De quoi est faite une trace d'exécution ?

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

RECHERCHE

De quoi est faite une trace d'exécution ?

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

DALI, Université de Perpignan Via Domitia F-66860 Perpignan cedex 9,

LIRMM, CNRS, UMR 5506, Université Montpellier 2 F-34095 Montpellier cedex 5,

{goossens,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 peut augmenter sans limite avec la longueur de la trace considérée et l'autre est d'ILP très faible (voisin de 1), qui peut s'allonger sans limite avec la trace. La première partie vient des contrôles des boucles à borne initiale constante alors que la seconde vient des transmissions de paramètres/résultats entre fonctions. Ces résultats montrent que, sans modification de la distribution du parallélisme d'instructions, un processeur doit être capable de concilier une très grande demande de ressources en début d'exécution avec une exécution quasi séquentielle en fin d'exécution.*

ABSTRACT. *This paper presents the structure of program execution traces. This study extends previous works from many researchers aiming to quantify ILP. Our goal is to understand the general structure of a run and the instruction parallelism it offers. This structure is mainly made of two opposing parts. One has a high ILP which can infinitely increase when the trace lengthens and the other has a very low ILP (close to 1) which can extend infinitely with the trace. The first part comes from loops controls (loops with a counter set from a constant). The second part comes from functions parameters transmissions. Our results show that, without any modification of the ILP distribution, a processor must conciliate a high resource demand at the start of the run and a nearly sequential execution at the end.*

MOTS-CLÉS : *parallélisme d'instructions, machine idéale, structure de trace.*

KEYWORDS: *instruction level parallelism, ideal machine, trace structure.*

DOI:10.3166/TSI.32.229-252

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 termes 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 tâches 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 communications,

– 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.

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, Flynn, 1970) ont mesuré la quantité de parallélisme disponible dans une fenêtre de 2 à 10 instructions. Avec un matériel adapté, on pourrait atteindre un IPC de 1.86.

En 1984, (Nicolau, Fisher, 1984) ont mesuré 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 (1 000 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 sur le parallélisme que le compilateur

peut détecter pour former des vecteurs d'instructions devant alimenter un processeur VLIW.

L'article de (Jouppi, Wall, 1989), qui étudie les différences entre exécution en parallèle (processeur superscalaire) et en série (processeur superpipeline) conclut que les deux sont équivalentes et que surtout, le parallélisme d'instructions disponible (c'est-à-dire exploitable) est très limité (6 instructions exécutables par cycle au mieux). Mais leur expérience ne comprend pas de mesure d'ILP. En particulier, les dépendances de contrôle sont conservées.

Au début des années 1990, (Wall, 1990) a mené la première véritable étude sur l'ILP. Sa note technique confirme que le taux de parallélisme exploitable est de l'ordre de 5 instructions (ce qui se trouve 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 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.

(Austin, Sohi, 1992) ont étudié l'ILP de benchmarks de la suite SPEC89. Par ailleurs, ils ont analysé la distribution des instructions le long des cycles sur une machine idéale avec prédiction parfaite des sauts et différents mécanismes de renommage (aucun renommage, renommage des registres, registres + pile, registres + mémoire). Leurs résultats montrent que le renommage est crucial à l'ILP. Ils ont aussi étudié l'effet d'une fenêtre d'instructions limitée et ont ainsi montré qu'une part importante de l'ILP est distante, provenant d'instructions arbitrairement loin du compteur de programme, et ne peut être exploitée que par un processeur ayant la capacité d'atteindre ces instructions distantes.

La même année, (Lam, Wilson, 1992) ont étudié l'impact du contrôle sur l'ILP. 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 en 1993 (Wall, 1993) en débridant la limite de lancement). Leurs résultats confirment ceux d'Austin et Sohi en montrant que de l'ILP distant existe. Ils concluent que pour capturer cet ILP, il faut que le processeur puisse faire de l'exécution spéculative (exécuter des blocs lointains en prédisant les sauts qui en contrôlent l'accès) et multiflot (exécuter en parallèle plusieurs fonctions indépendantes).

(Postiff *et al*., 1998) ont mesuré l'ILP présent dans la suite SPEC95. Ils ont surtout montré que les dépendances des références sur la pile sont un frein important à l'ILP. En éliminant les dépendances Lecture Après Ecriture sur le pointeur de pile, ils montrent que l'ILP progresse fortement (combiné avec le renommage de la mémoire qui élimine les dépendances Ecriture Après Ecriture ou Lecture, cela permet de voir la pile comme un arbre de cadre et d'exécuter en parallèle toutes les fonctions indépendantes). Ils remarquent qu'au premier cycle s'accumulent un très grand nombre d'instructions (initialisations, sauts immédiats).

(González, González, 1998) ont étudié l'impact de la prédiction de valeur, en particulier sur l'ILP. L'impact est très important lorsque l'on prédit les valeurs des instruc-

tions arithmétiques entières, surtout sur les codes flottants. D'autre part, un prédicteur parallèle (capable de prédire plusieurs valeurs d'une même instruction en parallèle) est bien plus efficace qu'un prédicteur en série (une seule prédiction à la fois).

(Balasubramonian *et al*., 2001) ont étudié les performances d'un processeur exécutant deux threads en parallèle, l'un principal et l'autre secondaire, chargés 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 ressource 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 micro-architecture sur une micro-architecture standard est de 17 % en moyenne pour un pic à 64 %.

En 2004, (Cristal *et al*., 2004) ont proposé une micro-architecture 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.

(Fahs *et al*., 2005) ont proposé une unité d'optimisation dynamique placée dans l'étage de renommage d'un processeur à exécution spéculative et en désordre. Ils ont évalué deux mécanismes, l'un visant à réduire le nombre de cycles d'exécution et l'autre visant à diminuer le nombre d'instructions exécutées. Le premier mécanisme propage les constantes et le second élimine les instructions de transfert entre registres. L'impact mesuré est une augmentation d'environ 10 % du nombre moyen d'instructions exécutées par cycle. La propagation des constantes et l'élimination des transferts entre registres sont des techniques très intéressantes pour agir sur l'ILP, notamment en le densifiant.

3. Analyse de la structure du parallélisme d'instructions

Les études sur l'ILP que nous avons mentionnées dans la section 1 ont essentiellement quantifié le nombre d'instructions exécutables à chaque cycle et pour deux d'entre elles, (Austin, Sohi, 1992) et (Postiff *et al*., 1998), ont permis de visualiser la distribution des instructions le long de l'exécution. En revanche, aucune étude n'a analysé le graphe de dépendances de ces instructions pour le relier aux structures du code source. Le graphe a été utilisé pour étudier d'une part l'évolution de l'ILP (Stefanovi, Martonosi, 2000) et d'autre part pour observer l'effet de certaines transformations du compilateur (Mak, 2011).

