

An Overview of the intel®IA-64 Compiler

yoann AUDOUIN, long QU

22 janvier 2010

- 1 Introduction
- 2 Architecture
- 3 Optimisation
- 4 Optimisation mémoire
- 5 Optimisation scalaire
- 6 Générateur du code
- 7 Conclusion

Article

- Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, David Sehr
- Microcomputer Software Laboratory, Intel Corporation

Plan

- 1 Introduction
- 2 Architecture
- 3 Optimisation
- 4 Optimisation mémoire
- 5 Optimisation scalaire
- 6 Générateur du code
- 7 Conclusion

Description

- Architecture optimisé
- Optimisation des programmes
- De nombreuses nouvelles méthodes
- Base de registre importante

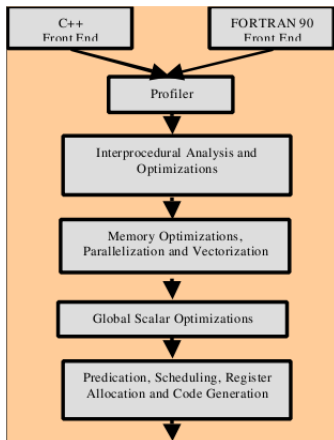
Buts

- Diminuer les temps d'accès mémoire
- Diminuer les temps de branchement
- Maximiser les opérations en parallèle

Plan

- 1 Introduction
- 2 Architecture**
- 3 Optimisation
- 4 Optimisation mémoire
- 5 Optimisation scalaire
- 6 Générateur du code
- 7 Conclusion

Architecture du compilateur



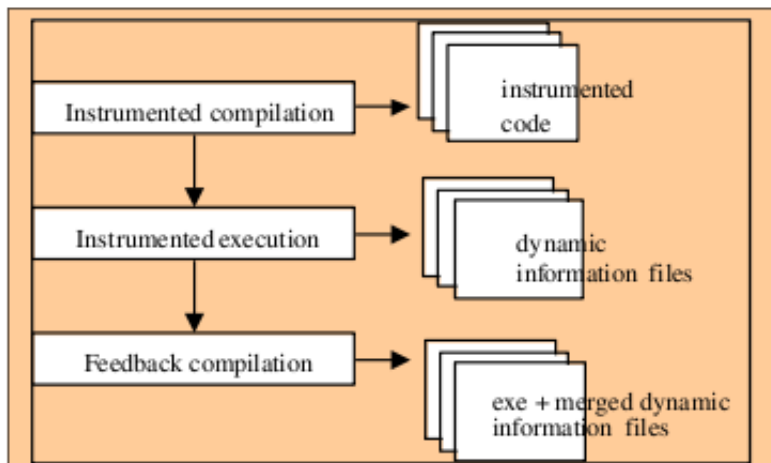
Plan

- 1 Introduction
- 2 Architecture
- 3 Optimisation**
 - Optimisation avec profile
 - Optimisation par analyse interprocedurale
- 4 Optimisation mémoire
- 5 Optimisation scalaire
- 6 Générateur du code

Principe

- Probabilité de prendre une branche
- Fréquence d'utilisation de bloc de code
- Deux types
 - Statique
 - Dynamique

Fonctionnement



Interets

- Re-positionnement des codes souvent effectués
- Réduction de l'utilisation du cache et échec TLB
- Meilleur ordonnancement des taches

Principe

- EPIC (Explicitly Parallel Instruction Computing)
- Plusieurs opérations par temps d'horloge
- Besoins de renseignement sur la memoire

Les techniques employées

- Analyse point à point
- Analyse mod/ref
- Propagation des effets de bord
- Propagation constante

Analyse point à point

```
1 r32=[r33 ]  
  [r37]=r34  
3 r35=[r33 ]
```

Desambiguation de la mémoire

- Memoire : Cout important dans le programme
- Réordonnancement des chargements et déchargements
- Necessaire pour les optimisations

Exemples

Structure

```
1 a.field1 = ...  
  ...  
3 ... = b.field2
```

Références indirectes

```
1 *p = ...  
  ...  
3 ... = *q
```

Référence de tableaux

```
1 do i=0,n  
  a(i)=a(i-1)+a(i-2);  
3 enddo
```

Appel de fonction

```
1 *p = ...  
  foo();  
3 *p = ...
```

Plan

- 1 Introduction
- 2 Architecture
- 3 Optimisation
- 4 Optimisation mémoire**
 - Optimisation du cache
 - Chargement et déchargement
 - Préchargement des données
- 5 Optimisation scalaire
- 6 Générateur de code

