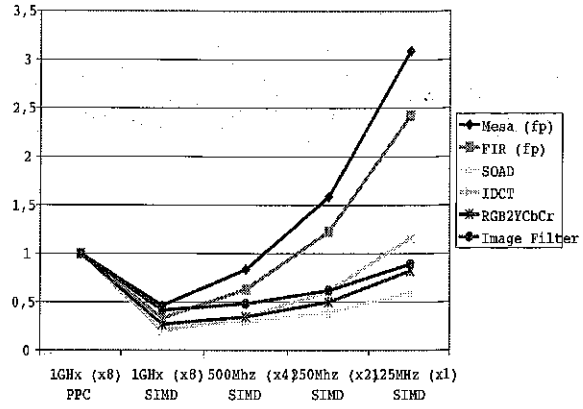

FIG. 4.3: Temps d'exécution relatif à une configuration sans SIMD.

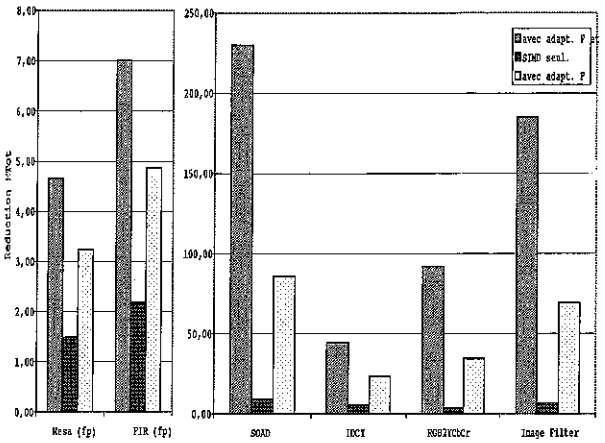

FIG. 4.4: Réduction de la consommation d'énergie pour une performance donnée.

ou égales à l'exécution sans SIMD. Les améliorations varient entre 4.8 et 7 pour les programmes flottants et 44 et 230 pour les applications entières. En effet, ces applications sont celles pour lesquelles l'amélioration de performance liée à l'utilisation du SIMD est la plus importante, et donc la réduction de la consommation également. L'adaptation de la fréquence et du voltage permet de combiner ces avantages et d'obtenir des réductions de consommation très importantes.

Références avec résultats détaillés : [75, 74, 71, 72].

4.4 Conclusion

Intégrer la mesure de la consommation dans l'évaluation d'une machine est important pour mesurer précisément l'impact d'une optimisation matérielle ou logicielle. Les optimisations proposées doivent être performantes en terme de temps, mais doivent également exhiber un coût énergétique raisonnable. Il peut alors être intéressant de traduire une augmentation en terme de vitesse d'exécution par une réduction de la consommation, notamment dans le cadre d'applications embarquées (exemple : PC portable sur batterie).

Dans nos travaux, la connaissance des besoins en performance pour certaines applications multimédia et le gain en performance obtenu avec les optimisations SIMD, ont abouti à une réduction importante de la consommation. Mais cette approche est globale et ne peut s'adapter qu'à des applications ayant un comportement régulier (adaptation de la fréquence et du voltage fixés pour toute l'exécution de l'application). Il faudra, par conséquent, exploiter ces résultats en essayant d'adapter dynamiquement la fréquence et le voltage en fonction du comportement local des applications.

CHAPITRE 5

Perspectives

Les travaux de recherche présentés dans ce mémoire concernent l'analyse et l'optimisation des processeurs généralistes dans des contextes d'utilisation et d'évaluation réalistes (applications multimédia, consommation d'énergie, ...). Ils portent notamment sur l'exploitation des différentes formes de parallélisme (parallélisme d'instructions, de données et de flots) et sur l'étude de la hiérarchie mémoire, deux aspects fondamentaux dans l'optimisation de performance des architectures de processeurs.

L'importance des modèles, des applications et des métriques tend à s'accentuer en raison de l'augmentation de la complexité des architectures et de la diversité des applications nécessitant de hautes performances (applications embarquées). L'analyse des architectures de processeurs superscalaires demande une modélisation fine du comportement fonctionnel et temporel de la machine en raison des intéractions complexes entre les composants et de l'utilisation de techniques de spéculation sophistiquées. Par ailleurs, l'utilisation de ces architectures dans des contextes divers demande de considérer des applications et des métriques adaptées (exemples : consommation, prédiction des performances).

Mais la complexité croissante des architectures de processeurs superscalaires rend difficile leur utilisation, d'une part pour des applications embarquées et d'autre part, parce qu'il est difficile d'exploiter les performances de ces architectures.

Par ailleurs, ces architectures complexes aux coûts de conception, fabrication et vérification élevés rendent la croissance exponentielle prévue par G. Moore de plus en plus difficile à satisfaire. De plus, il semble que des limites physiques risquent d'être atteintes d'ici 10 à 20 ans (prévisions de la SIA, *Semiconductor Industries Association*). Par conséquent, il faut commencer à envisager des alternatives aux architectures Von Neumann sur silicium : nouveaux procédés de fabrication et nouveaux matériaux (exemple : électronique moléculaire), ou à plus long terme, nouveaux principes physiques de traitement de l'information (exemples : ordinateur quantique, machines bio-inspirées).

Mes perspectives de recherche à moyen et long termes sont liées aux problèmes de complexité des architectures modernes et aux besoins en performance d'applications variées. Elles portent d'une part sur le problème de la génération de codes optimisés pour les systèmes haute-performance à base de processeurs VLIW plus adaptés aux contraintes des applications embarquées, et d'autre part, sur le problème d'exploitation des performances des architectures superscalaires (exploitation des capacités de ces architectures) et de leur adaptation aux contraintes des applications embarquées.

Augmenter la performance des systèmes embarqués Avec l'augmentation continue du nombre de transistors par puce, l'augmentation des performances des processeurs généralistes passe nécessairement par une augmentation du parallélisme, et notamment du parallélisme entre instructions. A l'heure actuelle, il existe deux approches pour exploiter le parallélisme entre instructions : une extraction statique du parallélisme à la compilation (VLIW, EPIC) et une extraction dynamique du parallélisme (processeurs superscalaires).

L'extraction dynamique du parallélisme ne sollicite pratiquement pas le compilateur, elle est presque entièrement implémentée de manière architecturale (stations de réservation, renommage de registres, ...) et est relativement efficace. Par contre, elle est complexe d'un point de vue matériel, ce qui entraîne notamment une consommation importante, un coût élevé, une sous-utilisation du matériel, etc. C'est essentiellement pour ces raisons que la plupart des architectures embarquées ne font pas appel au paradigme superscalaire et ont adopté le paradigme VLIW. Malheureusement, l'extraction du parallélisme à la compilation peut être très complexe pour nombre d'applications en raison des limitations actuelles de l'analyse statique. Par conséquent, il est difficile d'atteindre des performances élevées avec un processeur VLIW pour un grand nombre d'applications.

Aussi, nous proposons d'étudier une troisième approche qui combine les avantages du paradigme superscalaire et du paradigme VLIW, que nous avons appelée compilation matérielle. Le principe est basé sur l'utilisation de la simulation du processeur VLIW considéré combinée à un algorithme d'ordonnancement dynamique.

Cette approche peut accélérer le développement de logiciels pour les architectures spécialisées de conception particulière (non régulière), elle peut également s'avérer efficace pour les codes difficilement optimisables à la compilation, notamment les codes entiers dans les architectures VLIW (systèmes embarqués, mais aussi systèmes généralistes et haute-performance à base d'EPIC). Par ailleurs, elle peut être utilisée pour contrôler l'utilisation des ressources et donc contrôler, par exemple, la consommation. D'autres approches s'intéressent actuellement au problème de la génération de codes optimisés, notamment la compilation dynamique (transformation à l'exécution du binaire) et la compilation itérative (boucle de retour à la compilation).

D'un point de vue plus pragmatique, alors que la recherche sur les processeurs généralistes profite davantage à l'industrie américaine et asiatique (Intel, Sun, Mips, AMD, Transmeta, ...), l'industrie européenne joue un rôle important dans le domaine des processeurs embarqués (Siemens, Thomson, Philips, ...) et peut donc être particulièrement intéressée par ce type d'approches.