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 (les sauts sont tous résolus par un prédicteur parfait). Elle dispose aussi de ressources en quantité suffisante pour exécuter à chaque cycle tout ce qui est prêt (c'est-à-dire indépendant de toute donnée non calculée). La machine idéale est dotée d'opérateurs parfaits, opérant en un cycle (y compris les accès à la mémoire).

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). Les dépendances Lecture Après Ecriture sur les registres de pile rbp et rsp (en x86, les registres rbp et rsp encadrent le cadre de pile; rbp est la base de ce cadre et rsp en est le sommet) sont résolues par un prédicteur de valeur parfait.

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) :

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

3.1. Parallélisme relatif aux boucles

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

Dans ce code, le contrôle (instructions 4 à 6) est indépendant du corps (instruction 3). Le contrôle est exécutable dès le lancement du programme, un cycle après l'instruction d'initialisation 2 alors que le corps dépend du calcul de x. Indépendamment de la 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 $c > 1025$, ce sont les 1 024 instructions d'écriture en mémoire qu'on peut exécuter simultanément au cycle $c + 1$ puisqu'à cet instant, les 1 024 adresses des $t[i]$ ont été calculées et x vient de l'être.

De façon plus précise, l'exécution s'échelonne dans le temps comme indiqué figure 2. Sur cette figure, les instructions sont représentées dans une apparence dynamique.

L'instruction `xorl %eax, %eax` est écrite `eax=0`.

L'instruction `addl $1, %eax` devient en déroulant la boucle `eax=1` à la première itération, puis `eax=2` et ainsi de suite jusqu'à `eax=1024`.

Dans l'instruction de comparaison `cmpl $1024, %eax`, on a substitué au registre sa valeur dans l'itération associée : au premier tour, `cmp 1024,1`, au dernier tour, `cmp 1024,1024`. Chaque occurrence de l'instruction `jne loop` correspond à une itération avec `jne loop-1` pour la première et `jne loop-1024` pour la der-

nière. Enfin, l'unique instruction formant le corps de boucle `movl %ebx, t(%eax)` apparaît à la dernière ligne, sous la forme `t[0]=x` jusqu'à `t[1023]=x`.

```
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

Le contrôle de cette boucle ajoute peu au parallélisme du début de la trace, trois instructions par cycle tout au plus, comme le montre l'ordonnancement présenté figure 2. Sur cette figure, les instructions de contrôle d'une même itération forment une diagonale, par exemple `eax=1; cmp 1024,1; jne loop-1` pour la première itération.

En revanche, les 1 024 exécutions de l'instruction 3 (`movl %ebx, t(%eax)`) qui sont placées au cycle 10 001, en supposant que le registre `%ebx` qui contient x est établi au cycle 10 000, constituent un pic d'ILP pour un cycle.

```
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      jne loop-1
5  cycle[    5]   eax=4      cmp 1024,3      jne loop-2
6  ...
7  cycle[ 1024]   eax=1023   cmp 1024,1022   jne loop-1021
8  cycle[ 1025]   eax=1024   cmp 1024,1023   jne loop-1022
9  cycle[ 1026]              cmp 1024,1024   jne loop-1023
10 cycle[ 1027]                               jne loop-1024
11 ...
12 cycle[10001]   t[0]=x  t[1]=x  t[2]=x  ...  t[1023]=x
```

Figure 2. Exécution pipelinée du contrôle de boucle

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 1 024 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 1 025 initialisations de variables ($i = 0$ et 1 024 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 2 048 (1 024 incrémentations et 1 024 comparaisons), puis 3 072 le cycle suivant.

La traduction en assembleur x86 est à gauche de la figure 3. Cette traduction est peu optimisée. Les instructions des lignes 7 et 8 pourraient sortir de la boucle interne. Mais cela ne change pas l'observation du parallélisme massif du contrôle de la boucle interne.

Le corps de la boucle interne dépend de x, qui est établi après le retour de la fonction f. En supposant que celle-ci fournisse x en plus de 1 025 cycles, par exemple au cycle 10 000, les 1 024*1 024 exécutions de l'instruction `movb %al, t(%ecx)` correspondant à `t[i][j]=x` interviennent simultanément au cycle 10 001.

La figure 4 montre l'ordonnancement des initialisations. Le registre esi représente la variable contrôlée i de la boucle externe. Les noms $edi[0]$, ..., $edi[1023]$ désignent les 1 024 occurrences du registre edi, dédoublées par renommage et contenant les compteurs j de chacune des 1 024 exécutions de la boucle interne.

```
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+j                    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

Un outil de calcul automatique d'ILP tel que PerPI (Goossens *et al.*, 2012) montre que cette double boucle exécute 7 344 130 instructions en 1 028 cycles, soit un ILP de 7 144 (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 8 (a) montre l'histogramme de l'ILP pour les 1 028 cycles. On voit que dès les premiers cycles, plus de 5 000 instructions sont exécutables.

Un zoom ferait apparaître qu'on passe de 1 026 instructions au cycle 1 (dont 1 024 fois `movl $0, %edi`) à 2 049 au cycle 2 (dont 1 024 fois `movl %esi, %ecx` et

```
1 cycle[    1]   esi=0      edi[0]=0     ...  edi[1023]=0
2 cycle[    2]   esi=1      edi[0]=1     ...  edi[1023]=1
3 cycle[    3]   esi=2      edi[0]=2     ...  edi[1023]=2
4 ...
5 cycle[ 1024]   esi=1023   edi[0]=1023  ...  edi[1023]=1023
6 cycle[ 1025]   esi=1024   edi[0]=1024  ...  edi[1023]=1024
```

Figure 4. Exécution pipelinée des initialisations du contrôle de boucles imbriquées

`addl $1, %edi`). Il y a 4 098 instructions au cycle 3 (1 024 fois les précédentes et `sall $10, %ecx`, `cmpl $1023, %edi`) et enfin 5 126 au cycle 4 (les précédentes plus `jle .L4`). Ensuite, le nombre d'instructions exécutées augmente de 4 à chaque cycle, pour culminer à 9 271 instructions au cycle 1 025.

