ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Exécution en parallèle d'un programme

Djallal Rahmoune, David Parello et Bernard Goossens

DALI, Université de Perpignan Via Domitia 66860 Perpignan Cedex 9 France,
LIRMM, CNRS : UMR 5506 - Université Montpellier 2 34095 Montpellier Cedex 5 France,
djallal.rahmoune@etudiant.univ-perp.fr, david.parello@univ-perp.fr,
bernard.goossens@univ-perp.fr

Résumé
Cet article présente un modèle d'exécution des programmes basé sur un découpage en blocs de base. Pour être mis en œuvre, il nécessite l'emploi d'un processeur à beaucoup de cœurs (*many-core*). Ses spécificités sont i) l'extraction en parallèle (d'une fonction et de ce qui suit son appel, des itérations d'une boucle vectorisable et des deux alternatives de tout saut conditionnel), ii) un renommage étendu à la mémoire et parallèlisé, iii) une exécution en désordre de toutes les instructions de la trace et iv) un retrait en parallèle. Le modèle présenté est évalué à partir du simulateur PerPI appliqué à la suite de benchmarks PBBS composée de programmes issus d'algorithmes parallèles. Il est comparé au modèle implanté actuellement dans les processeurs à quelques cœurs (*multi-core*). L'évaluation montre d'une part que le modèle présenté capture mieux le parallélisme d'instructions que le modèle actuel et d'autre part que la quantité de parallélisme captée augmente avec la taille des données des algorithmes parallèles.

Mots-clés : parallèlisme d'instructions, extraction parallèle, renommage parallèle, processeur many-core, benchmarks PBBS.

---

1. Introduction

Depuis près de vingt ans, nos processeurs hautes performances exécutent spéculativement et en désordre. Ils appliquent un algorithme inventé par Tomasulo [6] en 1967. Ce faisant, ils exploitent en partie le parallélisme d'instructions. En quelque sorte, le matériel parallélise l'exécution, à une toute petite échelle. Dans les années 90, le parallélisme d'instructions a été très étudié. On s'est aperçu que le code fourni par le compilateur est naturellement très peu parallèle. Tjaden et Flynn [5] ont montré qu'on ne pouvait pas exécuter plus de deux instructions en parallèle en moyenne. En renommant les destinations en registres et en prédisant les sauts, c'est-à-dire en éliminant la plupart des fausses dépendances de données (dépendances Ecriture Après Ecriture ou EAE et dépendances Ecriture Après Lecture ou EAL) et des dépendances de contrôle (direction des sauts conditionnels), Wall [8] a mesuré qu'on peut atteindre une moyenne de cinq instructions exécutées par cycle.
Le compilateur produit un code imposé par l'architecture. Les données et résultats du programme sont mappés sur un petit ensemble de registres, ce qui génère d'innombrables dépendances EAE et EAL. De plus, l'enchaînement des appels de fonctions est implémenté à partir d'une pile. L'empilement et le dépilement créent encore des dépendances, dues aux manipulations du sommet de pile mais aussi au partage de l'espace de stockage. Enfin, les structures de

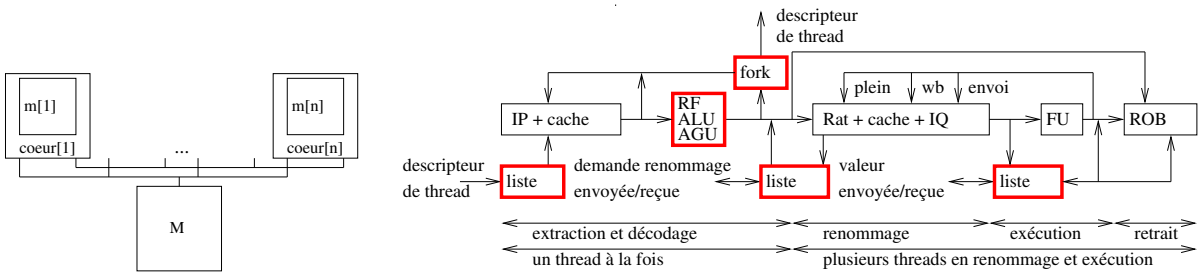

FIGURE 1 – Un modèle de processeur à beaucoup de cœurs

contrôle des langages évolués, boucles, tests, fonctions, s'expriment toutes à partir de l'instruction de saut conditionnel. Le mécanisme d'accès au code, basé sur un pointeur de programme qui progresse séquentiellement le long de l'exécution, rencontre en permanence des ruptures du flot d'instructions par les sauts. Tout ceci fait que d'un algorithme présentant un certain degré de parallélisme, on aboutit par compilation à un code totalement sérialisé par ses dépendances. Porada [3] explique que tout calcul est inséré dans un carcan architectural. Pour retrouver le parallélisme initial éventuel, il faut faire abstraction de ce carcan. C'est en partie ce que l'on fait dans les processeurs actuels en renommant les registres et en prédisant les sauts : on re-parallélise en éliminant certaines dépendances, mais trop partiellement.

Pour éliminer les dépendances, on emploie deux techniques. Pour les fausses dépendances, on renomme. En supposant qu'on puisse donner un nom unique à toute destination, les dépendances EAE et EAL disparaissent. Pour les vraies dépendances (dépendance Lecture Après Ecriture ou LAE), on prédit. Plutôt que d'attendre un résultat, on peut le prédire. La prédiction est surtout utilisée pour éliminer les dépendances de contrôle.

Cet article montre qu'en utilisant le renommage et la prédiction et en changeant le mode d'extraction du code on peut paralléliser des exécutions. La section 2 présente un modèle d'exécution en parallèle. La section 3 rapporte les résultats d'une évaluation de ce modèle en comparaison avec le modèle d'exécution des processeurs actuels.

2. Un modèle d'exécution parallèle

2.1. Un modèle de processeur à beaucoup de cœurs (*many-core*)

La figure 1 montre à gauche la microarchitecture adaptée au modèle d'exécution parallèle que nous allons définir. Le processeur se compose de n cœurs identiques reliés par un réseau d'interconnexion permettant des communications point à point. Chaque cœur dispose d'une mémoire privée et partage une mémoire commune. Le processeur ne contient aucun mécanisme de gestion de cohérence de la mémoire car le modèle d'exécution parallèle, par le renommage de la mémoire, impose un écrivain unique pour chaque destination.

