Optimisation de code

Brique ASC

Samuel Tardieu
sam@rfc1149.net

École Nationale Supérieure des Télécommunications

But

L'optimisation cherche à

- améliorer les performances
- réduire l'encombrement du code
- réduire l'encombrement mémoire à l'exécution

Ces contraintes sont souvent contradictoires.

Phases d'optimisation

- À l'écriture du code :
  - choix d'un meilleur algorithme
- Sur l'arbre intermédiaire :
  - élimination du code mort
  - optimisation des sous-expressions
  - optimisation des boucles
- Sur le générateur de code :
  - choix des instructions
  - ordonnancement des instructions

Optimisation algorithmique

- À la louche : 90% du temps est passé dans 10% du code
- Optimiser ces 10% est primordial
- L'optimisation des 90% restants est pénible et frustrante

Exemples d'optimisations

Dans certains cas (seulement), des optimisations simples peuvent améliorer grandement les performances :

- Remplacer un algorithme de tri par un autre
- Utiliser un tableau plutôt qu'une liste chaînée
- Utiliser l'allocation sur la pile (`alloca()`) plutôt que dans le tas (`malloc()`)
- Faire calculer le plus de choses possibles à la compilation (Forth, C++)

Mesure de performance

```
void f () {
   unsigned int i;
   for (i = 0; i < 10000000; i++) {
      free (malloc (10000));
   }
}
```

Les temps sur un Pentium III 650 (FreeBSD) sont :

- avec `malloc()` : 3,62 secondes
- avec `alloca()` : 0,03 secondes

Outils d'évaluation

- Horloge "au mur" : `time commande` sous Unix
- Profiler : `gprof`
  - donne les temps d'exécution et le nombre de passages dans chaque fonction
  - indique les graphes d'appel et les cycles

Représentation du programme

Pour l'optimisation, on préfère travailler sur un graphe :

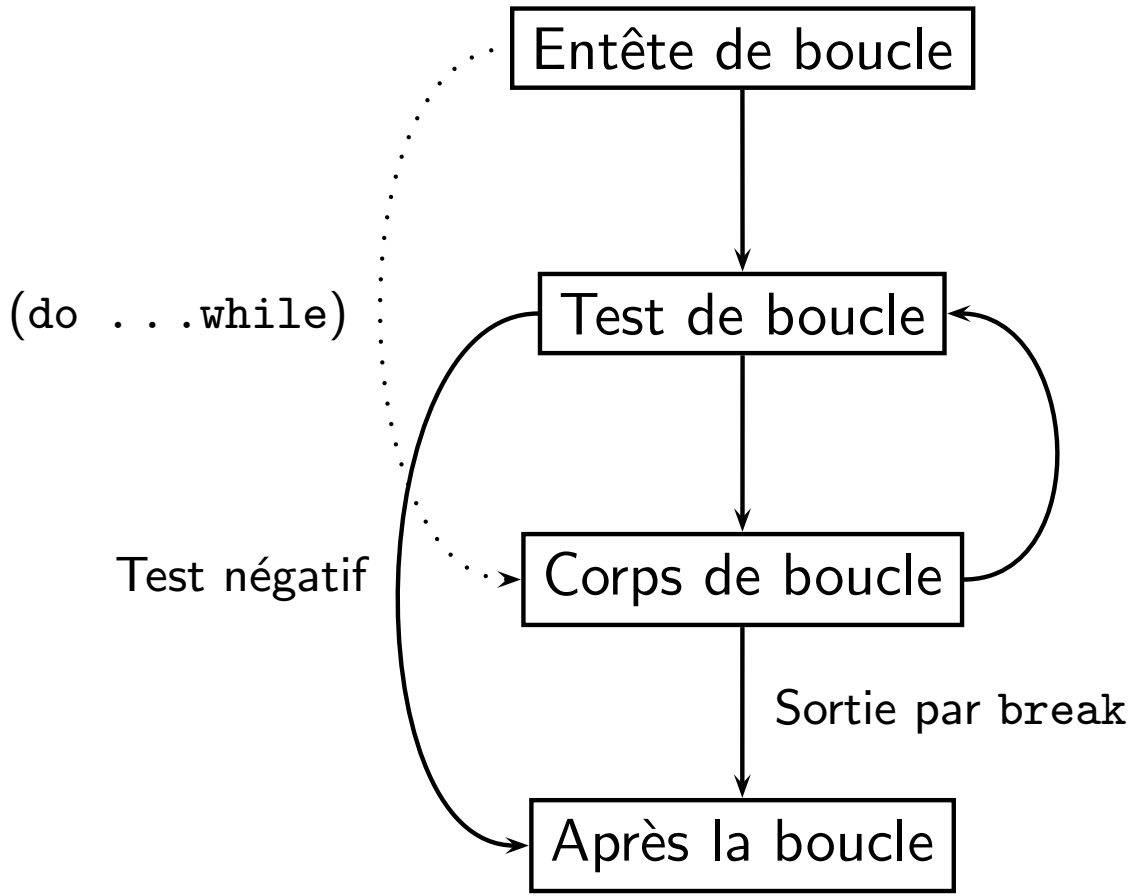
- Les nœuds sont les instructions élémentaires
- Une arête de A vers B représente un référencement de B par A :
  - fin du programme $\Rightarrow$ 0 départ
  - branchement $\Rightarrow$ 1 ou plusieurs départs
  - pointeur sur fonction $\Rightarrow$ 1 départ supplémentaire
  - instruction « normale » $\Rightarrow$ 1 départ