L'instruction `addl %edi, %ecx` (instruction 9 de la figure 3) est typique de ce qui rend très piégeux l'analyse à la main de l'ILP. Cette instruction est la dernière étape du calcul du déplacement $i*$ 1 024 $+j$ pour l'accès à $t[i][j]$. Dans un premier temps, c'est la source `ecx`, c'est-à-dire $i*$ 1 024, qui est la plus tardive, à cause du délai de calcul échelonné sur 3 cycles : établissement de i, transfert dans `ecx` et décalage. Pour $i = 0$, le registre `esi` qui contient i est fixé au cycle 1 et le registre `ecx` est prêt à la fin du cycle 3. Pour $i = 1$, le registre `esi` est fixé au cycle 2 et le registre `ecx` l'est à la fin du cycle 4. Il en est de même pour chaque i successif, avec un décalage d'un cycle à chaque itération.

```
1  cycle[    1] edi[0]=0      ... edi[1023]=0
2  cycle[    2] edi[0]=1      ... edi[1023]=1
3  cycle[    3] edi[0]=2      ... edi[1023]=2        ecx[0][0]=0
4                                                ... ecx[0][1023]=0
5  cycle[    4] edi[0]=3      ... edi[1023]=3        ecx[1][0]=1024
6                                                ... ecx[1][1023]=1024
7  cycle[    5] edi[0]=4      ... edi[1023]=4        ecx[2][0]=2048
8                                                ... ecx[2][1023]=2048
9  ...
10 cycle[1024] edi[0]=1023 ... edi[1023]=1023     ecx[1023][0]=1023*1024
11                                               ... ecx[1023][1023]=1023*1024
12 cycle[1025] edi[0]=1024 ... edi[1023]=1024
```

Figure 5. Ordonnancements comparés des contenus successifs des registres edi et ecx

La seconde source de l'instruction `addl %edi, %ecx` est le registre `edi` qui contient j. Comme le montre la figure 4, le registre `edi` est établi au cycle 1 pour les 1 024 initialisations de j à 0 des 1 024 itérations de la boucle externe. Il est établi au cycle 1 pour les 1 024 initialisations de j à 1 et ainsi de suite avec un décalage d'un cycle pour chaque incrémentation de j. La figure 5 montre les ordonnancements comparatifs des différents contenus des registres `edi` et `ecx`. Il y a 1 024*1 024 initialisations de `edi` et autant de calculs de `ecx` ($i*$ 1 024), ceux-ci ne dépendant pas de j mais seulement de i (on s'intéresse à la valeur de `ecx` avant de l'ajouter à `edi`). Sur

la figure, `ecx[0][0]` correspond à l'exécution de l'instruction `sall $10, %ecx` pour $i = j = 0$. De même, `ecx[1023][1023]` correspond à $i = j = 1\,023$.

On voit sur la figure 5 que pour le calcul de la somme à l'itération $i = j = 0$, instruction `addl %edi, %ecx`, la source `ecx` est calculée au cycle 3 alors que la source `edi` est prête au cycle 1. L'instruction est donc exécutée au cycle 4, en attente de la plus tardive des deux sources qui est `ecx`. Pour l'exécution de la même instruction à l'itération $i = 0$ et $j = 3$, la source `edi` est prête au cycle 4 (`edi[0]=3`) alors que la source `ecx` correspondante est calculée au cycle 3 (`ecx[0][3]=0`). L'addition attend cette fois la valeur de `edi`.

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 exécutés qu'après le calcul de n. En revanche, l'initialisation et les incrémentations restent indépendantes, donc exécutables dès le premier cycle (voir figure 6, à gauche). C'est encore moins favorable si la boucle est décroissante (voir figure 6, 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;
```

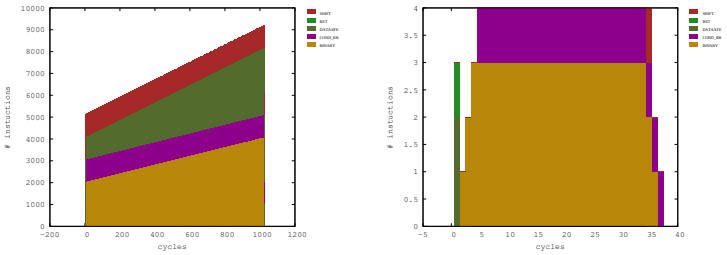
Figure 6. Boucles à bornes calculées dynamiquement

Pour ce qui est d'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 7, partie droite pour le code 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 7. Calcul de racine carrée : exemple de boucle dont le nombre d'itérations n'est pas connu

L'outil PerPI montre 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 8 (b) montre l'histogramme de l'ILP tout au long des 37 cycles d'exécution.



(a) (b)

Figure 8. Histogramme des traces de la double boucle (a) et "racine" (b)

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.

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 ($x = 2\,048$) et 3,95 pour 257 itérations ($x = 65\,536$). Dans l'exemple, cela tient à la dépendance de l'instruction `addl %eax,%ecx` avec elle-même (c+=i). A cause d'elle, on ne peut lancer plus d'une itération par cycle, ce qui fait que le nombre de cycles 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 9 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 9. Fusion de boucles

L'ILP local (de la boucle) est amélioré mais pas l'ILP global (du programme). Si une itération de la boucle $corps1$ se compose de $n1$ instructions et si une itération de $corps2$ en contient $n2$, 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.

L'ILP local de la boucle présentée figure 1 tend vers 4 (4 096 instructions exécutées en 1 024 cycles, soit $(4 * 1024)/1024$). L'ILP global de la boucle au sein d'un programme qui établit x au cycle 10 000 est de $(4 * 1024)/10001 = 0.41$ (les 3 * 1 024 instructions du contrôle s'exécutent en 1 024 cycles mais les 1 024 initialisations patientent jusqu'au cycle 10 001).

D'une façon générale, le compilateur, lorsqu'il procède à des améliorations par réorganisation du code qu'il produit, ne fait le plus souvent 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 augmenter 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. Quand elle est initialisée par une constante, le contrôle se place au début du graphe de dépendance, la chaîne de récurrence sur la variable contrôlée commençant au cycle 1. C'est plus pénalisant pour l'ILP quand la valeur initiale de la variable contrôlée est une variable. Dans ce cas, la chaîne de récurrence prend place au milieu du graphe, juste après le calcul de la valeur initiale.

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).

Par ailleurs, le compilateur produit des séquences conventionnelles d'appel et de retour de fonction. La figure 10(a) montre l'effet en pile de l'exécution d'une séquence

d'appel sur un processeur x86 (utilisé en mode 32 bits). Avant l'appel (exécution d'une instruction assembleur `call`), le sommet de pile est pointé par le registre `rsp`. Le registre `rbp` pointe sur la base du cadre de pile utilisé par la fonction en cours d'exécution (celle qui va effectuer l'appel, qui est désignée $appelante$ sur la figure).

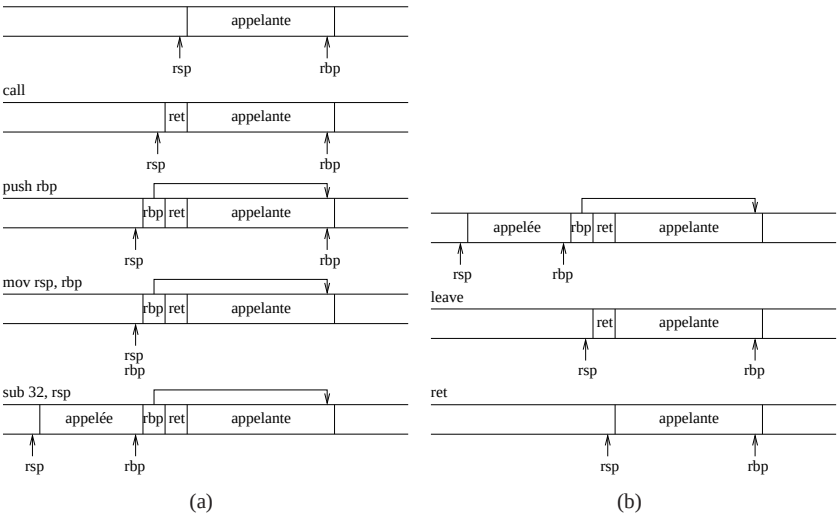


Figure 10. Séquence d'appel (a) et de retour (b)

La figure 10(b) montre l'effet en pile de l'exécution d'une séquence de retour. L'instruction `leave` enchaîne un déplacement du sommet de pile pour le ramener en `rbp` et un dépilement dans `rbp` de la base du cadre appelant. L'instruction `ret` dépile l'adresse de retour.

D'une façon générale, le compilateur produit un code qui alloue au sommet de pile un cadre pour la fonction appelée. Ce cadre est délimité par deux registres pointeurs. Le code d'allocation est placé en prologue de la fonction. En épilogue, le compilateur produit une séquence d'instructions qui replace les bornes de cadre à leurs valeurs avant l'appel, libérant le cadre de la fonction appelée et restaurant celui de la fonction appelante. En mode 64 bits, le compilateur gcc se contente d'un seul pointeur, la fin du cadre appelant étant confondue avec le début du cadre appelé.

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 11 (le code C est à 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).