Chaque cœur (voir la figure 1, à droite) a les mêmes fonctionnalités qu'un cœur actuel mais sans opérateur vectoriel ni prédicteur de saut.

2.2. Un modèle d'exécution basé sur les *threads*

Chaque cœur extrait, renomme et exécute des *threads* (voir la figure 1 à droite). Un *thread* est une suite d'instructions contiguës terminée par une instruction de contrôle (saut inconditionnel, saut conditionnel, appel de fonction, retour de fonction). La proportion d'instructions de contrôles dans une exécution varie entre 10% pour les applications flottantes à 20% pour les applications entières [2]. Les *threads* sont très courts : 5 à 10 instructions en moyenne.

```
1  somme: pushq   %rbx              // sauver t
2         pushq   %rbp              // sauver n
3         subq    $8, %rsp          // allouer temp
4         cmpq    $2, %rbp          // n==2
5         jne     .L2               // si (n!=2) vers .L2
6         movq    8(%rbx), %rax     // rax=t[1]
7         addq    (%rbx), %rax      // rax+=t[0]
8         jmp     .L3               // vers .L3
9  .L2:   shrq    %rbp              // rbp=n/2
10        call    somme             // somme(t,n/2)
11        movq    %rax, 0(%rsp)     // temp=somme(t,n/2)
12        leaq    (%rbx,%rbp,8), %rbx // rbx=&t[n/2]
13        call    somme             // somme(&t[n/2],n/2)
14        addq    0(%rsp), %rax     // rax+=temp
15 .L3:   addq    $8, %rsp          // libérer temp
16        popq    %rbp              // restaurer n
17        popq    %rbx              // restaurer t
18        ret                       // somme(t,n) dans rax
```

FIGURE 2 – Une réduction de somme en assembleur X86

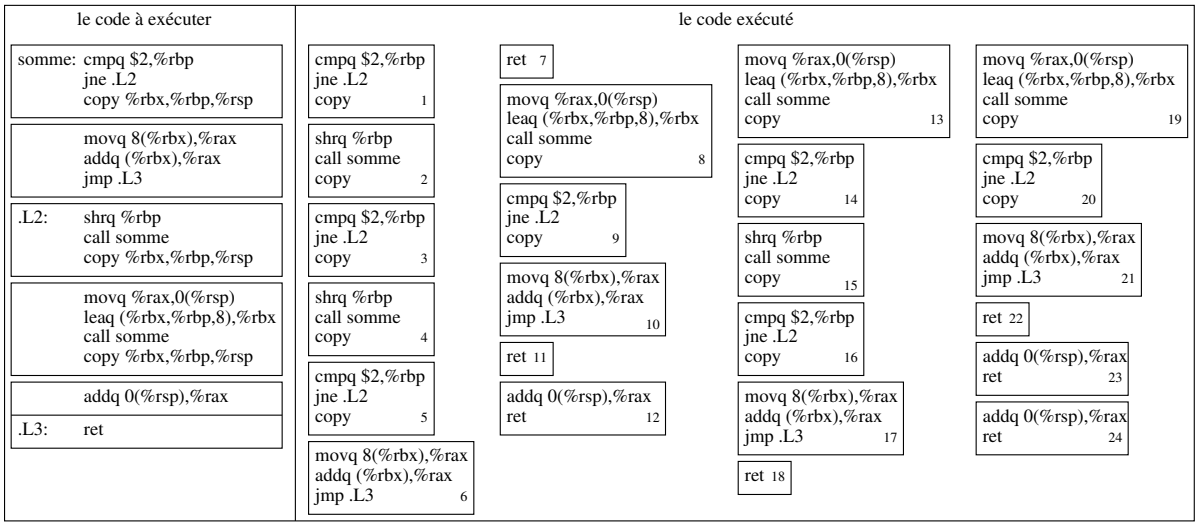

FIGURE 3 – Un code assembleur x86 découpé en *threads*

Le mécanisme d'extraction extrait des *threads* complets. Chaque cœur extrait un *thread* à la fois (en un ou plusieurs accès contigus) et en renomme les destinations. Puis, quand une instruction de contrôle est décodée, l'unité de *fork* crée le ou les *threads* suivants. Un *thread*, une fois créé, est mis en attente dans la liste de l'étage d'extraction du cœur où il est placé (liste de gauche sur la figure 1). Dans trois situations, la fin d'un *thread* donne naissance à plusieurs *threads* :

— à l'appel d'une fonction, on crée un *thread* appelé et un *thread* de reprise après le retour ;
— à l'entrée d'une boucle vectorisable, on crée un *thread* par itération et un *thread* de sortie ;
— pour un saut conditionnel, on crée un *thread* du saut pris et un autre du saut non pris.

Les deux premières situations parallélisent. La troisième fait de l'exécution gloutonne des sauts conditionnels (*eager execution* [7]), ce qui permet de s'affranchir d'un prédicteur de sauts tout en ayant l'équivalent d'une prédiction parfaite.

Un *thread* créateur place le *thread* qu'il crée sur son propre cœur. Quand il crée plusieurs *threads*, il place les autres *threads* sur d'autres cœurs de son choix, où ils attendent d'être extraits.

Un *thread* créé est représenté par un descripteur qui comprend l'adresse de sa première instruction, des liens vers d'autres *threads* et une copie de registres. Les registres copiés sont choisis par le compilateur. Il en fait copier le moins possible pour raccourcir le message de création de *thread*. Mais il fait copier tout ce qui est utilisé par le *thread* créé et ses successeurs pour les rendre plus indépendants de leur créateur car les registres non copiés mais utilisés nécessitent une communication inter-*thread*. Par exemple, le compilateur peut copier le pointeur de pile du *thread* créateur d'un appel de fonction vers le code après le retour. Il peut de la même façon

