Optimisations de code dans GCC

Pop Sébastian

July 26, 2002

1 Introduction

Ce rapport montre qu'il est possible d'implémenter des optimisations sur les représentations de haut niveau de GCC. Ce travail préalable a contribué à mettre en place une représentation intermédiaire *Simple* au niveau des arbres syntaxiques. *Simple* sera une représentation commune à tous les front-ends sur laquelle seront implémentées les optimisations et les analyses de haut niveau.

Une passe de normalisation du flot de contrôle a été implémentée sur cette représentation intermédiaire. Cependant les analyses interprocédurales sont très difficilement adaptables à l'architecture de GCC. Une solution possible est décrite dans ce rapport.

Le projet GCC a été créé en 1987 par Richard Stallman. Ce projet a été développé par des particuliers, mais aussi par des entreprises comme RedHat, Codesourcery, Apple, ou HP. En 2002 ce projet dépasse le million de lignes de code, génère du code pour 43 architectures différentes et possède 8 front-ends : C, C++, ObjectiveC, Java, Ada, Fortran77, Mercury, Chill. Les optimisations sont rassemblées dans la partie commune du compilateur de sorte à optimiser tous les front-ends.

GCC propose un nombre important d'optimisations. Cependant l'évolution de GCC est lente et son architecture est orientée vers la portabilité plutôt que vers la performance.

2 Architecture de GCC

2.1 Front-ends, back-end

L'ensemble des compilateurs de GCC partagent une partie commune dans laquelle sont implémentées la plupart des optimisations : le back-end. Cette partie du compilateur manipule le source d'un programme de manière indépendante du langage compilé, mais dépendante de l'architecture de destination. Cette représentation intermédiaire est connue sous le nom de RTL (Register Transfer Language).

La première étape lors de la compilation d'une fonction construit une représentation d'arbre syntaxique (AST). Les front-ends de GCC partagent une structure commune d'AST. Cependant la sémantique des noeuds de l'AST peut être différente d'un front-end à l'autre : la sémantique des noeuds est documentée sous forme de commentaires dans les fichiers <*.def>. Comme aucune transformation n'a encore été appliquée sur les AST, cette représentation de haut niveau contient une description complète des informations portées par le programme d'origine :

- informations de typage
- accès explicite des membres des structures
- accès explicite des éléments dans les tableaux
- flot de contrôle
- flot de données

L'étape suivante consiste à traduire les AST sous forme de RTL. Pendant cette traduction une partie importante des informations portées par les arbres syntaxiques est perdue. Ainsi les optimisations qui se trouvent dans le back-end ne peuvent plus se baser sur la représentation qu'ils ont du programme pour valider leurs transformations.

La solution adoptée par GCC a été d'implémenter une partie des analyseurs dirrectement dans les front-ends et de propager l'information extraite jusqu'aux optimiseurs du back-end.

Mark Mitchell a implémenté de cette manière une passe qui détermine des ensembles d'alias, en se basant sur les types.

Une solution plus récente est d'implémenter les transformations et optimisations dirrectement dans les front-ends où l'information disponible est exacte. Cependant sans une représentation commune, il serait nécessaire de réecrire les mêmes optimisations pour chaque front-end. Comme les AST partagent la même structure de données, il est possible de transformer les AST sous une unique forme intermédiaire : un sous ensemble de noeuds dont la sémantique est bien déterminée. Les avantages d'une telle représentation sont :

- représentation commune à tous les front-ends
- analyse des informations simplifiées
- les informations portées par les AST sont gardées intactes

Cette représentation intermédiaire a été étudiée et implémentée par l'université McGill de Montreal dans le compilateur McCAT. Cependant ce compilateur est resté privé et seuls les rapports de recherche [?] sont disponibles dans le domaine public.

La prochaine section décrit de manière générale comment sont implémentés les arbres syntaxiques, puis nous verons pourquoi la représentation intermédiaire *Simple* est nécessaire à l'analyse systématique des expressions et finalement nous verons comment les informations contenues dans les AST peuvent être rafinnées, plus particulièrement comment obtenir des structures régulières de flot de contrôle.

