Génération de code (suite)

Rémi Forax

Plan

- Génération assembleur
- Allocation de registre
- Optimisations

Génération de code assembleur

- Partie la plus difficile de la compilation
- On génère le code machine à partir de du code intermédiaire
  - Choix des instructions
  - Choisir si les variables sont stockés dans les registres ou sur la pile
- On choisit en fonction du modèle de performance de chaque instruction qui dépend de chaque processeur

Choix des instructions

- Il y a souvent plusieurs instructions possibles pour plusieurs instructions du code intermédiaire

```
iload_1
iconst_2
idiv
```

- Traductions possibles
  - sdiv i32 %1, 2
  - shr i32 %1, 1

Macro instruction

- Une instruction assembleur (CISC) peut effectuer plusieurs micro-instructions
  - Exemple

    aload 1
    aload 2
    iadd
    istore 3
  - Traduction
    - leal (%1, %2), %3

Choix des instructions

- L'algorithme générale consiste à essayer tout les pavages possibles
  - Comme c'est NP-complet,
    Obliger d'utiliser un heuristique
- Par exemple, heuristique gloutonne
  - On met un poids chaque chaque pavage
  - On choisi le pavage de poids minimal sans se préoccuper des pavages suivants

Attribution des registres

- Comme on a pas un nombre de registre infini, il faut choisir quelle variable va dans quel registre
- Si on ne peut pas tout stocker, les variables restantes seront sur la pile (perf !)

- Pour cela on détermine la zone de vie des variables

Zone de vie des variables

- Première approximation
  - On prend les variables par scope et à l'intérieur d'un scope dans l'ordre d'arrivées
  - En fait c'est une très mauvaise approximation,
    - Il faut mieux regarder les affectations/usages des variables

Première approximation

- i = argc * 2
  j = i * 3 + 5
  k = i – j
  l = j + 1
  print k l
- 4 registre %1, %2, %3, %4
  %2 = %1 * 2
  %3 = %2 * 3 + 5
  %4 = %2 - %3
  push %2 // finalement sert à rien
  %2 = %3 + 1
  print %3 %2

Declaration/Utilisation

```
                          i        j        k        l
i = argc * 2         begin
j = i * 3 + 5                 begin
k = i – j            end                begin
l = j + 1                     end                begin
print k l                                   end      end
```

- i et k peuvent être stocké dans le même registre

Graph de flot d'instructions

- Graphe orienté dont les noeuds sont les instructions, et qui relie les instructions qui peuvent se suivre pendant l'exécution du programme :
  - une flèche entre deux instructions qui se suivent quand la première n'est pas un saut inconditionnel
  - une entre entre un instruction de saut (conditionnel ou non) et sa destination

Calcul de zone de vie

- On utilise la forme SSA
- On suit les flèches du graphe de flot d'instruction en **sens inverse**
- On part de la dernière utilisation d'une variable et on remonte à son assignation
- La zone de vie est la zone entre l'assignation et la dernière utilisation

Exemple de zone de vie

```
int x = 1
int y;
int z = x + 3;
if (condition)
  z = 7
  y = 2 * x
  print z
else
  z = 3
  print x
  y = 5
print y
print z
```

Exemple de zone de vie: SSA

```
a1 = 1
a2 = a1 + 3;
if (condition)
  a3 = 7
  a4 = 2 * a1
  print a3
else
  a5 = 3
  print a1
  a6 = 5
a7 = φ(a4, a6)
a8 = φ(a3, a5)
print a7
print a8
```

Zone de vie

```
a1 = 1              a1:b
a2 = a1 + 3;              a2:b/e
if (condition)
  a3 = 7                        a3:b
  a4 = 2 * a1    a1:e                 a4:b
  print a3
else
  a5 = 3                                  a5:b
  print a1         a1:e
  a6 = 5                                        a6:b
a7 = φ(a4, a6)                      a4:e        a6:e a7:b
a8 = φ(a3, a5)                 a3:e       a5:e                 a8:b
print a7                                                 a7:e
print a8                                                      a8:e
```

