Ecole thématique Archi'2011, Mont-Louis, 13–17 juin 2011

PerPi: Performance et Parallélisme d'instruction

Bernard Goossens, David Parello, Mourad Bouache, Ali El Moussaoui et Ke Chen

Université de Perpignan Via Domitia, France
DALI, équipe LIRMM

UPVD
Université de Perpignan Via Domitia

DALI
Digits, Architectures et Logiciels Informatiques

Plan de l'exposé.

Qu'est-ce que l'ILP?

ILP: Instruction Level Parallelism / Parallélisme d'instruction.
L'ILP mesure la capacité d'exécution en parallèle des instructions (sur un matériel aux ressources illimitées).
On définit l'ILP d'un programme p exécuté sur un jeu de données d ainsi:

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

Le terme n est le nombre d'instructions machines exécutées.
Le terme c est le nombre de cycles machines de l'exécution.
Qu'est-ce qu'une instruction machine?
Qu'est-ce qu'un cycle machine?

Quelle machine?

- Machine idéale: contrainte que par les liens de production/consommation.
- Jeu d'instruction de la machine idéale: jeu d'instructions x86.
- Cycle de la machine idéale: temps d'exécution d'une instruction x86.
- Exécution atomique, de l'extraction à la sortie.

Comment fonctionne une machine idéale?

- Exécution dirigée par les données.
- Démarrage le cycle suivant le plus tardif des octets consommés.
- Terminaison: écriture de tous les octets produits.
- A chaque cycle, exécution en parallèle des instructions prêtes.

Les sauts dans une machine idéale.

- Aucune dépendance de contrôle.
- Trace disponible au début de l'exécution.
- Cycle 1: instructions constantes de la trace.
- Cycle n: instructions de la trace déclenchées par instructions du cycle $n-1$.
- Sauts: instructions sans successeur.

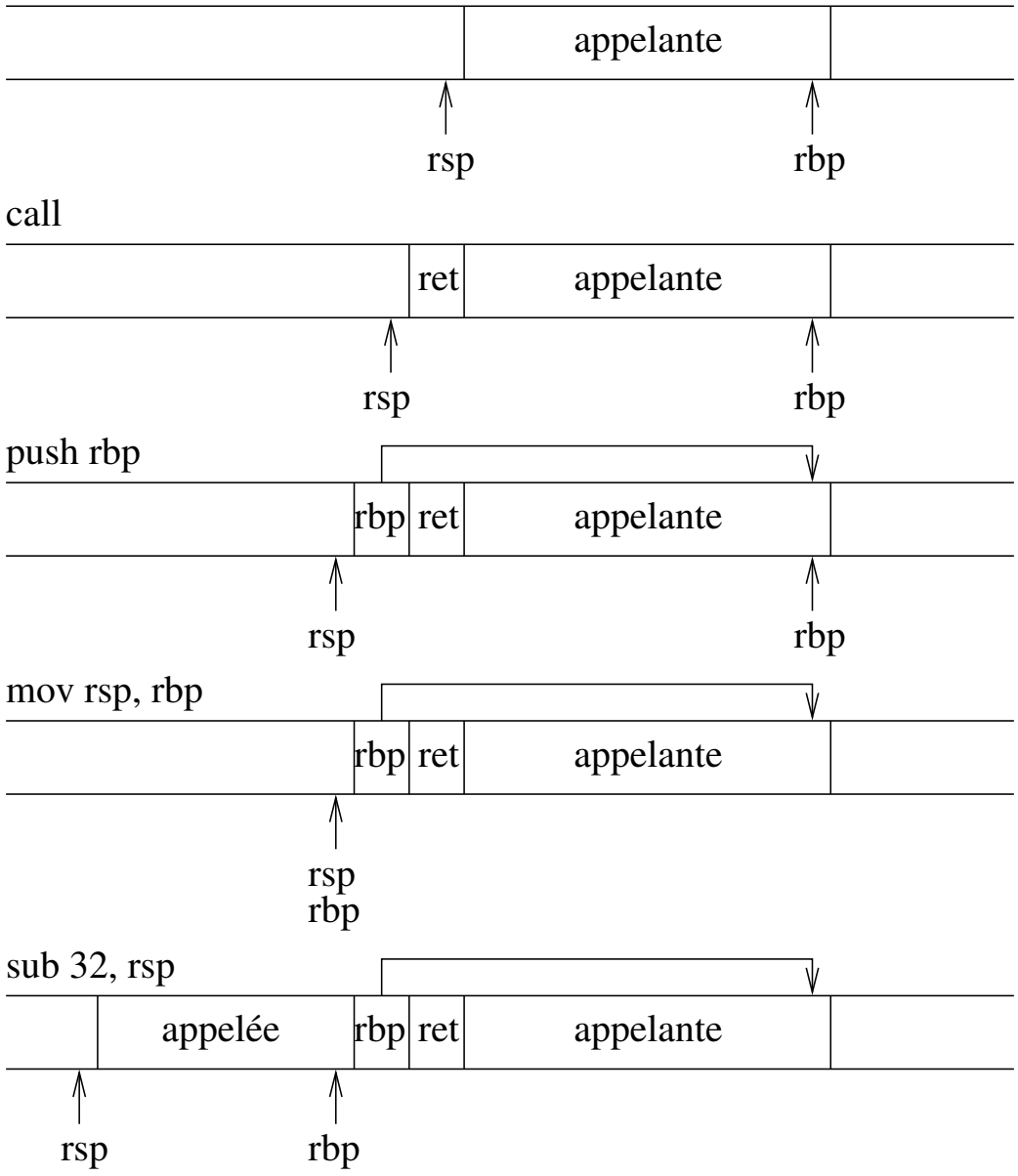
La mémoire dans une machine idéale.

- Dépendances de données (producteur/consommateur).
- Trace mémoire disponible au début de l'exécution.
- Dépendances WAR éliminées par renommage (calcul dans m en parallèle avec utilisation de m).
- Dépendances WAW éliminées par renommage (deux calculs dans m en parallèle).
- Lecture: dépend de l'adresse et de la donnée lue (mémoire).
- Ecriture: dépend de l'adresse et de la donnée à écrire (registre).

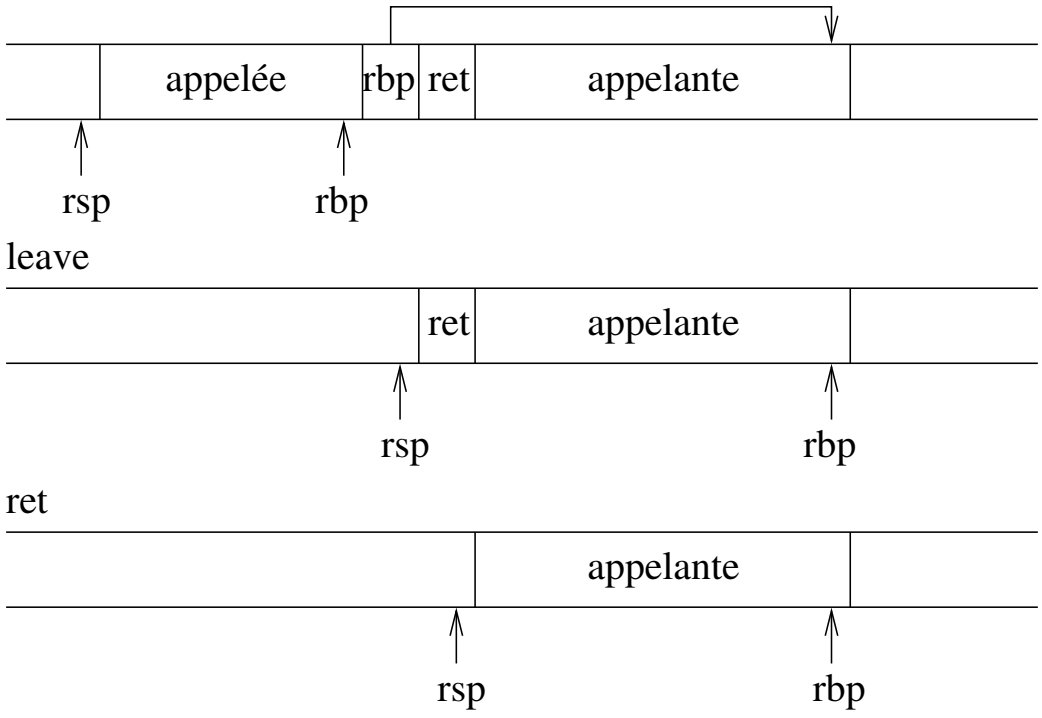
La pile dans une machine idéale.

- Renommage spécial PUSH/POP/CALL/RET/SP+=k/SP-=k.
- SP dépend de SP initial +/- déplacement constant connu au renommage.
- CALL ne dépend de rien (cycle 1).
- RET dépend de CALL (adresse de retour empilée) (cycle 2).
- PUSH dépend de la donnée à empiler (ne dépend pas de SP, empilement ooo).
- POP dépend de la donnée dépilée (PUSH) (ne dépend pas de SP, dépilement ooo).
- Empilement et dépilement ooo, même dans cadres en recouvrement (renommage).

Désambiguer les places en pile: Séquence d'appel.



Désambiguer les places en pile: Séquence de retour.



Désambiguer les places en pile: Exécution de hanoi.

Exécution sur une machine réelle

	h(0) h(0) h(0) h(0) h(0) h(0) h(0) h(0)	h(1) h(1) h(1) h(1)	h(2) h(2) 	h(3) 	

Exécution sur la machine idéale

	h(0)	h(1)	h(0)	h(2)	h(3)	h(0)	h(1)	h(0)	
	h(0)	h(0)		h(1)	h(0)	h(2)	h(1)	h(0)	

- Machine réelle: appels séquentialisés par réutilisation des cadres.
- Machine idéale: appels parallélisés par allocation en tas des cadres.

Un exemple simple de calcul d'ILP.

e = (a+b) + (c+d)

code machine x86

```
...
i1  mov   -20(ebp),eax
i2  add   -16(ebp),eax
i3  mov   -12(ebp),ecx
i4  add   -8(ebp),ecx
i5  add   ecx,eax
i6  mov   eax,-4(ebp)
...
```

Un exemple simple de calcul d'ILP.

e = (a+b) + (c+d)

code machine x86

```
...
i1  mov   -20(ebp),eax
i2  add   -16(ebp),eax
i3  mov   -12(ebp),ecx
i4  add   -8(ebp),ecx
i5  add   ecx,eax
i6  mov   eax,-4(ebp)
...
```

Instructions exécutées

Cycle 1: i1 i3

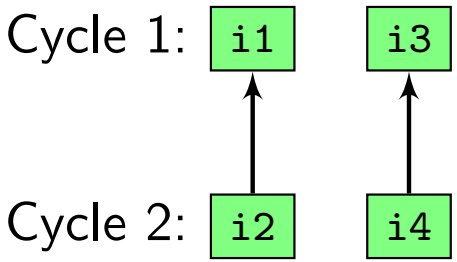
Un exemple simple de calcul d'ILP.

e = (a+b) + (c+d)

code machine x86

```
...
i1  mov   -20(ebp),eax
i2  add   -16(ebp),eax
i3  mov   -12(ebp),ecx
i4  add   -8(ebp),ecx
i5  add   ecx,eax
i6  mov   eax,-4(ebp)
...
```

Instructions exécutées



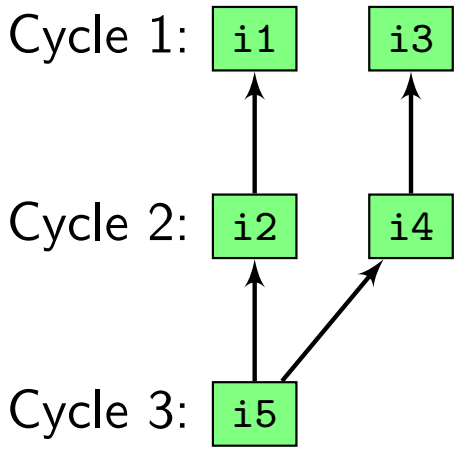
Un exemple simple de calcul d'ILP.