transmettre des registres non volatiles.
Cette façon de dupliquer les registres s'apparente à de la prédiction de valeur. La duplication est en quelque sorte une anticipation sure qui élimine de vraies dépendances des consommateurs de copies envers leurs producteurs. Comme c'est le compilateur qui choisit les registres à copier, et par complémentarité ceux à ne pas copier, il rend les *threads* dépendants ou non.
Le code après le retour dépend en général du *thread* interne à la fonction appelée parce qu'il en consomme le résultat, placé par exemple dans un registre (soit `r`). Cette dépendance est préservée en ne copiant pas, à l'appel, le registre `r` du *thread* d'appel au *thread* après le retour.
La figure 2 montre le code X86 d'une réduction de somme (somme des éléments d'un vecteur $t[n]$ ($n = 2^p$) en divisant-pour-régner). La figure 3 montre le découpage en *threads*.
Les *threads* de la figure 3 ne font pas apparaître les instructions `push`, `pop` et celles déplaçant le sommet de pile (`subq 8, rsp` et `addq 8, rsp`). Le maintien du pointeur de pile est inutile car il est naturellement assuré par sa copie de *thread* à *thread*. L'instruction `ret`, quant à elle, ne sert qu'à délimiter le *thread* de retour car l'adresse de retour n'est pas empilée.
L'exécution se compose de 24 *threads*. La figure 4 montre l'arbre de dépendances des *threads* (les parties haute et basse sont liées). Les branches coupées sont les chemins annulés des sauts conditionnels. Chaque rectangle contient un *thread* représenté par ses instructions et son numéro. Les flèches indiquent une création (par exemple, le *thread* 4 crée les *threads* 5 et 8).
L'arbre correspond à l'ordre partiel d'exécution. Le *thread* 2 crée les *threads* 3 et 13, qui s'exécutent en parallèle. Le parallélisme est réel parce que les *threads* 3 et 13 ont été rendus indépendants par la copie des registres `rbx`, `rbp` et `rsp`. Ainsi, les *threads* 14 à 23 sont indépendants des *threads* 3 à 12. Seule l'instruction *movq* du *thread* 13 dépend, par `rax`, de l'instruction *addq* du *thread* 12. Cette dépendance entraine une communication du *thread* 13 vers son prédécesseur (le *thread* 12) pour y rechercher la dernière valeur de `rax`.

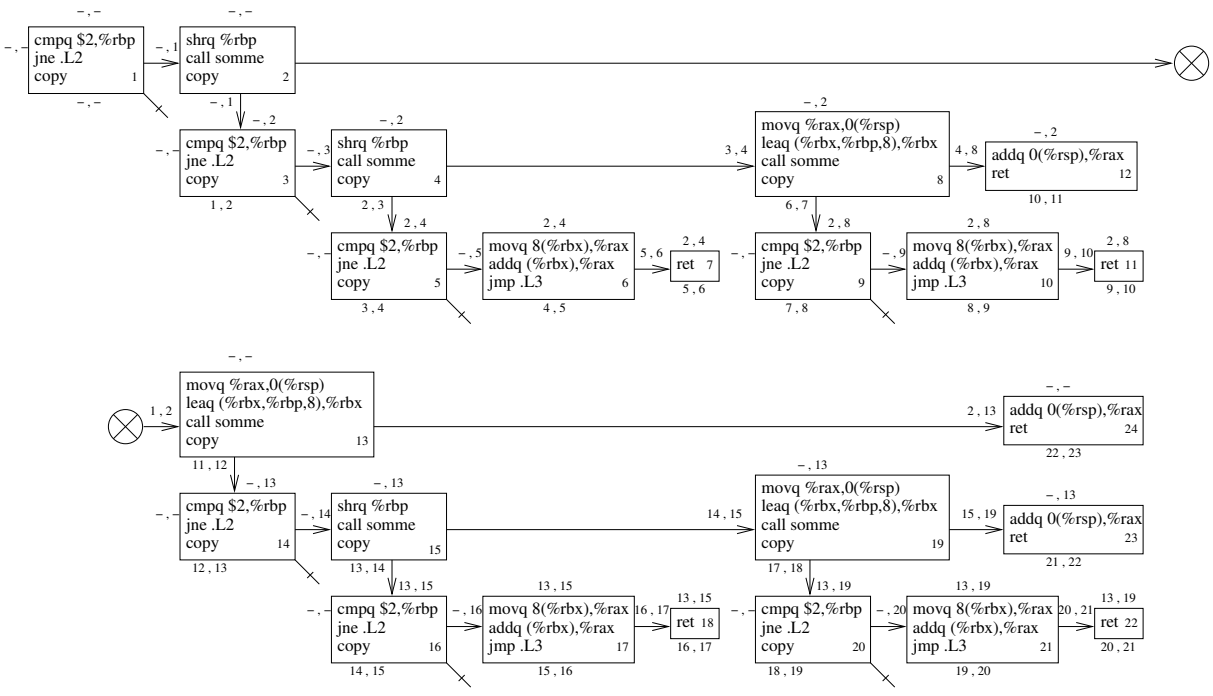

FIGURE 4 – L'arbre de dépendances des *threads*

2.3. La parallélisation et l'extension du renommage

2.3.1. Le renommage des destinations

Tout est renommé, la mémoire comme les registres.
Par l'évolution du pointeur de pile, deux appels de même niveau se partagent le même emplacement en mémoire pour y placer leurs variables `temp` respectives. Cela crée des dépendances EAE et EAL que le renommage étendu à la mémoire élimine. Ainsi, toutes les variables `temp` d'un même niveau d'appel sont disjointes.
Chaque *thread* renomme ses destinations indépendammment des autres *threads*. Sur la figure 1, la première partie du pipeline d'un cœur travaille sur un seul *thread*. L'extraction et le renommage des destinations de ce *thread* se poursuit jusqu'au décodage de son instruction de fin. Au fur et à mesure, les instructions entrent dans la seconde partie du pipeline (renommage des sources, exécution et retrait), où elles se mélangent à celles de *threads* antérieurs encore actifs.
Le renommage d'une destination alloue un emplacement dans la mémoire privée du cœur.

2.3.2. La parallélisation du renommage

La parallélisation de l'extraction induit celle du renommage. Par exemple, le *thread* de continuation effectue ses renommages en même temps ou même avant les *threads* de l'appel.
Les noms donnés aux destinations en parallèle par plusieurs *threads* doivent tous être uniques. Ils sont construits à partir de l'identité du *thread* et du numéro de l'instruction dans le *thread*. L'unicité vient de celle des identités des *threads* au sein d'un cœur. L'identité d'un *thread* est celle du cœur sur lequel il est créé complétée d'un numéro d'ordre. On crée des *threads* en parallèle sur différents cœurs ayant tous des noms distincts.

