De quoi est fait une trace d'exécution ?

Bernard Goossens, Ali El Moussaoui, Ke Chen, David Parello

RESEARCH REPORT

De quoi est fait une trace d'exécution ?

Bernard Goossens, Ali El Moussaoui, Chen Ke, David Parello

Univ. Perpignan Via Domitia,
Digits, Architectures et Logiciels Informatiques,
F-66860,Perpignan , France

Univ.Montpellier II,
Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier,
UMR 5506,
F-34095, Montpellier, France

CNRS,
Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier,
UMR 5506,
F-34095, Montpellier, France

RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

De quoi est fait une trace d'exécution ?

Bernard Goossens, Ali El Moussaoui, Chen Ke, David Parello

DALI, Université de Perpignan Via Domitia 66860 Perpignan Cedex 9 France,
LIRMM, CNRS : UMR 5506 - Université Montpellier 2 34095 Montpellier Cedex 5 France,
goossens@univ-perp.fr, ali.elmoussaoui@univ-perp.fr, ke.chen@univ-perp.fr,
david.parello@univ-perp.fr

Résumé

Cet article présente la structure des traces d'exécutions des programmes. Cette étude prolonge les travaux menés jusqu'ici par de nombreux chercheurs dans le but de quantifier le parallélisme d'instructions (ILP). Elle a pour but de comprendre la structure générale d'une exécution et le parallélisme qu'elle offre. Cette structure se compose essentiellement de deux parties opposées : l'une est d'ILP élevé, qui augmente sans limite avec la longueur de la trace considérée et l'autre est d'ILP très faible (voisin de 1), qui s'allonge sans limite avec la trace. De l'une et l'autre résulte l'ILP de la trace. Quand la première partie domine, l'ILP est fort. Dans le cas contraire, l'ILP est faible. La première partie vient des contrôles de boucles alors que la seconde vient des transmissions de paramètres/résultats entre fonctions appelantes et appelées.

1. Introduction

La mode actuelle n'est pas au parallélisme d'instructions. Les constructeurs de processeurs sont confrontés à un manque certain de résultats en terme de performance quand ils tentent de consacrer une partie des transistors disponibles à l'extension des capacités de traitement du cœur de calcul. Depuis près d'une dizaine d'années, on ne compte plus ni sur le découpage du pipeline en tranches plus fines, ni sur l'élargissement du chemin superscalaire des instructions pour améliorer la performance du processeur. On table plutôt sur la duplication des cœurs, à charge pour la chaîne logicielle d'y répartir les calculs.

Néanmoins, le parallélisme d'instructions ne doit pas être perdu de vue pour deux raisons :

– c'est un parallélisme naturellement présent dans le code, que le compilateur peut augmenter et que le matériel peut extraire automatiquement, par opposition au parallélisme de tâche qu'il faut exprimer manuellement par la séparation, la synchronisation et la communication de threads, et par opposition au parallélisme de données, présent parfois mais pas toujours, et qui nécessite que le compilateur organise l'espace des données pour le rendre exploitable avec le maximum de simultanéité et le minimum de communication,
– la loi d'Amdahl est là pour nous mettre en garde : accélérer les parties parallèles ne sert plus à rien quand les parties séquentielles dominent le temps d'exécution.

Cet article étudie le parallélisme des traces d'exécution. Il est organisé comme suit : la section 2 présente les travaux relatifs au parallélisme d'instructions, la section 3 analyse le parallélisme d'instructions des boucles et des fonctions, la section 4 fait une synthèse des analyses précédentes et montre la structure du graphe de dépendances des instructions d'une trace. Les conclusions et les futurs travaux sont décrits dans la section 5.

2. Travaux relatifs

Des études sont régulièrement menées sur l'ILP (Instruction Level Parallelism) des programmes. Elles sont systématiquement contradictoires, tantôt affirmant qu'il n'y a pas assez d'ILP exploitable par un état donné de la technologie, tantôt affirmant le contraire. A chaque fois, les mesures antérieures les plus pessimistes sont remises en cause en diminuant les contraintes de séquentialité considérées dans le calcul d'ILP.

En 1970, Tjaden et Flynn [8] mesurent la quantité de parallélisme disponible dans une fenêtre de 2 à 10 instructions. Ils constatent qu'avec un matériel adapté, on pourrait en moyenne exécuter 1.86 instructions par cycle.

En 1984, Nicolau et Fisher [7] mesurent le parallélisme disponible pour les architectures VLIW (Very Large Instruction Word). Au passage, une mesure d'ILP est menée qui conclut pour la première fois à la présence d'ILP élevé (1000 pour certains programmes scientifiques). Mais leur étude principale porte sur la présence de parallélisme au sein d'un bloc de base, c'est-à-dire le parallélisme que le compilateur peut détecter pour former des vecteurs d'instructions devant alimenter un processeur VLIW.

En 1989, l'article de Jouppi et Wall [5], qui étudie les différences entre exécution en parallèle (processeur superscalaire) et en série (processeur superpipeline) conclut que les deux sont équivalents et que surtout, le parallélisme d'instructions disponible (c'est-à-dire exploitable) est très limité (6 instructions exécutables par cycle au mieux).

Au début des années 90, David Wall mène la première véritable étude sur l'ILP [9]. Il est assez sceptique au départ, étant donné les résultats de son travail publié en 1989 avec Jouppi. Sa note technique de 1990 confirme que le taux de parallélisme exploitable est de l'ordre de 5 instructions (en gros, à cette époque, ce qui est dans un bloc de base, étendu aux blocs voisins : les instructions que le compilateur peut réarranger autour d'un saut pour étendre le parallélisme). Son étude s'étend à une machine parfaite (limitée toutefois à 64 lancements par cycle) pour laquelle un taux élevé de parallélisme est détecté sur certains codes, ce qui confirme les mesures de Nicolau et Fisher. En 1993, Wall reprend ses mesures en les complétant [10]. En particulier, il enlève la limite de 64 instructions sur sa machine parfaite, ce qui fait apparaître que, pour certains codes, l'ILP est bien supérieur (au-delà de 600).