Buts

- Améliorer le comportement des accès mémoire
- Mettre en évidence le parallélisme (petite et grande échelle)
- Vectoriser

Linear loop transformation

```
1 do i = 1,1000  
  do j = 1,1000  
3    c(j)=c(j)+a(i,j)*b(j)  
  enddo  
5 enddo
```

```
1 do j = 1,1000  
  do i = 1,1000  
3    c(j)=c(j)+a(i,j)*b(j)  
  enddo  
5 enddo
```

Loop fusion

```
1 do i = 1,1000  
  a(i)=x  
3 enddo  
  do i = 1,1000  
5    c(i)=a(i-1)+d(i-1)  
    d(i)=c(i)  
7 enddo
```

```
1 do i = 1,1000  
  a(i)=x  
3    c(i)=a(i-1)+d(i-1)  
    d(i)=c(i)  
5 enddo
```

Loop block-unroll-jam

```
1 do i = 1,2*n  
  do j = 1,2*n  
3     b(j,i)=a(j,i-1)+a(j,i  
      )+a(j,i+1)  
  enddo  
5 enddo
```

```
1 do i = 1,2*n,2  
  do j = 1,2*n  
3     b(j,i)=a(j,i-1)+a(j,i  
      )+a(j,i+1)  
     b(j,i+1)=a(j,i)+a(j,i  
      +1)+a(j,i+2)  
5  enddo  
  enddo
```

Scalar Replacement

- Rotation des registres
- Utilisation du *data dependence graph*

```
do i = 1,2*n
  a(i)=a(i-1)*...
  ...=a(i)-a(i-1)
enddo
```

```
t1 = a(1)
2 do i = 1,2*n
  t2 = t1*...
  4 a(i)=t2
  ...=t2-t1
  6 t1=t2
enddo
```

Register blocking

```
1 do i = 1,2*m
  do j = 1,2*n
3     a(i,j)=a(i-1,j)+a(i
        -1,j-1)
  enddo
5 enddo
```

```
1 do i = 1,2*m,2
  do j = 1,2*n,2
3     a(i,j)=a(i-1,j)+a(i
        -1,j-1)
     a(i+1,j)=a(i,j)+a(i,j
        -1)
5     a(i,j+1)=a(i-1,j+1)+a
        (i-1,j)
     a(i+1,j+1)=a(i,j+1)+a
        (i,j)
7  enddo
  enddo
```

Fonctionnement

- Coûte des ressources
- Principe de localité spatiale et temporelle
- Augmentation du code
- Optimisation pour cas particulier

Prefetching

```
1 do j = 1,2*n  
2   do i = 1,2*m  
3     a(i,j)=a(i,j)+b(0,i)+  
4       b(0,i+1)  
5   enddo  
6 enddo
```

```
1 do j = 1,2*n  
2   do i = 1,2*m  
3     a(i,j)=a(i,j)+b(0,i)+  
4       b(0,i+1)  
5     if (mod(i,8)==0)  
6       call prefetch(a(i+k  
7         ,j))  
8     if (j==1)  
9       call prefetch(b(0,i  
10        +k+1))  
11   enddo  
12 enddo
```

Support du parallélisme

- OpenMP (compilateur)
- Traiter plusieurs opérations de flottant en //
- Load-Pairs

```
1 do j = 1,1000  
   y(j)=y(j)+a*x(j)  
3 enddo
```

```
1 do j = 1,1000  
   t1,t2 = ldfpd(x(j),x(j  
   +1))  
3   t3,t4 = ldfpd(y(j),y(j  
   +1))  
   y(j)=t3+a*t1  
5   y(j+1)=t4+a*t2  
enddo
```

Plan

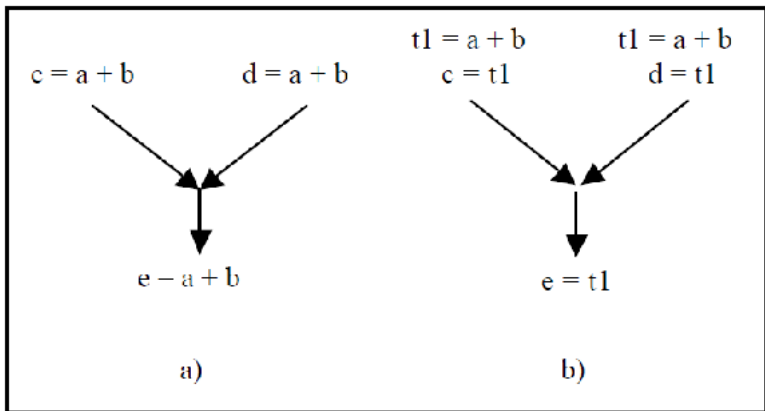
- 1 Introduction
- 2 Architecture
- 3 Optimisation
- 4 Optimisation mémoire
- 5 Optimisation scalaire**
 - PRE traditionnel
 - Extension du PRE
 - Partial Dead Store Elimination
- 6 Générateur du code