Auto-configurer et spécialiser l'architecture des processeurs Les processeurs généralistes sont conçus pour exhiber de bonnes performances sur une grande variété de programmes. Avec l'augmentation continue du nombre de transistors par puce, ils intègrent des composants de plus en plus complexes adaptés aux propriétés d'une grande majorité de programmes (exemples : prédiction de branchement hybride, grande taille de fenêtre d'instructions). Ils intègrent également des composants dont les caractéristiques sont figées quelle que soit l'application (exemple : caractéristiques de la mémoire cache). Cette complexité et cette rigidité ont un coût et conduisent souvent à une utilisation inefficace du matériel et à une surconsommation. Les systèmes embarqués qui requièrent des performances de plus en plus importantes notamment pour les applications de traitement de l'information et de la communication sont également fortement pénalisés par cette sous-utilisation du matériel (augmentation du coût, de la consommation, ...).

Ces constatations plaident pour une spécialisation du matériel amorcée avec l'implémentation de composants spécialisés comme les unités vectorielles destinées aux applications multimédia. Par ailleurs, les évolutions des processeurs, que ce soit à moyen terme au travers de la logique reconfigurable ou à plus long terme au travers des nouvelles technologies (électronique moléculaire, ADN, ...) peut laisser présager que le matériel aura des capacités à se construire lui-même, à s'adapter en fonction des propriétés/comportements de l'application (via un logiciel ou via une réaction chimique), à s'auto-spécialiser. Cette faculté d'auto-adaptation nous semble importante pour les années compte tenu des différentes évolutions et possibilités technologiques.

A l'heure actuelle, il semble tout à fait envisageable de proposer une architecture généraliste performante (en terme de temps d'exécution et de consommation) auto-adaptable/auto-reconfigurable (adapter à l'ap-

plication) sans intégrer de nouvelles technologies. Certains travaux existent sur la possibilité de configurer localement des composants de l'architecture (prédicteurs de branchement [49], fenêtre d'instructions [17], hiérarchie mémoire [4], ...). Une autre étude très récente [26] tente de déterminer dynamiquement le changement de phases dans une exécution, d'identifier des phases récurrentes et d'estimer la taille des ensembles de travail afin de déterminer une configuration adaptée.

Dans ces perspectives, l'architecture doit être capable de considérer plusieurs configurations matérielles et de choisir la configuration la mieux adaptée à l'application cible et aux contraintes imposées (performance, consommation, ...). L'architecture doit s'adapter aux différentes classes d'applications et contrôler les différents éléments utilisés, elle pourra également s'adapter à la vitesse de fonctionnement des différents blocs (asynchronisme) et intégrer des éléments spécialisés destinés à améliorer certaines classes d'applications. L'idée n'est pas uniquement de considérer les configurations et les composants existants, mais en identifiant les besoins spécifiques d'applications de proposer de nouvelles organisations ou de nouveaux composants / fonctionnalités. Cette approche intègre les travaux débutés sur l'adaptation de la consommation et de la performance dans le chapitre 4. Nous qualifierons cette reconfigurabilité de statique, car les composants et configurations disponibles doivent être prévues sur le processeur.

Mais cette reconfigurabilité peut également être réalisée dynamiquement à l'aide de circuits FPGA (*Field Programmable Gate Arrays* ou réseaux logiques programmables) constitués d'un réseau de cellules logiques identiques placées dans une infrastructure de lignes d'interconnexion. L'utilisateur peut programmer la fonction de chaque cellule, ainsi que les interconnexions entre les cellules. L'idée est d'optimiser dynamiquement une architecture ou une partie d'une architecture pour une application précise et non plus de façon générale et/ou statiquement. L'utilisation des FPGA n'est pas nouvelle ; ils exhibent d'ailleurs de bonnes performances comme architecture dédiée pour un certain nombre d'applications (décryptage RSA, séquençage ADN, traitement du signal, etc.). Une utilisation envisageable est de combiner un processeur généraliste programmable avec de la logique reconfigurable pouvant permettre de masquer les temps de configuration assez longs des circuits reconfigurables. Dans ces perspectives, la partie reconfigurable serait vue comme un coprocesseur spécialisé avec un gros grain de reconfigurabilité (exemple : unités fonctionnelles). Les bénéfices potentiels d'une telle implémentation sur du matériel reconfigurable sont une basse consommation, une réduction des coûts de conception et de fabrication et des performances élevées.

Par contre, que ce soit pour la reconfigurabilité statique ou dynamique, il nous faudra trouver un moyen efficace de détecter les spécificités des applications (i.e., mettre en avant des configurations adaptées, des éléments spécialisés) et de les exploiter efficacement. Les capacités de calcul doivent pouvoir être exploitées par les programmes sans difficulté pour le programmeur, ce qui n'est pas actuellement le cas pour les architectures *a priori* performantes (superscalaire, EPIC, SIMD, ...).

Vers la nanoarchitecture A long terme, on commence à envisager des alternatives au silicium, aux circuits et architectures actuelles en trouvant soit des matériaux remplaçant le silicium ou de nouveaux procédés de fabrication (exemple : électronique moléculaire), soit de nouveaux principes physiques de traitement de l'information (exemple : ordinateur quantique).

La nanoélectronique[1] s'impose en raison des défis croissants auxquels se heurtent les progrès de la technologie CMOS. La recherche en nanoélectronique et la conception de nouveaux produits recouvrent de nombreux domaines allant de la physique fondamentale à l'électronique, en passant par la chimie organique ; ce domaine est pluridisciplinaire. Des recherches au niveau architectural doivent accompagner ces recherches technologiques qui auront peut-être un impact important sur les architectures des futurs ordinateurs.

[1] Electronique à l'échelle de la molécule. Le but de la nanotechnologie moléculaire est d'arriver à un contrôle précis et individuel des atomes.

Les architectes de machines ont un rôle important à jouer face à l'enjeu de l'industrie électronique et ceci en collaboration avec les physiciens, chimistes et électroniciens. Les recherches sur certaines structures (exemple : nanotubes de carbone) sont arrivées à un stade où les propriétés sont de mieux en mieux connues, et par conséquent, exploitables pour étudier des architectures. Ces technologies promettent une augmentation du nombre de transistors encore plus importante (10^{10} portes par cm^2), mais il faudra trouver la meilleure façon de les utiliser. Actuellement, les architectures de Von Neumann se dimensionnent difficilement comme en témoigne la complexité croissante des architectures de processeurs et les limitations des optimisations destinées à exploiter les performances de ces architectures. Il faudra donc conduire une refléxion profonde sur les nouveaux paradigmes d'architecture que ce soit face à l'augmentation du nombre de transistors (poursuite de la miniaturisation actuelle), que face aux nouvelles technologies.

ANNEXE A

Méthodologie : simulations et applications

Afin d'analyser et d'optimiser les architectures de processeurs, les architectes doivent disposer d'outils de simulation représentant le comportement fonctionnel et temporel des architectures cibles, ainsi que d'applications représentatives de l'utilisation de ces architectures.

A.1 Simulations

Les performances (en terme de vitesse) d'une architecture peuvent être évaluées de façon réelle en exécutant un programme donnée sur l'architecture matérielle analysée. Malheureusement, il est difficile d'obtenir des analyses fines de chacun des composants de la machine. Il existe bien la possibilité d'avoir accès à des compteurs matériels, mais les informations recueillies sont limitées et ne suffisent pas à comprendre de façon précise le fonctionnement des composants du processeur. Par ailleurs, avec cette technique, il n'est pas possible d'étudier de nouvelles architectures ou de nouvelles optimisations matérielles.

De même, la puissance dissipée peut être calculée matériellement à l'aide de sondes ou encore par simulation électrique du circuit (connaissance exacte des schémas d'implémentation physique). Mais ces deux approches sont en pratique difficiles à mettre en oeuvre : il faut avoir à disposition la plateforme complète pour l'utilisation de sondes, une simulation complète est impossible à cause du nombre de portes logiques à simuler pour la simulation électrique.

Ainsi compte tenu de ces difficultés, l'évaluation des performances (en terme de vitesse, consommation, ...) d'un processeur ou d'un mécanisme architectural est réalisée avant tout prototypage à l'aide de la simulation logicielle. Cette simulation consiste à exécuter un simulateur qui modélise l'architecture à évaluer. Afin de simuler l'exécution d'un programme sur l'architecture, une suite d'instructions provenant d'un programme réel représentatif de la charge à laquelle sera soumise l'architecture, est injectée dans le simulateur.