Découpage en blocs

Les nœuds peuvent être regroupés en bloc :

- Les nœuds d'un bloc, sauf le premier, n'ont qu'une arête provenant du nœud précédent du bloc
- Les nœuds d'un bloc, sauf le dernier, n'ont qu'une arête vers le nœud suivant du bloc
- Les blocs les plus longs possibles sont constitués

Exemple : une boucle



Optimisations du graphe

Les optimisations se rangent en deux catégories :

- les **optimisations locales**, à l'intérieur d'un bloc unique
- les **optimisations globales**, impliquant plusieurs blocs consécutifs

Élimination du code mort

- Un bloc auquel n'arrive aucune arête est du **code mort** et peut être éliminé (ainsi que les arêtes qui en partent) sans altération sémantique
- L'élimination du code mort peut faire apparaître du nouveau code mort
- L'application d'autres optimisations peut faire apparaître du code mort

```
if (debugFlag) { ...code mort... }
```

Code vivant

Le point d'entrée d'un sous-programme exporté est vivant :

- ce sous-programme peut être appelé par du code se trouvant dans un autre fichier
- `main()` en C est un exemple
- les fonctions `static` en C ne sont pas exportées et peuvent être éliminées

Sous-expressions communes

Souvent, des expressions ou sous-expressions sont calculées plusieurs fois dans un même bloc sans que les paramètres changent. Ces expressions peuvent être implicites, et non optimisables par le programmeur.

```
int t[10], u[10];
...
for (i = 0; i < 10) {
   u[i] = t[i] + 1;  /* double calcul de i*4 */
}
```

Exemple

À l'intérieur du corps de la boucle, on a :

$$\begin{aligned} r_1 &\leftarrow i * 4 \\ r_2 &\leftarrow [t + r_1] \\ r_3 &\leftarrow r_2 + 1 \\ r_4 &\leftarrow i * 4 \\ r_5 &\leftarrow u + r_4 \\ [r_5] &\leftarrow r_3 \end{aligned}$$

r_4 peut être éliminé et remplacé par r_1

Exemple avec optimisation

À l'intérieur du corps de la boucle, on a maintenant :

$$\begin{aligned} r_1 &\leftarrow i * 4 \\ r_2 &\leftarrow [t + r_1] \\ r_3 &\leftarrow r_2 + 1 \\ r_5 &\leftarrow u + r_1 \\ [r_5] &\leftarrow r_3 \end{aligned}$$