e = (a+b) + (c+d)

code machine x86

```
...
i1  mov   -20(ebp),eax
i2  add   -16(ebp),eax
i3  mov   -12(ebp),ecx
i4  add   -8(ebp),ecx
i5  add   ecx,eax
i6  mov   eax,-4(ebp)
...
```

Instructions exécutées



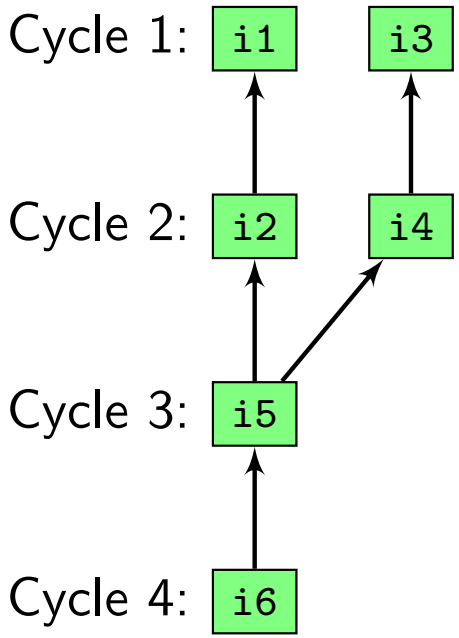
Un exemple simple de calcul d'ILP.

e = (a+b) + (c+d)

code machine x86

```
...
i1  mov   -20(ebp),eax
i2  add   -16(ebp),eax
i3  mov   -12(ebp),ecx
i4  add   -8(ebp),ecx
i5  add   ecx,eax
i6  mov   eax,-4(ebp)
...
```

Instructions exécutées



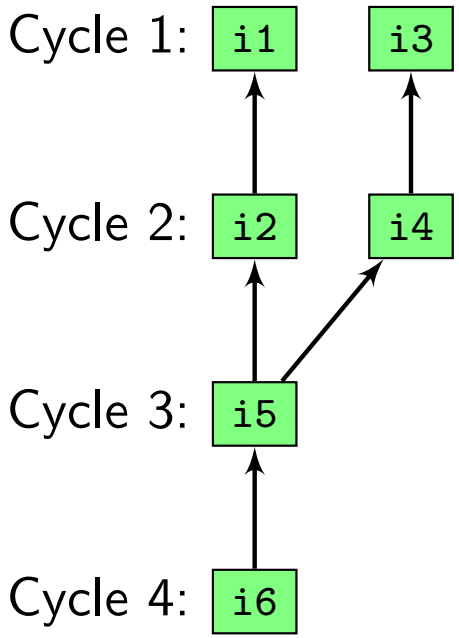
Un exemple simple de calcul d'ILP.

e = (a+b) + (c+d)

code machine x86

```
...
i1  mov   -20(ebp),eax
i2  add   -16(ebp),eax
i3  mov   -12(ebp),ecx
i4  add   -8(ebp),ecx
i5  add   ecx,eax
i6  mov   eax,-4(ebp)
...
```

Instructions exécutées



6 instructions, 4 cycles

ILP = 6/4 = 1.5

PerPi: un outil PIN.

PerPi: un outil PIN.

- Une fonction: calcul d'ILP à la main.
- Une application: 1s de temps de calcul = 6G instructions.
- Calcul automatique avec PerPi.
- PerPi est un outil PIN.
- PerPi n'exécute pas l'application: c'est le processeur.
- PerPi ne connait pas le langage machine du processeur (x86): c'est PIN.

PIN: principe de fonctionnement.

- Ce n'est ni l'application ni PerPI le programme principal: c'est PIN.
- PIN est un JIT compiler.
- PIN charge le code de l'application dans sa mémoire.
- PIN instrumente l'exécutable de l'application.
- PIN combine l'exécutable avec l'instrumentation PerPi.
- PIN contrôle: exécute le code à mesurer, exécute la mesure PerPi.

PIN: préparation du code.

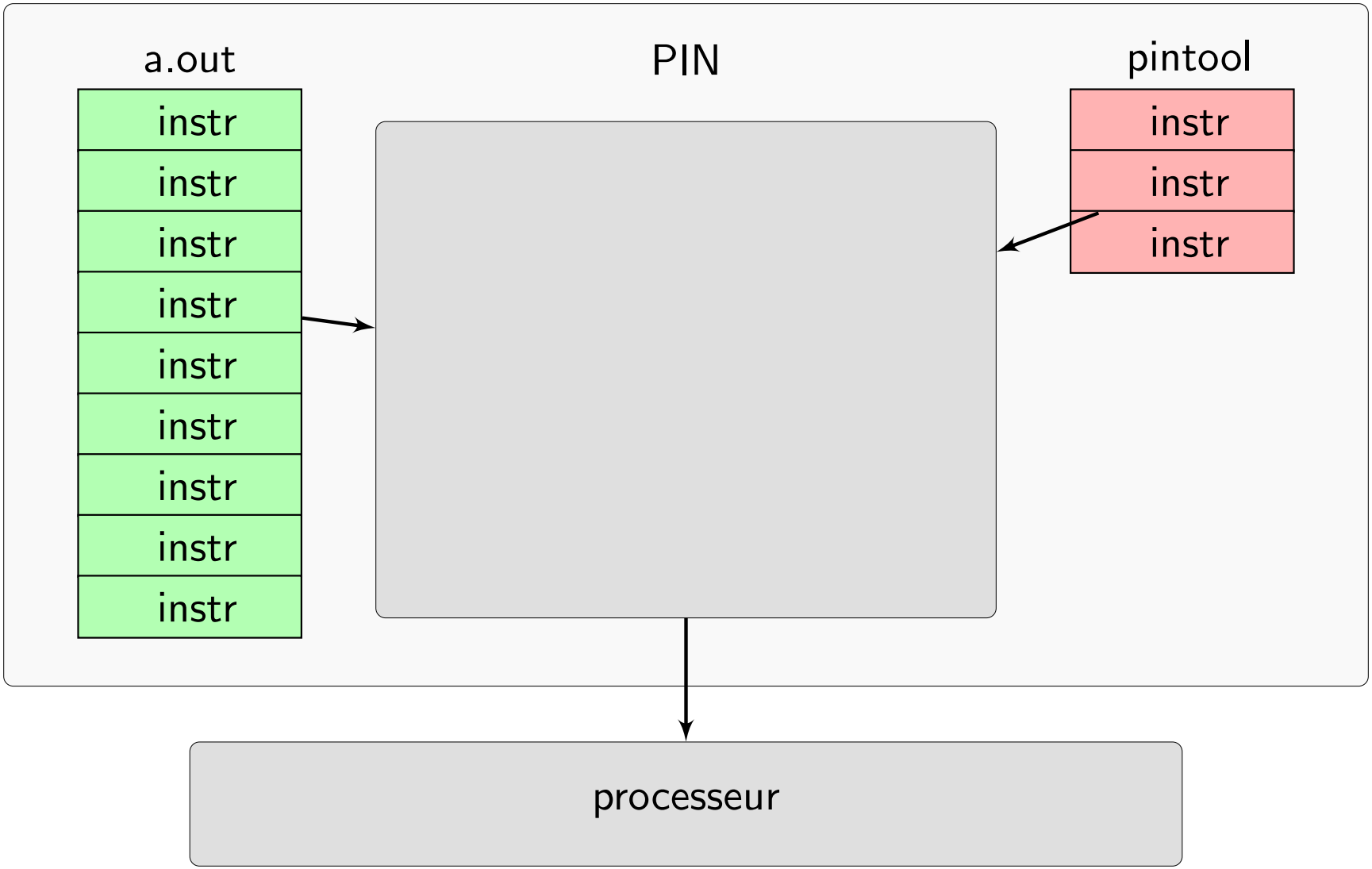
- PIN charge le bloc de base suivant de l'application à mesurer.
- PIN remappe les registres employés par le morceau de code.
- PIN ajoute le code d'instrumentation, lui aussi remappé.
- Avant/Après Instruction/Bloc de base/Routine.
- Directives d'instrumentations de PerPi: Avant Instruction.
- PIN ajoute la fonction d'analyse de PerPi avant chaque instruction x86 de l'application à mesurer.
- PIN ajoute un retour en fin de bloc pour récupérer le contrôle.
- Le bloc instrumenté et remappé est rangé dans un cache.

PIN: exécution du code.

- PIN donne le contrôle au bloc instrumenté dans le cache.
- Le code d'origine (mais remappé) est exécuté.
- Le code d'analyse (remappé pour analyser les bons registres) est exécuté.
- PIN reprend le contrôle et passe au bloc de base suivant.
- Le bloc est cherché dans le cache: hit = transfert immédiat; miss = construction.

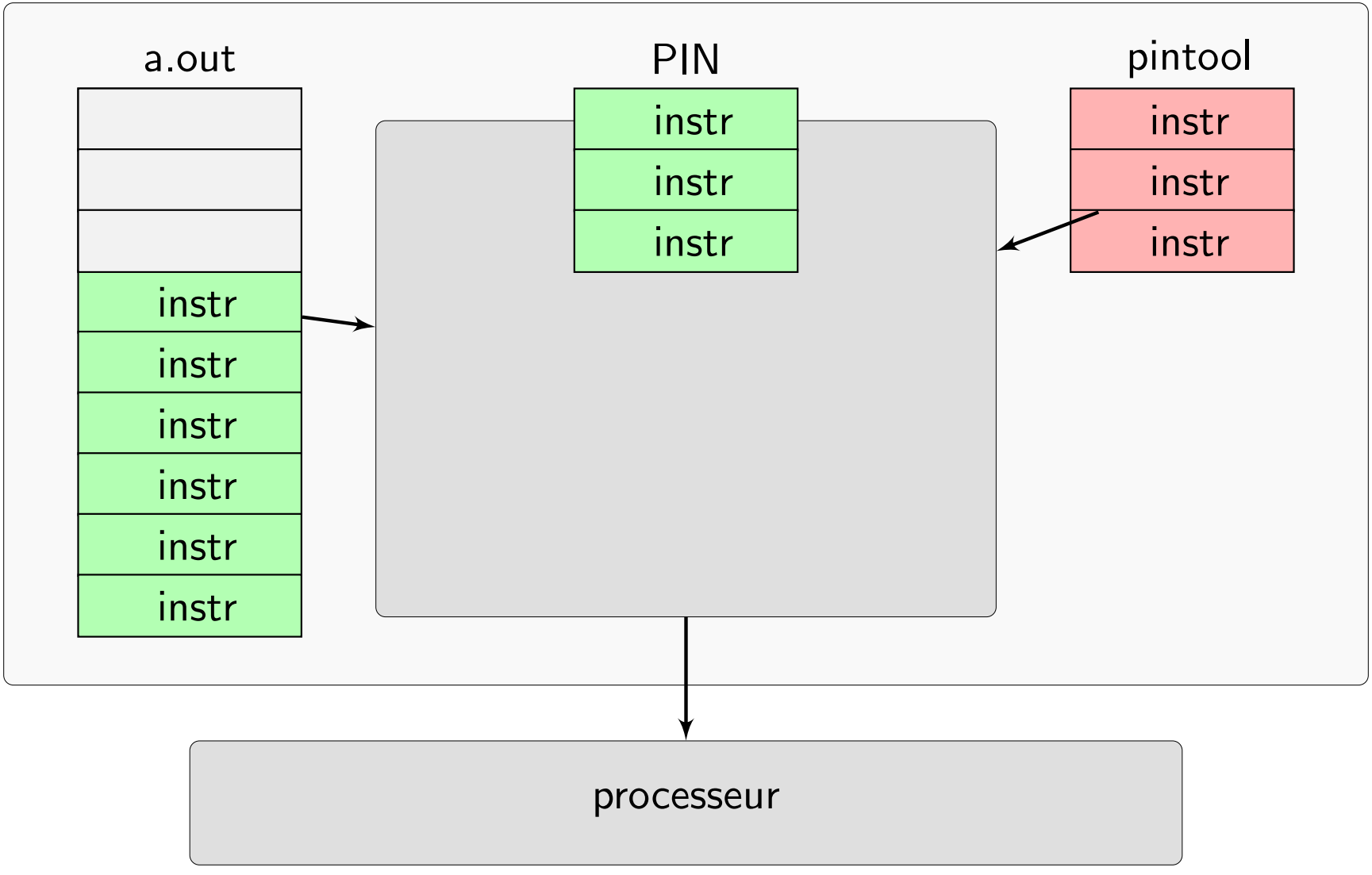
PIN sur un exemple.

```
$> pin -t pintool -- ./a.out
```