A.1.1 Exécution d'une application sur un simulateur

La simulation peut être guidée par la trace (*trace-driven simulation*) ou guidée par l'exécution (*trace-driven execution*). Dans ces deux méthodes, il est nécessaire d'espionner les programmes afin d'obtenir les informations envoyées ensuite au simulateur. Ces méthodes d'espionnage sont purement logicielles. Dans le cas de la simulation guidée par la trace (*trace-driven simulation*), le programme est exécuté sur une machine réelle afin d'en générer la trace d'exécution qui contient notamment la liste des instructions exécutées, la liste des accès mémoire, etc. Mais pour certaines applications parallèles, le contenu de la trace peut dépendre de l'architecture sur laquelle l'application est exécutée. C'est le cas pour les applications allouant dynamiquement les tâches à effectuer aux flots d'exécution. Une solution est de synchroniser l'exécution de l'application sur la machine hôte avec le système simulé. Ce type de simulation est dite *guidée par l'exécution*.

Lors de l'exécution de l'application, dès qu'un flot atteint un point de synchronisation, son exécution est gelée. Les instructions exécutées et les événements multiprocesseurs rencontrés sont alors envoyés au simulateur qui simule l'exécution des instructions et des événements et informe la machine hôte du temps d'exécution consommé. Mais la simulation guidée par la trace est utilisable pour la grande majorité des applications à allocation statique.

Une autre méthode consiste à émuler les programmes directement sur l'architecture cible. La différence avec les techniques décrites précédemment est que la machine hôte n'est plus une machine réelle, mais un programme (appelé émulateur). Les instructions du programme sont exécutées par l'émulateur qui dispose à chaque cycle du contenu des registres et peut donc, si le simulateur le lui demande exécuter un mauvais chemin ce qui permet la simulation des instructions spéculatives. Parmi les outils actuels utilisant cette technique on citera SimpleScalar [2] et SimOS [69].

Afin de récupérer la trace des instructions exécutées, on instrumente (on annote) le code de l'application considérée en y ajoutant du code d'instrumentation. L'instrumentation peut s'effectuer au niveau du code source assembleur, au niveau du code objet ou au niveau du code exécutable de l'application.

Pour les premiers travaux présentés (travaux sur le préchargement), nous avons utilisé l'outil Spa développé par G. Irlam [41] pour machines SPARC et pour lequel il faut disposer des programmes sources (il faut recompiler les codes avec des options spécifiques).

Dans la suite des travaux réalisés sur les différentes formes de parallélisme, nous avons essentiellement utilisé PopPSY [56] un instrumenteur de code PowerPC au niveau de l'exécutable développé dans notre équipe de recherche. PopSPY n'est pas dépendant d'un compilateur donné et permet d'espionner les applications séquentielles, multiflot et multiprocessus. PopSPY utilise une bibliothèque partagée qui peut soit générer la trace d'exécution des applications espionnées, soit diriger un simulateur guidé par l'exécution. De plus, les bibliothèques dynamiques sont espionnées.

Nous avons également utilisé les outils fournis par Apple et Motorola. Certains tests (analyses SIMD) ont été réalisés sur un PowerMac G4 450 sous MacOS 8.6. Nous avons donc utilisé le générateur de traces d'exécution pittsTT6. Celui-ci fonctionne sous forme d'une bibliothèque partagée. Les traces d'exécution générées sont utilisées en entrée du simulateur de PowerPC G4, qui donne en sortie un certain nombre de statistiques tel le temps d'exécution en cycle, des informations sur l'occupation des unités fonctionnelles et les statistiques d'accès aux caches.

A.1.2 Simulateur

Comme nous l'avons déjà mentionné, l'analyse et l'optimisation des processeurs demandent une simulation précise du comportement fonctionnel et temporel de l'architecture. Ainsi pour nos travaux de recherche, nous développons et utilisons des simulateurs de processeurs. Un simulateur est un programme reproduisant le fonctionnement de tout ou partie du processeur étudié. Ce simulateur permet d'analyser en détail le fonctionnement du processeur et de comprendre les améliorations à apporter.

Malgré le grand nombre d'équipes en architecture, le développement de simulateurs n'est pas standardisé. Nous avons développé nous-même ces dernières années un certain nombre de simulateurs que nous avons utilisés dans différents travaux présentés dans ce mémoire. Cette situation a posé différents problèmes, notamment des développements très redondants, l'impossibilité de comparer les résultats (avec des études similaires), des risques accrus d'erreurs et un manque de précision. Depuis 1997, il existe un outil dont l'utilisation s'est largement répandue, SimpleScalar, mais il est relativement complexe et difficile à maîtriser (structure monolithique). Les modifications de l'outil pour intégrer par un exemple un nouveau mécanisme matériel ne sont pas aisées (modifications à différents endroits du code) et potentiellement source d'erreurs.

Dans l'équipe, nous avons donc décidé de développer un environnement de simulation dont le but est de permettre le développement rapide de simulateurs et de faciliter la réutilisation ou l'échange de travaux (de développements) entre membres de l'équipe dans un premier temps, puis plus largement. L'environnement développé par Jean-Luc Béchennec, ASF [5], est un *framework* C++ permettant de gérer efficacement les événements entre les boîtes (composants de l'architecture). La plupart des simulateurs utilisés dans la partie optimisation des applications multimédia sont basés sur cet environnement.

Afin de permettre une standardisation, nous avons récemment adopté un autre environnement déjà répandu dans le domaine de l'embarqué, System C (www.systemc.org), et adopté par 50 compagnies du domaine de l'embarqué. A partir de System C, nous développons une bibliothèque libre de composants qui va nous permettre d'assembler un certain nombre de processeurs existants. De plus, les optimisations ne se limitant pas uniquement à la performance, mais également à la consommation comme nous l'avons vu dans ce mémoire, nous comptons intégrer dans la bibliothèque de composants un modèle de consommation afin de pouvoir analyser et optimiser les applications vis à vis du critère de consommation.

A.2 Applications

Le choix des programmes tests utilisés comme données (entrées) des évaluations est important et conditionne les optimisations et architectures proposées. Ces programmes doivent être représentatifs de l'utilisation de la machine et doivent être modifiés en fonction de l'évolution de ces utilisations, qui sont elles-mêmes dépendantes de l'évolution de la performance des systèmes informatiques.

A.2.1 Suite de programmes standard

Les suites de programmes sont censées représenter un large spectre d'applications. Pour les processeurs généralistes, la suite la plus communément utilisée est SPEC CPU remise à jour régulièrement (SPEC92, SPEC95, SPEC2000). Elle a été conçue afin de mesurer les performances de calculs dits intensifs sur une grande variété de machines. Les applications sont développées par de vrais utilisateurs (par opposition à des programmes tests ou synthétiques) et permettent de mesurer la performance des processeurs, de la mémoire et des compilateurs.

La suite SPEC comprend des programmes entiers et des programmes flottants. Les programmes entiers sont des programmes de compression (gzip), des compilateurs (gcc, perl, lisp), des bases de données (vortex), etc. et les programmes flottants sont essentiellement des programmes de simulation numérique écrits en Fortran (swim, mgrid, applu, galgel, etc.) Les caractéristiques de ces applications sont bien connues dans la communauté architecture et nous ne les détaillons donc pas dans cet annexe (voir www.spec.org).

Mais les applications multimédia devenues la principale charge de travail exécutée par un ordinateur personnel sont sous-représentées dans cette suite (une seule application multimédia Mesa dans les SPEC2000).

A.2.2 Applications multimédia

Les applications multimédia manipulent des données telles que de l'audio, des images 2D et 3D, et des animations. Dans notre travail, nous n'avons considéré que des applications multimédia dédiées au traitement ou à la génération d'images. Les catégories d'applications représentées sont :

- la compression-décompression d'une image fixe en deux dimensions et la compression-décompression en temps réel d'une vidéo constituée de plusieurs images en deux dimensions.
- la synthèse d'image, i.e., la génération en temps réel de scènes en trois dimensions.

Dans cette section, nous allons décrire les caractéristiques des applications utilisées pour la plupart des évaluations (chapitres 3 et 4) et les programmes étudiés.

Compression-décompression

Compression d'images fixes Avec l'arrivée d'Internet, il a fallu considérer le problème de la compression de données transitant à travers ce réseau et on a donc assisté à l'émergence de différents formats de compressions de données et notamment du format JPEG.

