Les greffons du compilateur GCC : Votre compilateur GCC sur mesure!

Zbigniew CHAMSKI [1]   Basile STARYNKÉVITCH [2]

zbigniew.chamski@gmail.com   basile@starynkevitch.net

Infrasoft IT Solutions

17 mars 2010 / Solutions Linux - svn `$Revision: 33 $`

1. avec le soutien d'INRIA/Alchemy
2. basile.starynkevitch@cea.fr

Plan

1 Introduction
- Les greffons dans un compilateur
- Le défi des compilateurs : la complexité
- Etendre GCC - pourquoi ?

2 Un dilemme pour motiver
- Choix des paramètres de compilation du noyau Linux

3 Interfaces de GCC
- Représentations internes
- Passes

4 Greffons simples
- Interfaces de programmation des greffons dans GCC
- Interface directe
- Interface abstraite MILEPOST

5 Greffons pour les tâches complexes : l'outil MELT
- motivations, contexte, objectifs de MELT
- Traits du langage MELT
- codez vos modules en MELT

6 Pour conclure. . .

Avertissement

Avertissement important

Les **opinions** exprimées ici sont **seulement celles des auteurs** et pas celles de leur employeur, clients, de la communauté GCC ou autre . . .

La version finale des transparents est disponible sur `http://starynkevitch.net/Basile/` ou `http://gcc.gnu.org/wiki/Plugins/`

Introduction

Public attendu

1. utilisateurs habituels du compilateur libre GCC sous Linux
2. développeurs seniors de logiciel
3. responsables de projet logiciel compilé avec GCC
4. responsables qualité, outillage, méthodologie, . . . de logiciels importants

Pas de connaissance préalable requise en théorie de la compilation.

Un sondage pour démarrer

- Combien d'entre vous ont compilé (ou fait compiler) GCC ? Le noyau Linux ?
- Combien d'entre vous ont modifié (ou fait modifer) GCC ? Le noyau Linux ?
- Combien d'entre vous sont confrontés à des projets logiciels de (très) grande taille - 1, 10, 100 millions de lignes de code ?
- Combien d'entre vous travaillent sur des systèmes fortement contraints (embarqué, haute performance, fiabilité accrue, temps réel...) ?

Greffons dans un compilateur ? ! Pour quoi faire ? !

1. Accéder aux informations dont dispose le compilateur et :
   - combler les lacunes du compilateur
   - "détourner" le compilateur pour d'autres usages
   - essayer des idées nouvelles. . .
2. Réduire l'effort nécessaire à l'expérimentation avec GCC
   - limiter le seuil de connaissance minimum requise
   - éviter les longues recompilations
   - changer "juste les valeurs qu'il faut"

En d'autres mots, faire des choses nouvelles avec GCC sans avoir à le comprendre dans son auguste totalité de **4 millions** de lignes de code.

Greffons dans un compilateur ?

Les grandes caractéristiques :

- code *extérieur* au compilateur
- conditionné sous forme de *bibliothèque partagée*[3]
- *chargé à la demande* lors du lancement du compilateur
- invoqué dans des *situations spécifiques pendant la compilation*
- fournissant une **extension** : son absence ne gène pas le bon fonctionnement du compilateur

L'analogie avec les greffons d'un navigateur internet ou les modules du noyau Linux est grande. . .

3. fichier `.so` *"shared object"* chargé par `dlopen`

Greffons de GCC : une technologie en devenir

Une (r)évolution *en cours* dans GCC :

- 2009 : une branche de développement de GCC 4.4
- 2010 : partie intégrante de GCC 4.5
- accent sur l'**infrastructure**, pas sur les greffons eux-mêmes
- un *changement de la licence de GCC* a été nécessaire [4]
- *Peu* de greffons sont actuellement *disponibles* : DraggonEgg [5], MELT, MILEPOST/ICI, TreeHydra [6], . . .
- les greffons de GCC sont une innovation récente

4. licence `http://www.gnu.org/licenses/gcc-exception.html` motivée dans `http://www.gnu.org/licenses/gcc-exception-3.1-faq.html`
5. front-end GCC pour `llvm.org`
6. vérificateur pour Mozilla

Initialement, deux principales communautés impliquées :

- développeurs de GCC
  les greffons ont été poussés par les développeurs "utilisateurs avancés" (Google, IBM, . . .) de GCC, et non par les fabricants de processeurs
- chercheurs en compilation (académiques autant qu'industriels)

Le véritable potentiel des greffons se manifeste à l'usage - la troisième communauté = vous = *usagers avertis développant vos greffons. . .*

Un compilateur c'est. . .

Compilateur (*“compiler”*) =

- *programme* informatique
- *traduisant*
- un langage informatique *source* : C, C++, Java, Fortran, Ada, Objective-C, . . .
- en un autre langage *cible*
- en *préservant* ± la *sémantique*
- cibles de GCC : assembleurs x86, AMD64, ARM, PowerPC, Sparc, . . .