```
1  hanoi: pushq %rbp              ;]ret, ... ,rbp]   hanoi(int n,int d,int a){
2         movq  %rsp, %rbp        ;]rsp, ... ,rbp]      int i = 6-d-a;
3         subq  $32, %rsp         ;32 octets            if (n==0) return;
4         movl  %edi, -20(%rbp)   ;sauver n             hanoi(n-1,d,i);
5         movl  %esi, -24(%rbp)   ;sauver d             deplacer(n,d,a);
6         movl  %edx, -28(%rbp)   ;sauver a             hanoi(n-1,i,a);
7         movl  $6, %eax          ;                  }
8         subl  -24(%rbp), %eax   ;
9         subl  -28(%rbp), %eax   ;                  deplacer(int n,int d,int a){
10        movl  %eax, -4(%rbp)    ;sauver 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 11. Tours de Hanoi : fonction récursive

La partie gauche de la figure 11 montre la traduction "gcc" de la fonction "hanoi" en assembleur x86 (sans optimisation ; on peut faire nettement mieux, même à la main).

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 8 fois.

En effet, hanoi(3) fixe son cadre et appelle hanoi(2), qui fixe son cadre à la suite de celui de hanoi(3). Puis hanoi(2) appelle hanoi(1) qui fixe son cadre à la suite de celui de hanoi(2). Enfin, hanoi(1) appelle hanoi(0), qui fixe son cadre à la suite. Il y a 4 cadres en séquence sur la pile : hanoi(3), hanoi(2), hanoi(1), hanoi(0). Au retour de hanoi(0), son cadre est libéré et on revient au cadre de hanoi(1), qui effectue son second appel à hanoi(0). Un nouveau cadre est alloué sur la pile, qui occupe la même place que celui de l'appel précédent à hanoi(0). Lors des appels suivants, on réutilisera encore deux autres fois ce même morceau de pile pour les deux autres cadres des deux autres appels à hanoi(0). Il en est de même des cadres des 4 appels à hanoi(1).

La figure 12(a) montre la position des cadres dans la pile au cours de l'exécution de "hanoi" sur une machine réelle. On voit que la même portion de mémoire est réutilisée pour héberger les cadres de différents appels de même niveau. Par exemple, la zone mémoire attribuée au cadre du premier appel à "hanoi(0 ...)" est réutilisée 8 fois pour chacun des 8 appels successifs à "hanoi(0 ...)". Sur la figure 12(b), on a une gestion de la pile en tas, ce qui permet de placer tous les cadres sans aucun recouvrement en mémoire et donc de paralléliser totalement l'exécution récursive. Sur cette partie de la figure, les 8 cadres des appels à "hanoi(0 ...)" sont disjoints en mémoire.

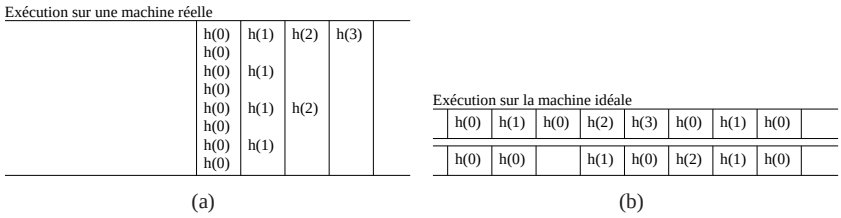


Figure 12. Cadres en pile de l'exécution de "hanoi" sur une machine réelle (a) ou idéale (b)

Pour le simulateur PerPI, à chacun des 32 octets qui composent chaque cadre sont associées autant de copies deux à deux distinctes que nécessaires, donc toutes utilisables en parallèle. Cela correspond à la figure 12(b).

Ainsi, toutes les instructions d'appels peuvent être parallélisées. Sur la figure 13(a) 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 à "hanoi(0 ...)", 4 appels à "hanoi(1 ...)", 2 appels à "hanoi(2 ...)", 1 appel à "hanoi(3 ...)" et 7 appels à "deplacer(...)"), tous exécutables dès le cycle 1. Tous les appels peuvent 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 empilements (instructions `pushq %rbp`) de `rbp` sont concentrés au cycle 1 puisque les bornes `rsp` et `rbp` de tous les cadres sont supposées fixées avant le cycle 1.