Le format de compression JPEG défini par une association regroupant des représentants d'organismes de normalisation (*Joint Photographic Expert Group*) a été créé spécifiquement pour la transmission d'images photographiques de qualité. Cet algorithme de compression avec perte, c'est à dire que certaines informations de l'image seront supprimées lors de la compression et ne seront pas récupérées lors de la décompression, identifie des zones dans l'image où les couleurs sont très similaires et supprime les différences qui sont invisibles à l'oeil nu. La compression JPEG comporte trois étapes principales :

- étape de transformation : transformation des informations sur les pixels du format RVB (Rouge, Vert, Bleu) au format YCbCr. L'information sur l'image est alors exprimée sous forme de chrominance (Cb, Cr : valeurs de couleurs) et de luminance (Y : brillance). Puis, la transformée en cosinus discrète (DCT) est appliquée sur les pixels de chaque bloc (groupe de 8 x 8 pixels, soit 64 pixels) de l'image. Cette transformation numérique utilise la transformée de Fourrier. Elle permet de décrire chaque bloc en un graphique de fréquences (correspondant à l'importance et à la rapidité d'un changement de couleur) et en amplitudes (qui est l'écart associé à chaque changement de couleur).
- étape de quantification (l'étape destructive) : la quantification est l'étape où certaines informations sont ignorées, de sorte que la somme finale d'informations définissant l'image diminue. Dans la matrice obtenue à la suite de la DCT, une division par une matrice de quantification est effectuée afin de mettre à zéro les coefficients ayant le moins d'importance pour le rendu de l'image du point de vue de l'oeil humain ; ce sont principalement les coefficients des hautes fréquences. On obtient ainsi une matrice creuse. Pour chaque bloc, plus on conserve de fréquences, plus la reconstruction du bloc à partir de la liste des fréquences sera proche de l'original, mais plus le fichier sera gros (puisque l'on doit stocker plus de fréquences). Si l'on mémorise moins de fréquences, le fichier résultant est plus petit, mais l'image reconstruite est alors moins fidèle à l'originale.
- étape d'encodage : il faut ensuite coder les données produites. Le codage va s'effectuer en deux étapes, une étape qui modifie le stockage dans la matrice pour faire apparaître davantage de 0 consécutifs (on linéarise la matrice creuse précédemment obtenue, méthode dite de *zig-zag*). On effectue ensuite la compression proprement dite (exemple : algorithme de Huffman).

Compression d'images animées La numérisation d'images vidéo se développe de plus en plus, du fait de l'augmentation des débits d'Internet, de l'apparition de la vidéo numérique et de la télévision haute définition.

Une séquence vidéo occupe une très grande taille après numérisation, d'autant plus importante que la définition de l'image (le nombre de points qui la constituent) est élevée. Il est donc primordial de compresser les données résultant de la numérisation afin de diminuer l'espace nécessaire à leur stockage. Par ailleurs, pour obtenir des débits permettant une diffusion en temps réel sur une ligne de télécommunication, il est nécessaire que la vidéo soit compressée, et qu'elle soit décompressée et affichée en temps réel.

Le principal standard de compression audio et vidéo est le MPEG (*Moving Picture Experts Group*). Les versions MPEG1 et MPEG2 se sont succédées pour obtenir des taux de compression toujours plus élevés.

La norme la plus récente est MPEG4 [42], qui est la base du nouveau format de compression vidéo DivX [1], développé principalement pour permettre le téléchargement par Internet de films plein écran haute qualité.

La compression MPEG repose sur le fait que très souvent, peu de différences existent entre deux images consécutives d'un film (lorsque la caméra est fixe ou bouge lentement). Le principe de la compression est de stocker à intervalles de temps réguliers des images complètes compressées en JPEG (les frames I). Toutes les images entre deux frames I, appelées frames P, sont codées sous la forme d'une différence avec l'image précédente. Dans le format MPEG4, les frames I sont espacées d'au moins 300 images.

MPEG utilise toujours l'algorithme de JPEG pour compresser un certain nombre d'images dans la vidéo, les autres étant prédites à l'aide d'algorithmes reposant sur l'estimation de mouvement entre deux scènes (estimer la différence entre 2 blocs d'image). L'estimation de mouvement représente plus de la moitié du temps d'exécution pour la compression vidéo et influe sur la qualité de la vidéo compressée.

Pour coder une image sous forme de frame P, on commence par la diviser en blocs de 8x8 pixels, appelés macro-blocs. L'étape suivante consiste à rechercher l'ancienne position du macro-bloc dans l'image précédente. Pour cela, on le compare à celui ayant les mêmes coordonnées et à ceux dont les positions sont proches. La position retenue est celle du macro-bloc ayant le plus de pixels en commun avec le macro-bloc recherché. Si le nombre de pixels identiques est inférieur à un certain seuil, le macro-bloc est considéré perdu (c'est par exemple le cas lors d'un changement de scène dans le film ou quand un objet a bougé trop vite).

Une fois déterminée la position précédente du macro-bloc, on calcule une matrice contenant la différence de lumière et de couleur entre l'ancien et le nouveau macro-bloc. Cette matrice est compressée en JPEG. La valeur envoyée sur le flux de sortie pour ce macro-bloc se compose de la matrice de différences et du vecteur de mouvement du macro-bloc entre l'image courante et la précédente.

Les principales opérations utilisées pour effectuer ces traitements sont les suivantes :

- somme des valeurs absolues des différences (SOAD) : cette opération est au cœur de la recherche d'un macro-bloc dans l'image précédente, elle permet de trouver le vecteur de mouvement correspondant. La valeur de chaque pixel du macro-bloc recherché est soustraite à la valeur du pixel de macro-bloc actuellement examiné dans l'image précédente. La valeur absolue du résultat est ajoutée à une somme globale. Parmi tous les macro-blocs examinés dans l'image précédente, c'est celui ayant la plus petite SOAD qui est choisi (à condition qu'au moins une SOAD soit en dessous du seuil précédemment évoqué).
- transformée en cosinus discrète (DCT pour *Discrete Cosinus Transform*) : cette opération est utilisée pour la compression JPEG. La DCT est une transformée qui génère une représentation dans le domaine fréquentiel d'une matrice de points du domaine spatial. Dans la compression JPEG, elle est appliquée à des blocs de 8x8 pixels. Dans la matrice obtenue, une division par une matrice de quantification est effectuée afin de mettre à zéro les coefficients ayant le moins d'importance pour le rendu de l'image du point de vue de l'oeil humain ; ce sont principalement les coefficients des hautes fréquences. On obtient ainsi une matrice creuse que l'on linéarise avec la méthode *zig-zag* afin d'obtenir une suite de nombres compressible efficacement par un algorithme conservatif.
- transformée en cosinus discrète inverse (iDCT) : pour trouver la position d'un macro-bloc dans l'image précédente, on utilise en fait le résultat de la décompression de la frame précédente, qui sera l'image effectivement visualisée par le spectateur. De ce fait, après avoir compressé une image, il faut la décompresser pour pouvoir traiter l'image suivante. Cette décompression correspond notamment à l'application de la quantification inverse (qui sera cette fois une suite de multiplications) et de la transformée en cosinus discrète inverse.

[1] www.DivX.com

Programme-test utilisé Pour la compression d'images fixes, le programme test utilisé est JPEG. Pour la compression d'images animées, le programme-test utilisé est OpenDivX (la version *open source* de DivX) qui compresse un film constitué d'images de 900 Ko chacune. En fait, nous n'avons pas présenté les résultats concernant OpenDivX dans ce manuscrit. Par contre, nous avons utilisé certains microbenchmarks extraits de ces applications tels que SOAD, transformation de RGB en YCbCr, IDCT, ...

Synthèse d'images

La synthèse d'images d'une scène est obtenue en simulant par le calcul les phénomènes physiques (plus ou moins simples) appliqués aux images (le but est de produire une image qui se rapproche le plus de la réalité). Un système temps réel génère des images avec une latence de $\frac{1}{60}$ème de seconde et une bande passante de 60 images par seconde (autrement, l'image est saccadée), il faut donc disposer d'une puissance de calcul importante. Par ailleurs, ces applications ont pour caractéristiques notamment de traiter des tâches indépendentes. Ainsi, un certain nombre de machines parallèles dédiées à la synthèse d'images ont été fabriquées (exemple : Infinite Reality de Silicon Graphics), mais sont basées sur des composants spécialisés, elles sont donc coûteuses et à diffusion restreinte. Depuis quelques années sont également apparus des composants très bon marché destinés à accélérer les jeux dans les PC et consoles de jeux (notamment les jeux en image de synthèse, principaux consommateurs de calculs sur ces machines) et qui ont subi des améliorations importantes notamment technologiques (exemple : la PlayStation2 utilise une technologie de 0.18 micron et comporte 10,5 millions de transistors). Sur tous les PC vendus jusqu'à la sortie de la GeForce256 de Nvidia, le processeur hôte exécute toutes les étapes géomètriques et la carte 3D se charge de l'application de textures.