[d'après wikipedia]

Complexité et imperfection des compilateurs

Les compilateurs sont de plus en plus complexes, à cause de :

- la complexité grandissante des machines (ou des langages cibles)
- le fossé croissant entre langage source et cible
- l'évolution des spécifications de langages
- les exigences des utilisateurs
- le besoin croissant d'optimisations ou de diagnostics
- la loi de Moore (performance $\times 2$ tous les 18 mois) devient fausse

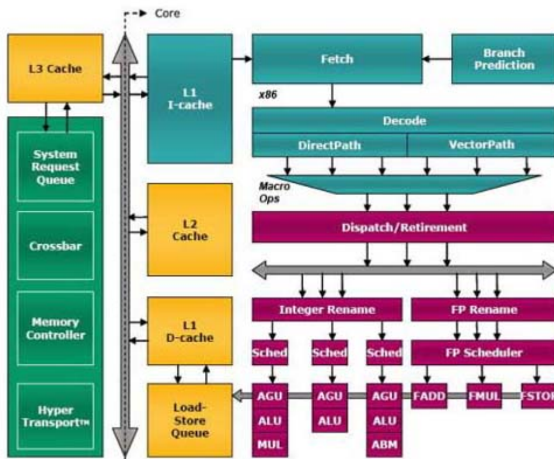
Donc **tout compilateur est imparfait**. Mais voir le projet *CompCert* `http://compcert.inria.fr/` de compilateur C certifié (prouvé par Coq).

Besoin de certification [7] des outils pour les logiciels critiques (avionique DOI178B ...). Mais n'*incriminez pas GCC pour **vos** bogues* !

7. Certaines versions de GCC dans des modes d'utilisation restreints sont certifiées, validées par leur popularité et des tests extensifs.

micro-architecture AMD Phenom II - 1 cœur (parmi 4)

`http://www.xcpus.com/Images/Docs/doc117/AMD-slide2.jpg`



Cache et mémoire

Les processeurs ont quasi-tous des caches mémoire

- accès $\approx$ 100 fois plus rapide aux données et instructions utilisées fréquemment
- taille et organisation varient fortement entre modèles et fabricants
- processeurs embarqués : parfois simplement 32ko I + 32ko D
- processeurs haut-de-gamme (exemple : AMD Phenom 2) :
  - L1 : 64ko I + 64ko D par cœur
  - L2 : 512ko par cœur
  - L3 : 6Mo partagé entre cœurs.

Les *défauts de cache* (*"cache miss"*) au dernier niveau sont *très pénalisants* : on aurait pu exécuter plusieurs centaines d'instruction machine.

Parallélisme interne à un cœur

Les processeurs sont *pipelinés* (travail à la chaîne) et *super-scalaires* (plusieurs unités d'exécution)

- impossible de prédire le temps de calcul d'une petite boucle
- souvent seule l'exécution permet de déterminer la performance

Des instructions machine consécutives gagnent à être indépendantes

- exécution en parallèle sur plusieurs unités de traitement d'un cœur
- ordre d'exécution déterminé lors de l'*ordonnancement* (*"scheduling"*) d'instructions
- tâche fastidieuse et difficile : parfaite pour les compilateurs.

exemple de code source et cible

La fonction suivante contient des sauts et des tests.

```
/* fichier som4.c */
long som4 (long t[]) {
  long s = 0;
  for (int i = 0; i < 4; i++)
    if (t[i] > 0)
      s += t[i];
  return s;
}
```

On peut la compiler avec
`gcc -S -O3 -std=gnu99 -fverbose-asm som4.c`
pour obtenir le fichier assembleur `som4.s`

```
        .type   som4, @function
som4:
        xorl    %edx, %edx              # s
        cmpq    $0, (%rdi)              #,* t
        movq    8(%rdi), %rcx           #, D.2699
        cmovns  (%rdi), %rdx            #* t,, s
        testq   %rcx, %rcx              # D.2699
        leaq    (%rdx,%rcx), %rax       #, s
        movq    16(%rdi), %rcx          #, D.2699
        cmovle  %rdx, %rax              # s,, s
        leaq    (%rax,%rcx), %rdx       #, s
        testq   %rcx, %rcx              # D.2699
        movq    24(%rdi), %rcx          #, D.2699
        cmovle  %rax, %rdx              # s,, s
        leaq    (%rcx,%rdx), %rax       #, s
        testq   %rcx, %rcx              # D.2699
        cmovle  %rdx, %rax              # s,, s
        ret
```

code *sans branchements*, avec chargements conditionnels `cmov`...

Complexité des compilateurs

Un compilateur se charge généralement de :

1. d'abord lire et analyser le texte du code source
2. construire une représentation arborescente - l'arbre de syntaxe abstrait (AST = *"abstract syntax tree"*).
3. transformer l'AST en d'autres représentations internes
4. optimiser ces representations internes
5. finalement émettre le langage cible (assembleur ou code machine).

Écrire un compilateur naïf est assez simple

- `tinycc` [37 KLOC environ] traduit *rapidement* "instruction par instruction" le C en instructions x86.
- pas d'optimisations

Plus qu'autrefois, il est essentiel pour les performances de bien optimiser le code source.

Mais **l'optimisation est un art difficile**[8] : il faut

- comprendre l'intention du programmeur
- comprendre les interactions dans le système complet
- trouver de judicieux compromis.