Les sauvegardes (instructions `movl %edi, -20(%rbp)` pour n, instructions `movl %esi, -24(%rbp)` pour d et `movl %edx, -28(%rbp)` pour a) se font dès que la variable à sauvegarder est connue. 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 ...)"). Pour les paramètres d et a, les valeurs initiales sont rangées au cycle 1, puis à chaque cycle un nouveau paramètre s'établit par transfert de registre (instruction `mov`). Ce sont ces transferts et calculs de paramètres qui créent des chaînes de dépendances, non seulement entre les appels ayant un lien de descendance mais aussi entre ceux n'en ayant pas.

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. On pourra observer sur l'histogramme de la figure 13(a) la pente marquée par les instructions de sauts conditionnels, cycles 4, 8, 12 et 16, au nombre de 1, 2, 4 et 8. 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).

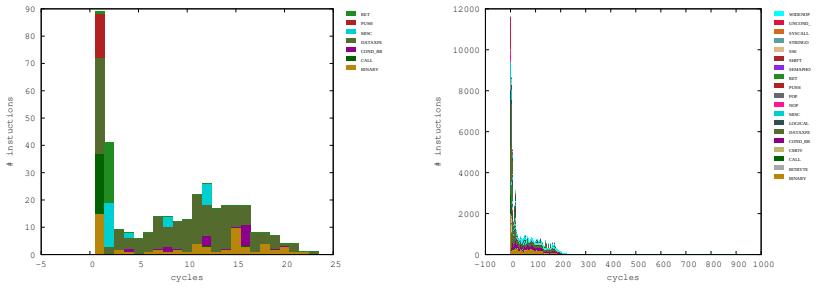
4. La structure d'une trace d'exécution

Dans cette section, les résultats rapportés, et notamment les histogrammes, sont issus de l'outil PerPI. Les programmes (benchmarks issus de la suite Mibench) ont été compilés sous Linux-Ubuntu-09-04, avec gcc 4.4, en mode 64 bits avec un niveau d'optimisation "O3". L'outil PerPI est un outil "pin" (Luk *et al.*, 2005). Le programme à analyser est exécuté sous le contrôle de "pin" ("pin" agit comme un déboggeur). Pin instrumente le code (PerPI est le code d'instrumentation) au cours de son exécution. L'instrumentation de PerPI s'applique à chaque instruction machine exécutée, juste avant son exécution. L'instrumentation démarre au début de la fonction "main"

et s'arrête à son retour. Les bibliothèques sont instrumentées mais pas les appels systèmes.

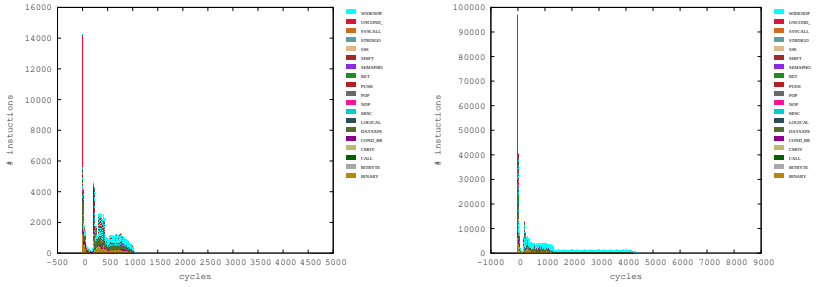
L'instrumentation tient à jour le nombre d'instructions exécutées ainsi que le cycle d'exécution le plus élevé. Le rapport des valeurs finales obtenues donne l'ILP. Pour chaque instruction, PerPI calcule son cycle d'exécution à partir des cycles d'exécutions des instructions produisant les sources, avec un cycle de délai.

Les exemples de codes des sections précédentes ont illustré que pour les boucles, les chaînes séquentielles limitant l'ILP sont une succession d'incrémentations 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.



(a) (b)

Figure 13. Histogramme des traces de "hanoi" (a) et "stringsearch" (b)



(a) (b)

Figure 14. Histogramme de "cjpeg", 1M d'instructions (a), 7,2M d'instructions (b)

La figure 13(b) montre l'histogramme de "stringsearch" et la figure 14(a) est l'histogramme de "cjpeg", deux des applications de la suite Mibench. Dans le cas de "stringsearch", l'histogramme présenté correspond à l'exécution complète de l'application sur le jeu de données "small". Pour "cjpeg", les histogrammes correspondent à une exécution tronquée sur le jeu de données "large".

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" (161 119 instructions exécutées en 946 cycles; ILP = 170), il y a 11 638 instructions exécutables au cycle 1 (dont 937 instructions `call`) et 3 540 au cycle 2 (dont 937 instructions `ret`). Pour "cjpeg", seules les 1M instructions du début sont représentées, exécutées en 4 520 cycles (ILP = 221). Au premier cycle, il y a 14 214 instructions exécutées. Au second cycle, il y en a 4 835. 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 3 500 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, les nouveaux appels de fonctions concatènent en fin de graphe les transmissions des paramètres dépendant des résultats les plus tardifs. La figure 14(b) 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 14 214 à 97 148 et le nombre total de cycles d'exécution passe de 4 520 à 8 774.

On déduit de ces observations 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 des paramètres ainsi que sur les récurrences dans les boucles. Il faut favoriser les sous-programmes plutôt longs et peu nombreux avec peu de paramètres/résultats et les boucles avec peu d'itérations de corps longs. Pour cela, les optimisations d'*inlining* et de déroulage de boucle des compilateurs sont à mettre à contribution.