Optimisation scalaire

- Introduction :
 - Minimiser :
 - Le nombre d'opération de calcul
 - Le nombre de référence de mémoire
 - Réduire :
 - READ/WRITE

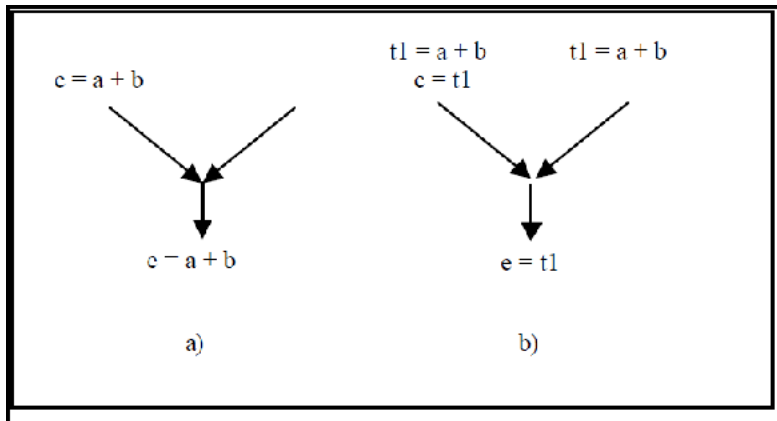
PRE traditionnel

- L'information est redondante dans tous les chemins



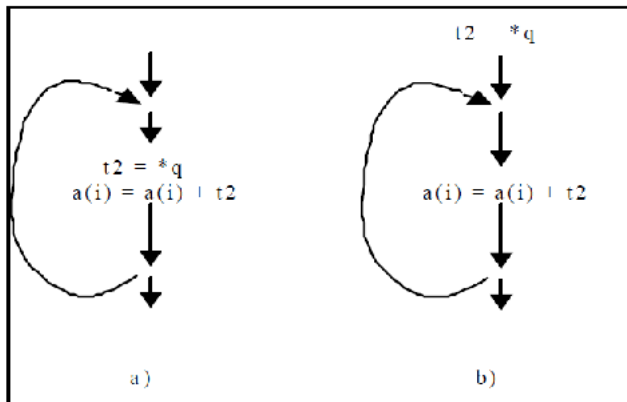
PRE traditionnel

- L'information est uniquement redondante pour certain



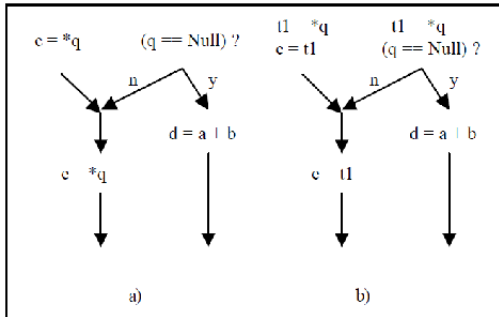
PRE traditionnel

- La redondance dans la boucle



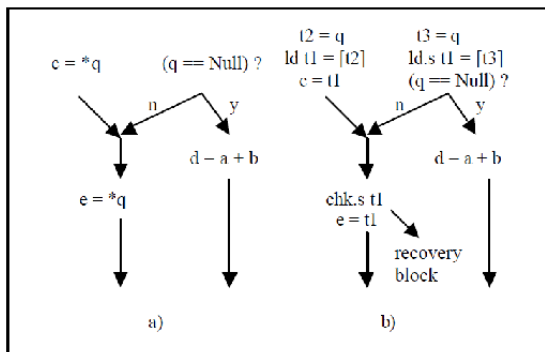
PRE traditionnel

- Attention !
- Ca peut être incorrect après l'insertion.
- Réduction / Augmentation