8. C'est un problème indécidable !

Etendre GCC - pourquoi ?

Deux tâches-clés durant le développement et l'intégration d'un système dominé par le logiciel :

- analyser / **comprendre** / modifier le **code source** du logiciel
- **optimiser** le processus et le résultat de la compilation

Ces tâches ne peuvent plus être réalisées manuellement si le code est “suffisamment” grand

- vœu pieux : si seulement le compilateur savait faire ça. . .

Un compilateur n'est jamais optimal :

- les configurations par défaut ne sont pas toujours les meilleures
- les mécanismes de contrôle (options, directives,. . .) sont insuffisants/inadaptés à votre cas
- il manque “juste la bonne transformation”
- les diagnostics sont inadaptés ou insuffisants. . .

Pourquoi faire vos greffons de GCC ?

Les greffons de GCC que **vous** développerez vous permetteront d'avoir un compilateur sur mesure pour :

1. mieux adapter le système à vos besoins et contraintes :
   - affiner l'optimisation pour votre système embarqué
   - gagner encore en performance ou en place donc . . .
   - diminuer vos coûts
2. appréhender et travailler plus efficacement sur votre code source
   - construire l'outillage
   - augementer la productivité des développeurs
   - faciliter l'utilisation de vos bibliothèques

GCC augmenté de votre greffon peut mieux compiler, ou faire tout autre chose. . .

Comment influencer le compilateur de manière "automatique" ?

Dans une invocation, par ordre de popularité :

- options de la ligne de commande
- paramètres internes du compilateur
- **problème :** application non sélective, à tout un module source

D'une invocation à l'autre

- résultats de profilage (nombre d'exécutions ou branchements, intervalles de valeurs)
- **problème :** information valable pour juste une exécution

Le potentiel : plus de 10% de performance à portée de main

Le vrai défi : adaptivité

Les greffons comme outil d'analyse

Le compilateur parcourt tout le code de l'application . . .
Il pourrait :

- vérifier le *respect des règles de codage* (même complexes)
- vérifier la *conformité du code à des spécifications* convenablement formalisées
- vérifier certaines *propriétés sémantiques* du code
- fournir des *diagnostics spécifiques* à un logiciel, un système, un projet, une entreprise
- obtenir des *métriques*, des *informations* du code *source*, ...

Les greffons et la programmation par aspects

Exemples d'aspects à explorer :

- instrumenter tous les appels `foo()` avec un argument positif
- suivi des allocations d'objects d'un type donné
- génération de code de sérialisation/desérialisation de données

Le compilateur avec le greffon peut disposer de l'information sur le programme et sur son environnement :

- optimisations de toute l'application (LTO)
- apprentissage machine pour le réglage de paramètres de GCC (MILEPOST, cTuning.org)
- meilleure exploitation des données de profilage (application, noyau ou les deux)

Desktop $\neq$ embarqué

Linux et GCC sont développés au quotidien sur les architectures x86

- grands caches (2 à 6 Mo) $\Rightarrow$ la taille du code cible peut croître "sans risque"
- le noyau tient dans les caches (au moins en grande partie)
- un noyau optimisé pour la vitesse est effectivement rapide

Sur une plateforme embarquée (PDA, téléphone portable, routeur ADSL. . .) ce n'est plus le cas :

- caches de petite taille (2 $\times$ 32 Ko) obligent à réduire la taille du code
- défaut de cache = ralentissement
- noyau optimisé pour la vitesse = jusqu'à 72% de cycles perdus dans les défauts de cache !

Question : quelle est la meilleure approche ?

L'art difficile du compromis

Solution idéale :

- optimiser la taille des fonctions qui coûtent cher en défauts de cache
- optimiser la vitesse des fonctions qui sont sur le chemin critique

Les défis :

- le compilateur sait faire l'un **ou** l'autre, mais pas les deux. . .
- comment choisir la bonne optimisation pour chacune des fonctions ?
- quels sont les bons critères de décision ?
- comment tester les différentes solutions sans recompiler GCC ?

La réponse de GCC 4.5 : déplacer cette fonctionnalité à l'extérieur de GCC, dans un **greffon** (*"plugin"*)

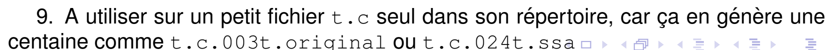
Les principales représentations internes de GCC

Pour un premier contact : dump des représentations internes

```
gcc -S -O3 -fdump-tree-all t.c
```
[9]

Le dump est une représentation *textuelle partielle* d'une représentation interne dans GCC.

9. A utiliser sur un petit fichier `t.c` seul dans son répertoire, car ça en génère une centaine comme `t.c.003t.original` ou `t.c.024t.ssa`



En **très gros** :

- représentations *Generic* dans les frontaux (propres à chaque langage source).
- représentations *Tree* des déclarations (et du code avant *"gimplification"*)
- représentations **Gimple** pour représenter le code dans le *"middle-end"* ; dont *Gimple/**Ssa*** *"Static Single Assignment"*
- représentations *Rtl* *"register transfer language"* dans le *"back-end"* spécifique à une cible.
- une foule d'autres structures de données et variables globales.