Par exemple (voir la figure 15), une boucle de 2^{20} itérations crée une chaîne d'autant d'incrémentations de la variable contrôlée. En transformant cette boucle en une boucle imbriquée avec une boucle externe et une boucle interne à 2^{10} tours chacune, on fait disparaître la chaîne de longueur 2^{20} pour la remplacer par 2^{10} chaînes de longueur 2^{10}.

```
1  for (i=0;i<(1<<20);i++)          for (i=0;i<1024;i++)
2    {corps1(i)}                      for (j=0;j<1024;j++)
3                                       {corps1(i*1024+j)}
```

Figure 15. Transformation de boucle pour favoriser l'ILP

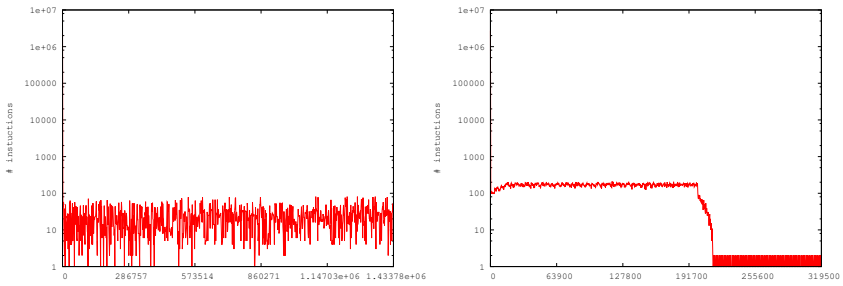
Les figures 16, 17, 18, 19, 20, 21 et 22 montrent les histogrammes de 14 applications de la suite Mibench ("basicmath", "quicksort", "cjpeg", "djpeg", "dijkstra", "pa-

tricia", "rawcaudio", "rawdaudio", "crc", "stringsearch", "susan -s", "susan -e", "susan -c" et "bitcount"). Ces applications ont été exécutées jusqu'à leur terme pour celles qui comptaient moins de 40M d'instructions ("djpeg", 20,7M, "stringsearch", 3,2M et "susan -c", 24M). Les autres ont été tronquées après 40M d'instructions exécutées. Toutes ont été exécutées avec en entrée le jeu de données "large".

L'échelle des ordonnées est logarithmique. En abscisse, pour alléger les figures, plutôt que de représenter tous les points de l'histogramme (parfois plusieurs millions), nous avons choisi de sélectionner 1 000 points également répartis sur l'intervalle des x. La légende de l'axe fait systématiquement apparaître 5 jalons équidistants. Par exemple, la figure 20(b) présente l'extrait de l'histogramme de "stringsearch". La totalité de l'exécution prend 49 915 cycles. Les jalons sont placés en $49915/5 = 9983$, $49915 * 2/5 = 19966$ et ainsi de suite. Les points choisis pour figurer dans l'extrait d'histogramme sont pris arbitrairement tous les millièmes. Pour "stringsearch", il s'agit du nombre d'instructions exécutées aux cycles 1, 51, 101, 151 ..., soit tous les 50 cycles ($49915/1000 + 1$) pour les 915 premières valeurs ($49915\%1000$) et tous les 49 cycles ($49915/1000$) pour les 85 dernières ($1000 - 49915\%1000$). Le choix arbitraire peut parfois faire apparaître une situation plus ou moins favorable que la tendance réelle autour du point choisi.

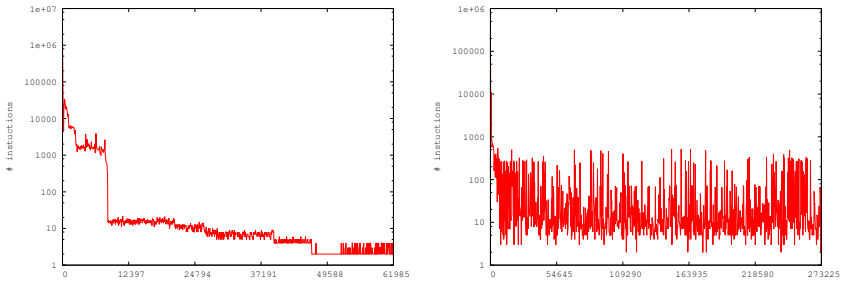
On peut remarquer que les premiers cycles présentent toujours un pic. Le tableau 1 récapitule le nombre d'instructions exécutées, le nombre de cycles de l'exécution sur la machine idéale, l'ILP et les proportions d'instructions exécutées aux deux premiers cycles pour chacun des 14 benchmarks. Cette proportion varie entre 0,45 % pour "susan -e" (2 cycles sur 3 545 soit 0,06 %) et 17,4 % pour "rawcaudio" (2 cycles sur 4 836 737). Dans le cas de "susan -e" et "susan -c", les figures 21 et 22 montrent que d'une part l'ILP reste fort tout au long de l'exécution et que d'autre part, un second pic apparaît vers le cycle 300.

La seconde remarque est la longueur de la queue, c'est-à-dire la portion de cycles pendant laquelle l'ILP est très bas, en dessous de 5. Nos exécutions ayant été tronquées parfois très précocement (une exécution complète de "basicmath" représente plus de deux milliards d'instructions, dont nous n'avons retenu que les premières 40 millions), la queue est parfois inexistante. C'est le cas de "basicmath", avec un ILP constant tout au long de l'exécution, voisin de la moyenne de 28 instructions par cycle. C'est aussi le cas de "patricia". Dans d'autres cas, la queue commence très vite. C'est le cas dès les premiers cycles de "crc" (l'ILP moyen est de 12 essentiellement grâce aux deux premiers cycles) et de "bitcount". Dans d'autres cas, l'ILP chute brutalement après un long préfixe, au-delà de la moitié de l'exécution. C'est le cas de "qsort" avec un effondrement vers le cycle 210 000, de "cjpeg" avec une longue descente qui passe en dessous des 5 instructions par cycle vers le cycle 40 000, de "dijkstra" qui chute au cycle 70 000 et de "rawcaudio" et "rawdaudio" qui passent en dessous de 5 instructions par cycle vers le cycle 1,6M. Le tableau 2 donne le pourcentage de cycles où l'ILP est en dessous de 5, 10 et 40. Pour 6 benchmarks sur 14, la queue de l'ILP inférieur à 5 représente plus de 30 % des cycles de l'exécution et pour 9 sur 14, la queue de l'ILP inférieur à 40 représente plus de 30 % des cycles de l'exécution.