En 1992, Lam et Wilson [6] étudient l'impact du contrôle sur le parallélisme. Ils montrent que l'ILP mesuré pour une machine dotée d'un prédicteur parfait va bien au-delà de ce qui est connu alors (ce qui incitera Wall à reprendre son expérience sur une machine parfaite). C'est la première fois qu'on fait ressortir que de l'ILP distant existe (des instructions indépendantes arbitrairement loin du point d'extraction). Ils concluent que pour capturer cet ILP dans un processeur, il faut qu'il soit doté de capacité d'exécution spéculative (exécuter des blocs lointains, avant que les sauts qui en contrôlent l'accès ne soient calculés).

Vers la fin des années 90, l'absence de progrès significatif dans l'exploitation de l'ILP a poussé les chercheurs à explorer de nouvelles voies pour franchir l'obstacle des dépendances de données (le lien producteur/consommateur via un registre ou un mot mémoire partagé). En 1998, Gonzalez et Gonzalez [3] ont étudié l'impact de la prédiction de valeur sur l'ILP (machine parfaite) et sur l'IPC (processeur superscalaire de degré 8). Sur la machine parfaite, l'impact est très important (facteur d'accélération de 50 sur les SPECINT95 et 2000 sur les SPECFP95). Mais sur la machine réaliste, le gain est limité (12%).

Simultanément, les chercheurs ont mis l'accent sur le fait que pour exécuter plus d'instructions en parallèle, il fallait aller les chercher de plus en plus loin du point d'extraction.

En 2001, Balasubramonian, Dwarkadas et Albonesi [1] étudient les performances d'un processeur exécutant deux threads en parallèle, l'un principal et l'autre secondaire, chargé de trouver des instructions indépendantes dans la future trace spéculative. Le thread secondaire n'est activé que quand le thread principal est suspendu. Le thread secondaire avance en appliquant une politique séparant les instructions dépendantes (de ce qui est encore en cours de calcul dans le thread principal) des instructions indépendantes. Les premières ne conservent aucune ressources et les dernières ne valident pas leur résultat. L'effet est principalement de précharger la mémoire et de précalculer ce qui doit être repris plus tard par le thread principal. L'amélioration mesurée d'une telle microarchitecture sur une microarchitecture standard est de 17% en moyenne pour un pic à 64%.
En 2004, Cristal, Santana et Valero [2] proposent une microarchitecture capable d'exécuter un kilo-instructions en parallèle. Le titre est assez explicite et montre que les chercheurs savent que c'est très loin du point d'extraction qu'il faut chercher le parallélisme. L'idée exposée dans l'article est de permettre à plus d'instructions d'être en exécution simultanément en optimisant l'allocation et la libération des ressources de stockage qu'elles utilisent.
Depuis cette époque, l'intérêt des microarchitectes s'est petit à petit déplacé vers la gestion d'un ensemble de cœurs au sein d'un processeur multi-cœur.

3. Analyse de la structure du parallélisme d'instructions

Plutôt que de compter le parallélisme comme l'ont fait tous les chercheurs que nous avons cités, nous avons essayé de l'analyser. Les études sur l'ILP que nous avons mentionnées dans l'introduction ont essentiellement quantifié le nombre d'instructions exécutables à chaque cycle. En revanche, aucune étude n'a analysé le graphe de dépendances de ces instructions.
Pour bien comprendre la suite, il faut se placer dans le contexte d'une machine idéale, c'est-à-dire la meilleure machine possible pour capturer tout le parallélisme d'instructions disponible. Une machine idéale dispose d'emblée de la totalité de la trace des instructions à exécuter. Elle dispose aussi de ressources en quantité suffisante pour exécuter à chaque cycle tout ce qui est prêt (indépendant de toute donnée non calculée). Cela signifie qu'à chaque cycle, toutes les instructions de la trace qui sont prêtes sont exécutées. La machine idéale est dotée d'opérateurs parfaits, opérant en un cycle (y compris les accès à la mémoire).
En particulier, cela implique que la machine idéale a les ressources pour dédoubler un registre ou un emplacement mémoire dès lors qu'il est employé dans deux calculs indépendants (élimination de toutes les dépendances Ecriture Après Ecriture ou Lecture).
L'ILP est ainsi défini (pour l'exécution en c cycles (machine idéale) de n instructions en langage machine d'un programme p appliquée à un jeu de données d) :

$$\mathrm{ILP}(p, d) = \frac{n}{c}$$

3.1. Parallélisme relatif aux boucles

Prenons l'exemple d'une boucle de la forme présentée dans la figure 1 où le source C figure à droite et l'assembleur x86 (syntaxe Gnu) est à gauche.

Dans ce code, le contrôle est indépendant du corps. Le contrôle est exécutable dès le lancement du programme alors que le corps dépend du calcul de x. Indépendamment de la

```
1                                ;(ebx contient x)          for (i=0;i<1024;i++)
2       xorl  %eax,  %eax        ;eax = 0                     t[i]=x;
3  loop: movl  %ebx,  t(%eax)    ;t[i] = x
4       addl  $1,    %eax        ;i++
5       cmpl  $1024, %eax        ;(i ==? 1024)
6       jne   loop               ;si (i != 1024) vers loop
```

