Regards en aval et en amont de la chaîne de compilation

Didier Rémy

2000 - 2001

http://cristal.inria.fr/~remy/poly/compil/9/

http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/compil/9/

En aval

La chaîne de compilation du cours s'arrête à la production de code assembleur pour le MIPS.

En aval de la chaîne de compilation

On pourrait considérer encore quelques étapes :

- L'assemblage.
- L'édition de liens.
- L'ordonnancement des instructions

La gestion mémoire

Les langages moderne offrent une gestion automatique de la mémoire. Un minimum d'interaction entre un gestionnaire mémoire et le compilateur est nécessaire, ne serait-ce que la représentation des données allouées.

Assemblage

Avant d'être exécuté, le code assembleur (fichier ASCII) doit être traduit en code machine (fichier binaire) : il reste à éliminer les pseudo-instructions, à résoudre les étiquettes, générer le code pour le chargement des déclarations de constantes, etc.

(Dans certains cas, l'assemblage peut également comporter une partie d'ordonnancement des instructions.)

En général, l'assemblage est un programme séparé de la compilation, parce qu'il ne dépend que de la machine cible et peut être partagé entre plusieurs compilateurs pour des langages sources différents.

Le programme d'assemblage est aussi, en général, fourni par le constructeur.

Édition de liens

Un programme peut se composer de plusieurs morceaux, parce que le source est composé de plusieurs fichiers ou utilise des librairies compilés séparément.

(Il faut pour cela avoir respecter les même conventions d'appels, de préférence les conventions standards, dans les librairies partagées.)

L'édition de liens peut rassembler plusieurs fichier en un seul.

Les identificateurs déclarés globaux sont partagés. Les étiquettes locales doivent être renommées. Le code doit aussi être relogé (translaté). Pour les librairies partagées entre plusieurs programme, chaque programme processus à sa table d'indirection par rapport aux adresses des fonctions de la libairies partagée.

Ordonnancement des instructions

Les instructions machines se décomposent en des opérations encore plus élémentaires (micro-instructions) exécutées par des unités de calcul du processeur.

Ordonnancement des instructions

Les instructions machines se décomposent en des opérations encore plus élémentaires (micro-instructions) exécutées par des unités de calcul du processeur.

- Parallèlisme
  Les processeurs peuvent exécuter plusieurs micro-instructions en parallèles mais avec certaines contraintes de resource (par exemple un multiplicateur a aussi besoin de l'additionneur).
  Lorsqu'une des unités de calcul est occupée elle va au mieux bloquer le déroulement des calculs en parallèle, au pire interférer avec le calcul en parallèle et fournir un résultat erroné.

Ordonnancement des instructions

Les instructions machines se décomposent en des opérations encore plus élémentaires (micro-instructions) exécutées par des unités de calcul du processeur.

- Parallèlisme
- Pipelining

  Pour profiter au mieux de la possibilité d'effectuer des calculs en parallèle, les processeurs modernes utilisent le pipelining des instructions. Le processeur commence l'exécution d'une instructions i_1 au cycle k qui se poursuit sur plusieurs cycles, mais il entame l'instruction suivante i_2 dès de cycle $k+1$ alors que la première instruction n'a pas forcément fini le calcul.

Ordonnancement des instructions

Les instructions machines se décomposent en des opérations encore plus élémentaires (micro-instructions) exécutées par des unités de calcul du processeur.

- Parallèlisme
- Pipelining

  Par exemple, parce que l'instruction mul $t1, $t1, $t3 est plus lente que l'addition, suivi de l'instruction add $t1, $t1, $t4. La valeur de $t1 prise par l'addition sera alors sa valeur avant sa multiplication par $t3.

Ordonnancement des instructions

Les instructions machines se décomposent en des opérations encore plus élémentaires (micro-instructions) exécutées par des unités de calcul du processeur.

- Parallèlisme
- Pipelining
- Scheduling

  Les instructions sont réordonnées pour

  - réduire les temps d'attentent, en lançant certains calculs plus tôt.
  - éviter des incohérences liés au pipelining.