Graphe d'incompatibilité

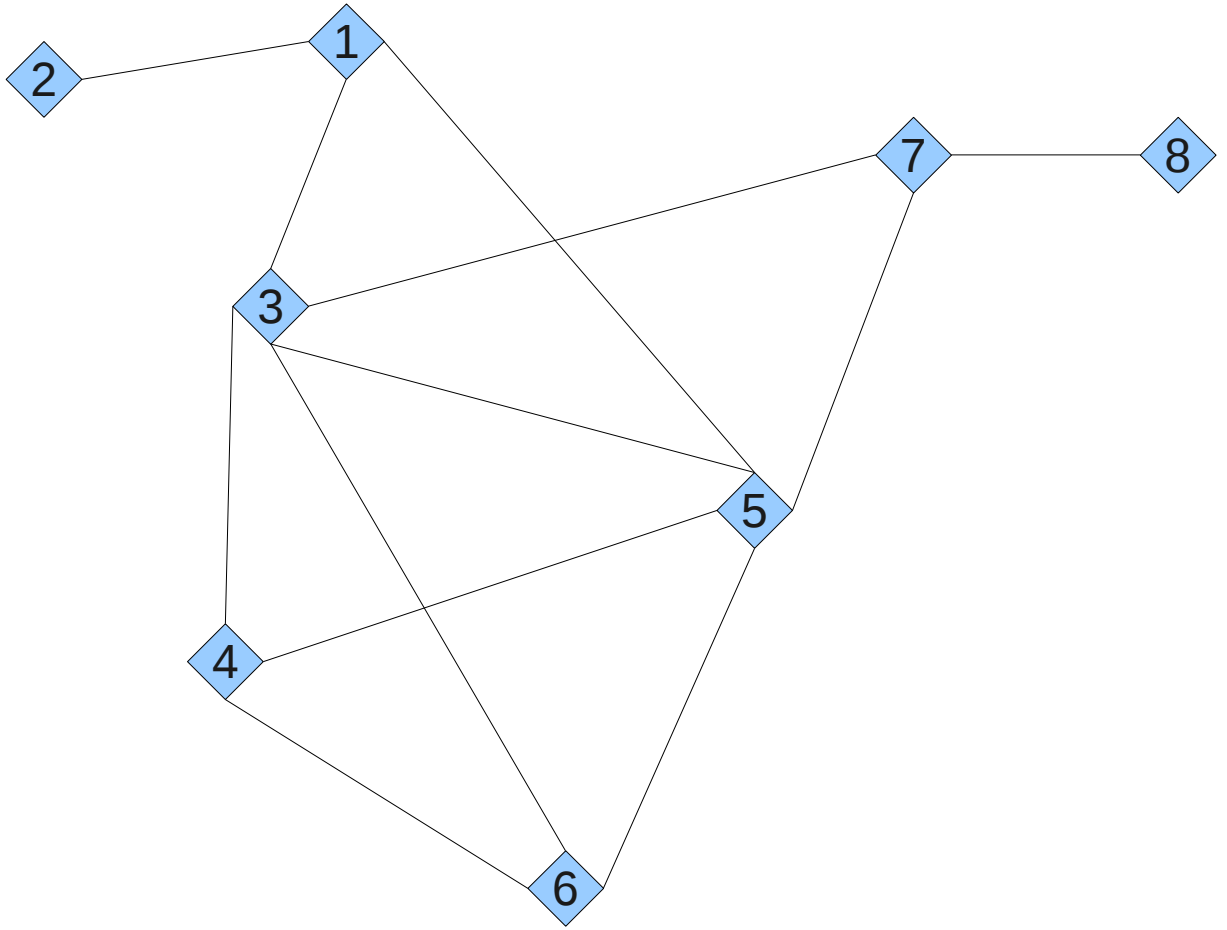
- Deux variables sont incompatibles quand leurs zones de vie se recouvrent
- On dessine un graphe d'incompatibilité, dont les noeuds sont les variables, et qui relie les variables incompatibles
- On effectue un coloriage du graphe, et à chaque couleur correspond un registre (encore un problème NP-complet)
- S'il n'y a pas assez de registres, il faut choisir de stocker les variables les moins utilisées (attention aux boucles) en mémoire.