FIGURE 1 – Une boucle à bornes calculées statiquement

position de cette boucle dans la trace d'exécution, sur une machine parfaite, les instructions du contrôle de boucle s'exécutent à partir du premier cycle alors que celles du corps ne s'exécutent qu'après le calcul de x (en supposant que x soit établi au cycle $c > 1024$, ce sont les 1024 instructions d'écriture en mémoire qu'on peut exécuter simultanément au cycle $c + 1$). Le contrôle de cette boucle ajoute peu au parallélisme du début de la trace, 3 instructions tout au plus, comme le montre le déroulement qui suit (les instructions d'une même itération forment une diagonale, par exemple `eax=1; cmp 1024,1; bnz loop`) voir figure 2 :

```
1  cycle[1]   eax=0
2  cycle[2]   eax=1
3  cycle[3]   eax=2   cmp 1024,1
4  cycle[4]   eax=3   cmp 1024,2  bnz loop
5  cycle[5]   eax=4   cmp 1024,3  bnz loop
6  ...
```

FIGURE 2 – Exécution pipelinée du contrôle de boucle

```
1  main: ...                ; calcul de x=f(...)      #define N 1024
2        movb x, %al        ; al=x                    char t[N][N];
3        movl $0, %esi      ; i=0                     char x;
4  .L5:  ; debut de la boucle externe                  main(){
5        movl $0, %edi      ; j=0                      int i,j;
6  .L4:  ; debut de la boucle interne                   x=f(...);
7        movl %esi, %ecx    ; ecx=i                    for (i=0;i<N;i++)
8        sall $10, %ecx     ; ecx=ecx*1024               for (j=0;j<N;j++)
9        addl %edi, %ecx    ; ecx=ecx++                    t[i][j]=x;
10       movb %al, t(%ecx); t[i][j]=x                  }
11       addl $1, %edi      ; j++
12       cmpl $1023, %edi   ; (j <=? 1023)
13       jle  .L4           ; si (j<=1023) vers .L4
14       ; fin de la boucle interne
15       addl $1, %esi      ; i++
16       cmpl $1023, %esi   ; (i <=? 1023)
17       jle  .L5           ; si (i<=1023) vers .L5
18       ; fin de la boucle externe
19       ret
```

FIGURE 3 – Boucles imbriquées à bornes calculées statiquement

Cependant, toutes les boucles de ce type (boucle "for" décrivant un intervalle dont les deux extrémités sont des constantes) ajoutent leur contrôle au parallélisme du début de trace. Quand elles sont nombreuses, le parallélisme est très élevé.

Dans l'exemple de la figure 3 (le code C, à droite), les contrôles des 1024 exécutions de la boucle interne peuvent démarrer tous ensemble, au premier cycle, en même temps que le contrôle de la boucle externe. A condition de disposer des ressources nécessaires, on pourrait exécuter au premier cycle 1025 initialisations de variables ($i = 0$ et 1024 fois $j = 0$). Au second cycle, ce sont autant d'incrémentations qui sont prêtes et qu'on peut exécuter en parallèle. Dès le troisième cycle, on dispose d'un ILP de 2048 (1024 incrémentations et 1024 comparaisons), puis 3072 le cycle suivant.
Le corps de la boucle interne dépend de x, qui est établi après le retour de la fonction `f`. En supposant que celle-ci s'exécute en plus de 1024 cycles, le corps est calculé après le contrôle. La traduction en assembleur x86 est à gauche de la figure 3.
Un outil de calcul automatique d'ILP tel que PerPI [4] montrerait que cette double boucle exécute 7344130 instructions en 1028 cycles, soit un ILP de 7144 (en supposant cette fois que la valeur de x est disponible dès le début de l'exécution). S'il est aisé de calculer "à la main" le nombre d'instructions exécutées (7 instructions dans le corps de boucle interne, répétées 2^{20} fois, 4 instructions répétées 2^{10} fois et deux instructions initiales, soit $7 * 2^{20} + 2^{12} + 2 = 7344130$), il est autrement plus complexe de calculer le nombre de cycles et l'ILP.
La figure 6 à gauche montre l'histogramme de l'ILP tout au long des 1028 cycles. On voit sur cet histogramme que dès les premiers cycles, plus de 5000 instructions sont exécutables. Un zoom ferait apparaître qu'on passe de 1026 instructions au premier cycle (dont 1024 fois `movl $0, %edi`) à 2049 au second (dont 1024 fois `movl %esi, %ecx` et `addl $1, %edi`), 4098 au troisième (1024 fois les précédentes et `sall $10, %ecx`, `cmpl $1023, %edi`) et enfin 5126 au quatrième (les précédentes plus `jle .L4`). Ensuite, le nombre d'instructions exécutées augmente de 4 à chaque cycle, pour culminer à 9271 instructions au cycle 1025.
L'instruction `addl %edi, %ecx` est typique de ce qui fait que l'analyse à la main de l'ILP est très piégeux. Dans un premier temps, c'est la source `%ecx` qui est la plus tardive, insérant l'instruction "addl" dans la chaîne locale des calculs de l'adresse du rangement qui suit (instruction `movb %al, t(%ecx)`). Par la suite, c'est `%edi` qui devient plus tardif que `%ecx`, insérant le "addl" dans la chaîne globale de la progression de la boucle interne, par le biais de l'incrémentation de `%edi`. Cela décale d'un cycle (de retard) la position des "addl" dans la progression temporelle, décalant du même coup d'autant le `movb %al, t(%ecx)`.
Toutes les boucles ne sont pas aussi favorables à l'ILP. Si la limite supérieure de l'intervalle est une variable calculée n, la comparaison de fin de boucle ($i < n$) et le saut qui en dépend ne peuvent être exécutées qu'après le calcul de n. En revanche, l'initialisation et les incrémentations restent indépendantes, donc exécutables dès le premier cycle (voir figure 4, à gauche). C'est encore moins favorable si la boucle est décroissante (voir figure 4, à droite). Dans ce cas, même l'initialisation et la décrémentation sont dépendantes du calcul de n. Tout est retardé jusqu'à ce que n soit établi (pour le contrôle ; pour le corps, il faut attendre x).

```
1  for (i=0;i<n;i++)          for (i=n-1;i>=0;i--)
2    t[i]=x;                    t[i]=x;
```