2.3.3. Le renommage de la mémoire

Les liens d'un *thread*

La dépendance LAE d'un chargement nécessite une recherche complexe du rangement dont il dépend. Dans l'exemple de la figure 2, l'instruction de chargement en ligne 14 doit trouver le nom attribué à `temp` par l'instruction de rangement en ligne 11 alors qu'un nombre arbitraire de rangements peuvent s'intercaler par l'appel en ligne 13.
Les *threads* sont reliés les uns aux autres pour permettre des parcours rapides. Chaque *thread* est lié à ses deux prédécesseurs dans la trace, à ses deux frères précédents, à son père et à son grand-père. Pour accélérer le renommage des sources, on multiplie les liens de façon à atteindre le *thread* producteur en passant par le minimum de *threads* intermédiaires.
Sur la figure 4, les liens sont désignés autour de chaque rectangle. Le père et le grand-père sont au-dessus du rectangle, les frères sont à gauche et les prédécesseurs de trace sont en dessous. Par exemple, le *thread* 3 a pour prédécesseurs de trace les *thread* 1 et 2. Le *thread* 8 a pour prédécesseurs de niveau les *threads* 3 et 4. Le *thread* 6 descend des *threads* 2 et 4.
Les liens sont ordonnés par l'ordre chronologique des *threads* liés : le prédécesseur de trace est le voisin le plus récent et les liens ascendants désignent les voisins les plus anciens. Par exemple, pour le *thread* 19, les liens 17 et 18 (prédécesseurs de trace) sont les plus récents. Ensuite, ce sont les liens 14 et 15 (prédécesseurs de niveau) et enfin, le lien 13 (ascendant).

Le renommage d'une source produite localement

Les registres copiés d'un *thread* créateur à un *thread* créé sur un autre cœur sont installés dans la file RF (voir la figure 1). Les références à ces registres dans le même *thread* accèdent à la valeur dans RF. Cette valeur est maintenue à jour grâce aux opérateurs (ALU et AGU) de l'étage de décodage tant que les calculs ne concernent que des constantes et des valeurs disponibles

dans RF. Une fois passé le décodage, ces références apparaissent comme des constantes. Par exemple, l'instruction 9 de la somme (`shrq rbp`), qui divise `rbp` par deux, pré-calcule `rbp/2` à partir de la copie de `rbp` transmise par l'appelant. L'entrée `rbp` de la file RF est mise à jour et une instruction d'écriture de la constante pré-calculée dans le nouvel `rbp` alloué est envoyée à la file IQ (figure 1) des instructions en attente d'exécution.
Quand un registre est référencé dans le *thread* où il est modifié (dépendance LAE locale), le renommage est classique. La table de correspondance entre les noms architecturaux et les renommages (souvent appelée RAT) est remplacée par une table extensible en hiérarchie mémoire (mémoire privée du cœur). Cela permet d'y maintenir tous les renommages de registres et de mémoires des *threads* hébergés par le cœur.

Le renommage spéculatif d'une source produite par un autre *thread*

Le renommage d'une source externe `r` s'effectue par diffusion de `r` aux *threads* prédécesseurs (sur la figure 1, les requêtes de renommages de sources sont mises dans la liste du milieu), afin de retrouver le producteur. La multiplicité des liens permet d'atteindre rapidement le *thread* producteur en parallélisant la recherche.
On marque chaque chemin parcouru pour le retour de la valeur produite au consommateur (c'est la valeur qui est retournée et non le nom ; elle est reçue dans la liste de droite de la figure 1, qui est une unité de calcul parmi celles de FU). Chaque producteur de `r` (toute instruction antérieure qui écrit dans `r`) répond au consommateur par son chemin de retour. Les réponses sont spéculatives. Si une autre réponse provient d'un voisin plus récent, la nouvelle valeur reçue remplace l'ancienne (si elle est différente) et le calcul est repris. Pour cela, l'instruction reste dans la file IQ tant que ses sources sont spéculatives. Si une autre réponse provient d'un voisin plus ancien, elle est écartée. La plupart des chemins suivis n'aboutissent à aucun producteur. L'échec est propagé le long du chemin de retour. Quand tous les voisins du consommateur ont retourné soit une valeur soit une indication d'échec, le renommage est définitif, le dernier calcul se poursuit et l'instruction est retirée.
Cette stratégie permet de trouver très vite un producteur, même d'un *thread* très éloigné du consommateur, et sans attendre les *threads* intermédiaires (qui peuvent ne même pas être extraits). Par exemple, l'instruction 14 du *thread* 12 (qui lit `temp` en pile) est liée à l'instruction 11 du *thread* 8 (qui écrit dans `temp`). Le renommage s'effectue en passant directement du *thread* 12 au *thread* 8, qui est son prédécesseur de niveau, sans attendre les *threads* 9 à 11.
Le renommage est peu coûteux, contrairement aux apparences. La plupart des renommages restent locaux grâce à la copie des registres. Pour les renommages externes, la valeur importée vient le plus souvent d'un voisin proche et les renommages de *threads* différents empruntent des chemins disjoints, ce qui permet de parallèliser. Par exemple, les *threads* du second appel récursif de `somme` effectuent leurs renommages en parallèle avec ceux du premier appel.
Le résultat d'une fonction lie la fonction appelée qui le produit au code après le retour qui le consomme. Dans l'exemple, l'instruction 11 du *thread* 23 consomme le registre `rax` produit par l'instruction 7 du *thread* 21. Un des liens existants permet de court-circuiter le *thread* 22.
Le renommage de variables globales nécessite une recherche distante. La figure 5 montre le programme principal appelant la somme. Ce programme initialise (fonction `init` au centre de la figure 5) le vecteur à sommer. Les instructions 6 et 7 des *threads* 6, 10, 17 et 21 (code de `somme` en figure 2) lisent les huit éléments de `t`. Ces instructions consomment ce que la boucle .L1 de la fonction `init` a produit (instruction 34 exécutée huit fois, par les itérations 0 à 7). La recherche des huit renommages lie chaque *thread* consommateur à l'itération productrice. Par exemple, le *thread* 17 est lié aux itérations 4 et 5. Le chemin suivi est assez indirect (5 étapes, soit 10 *threads*

```
20 main: movq $t,%rbx      //&t
21       movq $8,%rbp      //n=8
22       call init         //t[0..7]=[0..7]
23       call somme        //rax=somme(t,8)
24       movl $.LC2,%esi   //"somme: %lu\n"
25       movq %rax,%rdx    //rdx=somme(t,8)
26       movl $1,%edi
27       xorl %eax,%eax
28    //  printf("somme: %lu\n",rdx)
29       call __printf_chk
30       ret
```

```
31 init: pushq %r12
32       pushq %rbx
33       xorq  %r12,%r12    //i=0
34 .L1:  movq  %r12,(%rbx) //t[i]=i
35       addq  $1,%r12      //i++
36       addq  $8,%rbx
37       cmpq  %r12,%rbp    //i!=n
38       jne   .L1
39       popq  %rbx
40       popq  %r12
41       ret
```

```
31    // for (r12=0; r12<rbp; r12++)
32 init: loop %r12,0,%rbp,+1,x
33       addq *,%r12
34       movq %r12, (%rbx)
35       addq $8, %rbx
36       endloop
37 x:    movq %rbp, %r12
38       ret
```