Graphe d'incompatibilité

	a1	a2	a3	a4	a5	a6	a7	a8
a1		X	X		X			
a2								
a3				X	X	X	X	
a4					X	X		
a5						X	X	
a6								
a7								X
a8								

Coloriage possible

a1	a2	a3	a4	a5	a6	a7	a8
r1	r2	r2	r1	r3	r4	r1	r2



Avec les registres

```
r1 = 1
r2 = r1 + 3;
if (condition)
  r2 = 7
  r1 = 2 * r1
  print r2
else
  r3 = 3
  print r1
  r4 = 5
r1 = φ(r1, r4)
r2 = φ(r2, r3)
print r1
print r2
```

En supprimant les phis

```
r1 = 1
r2 = r1 + 3;
if (condition)
  r2 = 7
  r1 = 2 * r1
  print r2
else
  r3 = 3
  print r1
  r4 = 5
  r1 = r4
  r2 = r3
print r1
print r2
```

Conclusion

- Ce n'est peut-être pas le meilleur coloriage
- Ici pour l'exemple, on a oublié la phase de suppression des instructions inutiles après le SSA
- Il existe une version du SSA dit pruned SSA ou le calcul des phis ne se fait que si nécessaire (si la variable est accédée)

Optimisations

- Les optimisations ont lieu sur le code intermédiaire et plus rarement sur l'assembleur final

- Les catégories habituelles sont
  - Éliminations de code redondant et non nécessaire
  - Optimisation des boucles
  - Suppressions de certaine constructions
    - Appel de fonction, allocation, synchronized, appel polymorphe(virtuel), ...

Constant Folding

Précalcul des constantes

Exemple:

- int f(int x) {
  return x+3;
  }
- void main() {
  char* msg;
  if (f**(2 + 2**) < 7) { => if (f(**4**) < 7)
  msg = "foo";
  } else {
  msg = "bar";
  }
  printf(msg);
  }

Inlining

- Insert le code d'une fonction au site d'appel
- Le code de f reste car il peut être appelé par un autre site d'appel

- ```
  int f(int x) {
      return x+3;
  }
  ```

- ```
  void main() {
      char* msg;
      if (f(4) < 7) {          => if ( (4 + 3) < 7)   => if ( 7 < 7)  => if ( false )
          msg = "foo";
      } else {
          msg = "bar";
      }
      printf(msg);
  }
  ```

Dead code removal

- Supprime les branches de code inutile après inlining

- int f(int x) {
  return x+3;
  }

- void main() {
  char* msg;
  if (**false**) { => {
  msg = "foo"; msg = "bar";
  } else { }
  msg = "bar";
  }
  printf(msg);
  }

Code après optimisation

- ```
  int f(int x) {
      return x+3;
  }
  ```
  ```
  void main() {
      printf("bar");
  }
  ```

Soit en assembleur (x64):

- ```
  @.str1 = private constant [4 x i8] c"bar\00", align 1 ; <[4 x i8]*> [#uses=1]
  ```
- ```
  define i32 @f(i32 %x) nounwind readnone {
  entry:
    %0 = add nsw i32 %x, 3                          ; <i32> [#uses=1]
    ret i32 %0
  }
  ```
- ```
  define void @main() nounwind {
  entry:
    %0 = tail call i32 (i8*, ...)* @printf(i8* noalias getelementptr inbounds ([4 x i8]*
              @.str1, i64 0, i64 0)) nounwind ; <i32> [#uses=0]
    ret void
  }
  ```
- ```
  declare i32 @printf(i8* nocapture, ...) nounwind
  ```
  ```

Common subexpression elimination

- void main() {
  double d;
  scanf("%lf", &d);
  printf("%lf %lf\n", **sin(d)**, **sin(d)**);
  }

- En assembleur:

- define void @main() nounwind {
  entry:
  %d = alloca double, align 8 ; <double*> [#uses=2]
  %0 = call i32 (i8*, ...)* @scanf(i8* noalias getelementptr inbounds ([4 x i8]* @.str,
  i64 0, i64 0), double* %d) nounwind ; <i32> [#uses=0]
  %1 = load double* %d, align 8 ; <double> [#uses=1]
  %2 = call double @sin(double %1) nounwind readonly ; <double> [#uses=2]
  %3 = call i32 (i8*, ...)* @printf(i8* noalias getelementptr inbounds ([9 x i8]* @.str1,
  i64 0, i64 0), double %2, double %2) nounwind ; <i32> [#uses=0]
  ret void
  }

Optimisation des boucles

- loop hoisting & invariant
- range check elimination
- loop-unrolling
- partial loop-unrolling

Loop hoisting

- Les valeurs invariant de la boucle peuvent être sortie de la boucle