Les applications basées sur l'affichage de scènes en trois dimensions en temps réel sont de plus en plus présentes sur les machines généralistes. En effet, ces applications permettent à l'utilisateur de s'immerger totalement dans un monde virtuel ou de visualiser en trois dimensions des objets inexistants ou difficiles à observer dans la réalité (jeux en 3D, applications de CAO, visualisation des organes internes en médecine, ...). Dans toutes ces utilisations, le but est que l'utilisateur ne puisse plus faire la différence entre les images de synthèse et la réalité.

Même si des modèles plus complexes et plus fins on été développés, le secteur de l'image utilise souvent des modèles tridimensionnels à base de facettes, plus simples à gérer. A ces modèles sont associées des techniques qui permettent de rendre plus réalistes les images/scènes : lissage de Gouraud (peindre la surface comme si elle était courbe alors qu'elle est constituée de facettes), éclairage de Phong (prendre en compte dans le lissage de Gouraud des reflets), placage de texture de Catmull (coller une image sur les surfaces des modèles tridimensionnels, méthode du papier peint). Le constructeur qui va réaliser une machine est obligé d'adapter son modèle en fonction de la puissance de calcul et du temps qui lui est alloué.

Le dessin d'un objet texturé par un moteur 3D polygonal temps réel sur un écran se fait en 3 étapes [95] appelé pipeline graphique. Les deux phases principales sont les transformations géométriques et la ratérisation effectuée systématiquement par un coprocesseur graphique intégré à la carte vidéo. Récemment, les transformations géométriques ont également été intégrées à ces coprocesseurs.

Lors de l'étape géométrique, l'objet est découpé en triangles élémentaires plats. Ces triangles subissent alors rotations, translations, projections et calculs d'éclairage de leurs sommets. D'un point de vue calculatoire, les transformations géométriques consistent en opérations sur des matrices flottantes de dimension 4. Elles réalisent la transformation des coordonnées des objets 3D en des coordonnées en 2D pour pouvoir afficher ces objets sur un écran. Lors de l'étape d'application de texture, ces triangles sont balayés ligne par ligne et pour chaque point à afficher, est calculé un texel (point dans la texture) correspondant. L'étape de rastérisation qui consiste à dessiner ces objets à l'écran nécessite essentiellement du calcul entier et est

implémentée en matériel dans l'ensemble des cartes vidéos actuelles. Enfin, lors de l'étape de visibilité, on détermine à l'aide d'un tampon de profondeur quels sont les points qui sont effectivement devant les autres.

Représentation d'une scène Pour représenter les objets composant la scène, on décompose la surface de chacun d'eux en facettes triangulaires, afin d'approcher sa forme réelle, de la même façon que la forme d'une courbe peut être approchée par plusieurs droites ; plus le nombre de droites utilisé est grand, plus le dessin obtenu se rapproche de la courbe de départ. De même, le réalisme d'un objet 3D dépend directement du nombre de facettes qui le composent. Comme l'utilisateur veut toujours plus de réalisme, il faut pouvoir afficher en temps-réel un maximum de triangles par seconde afin d'afficher les scènes les plus complexes possible, tant en nombre d'objets qu'en nombre de triangles. La performance des machines actuelles ne permet toujours pas d'obtenir un réalisme parfait et la marge de progression nécessaire est encore très grande.

Le pipeline 3D L'affichage de la scène 3D n'est pas effectué directement par l'application. C'est le moteur de rendu qui se charge de cette opération, l'application se contentant de lui envoyer les coordonnées des triangles à afficher. La communication entre l'application et le moteur de rendu s'effectue par l'intermédiaire d'une interface de programmation (*API*), les plus utilisées étant OpenGL [77] et DirectX [60]. On fournit ainsi au programmeur une abstraction de l'affichage, certaines parties étant effectuées par du matériel spécialisé.

Etape géométrie Cette étape a pour but de déterminer diverses informations sur chaque sommet afin de permettre l'affichage du triangle correspondant par l'étage de rasterisation. Le nombre d'étages et la nature des étages dépend des besoins de l'application. Nous donnons ici les étages les plus couramment utilisés :

- transformation des sommets : dans cette étape, les coordonnées des sommets (sous forme de vecteurs) sont transformées à l'aide d'une multiplication de matrice 4x4 afin de les placer dans la scène (rotation, translation) et de les projeter dans le repère de l'observateur.
- *clipping* : l'application envoie au moteur uniquement les objets qui seront visibles. Cependant, certains objets ne sont que partiellement dans le champ de vision de la caméra et doivent donc être coupés aux frontières de l'écran afin de ne pas être inutilement traités par la suite. C'est le *clipping* qui se charge de cette opération.
- transformation des normales : les coordonnées des vecteurs normaux associés à chaque sommet vont être utilisés dans les étapes suivantes du pipeline afin de calculer la visibilité du triangle et l'éclairage aux sommets. Leurs coordonnées doivent donc être, elles-aussi, transformées de la même façon que celles des sommets.
- élimination des faces cachées : la normale à un triangle permet de déterminer, à l'aide d'un produit vectoriel, si le triangle fait face à l'observateur. Si ce n'est pas le cas, il n'a pas à être affiché, car il ne sera pas visible
- éclairage : l'intensité et la nature de l'éclairage sont calculées en chaque sommet.

Etape de rasterisation Lors de cette étape, on va dessiner le contenu de chaque triangle et calculer ainsi ce qu'il faut afficher à l'écran pour chaque pixel. Ceci se fait en balayant tous les pixels du triangle. Le balayage le plus utilisé est le balayage par ligne (*scanline*). On part du point le plus haut, et on descend sur les deux arêtes qui en partent, en balayant chaque ligne. Lorsque l'on atteint un sommet, on continue en balayant les deux autres arêtes.

L'étape de rasterisation commence par le transfert des triangle et la génération des arêtes. Lors de cet étage appliqué à chaque triangle, on calcule les pentes du parcours. Sur certaines machines, on élimine les

faces cachées à cet étage qui porte souvent le nom générique de *Setup*. Les autres étages sont appliqués à chaque pixel et réalisent l'éclairage (interpolation linéaire de la couleur de chacun des sommets), le calcul de perspective, le calcul de la position de chaque point dans les textures, le filtage de la texture, le calcul de la brume et l'anticrenalage.

Ensuite, on passe à l'étage de visibilité. Si deux objets différents se projettent sur le même pixel de l'écran, il est nécessaire de déterminer lequel est visible. Pour cela, on utilise l'algorithme du tampon de profondeur. On conserve en mémoire l'ensemble des coordonnées z de chaque pixel de l'écran qui ont été préalablement affichées (stockage dans le *z-buffer*). Lorsqu'on veut afficher un pixel, on s'assure qu'il se trouve devant celui qui a préalablement été affiché avant de l'écrire dans la mémoire écran (appelée aussi *frame Buffer*). Cet étage est également celui où sont gérés les problèmes de transparence. En effet, si un objet transparent se trouve devant un objet opaque, il faut que son effet soit pris en compte. Ceci implique que tous les objets transparents soient triés au préalable selon leur distance à l'observateur et soumis dans l'ordre, du plus proche au plus lointain (ou inversement).

Généralement, l'affichage de l'ensemble de l'image sur l'écran se fait d'un seul coup, après que l'image ait été entièrement calculée. Ceci permet d'avoir toujours à l'écran une image stable qui sera rafraîchie sur l'écran à une fréquence plus élevée. Pour cela, on utilise un mécanisme de *double-buffering* qui consiste à travailler avec deux zones mémoires utilisées alternativement pour l'affichage et le dessin.