FIGURE 4 – Boucles à bornes calculées dynamiquement

Pour ce qui est d'une boucle "while" (une boucle dont le nombre d'itérations n'est pas connu) voici un exemple de calcul de racine carrée entière (arrondi par défaut) (voir figure 5, partie

droite pour le source C et partie gauche pour la traduction en x86).

```
1  x:     .long  1024          ; x=1024                 unsigned int x=1024;
2  main:  movl   $-1, %eax     ; i=(-1)                 main(){
3         movl   $0, %ecx      ; c=0                     unsigned int c=0,r;
4  .L2:   addl   $2, %eax      ; i+=2                    int i=(-1);
5         addl   %eax, %ecx    ; c+=i                    do {
6         cmpl   x, %ecx       ; (c<=?x)                  i+=2; c+=i;
7         jbe    .L2           ; si (c<=x) vers .L2      } while (c<=x);
8         sarl   %eax          ; r=i/2                   r=(i/2);
9         ret                                           }
```

FIGURE 5 – Calcul de racine carrée : boucle à nombre d'itérations variable

L'outil PerPI montrerait que cette boucle composée de 33 itérations, donc 135 instructions (4*33 + 3) s'exécute en 37 cycles sur une machine idéale, avec un ILP de 3.65. La figure 6 à droite montre l'histogramme de l'ILP tout au long des 37 cycles d'exécution. On voit sur cet histogramme qu'on n'exécute jamais plus de 4 instructions à la fois. Les 4 instructions du corps forment une chaîne calculant à la fois le contrôle et l'étape suivante du résultat. Quand le contrôle et le calcul sont liés, le parallélisme est faible.

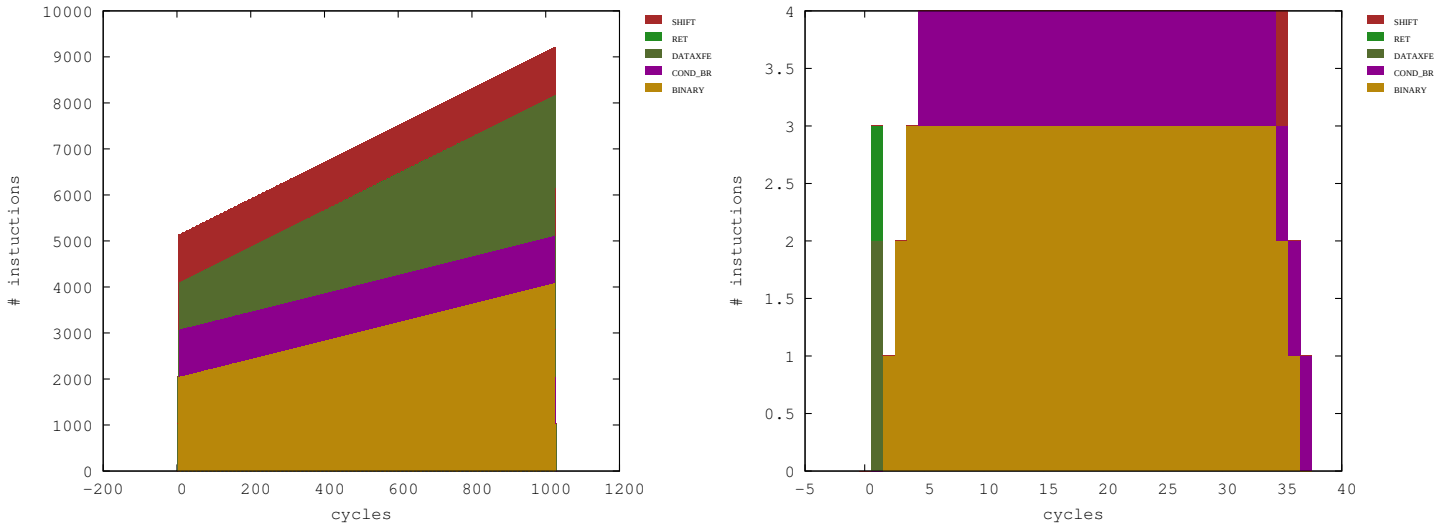

FIGURE 6 – Histogramme des traces de la double boucle (à gauche) et "racine" (à droite)

L'ILP d'une boucle interne tend vers le nombre d'instructions de son corps quand le nombre d'itérations augmente. Par exemple, la boucle précédente a un ILP de 3.65 pour 4 instructions itérées. Cet ILP passe à 3.74 pour 46 itérations ($x = 2048$) et 3.95 pour 257 itérations ($x = 65536$). Dans l'exemple, cela tient à la dépendance de l'instruction `addl %eax,%ebx` avec elle-même. A cause d'elle, on ne peut lancer plus d'une itération par cycle, ce qui fait que le nombre de cycle est au moins égal au nombre d'itérations. Cela concerne donc toutes les boucles à variable contrôlée, mais plus généralement, toutes les boucles ayant une instruction récurrente (i++ ou c+=i).

3.2. Parallélisme local et parallélisme global

On peut facilement augmenter l'ILP local d'une boucle en la déroulant statiquement pour augmenter le nombre d'instructions dans le corps.

On peut aussi fusionner des corps de boucles dont le contrôle est identique. Par exemple, on peut fusionner les deux boucles à gauche de la figure 7 en une seule, à droite).

```
1  for (i=0;i<1024;i++)          for (i=0;i<1024;i++)
2    {corps1}                      {corps1; corps2}
3  for (i=0;i<1024;i++)
4    {corps2}
```

FIGURE 7 – Fusion de boucles

Dans ce cas, l'ILP local (de la boucle) est amélioré mais pas l'ILP global (du programme). En supposant qu'une itération de la boucle corps1 se compose de $n1$ instructions et qu'une itération de corps2 en contienne $n2$, l'ILP de la boucle corps1 est $n1$ (l'ILP tend vers le nombre d'instructions du corps), celui de la boucle corps2 est $n2$ et celui de la boucle unique est $n1 + n2$. Néanmoins, l'ILP des deux boucles en séquence est aussi $n1 + n2$ puisqu'on exécute $(n1 + n2) * 1024$ instructions en 1024 cycles, les contrôles des deux boucles étant exécutés en parallèle.