FIGURE 5 – L'initialisation de t et le lancement de la somme

```
1  Code en X86        Signification      Renommage          //Commentaire
2
3  ...                                                      //c est %rcx renommé RR0 et a est %rax renommé RR1
4  addq %rax,%rcx     c=c+a              RR10=RR0+RR1       //nouveau c dans RR10
5  ...
6  movq %rcx,%rbx     sauver c dans b    RR11=RR10          //nouveau b dans RR11 qui pointe sur le c de la ligne 4
7  ...                                                      //RR11 reçoit le nom RR10 puis sa valeur
8  movq $18,%rcx      réutiliser c       RR12=18            //nouveau c dans RR12
9  ...
10 movq %rbx,%rcx     restaurer c        RR13=RR10          //nouveau c dans RR13 qui pointe sur le c de la ligne 4
11 ...                                                      //x est %r12 renommé RR2
12 addq %rcx,%r12     x=x+c              RR14=RR2+RR10      //nouvel x dans RR14 et c remplacé par le c de la ligne 4
13 ...                                                      //RR13 renomme c et pointe sur RR10
```

FIGURE 6 – Renommage d'un enchaînement de mouvements

visités au cours de l'aller-retour) mais sa longueur est la même pour toutes les lectures des éléments du vecteur et elle est proportionnelle au logarithme du nombre d'éléments.
Pour limiter le nombre de parcours, on cache les précédents renommages au sein de chaque cœur, qu'ils aient été fructueux ou pas.

2.3.4. La vectorisation des boucles

L'instruction 35 (i++) de la boucle d'initialisation de la fonction `init` (centre de la figure 5) dépend d'elle-même. Cette dépendance LAE sérialise la boucle. Pour vectoriser une telle boucle, le compilateur produit pour `init` le code à droite de la figure 5.
L'instruction `loop` identifie l'entrée dans une boucle vectorisable. Ses arguments sont la variable d'itération (ici, `r12`), sa valeur initiale (ici 0), la valeur limite (ici `rbp`), la progression (ici, +1) et l'adresse de continuation après la boucle (ici, l'étiquette `x`).
A l'exécution, l'instruction `loop` termine le *thread*. L'unité `fork` lance `rbp` + 1 *threads* (`rbp` itérations et un *thread* débutant en `x`). Les *threads* d'itérations exécutent le code d'itération. La première instruction de ce code est `addq` $*$,`r12`. L'étoile est remplacée au décodage par le numéro de l'itération reçu à la création du *thread*. L'instruction `endloop` termine le *thread* d'itération.

2.3.5. Le renommage des mouvements

La figure 6 montre des vraies dépendances par mouvements de registres et leur élimination par le renommage. L'instruction 12 ne dépend pas de l'instruction 10 mais de l'instruction 4. Nous supposons que les instructions appartiennent toutes à des *threads* différents.
Au renommage de l'instruction 4, le nom RR10 est attribué à `c`. L'instruction 6 renomme `c` par son producteur, en l'occurrence l'instruction 4, qui donne la référence allouée, c'est-à-dire RR10. L'instruction 6 conserve cette référence en attendant la valeur.
Le renommage de la source `c` de l'instruction 10 se propage jusqu'au *thread* de l'instruction 6 qui retourne la référence à RR10. L'instruction 6 transmet la demande de valeur au *thread* de

l'instruction 4, qui la retourne au *thread* de l'instruction 10 dès qu'elle est établie.
Le renommage de la source c dans l'instruction 12 s'effectue de la même façon. Quand la valeur de RR10 est établie, les instructions 6, 10 et 12 reçoivent leur copie de RR10 en même temps. Sans le renommage des instructions de mouvement, la valeur produite par l'instruction 4 serait transmise en chaîne de 4 à 6, de 6 à 10 puis de 10 à 12. La valeur de RR10 est transmise aux *threads* consommateurs où elle remplace les pointeurs RR10.
Le renommage unifié à la mémoire permet ce traitement efficace de tous les mouvements. Ceux-ci sont reconnus au renommage et sortent du chemin critique de l'exécution.

2.4. Le retrait des instructions et le traitement des exceptions

Après son extraction, chaque instruction d'un *thread* alloue une entrée dans un *reorder buffer* (ROB) du *thread* (le ROB est organisé en cache ; chaque *thread* créé ouvre un nouveau ROB). Cette entrée regroupe la destination de l'instruction et un indicateur d'exception.
Les instructions sont retirées dans l'ordre au sein d'un *thread* mais les *threads* retirent en parallèle. Une instruction est retirée dès lors que :

— elle est exécutée ;
— les renommages de ses sources sont confirmés ;
— l'instruction qui la précède dans le *thread* est retirée.