Programme-test utilisé Pour évaluer la géométrie 3D polygonale temps-réel, nous avons considéré PMesa qui est une bibliothèque parallèle implémentant l'API OpenGL[2] développée dans notre équipe de recherche. PMesa est basé sur la bibliothèque open source Mesa et accélère les applications OpenGL sans nécessiter aucun changement de code. PMesa parallélise l'étage géomètrie du pipeline 3D, il utilise une parallélisation statique pour allouer les tâches aux différents flots[3]

Pour l'étape de transformations géométriques, nous utilisons *Glutspeed*, application qui envoie ses triangles à PMesa. Glutspeed utilise un mode de rendu qui correspond à ce qu'utilisent aujourd'hui les applications OpenGL de CAO et les jeux haut de gamme : transformations des sommets et des normales et éclairage par 3 sources de lumière dynamiques. Elle crée une *display list* avec un objet composé de 122000 sommets qui sera affiché trois fois.

Pour l'étape d'application des textures, des scènes (3D) ont été créées à l'aide de Strata Studio pro et visualisées à l'aide de l'afficheur QuickDraw3D fournit avec MacOS. *Teapot.full* est une théière recouverte d'une seule texture et permet donc de mesurer de façon élémentaire le comportement d'une machine faisant l'application de textures. L'autre scène, *room3*, est très complexe (100000 triangles) et comporte beaucoup d'objets recouverts chacun d'une texture différente.

Par ailleurs pour les autres scènes, les benchmarks utilisés sont extraits de jeux utilisant la réalité virtuelle et permettant à des joueurs d'interagir dans ces espaces à travers un réseau. Les benchmarks utilisés sont des enregistrements (*démos*) de parties jouées en réseau. Toutes ces applications utilisent l'API OpenGL. Nous avons choisi de prendre, dans la démo, une seule scène. Celle-ci a été choisie en analysant les statistiques (nombre de triangles et de textures) de toute la démo et en sélectionnant celle qui était la plus complexe. On peut trouver toutes les démos sur le site *www.voodooextreme.com*. Les programmes-tests sont la concaténation

[2] OpenGL est une API graphique 3D (*Application Programmer Interface*), i.e., une librairie de fonctions et de paramètres permettant d'effectuer le rendu d'objets graphiques en 2D et 3D. Elle est une des librairies graphiques les plus utilisées pour la représentation en temps réel de scènes en 3D.

[3] Dans nos différentes études, nous n'utilisons que des traces multiprocesseurs d'applications à équilibrage de charge statique. Les traces des triangles sont extraites à partir d'applications séquentielles, la distribution du travail étant entièrement réalisée par le simulateur.

Application	Nom de la scène	Résolution de l'écran	Pixels dessinés (million)	Nombre de triangles	Nombre de textures	Texture utilisées (Mo)
QuickDraw3D	*room3*	1280 × 1024	13	163000	24	1,5
QuickDraw3D	*teapot.full*	1280 × 1024	2,8	10000	1	6
MacQuake	*quake*	1152 × 870	2	7400	954	5,2
Quake2	*massive11255*	1600 × 1200	8	13000	1055	1
Quake2	*32massive11255*	1600 × 1200	8	13000	1055	3,4
Half Life	*blowout775*	1600 × 1200	5,9	5947	1778	0,8
Half Life	*truc640*	1600 × 1200	8,3	12195	1530	1,2

TAB. A.1: Caractérisques des programmes tests.

de la démo et du numéro de l'image utilisée. La scène *quake* est extraite du jeu Quake 1 d'Id Software et de la demo *bigass1* sous MacOS. La scène *Massive11255* est extraite de Quake 2 sous Windows NT. Les démos *blowout* et *truc1* utilisent le jeu Half-Life de l'éditeur Valve. Les caractéristiques de ces scènes sont résumées table A.1.

Références du mémoire

[1] K. Akeley. RealityEngine graphics. *Computer Graphics*, pages 109–116, 1993.

[2] T. Austin and D. Burger. Simplescalar. *Tutorial - International Symposium on Microarchitecture*, 1997.

[3] J.-L. Baer and T-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. *International Conference on Supercomputing*, pages 176–186, 1991.

[4] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. *International Symposium on Microarchitecture*, pages 245–257, 2000.

[5] J.-L. Béchennec. Asf: A teaching and research objectoriented simulation tool for computer architecture design and performance evaluation. *Workshop on Computer Architecture Education of Internationnal Symposium on Computer Architecture*, 1998.

[6] D. Bernstein, D. Cohen, A. Freund, and D. Maydan. Compiler techniques for data prefetching on the powerpc. *International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, 1995.

[7] R. Bhargava, L. Kurian John, B. L. Evans, and R. Radhakrishnan. Evaluating MMX technology using DSP and multimedia applications. *International Symposium on Microarchitecture*, pages 37–46, 1998.

[8] A. J.C. Bik, P. M. Grey, M. Girkar, and X. Tian. Experiments with automatic vectorization for the pentium(r) 4 processor. *Workshop on Compilers for Parallel Computers*, 2001.

[9] F. Bodin, C. Eisenbeis, W. Jalby, and D. Windheiser. A Quantitative Algorithm for Data Locality Optimization. *Code Generation-Concepts, Tools, Techniques*, 1992.

[10] F. Bodin, G. L. Fol, and F. Raimbault. Oco, manuel de l'utilisateur. *Publication interne IRISA n930*, 1995.

[11] K. Bolland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, pages 59–66, 1994.

[12] V. Branger. Etude du préchargement de données sur les caches de second niveau dans les processeurs superscalaires. *Doctorat de l'Université de Paris-Sud*, 1997.

[13] V. Branger and N. Drach. Etude de la localité des références sur le second niveau de cache. *Symposium Architectures Nouvelles des Machines*, 1996.

[14] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *International Symposium on Computer Architecture*, pages 83–94, 2000.

[15] D. Burger, J. Goodman, and A. Kagi. Memory bandwidth limitations of future processors. *International Symposium on Computer Architecture*, pages 78–89, 1996.

[16] J. Butts and G. Sohi. A static power model for architects. *International Symposium on Microarchitecture*, pages 191–201, 2000.

[17] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. H. Albonesi. An adaptive issue queue for reduced power at high performance. *Workshop on Power-Aware Computer Systems, held at the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[18] T.-F Chen. Data prefetching for high-performance processors. Technical Report TR-93-07-01, University of Washington, USA, 1993.

[19] T.-F. Chen. An effective programmable prefetch engine for on-chip caches. *International Symposium on Microarchitecture*, pages 237–242, 1995.

[20] J. Corbal, R. Espasa, and M. Valero. Dlp+tlp processors for the next generation of media workloads. *International Symposium on High-Performance Computer Architecture*, pages 219–228, 2001.

[21] M. Cox, N. Bhandari, and M. Shantz. Multi-level texture caching for 3D graphics hardware. *International Symposium on Computer Architecture*, pages 86–97, 1998.

[22] V. Cuppu, B. L. Jacob, B. Davis, and T. N. Mudge. A performance comparison of contemporary DRAM architectures. *Internationnal Symposium on Computer Architecture*, pages 222–233, 1999.

[23] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.

[24] P. Denning. On modeling program behavior. *Spring Joint Computer Conference*, pages 937–944, 1972.

[25] D. DeVries and C. G. Lee. A vectorizing suif compiler. *First SUIF Compiler Workshop*, pages 59–67, 1996.

[26] A. S. Dhodapkar and J. E. Smith. Managing configurable hardware via dynamic working set analysis. *International Symposium on Computer Architecture*, 2002.

[27] C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. *International Parallel and Distributed Processing Symposium*, 2000.

[28] N. Drach. Hardware implementation issues of data prefetching. *International Conference on Supercomputing*, pages 245–254, 1995.

[29] N. Drach, J.-L. Béchennec, and O. Temam. Increasing hardware data prefetching performance using the second-level cache. *Journal of Systems Architecture, éditeur Elsevier Science. A paraître.*

[30] A. Eustace and A. Srivastava. Atom: A flexible interface for building high performance program analysis tools. *USENIX Technical COnference on UNIX and Advanced Computing Systems*, pages 303–314, 1995.

[31] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The realization. *Eurographics Workshop on Graphics Hardware*, pages 57–68, 1997.

[32] J. Ferrante, V. Sarkar, and W. Thrash. On Estimating and Enhancing Cache Effectiveness (Extended Abstract). *Workshop on Programming Languages and Compilers for Parallel Computing*, 1991.

[33] R. J. Fisher and H. G. Dietz. Compiling for SIMD within a register. *Languages and Compilers for Parallel Computing*, pages 290–304, 1998.