Gain : 1 instruction sur 6

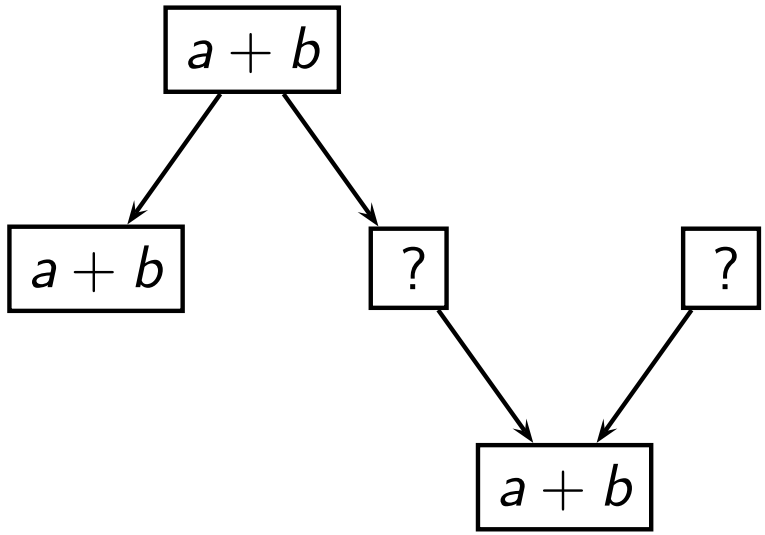
GCSE

GCSE : **Global** Common Sub-Expression Elimination

- Extension de l'élimination de sous-expressions communes à un ensemble de blocs
- Nécessite d'examiner les chemins de données entre les blocs
- Existe en deux variantes : classique et PRE (Partial Redundancy Elimination)

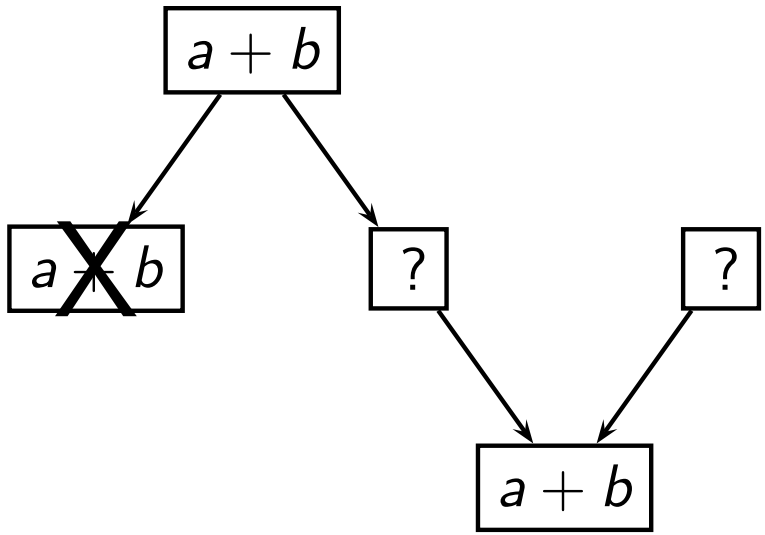
Situation de départ

Plusieurs blocs calculent $a + b$. Quels calculs redondants supprimer ?