3 Les arbres syntaxiques

La structure d'arbre syntaxique est accessible en utilisant une interface de macros. Puisque les structures de données ne sont jamais accédées dirrectement par le programmeur, elles peuvent être changées de manière beaucoup plus souple : la seule contrainte est de garder une implémentation similaire à celle d'origine. Ainsi il est possible d'ajouter en plus de l'accès aux structures un système de vérification par typage des noeuds de l'AST. L'implémentation des arbres n'est pas changée, et donc c'est une implémentation valide pour les arbres. Sur les versions stables (de production) de GCC, cette vérification est supprimée pour augmenter la vitesse de compilation.

3.1 Interface des AST

Cette interface est décrite dans la page info de gcc dans la section *Trees*. Dans ce rapport je décris un simple exemple d'utilisation de cette interface pour modifier l'AST.

Supposons qu'on veut ajouter une instruction qui multiplie la valeur de i par 7. Cette multiplication doit être placée après une instruction sur laquelle on a un pointeur : *prev*.

La première étape consiste à construire la nouvelle instruction :

```
{
  tree stmt, modify, mult, sept, next;
  sept = build_int_2 (7, 0);
  mult = build (MULT_EXPR, integer_type_node, i, sept);
  modify = build_modify_expr (i, NOP_EXPR, mult);
  stmt = build_stmt (EXPR_STMT, modify);
```

Dans une deuxième étape il suffit d'insérer cette nouvelle instruction dans l'AST:

```
  next = TREE_CHAIN (prev);
  TREE_CHAIN (prev) = stmt;
  TREE_CHAIN (stmt) = next;
}
```

4 Simple

L'ensemble des expressions autorisées pour écrire un programme C est bien trop important pour pouvoir traiter de manière systématique l'ensemble des cas. L'analyse et les transformations deviennent possibles si on restreint le nombre d'expressions à un ensemble minimal, et si la composition d'expressions est limitée. La transformation vers cet ensemble minimal est effectuée sans pertes d'information : la sémantique du programme d'origine est conservée.

4.1 Combiner les expressions

La grammaire du langage C permet la combinaison d'expressions. De cette manière le programmeur n'est pas contraint à une syntaxe stricte propre à l'analyse d'un compilateur. Ainsi le nombre d'expressions possibles dans une telle grammaire est bien trop important pour leur analyse systématique dans un compilateur.

Un simple exemple consiste à imaginer un langage qui fournit dix expressions élémentaires et toute combinaison d'expressions élémentaires. Si on restreint le nombre de combinaisons possibles dans un statement à trois, le compilateur devrait gérer l'ensemble des cas correspondant aux expressions élémentaires (10 cas), à la combinaison de deux expressions élémentaires (100 cas), et la combinaison de trois expressions élémentaires (1000 cas), ce qui fait 1110 cas à traiter.

4.2 Simplifier les expressions

Puisque l'évaluation d'une expression se fait selon un ordre précis décrit dans la spécification du langage C, il est possible de décomposer une expression en expressions élémentaires en tenant compte de cet ordre implicite d'évaluation. La suite d'instructions qu'on obtient décrit le même programme, mais l'ordre d'execution est maintenant explicite.

Il est possible d'adapter la grammaire de *Simple* de sorte à faciliter le travail des analyseurs et des optimiseurs.

4.3 Une grammaire simplifiée : *Simple*

Cette grammaire a été construite principalement pour supporter le langage C, puisque le but de l'université McGill était de construire un compilateur optimiseur C. Cependant il est possible d'étendre *Simple* de sorte à supporter d'autres langages de programmation : dans une première étape *Simple* sera adapté pour satisfaire le C++ qui est très proche de la grammaire du C, puis le java et le fortran.

5 Vers un flot de contrôle régulier

Une fois que les AST des front-ends ont étés ramenés à la même représentation intermédiaire, il est possible d'effectuer des analyses et des transformations qui optimisent les AST. La première passe d'optimisation que j'ai implémenté sur *Simple* est la normalisation du flot de contrôle. Je décrirai brièvement dans cette section l'élimination des break, continues et des goto. Cette implémentation se base sur le travail de recherche de l'université de McGill. Les publications [**?**] [**?**] contiennent une description complète des algorithmes pour l'élimination des gotos. Dans cette section je décrirai seulement les changements apportés à ces algorithmes.