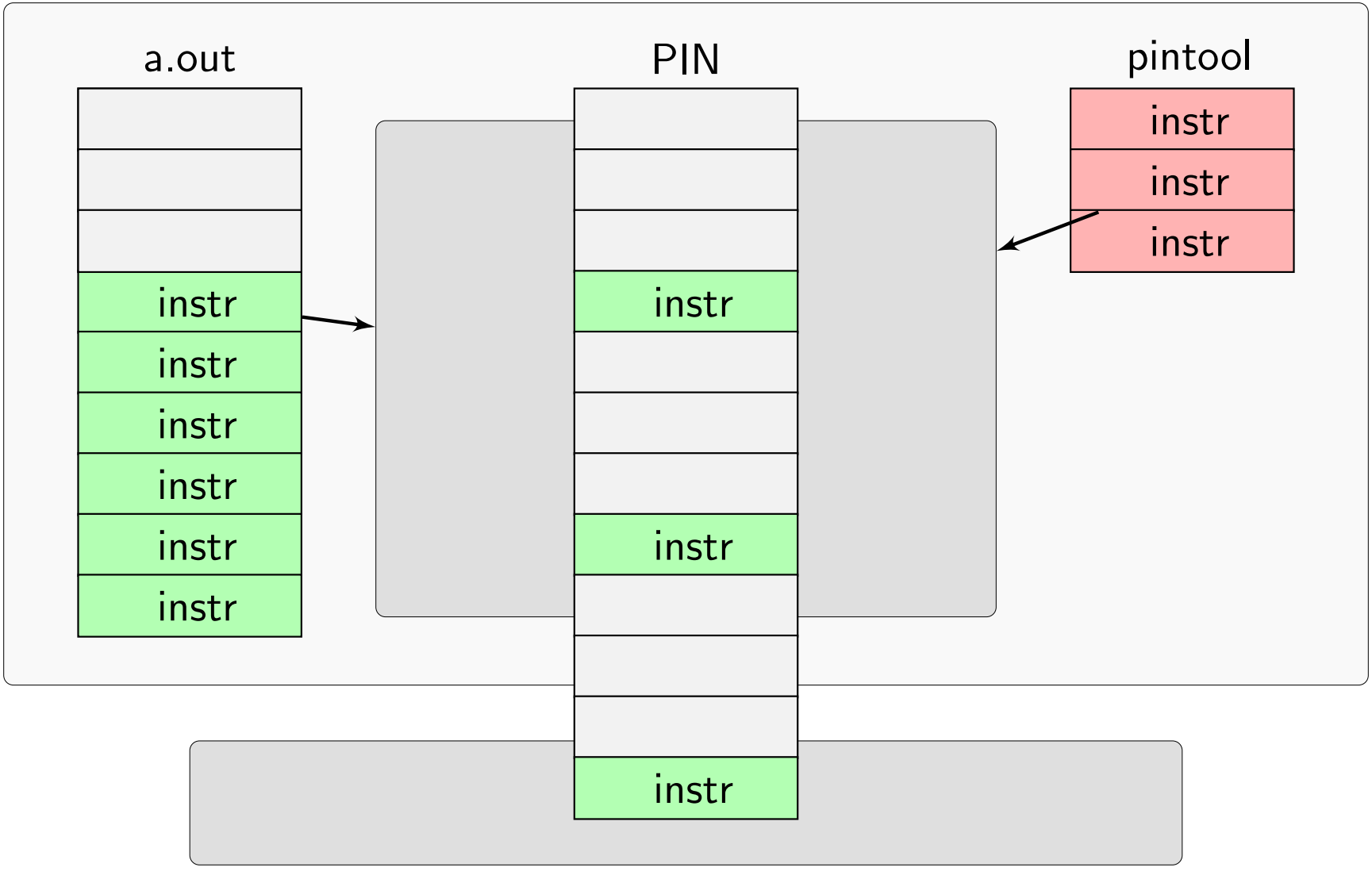
PIN sur un exemple.

```
$> pin -t pintool -- ./a.out
```



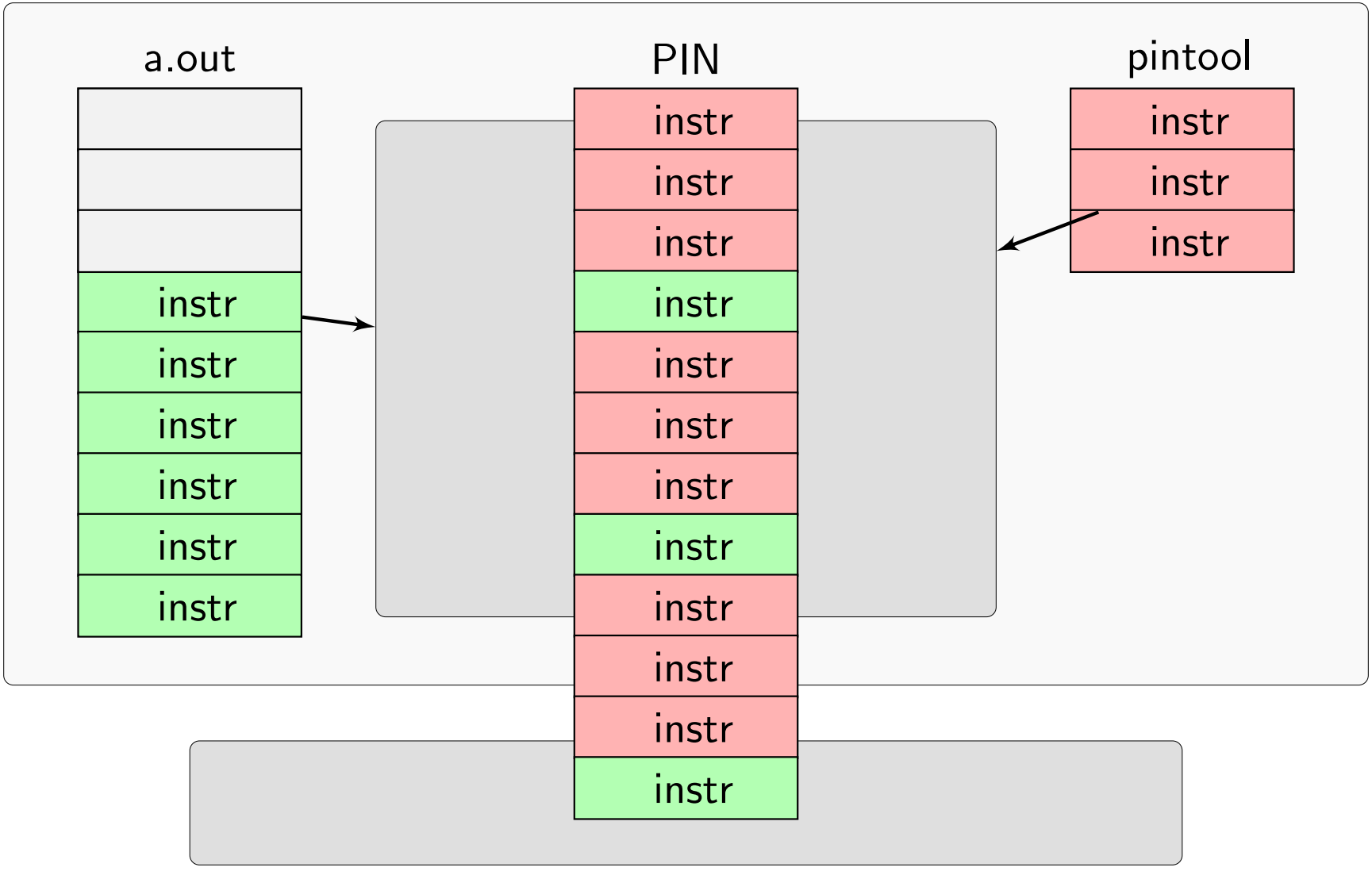
PIN sur un exemple.

```
$> pin -t pintool -- ./a.out
```



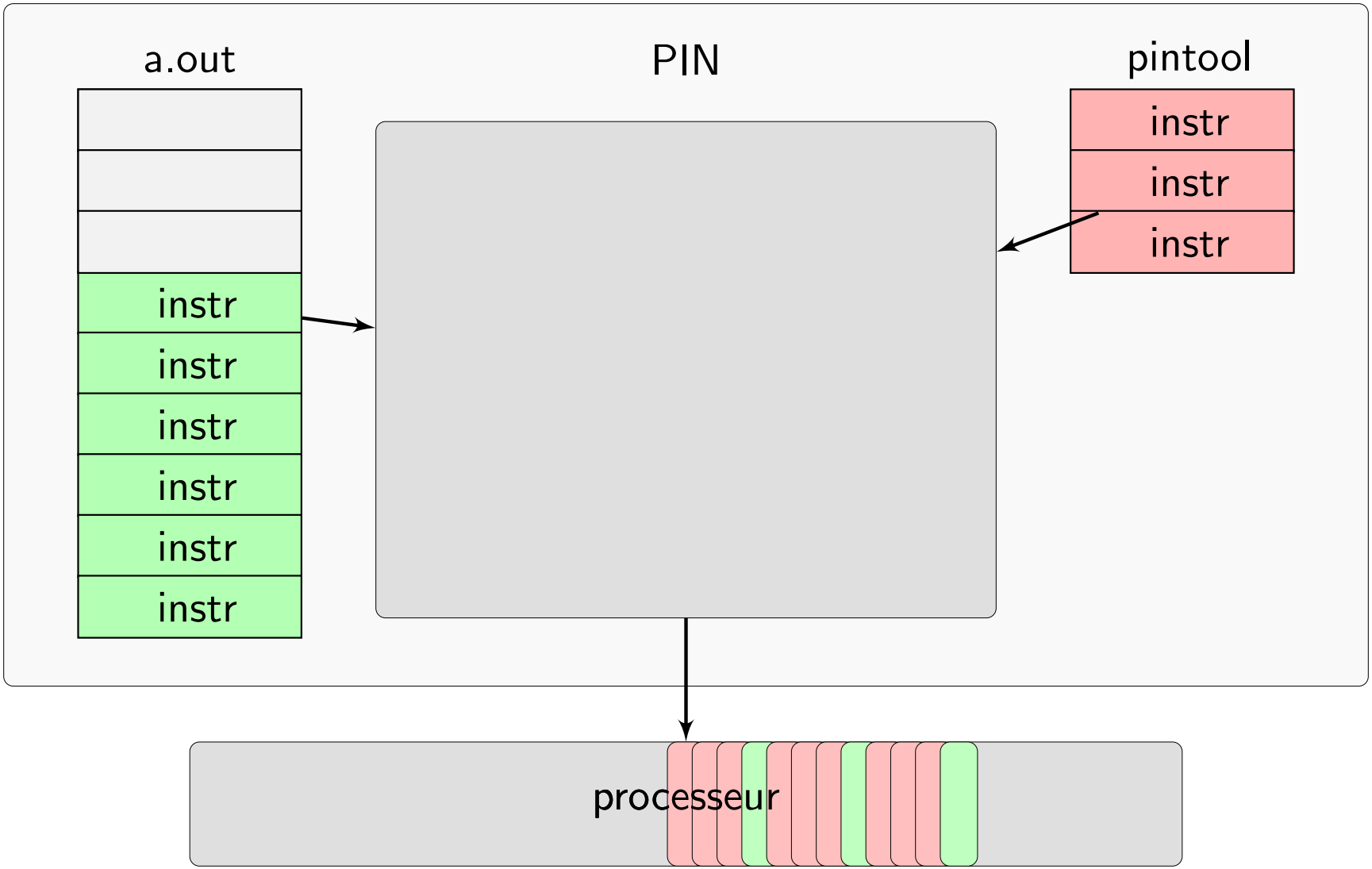
PIN sur un exemple.

```
$> pin -t pintool -- ./a.out
```