```
double f(int val, int max) {
  double result = 0;
  for(int i = 0; i< max; i++) {
    result += sin(val*val);
  }
  return result;
}
```

Loop hoisting

- ```
  define double @_Z1fii(i32 %val, i32 %max) nounwind readonly {
  entry:
    %0 = icmp sgt i32 %max, 0                       ; <i1> [#uses=1]
    br i1 %0, label %bb.nph, label %bb2

  bb.nph:                                           ; preds = %entry
    %1 = mul i32 %val, %val                         ; <i32> [#uses=1]
    %2 = sitofp i32 %1 to double                    ; <double> [#uses=1]
    %3 = tail call double @sin(double %2) nounwind readonly ; <double> [#uses=1]
    br label %bb

  bb:                                               ; preds = %bb, %bb.nph
    %result.05 = phi double [ 0.000000e+00, %bb.nph ], [ %4, %bb ] ; <double>
  [#uses=1]
    %i.04 = phi i32 [ 0, %bb.nph ], [ %5, %bb ]     ; <i32> [#uses=1]
    %4 = fadd double %3, %result.05                 ; <double> [#uses=2]
    %5 = add nsw i32 %i.04, 1                       ; <i32> [#uses=2]
    %exitcond = icmp eq i32 %5, %max                ; <i1> [#uses=1]
    br i1 %exitcond, label %bb2, label %bb

  bb2:                                              ; preds = %bb, %entry
    %result.0.lcssa = phi double [ 0.000000e+00, %entry ], [ %4, %bb ] ; <double>
  [#uses=1]
    ret double %result.0.lcssa
  }
  ```
  ```

Range check elimination

- En Java, on doit tester les bornes des tableaux

```
int[] array = ...
for(int i=0; i< array.length; i++) {
  System.out.println(array[i]);
}
```

Ici, ce n'est pas nécessaire car
0<=i< array.length est toujours vrai

Range check elimination

- On peut détecter les erreurs de borne en amont

```
int max = ...
int[] array = ...
for(int i=0; i< max; i++) {
  System.out.println(array[i]);
}
```

-
```
  if (max<0) goto end
  iconst_0
  istore i
  if (max <= array.length) goto start
  throw new AIOOB
start:
  aload array
  iload i
  iaload
  print
  iinc 1 i
  if (i<max) goto start
end:
```

Loop unrolling

- void main() {
  for(int i=0; i<3; i++) {
  printf("%d\n", i);
  }
  }

- assembleur:

- define void @main() nounwind {
  entry:
  %0 = tail call i32 (i8*, ...)* @printf(i8* noalias getelementptr inbounds ([4 x i8]* @.str, i64 0, i64 0), i32 0) nounwind ; <i32> [#uses=0]
  %1 = tail call i32 (i8*, ...)* @printf(i8* noalias getelementptr inbounds ([4 x i8]* @.str, i64 0, i64 0), i32 1) nounwind ; <i32> [#uses=0]
  %2 = tail call i32 (i8*, ...)* @printf(i8* noalias getelementptr inbounds ([4 x i8]* @.str, i64 0, i64 0), i32 2) nounwind ; <i32> [#uses=0]
  ret void
  }

Loop unrolling

- ```
  int f(int val, int max) {
    int result = 0;
    for(int i = 0; i< max; i++) {
      result += val;
    }
    return result;
  }
  ```
- ```
  define i32 @_Z1fii(i32 %val, i32 %max) nounwind readnone {
  entry:
    %0 = icmp sgt i32 %max, 0                       ; <i1> [#uses=1]
    br i1 %0, label %bb.nph, label %bb2

  bb.nph:                                           ; preds = %entry
    %tmp = mul i32 %max, %val                       ; <i32> [#uses=1]
    ret i32 %tmp

  bb2:                                              ; preds = %entry
    ret i32 0

  }
  ```

Partial loop unrolling

```
{
  int max;
  scan max;
  for(i in [0..max]) {
    if (i==0)
      println "first"
    println i
  }
}
```

- On déroule le premier tour de boucle

```
scan_int &max
if (i<0) goto end
println "first"
loop:
println i
i = i + 1;
if (i<max) goto loop
end:
```