Université de Montréal

Structure globale

Transformation de l'AST en un graphe de blocs de base

Sélection des instructions

Allocation des registres

Ordonnancement des instructions

Blocs de base

Un bloc de base est une séquence d'instructions avec

- Un seul point d'entrée, au début du bloc
- Un seul point de sortie, à la fin du bloc

Généralement courts

Les appels de fonction peuvent y apparaître, selon les cas

Une liste d'instructions ponctuée par un branchement

Blocs de base étendus

Un bloc de base *étendu* est une séquence d'instructions avec

- Un seul point d'entrée, au début du bloc
- Un point de sortie à la fin du bloc, pas forcément seul

Donc, l'extension est de permettre de quitter le bloc en cours de route

Encore courts, mais moins que les blocs de base

Une liste d'instructions ponctuée par un branchement

D'autres branchments peuvent être inclus dans la liste

Université de Montréal

Graphe de flot de contrôle (CFG)

Un CFG est un graphe dont les nœuds sont des blocs de base

Chaque fonction est représentée par un CFG

Université de Montréal

Exemple (Source)

```
struct list { int x; struct list *next; }

struct list *list_rev (struct list *l) {
  struct list *r = NULL;
  while (l) {
    struct list *new = malloc (sizeof (struct list)
    new->x = l->x;
    new->next = r;
    r = new;
    l = l->next;
  }
  return r;
}
```

Université de Montréal

Exemple (BBs)

```
BB0     struct list *r = NULL; goto BB1

BB1     if null l, goto BB3 else goto BB2

BB2     list *new = malloc (sizeof (list));
        new->x = l->x;
        new->next = r;
        r = new;
        l = l->next;
        goto BB1

BB3     return r;
```

Université de Montréal

Exemple (EBBs)

```
BB0     struct list *r = NULL; goto BB1

BB1     if null l, goto BB2 else
        list *new = malloc (sizeof (list));
        new->x = l->x;
        new->next = r;
        r = new;
        l = l->next;
        goto BB1

BB2     return r;
```

Static Single Assignment (SSA)

Représentation où chaque variable n'est affectée qu'à un endroit

Chaque affectation introduit une nouvelle variable

Chaque BB commence avec des *fonction* ϕ pour compenser

- E.g. $x_9 = \phi(x_1, x_3, x_6)$

Représentation plus explicite du flot de données

Découpe chaque usage de variable en éléments indépendants

- Évite contraintes arbitraires dues à réutilisation de variable

Université de Montréal

Exemple (SSA)

```
BB0     struct list *r0 = NULL; goto BB1

BB1     l1 = phi(l0, l2); r1 = phi(r0, r2);
        if null l, goto BB2 else
        list *new0 = malloc (sizeof (list));
        new->x = l1->x;
        new->next = r1;
        r2 = new0;
        l2 = l1->next;
        goto BB1

BB2     return r1;
```

Conversion vers SSA

Conversion naïve:

- Ajouter une fonction ϕ pour chaque variable au début de chaque BB
- Autant d'arguments que de liens entrant au BB

Minimization:

- Éliminer fonctions ϕ de la forme $x_i = \phi(x_a, x_b, x_c, x_d, ...)$
  où $\{a, b, c, d, ..\} \in \{i, k\}$: Remplacer par x_i par x_k.

Nettoyage (*pruning*):

- Éliminer fonctions ϕ mortes

Université de Montréal

Exemple (autre SSA)

```
BB0()        struct list *r0 = NULL; goto BB1(l0, r0);

BB1(l1,r1)  if null l, goto BB2() else
            list *new0 = malloc (sizeof (list));
            new->x = l1->x;
            new->next = r1;
            r2 = new0;
            l2 = l1->next;
            goto BB1(l2,r2);

BB2()      return r1;
```

Université de Montréal

Conversion minimale vers SSA

***Dominance stricte*:** Un *nœud* A domine strictement un nœud B si on passe nécessairement par A avant B

***Dominance*:** Un *nœud* A domine un nœud B si A = B ou si A domine strictement B.

***Frontière de dominance*:** Un nœud B est dans la frontière de dominance d'un nœud A si A ne domine pas B strictement, mais il domine un prédecesseur immédiat de B.