Gestion mémoire dans GCC

Plusieurs données sont gérées manuellement (`xmalloc/xfree`), mais :

Un **ramasse-miettes** Ggc *"Gcc Garbage Collector"* gère la plupart des données importantes (dont Gimple et Tree) :

- ramasse-miettes marqueur précis
- sert aussi aux *"Pre-Compiled Headers"*
- annotations `GTY` sur les `struct`-ures de données et les variables (statiques ou globales)
- utilitaire `gengtype` pour générer le code de marquage
- *ne traite pas les variables locales* : il faut copier les pointeurs de données à conserver
- Ggc est lancé entre les passes de compilation [10], pas implicitement par l'allocateur `ggc_alloc`

10. Ou plus rarement explicitement par `ggc_collect`

Pratiquement :

- seules des données transmises entre passes sont traitées par Ggc
- Ggc est de mon point de vue insuffisant, donc peu investi et trop peu utilisé
- les données internes à une passe sont gérées à la main
- certaines données sont allouées à la main, ou parfois invalides.
- (presque) pas d'organisation objet avec héritage
  $\Rightarrow$ on ne peut pratiquement pas définir des champs supplémentaires dans les données de GCC (il faut les associer ailleurs : hash-tables).
- quelques conteneurs génériques : vecteurs, hash-tables.

Les *Tree*-s

Représentation du code proche de *Generic* utilisée aussi pour les déclarations (dans Gimple). Fichiers **gcc/tree.def** gcc/tree.h gcc/tree.c
≈ 100 nœuds possibles :

- `IDENTIFIER_NODE` . . . pour les symboles
- `ENUMERAL_TYPE`, `ARRAY_TYPE` . . . pour les types ;
- `INTEGER_CST`, . . . pour les constantes ;
- `PARM_DECL`, `FIELD_DECL`, `VAR_DECL`, . . . pour les déclarations ;
- `MODIFY_EXPR` . . . `PLUS_EXPR` . . . pour les affectations et les expressions
- `TRY_CATCH_EXPR` . . . `GOTO_EXPR`, `SWITCH_EXPR`, `OMP_PARALLEL` . . . pour les instructions . . .
- etc . . .

Petits extraits du fichier `som4.c.001t.tu` (les trees)

```
@2760   identifier_node   strg: som4      lngt: 4
@2761   function_type     size: @12       algn: 8        retn: @16
                          prms: @2769
@2762   parm_decl         name: @2770     type: @2771    scpe: @2753
                          srcp: som4.c:2                 argt: @2771
                          size: @19       algn: 64       used: 1
```

puis

```
@2769   tree_list         valu: @2771     chan: @160
@2770   identifier_node   strg: t         lngt: 1
@2771   pointer_type      size: @19       algn: 64       ptd : @16
@2772   tree_list         valu: @25       chan: @160
@2773   tree_list         valu: @30       chan: @2777
@2774   tree_list         valu: @2584     chan: @2778
```

Les “tree”s sont verbeux, avec des listes chaînées presque partout.

Les *Gimple*-s et leurs transformations

Gimple = Représentation**s** *normalisée***s** du code [= instructions].
Fichiers **`gcc/gimple.def`** `gcc/gimple.h` `gcc/gimple.c` (36 nœuds possibles)

- `GIMPLE_COND` pour les conditionnelles.
- `GIMPLE_LABEL`, `GIMPLE_GOTO`, `GIMPLE_SWITCH` pour les sauts
- `GIMPLE_ASSIGN` pour les affectations et le calcul [11]
- `GIMPLE_CALL` pour les appels
- `GIMPLE_PHI` pour les nœuds ϕ en SSA
- une douzaine de `GIMPLE_OMP*` pour le support d'OpenMP.

Les nœuds *Gimple* utilisent des *tableaux* pour les séquences d'arguments ⇒ *représentation plus compacte et plus efficace*. Les valeurs (variables, SSA, sous-expressions) y sont des *Trees*.

11. calcul dont un *tree* donne le détail.

dump *Gimple* fichier `som4.c.004t.gimple`

```
som4 (long int * t) {
  long unsigned int D.1597;
  long unsigned int D.1598;
  long int * D.1599;
  long int D.1600;
  long int D.1603;
  long int s;
  s = 0;
  {
    int i;
    i = 0;
    goto <D.1595>;
    <D.1594>:
    D.1597 =
        (long unsigned int) i;
    D.1598 = D.1597 * 8;
    D.1599 = t + D.1598;
    D.1600 = *D.1599;
    if (D.1600 > 0) goto <D.1601>;
     else goto <D.1602>;
    <D.1601>:
    D.1597 =
        (long unsigned int) i;
    D.1598 = D.1597 * 8;
    D.1599 = t + D.1598;
    D.1600 = *D.1599;
    s = D.1600 + s;
    <D.1602>:
    i = i + 1;
    <D.1595>:
    if (i <= 3) goto <D.1594>;
     else goto <D.1596>;
    <D.1596>:
  }
  D.1603 = s;
  return D.1603;
}
```