GCSE classique

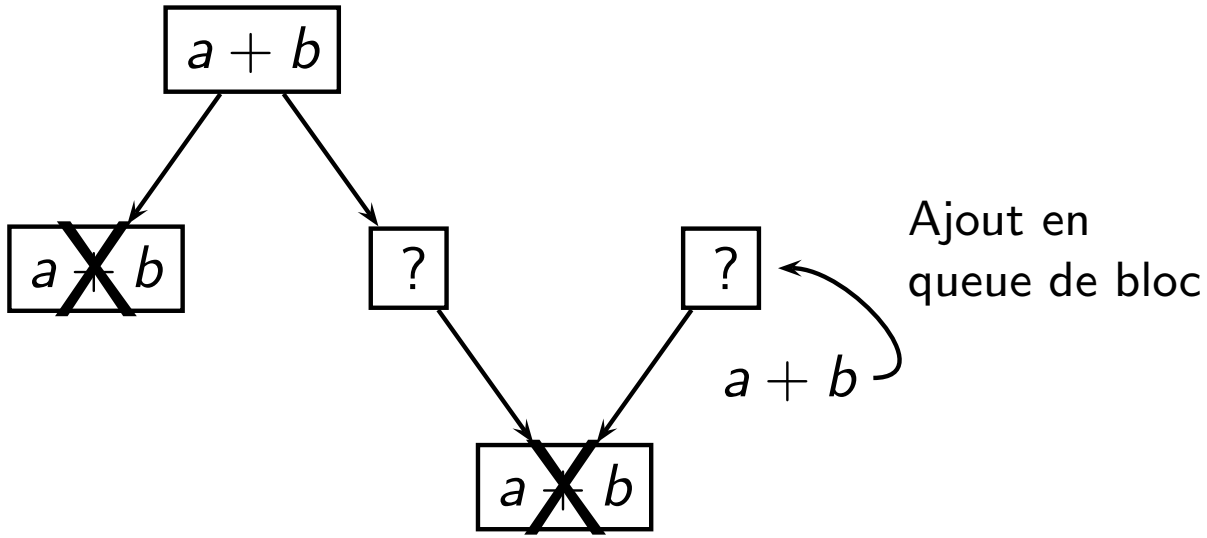
On peut supprimer un calcul, car $a + b$ a déjà été calculé sur le chemin :



Le second ne peut pas être supprimé.

GCSE avec PRE

On rajoute un calcul sur l'autre chemin :



C'est de toute façon plus efficace.

Inconvénients du GCSE

- La durée de vie des registres contenant les sous-expressions est allongée : problème avec certaines architectures CISC
- Plus de code peut être généré (PRE) si le calcul doit être ajouté dans plusieurs blocs. Toutefois, on peut générer un bloc supplémentaire dédié au calcul de cette expression

Affectations multiples

À l'intérieur d'un bloc, une affectation à une variable (ou registre) cible peut être supprimée si les conditions suivantes sont vérifiées simultanément :

- la variable est affectée une seconde fois ou elle atteint sa limite de durée de vie dans le bloc (dans ce cas, une affectation fictive est supposée en fin de bloc)
- entre les deux affectations, la variable n'est pas utilisée en lecture

Propagation des copies

- Idée : lorsqu'une copie est faite et utilisée, on utilise l'original plutôt que la copie
- But : pouvoir éliminer l'affectation par la suite

Considérons le code :

$$\begin{array}{lcl} x & \leftarrow & r_1 \\ & \ldots & (\text{code n'utilisant pas } r_1) \\ x & \leftarrow & x+1 \end{array}$$

Résultat de la propagation

$$
\begin{array}{lcl}
x & \leftarrow & r_1 \\
 & \ldots & \text{(code n'utilisant pas } r_1\text{)} \\
x & \leftarrow & r_1 + 1
\end{array}
$$

Étape suivante : (affectations multiples)

$$
\begin{array}{lcl}
 & \ldots & \text{(code n'utilisant pas } r_1\text{)} \\
x & \leftarrow & r_1 + 1
\end{array}
$$

Simplification d'expressions

Il est possible de simplifier certaines expressions :

- en précalculant certaines valeurs
- en utilisant les identités remarquables

Précalcul de valeurs

Il est possible de précalculer certaines valeurs (fonctions standards, opérations arithmétiques sur constantes, etc.)

- Le résultat doit être équivalent à ce qui aurait été calculé à l'exécution
- Une solution existe pour ne pas interpréter le code : compiler l'expression et prendre l'évaluation

Identités remarquables

$$
\begin{aligned}
x \times 0 &= 0 \\
x \times 1 &= x \\
x \times 2 &= x + x \\
x^2 &= x \times x \\
x + 0 &= x \\
x - C &= x + (-C) \\
x - x &= 0 \\
0/x &= 0 \\
&\dots
\end{aligned}
$$