Un nœud de la forme $x = e$ nécessitera une fonction ϕ à l'entrée de chaque BB dans la frontière de dominance de ce nœud

Arbre de dominance

***Dominateur immédiat*:** nœud qui domine strictement A mais ne domine strictement aucun autre dominateur strict de A.

***Dominator Tree*:** arbre constitué par la relation de *dominance immédiate*. Il a pour racine le nœud de départ et chaque nœud a pour enfants les nœuds dont il est le *dominateur immédiat*

Représentation efficace: chaque nœud a un champ `idom`

$$\begin{aligned} \mathit{IDom}(n) &= n{\rightarrow}\mathsf{idom} \\ \mathit{Dom}(n) &= \{n, n{\rightarrow}\mathsf{idom}, n{\rightarrow}\mathsf{idom}{\rightarrow}\mathsf{idom}, \ldots\} \\ \mathit{SDom}(n) &= \{n{\rightarrow}\mathsf{idom}, n{\rightarrow}\mathsf{idom}{\rightarrow}\mathsf{idom}, \ldots\} \end{aligned}$$

On peut utiliser n lui-même comme représentation de $\mathit{Dom}(n)$

Université de Montréal

A Simple, Fast Dominance Algorithm, Cooper et.al.

Analyse de type *forward dataflow*:

$$\mathit{SDom}(n) = \left(\bigcap_{m \in \mathsf{Pred}(n)} \mathit{Dom}(m) \right)$$

```
for n ∈ nodes, n→idom = ⊤; changed = true; n0→idom = ∅
while changed
  changed = false
  for n ∈ nodes
    idom = (⋂ m∈pred(n) m)
    if n→idom ≠ idom
      n→idom = idom; changed = true
```

Université de Montréal

Intersection efficace de Dom(n)

Numéroter en *preorder* depuis n_0

$$m \in \mathit{SDom}(n) \;\Rightarrow\; \mathit{PreOrder}(m) < \mathit{PreOrder}(n)$$

donc $\mathit{Dom}(n)$ est naturellement trié: $n{\rightarrow}\text{idom} < n$

$$\begin{aligned}
&\mathit{inter}\ n_1\ n_2 = \\
&\quad \texttt{if}\ n_1 = \top\ \texttt{then}\ n_2 \\
&\quad \texttt{else if}\ n_2 = \top \vee n_1 = n_2\ \texttt{then}\ n_1 \\
&\quad \texttt{else if}\ n_1 < n_2\ \texttt{then}\ \mathit{inter}\ n_1\ (n_2{\rightarrow}\text{idom}) \\
&\quad \texttt{else}\ \mathit{inter}\ (n_1{\rightarrow}\text{idom})\ n_2
\end{aligned}$$

Itérer en *preorder* aussi!

Allocation de registres

Analyse de *liveness*

Création du *graphe d'interférence*

Allocation et *spilling*

Passage d'arguments

Coalescing

Simplifications

Université de Montréal

Analyse de liveness

Analyse de type *backward dataflow*:

$$\begin{aligned} \textit{Live-in}(i) &= (\textit{Live-out}(i) - \textit{Defs}(i)) \cup \textit{Use}(i) \\ \textit{Live-out}(i) &= \left(\bigcup_{j \in \text{succ}(i)} \textit{Live-in}(j) \right) \end{aligned}$$

i est une *instruction*

Ensembles généralement implémentés par des *bitset*

Université de Montréal

Construction du graphe d'interférences

Parcours simple du *control flow graph* (CFG)

Interférences entre *defs*(i) et $(\textit{Live-out}(i) - \textit{Defs}(i))$

Graphe typiquement représenté par deux structures de données redondantes

- *bitmatrix IF*, telle que *IF*(v_1, v_2) indique s'il y a interférence
- *edge sets*, où *ES*(v) liste les variables qui interfèrent