    (Parfois, il est nécessaire des instruction de délai (qui ne calculent pas).

Ordonnancement des instructions

Les instructions machines se décomposent en des opérations encore plus élémentaires (micro-instructions) exécutées par des unités de calcul du processeur.

- Parallèlisme
- Pipelining
- Scheduling

  Le scheduling peut être pris en compte par l'assembleur (ordonnancement statique) ou directement par le processeurs (ordonnancement dynamique).

  En effet, le meilleur ordonnancement peut dépendre de paramètres dynamiques tels que la localisation du code dans le cache, ce qu'un ordonancement dynamique ne peut pas prévoir

Ordonnancement des instructions

Les instructions machines se décomposent en des opérations encore plus élémentaires (micro-instructions) exécutées par des unités de calcul du processeur.

- Parallèlisme
- Pipelining
- Scheduling

Gestion mémoire

Glanage des Cellulues (GC) encore appelé ramasse miettes.

La mémoire se présente comme un graphe orienté donc les nœuds sont des blocs mémoire de longeur variable et les arcs sont des pointeurs d'un bloc vers un autre.

À un instant particulier du calcul, la mémoire est accessible par le processeur à partir d'un ensemble de points d'entrée appelés les racines : ceux-ci incluent les registres du processeur, la pile d'éxécution, et les variables globales. (On ne connait pas toujours l'ensemble exact des racines, mais mais un sur-emsemble qui contient au moins tous les points d'entrée possibles).

La mémoire vivante à un instant donné est la partie de la mémoire atteignable à partir des racines en suivant tous les arcs possibles. La partie qui n'a pu être atteinte est définitivement déconnectée des racines et peut être recyclée.

Les différents types de GC

- Glanage à compteurs de références.

Les différents types de GC

- Glanage à compteurs de références.
  - On maintient en têtre de chaque bloc dans un compteur le nombre total de fois que ce bloc est référencé (par un autre bloc ou par une racine du GC). Lorsque que le bloc n'est plus référencé il n'est définitivement plus accessible et peut être recyclé.

Les différents types de GC

- Glanage à compteurs de références.
  - Il faut instrumenter le compilateur pour insérer des instructions de comptage durant l'exécution (la copie d'une adresse d'un registre dans un autre, la mise en pile, etc. modifie le nombre de références).
  - Ces instructions peuvent devenir coûteuses (gestion proportionnelle au parcourt de la taille de la mémoire).
  - De plus, le GC à compteur de référence n'est pas capable de récupérer les cycles (qui se référencent réciproquement sans être référencé de l'extérieur).

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
  - Lorsqu'il n'y a plus de mémoire disponible, on parcourt la mémoire à partir des racines et on marque la partie parcourue, puis on balaye toute la mémoire et on récupère les parties non marqueés.
  - Le temps d'un GC est proportionnel à l'espace alloué. Les données vivantes ne sont pas déplacés (fragmentation possible).

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
- Stop and copy.

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
- Stop and copy.
  - On sépare la zône mémoure entre une zône de travail et une zône vierge. Lorsqu'il n'y a plus de mémoire, on parcourt la mémoire à partir des racine et on la recopie dans la zône vierge (il faut préserver sa structure de graphe). La zône vierge devient la zône de travail et inversement.
  - Le coût est proportionnel à l'espace vivant. C'est donc très efficicace pour des allocations de durée de vie très courte. Les données vivantes sont recopiées (pas de compaction).

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
- Stop and copy.
- Stop and copy à générations.

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
- Stop and copy.
- Stop and copy à générations.
  - Après un GC majeure, on sépare considère la zône vierge libre comme une mémoire de génération mineure que l'on coupe en deux et sur laquelle on peut implanté un GC mineure (qui ne parcourt que la zône mineure, donc que les données fraîches). Quand la zône mineure devient trop dense, on fait un GC majeure et on recommence.
  - En première approximation, les données majeures sont plus persistantes (peu récupérables) alors que les données mineures sont éphémères. Le GC mineure est donc très rapide (en supposant qu'il récupère presque tout).