Un peu plus haut, nous avons montré que l'ILP local de la boucle présentée figure 1 tend vers 4 ($(4 * 1024)/1024$). L'ILP global de la boucle, en supposant que x soit établi au cycle 10000 (ILP de la boucle insérée dans la globalité du programme) est de $(4 * 1024)/10001 = 0.41$.

D'une façon générale, le compilateur le plus souvent, lorsqu'il procède à des améliorations par réorganisation du code qu'il produit, ne fait que déplacer du parallélisme local sans changer le parallélisme global. Pour accroître ce dernier, il faut diminuer le nombre de cycles, ce qui nécessite d'agir sur les liens unissant toutes les instructions figurant dans les plus longues chaînes de dépendances. Ainsi, dérouler une boucle qui ne fait partie d'aucune de ces chaînes n'améliore que l'ILP local de la boucle sans réduire en rien l'ILP global.

Localement, l'exécution d'une boucle sur une machine idéale concerne d'un côté son contrôle et de l'autre son corps. L'un et l'autre peuvent être liés (cas des boucles "while"). En ce qui concerne le contrôle, son calcul peut démarrer dès le premier cycle (borne initiale constante). Il peut aussi démarrer après le calcul de la borne initiale (par exemple, $n-1$ dans une boucle décroissante). Dans les deux cas, il se poursuit pendant autant de cycles qu'il y a d'itérations. Il se termine par deux cycles supplémentaires : comparaison et saut conditionnel non pris.

Globalement, les boucles se placent dans le graphe d'ordre partiel des instructions de la trace, toutes avec leur contrôle en tête (du cycle 1 au cycle n pour une boucle à n tours). Certains corps sont mélangés au contrôle dans ce préfixe des n premiers cycles et d'autres en sont détachés (quand ils dépendent d'un calcul tardif).

Pour résumer notre étude sur les boucles, ce qui pèse sur l'ILP en allongeant les chaînes de dépendances, c'est leur contrôle, à cause de la récurrence du calcul de la variable contrôlée.

3.3. Parallélisme relatif aux appels de fonctions

Les fonctions sont contrôlées par des opérations de transferts des paramètres (liens de l'appelant vers l'appelé) et des résultats (liens de l'appelé vers l'appelant).

Dans une machine idéale, tout emplacement en mémoire peut se dédoubler quand il est employé dans plusieurs calculs indépendants. C'est en particulier le cas de la pile.
Dans la fonction "hanoi" de la figure 8, partie droite, la variable locale `i` est placée en pile, ainsi que des copies de sauvegarde du contexte (variables `n`, `d` et `a`). Les deux appels à "hanoi" utilisent le même espace mémoire pour leur cadre (celui alloué pour le premier appel est libéré à son retour et réalloué par le second appel). Cela crée des dépendances de noms que la machine idéale élimine par renommage, permettant des accès parallèles (les deux appels peuvent démarrer simultanément en travaillant chacun sur une copie de la portion mémoire unique contenant leur cadre).
La partie gauche de la figure 8 montre la traduction "gcc" de la fonction "hanoi" en assembleur x86 (sans optimisation ; on peut faire nettement mieux, même à la main) :

```
1  hanoi: pushq %rbp            ;en pile: ]ret,...,rbp]        void hanoi(
2         movq  %rsp, %rbp      ;nouveau ]rsp, ..., rbp]        int n,
3         subq  $32, %rsp       ;reserver 32 octets de cadre    int d,
4         movl  %edi, -20(%rbp) ;sauver n                       int a){
5         movl  %esi, -24(%rbp) ;sauver d                       int i = 6-d-a;
6         movl  %edx, -28(%rbp) ;sauver a                       if (n==0) return;
7         movl  $6, %eax        ;                               hanoi(n-1,d,i);
8         subl  -24(%rbp), %eax ;                               deplacer(n,d,a);
9         subl  -28(%rbp), %eax ;                               hanoi(n-1,i,a);
10        movl  %eax, -4(%rbp)  ;sauver 6-d-a (i)             }
11        cmpl  $0, -20(%rbp)   ;n ==? 0
12        je    .L8             ;si (n==0) vers L8 (retour)
13        movl  -20(%rbp), %eax ;eax = n
14        leal  -1(%rax), %ecx  ;ecx = n-1
15        movl  -4(%rbp), %edx  ;edx = i
16        movl  -24(%rbp), %eax ;eax = d
17        movl  %eax, %esi      ;esi = d
18        movl  %ecx, %edi      ;edi = n-1
19        call  hanoi           ;hanoi(n-1,d,i)
20        movl  -28(%rbp), %edx ;restaurer a
21        movl  -24(%rbp), %ecx ;restaurer d
22        movl  -20(%rbp), %eax ;restaurer n
23        movl  %ecx, %esi      ;esi = d
24        movl  %eax, %edi      ;edi = n
25        call  deplacer        ;deplacer(n,d,a)
26        movl  -20(%rbp), %eax ;eax = n
27        leal  -1(%rax), %ecx  ;ecx = n-1
28        movl  -28(%rbp), %edx ;edx = a
29        movl  -4(%rbp), %eax  ;eax = i
30        movl  %eax, %esi      ;esi = i
31        movl  %ecx, %edi      ;edi = n-1
32        call  hanoi           ;hanoi(n-1,i,a)
33 .L8    leave                 ;ancien ]rsp, ..., rbp]
34        ret                   ;depiler l'adresse de retour
```