Exemple

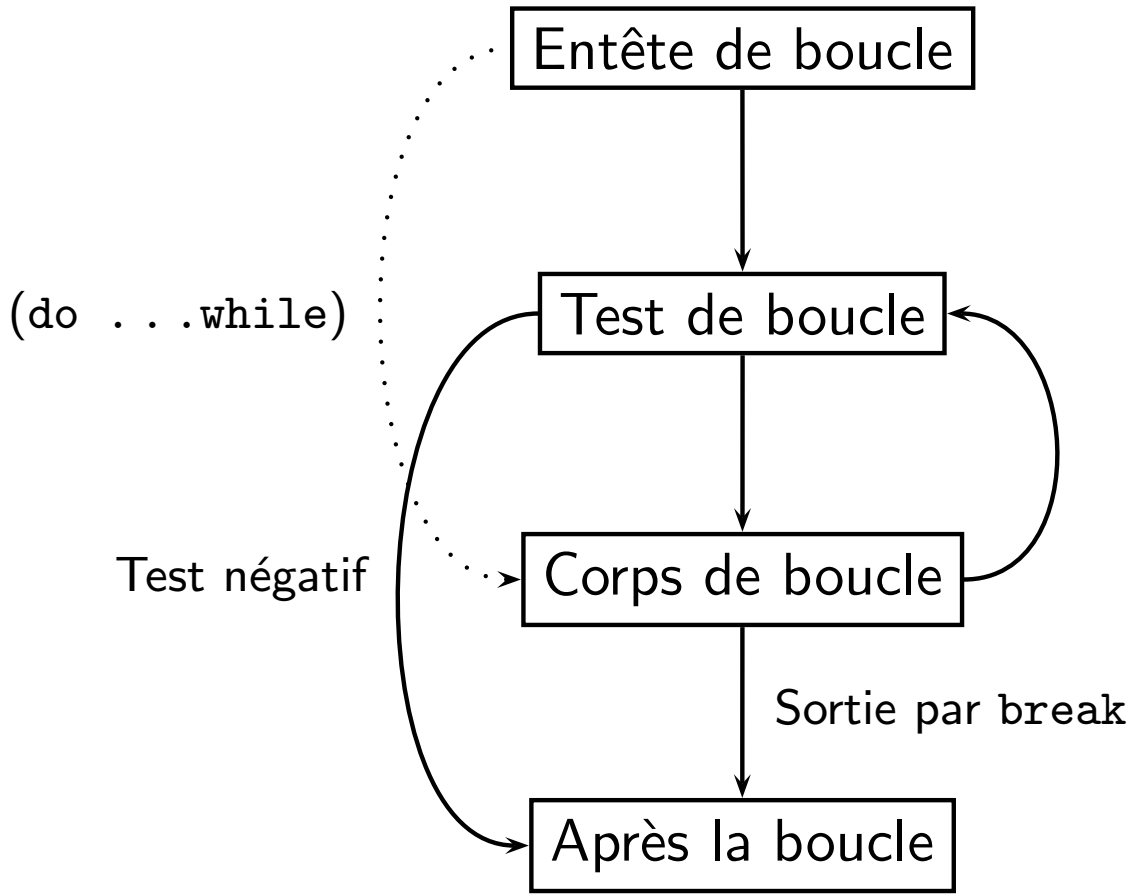
Le code :

```
int x;
int g () {
   return (3*4 + x*7 - 5) / 7;
}
```

génère avec GCC sur IA32 :

```
g:
        movl x,%eax
        incl %eax
        ret
```

Structure d'une boucle



Optimisations de boucles

Les boucles représentent en général une part importante du temps consommé dans un programme. Les optimisations sont :

- Déplacement de code
- Optimisation des variables induites
- Élimination du code mort

Déplacement de code

- Idée : sortir du code invariant de la boucle
- Solution : le placer dans l'entête

```
for (i = 0; i < limit*2; i++) {
    ...
}
```

devient

```
int j = limit*2;
for (i = 0; i < j; i++) {
    ...
}
```

Variables induites

Soit une boucle d'indice i. Une variable j est directement induite si :

- i est incrémenté d'une valeur constante à chaque tour de boucle
- j est affecté dans la boucle avant toute lecture
- j est de la forme $i \times C$, où C est un invariant de la boucle