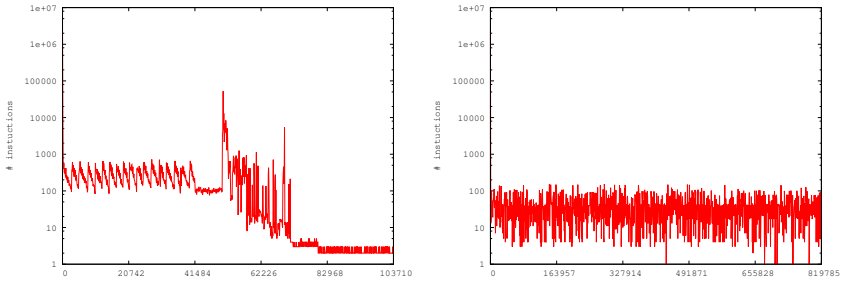
(a) (b)

Figure 16. Histogramme de 40M d'instructions de "basicmath" (a) et "qsort" (b)



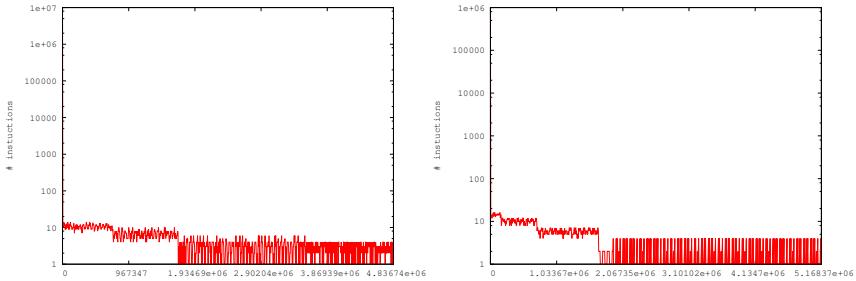
(a) (b)

Figure 17. Histogramme de 40M d'instructions de "cjpeg" (a) et de "djpeg" en entier (20 686 087 instructions) (b)



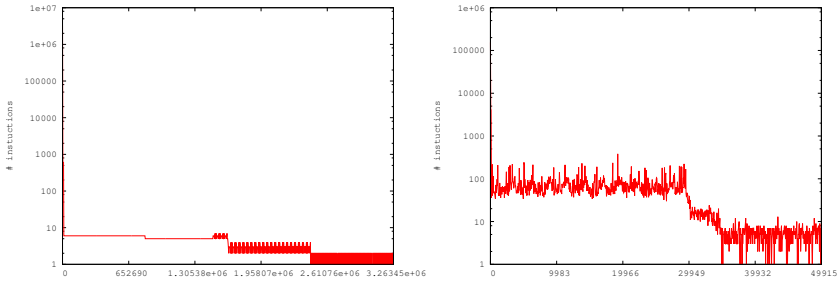
(a) (b)

Figure 18. Histogramme de 40M d'instructions de "dijkstra" (a) et "patricia" (b)



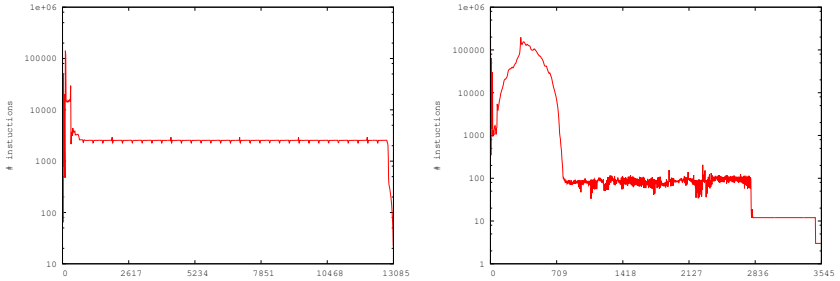
(a) (b)

Figure 19. Histogramme de 40M d'instructions de "rawcaudio" (a) et "rawdaudio" (b)



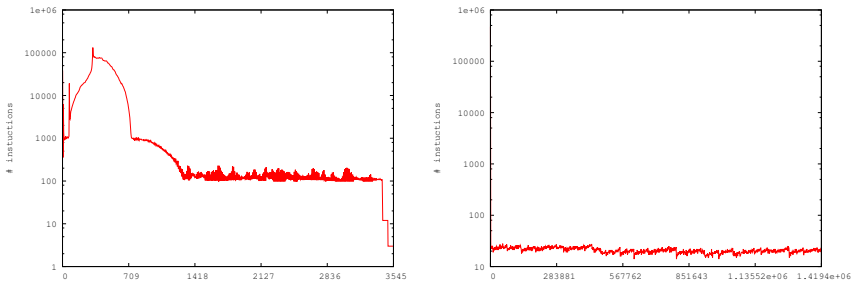
(a) (b)

Figure 20. Histogramme de 40M d'instructions de "crc" (a) et de "stringsearch" en entier (3 184 888 instructions) (b)



(a) (b)

Figure 21. Histogramme de 40M d'instructions de "susan s" (a) et "susan e" (b)



(a) (b)

Figure 22. Histogramme de tout "susan c" (23 973 515 instructions) (a) et 40M d'instructions de "bitcount" (b)

Tableau 1. Mesures d'ILP pour 14 benchmarks de la suite Mibench

Name	#instructions	#cycles	ILP	#i au cycle 1	%	#i au cycle 2	%
bitcount	40M	1419405	28.18	473253	1.18 %	946325	2.37 %
basicmath	40M	1433786	27.90	2575217	6.44 %	649260	1.62 %
cjpeg	40M	61985	645.32	1446105	3.62 %	342164	0.86 %
crc	40M	3263454	12.26	3264744	8.16 %	1632367	4.08 %
djpeg	20686087	273225	75.71	266606	1.29 %	26558	0.13 %
dijkstra	40M	103714	385.68	2138358	5.35 %	380196	0.95 %
patricia	40M	819787	48.79	2400831	6.00 %	819664	2.05 %
qsort	40M	319503	125.19	2714555	6.79 %	799052	2.00 %
rawcaudio	40M	4836737	8.27	4526581	11.32 %	2430481	6.08 %
rawdaudio	40M	5168370	7.74	880789	2.20 %	874765	2.19 %
search	3184888	49918	63.80	154910	4.86 %	70131	2.20 %
susanc	23973515	3546	6760.72	38434	0.16 %	110182	0.46 %
susane	40M	3545	11283.50	177966	0.44 %	5650	0.01 %
susans	40M	13087	3056.47	220660	0.55 %	1652	0.00 %

5. Conclusion

Pour profiter de l'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 une exécution purement séquentielle et *in-order* pour les instructions se trouvant en fin de graphe. Il semble crucial de compacter l'ILP pour en premier lieu réduire la longueur de la queue à faible ILP. Pour cela il faut agir sur le contrôle des boucles, par exemple en appliquant une technique de propagation dynamique des constantes (proposée dans (Fahs *et al.*, 2005)) pour rendre les occurrences d'une instruction récurrente dépendantes de la seule initialisation ou une technique statique de décomposition des boucles longues par imbrication. Il faut aussi agir sur la transmission des paramètres, par exemple en procédant à une élimination dynamique des transferts entre registres ou en intensifiant les techniques d'inlining lors de la compilation.

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).