FIGURE 8 – Tours de Hanoi : fonction récursive

Les instructions d'appels ("call") écrivent l'adresse de retour en mémoire. Par effet de bord, elles modifient le registre sommet de pile `rsp`. En apparence, cela crée une dépendance de donnée : un appel dépend de la dernière instruction ayant mofidié `rsp`. Dans une machine idéale, on s'affranchit de cette fausse dépendance en considérant que les manipulations du registre `rsp` sont transparentes, comme si l'adresse de tout élément de la pile était une

constante calculable au début de l'exécution. En quelque sorte, lorsque l'exécution démarre, la machine idéale dispose non seulement de la trace, mais aussi de la mémoire, avec autant de copies d'emplacements que d'utilisations de ces emplacements. Dans l'exemple de "hanoi" appliqué à "hanoi(3,1,3)", le cadre le plus profond sur la pile est réutilisé 16 fois. A chacun des 32 octets qui le compose sont associées 16 copies deux à deux distinctes, donc toutes utilisables en parallèles.
Ainsi, toutes les instructions d'appels peuvent être parallélisées. Sur la figure 9 à gauche on voit l'histogramme de l'exécution de "hanoi(3,1,3)" en 23 cycles. Lors de cette exécution, il y a 22 appels (8 appels de "hanoi(0,...)", 4 appels de "hanoi(1,...)", 2 appels de "hanoi(2,...)", 1 appel de "hanoi(3,...)" et 7 appels de "deplacer(...)"), tous exécutables dès le cycle 1. Tous les appels vont pouvoir s'exécuter en parallèle, avec un ordre des instructions déterminé par les seules dépendances producteur/consommateur. On voit sur l'histogramme les 22 instructions "call" concentrées au cycle 1.
De la même façon, la sauvegarde du cadre appelant, par un empilement du registre `rbp` et un déplacement du registre `rsp`, peut se faire dès le premier cycle, pour tous les cadres en même temps (il s'agit d'initialiser chaque copie des registres `rbp` et `rsp` avec les bornes constantes de la copie de cadre attribué à chaque appel). On voit sur l'histogramme les instructions "push" au cycle 1 (une par appel à "hanoi", soit 15 au total).
Les libérations de cadres et les retours de toutes les fonctions sont simultanés, au cycle 2. Les instructions "ret" apparaissent en cycle 2 sur l'histogramme (22 instructions).
Les sauvegardes se font dès que la variable à sauvegarder est connue. Par exemple, dans un appel à "hanoi", on sauvegarde n, d et a en pile quand l'appelant en a fixé la valeur. Pour n, la valeur initiale est rangée au cycle 1. Ensuite, à chaque cycle, n est décrémenté (deux copies de $n-1$ sont produites en parallèle pour chacun des deux appels à "hanoi(n-1,...)").
La fonction "hanoi" exécute ses 365 instructions en 23 cycles, soit un ILP de 15,87 (ce qui inclut une fonction "main" appelant "hanoi(3,1,3)" et une fonction "deplacer" ne contenant qu'une instruction de retour).
Les sauts conditionnels (instruction "je .L8") s'échelonnent dans le temps, selon la profondeur d'appel à laquelle ils appartiennent (ils dépendent de n). Il y a 15 sauts exécutés, dont 8 pris (dans 8 appels à "hanoi(0,...)") et 7 non pris (4 dans les 4 appels à "hanoi(1,...)", 2 dans les 2 appels à "hanoi(2,...)" et 1 dans l'appel à "hanoi(3,...)"). Les 8 sauts pris sont exécutés au cycle 16. Les 4 sauts le sont au cycle 12, les 2 sauts au cycle 8 et le saut de "hanoi(3,...)" est exécuté au cycle 4. Cela illustre le fait que les appels sont exécutés en parallèle avec une séquentialisation des corps imposée par la transmission des paramètres (lecture dans le cadre appelant, calcul du paramètre effectif, sauvegarde et utilisation dans le cadre appelé).

4. La structure d'une trace d'exécution

Pour les boucles, les chaînes séquentielles limitant l'ILP sont une succession d'incrémentation de la variable contrôlée. Pour les fonctions, d'autres chaînes existent venant de la transmission de paramètres/résultats, qui propagent des données d'un cadre à l'autre.
La figure 9 à droite montre l'histogramme de "stringsearch" et la figure 10 à gauche est l'histogramme de "cjpeg", deux des applications de la suite Mibench. Les deux histogrammes paraissent peu remplis. Cela tient à l'échelle imposée par la grande quantité d'instructions exécutables aux deux premiers cycles. Pour "stringsearch" (161119 instructions exécutées en 946 cycles ; ILP = 170), il y a 11638 instructions exécutables au cycle 1 (dont 937 instructions

"call") et 3540 au cycle 2 (dont 937 instructions "ret"). Pour "cjpeg", seules les 1M instructions du début sont représentées, exécutées en 4520 cycles (ILP = 221). Au premier cycle, il y a 14214 instructions exécutées. Au second cycle, il y en a 4835. On voit dans ces deux histogrammes une structure générale commune à tous les graphes de dépendances, avec les deux premiers cycles ultra dominants, puis une masse, d'un millier de cycles pour "cjpeg", de 200 cycles pour "stringsearch", correspondant aux contrôles des boucles, puis une longue queue (700 cycles pour "stringsearch" et 3500 cycles pour "cjpeg") avec un ILP oscillant entre 1 et 2 (transmission de paramètres ou de résultat).