PIN sur un exemple.

```
$> pin -t pintool -- ./a.out
```



Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

	`...`
i1	`mov -20(ebp),eax`
i2	`add -16(ebp),eax`
i3	`mov -12(ebp),ecx`
i4	`add -8(ebp),ecx`
i5	`add ecx,eax`
i6	`mov eax,-4(ebp)`
	`...`

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

ni = 0

nc = 0

RC (registre cycle)

eax	:	0
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i1):  ebp
cycle(i1) = RC[ebp]+1 = 1
```

```
ni++; nc = max(nc, nc(i1))
```

ni = 1

nc = 1

RC (registre cycle)

eax	:	0
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i1):  ebp
cycle(i1) = RC[ebp]+1 = 1
destination(i1):  eax
RC[eax] = cycle(i1) = 1
```

RC (registre cycle)

eax	:	1
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

ni = 1

nc = 1

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

ni = 1

nc = 1

RC (registre cycle)

eax	:	1
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i2):  ebp, eax
cycle(i2)=max(RC[ebp],RC[eax])+1=2
```

```
ni++; nc = max(nc, nc(i2))
```

ni = 2

nc = 2

RC (registre cycle)

eax	:	1
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i2):  ebp, eax
cycle(i2)=max(RC[ebp],RC[eax])+1=2
destination(i2):  eax
RC[eax] = cycle(i2) = 2
```

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

ni = 2

nc = 2

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
    ...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
    ...
```

Routine analyse

ni = 2

nc = 2

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i3):  ebp, ecx
cycle(i3)=max(RC[ebp],RC[ecx])+1=1
```

```
ni++; nc = max(nc, nc(i3))
```

ni = 3

nc = 2

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	0
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i3):  ebp, ecx
cycle(i3)=max(RC[ebp],RC[ecx])+1=1
destination(i3):  ecx
RC[ecx] = cycle(i3) = 1
```

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	1
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

ni = 3

nc = 2

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

ni = 3

nc = 2

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	1
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i4):  ebp, ecx
cycle(i4)=max(RC[ebp],RC[ecx])+1=2
```

```
ni++; nc = max(nc, nc(i4))
```

ni = 4

nc = 2

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	1
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i4):  ebp, ecx
cycle(i4)=max(RC[ebp],RC[ecx])+1=2
destination(i4):  ecx
RC[ecx] = cycle(i4) = 2
```

ni = 4

nc = 2

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
    ...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
    ...
```

Routine analyse

ni = 4

nc = 2

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i5):  eax, ecx
cycle(i5)=max(RC[eax],RC[ecx])+1=3
```

```
ni++; nc = max(nc, nc(i5))
```

ni = 5

nc = 3

RC (registre cycle)

eax	:	2
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i5):  eax, ecx
cycle(i5)=max(RC[eax],RC[ecx])+1=3
destination(i5):  eax
RC[eax] = cycle(i5) = 3
```

RC (registre cycle)

eax	:	3
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

ni = 5

nc = 3

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
    ...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
    ...
```

Routine analyse

ni = 5

nc = 3

RC (registre cycle)

eax	:	3
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
...
```

Routine analyse

```
source(i6):  eax, ebp
cycle(i6)=max(RC[eax],RC[ebp])+1=4
```

```
ni++; nc = max(nc, nc(i6))
```

ni = 6

nc = 4

RC (registre cycle)

eax	:	3
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

	...
a1	call analyse
i1	mov -20(ebp),eax
a2	call analyse
i2	add -16(ebp),eax
a3	call analyse
i3	mov -12(ebp),ecx
a4	call analyse
i4	add -8(ebp),ecx
a5	call analyse
i5	add ecx,eax
a6	call analyse
i6	mov eax,-4(ebp)
	...

Routine analyse

```
source(i6):  eax, ebp
cycle(i6)=max(RC[eax],RC[ebp])+1=4
destination(i6):  mem[ebp-4]
RC[mem[ebp-4]] = cycle(i6) = 4
```

ni = 6

nc = 4

RC (registre cycle)

eax	:	3
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

```
    ...
a1  call   analyse
i1  mov    -20(ebp),eax
a2  call   analyse
i2  add    -16(ebp),eax
a3  call   analyse
i3  mov    -12(ebp),ecx
a4  call   analyse
i4  add    -8(ebp),ecx
a5  call   analyse
i5  add    ecx,eax
a6  call   analyse
i6  mov    eax,-4(ebp)
    ...
```

Routine analyse

ni = 6

nc = 4

RC (registre cycle)

eax	:	3
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

Comment PerPi calcule l'ILP?

e = (a+b) + (c+d)

code machine x86

	...
a1	call analyse
i1	mov -20(ebp),eax
a2	call analyse
i2	add -16(ebp),eax
a3	call analyse
i3	mov -12(ebp),ecx
a4	call analyse
i4	add -8(ebp),ecx
a5	call analyse
i5	add ecx,eax
a6	call analyse
i6	mov eax,-4(ebp)
	...

Routine analyse

ni = 6

nc = 4

ILP = 6/4 = 1,5

RC (registre cycle)

eax	:	3
ebx	:	0
ecx	:	2
edx	:	0
...	:	0
...	:	0
ebp	:	0
...	:	0

La routine d'analyse.

- Calcule le cycle d'exécution.
- Met à jour le cycle de disponibilité des destinations.
- Met à jour le cycle max.
- Ajoute l'instruction à sa place dans le tableau d'histogramme (si activé).
- Ajoute l'instruction à la trace désassemblée (si activé).
- Ajoute l'instruction au graphe de dépendances (si activé).
- Met à jour le sous-graphe de contrôle si nécessaire (si activé).

La routine de terminaison.

- Exécutée à la fin de l'exécution du code à mesurer.
- Calcule l'ILP et l'ILP de contrôle.
- Ecrit l'histogramme sur fichier (à relire avec gnuplot).
- Ecrit le graphe de dépendances sur fichier.
- Calcule et écrit sur fichier une plus longue chaine de dépendances (ldc).
- Calcule et écrit sur fichier une version synthétisée de la ldc.

PerPi: l'installation.

L'installation de gnuplot.

- Tout est sur la clé USB, répertoire PerPi-pour-Archi11.
- Copier PerPi-pour-Archi11 sur le disque dur, répertoire $HOME.
- Installer Gnuplot version 4.4.3 (mars 2011).

```
$> cd
$> cp -R /media/USB/PerPi-pour-Archi11 .
$> cd PerPi*/Gnuplot
$> tar -xzvf gnuplot*.gz
$> cd gnu*
$> ./configure
$> make
$> sudo make install
```

Essai de gnuplot.

```
$> cd $HOME/PerPi*/Demo
$> gnuplot
    G N U P L O T
    Version 4.4 patchlevel 2
    last modified ... Sep 22 ... 2010
    ...
Terminal type set to 'wxt'
gnuplot> quit
$>
```

L'installation de pin.

- Installer Pin.

```
$> cd
$> cd PerPi*/pin
$> tar -xzvf pin*.gz
$> cd pin*39599*/source/tools/Man*
$> make test
```

- Linux-64-bits:

```
$> ../../../pin -t obj-intel64/pinatrace.so -- /bin/ls
```

- Linux-32-bits:

```
$> ../../../pin -t obj-ia32/pinatrace.so -- /bin/ls
```

- Consulter le résultat:

```
$> head pinatrace.out
```

L'installation de PerPi.

- Installer PerPi.

```
$> cd $HOME/PerPi-pour-Archi11/Pilpgoo
$> make
```

Un essai.

```
$> cd ../Demo
$> gcc -o hello hello.c
$> vi _command
i
CONFIG
0
-1
-1
0
.
:wq
$> $HOME/pin*39599*/pin -t ../Pilpgoo/ilpnowrip -- ./hello
$> cat _profile
ILP of the full trace
I[1694]::C[68]::ILP[24.9118]...
$>
```

Dernière distrib Linux (11.04).

```
$> sudo su
# echo 0 > /proc/sys/kernel/yama/ptrace_scope
# exit
$>
```

Un essai vérifiable.

```
$> cat ma_boucle.s

main:    xorl    %eax,%eax        i=0
.L2:     movl    v(%eax), %edx    d=v[i]
         addl    u(%eax), %edx    d+=u[i]
         movl    %edx, t(%eax)    t[i]=d
         addl    $4, %eax         i+=4
         cmpl    $64, %eax        i==64?
         jne     .L2              si i!=64 vers .L2
         ret                      retour

$> gcc -o ma_boucle ma_boucle.s
$> $HOME/pin*39599*/pin -t ../Pilpgoo/ilpnowrip -- ./ma_boucle
$> cat _profile
ILP of the full trace
I[98]::C[19]::ILP[5.15789]...
$>
```

PerPi: un exemple vérifiable.