Gimple/SSA

static single assignment : chaque variable est affectée **une seule fois** ⇒ nœuds Φ en tête de bloc pour joindre des valeurs - dump *Gimple* `som4.c.056t.phiprop`

```
som4 (long int * t) {
  int i;
  long int s;
  long int D.1600;
  long int * D.1599;
  long unsigned int D.1598;
  long unsigned int D.1597;
<bb 2>:
  goto <bb 6>;
<bb 3>:
  D.1597_6 = (long unsigned int) i_3;
  D.1598_7 = D.1597_6 * 8;
  D.1599_9 = t_8(D) + D.1598_7;
  D.1600_10 = *D.1599_9;
  if (D.1600_10 > 0)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  D.1597_11 = (long unsigned int) i_3;
  D.1598_12 = D.1597_11 * 8;
  D.1599_13 = t_8(D) + D.1598_12;
  D.1600_14 = *D.1599_13;
  s_15 = D.1600_14 + s_2;
<bb 5>:
  # s_1 = Φ <s_2(3), s_15(4)>
  i_16 = i_3 + 1;
<bb 6>:
  # s_2 = Φ <0(2), s_1(5)>
  # i_3 = Φ <0(2), i_16(5)>
  if (i_3 <= 3)
    goto <bb 3>;
  else
    goto <bb 7>;
<bb 7>:
  return s_2;
}
```

types de données *gimple*

Il y a un seul (gros) [12] type de donnée *gimple* dans les fichiers `gcc/gimple.def gcc/gimple.h gcc/gimple.c`

Les différentes formes (SSA ou non, ...) de *gimple* sont représentées dans le même type de données en C, plus ou moins contraintes.

Un nœud *gimple* contient ses arguments dans un tableau de pointers `tree`, par exemple les $\Phi-$nœuds dans `gcc/gimple.h` :

```
struct GTY(()) gimple_statement_phi {
  /* [ WORD 1-4 ]  */
  struct gimple_statement_base gsbase;
  /* [ WORD 5 ]  */
  unsigned capacity;
  unsigned nargs;
  /* [ WORD 6 ]  */
  tree result;
  /* [ WORD 7 ]  */
  struct phi_arg_d GTY ((length ("%h.nargs"))) args[];
};
```

12. L'union des `struct`-ures déclarées dans `gcc/gimple.h`

le graphe de flot de contrôle

Il est représenté par plusieurs structures de données :

1. les séquences de gimple `gimple_seq`
2. les blocs élémentaires `basic_block` dans `gcc/basic-block.h`
3. les arêtes entre blocs `edge`
4. etc...

Chaque bloc connaît les arêtes en partant et y arrivant.
Chaque arête connait ses extrémités.
Chaque instruction Gimple connait le bloc la contenant.

Chaque fonction connait ses blocs initial et final.
La “variable” `cfun` de type `struct function` dans `gcc/function.h` est la fonction compilée courante.
`gcc/cgraph.h` décrit les liens d'appels entre fonctions...

les traitements internes de GCC

La compilation est structurée en **passes** de compilation.

- Il y a *beaucoup* (≈ 200) de passes dans GCC.
- Leur lancement dépend des optimisations demandées.
- Chacune est déclarée dans `gcc/tree-passes.h` et lancée par `gcc/passes.c`.
- Chaque passe (ou jeu de passes) a son propre fichier source.
- Une passe peut être lancée 0 ou plusieurs fois.
- l'ordre des passes est (plus ou moins) dynamique.
- un greffon peut ajouter (ou ôter) des passes.
- une passe est organisée en arborescence (avec des sous-passes).

les passes internes de GCC

Il s'agit des passes après analyse syntaxique.

- passe simple GIMPLE `GIMPLE_PASS` sur une fonction
- passe interprocédurale simple *"Inter Procedural Analysis"* `SIMPLE_IPA_PASS`
- passe interprocédurale complexe `IPA_PASS`
- passe de génération *"Register Transfer Language"* `RTL_PASS`

Chaque passe a une fonction de porte *"gate"* (décidant si on la lance ou non) et une fonction d'exécution *"execute"*.

```
//descripteur de passe dans tree-pass.h
struct opt_pass {
  enum opt_pass_type type; //le type, GIMPLE_PASS etc..
  const char *name; //le nom de la passe
  bool (*gate) (void); //la fonction porte
  unsigned int (*execute) (void); //la fonction d'execution
  /* A list of sub-passes to run, dependent on gate predicate.  */
  struct opt_pass *sub;
  /* Next in the list of passes to run, independent of gate predicate.
  struct opt_pass *next;
  /* Static pass number, used as a fragment of the dump file name.  */
  int static_pass_number;
  /* The timevar id associated with this pass.  */
  timevar_id_t tv_id;
  /* Sets of properties input and output from this pass.  */
  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  /* Flags indicating common sets things to do before and after.  */
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
};
```

Il y a deux approches complémentaires pour programmer des greffons de GCC :