Induction indirecte

Soit une boucle et une variable induite j. Une variable k est indirectement induite si :

- k est affecté dans la boucle avant toute lecture
- k est de la forme $j + C$ où C est un invariant de la boucle

Strength reduction

- Idée : soit une boucle d'indice i et d'incrément Δ, optimiser une variable induite k
- Solution :
  - k est de la forme $i \times A + B$ où A et B sont des invariants de la boucle
  - on place $k \leftarrow i_0 \times A + B$ dans l'entête de la boucle
  - on place $k \leftarrow k + \Delta \times A$ à la fin du corps de la boucle
- Dans certains cas, $\Delta \times A$ est même constant

Exemple

Soit le code Ada :

```
with F;
procedure Strength is
begin
   for I in 1 .. 10 loop
      F (I*17+3);
   end loop;
end Strength;
```

Exemple (suite)

Le code devient, sur IA32 :

```
        movl $1,%esi
        movl $20,%ebx
        .align 4
.L5:
        [...]
        addl $17,%ebx
        incl %esi
        cmpl $10,%esi
        jle .L5
```

Le compilateur aurait pu mieux faire.

Inversion du sens de la boucle

Lorsque l'indice de boucle n'est pas utilisé, le compilateur peut inverser la boucle.

```
for I in 1 .. 10 loop
   null;
end loop;
```

est compilé, sur IA32, comme :

```
        movl $9,%eax
        .align 4
.L5:
        decl %eax
        jns .L5      ; jump if no sign
```

Déroulement

Il est parfois possible de “dérouler” une boucle, notamment lorsque le nombre d'itérations est constant.
Exemple :

```
int a;

void f()
{
    int i;
    for (i = 0; i < 4; i++) {
        a++;
    }
}
```

Exemple de déroulement

```
f:
        movl $3,%ecx
.L6:
        incl a
        decl %ecx
        jns .L6
        ret
```

devient, après déroulement : (IA32)

```
f:
        addl $4,a
        ret
```

Récursion terminale

Dans certains cas, un sous-programme peut se rappeler récursivement sans passer par un appel de sous-programme :

- la valeur retournée est le résultat direct d'un appel de sous-programme
- aucun destructeur ne doit être exécuté à la sortie du sous-programme
- aucune exception ne peut être rattrapée dans le sous-programme

Exemple

Le code suivant peut bénéficier de la récursion terminale :

```
unsigned int
tail (unsigned int n)
{
  if (n % 2 == 0)
    return tail (n/2);
  return n;
}
```

Exemple : code généré

```
tail:
        movl 4(%esp),%eax
.L4:
        testb $1,%al
        jne .L3
        shrl $1,%eax
        jmp .L4
.L3:
        ret
```

Le paramètre est placé dans %eax au début (prologue), et on y retourne.

Inlining

- Principe : remplacer les appels à des sous-programmes courts par le code du sous-programme lui-même
- Avantages :
  - économise le coût de l'appel
  - permet une meilleure optimisation et utilisation des registres
- Inconvénients :
  - génère du code plus volumineux
  - peut "casser" une ligne de cache

Inlining : exemple

```
int square (int n) { return n*n; }
int f (int n) { return square(n)*2; }
```

génère (IA32)

```
f:
        movl 4(%esp),%eax
        imull %eax,%eax
        addl %eax,%eax
        ret
```

Conclusion partielle

Les optimisations au niveau des blocs de code sont nombreuses :

- élimination du code mort
- élimination des sous-expressions communes (locale et globale)
- propagation des copies et simplification d'expressions
- optimisation des boucles
- récursion terminale, inlining

Optimisation : génération

La génération de code proprement dite donne lieu à plusieurs familles d'optimisations :

- choix des instructions machines
- choix des registres
- optimisation du code assembleur
- ordonnancement des instructions

Choix des instructions

- Principe : faire correspondre les nœuds aux instructions
- Optimisation : réunir des nœuds successifs d'un même bloc :
  - code plus petit
  - code plus rapide
- Deux techniques possibles :
  - une opération de base, une instruction
  - recherche de couverture optimale à partir d'une description du processeur