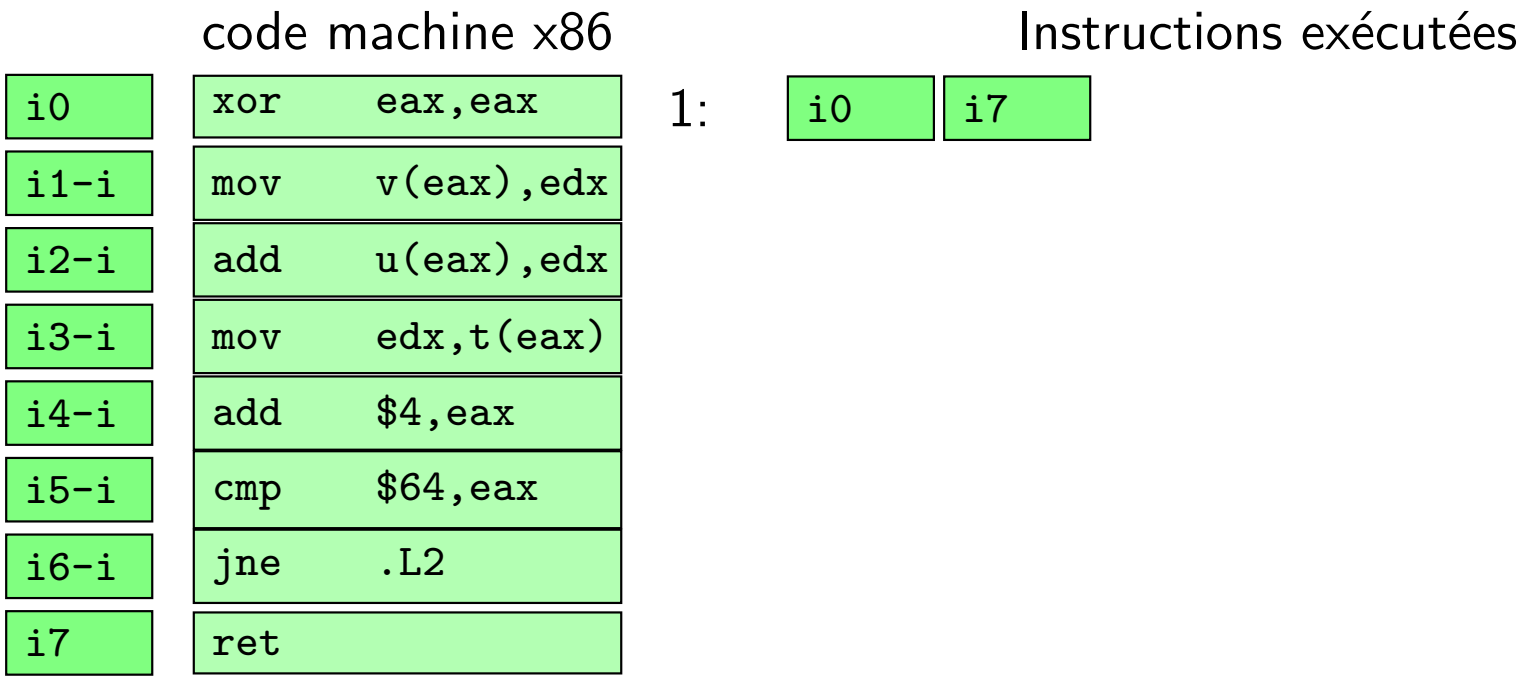
$1 + 16\text{x}6 + 1 = 98$ instructions

code machine x86

i0	`xor eax,eax`
i1-i	`mov v(eax),edx`
i2-i	`add u(eax),edx`
i3-i	`mov edx,t(eax)`
i4-i	`add $4,eax`
i5-i	`cmp $64,eax`
i6-i	`jne .L2`
i7	`ret`

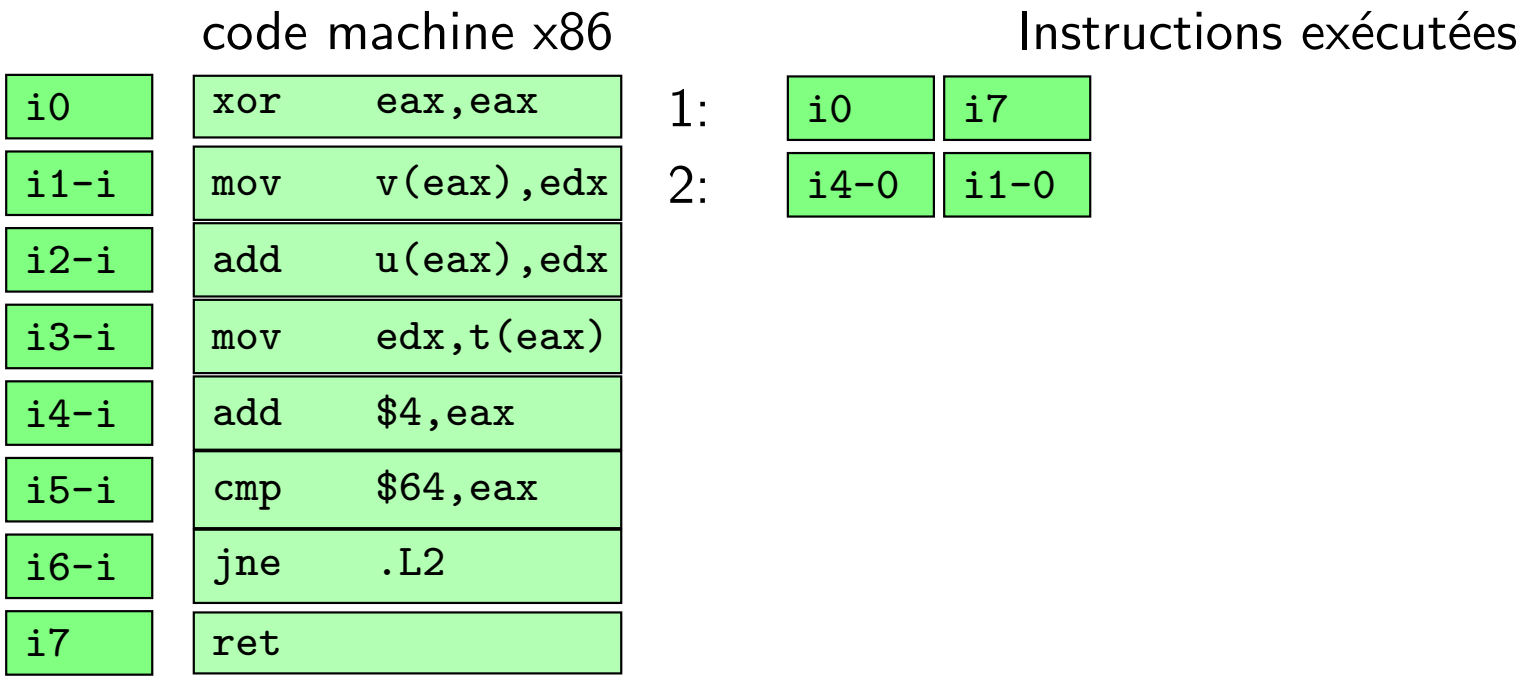
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



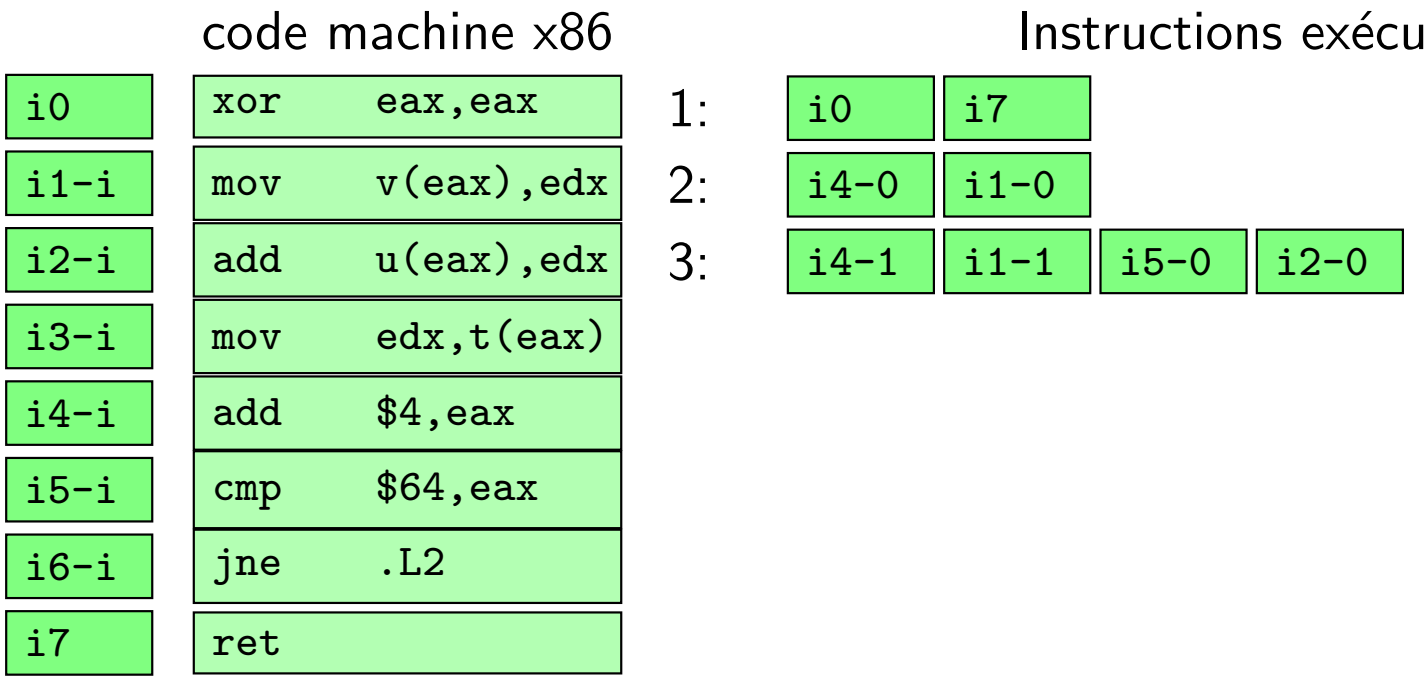
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



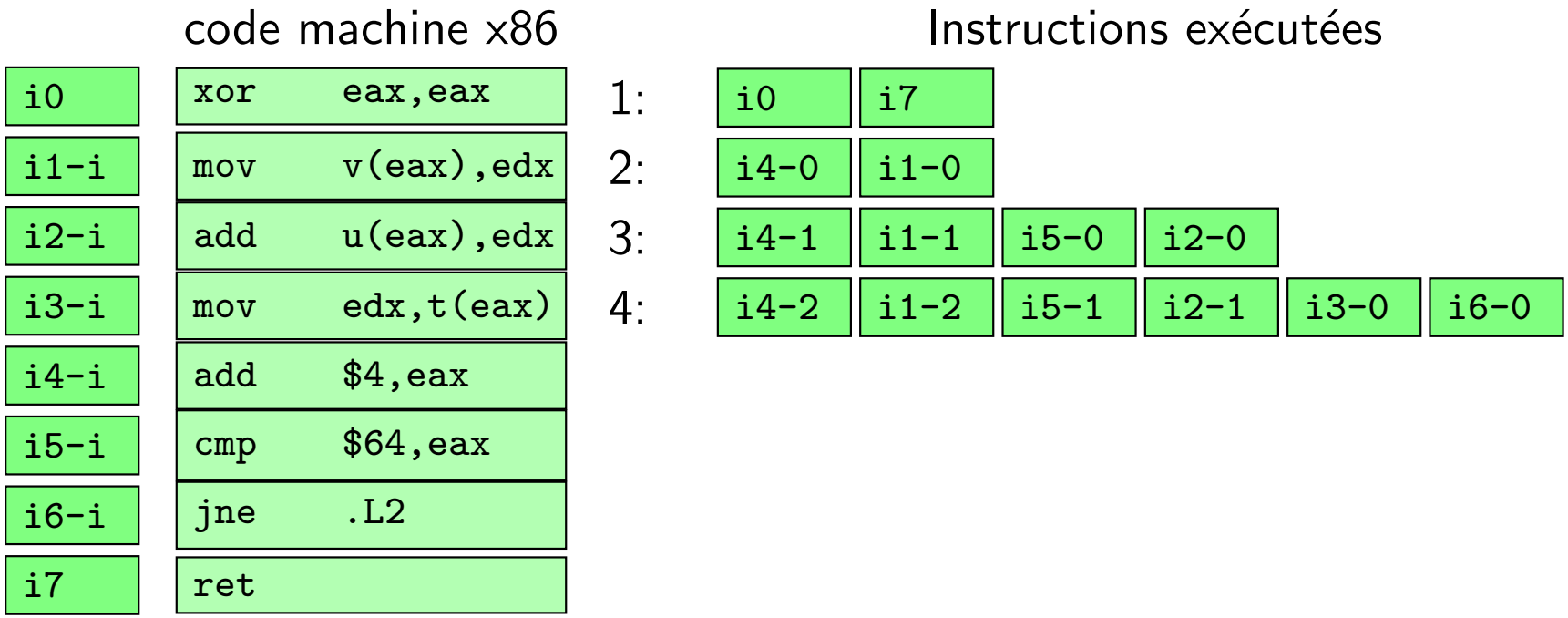
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