Par la seconde condition, une instruction est spéculative tant que son renommage n'est pas établi. Dès que le renommage devient définitif, l'instruction exécutée passe de l'étage de lancement à l'étage de retrait en reportant son résultat et ses exceptions dans son entrée de ROB.
Le retrait vise à libérer le cœur du *thread*. On exporte les destinations d'un *thread* t vers le *thread* précédent dès que celui-ci a fini le renommage de ses propres destinations. De là, la valeur v d'une destination d de t est propagée jusqu'au plus récent prédécesseur t' écrivant dans d. Quand t' reçoit v, la valeur prend place dans l'entrée de ROB allouée par t' pour sa dernière instruction écrivant en d. Il n'est pas nécessaire d'attendre que cette instruction soit exécutée. Lorsqu'elle l'est, si elle se termine sans exception, son résultat v' est ignoré (il est recouvert par v). En cas d'exception v est recouverte par v', l'instruction ayant produit v étant annulée.
Quand toutes les instructions d'un *thread* ont été exportées vers des voisins, le *thread* disparaît.
La valeur v d'une destination exportée d est propagée le long de la trace jusqu'à ce qu'un *thread* prédécesseur l'accueille (parce qu'une de ses instructions écrit aussi dans d) ou jusqu'à ce qu'on sache qu'il n'y a plus de *thread* en amont écrivant dans d. Alors, la valeur est reportée en mémoire globale dans son adresse de destination.
Quand une exception est levée dans un *thread*, elle annule les *threads* postérieurs. Elle est propagée aux *threads* antérieurs (dans l'ordre de la trace) jusqu'à rencontrer soit une autre exception, ce qui l'annule, soit le *thread* le plus ancien, ce qui la déclenche. Ainsi, les exceptions sont traitées en parallèle mais déclenchées dans l'ordre.

3. Evaluation du modèle

3.1. Le simulateur PerPI

Pour évaluer notre modèle d'exécution en parallèle, nous en avons développé un simulateur au sein de l'outil PerPI [1]. Le simulateur analyse chaque instruction exécutée et en détermine les cycles de traitements de chaque phase (extraction, renommage des registres, renommage d'adresse, exécution, retrait) dans deux modèles qui peuvent ainsi être comparés. A partir du nombre d'instructions exécutées et du nombre de cycles d'exécution, on calcule l'ILP (Instruction-Level Parallelism) de l'exécution. L'objectif de la simulation est de montrer que

Benchmark	Algorithme
comparisonSort	1-01 quickSort
	1-02 sampleSort
	1-03 serialSort
integerSort	1-04 blockRadixSort
	1-05 serialRadixSort
removeDuplicates	1-06 deterministicHash
	1-07 serialHash
dictionary	1-08 deterministicHash
	1-09 serialHash
breadthFirstSearch	1-10 deterministicBFS
	1-11 hybridBFS
	1-12 ndBFS
	1-13 serialBFS

Benchmark	Algorithme
maximalIndependentSet	2-14 incrementalMIS
	2-15 luby
	2-16 ndMIS
	2-17 serialMIS
maximalMatching	2-18 incrementalMatching
	2-19 ndMatching
	2-20 serialMatching
spanningTree	2-21 incrementalST
	2-22 ndST
	2-23 newST
	2-24 serialST
setCover	2-25 manis
	2-26 serialDFG
	2-27 serialGreedy

Benchmark	Algorithme
minSpanningTree	3-28 parallelKruskal
	3-29 serialMST
convexHull	3-30 quickHull
	3-31 serialHull
delaunayTriangulation	3-32 serialDelaunay
	3-33 incrementalDelaunay
nearestNeighbors	3-34 octTree2Neighbors
spmv	3-36 pSPMV
	3-37 sSPMV
triangleCounting	3-38 orderedMerge
	3-39 serialNaive
nBody	3-40 parallelCLK

TABLE 1 – *Benchmarks* de la suite PBBS

d'une part le modèle proposé donne un ILP toujours supérieur à celui du modèle d'exécution des processeurs d'aujourd'hui et que d'autre part l'ILP peut croître avec la taille de la donnée. La simulation n'évalue pas le coût en ressources mais le gain potentiel en performance.

3.1.1. Le modèle séquentiel

Le modèle séquentiel représente le modèle d'exécution des processeurs actuels. Il simule un processeur dont le prédicteur est parfait, le renommage des registres est illimité, le nombre d'unités de calculs est suffisant pour lancer toutes les instructions prêtes en parallèle, les unités ont toutes une latence d'un cycle, y compris la hiérarchie mémoire. Il a la performance maximale que pourrait atteindre un cœur basé sur le modèle d'exécution actuel.

3.1.2. Le modèle parallèle

Le modèle parallèle représente le modèle d'exécution que nous proposons. Il simule un processeur à beaucoup de cœurs qui parallélise ses extractions, ses renommages, ses exécutions et ses retraits. Dans la simulation, le nombre de cœurs est illimité : chaque *thread* créé démarre au cycle qui suit sa création. Le *thread* de continuation après un appel est traité en parallèle avec le *thread* d'appel. Les *threads* d'itérations vectorisables sont traités en parallèle, ainsi que le *thread* de sortie. La latence des communications est d'un cycle pour aller d'un cœur à un autre et un renommage distant prend autant de cycles qu'il y a de *threads* intermédiaires visités par le chemin aller-retour. Le modèle parallèle a la performance maximale que pourrait atteindre un processeur à beaucoup de cœurs.

3.2. La suite de benchmarks PBBS

L'expérience de simulation consiste à comparer l'ILP obtenu par les deux modèles sur la suite de *benchmarks* PBBS [4]. Les *benchmarks* de la suite PBBS sont issus de problèmes parallèlisables. Chaque *benchmark* est décliné en plusieurs implémentations, chacune correspondant à un algorithme (méthode séquentielle ou parallèle, types de données différents, entiers, flottants ...). Pour chaque implémentation, sept jeux de données de tailles croissantes ont été utilisés (produits par les outils fournis avec la suite PBBS), afin d'obtenir des traces de longueur doublée à chaque fois (de 64K à 8M d'instructions).
La table 1 donne la liste des 16 *benchmarks* de la suite PBBS et de leurs 41 variantes. Les numéros correspondent à ceux utilisés dans les figures qui suivent.