Nos extraits de trace, bien que courts, sont représentatifs de l'application dont ils proviennent. Allonger la trace revient à distribuer les nouvelles instructions dans le graphe de dépendances : les initialisations et appels viennent grossir le premier cycle, les retours s'ajoutent au second cycle, les nouvelles boucles à initialisation constante se placent dans le groupe des premiers cycles et les autres en milieu de graphe et enfin, la suite des nouveaux appels de fonction se concatène en fin de graphe. La figure 10 à droite montre l'histogramme de 7,2M d'instructions de l'application "cjpeg". On voit que le profil reste le même, avec un élargissement des deux axes : le nombre d'instructions exécutées au premier cycle passe de 14214 à 97148 et le nombre total de cycles d'exécution passe de 4520 à 8774.

On déduit de ces observations d'une part que pour favoriser un ILP élevé, il faut réduire la longueur du graphe, c'est-à-dire agir sur l'enchaînement des appels/retours et la transmission de paramètres (des sous-programmes plutôt longs et peu nombreux avec peu de paramètres/résultats et des boucles avec peu d'itérations de corps longs ; les optimisations d'*inlining* et de déroulage de boucle des compilateurs sont à mettre à contribution). D'autre part, pour profiter de cet ILP, il faudrait disposer d'un processeur aux ressources quasi infinies en début d'exécution (le nombre de ressources nécessaires augmente avec la longueur du code à exécuter) et s'amenuisant au fur et à mesure qu'on avance (avec quelques élargissements ponctuels), jusqu'à un processeur purement séquentiel et *in-order* pour exécuter les instructions se trouvant en fin de graphe.

L'étude de l'ILP des boucles et des fonctions montre que le parallélisme est très élevé (voire maximum) dans les premiers cycles de l'exécution. On peut même dire que cet ILP du début augmente au fur et à mesure que la trace considérée s'allonge car on trouve des instructions indépendantes tout au long d'une exécution (les initialisations de registres).

Ce fort parallélisme de départ est très utile parce qu'il permet de disposer d'un stock d'instructions exécutables dans lequel on peut choisir, en quantité égale aux ressources disponibles, celles qui vont alimenter régulièrement l'ensemble des opérateurs, pour peu qu'on puisse aller les chercher. La politique de choix fait que les instructions écartées de la sélection se verront exécutées plus tardivement qu'elles ne pourraient l'être. Il convient de ne retarder que des instructions qui (1) ne sont pas des maillons de la chaîne de dépendances la plus longue et qui (2) ne retardent pas une instruction de cette chaîne. De la sorte, ces instructions sont retardées pour répartir les calculs dans le temps, sans pour autant allonger la chaîne la plus longue, donc sans augmenter le nombre de cycles optimal de la machine idéale. Nous avons établi que les incrémentations de variable contrôlée pour les boucles et les transmissions de paramètres/résultats pour les fonctions étaient à l'origine des longues chaînes. Ce sont donc ces instructions qu'il faut favoriser lors de la sélection.