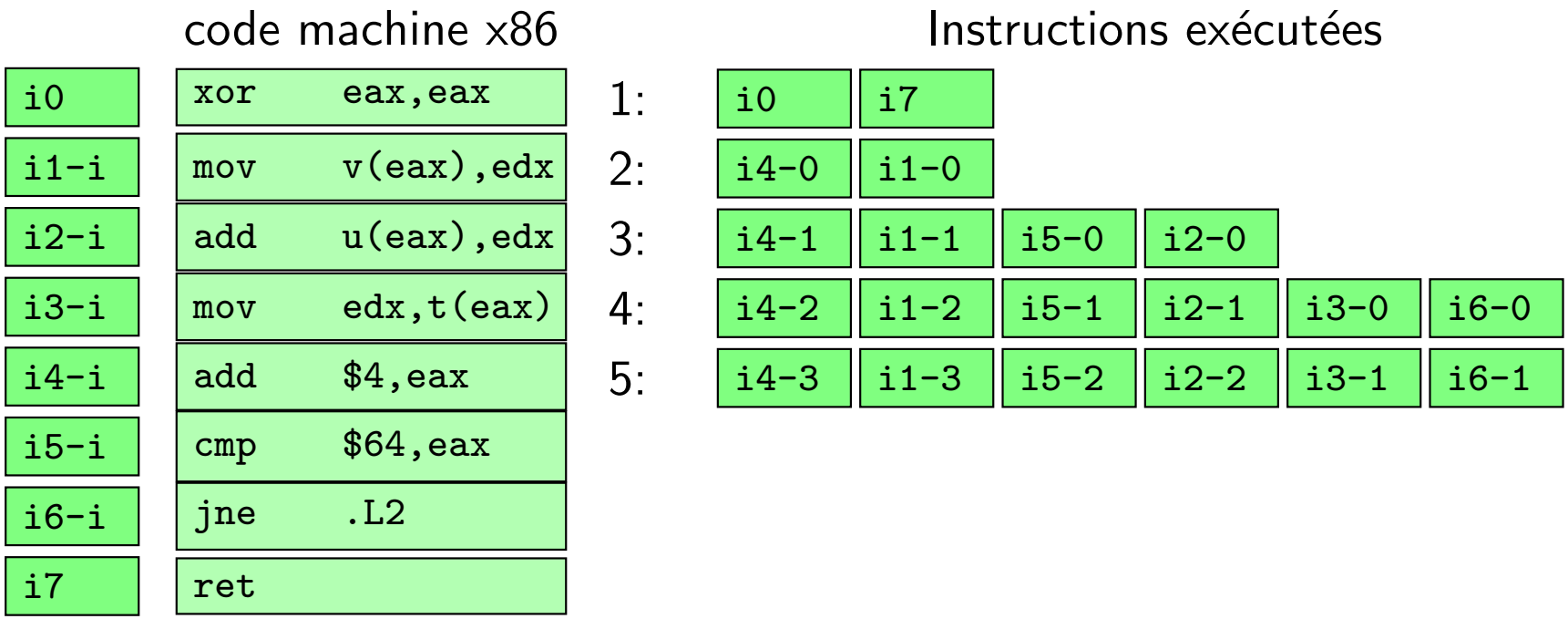
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



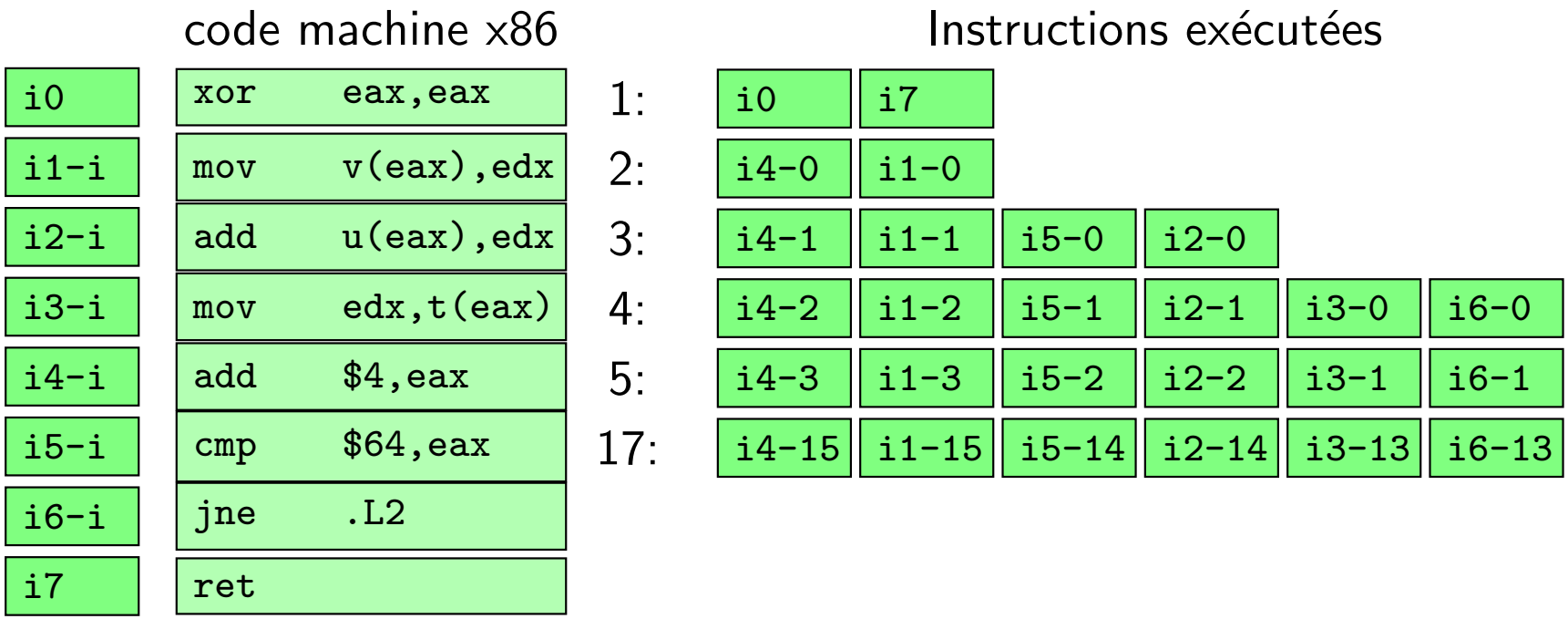
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



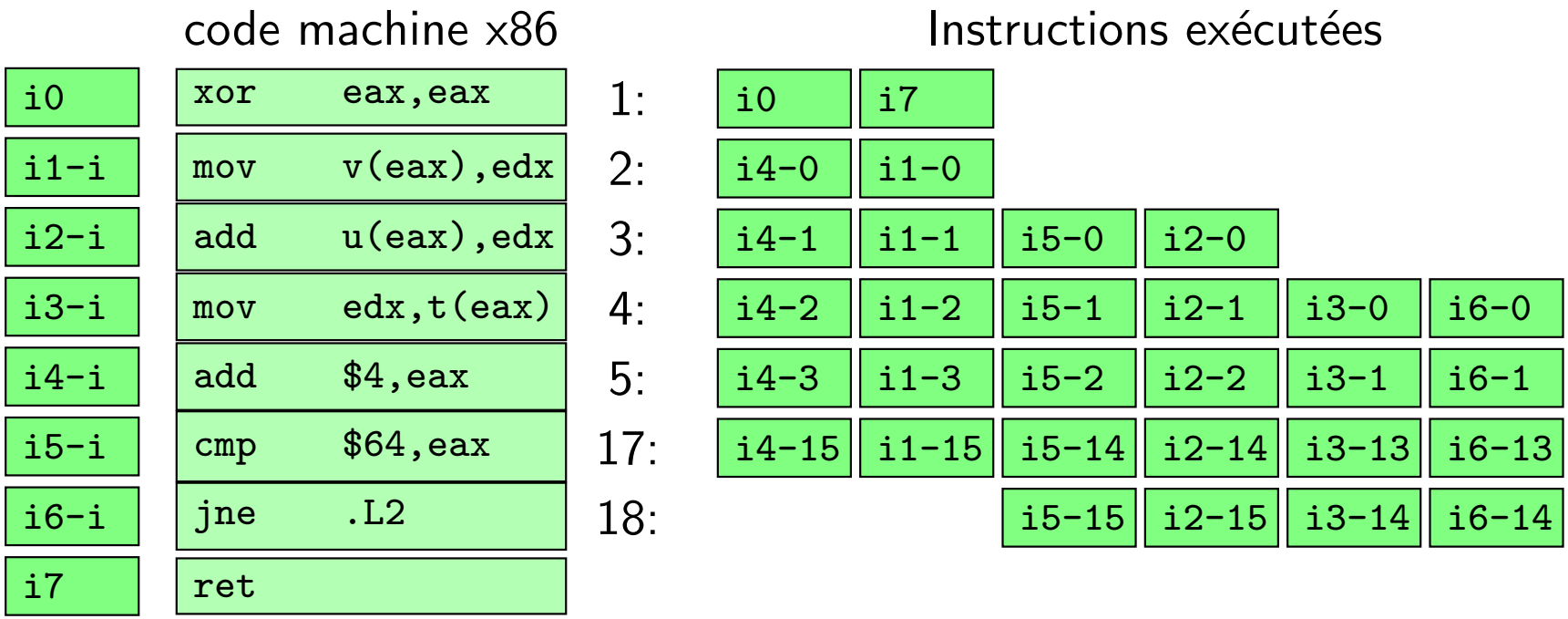
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



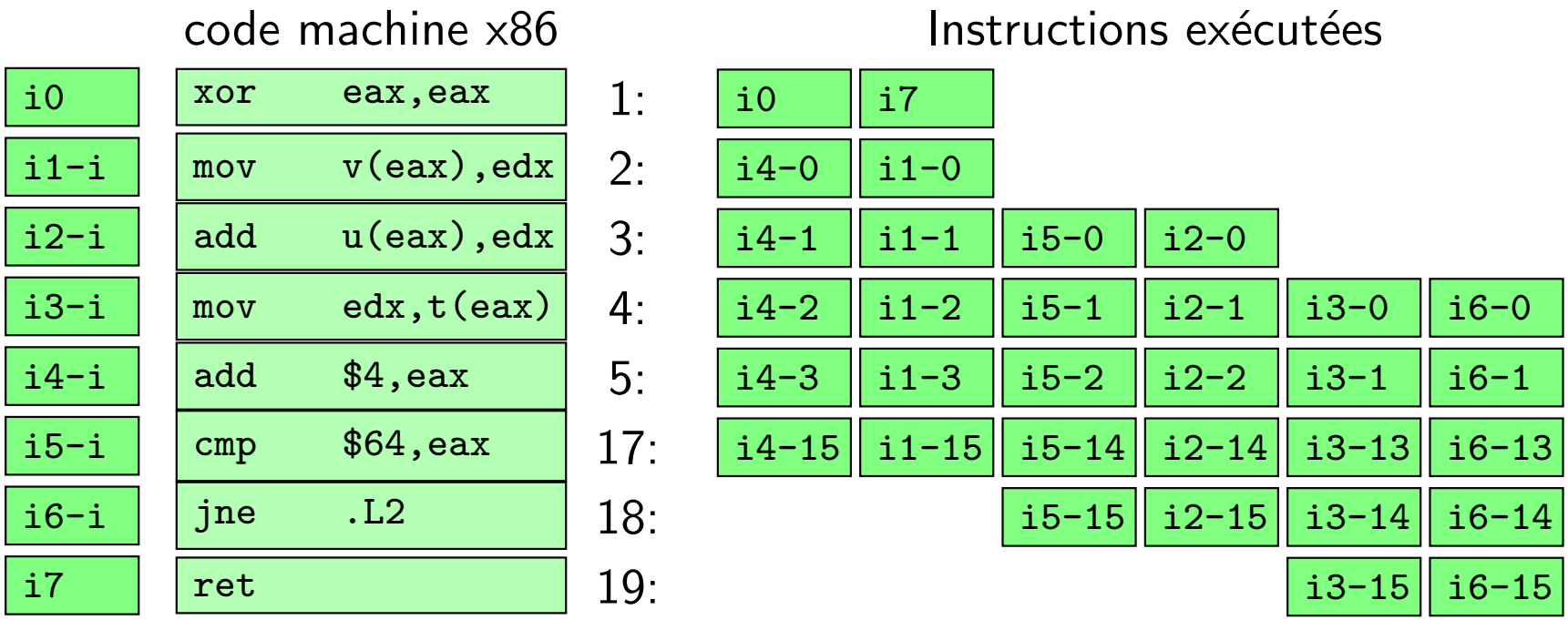
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