3.3. L'ILP du modèle séquentiel

Les *benchmarks* ont été compilés en -O3. Chaque exécution est menée à son terme. Les figures 7, 8 et 9 montrent les histogrammes des ILP du modèle séquentiel. L'ILP est très stable. Il décroît avec la taille de la trace mais faiblement. Il est toujours inférieur à 10, ce qui confirme les

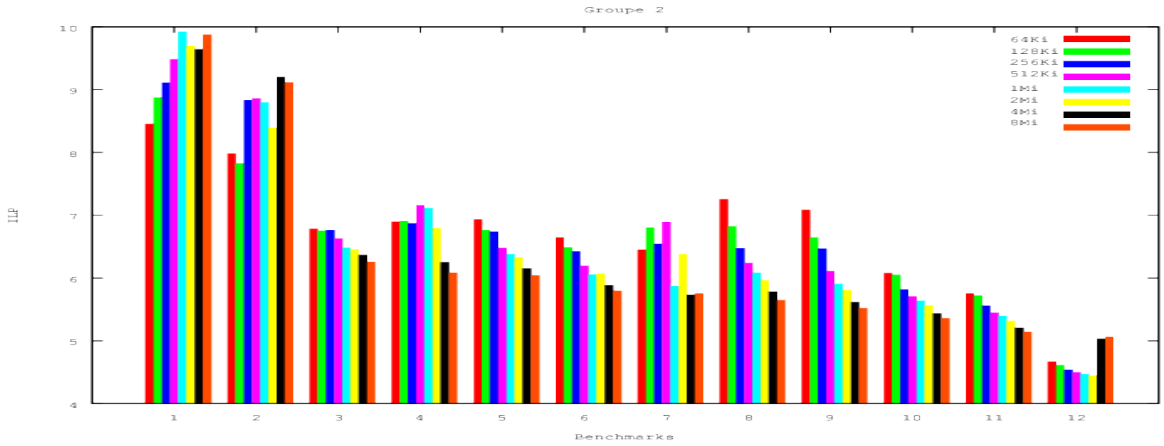

FIGURE 7 – L'ILP dans le modèle séquentiel, groupe 1 des benchmarks de la suite PBBS

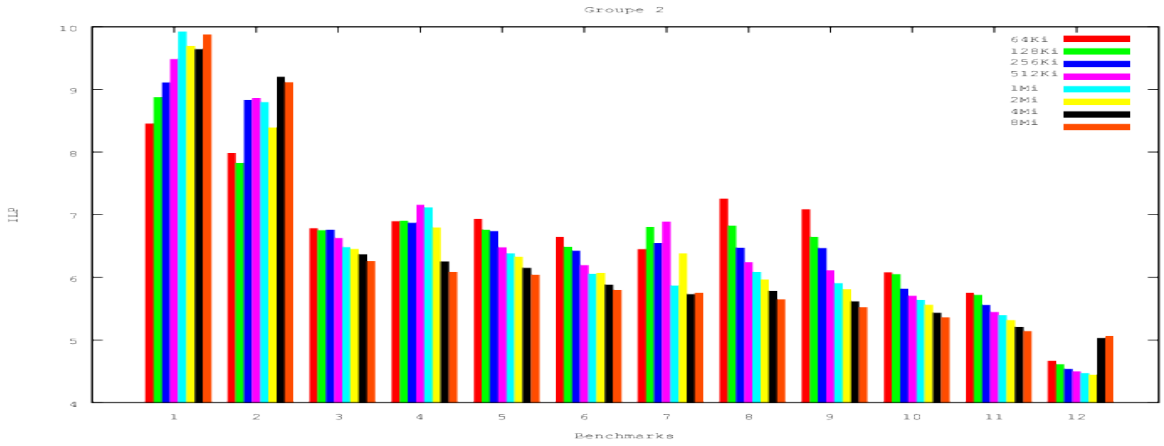

FIGURE 8 – L'ILP dans le modèle séquentiel, groupe 2 des benchmarks de la suite PBBS

mesures rapportées dans différents articles des années 90 parmi lesquels celui de [8].

3.4. L'ILP du modèle parallèle

Les *benchmarks* ont été compilés en -O1 pour éviter qu'en appliquant des optimisations sur du code séquentiel, le compilateur n'enlève encore plus de parallélisme. Malgré cette différence, le nombre d'instructions exécutées varie peu (moins de 1%). On conserve les tailles de traces du modèle séquentiel (64K à 8M d'instructions).
Les figures 10, 11 et 12 montrent que l'ILP du modèle parallèle est toujours très largement au-dessus de celui du modèle séquentiel. De plus, cet ILP croît avec la taille de la donnée.

4. Conclusion

Bibliographie

1. Goossens (B.), Langlois (P.), Parello (D.) et Petit (E.). – Perpi : A tool to measure instruction level parallelism. *In : PARA'2010 (1)*, pp. 270–281.

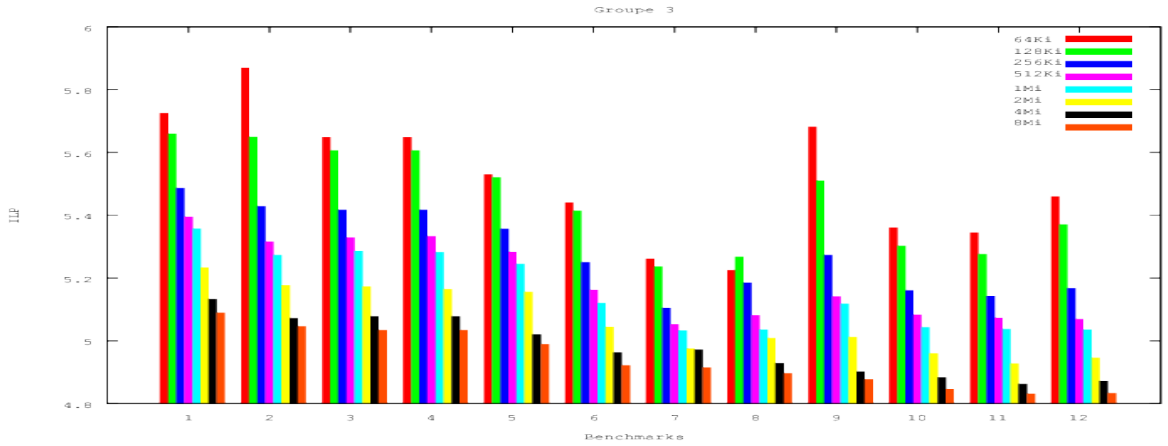

FIGURE 9 – L'ILP dans le modèle séquentiel, groupe 3 des benchmarks de la suite PBBS