L'approche de GCC

Pour chaque architecture supportée, un fichier décrit (entre autres) :

- pour chaque opcode de la cible :
  - les contraintes sur ses arguments et leur type
  - les effets dans un "pseudo-langage" propre à GCC
  - des attributs de description de l'effet (mémoire, *delay slots*, etc.)
- comment construire le prologue et l'épilogue

Couverture optimale

- But : trouver un ensemble d'instructions s'exécutant en un temps minimal
- Méthode : essayer les différentes combinaisons possibles permettant d'éliminer les conflits de registres
- Aides : dans GCC, certaines fonctionnalités peuvent être nommées pour faire correspondre des opérations données à du code machine

Peepholes

- But : remplacer une suite d'instructions assembleur par des séquences codées en dur plus efficaces
- Méthode : regarder le code généré à travers une "fenêtre"
- Exemples :
  - un décalage de pile suivi d'un `push`
  - "substract one and jump if non zero" instruction sur le 1750a

Ordonnancement

- But : ordonner les instructions pour que l'exécution soit plus rapide sans perturber la sémantique
- Méthode :
  - utiliser les attributs d'unités fonctionnelles et de *delay slots* correspondant au processeur
  - générer des *prefetch* et des *streaming stores* pour mieux contrôler le cache

Conclusion partielle

L'optimisation au niveau du générateur de code

- est très liée à l'architecture
- demande énormément de tests du compilateur pour comparer les différents codes générés
- permet de corriger des problèmes liés à la conception du compilateur par des optimisations fines (*peepholes*)

Autres optimisations

- Aiguillage dynamique
- Minimalisation des tests de contraintes
- Exceptions à “coût zéro”
- Page zéro
- Serveurs de compilation
- Caches de compilation

Aiguillage dynamique

- But : transformer les aiguillages dynamiques des langages objets en aiguillages statiques
- Méthode :
  - analyse globale du système
  - inférence de type lorsque c'est possible
  - remplacement des appels dynamiques par des appels statiques
- Implémentations : Eiffel

Tests de contraintes

- But : éliminer des tests de contraintes inutiles
- Méthode :
  - analyse des types et des bornes des données
  - sortie de boucle des tests sur des données invariantes
  - utilisation de vérifications matérielles dès que possible
- Implémentations : GNAT

Exceptions

- But : réduire le coût des exceptions :
  - une exception non levée ne doit rien coûter
  - une exception levée peut prendre du temps
- Méthode :
  - génération dans l'exécutable de tables (code, exception, traite-exception)
  - lors de la levée d'une exception, analyse du graphe d'appel et détermination du traite-exception à appeler
- Implémentations : GNAT

Page zéro

- But : permettre le *pipelining* de `if (p && *p) {...}`
- Problème : en présence d'un *pipe* de plusieurs niveaux, la lecture de *p doit être différée si p vaut zéro (erreur de lecture, adresse non valide)
- Solution : le système d'exploitation autorise la page contenant l'adresse zéro en lecture

Temps de compilation

On peut réduire le temps de compilation :

- Serveur de compilation : un serveur garde en mémoire les arbres sémantiques (langages avec prototypes)
- Cache de compilation : un serveur garde sur disque les versions compilées des fichiers, en cas de recompilation du même fichier avec les mêmes options

Compilation interactive

En Forth,

- il n'y a pas de compilateur central (chaque mot cité génère un appel à lui-même en mode compilation)
- chaque mot peut être rendu "intelligent", optimiser en fonction des mots précédents et positionner des indices pour les mots suivants
- par exemple, `dup drop` fera que `drop` annulera l'effet du `dup`

Conclusion

- L'optimisation a lieu à tous les niveaux, du code source au code machine
- Certaines optimisations prennent du temps à la compilation pour un faible gain, d'autres prennent peu de temps pour un gain énorme
- Plus la connaissance de la machine cible est grande, meilleure est l'optimisation possible
- Toute nouvelle optimisation est source d'erreur potentielle