5.1 Elimination des break et continue

Un BREAK_STMT se réfère à la structure de boucle ou de switch qui le contient. Pour l'éliminer, on introduit une variable logique qui prend la valeur *vrai* lorsque le flot de contrôle atteint le BREAK_STMT. Puis des transformations élémentaires sont appliquées récursivement jusqu'à ce que le BREAK_STMT arrive au niveau du corps de boucle auquel il se refère. Une fois à ce niveau, la condition de boucle est modifiée de sorte à ce que son execution s'arrête dès que la variable logique devient *vrai*.

L'élimination des CONTINUE_STMT se fait sur le même principe, mais l'execution se poursuit sur l'itération suivante.

Lors de l'élimination de CONTINUE_STMT et de BREAK_STMT il est possible de faire de l'élimination de code : *dead code elimination*. L'unique contrainte est que le code qui suit le break/continue inconditionnel ne comporte pas de point d'entrée : qui peut être par exemple un LABEL_STMT.

5.1.1 Exemple d'élimination de break

```
{
  while (k != 0)
    {
      k++ ;
      if (j != 0)
        break;
      i++ ;
    }
}
---- Break/Continues :
{
  int T.1 = 0;
  while (T.1 == 0 && k != 0)
```

```
    {
      k++ ;
      if (j != 0)
        {
          T.1 = 1;
        }
      if (T.1 == 0)
        {
          i++ ;
        }
    }
  T.1 = 0;
}
```

5.1.2 Exemple d'élimination de continue

```
{
  while (k != 0)
    {
      k++ ;
      if (j != 0)
        continue;
      i++ ;
    }
}

---- Break/Continues :
{
  while (k != 0)
    {
      int T.1 = 0;
      k++ ;
      if (j != 0)
        {
          T.1 = 1;
        }
      if (T.1 == 0)
        {
          i++ ;
        }
    }
}
```

5.1.3 Exemple de dead-code elimination

```
{
  while (k != 0)
    {
      k++ ;
      continue;
      i++ ;
    }
}

---- Break/Continues :
void tinytoons (int j, int i, int k)
{
  int T.1 = 1;
  while (k != 0)
    {
      k++ ;
    }
}
```

5.2 Elimination des goto

L'élimination des GOTO_STMT telle que je l'ai implémentée dans GCC a été décrite et implémentée par Ana Erosa [?] [?] dans le compilateur McCAT. Une différence notable par rapport à l'implémentation décrite dans ces publications est la transformation des breaks en contrôle dirigé par variables logiques, ce qui complète la méthode décrite par Ana Erosa. De plus le problème de la capture de break lors de la transformation d'un goto vers une boucle do/while disparait.

Une autre extension possible serait de ne pas autoriser de switch dans la représentation *Simple* sur laquelle l'élimination de goto est apliquée, ce qui facilite le travail d'implémentation. C'est d'ailleurs en grande partie la raison pour laquelle on a adopté *Simple* : simplifier le travail de transformation et d'analyse de code.

5.2.1 Exemples d'élimination des gotos

L'exemple suivant élimine le goto en le transformant en un *if*

```
{
  goto L1;
  j-- ;
L1:;
}
```

```
---- GOTO elimination :
{
  int T.1 = 0;
  if (1 != 0)
    {
      T.1 = 1;
    }
  if (T.1)
    {
    }
  else
    {
      j-- ;
    }
L1:;
  T.1 = 0;
}
```

Si l'offset du label est inférieur à l'offset du goto, dans ce cas le goto est éliminé par une boucle while. L'exemple suivant illustre ce cas.

```
{
L1:;
  j-- ;
  goto L1;
}

---- GOTO elimination :
{
  int T.1 = 0;
  do
    {
      T.1 = 0;
    L1:;
      j-- ;
      if (1 != 0)
        {
          T.1 = 1;
        }
    }
  while (T.1);
}
```

5.3 Pourquoi éliminer les gotos?

Les gotos sont une forme d'expression très compacte pour spécifier un flot de contrôle complexe. Ce genre de contrôle de flot a soulevé une énorme vague de commentaires de la part des chercheurs en algorithmique comme Dijkstra [?] qui considéraient ce genre de programmation dangereuse.