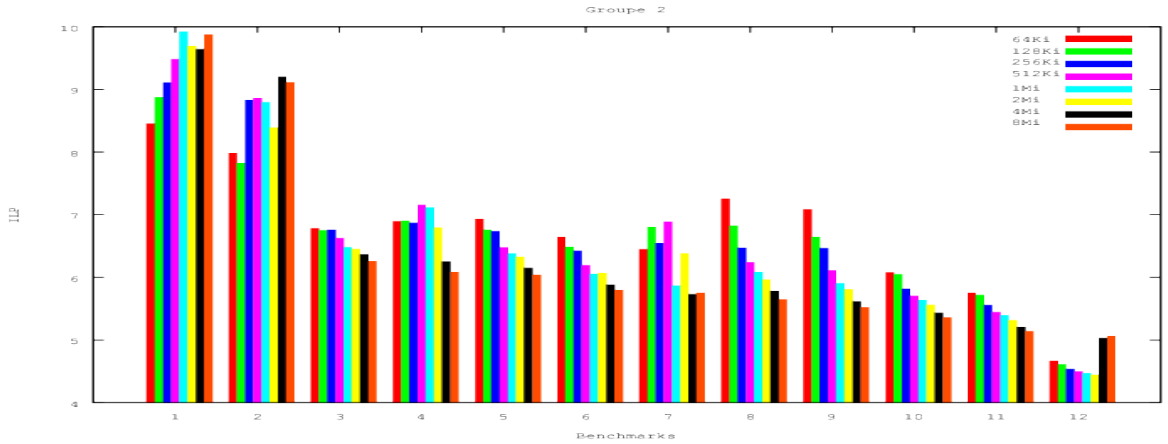

FIGURE 10 – L'ILP dans le modèle séquentiel, groupe 1 des benchmarks de la suite PBBS

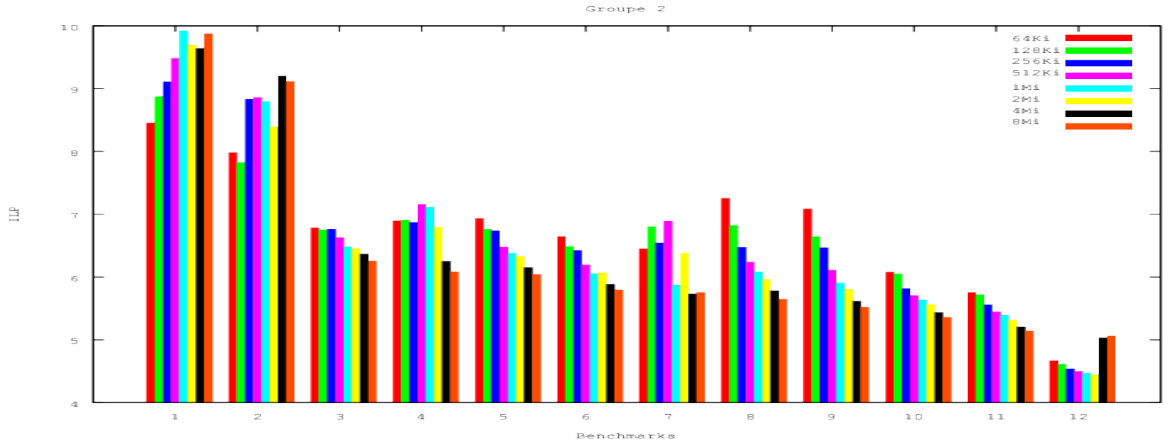

FIGURE 11 – L'ILP dans le modèle séquentiel, groupe 2 des benchmarks de la suite PBBS

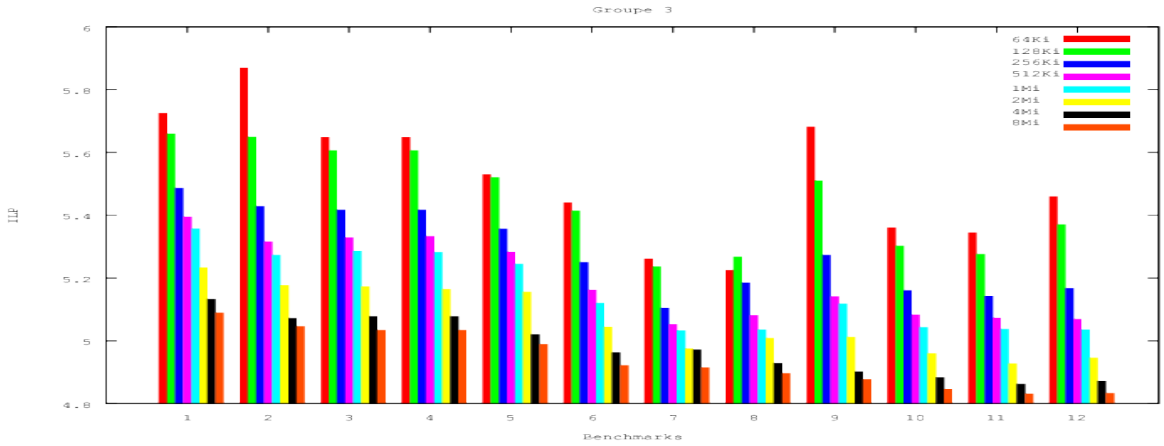

FIGURE 12 – L'ILP dans le modèle séquentiel, groupe 3 des benchmarks de la suite PBBS

2. Hennessy (J. L.) et Patterson (D. A.). – *Architecture des ordinateurs, Deuxième édition : Approche quantitative*. – Thomson Publishing France, 1996, 112–113p.
3. Porada (K.), Parello (D.) et Goossens (B.). – Analyse et réduction du chemin critique dans l'exécution d'une application. *In : Soumis à Compas'14*.
4. Shun (J.), Blelloch (G. E.), Fineman (J. T.), Gibbons (P. B.), Kyrola (A.), Simhadri (H. V.) et Tangwongsan (K.). – Brief announcement : The problem based benchmark suite. *In : Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 68–70.
5. Tjaden (G. S.) et Flynn (M. J.). – Detection and parallel execution of independent instructions. *IEEE Trans. Comput.*, vol. 19, n10, octobre 1970, pp. 889–895.
6. Tomasulo (R. M.). – An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, vol. 11, n1, janvier 1967, pp. 25–33.
7. Uht (A.) et Sindagi (V.). – Disjoint eager execution : An optimal form of speculative execution. *In : Proceedings of the 28th annual ACM/IEEE international symposium on Microarchitecture*, pp. 313–325.
8. Wall (D. W.). – Limits of instruction-level parallelism. *In : Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188.