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
- Stop and copy.
- Stop and copy à générations.
- Glanage incrémental

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
- Stop and copy.
- Stop and copy à générations.
- Glanage incrémental

  on essaye de réduire le temps de latence dû au GC en alternant calcul et recopie de la mémoire de la zône de travail dans la zône vierge.

Les différents types de GC

- Glanage à compteurs de références.
- Mark and sweep.
- Stop and copy.
- Stop and copy à générations.
- Glanage incrémental

Optimizations fréquentes

- Mise en ligne.
  - L'appel aux petites fonctions auxilliaires est remplacé par leur corps.
  - Élimine un appel à une fonction, augmente la vitesse d'exécution, mais aussi pour de très petites fonctions, la taille du code produit.
  - L'utilisation de petites fonctions est très fréquente dans les langages fonctionnels.

    De façon général, il est bon d'encourager la définition de fonctions auxilliaires, qui rendent le code plus lisible, sans diminuer l'efficacité.

- Déroulement des boucles.

Optimizations fréquentes

- Mise en ligne.
  - La mise en œuvre est facile (Renommer les variables)
  - Le gain est significatif.
  - Rend les optimisations inter-procédurale moins nécessaires.

Optimizations fréquentes

- Mise en ligne.
- Appels terminaux.
  - Dans une fonction f, un appel à une fonction g est terminal si au retour de g la fonction f retourne immédiatement.
  - On peut alors exécuter le postlude de f (ajustement de la pile) avant l'appel à g. Au retour de g, il n'y a plus besoin de repasser par f. La fonction f peut donc faire un appel à g sans retour, laissant g retourner directement là ou f devait retourner.
  - Le gain est un retour plus rapide mais la libération anticipée de l'espace en pile utilisé par f avant l'appel à g.
    Cela permet des appels récursifs à une profondeur très grande sans risque de débordement de pile.

- Invariants de boucles.

Optimizations fréquentes

- Mise en ligne.
- Appels terminaux.
- Appel récursifs terminaux.

  Dans le cas particulier des appels récursifs terminaux, de plus, le postlude de f va être suivi du prélude de f, on peut court-circuiter (i.e. calculer la composition du postlude et du prélude) et se brancher après le prélude.

  Cela évite l'ajustement de la pile et les sauvegardes des registres callee-save et en pratique rend la récusion terminale équivalente à une boucle while.

Optimizations fréquentes

- Mise en ligne.
- Appels terminaux.
- Appel récursifs terminaux.
- Propagation des constantes.
  - Les constantes sont utilisées pour régler certaines valeurs par défaut. Une constante est une variable qui a la même valeur dans tout les programme. D'autres variables peuvent se retrouver dans la même situation.
  - On peut alors effectuer certains calculs, voir certains branchements à la compilation.
    Un exemple typique est une variable debug. Lors que le programme est compilé sans l'option debug, on veut éliminer les branches mortes (réserver à la mise au point).

Optimizations fréquentes

- Mise en ligne.
- Appels terminaux.
- Appel récursifs terminaux.
- Propagation des constantes.
- Invariants de boucles.
  - Un invariant de boucle est un calcul à l'intérieur d'une boucle qui est inchangé à chaque itération.
  - Un tel calcul peut être effectuer une seule fois avant d'entrer dans la bouche.

Optimizations fréquentes

- Mise en ligne.
- Appels terminaux.
- Appel récursifs terminaux.
- Propagation des constantes.
- Invariants de boucles.
- Déroulement des boucles.

  On peut dérouler les boucles, de petites tailles pour réduire le coût des sauts (deviennent moins fréquents) ou pour aligner des caractères sur des mots.

Optimizations fréquentes

- Mise en ligne.
- Appels terminaux.
- Appel récursifs terminaux.
- Propagation des constantes.
- Invariants de boucles.
- Déroulement des boucles.