Tableau 2. Portion d'exécution à faible ILP

Name	n° cycle ilp<5	%	n° cycle ilp<10	%	n° cycle ilp<40	%
bitcount	1419394	0.00 %	1419386	0.00 %	29	100.00 %
basicmath	1433765	0.00 %	1433754	0.00 %	20	100.00 %
cjpeg	39582	36.14 %	30059	51.51 %	8461	86.35 %
crc	1631748	50.00 %	12323	99.62 %	12323	99.62 %
djpeg	273213	0.00 %	273213	0.00 %	261814	4.18 %
dijkstra	71383	31.17 %	71383	31.17 %	71383	31.17 %
patricia	819786	0.00 %	819783	0.00 %	819745	0.00 %
qsort	214584	32.84 %	214583	32.84 %	208731	34.67 %
rawcaudio	1688585	65.09 %	828486	82.87 %	1004	99.98 %
rawdaudio	1689046	67.32 %	726236	85.95 %	1003	99.98 %
search	49917	0.00 %	49914	0.00 %	30020	39.86 %
susanc	3485	1.72 %	3485	1.72 %	3427	3.36 %
susane	3483	1.72 %	3483	1.72 %	2792	21.24 %
susans	13084	0.02 %	13082	0.04 %	13067	0.15 %

Le parallélisme mesuré par PerPI est celui présent dans le programme en faisant abstraction de l'architecture et de la micro-architecture. Les registres architecturaux et la mémoire, dont la pile, sont renommables à l'infini. Les sauts sont parfaitement prédits, les opérations ont une latence d'un cycle, y compris les accès à la mémoire. PerPI peut être ajusté pour mesurer un temps d'exécution plus réaliste. Il suffit d'ajouter des contraintes, représentatives de la micro-architecture ou de l'architecture, sur le calcul du cycle d'exécution d'une instruction. Une telle version existe qui simule l'équivalent d'un cœur *out-of-order* actuel. Les mesures de cette version du simulateur sortent du cadre de cette étude. Néanmoins, la seule contrainte d'extraction (hiérarchie mémoire d'instructions et prédicteur de sauts hybride à deux niveaux) suffit à ramener l'ILP en dessous de 10.

Le 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 variables contrôlées 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.

Le graphe de dépendances des instructions d'une exécution est structuré comme suit :

– 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" et les initialisations des contrôles de boucles, par exemple "for i=0") ;

– une séquence de longueur variable où s'additionnent les ILP de toutes les boucles et de toutes les chaînes de calculs démarrant par une constante ; cette part d'ILP s'amenuise au fur et à mesure que les boucles s'achèvent ;

– une séquence de longueur variable et d'ILP très faible, voisin de 1, où s'enchaînent les transmissions de paramètres/résultats des successions d'appels/retours de fonctions ;

– en certains endroits des montées de l'ILP provenant de boucles ou de calculs différés (valeur initiale du calcul ou de la variable contrôlée dépendant d'un paramètre de fonction) ;

Cela nous 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 ou la prédiction de valeurs pour accélérer les contrôles de boucles ;

– disposer d'un processeur de degré superscalaire moyen et de techniques multiflots permettant d'atteindre le parallélisme très distant (plusieurs dizaines de milliers de cycles) pour répartir dans le temps un gros stock d'instructions prêtes et ainsi conserver un débit d'instructions exécutées constant, le plus proche possible de la performance crête.

Bibliographie

Austin T., Sohi G. S. (1992). Dynamic dependency analysis of ordinary programs. In *Isca*, p. 342-351.

Balasubramonian R., Dwarkadas S., Albonesi D. H. (2001). Dynamically allocating processor resources between nearby and distant ilp. In *Isca*, p. 26-37.

Cristal A., Santana O., Valero M., Martínez J. (2004). Toward kilo-instruction processors. *ACM Trans. Arch. Code Optim.*, vol. 1, n° 4, p. 389-417.

Fahs B., Rafacz T., Patel S. J., Lumetta S. S. (2005). Continuous optimization. In *Isca*, p. 86-97.

González J., González A. (1998). Data value speculation in superscalar processors. *Microprocessors and Microsystems*, vol. 22, n° 6, p. 293-301.

Goossens B., Langlois P., Parello D., Petit E. (2012). Perpi: A tool to measure instruction level parallelism. *Springer LNCS*, vol. 7133, p. 270-281.

Jouppi N. P., Wall D. W. (1989). Available instruction-level parallelism for superscalar and superpipelined machines. In *Asplos*, p. 272-282.

Lam M., Wilson R. (1992). Limits of control flow on parallelism. In *Isca*, p. 46-57.

Luk C., Cohn R., Muth R., Patil H., Klauser A., Lowney G. *et al*. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Pldi '05: Proceedings of the 2005 acm sigplan conference on programming language design and implementation*, p. 190-200.

Mak J. (2011). *Facilitating program parallelisation: a profiling-based approach*. Rapport technique n° UCAM-CL-TR-796. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-796.pdf

Nicolau A., Fisher J. A. (1984). Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. on Comp.*, vol. 33, n° 11, p. 968-976.

Postiff M. A., Greene D. A., Tyson G. S., Mudge T. N. (1998). The limits of instruction level parallelism in spec95 applications. In *Interact-3, workshop on interaction between compilers and computer architecture.*

Stefanovi D., Martonosi M. (2000). Limits and graph structure of available instruction-level parallelism. *Springer LNCS*, vol. 1900, p. 1018-1022.

Tjaden G. S., Flynn M. (1970). Detection and parallel execution of independent instructions. *IEEE Trans. on Comp.*, vol. 19, n° 10, p. 889-895.

Wall D. W. (1990). *Limits of instruction-level parallelism*. technical note TN-15. WRL.

Wall D. W. (1993). *Limits of instruction-level parallelism*. research report 93/6. WRL.

Article reçu le 3 novembre 2011

Accepté après révisions le 27 septembre 2012

Bernard Goossens *est professeur à l'université de Perpignan. Il est responsable de l'axe "Performance des calculs" au sein de l'équipe DALI qui est une équipe-projet du LIRMM. Ses travaux de recherche concernent la micro-architecture des processeurs, et depuis une dizaine d'années, le parallélisme d'instructions.*

David Parello *est maître de conférences à l'université de Perpignan. Il est membre de l'axe "Performance des calculs" au sein de l'équipe DALI. Ses travaux de recherche concernent la micro-architecture des processeurs et leur simulation.*

Ali El Moussaoui et **Ke Chen** *sont doctorants à l'université de Perpignan. Ils effectuent leur travail de thèse sous la co-direction de B. Goossens et de D. Parello. La thèse d'Ali porte sur la simulation et la mesure d'ILP. Celle de Ke est consacrée à la parallélisation automatique des programmes.*