Si on analyse la situation aujourd'hui on s'appercoit que la plupart des programmeurs n'utilisent plus de gotos dans leurs programmes.

Cependant des outils de génération automatique d'automates à états finis (comme les parseurs, ou les automates de vérification de protocoles) utilisent les gotos pour simuler les transitions entre les états.

GCC comme beaucoup d'autres compilateurs ne sait pas traiter correctement et optimiser les programmes contenant des gotos. C'est la raison pour laquelle une passe d'élimination des gotos est nécessaire pour la normalisation du flot de contrôle.

5.4 Pourquoi éliminer les breaks et continues?

L'élimination des breaks et continues est déjà plus spécifique à l'architecture de GCC, puisque d'autres compilateurs arrivent à traiter correctement les breaks et les continues. Le problème de l'architecture de GCC est que pendant la traduction des AST vers le RTL, les breaks et les continues sont traduits en gotos locaux à la boucle, ce qui fait que les passes d'optimisation dans le back-end ne peuvent plus distinguer un break d'un goto.

6 Vers l'analyse des dépendances

6.1 Le problème des alias

Prenons une fonction simple:

```
int simple(int* A, int* B )
{
  int i;
  for (i=0; i<N ; i++)
    A[i] = B[i];
}
```

A première vue on se trouve devant un cas simple : les tableaux A et B sont distincts, et par consequent il n'y a pas de dépendances ...

Mais que se passe-t-il quand cette fonction est appellée par:

```
int A[100];
simple (A, A+1);  // crée une anti-dépendance
simple (A, A-1);  // crée une dépendance de flot
```

Ce genre de dépendances ne peut être découvert que par une analyse interprocédurale des alias. Cette analyse propage l'information des *[may|must]-points-to* dans le graphe d'appel des fonctions. En absence d'une telle analyse, le compilateur doit considérer le cas le plus pessimiste, qui est A et B apartiennent au même ensemble d'alias.

Une autre solution à ce probleme serait d'inclure le corps de la fonction *simple* au niveau de l'appel : c'est ce qu'on appelle l'*inlining*. Cependant cette technique augmente la taille du code et par consequent ne peut être appliquée qu'à un nombre restreint de fonctions.

6.2 Analyses interprocédurales dans GCC

Gcc n'a pas d'analyses interprocédurales puisqu'il se base sur le modèle du traitement d'un seul fichier à la fois, ce qui restreint la visibilité des fonctions au fichier courant. Ainsi il ne peut réaliser qu'un nombre très réduit d'optimisations interprocédurales comme l'inlining de fonctions.

Une solution à ce problème serait un linker au niveau des codes sources (ce qui a été implémenté dans le compilateur McCAT à l'université de McGill). L'analyse interprocédurale commence par la génération complète des AST. Cependant produire une telle représentation en mémoire sur un projet de grande taille (comme GCC ou le kernel Linux) demande énormément de mémoire, ce qui peut sembler prohibitif sur les architectures actuelles.

Une autre solution serait de récupérer les informations nécessaires à l'analyse interprocédurale dans une première passe sur l'ensemble du projet, puis se basant sur cette information d'optimiser le code en une deuxième passe. Ainsi on peut éviter la construction complète des AST, et ne produire que l'information essentielle permettant de décider l'ensemble des fonctions candidates pour l'inlining, ou les informations d'alias permettant une analyse plus précise des accès mémoire. Là encore la forme et le contenu de la base de connaissances peut être critique pour l'efficacité des optimisations.