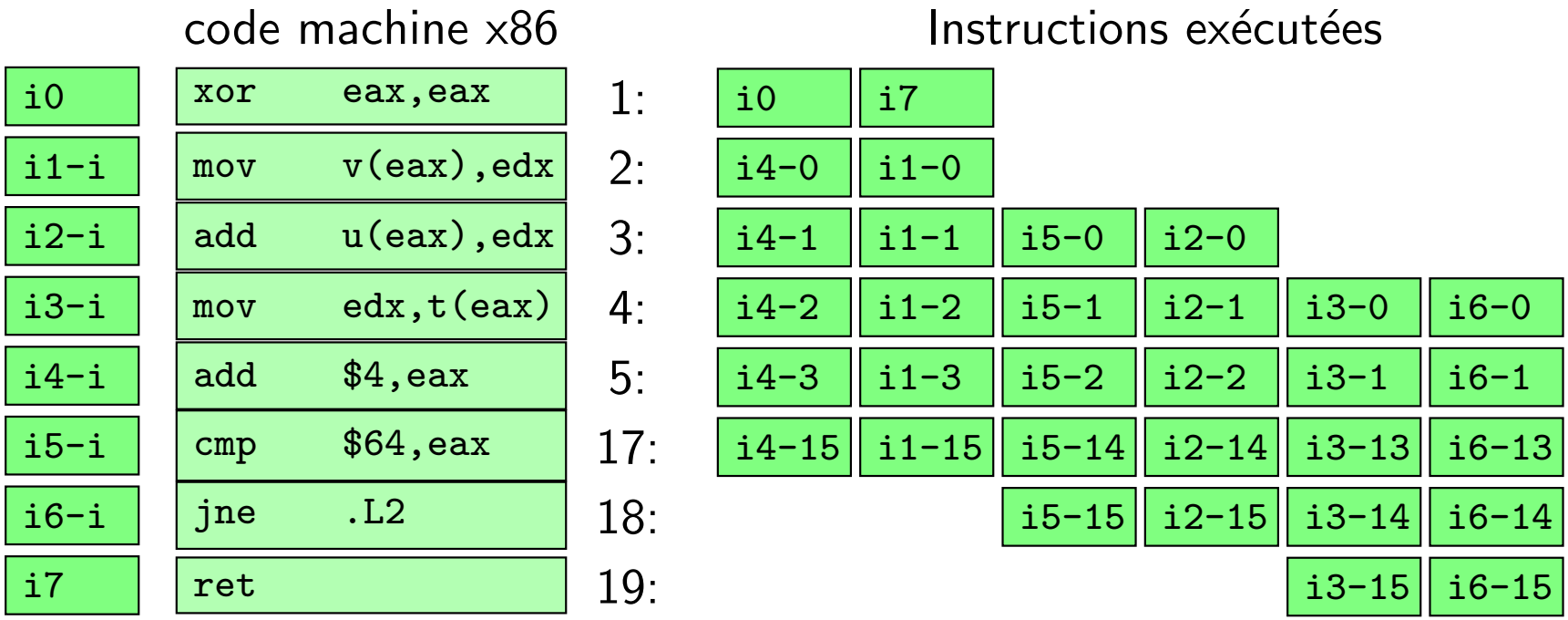
PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



PerPi: un exemple vérifiable.

$1 + 16x6 + 1 = 98$ instructions



19 cycles

PerPi: mesures reproductibles.

- Démarrage à main.
- Fin au retour de main.
- SYSCALL non mesurés.
- Fonctions de librairies mesurées.
- Pour un même jeu de données, toutes les exécutions ont le même nombre d'instructions.
- ILP reproductible (même distrib, même compilo).

PerPi: organisation du code.

La structure générale de PerPi.

- main dans main.cpp.
- Pin lance PerPi: main(argc,argv).
- argv: -- appli-à-mesurer [arguments].
- Lecture des options (option_read()) du fichier _command.

La structure générale de PerPi.

- Initialisation de la partie calcul de cycles (fichier _profile): profile_init().
- profile_init(): registres et mémoire prêts au cycle 0.
- Si histogramme activé, histo_init().
- histo_init(): allocation et initialisation de la table d'histogramme.
- Si traçage activé, trace_init().
- trace_init(): initialisation des registres et mémoire, partie producteur (instruction 0), et table d'instructions listant les dépendances (graphe en table vide).

La structure générale de PerPi.

- Appel d'API PIN.
- Initialiser pin: PIN_init.
- Enregistrer les fonctions d'activation/désactivation de l'analyse pour les SYSCALL: PIN_AddSyscall...Function.
- Enregistrer la fonction d'instrumentation: INS_AddInstrumentFunction(InstrumentInstruction), qui contrôle l'insertion de la fonction d'analyse AnalyseInstruction() dans le code à mesurer.
- Enregistrer la fonction de terminaison: PIN_AddFiniFunction(IntrumentFini), qui écrit les résultats et éventuellement, calcule une plus longue chaine de dépendances.
- Lancer la simulation (retour vers pin): PIN_StartProgram().

L'instrumentation de PerPi.

- Fonction InstrumentInstruction du fichier instrument.cpp.
- Appelée par pin à chaque chargement de bloc de base et pour chacune des instructions chargées.
- ins: structure pin contenant l'instruction chargée (infos statiques).
- Détecter le début main + préfixe et la fin main + préfixe + longueur de l'analyse (effet de bord sur la fonction AnalyseInstruction).
- API pin INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(AnalyseInstruction),...).
- Insérer avant l'instruction un appel à AnalyseInstruction(...).
- Plusieurs variantes de AnalyseInstruction: avec/sans lecture en mémoire, avec/sans écriture en mémoire.

L'analyse des instructions par PerPi.

- Fonction AnalyseInstruction du fichier instrument.cpp.
- Aucune analyse si en dehors de [début,fin] (mais exécution instrumentée avec appel à la fonction d'analyse et retour immédiat).
- Désactiver la trace si longueur atteinte (trace_length).
- Compter l'instruction (numinst++).
- Calculer le cycle (cexe) de la source la plus tardive (registre ou mémoire).
- Mettre à jour (à cexe) le cycle des destinations (registre ou mémoire).
- Mettre à jour le nombre de cycles de l'exécution si nécessaire (nbcycles).
- Si l'histogramme est actif, le compléter.
- Si le traçage est actif, ajouter l'instruction au fichier _trace et compléter le graphe en table operations.

La terminaison de PerPi.

- Fonction InstrumentFini du fichier instrument.cpp.
- Calcul de l'ilp et report dans le fichier _profile (profile_dump()).
- Si l'histogramme est actif, report dans les fichiers _histo (détail par type d'opcode) et _histo2 (global).
- Si le traçage est actif, report du graphe dans le fichier _graph et calcul d'une plus longue chaine de dépendances _ldc_instr, avec version synthétisée _ldc_synth.

Exemple d'investigation avec PerPi.

Création d'histogramme.

- Changer la troisième ligne de _command: remplacer -1 par 0.

```
$> cat _command
CONFIG
0
0
-1
0
.
$> $HOME/pin*39599*/pin -t ../Pilpgoo/ilpnowrip -- ./ma_boucle
```

Visualisation d'histogramme.

```
$> gnuplot
gnuplot> load 'gp.gpi'
gnuplot> plot for [i=2:6] '_histo' using i
```

Sauvegarde d'histogramme.

```
gnuplot> load 'eps.gpi'
gnuplot> plot for [i=2:6] '_histo' using i
gnuplot> quit
$>
```

Un vrai benchmark: cjpeg de Mibench.

- Construire l'histogramme complet de cjpeg (83Mi, 200Kc).

```
$> cd ../Mibench/consumer/jpeg/jpeg-6a
$> make
$> cp cjpeg ../input_large.ppm ../../../../Demo
$> cd ../../../../Demo
$> $HOME/pin*39599*/pin -t ../Pilpgoo/ilpnowrip
-- ./cjpeg -dct int -progressive -opt
-outfile output_large_encode.jpeg input_large.ppm
$> cat _profile
ILP of the full trace
I[83842673]::C[204983]::ILP[409.023]...
$> gnuplot
gnuplot> load 'gp.gpi'
gnuplot> plot for [i=2:22] '_histo' using i
gnuplot> quit
$>
```

Combien d'instructions au cycle 1?

```
$> head _histo2
         0        0
         1 3015401
         2  499193
         3  313821
         4  188720
         5  148713
         6  128783
         7  118263
         8  106490
         9  104074
$>
```

Quelle distribution?

- 3M d'instructions sur 84M (3,6%) au cycle 1.
- $<$ 10K IPC après le cycle 1800 (0,9% des cycles).
- $<$ 1K IPC après le cycle 8400 (4,1% des cycles).
- $<$ 409 (ILP) IPC après le cycle 8460 (4,1% des cycles).
- $<$ 100 IPC après le cycle 8467 (4,1% des cycles).
- $<$ 10 IPC après le cycle 46800 (23% des cycles).
- $<$ 5 IPC après le cycle 100000 (50% des cycles).

Zoom sur la descente.

```
$> gnuplot
gnuplot> load 'gp.gpi'
gnuplot> set xrange[2:10000]
gnuplot> set yrange[0:500000]
gnuplot> plot for [i=2:22] '_histo' using i
```

Zoom sur la descente 2.

```
gnuplot> set xrange[10:10000]
gnuplot> set yrange[0:100000]
gnuplot> plot for [i=2:22] '_histo' using i
```

Zoom sur la descente 3.

```
gnuplot> set xrange[8000:50000]
gnuplot> set yrange[0:1500]
gnuplot> plot for [i=2:22] '_histo' using i
gnuplot> quit
$>
```

L'ILP n'est pas uniformément réparti.

- Pic au cycle 1.
- Chute au cycle 2.
- Nouvelle chute au cycle 3.
- Amas dans les 5% premiers cycles.
- Queue de 75% des cycles avec un ILP $<$ à 10.

Est-ce propre à cjpeg?

- Quelles instructions au cycle 1?
- Quelles instructions au cycle 2?
- D'où viennent les instructions de l'amas?
- D'où viennent les instructions de la queue?
- Histogramme inadapté.
- Plus longue chaine de dépendances.

Qu'est-ce qu'une plus longue chaine de dépendances (ldc)?

- C cycles => C instructions en chaine, du cycle 1 au cycle C.
- Instruction au cycle c: un (au moins) prédécesseur au cycle $c-1$ (pas de trou dans la chaine).
- Graphe de dépendances: tableau d'instructions liées.
- Dernière instruction de la trace exécutée au dernier cycle = fin de ldc.
- Liste des prédécesseurs.
- Premier prédécesseur exécuté au cycle précédent = précédent de chaine.

Paramètres du fichier de configuration.