- l'interface directe
  - accède directement aux structures de données internes de GCC
  - très efficace mais tributaire des changements internes de GCC
  - requiert une connaissance approfondie de GCC
  - déployée dans GCC 4.5
- l'interface abstraite
  - destinée à l'expérimentation
  - callbacks sans paramètres explicites, paramètres accédés par nom arbitraire
  - expose uniquement des entités abstraites, e.g., les paramètres d'une optimisation
  - isole le greffon des changements internes de GCC
  - déployée dans des projets de recherche ouverts basés sur GCC 4.4 - cTuning.org, MILEPOST gcc (www.milepost.eu)

Interface directe (gcc/gcc-plugin.h)

```
struct plugin_argument {
  char *key;    /* key of the argument.  */
  char *value;  /* value is optional and can be NULL.  */
};

/* Function type for the plugin initialization routine. Each plugin mo
   should define this as an externally-visible function with name
   "plugin_init."
   PLUGIN_INFO - plugin invocation information.
   VERSION     - the plugin_gcc_version symbol of GCC.
   Returns 0 if initialization finishes successfully.  */
typedef int (*plugin_init_func) (struct plugin_name_args *plugin_info,
                                 struct plugin_gcc_version *version);

/* Function type for a plugin callback routine.
   GCC_DATA  - event-specific data provided by GCC
   USER_DATA - plugin-specific data provided by the plugin  */
typedef void (*plugin_callback_func) (void *gcc_data, void *user_data);
```

Interface directe : utilisation

- Chargement d'un greffon :

```
-fplugin=/path/to/NAME.so
```

- Passage de paramètres au greffon :

```
-fplugin-arg-<name>-<key>[=<value>]
```

- Enregistrement d'un callback :

```
/* Called from the plugin's initialization code.
   Register a single callback.
   This function can be called multiple times.
   PLUGIN_NAME - display name for this plugin
   EVENT       - which event the callback is for
   CALLBACK    - the callback to be called at the event
   USER_DATA   - plugin-provided data   */

void register_callback (const char *plugin_name,
                        int event,
                        plugin_callback_func callback,
                        void *user_data)
```

Interface abstraite (MILEPOST GCC, highlev-plugin.h)

```
/* Callback type for high-level argument-less event callbacks */
typedef void (*event_callback_t) (void);

/* manipulation of event tables and callback lists */
extern void register_plugin_event (const char *name, event_callback_t func);
extern void unregister_plugin_event (const char *name);
extern int call_plugin_event (const char *event_name);
extern const char **list_plugin_events (void);

/* return codes for call_plugin_event */
#define PLUGEVT_SUCCESS         0
#define PLUGEVT_NO_EVENTS       1
#define PLUGEVT_NO_SUCH_EVENT   2
#define PLUGEVT_NO_CALLBACK     3

/* manipulation of event parameter (callback arg) tables */
extern const char **list_event_parameters (void);
extern void *get_event_parameter (const char *name);
```

Interface abstraite : utilisation

- Chargement d'un greffon : via l'environnement

```
ICI_PLUGIN=/path/to/plugin.so gcc -fici ....
```

- Passage de paramètres au greffon : à la charge du greffon
- Enregistrement d'un callback :

```
register_plugin_event ("some event name", &callback
```

- Déclenchement d'un évenement :

```
call_plugin_event("some event name");
```

Interface abstraite : accès aux paramètres

- côté GCC :

```
/* Code to control loop unrolling */
register_event_parameter ("loop_id", &(loop->num));
register_event_parameter ("loop_insns", &(loop->ninsns));
register_event_parameter ("loop_avg_insns", &(loop->av_ninsns));

register_event_parameter ("decision.times", &(loop->lpt_decision.times));
register_event_parameter ("decision.decision", &(loop->lpt_decision.decision));
register_event_parameter ("decision.unroll_at_runtime",
      loop->lpt_decision.decision == LPT_UNROLL_RUNTIME ? (void *) 1 : (void *) 0);
register_event_parameter ("decision.unroll_constant",
      loop->lpt_decision.decision == LPT_UNROLL_CONSTANT ? (void *) 1 : (void *) 0);

call_plugin_event("unroll_feature_change");

unregister_event_parameter ("loop_id");
unregister_event_parameter ("loop_insns");

unregister_event_parameter ("loop_avg_insns");
unregister_event_parameter ("decision.times");
unregister_event_parameter ("decision.decision");
```

- côté greffon :

```
int *decison = get_event_parameter("decision_times");
if ((decision_times != NULL) && ......
```

- **avantage :** le code du greffon n'a pas à connaître les détails internes de GCC

Greffons plus ambitieux avec l'outil MELT

"Middle End Lisp Translator"

outils et greffons pour analyser le code source

L'*analyse statique* de code source existe depuis longtemps :

- cadres théoriques : interprétation abstraite, model checking, préconditions de Hoare, . . .
- outils pratiques : Coverity, Mathworks Polyspace, . . .
  [libre LGPL] Frama-C `frama-c.cea.fr` . . .
- greffons spécifiques de GCC : TreeHydra `developer.mozilla.org/en/TreeHydra` pour Mozilla, CRISP `babel.ls.fi.upm.es` (règles de codage Misra-C, HIPP)

MELT facilite le développement de tels greffons de GCC.
Les logiciels importants devraient avoir leur greffon spécifique !