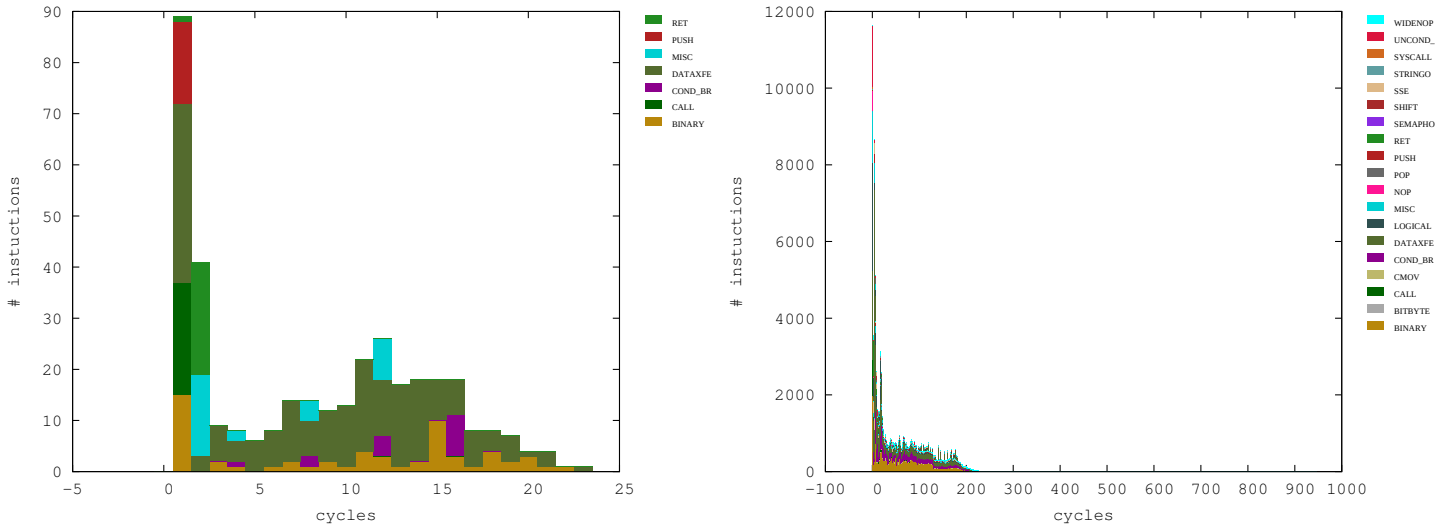

FIGURE 9 – Histogramme des traces de "hanoi" à gauche et "stringsearch" à droite

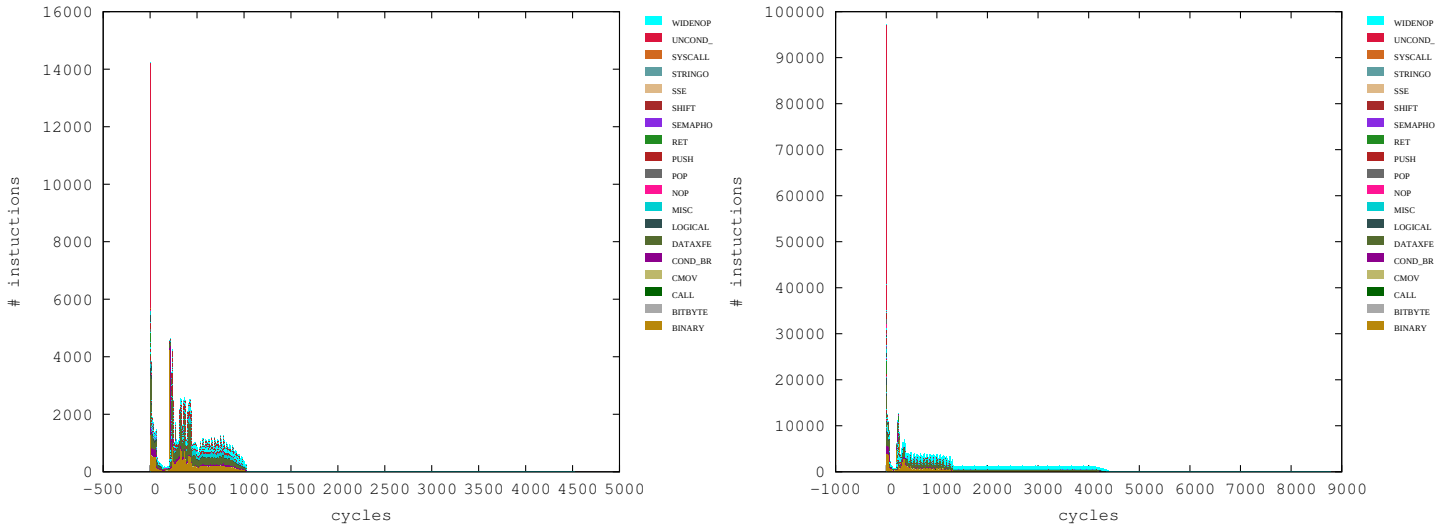

FIGURE 10 – Histogramme de 1M d'inst. de "cjpeg" à gauche et 7,2M à droite

5. Conclusion et travaux en cours et futurs

Cet article a fait ressortir que le graphe de dépendances des instructions d'une exécution se compose de cinq parties :

– le premier cycle, d'ILP maximum, où se mélangent toutes les initialisations de variables et tous les sauts immédiats (dont les instructions d'appel de fonctions "call") ;
– le second cycle, d'ILP très élevé, où les instructions de retour de fonctions remplacent celles d'appel et où on commence à utiliser les initialisations du premier cycle ;
– un préfixe de quelques centaines ou milliers de cycles (selon la longueur des boucles) d'ILP élevé, mélant le contrôle des boucles "for" à l'enchaînement des premières transmis-

sions de paramètres/résultats des premiers appels/retours ;
– une partie centrale assez plate, d'ILP faible, avec en certains endroits des excroissances de l'ILP provenant de boucles à démarrage différé (valeur initiale de la variable contrôlée dépendant d'un paramètre de fonction), et faites essentiellement de la concaténation des transmissions de paramètres/résultats ;
– une partie finale, très longue (d'autant plus longue que la trace s'agrandit), d'ILP très faible (voisin de 1), correspondant à l'enchaînement des appels/retours, représentés par la transmission des paramètres et résultats.

Cela suggère deux pistes assez différentes pour améliorer la performance des cœurs :
– faire migrer l'application d'un matériel disposant de ressources pour exploiter du parallélisme massif en début d'exécution vers un matériel adapté au calcul séquentiel, favorisant les chargements/rangements pour accélérer les transmissions de paramètres ;
– disposer d'un processeur de degré superscalaire moyen et de techniques permettant d'atteindre le parallélisme très distant (plusieurs dizaines de milliers de cycles) pour répartir un gros stock d'instructions prêtes dans le temps et ainsi, conserver un débit d'instructions exécutées constant, le plus proche possible de la performance crête.

Cela suggère aussi d'améliorer les codes en diminuant l'impact de la transmission des paramètres sur l'ILP (réduire la longueur de la queue du graphe de dépendances).

Notre travail sur l'ILP se poursuit avec le développement de l'outil Perpi d'analyse de l'ILP. Un travail futur sera la définition et l'évaluation d'une microarchitecture adaptée à la structure des graphes de dépendances.

Bibliographie

1. Rajeev Balasubramonian, Sandhya Dwarkadas and David H. Albonesi. Dynamically Allocating Processor Resources between Nearby and Distant ILP. *ISCA'01, 2001, 26–37*
2. Cristal, Adrián, Santana, Oliverio J., Valero, Mateo and Martínez, José F. Toward kilo-instruction processors *ACM Trans. Arch. Code Optim., Vol. 1, Issue 4, 389–417, Dec 2004*.
3. José González and Antonio González Data Value Speculation in Superscalar Processors *Microprocessors and Microsystems Volume 22, Issue 6, 30 November 1998, Pages 293-301* .
4. Bernard Goossens, Philippe Langlois, David Parello et Éric Petit Validated performance analysis of accurate summation algorithms *SCAN 2010*.
5. Norman P. Jouppi and David W. Wall Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines *ASPLOS-III, 1989, 272–282*.
6. Monica S. Lam and Robert P. Wilson Limits of Control Flow on Parallelism *ISCA'92, 1992, 46–57*.
7. Alexandru Nicolau and Joseph A. Fisher Measuring the Parallelism Available for Very Long Instruction Word Architectures *IEEE Trans. on Comp., Vol. c-33, NO. 11, Nov 1984, 968–976*.
8. G. S. Tjaden and M. Flynn Detection and Parallel Execution of Independent Instructions *IEEE Trans. on Comp., Vol. c-19, NO. 10, Oct 1970, 889–895*.
9. David W. Wall Limits of Instruction-Level Parallelism *WRL Technical Note TN-15, 1990*.
10. David W. Wall Limits of Instruction-Level Parallelism *WRL Research Report 93/6*.