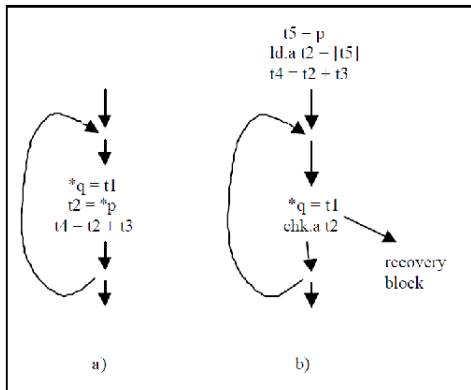
Extension du PRE dans IA-64

- Contrôle du Spéculation
- Check coûte moins cher
- N'utilise pas la mémoire (hidden load latency)



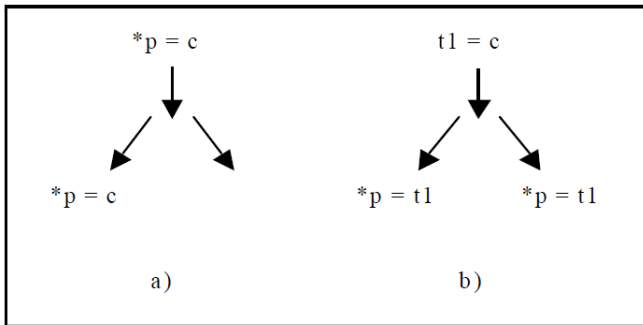
Extension du PRE dans IA-64

- Redondance dans les lectures



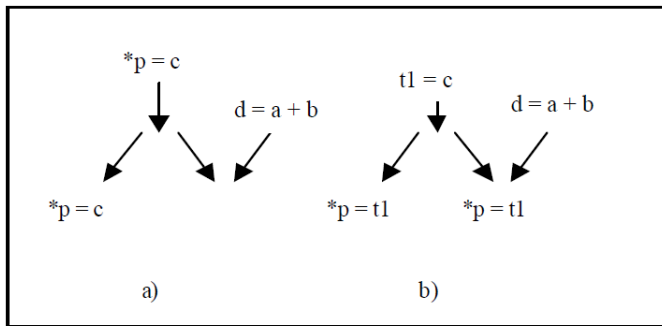
Partial Dead Store Elimination

- Enlève des redondances dans les écritures



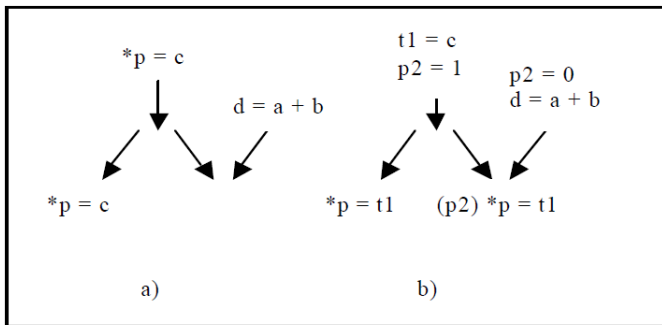
Partial Dead Store Elimination

- Exemple incorrecte :



Partial Dead Store Elimination

- La correction :



Plan

- 1 Introduction
- 2 Architecture
- 3 Optimisation
- 4 Optimisation mémoire
- 5 Optimisation scalaire
- 6 Générateur du code**
- 7 Conclusion

Générateur du code

- Prédiction
 - Améliorer l'efficacité de branchement
 - Éliminer des branchements
- Pipeline
- Global Code Scheduling
- Allocation de registre

Exemple de prédiction

```
2  if (a<b)
   s=s+a
4  else
   s=s+b
```

```
6  Cmp.lt p1,p2 =a,b
   (p1) s = s + a
   (p2) s = s + b
```

Exemple d'allocation de registre

```
8 (p1) v1 = 10  
  (p2) v2 = 20 ;;  
10 (p1) st4[v10] = v1  
    (p2) v11 = v2 + 1 ;;
```

```
12 (p1) r32 = 10  
    (p2) r32 = 20 ;;  
14 (p1) st4[r33] = r32  
    (p2) r34 = r32 + 1 ;;
```

Plan

- 1 Introduction
- 2 Architecture
- 3 Optimisation
- 4 Optimisation mémoire
- 5 Optimisation scalaire
- 6 Générateur du code
- 7 Conclusion**

- Organisation du compilateur
- Optimisation
 - Minimiser l'accès mémoire
 - Minimiser le branchement
 - Maximiser // niveau instruction
- Lien entre le compilateur et l'architecture IA64