Les logiciels importants imposent des habitudes et des règles d'utilisation

Un greffon de GCC pourrait aider à valider l'utilisation et à suivre les usages :

- verifier que le résultat de telles routines (`fopen` par exemple) est testé
- naviguer ou chercher dans un gros code source
- aider à suivre l'évolution (fonctions obsolètes, . . .)

motivations et contexte de MELT

MELT[13] = un *outil pour faciliter le codage de vos extensions* de GCC :
`http://gcc.gnu.org/wiki/MiddleEndLispTranslator`

- langage spécifique de haut niveau[14] pour le *"middle-end"* de GCC
- MELT est intéressant pour l'inspection et la modification des représentations internes médianes de GCC : Gimple, Tree, CFG, etc.
  1. language dynamique de haut niveau (ramasse-miettes, objets, valeurs fonctionnelles, filtrage = *"pattern-matching"*, . . .)
  2. language traduit en du code C conforme au style de GCC : un fichier `foo.melt` est traduit en `foo.c` puis compilé en le module `foo.so` chargé [`dlopen`] par GCC.
  3. langage adapté et adaptable aux évolutions de GCC

13. Financement (public français) par les projets : ITEA *GlobalGCC* (2006-2009) et FUI *OpenGPU* (2010-2011).
14. *"Domain Specific Language"*

MELT comme greffon

- MELT est lui-même un gros greffon de GCC `melt.so` [15].
- le traducteur MELT est codé en MELT
  (32KLOC `melt/warm*.melt` traduits en 667KLOC de C)
- MELT contient aussi des analyseurs/optimisations simples :
  `fprintf(stdout, ...)` → `printf(...)`,
  detection du test des `fopen`
- à terme également (OpenGPU) :
  génération d'OpenCL pour GPGPU.
- vos contributions sont bienvenues !

Contactez `basile@starynkevitch.net` [16] pour un exposé, une aide sur MELT !

---

15. qui charge vos modules (codés en MELT) dans le compilateur
16. ou `basile.starynkevitch@cea.fr`

Utilisations de MELT

Utilisation avec *le greffon et des modules* MELT déjà *installés* :
`gcc -fplugin=melt.so -fplugin-arg-melt-mode=`*`mode`*
`foobar.cc` par exemple avec *mode* ≡ `makegreen`

Utilisation avec **votre propre code** MELT *`mon.melt`*
`gcc -fplugin=melt.so -fplugin-arg-melt-mode=runfile -fplugin-arg-melt-arg=mon.melt -O2 -c machin.c`

1. traduction `mon.melt` → `mon.c`
2. compilation[17] interne `mon.c` → `mon.so`
3. chargement par le greffon `melt.so` du module `mon.so`
4. compilation `-O2 -c` améliorée (par `mon.so`) de votre `machin.c`

17. Le greffon `melt.so` lance lui-même par `fork+execve` un `make mon.so` qui est bien plus lent que la génération de `mon.c` !

Traduction de votre code `mon.melt` → `mon.c` :

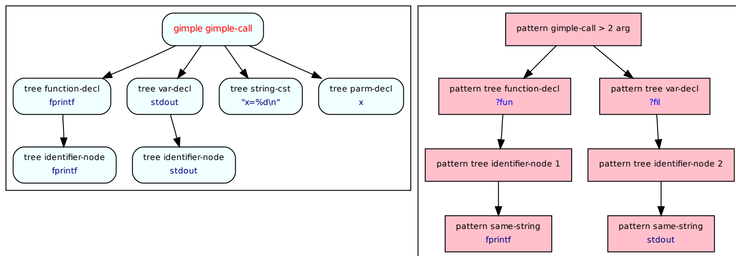
```
gcc -fplugin=melt.so
-fplugin-arg-melt-mode=translatefile
-fplugin-arg-melt-arg=mon.melt fichiervide.c
```

Autres options utiles :

- `-fplugin-arg-melt-module-make-command='make -j2'`
- `-fplugin-arg-melt-module-path=...`
- `-fplugin-arg-melt-source-path=...`
- `-fplugin-arg-melt-debug`
- `-fplugin-arg-melt-init=...`
- etc. . .

Nécessité du filtrage (pattern-matching)

Les représentations internes de GCC sont complexes, car elles représentent des arborescences,voire des graphes orientés cycliques. Les greffons pour l'analyse ou l'optimisation spécifiques à un logiciel doivent reconnaître des formes par **filtrage** [18] *"pattern-matching"*.



difficultés du filtrage

- le filtrage est connu depuis longtemps (1980, Prolog, ...)
- soucis d'efficacité (éviter de faire naïvement plusieurs fois le même test)
- en fait on fait passer la même donnée (Gimple par exemple) à travers plusieurs filtres (ou patrons = *"patterns"*) qui sont factorisés (sous-tests communs)
- les données de GCC peuvent évoluer, le filtrage de MELT suit l'évolution.
- MELT a des dispositifs linguistiques ad hoc
- un jeu de filtres est traduit en un graphe de tests, traitements et branchements