[34] D. Gannon, J. K. Lee, B. Shei, S. R. Sarukkai, S. Narayana, N. Sundaresan, D. Atapattu, and F. Bodin. Sigma ii: A tool kit for building parallelizing compiler and performance analysis systems. *Programming Environments for Parallel Computing*, pages 17–36, 1992.

[35] P. N. Glaskowsky. Networking gets xstream. *Microprocessor Report*, 2000.

[36] E. H. Gornish and A. V. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. *International Conference on Parallel Processing*, pages 281–284, 1994.

[37] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. *International Symposium on Computer Architecture*, pages 108–120, 1997.

[38] L. Harrison and S. Mehrotra. A data prefetch mechanism for accelerating general computation. *Technical Report 1351, CSRD, University of Illinois at Urbana-Champaign, USA*, 1994.

[39] IBM. Power4 system microarchitecture. Technical report, IBM Corporation, 2001.

[40] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1998.

[41] G. Irlam. Spa package. 1991.

[42] ISO/IEC. Mpeg-4 version 1 overview. *Document ISO/IEC JTC1/SC29/WG11 N2323, MPEG-4 Version 1 Overview*, 1998.

[43] D. A. Patterson J. L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1996.

[44] J. H. Patel J. W. C. Fu and B. L. Janssens. Stride directed prefetching in scalar processors. *International Symposium of Microarchitecture*, pages 102–110, 1992.

[45] Y. Jégou and O. Temam. Speculative prefetching. *International Conference on Supercomputing*, pages 57–66, 1993.

[46] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[47] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *International Symposium on Computer Architecture*, pages 364–373, 1990.

[48] S. Jourdan, P. Sainrat, and D. Litaize. Exploring configurations of functional units in an out-of-order superscalar processor. *International Symposium on Computer Architecture*, pages 117–125, 1995.

[49] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A thrid level of adaptivity for branch prediction. *International Symposium on Computer Architecture*, pages 155–166, 1998.

[50] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5):145–156, 2000.

[51] C. G. Lee and M. G. Stoodley. Simple vector microprocessors for multimedia applications. *International Symposium on Microarchitecture*, pages 25–36, 1998.

[52] C. Limousin. Multiflot simultané et multimédia : outil, étude et optimisation. *Doctorat de l'Université de Paris-Sud*, 2001.

[53] C. Limousin, N. Drach, and J.-L. Béchennec. Politiques d'émission pour un processeur smt. *6ème Symposium en Architectures Nouvelles de Machines*, 2000.

[54] C. Limousin, J. Sebot, A. Vartanian, and N. Drach. Amélioration des performances des transformations géométriques 3d temps réel sur un processeur smt simd. *7ème Symposium en Architectures Nouvelles de Machines*, 2001.

[55] C. Limousin, J. Sebot, A. Vartanian, and N. Drach. Improving 3d geometry transformations on a simultaneous multithreaded simd processor. *International Conference on Supercomputing*, pages 236–245, 2001.

[56] C. Limousin, A. Vartanian, and J.-L. Béchennec. Popspy: A powerpc instrumentation tool for multiprocessor simulation. *European Conference on Parallel Computing*, pages 262–256, 1999.

[57] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[58] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Corporation*, 2002.

[59] K. S. McKinley. *Automatic and Interactive Parallelization.* PhD thesis, Rice University, Technical Report CRPC-TR92214, USA, 1992.

[60] Microsoft. *Microsoft DirectX 3 SDK : Direct3D Overview*, 1996.

[61] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[62] J. S. Montrym, D. R. Baum, D. L. Dignam, and Christopher J. Migdal. InfiniteReality : A real-time graphics system,. *Computer Graphics*, pages 293–302, 1997.

[63] H. Nguyen and L. K. John. Exploiting simd parallelism in dsp and multimedia algorithm using altivec technology. *International Conference on Supercomputing*, pages 11–20, 1999.

[64] H. Oehring, U. Sigmund, and T. Ungerer. Simultaneous multithreading and multimedia. *Workshop on Multi-Threaded Execution, Architecture and Compilation*, 1999.

[65] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. *International Symposium on Computer Architecture*, pages 24–33, 1994.

[66] B. Paul. The mesa 3d-graphic library. http://www.mesa3d.org, Jan 1999.

[67] B. Peroche, J. Argence, D. Ghazanfarpour, and D. Michelucci. *La synthèse d'image.* Hermès, 1988.

[68] P. Ranganathan, S. V. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. *International Symposium on Computer Architecture*, pages 124–135, 1999.

[69] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[70] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. *International Symposium on Computer Architecture*, pages 264–273, 1997.

[71] J. Sebot and N. Drach. Simd isa extensions : Tradeoff between power consumption and performance on a superscalar processor. *In Kool Chips Workshop, International Symposium on Microarchitecture*, 2000.

[72] J. Sebot and N. Drach. Extensions simd et superscalaire dans l'ordre : une solution pour réduire la consommation des applications multimédia. *7ème Symposium en Architectures Nouvelles de Machines*, 2001.

[73] J. Sebot and N. Drach. Memory bandwidth: the true bottleneck of simd multimedia performance on a superscalar processor. *European Conference on Parallel Computing*, 2150:439–447, 2001.

[74] J. Sebot and N. Drach. Simd isa extensions : Reducing power consumption on a superscalar processor for multimedia applications. *International Symposium Cool Chips*, pages 109–119, 2001.

[75] J. Sebot and N. Drach. Simd isa extensions: Power efficiency on multimedia. *IEICE Transactions on Electronics*, pages 297–303, 2002.

[76] J. Sebot, A. Vartanian, J.-L. Béchennec, and N. Drach. A parallel algorithm for 3d geometry transformations in opengl. *European Conference on Parallel Computing*, pages 659–662, 1999.

[77] Mark Segal and Kurt Akelay. *The OpenGL Graphics System*, 1996.

[78] A. Seznec and F. Lloansi. Performance impact of the l2 contention on out-of-order execution superscalar processors. *IEEE TCCA Newsletter*, 1997.

[79] A. J. Smith. Cache memories. *ACM Computing Surveys, 14(3):473-530*, 1982.

[80] O. Temam and N. Drach. Software assistance for data caches. *International Symposium on High Performance Computer Architectures*, pages 154–163, 1995.

[81] O. Temam and N. Drach. Software assistance for data caches. *Future Generation Computer Systems - Version étendue*, 1996.

[82] O. Temam and Y. Jégou. Using virtual lines to enhance locality exploitation. *International Conference on Supercomputing*, pages 344–352, 1994.

[83] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *Internationnal Symposium on Computer Architecture*, pages 191–202, 1996.

[84] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *International Symposium on Computer Architecture*, pages 392–403, 1995.

[85] S. VanderWiel and D. Lilja. When caches aren't enough: Data prefetching techniques. *IEEE Computer, 30(7):23-30*, 1997.

[86] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[87] A. Vartanian. Architectures parallèles pour la 3d à base de composants faible coût. *Doctorat de l'Université de Paris-Sud*, 2000.

[88] A. Vartanian, J.-L. Béchennec, and N. Drach. Evaluation of high performance multicache parallel texture mapping. *International Conference on Supercomputing*, pages 289–296, 1998.

[89] A. Vartanian, J.-L. Béchennec, and N. Drach. Two schemes to improve the performance of sort-last 3d parallel rendering machine with texture caches. *European Conference on Parallel Computing*, pages 757–760, 1999.

[90] A. Vartanian, J.-L. Béchennec, and N. Drach. The best distribution for a parallel opengl 3d engine with texture caches. *International Symposium on High-Performance Computer Architecture*, pages 399–408, 2000.

[91] N. Vijaykrishnan, M. T. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. *International Symposium on Computer Architecture*, pages 95–106, 2000.

[92] J. Voldman and L. W. Hoevel. The software-cache connection. *IBM Journal of Research and Development*, 25(6):877–893, 1981.

[93] J. Voldman and L. W. Hoevel. The software-cache interaction. *IBM Journal of R&D, vol. 25*, pages 877–893, 1981.

[94] J. Voldman, B. Mandelbrot, L. W. Hoevel, J. Knight, and P. L. Rosenfeld. Fractal nature of software-cache interaction. *IBM Journal of Research and Development*, 27(2):164–170, 1983.