7 Conclusion

GCC est un compilateur qui possède beaucoup d'optimisations implémentées dans le back-end. Cependant l'infrastructure pour l'analyse des dépendences était inexistante. Ce projet a contribué la représentation intermédiaire *Simple* , la normalisation du flot de contrôle, le commencement de l'analyse externe à GCC du graphe d'appel, et le module d'affichage des arbres sous forme de programme C.

L'une des plus grandes récompenses de ce travail a été de suivre la communauté des développeurs GCC dans son évolution pendant près d'une année, et d'apprécier la qualité du travail qu'elle produit.

8 Appendix A : Une journée pour hacker GCC

Si vous souhaitez participer au développement de GCC, voici en quelques étapes les premiers pas.

8.1 Trouver le code source

```
> export CVSROOT=:pserver:anoncvs@subversions.gnu.org:/cvs/gcc
> cvs -z9 co -r AST-optimizer-branch gcc
```

8.2 Design

C'est la partie la plus difficile puisqu'il faut essayer d'adapter son algorithme aux passes qui existent déjà, et pour cela il faut avoir une vue globale de comment GCC fonctionne. Si vous avez des difficultés pour commencer, une première étape consiste à regarder la page info de gcc (la version la plus récente se trouve dans le code que vous avez chargé dans le répertoire gcc/gcc/doc/) Si vous n'avez toujours pas de réponse à la question après avoir cherché dans tous les manuels, il existe deux solutions : soit vous vous lancez dans le code source et essayez de trouver la réponse par vous même, soit vous posez la question sur la mailing list de gcc 'gcc@gcc.gnu.org', mais là une seule contrainte : la question doit être très précise. De plus vous n'êtes pas sûr que quelqu'un a la réponse à votre question : et dans ce cas la première solution s'impose, trouver la réponse dans le code.

8.3 Implémenter

Là encore c'est une partie difficile puisque les règles de codage sont assez strictes. Ces règles ont étés adoptées pour unifier le style de codage, et pour faciliter la maintenance du source. Ces règles tendent à devenir de plus en plus strictes, et c'est en faveur de la lisibilité du code source.

Par exemple toutes les fonctions doivent comporter un commentaire qui décrit la fonction. Ce genre de commentaire est d'une utilité inestimable quand on travaille avec les tags.

8.4 Bootstrap

Une fois que votre optimisation compile bien et que vous avez testé votre optimiseur, une dernière vérification s'impose : bootstrapper le compilateur.

```
> make bootstrap BOOT_CFLAGS="-g -fmon-optim"
```

- stage1 : La première étape consiste à créer le compilateur que vous avez modifié : appelons-le comp1. Ce compilateur est créé avec les moyens du bord, en general avec le compilateur installé sur votre machine et qui répond au nom de 'cc' : appelons-le comp0.

- stage2 : La deuxième étape construit un compilateur comp2, avec le compilateur comp1. Le compilateur comp2, a été optimisé avec votre optimiseur.
- stage3 : La troisième étape construit un nouveau compilateur comp3, avec le compilateur optimisé.

Pour résumer,

- comp0 : le compilateur local,
- comp1 : gcc avec l'optimiseur,
- comp2 : gcc optimisé,
- comp3 : gcc construit par le gcc optimisé.

La dernière étape consiste à comparer le code produit par comp1 et par comp2 appelés avec les mêmes options. Si les deux compilateurs produisent le même code, cela veut dire que le compilateur optimisé (comp2) produit le même code que le compilateur non optimisé (comp1), et donc utiliser le compilateur optimisé (comp2) ou le compilateur non optimisé (comp1) n'a pas d'influence sur le code que le compilateur produit.

8.5 Pour finir

Une fois l'optimiseur bootstrappé, vous pouvez creer le patch :

```
> cvs update
> cvs -z9 diff -dup ./gcc/
```

puis éditer l'entree dans le Changelog et finalement envoyer le patch sur la mailing list 'gcc-patch@gcc.gnu.org'.

9 Appendix B: SIMPLE C Grammar

La grammaire suivante a été écrite par Diego Novillo. Cette grammaire possède quelques règles qui ne correspondent plus à la version originale de McGill.

Il serait aussi intéressant de réduire encore cette grammaire de sorte à n'avoir plus qu'un seul type de structure pour :

- la sélection : un IF_STMT,
- la répétition : un FOR_STMT.