Allocation, assignation, et spilling

L'allocation est comparable à la coloration de graphe

Allocation par simplification du graphe d'interférences:

- Si v a moins que k voisins: enlever et ajouter à la pile
- Sinon, choisir un *candidat au spill*: enlever et ajouter à la pile

Ensuite, dépiler les variables dans l'ordre:

- Si registre disponible, *assigner*
- Sinon marquer le registre comme *spill*

S'il y a du *spill*: ajuster le code, et recommencer

Convention d'appel de fonction

Placement des arguments

- Ajouts d'instructions `move`

Placements des valeurs de retour

- Ajouts d'instructions `move`

Registres *caller-save*

- Interférences entre *caller-save* et $(\mathit{Live\text{-}out}(i) - \mathit{Defs}(i))$

Registres *callee-save*

- Ajouts de `move` $t_i := r_i$ ... `move` $r_i := t_i$

Coalescing

Élimination des `move`

Ajouter un graphe de *préférences* *Pref*

Durant la simplification du graphe d'interférence:

- Enlever d'abord les v qui ne sont pas dans *Pref*
- Si *Pref*(v_1, v_2) et ils ont (ensemble) moins que k voisins: *coalesce*

Lors de l'assignation, choisit la même couleur si disponible

Spilling

Choix des variables à spill:

- Coût estimé de placer la variable en mémoire
- Probabilité que faire ce spill de v évite d'autres spill

Choix possible:

$$\mathit{Value}(v) = \frac{|\mathit{ES(v)}|}{\mathit{Cost}(v)}$$

Possibilité aussi de faire du *splitting*

- Divise la variable en deux et ajoute un `move` entre les deux
- Peut parfois éviter le *spill*

Simplifications

Au lieu de tout recommencer, après un spill:

- Mettre à jour le graphe d'inférence
- Résultat *conservateur*

Allocation de registres en SSA

SSA augmente le nombre de variables mais simplifie le graphe:

- Clique de taille N $\Rightarrow$ N variables *live* simultanément!
- Coloration optimale: *greedy* par pré-ordre du dominator tree

Séparation d'allocation et assignation

- Ajout de spill pour réduire taille max de *live-set*
  - Conséquences et gains de chaque spill immédiatement visibles!
- Choix de couleur affecte seulement le *coalescing*
  - Choix optimal de *coalescing* coûteux: NP

Linear Scan Allocation

Allocation de registres très *rapide*, quoique moins bonne

Algorithme classique ans les compilateurs JIT

Ordonne toutes les instructions

- Choix de l'ordre affecte le résultat

Forces chaque *live-range* à être un intervalle contigu

- Approximation conservatrice

Allocation *greedy* en traversant le code linéairement

- S'il n'y a plus de registres, spill le *live-range* le plus long

Université de Montréal

Ordonnancement d'instructions

Rapprocher les instructions *indépendentes*

- Exploiter le parallélisme de type *ILP*

Éloigner les instructions *dépendentes*

- Éviter d'attendre le résultat d'une longue opération

Bloc de base = graphe acyclique de flot de données

- Ordonnancement est un encodage séquentiel
- Processeur OOO re-décode ensuite en un graphe de flot de donnée

Dépendances

RAW (Read After Write): vrai dépendance

WAR (Write after Read): anti-dépendance; lire avant de perdre

WAW (Write After Write): dép. de sortie; perdre la bonne écriture

¡Attention aux *exceptions*!

SSA ne garde que les dépendances vraies

Allocation de registres ré-introduit de fausses dépendances

Construction de graphe de dépendances:

- Graphe orienté
- Chaque arête porte la latence associée

Université de Montréal

Ordonnancement du graphe

L'ordonnancement est un tri topologique:

- Choisir une *source* du graphe
- Ajouter à la fin des instructions accumulées
- Enlèver du graphe; recommencer jusqu'à ce que le graphe soit vide

Choix de la source:

- Considère la latence entre instructions déjà choisies et sources
- Regarde disponibilité de ressource à ce point du code
- Préfère instructions sur le *critical path*

Peut aussi se faire depuis la *fin* en prenant les *sink*