[95] A. Watt and M. Watt. Advanced animation and rendering techniques. *Addison Wesley, first edition*, 1992.

[96] L. Williams. Pyramidal parametrics. *Computer Graphics*, pages 1–9, 1983.

[97] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *Conference on Programming Languages Design and Implementation*, pages 30–44, 1991.

[98] A. Wolfe and R. Boleyn. Two-ported cache alternatives for superscalar processors. *International Symposium on Microarchitecture*, pages 41–48, 1993.

[99] M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley Publishing Company*, 1996.

[100] C.-L. Yang, B. Sano, and A. R. Lebeck. Exploiting instruction level parallelism in geometry processing for three dimensional graphics applications. *International Symposium on Microarchitecture*, pages 14–24, 1998.

[101] K. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro, 16(2):28-41*, 1996.

[102] H. Zima and B. Chapman. Supercompilers for parallel and vector computers. *ACM Press Frontier Series, Addison-Wesley*, 1990.

RAPPORTS INTERNES AU LRI - ANNEE 2002

N°	Nom	Titre	Nbre de pages	Date parution
1300	COCKAYNE E J FAVARON O MYNHARDT C M	OPEN IRREDUNDANCE AND MAXIMUM DEGREE IN GRAPHS	15 PAGES	01/2002
1301	DENISE A	RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION D'UNE HABILITATION A DIRIGER DES RECHERCHES	81 PAGES	01/2002
1302	CHEN Y H DATTA A K TIXEUIL S	STABILIZING INTER-DOMAIN ROUTING IN THE INTERNET	31 PAGES	01/2002
1303	DIKS K FRAIGNIAUD P KRANAKIS E PELC A	TREE EXPLORATION WITH LITTLE MEMORY	22 PAGES	01/2002
1304	KEIICHIROU K MARCHE C URBAIN X	TERMINATION OF ASSOCIATIVE-COMMUTATIVE REWRITING USING DEPENDENCY PAIRS CRITERIA	40 PAGES	02/2002
1305	SHU J XIAO E WENREN K	THE ALGEBRAIC CONNECTIVITY, VERTEX CONNECTIVITY AND EDGE CONNECTIVITY OF GRAPHS	11 PAGES	03/2002
1306	LI H SHU J	THE PARTITION OF A STRONG TOURNAMENT	13 PAGES	03/2002
1307	KESNER D	RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION D'UNE HABILITATION A DIRIGER DES RECHERCHES	74 PAGES	03/2002
1308	FAVARON O HENNING M A	UPPER TOTAL DOMINATION IN CLAW-FREE GRAPHS	14 PAGES	04/2002
1309	BARRIERE L FLOCCHINI P FRAIGNIAUD P SANTORO N	DISTRIBUTED MOBILE COMPUTING WITH INCOMPARABLE LABELS	16 PAGES	04/2002
1310	BARRIERE L FLOCCHINI P FRAIGNIAUD P SANTORO N	ELECTING A LEADER AMONG ANONYMOUS MOBILE AGENTS IN ANONYMOUS NETWORKS WITH SENSE-OF-DIRECTION	20 PAGES	04/2002
1311	BARRIERE L FLOCCHINI P FRAIGNIAUD P SANTORO N	CAPTURE OF AN INTRUDER BY MOBILE AGENTS	16 PAGES	04/2002
1312	ALLARD G AL AGHA K	ANALYSIS OF THE OSSC MECHANISM IN A NON-SYNCHRONOUS TRANSMISSION ENVIRONMENT	12 PAGES	04/2002
1313	FOREST J	A WEAK CALCULUS WITH EXPLICIT OPERATORS FOR PATTERN MATCHING AND SUBSTITUTION	70 PAGES	05/2002
1314	COURANT J	STRONG NORMALIZATION WITH SINGLETON TYPES	19 PAGES	05/2002
1315	COURANT J	EXPLICIT UNIVERSES FOR THE CALCULUS OF CONSTRUCTIONS	21 PAGES	05/2002
1316	KOUIDER M LONC Z	STABILITY NUMBER AND (a,b)-FACTORS IN GRAPHS	12 PAGES	05/2002
1317	URBAIN X	MODULAR AND INCREMENTAL PROOFS OF AC-TERMINATION	20 PAGES	05/2002

RAPPORTS INTERNES AU LRI - ANNEE 2002

N°	Nom	Titre	Nbre de pages	Date parution
1318	THION V	A STRATEGY FOR FREE-VARIABLE TABLEAUX FOR VARIANTS OF QUANTIFIED MODAL LOGICS	12 PAGES	05/2002
1319	LESTIENNES G GAUDEL M C	TESTING PROCESSES FROM FORMAL SPECIFICATIONS WITH INPUTS, OUTPUTS AND DATA TYPES	16 PAGES	05/2002
1320	PENT C SPYRATOS N	UTILISATION DES CONTEXTES EN RECHERCHE D'INFORMATIONS	46 PAGES	05/2002
1321	DELORME C SHU J	UPPER BOUNDS ON THE LENGTH OF THE LONGEST INDUCED CYCLE IN GRAPHS	20 PAGES	05/2002
1322	FLANDRIN E LI H MARCZYK A WOZNIAK M	A NOTE ON A GENERALISATION OF ORE'S CONDITION	8 PAGES	05/2002
1323	BACSO G FAVARON O	INDEPENDENCE, IRREDUNDANCE, DEGREES AND CHROMATIC NUMBER IN GRAPHS	8 PAGES	05/2002
1324	DATTA A K GRADINARIU M KENITZKI A B TIXEUIL S	SELF-STABILIZING WORMHOLE ROUTING ON RING NETWORKS	20 PAGES	06/2002
1325	DELAET S HERAULT T JOHNEN C TIXEUIL S	ACTES DE LA JOURNEE RESEAUX ET ALGORITHMES REPARTIS, 20 JUIN 2002	52 PAGES	06/2002
1326	URBAIN X	MODULAR AND INCREMENTAL AUTOMATED TERMINATION PROOFS	32 PAGES	06/2002
1327	BEAUQUIER J JOHNEN C	ANALYZE OF RANDOMIZED SELF-STABILIZING ALGORITHMS UNDER NON-DETERMINISTIC SCHEDULER CLASSES	18 PAGES	06/2002
1328	LI H SHU J	PARTITIONING A STRONG TOURNAMENT INTO k CYCLES	14 PAGES	07/2002
1329	BOUCHERON S	RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION D'UNE HABILITATION A DIRIGER DES RECHERCHES	97 PAGES	08/2002
1330	JOHNEN C	OPTIMIZATION OF SERVICE TIME AND MEMORY SPACE IN A SELF-STABILIZING TOKEN CIRCULATION PROTOCOL ON ANONYMOUS UNIDIRECTIONAL RINGS	21 PAGES	09/2002
1331	LI H SHU J	CYCLIC PARTITION OF STRONG TOURNAMENTS	15 PAGES	09/2002
1332	TZITZIKAS Y SPYRATOS N	RESULT FUSION BY MEDIATORS USING VOTING AND UTILITY FUNCTIONS	30 PAGES	09/2002
1333	AL AGHA K	RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION D'UNE HABILITATION A DIRIGER DES RECHERCHES	63 PAGES	10/2002
1334	ALVAREZ-HAMELIN J I FRAIGNIAUD P	REDUCING PACKET-LOSS BY TAKING LONG RANGE DEPENDENCES INTO ACCOUNT	20 PAGES	10/2002

RAPPORTS INTERNES AU LRI - ANNEE 2002

N°	Nom	Titre	Nbre de pages	Date parution
1335	EGAWA Y ENOMOTO H FAUDREE R J LI H SCHIERMEYER I	TWO-FACTORS EACH COMPONENT OF WHICH CONTAINS A SPECIFIED VERTEX	16 PAGES	10/2002
1336	LI H WOZNIAK M	A NOTE ON GRAPHS CONTAINING ALL TREES OF GIVEN SIZE	10 PAGES	10/2002
1337	ENOMOTO H LI H	PARTITION OF A GRAPH INTO CYCLES AND DEGENERATED CYCLES	10 PAGES	10/2002
1338	BALISTER P N KOSTOCHKA A V LI H SCHELP R H	BALANCED EDGE COLORINGS	20 PAGES	10/2002
1339	HAGGKVIST R LI H	LONG CYCLES IN GRAPHS WITH SOME LARGE DEGREE VERTICES	16 PAGES	10/2002