On évite ainsi d'avoir des structures compliquées à traiter par la suite, comme les SWITCH_STMT. On évite aussi les différentes sémantiques des structures de boucles.

```
The BNF rules for SIMPLE C

  Statements

  all_stmts
  : stmtlist stop_stmt
  | stmtlist

  stmtlist
  : stmtlist stmt
  | stmt

  stmt
  : compstmt
  | expr ';'
  | IF '(' condexpr ')' stmt
  | IF '(' condexpr ')' stmt ELSE stmt
  | WHILE '(' condexpr ')' stmt
  | DO stmt WHILE '(' condexpr ')'
  | FOR '('exprseq ';' condexpr ';'exprseq ')' stmt
  | SWITCH '(' val ')' casestmts
  | ';'

  compsmt
  : '{' all_stmts '}'
  | '{' '}'
  | '{' decls all_stmts '}'
  | '{' decls '}'
```

Declarations

All the possible C declarations. The only difference is that in SIMPLE the declarations are not allowed to have initializations in them.

Expressions

```
exprseq
: exprseq ',' expr
| expr

stop_stmt
: BREAK ';'
| CONTINUE ';'
| RETURN ';'
| RETURN val ';'
| RETURN '(' val ')' ';'

casestmts
: '{' cases default'}'
| ';'
| '{' '}'

cases
: cases case
| case

case
: CASE CONST':' stmtlist stop_stmt

default
: DEFAULT ':' stmtlist stop_stmt

expr
: rhs
| modify_expr

call_expr
: ID '(' arglist ')'

arglist
: arglist ',' val
| val
```

```
modify_expr
: varname '=' rhs
| '*' ID '=' rhs

rhs
: binary_expr
| unary_expr

unary_expr
: simp_expr
| '*' ID
| '&' varname
| call_expr
| unop val
| '(' cast ')' varname

(cast here stands for all valid C typecasts)

binary_expr
: val binop val

unop
: '+'
| '-'

binop
: relop
| '-'
| '+'
....

relop
: '<'
| '<='
...

condexpr
: val
| val relop val

simp_expr
: varname
| CONST
```

```
   val
   : ID
   | CONST

   varname
   : arrayref
   | compref
   | ID

   arrayref
   : varname reflist => Original grammar only allowed
                               'ID reflist'.  This causes weird
splits like k.d[2][3] into

 t1 = &(k.d[2])
t2 = *(&t1[3])

   reflist
   : '[' val ']'
   | reflist '[' val ']'

   idlist
   : idlist '.' ID
   | ID

   compref
   : '(' '*' ID ')' '.' idlist
   : '&' ID '.' idlist => Original grammar does not allow
this.
   | idlist
```

10 Appendix C: Info page de *Simple*

10.1 A reduced syntax for Simple

Simple is a subset of the original syntax of the language. The simplify pass constructs a semantic equivalent program based on a minimal subset of the language syntax.

For example the post increment allowed by the C grammar can be very well transformed in an affectation and an addition of one, without any loss of information.

The set of syntax operators is different from a language to another, and worse, they can have for a same syntax operator different semantics. The simplify pass consists of factoring all these differences in a common subset of primitives on which we'll be able to perform analyses and optimisations.

For example, the C++ language defines a mechanism to handle exceptions, but this control flow mechanism is not present in the C language.

An interesting thing to be noted in this example is that the exception mechanism can be simulated in C by another set of syntactic expressions. That means that the C++ throw can be handled by a smaller syntax set that belongs to the C language. However in order to apply such a transformation, we need to prouve that the program has not lost information.

10.2 What are the benefits of adopting Simple?

From the simplification pass we obtain a program that has the following features:

- Control flow is structured and explicit. The number of loop primitives is reduced. The control flow is simplified by suppression of breaks, continues and gotos. This allows much more optimisations on loops since their structure becomes more regular.