- Fichier _command.
- Premier paramètre: #instructions analysées (IA).
- 0: sans limite, n>0: max 2G.
- Second paramètre: #cycles de l'histogramme (CH).
- -1: pas d'histogramme, 0: IA (max 2G), n>0: max IA.
- Troisième paramètre: #instructions graphe de dépendances (IG).
- -1: pas de graphe, 0: 40M, n>0: max inf(IA, 40M).
- Quatrième paramètre: #instructions préfixe non analysé (IP).
- 0: analyse dès main, n>0: après main + n.

Création de graphe.

- Changer la quatrième ligne de _command: remplacer -1 par 0.

```
$> cat _command
CONFIG
0
0
0
0
.
$> $HOME/pin*39599*/pin -t ../Pilpgoo/ilpnowrip -- ./ma_boucle
```

Visualisation de la trace de ma_boucle.

```
$> head _trace
      1  main            4004b4  xor eax, eax
      2  main            4004b6  mov edx, dword ptr [eax+0x6010c0]
      3  main            4004bd  add edx, dword ptr [eax+0x601080]
...
$>
```

Visualisation de la ldc de ma_boucle.

```
$> cat _ldc_instr
      1  main            4004b4  xor eax, eax
      5  main            4004cb  add eax, 0x4
     11  main            4004cb  add eax, 0x4
...
     95  main            4004cb  add eax, 0x4
     96  main            4004ce  cmp eax, 0x40
     97  main            4004d1  jnz 0x4004b6
$>
```

Exploiter les ldc simples: ma_boucle.

- Ldc à motif périodique: boucle.
- Outil _ldc_synth pour une vision synthétique de la ldc.

```
$> cat _ldc_synth
A[1(t16/i16)2].
$>
```

- Fonction A (voir _flist: main).
- Boucle à 16 itérations, 16 instructions dans ldc (1 par itération).
- 1 instruction avant, 2 instructions après = ldc de 19 instructions.

Exploiter les ldc complexes: cjpeg.

- Ldc sans motif périodique: paramètres/résultats de fonctions.

```
$> $HOME/pin*39599*/pin -t ../Pilpgoo/ilpnowrip
-- ./cjpeg -dct int -progressive -opt
-outfile output_large_encode.jpeg input_large.ppm
$> cat _profile
ILP of the trace segment
I[40000000]::C[61985]::ILP[645.317]...
$> wc -l _ldc_instr
61985 _ldc_instr
$> head _ldc_synth
A[(t1/i1)(t2/i1)2(t1/i1)...G[2]
...
$>
```

- Enchainements d'appels/retours.
- Fonctions O et K répétées: boucle englobant les appels?

A savoir sur les ldc.

- Ldc à motif périodique: boucle.
- Ldc sans motif périodique: paramètres/résultats de fonctions.
- Un segment de taille n peut avoir une ldc totalement différente de celle d'un segment de taille $m > n$.
- Tous les Mibench ont des ldc de type boucle, sauf cjpeg et djpeg.

Un autre benchmark: rawcaudio de Mibench.

- Rawcaudio: type boucle (655829045i, 79860244c, 8.2ilp).
- Long: se limiter à 10Mi, 10Mc (modifier _command).

```
$> cd ../Mibench/telecomm/adpcm/src
$> make
$> cp rawcaudio ../data/large.pcm ../../../../Demo
$> cd ../../../../Demo
$> $HOME/pin*39599*/pin -t ../Pilpgoo/ilpnowrip
-- ./rawcaudio < large.pcm > output_large.adpcm
```

Ldc à motif périodique.

- Ldc: boucle de 6 instructions.

```
$> more _ldc_instr
...
   130  adpcm_co        400635  add r8d, r9d
   132  adpcm_co        40063a  cmovz edx, r8d
   134  adpcm_co        400641  cmp edx, 0xffff8000
   135  adpcm_co        400647  cmovnl r9d, edx
   136  adpcm_co        40064b  cmp r9d, 0x7fff
   137  adpcm_co        400652  cmovnle r9d, r13d
...
$>
$> cat _ldc_synth
A[10((t5994/i999)2(t6000/i1000)2(t6000/i1000)...2].
$>
```

- Double boucle, boucle interne à 6 instructions, 1000 tours.

Ajuster PerPi à ses besoins: un exemple.

Peut-on mieux répartir l'ILP?

- 25% parallèle, 75% séquentiel: pas de processeur d'ILP possible.
- Première idée: replier les contrôles de boucle.
- $i++$, $i--$, $i+=k$, $i-=k$: renommage spécial (idem RSP).
- Chaque $i++$ dépend du $i=$ initial et d'une constante calculée au renommage.

Amélioration de l'ILP avec le repliage du contrôle de boucle.

- Ajouter la constante IS_NEW_LOOP_COUNTER_SEMANTIC_ENABLED.
- Inhiber mise à jour destination $i+=k$ et $i-=k$ (i reg ou mém).

Nom	ILP base	ILP i+=k/i-=k
basicmath	10.26	12.14
bitcnts	38.66	41.42
cjpeg	35.19	41.15
crc	11.99	11.99
dijkstra	120.29	186.86
djpeg	36.07	42.83
qsort	18.97	22.41
rawcaudio	8.20	8.20
rawdaudio	7.04	7.04
search	26.06	30.94
susanc	2257.16	2761.70
susane	47.86	47.93
susans	2899.02	49165.80

Peut-on mieux répartir l'ILP?

- Seconde idée: paralléliser les appels de fonction.
- $RSP+=k$, $RSP-=k$, $RBP+=k$, $RBP-=k$: renommage spécial.
- Chaque $RSP+=k$ dépend du $RSP=$ initial et d'une constante calculée au renommage.

Amélioration de l'ILP avec la parallélisation des fonctions.

- Ajouter la constante IS_STACK_RENAMED.
- Inhiber la mise à jour des registres RSP et RBP.

Nom	ILP base	ILP RSP/RBP
basicmath	10.26	32.20
bitcnts	38.66	193.31
cjpeg	35.19	409.04
crc	11.99	12.07
dijkstra	120.29	639.37
djpeg	36.07	71.60
qsort	18.97	184.54
rawcaudio	8.20	8.20
rawdaudio	7.04	7.04
search	26.06	65.34
susanc	2257.16	6544.93
susane	47.86	48.12
susans	2899.02	3076.77

Peut-on mieux répartir l'ILP?

- Troisième idée: replier les transmissions.
- MOV: renommage spécial.
- La destination pointe sur la source: elle dépend des producteurs de la source.

Amélioration de l'ILP avec le repliage des transmissions.

- Ajouter la constante IS_NEW_MOV_SEMANTIC_ENABLED.
- Modifier le calcul de cexe pour les MOV: cexe = cycle d'exécution source du mov.

Nom	ILP base	ILP MOV+RSP/RBP
basicmath	10.26	250.75
bitcnts	38.66	580.00
cjpeg	35.19	9952.05
crc	11.99	16.01
dijkstra	120.29	701.19
djpeg	36.07	8320.48
qsort	18.97	600.32
rawcaudio	8.20	109114.00
rawdaudio	7.04	93588.60
search	26.06	440.35
susanc	2257.16	6701.75
susane	47.86	544.72
susans	2899.02	406935.00

Amélioration de l'ILP: tout cumulé.

Nom	ILP base	ILP RSP	ILP MOV+RSP	ILP i+=k/i-=k	ILP tout
basicmath	10.26	32.20	250.75	12.14	287.26
bitcnts	38.66	193.31	580.00	41.42	580.00
cjpeg	35.19	409.04	9952.05	41.15	20360.90
crc	11.99	12.07	16.01	11.99	16.01
dijkstra	120.29	639.37	701.19	186.86	701.19
djpeg	36.07	71.60	8320.48	42.83	8320.48
qsort	18.97	184.54	600.32	22.41	600.32
rawcaudio	8.20	8.20	109114.00	8.20	109114.00
rawdaudio	7.04	7.04	93588.60	7.04	93588.60
search	26.06	65.34	440.25	30.94	440.25
susanc	2257.16	6544.93	6701.75	2761.70	6701.75
susane	47.86	48.12	544.72	47.93	6879.26
susans	2899.02	3076.77	406935.00	49165.80	660589.00

Le fichier const_config.hh.

- Quatre booléens d'activation/désactivation.
- IS_FETCH_ENABLED: activer/désactiver une contrainte de fetch.
- IS_NEW_MOV_SEMANTIC_ENABLED: MOV spéciaux.
- IS_NEW_LOOP_COUNTER_SEMANTIC_ENABLED: i+=k, i-=k spéciaux.
- IS_STACK_RENAMED: RBP/RSP spéciaux.
- Ajoutez vos booléens, désactivez/activez les miens, ajustez instrument.cpp.
- Penser à recompiler après une modif sur ce fichier!!

Deux remarques concernant les futures machines à ILP.

- 1) Crc récalcitrant (checksum: ldc = récurrence non repliable).
- Solution: réécrire le programme avec une méthode en arbre binaire.
- 2) Application de la contrainte de fetch => pas d'ILP.
- Contrainte: une instruction ne peut être exécutée avant d'avoir été extraite.
- Une instruction est extraite en même temps que celles de la même ligne de cache.
- Une nouvelle ligne de cache est extraite au cycle suivant.

CRC de base.

- ILP: 4.12.

```
#include <stdio.h>
#define SIZE (1<<16)
unsigned int t[SIZE];
unsigned int sum(unsigned int t[], unsigned int n);
main()
 unsigned int i, s=0;
 for (i=0;i<SIZE;i++) t[i]=random();
 s=sum(t,SIZE);
 printf("sum: %u\n",s);
 return(0);

unsigned int sum(unsigned int t[], unsigned int n)
 unsigned int i, s=0;
 for (i=0;i<n;i++) s+=t[i];
 return s;
```

CRC amélioré (arbre binaire itératif).

- ILP: 1608.79.

```
#include <stdio.h>
#define LSIZE 16
#define SIZE (1<<LSIZE)
unsigned int t[SIZE];
unsigned int sum(unsigned int t[], unsigned int n);
main()
 unsigned int i, s=0;
 for (i=0;i<SIZE;i++) t[i]=random();
 s=sum(t,SIZE);
 printf("sum: %u\n",s);
 return(0);

unsigned int sum(unsigned int t[], unsigned int n)
 unsigned int i, j, l;
 for (i=0;i<LSIZE;i++)
  l=(1<<(LSIZE-i-1));
  for (j=0;j<l;j++)
   t[j]+=t[j+l];
```

ILP avec contrainte de fetch.

Nom	ILP avec fetch	ILP sans fetch
basicmath	5.53	287.26
bitcnts	13.18	580.00
cjpeg	5.01	20360.90
crc	9.59	16.01
dijkstra	5.41	701.19
djpeg	7.99	8320.48
qsort	5.29	600.32
rawcaudio	4.92	109114.00
rawdaudio	8.43	93588.60
search	9.08	440.25
susanc	6.80	6701.75
susane	6.26	6879.26
susans	6.83	660589.00

Conclusion.

Conclusion 1.

- PerPi est un simulateur d'un nouveau genre.
- Il n'implémente aucune architecture: c'est pin (qui évolue avec le x86).
- Il n'implémente aucune microarchitecture: pas de pipeline, pas de caches, pas de prédicteur de sauts, pas de registres, pas de mémoire.
- Il suit la trace (fournie par pin) et applique à chaque instruction tour à tour un calcul (de cycle d'exécution, de consommation, de ce que vous voulez).
- On peut compléter PerPi pour affiner le calcul fait sur chaque instruction en ajoutant un modèle de contraintes implémentant une microarchitecture réelle (exemple: contrainte de fetch).

Conclusion 2.

- PerPi est maintenant votre outil.
- Utilisez-le tel quel pour faire de la recherche sur l'ILP.
- Complétez-le pour activer un mode réel (contraintes microarchitecturales d'un processeur actuel).
- Modifiez le calcul pour remplacer l'ilp par ce qui vous amuse.