- Accesses to structures and to arrays are retained and not broken to lower level statements. Thus, informations like the size of arrays and types is not lost. This allows the alias and dependence analysis to work properly on Simple trees. Function accesses to arrays is also simplified avoiding situations where these functions have side effects. A later pass reconstruct from Simple trees the correct access functions in function of loop indices. This allows the dependence pass to work on more cases.

- Data flow structure is not modified. The order of expression evaluations is not modified by the simplification pass. This is a crucial issue in the validation of our transformations, since floating point arithmetic depends on it.

10.3 Work state

10.3.1 What Simple does for the moment?

The only transformations we implemented up to now are based on the C/C++ front-ends, and they perform :

- Arithmetic simplifications.
- Pointer simplifications.

The next step is to reduce the number of loop structures to a minimal set : that is almost implemented to reduce all loop structures to a for statement, but in the case of a do_stmt we have to restructure loop indices in order to avoid to duplicate the loop body. Thus this simplification will be done after the retrieval of the loop indices and their normalisation (here normalisation means : a single induction variable by loop, if there is one; a step of one, and the induction variable starts at zero.)

A further step will be to extend the Simple representation to other front-ends like Java and Fortran. This should avoid the rewriting of analyses and optimisations performed on the front end. This step is not yet implemented, but the way it will be designed will influence optimisations implementation.

10.3.2 Future work

Further analyses and optimisations :

- Search of loop induction variables.
- Normalisation of array access functions.
- Construction of the loop dependence graph : 'dependence.c' to be readapted in order to take the results from tree-cfg (control flow graph), tree-ssa (static single assignment) and from alias analysis.
- Goto, break and continue elimination by transformation of the loop structure.
- Loop transformations for temporal locality of data references.

11 Appendix D: Ce qu'il reste à faire

Il y a enormément de passes qu'il reste à implémenter. Comme les choses changent très vite un fichier qui référence les prochaines étapes à effectuer sera mis en place sur le site de gcc : http://gcc.gnu.org . Voici la version d'origine.

11.1 AST-optimizer-branch TODO List

This list references a list of mini projects under the form of a stack : the first things on the stack are the most important to be done before going deeper.

- *Search of loop induction variables* : This could be implemented using Michael Wolfe's algorithm : Beyond Induction Variables. This algorithm is implemented in the Nascent Compiler.
- *Normalization of array access functions :* This pass should gather the information decomposed by Simplify into well formed access functions. A further simplification should simplify these functions in order to get functions of loop indices. This form is needed in dependence tests. The implementation of this normalization is very similar to the search of loop induction variables by following use-defs chains and reconstructing the corresponding tree. Arithmetic simplification of expressions can be performed since we work on integers. The only constraint is to keep in mind that overflow can occur. (I don't think that those who use overflow opertions on array accesses will use our dependence tester and optimisations, but we have to be conservative...)
- *Loop canonicalization :* This pass should construct Fortran like loops : a single induction variable by loop, the iteration begins at zero, and a step of one. If a loop has more than one induction variables, then it is possible to choose one of them, and express other variables in function of the first one.
- *Dependence graph :* This pass is almost implemented in dependence.c but is not used. We can adapt this work by keeping the dependence tester, but get the information about array subscripts and loops from the Simple representation.
- *Temporal locality of data references :* This constructs a geometrical representation of the order in which data sets are accessed. It transforms loop iterations in order to keep data references in caches. The algorithm is described in publication papers and implemented on a source to source transformer for Fortran.

References

[1] http://www-acaps.cs.mcgill.ca/info/McCAT/McCAT.html

[2] Taming Control Flow : A structured Approach to Eliminate Goto Statements, Ana M. Erosa and Laurie J. Hendren, in the proceedings of the 1994 International Conference on Computer Languages (ICCL'94) pp.229-240. ftp://ftp-acaps.cs.mcgill.ca/pub/doc/papers/ICCL94.ps.gz

[3] Erosa's MSc. Thesis, University McGill : A goto-elimination method and its implementation for the McCAT C compiler, http://www.sable.mcgill.ca/h̃endren/ftp/erosa/thesis.ps.gz

[4] Goto statement considered harmful. In the Communications of the ACM. 11:147-148 , March 1968.