Exemple de filtrage en MELT

```
(let ( (:gimple g ⌈du code pour obtenir un Gimple⌋) )
  (match g
     (?(gimple_assign_cast ?lhs ?rhs)
       ⌈traiter lhs et rhs pour une affectation de conversion⌋    )
     (?(gimple_assign_single
          ?lhs    ?(and ?rhs
                    ?(tree_var_decl ?(cstring_same "stdout"))))
       ⌈traiter lhs et rhs pour une affectation de stdout.⌋  )
     (?(gimple_call_2_more ?lhs
        ?(and ?callfndcl
           ?(tree_function_decl ?(cstring_same "fprintf") ?_))
        ?argfile ?argfmt ?nbargs)
       ⌈traiter l'appel à fprintf argfile avec argfmt⌋)
     (?_  ⌈autrement...⌋))
```

Coder ça en C serait très fastidieux !

Idées directrices de MELT

- traits linguistiques de haut niveau :
  1. valeurs fonctionnelles
  2. langage dynamique à objets
  3. filtrage *"pattern matching"*
  4. meta-programmation
- **génération de code C** conforme aux styles et usages internes de GCC
- langage amorcé (le traducteur MELT est codé en MELT).
- traite aussi bien des valeurs *"values"* (fermetures, objets, avec ramasse-miettes performant) que des trucs de GCC *"stuff"* (Gimple, Tree)
- **s'adapte facilement aux évolutions de GCC**
- constructions de haut niveau pour générer du C [19]
- syntaxe lispienne (une syntaxe alternative infixe apparait)

19. On peut insérer du code C dans du MELT, similairement au `asm` de C !

traits du langage MELT

- Les valeurs MELT sont typées dynamiquement (objets, fonctions [20], listes, tuples, . . .).
- Les trucs (`:gimple, :edge, :tree, :long, :cstring` ...) sont typés statiquement et déclarés [21]
- Chaque valeur (même Nil) a son discriminant (sa classe si c'est un objet) comme **DISCR_INTEGER**, ou **CLASS_CLASS** - c'est un objet MELT
- Plusieurs *dispositifs linguistiques* pour s'**adapter** facilement à GCC et *à son évolution* (définir des filtres, des itérateurs, des primitives par leur mode de traduction en C).

20. notamment anonymes **lambda**, donc fermetures *"closures"*
21. Arguments et résultats peuvent être des valeurs ou des trucs, déclarés comme tels.

Autres caractéristiques (sémantiquement proche de Python, Ruby, Scheme, voire Javascript ...) :

- tout est expression (y compris les “instructions”)
- liaisons locales **`let`** ou recursives **`letrec`**
- modularité (chaque fichier `*.melt` définit un module)
- reflexivité introspective (accès à l'environnement et à la pile)
- orthogonalité des valeurs
- système de macro (un peu à la Scheme)
- programmation fonctionnelle/applicative
- système d'objets (classes avec héritage simple) très dynamique
- on peut installer/ôter à tout moment des méthodes et envoyer un message à n'importe quelle valeur
- facilité de déboguage → `gdb` inutile car MELT crache rarement.

Installer une passe dans MELT

fichier `melt/xtramelt-ana-simple.melt`

```
(defun makegreen_docmd (cmd moduldata)
  (let ( (greenpass
  (instance class_gcc_gimple_pass
    :named_name '"melt_greenpass"
    :gccpass_gate makegreenpass_gate
    :gccpass_exec makegreenpass_exec
    :gccpass_data (make_maptree discr_map_trees 100)
    :gccpass_properties_required ()
     )   ) )
 ;;; register our pass after the "phiopt" pass
  (install_melt_gcc_pass greenpass "after" "phiopt" 0)
  (return greenpass)))
```

installer le mode qui active la nouvelle passe

```
;;the object describing our mode
(definstance makegreen_mode
  class_melt_mode
  :named_name '"makegreen"
  :meltmode_help '"MELT 'makegreen' mode; enable
a pass finding fprintf to stdout..."
  :meltmode_fun makegreen_docmd
)
;; install our mode descriptor
(install_melt_mode makegreen_mode)
```

Coder des passes en MELT

On peut définir ses passes de GCC en MELT, notamment :

1. analyse et navigation faciles grâce au filtrage (expressif, souple, puissant).
2. detecter des formes de code - à des niveaux variés de GCC (Generic, Gimple, Gimple/SSA, ...)
3. produire des diagnostics spécifiques à votre application
4. faire des optimisations spécifiques
5. extraire des informations via le compilateur (navigations, métriques, . . .)
6. valider des règles de codage métier
7. tout ce que vous imaginez !

- MELT est relativement efficace, par sa traduction (rapide) en du C [22].
- MELT fournit aussi un environnement d'exécution *"runtime"* performant et adapté (ramasse-miettes générationnel).
- Concision du code MELT ($\approx 3 - 8\times$ plus petit qu'en C) ; on peut donc coder en MELT des traitements économiquement irréalistes à coder dans un greffon en C.

22. Le temps de compilation du code C généré est plus grand que le temps de traduction et génération de ce code. Il y a eu des progrès très récents, car MELT ne génère plus des routines énormes.

Inconvénients et avantages des modules MELT

Difficultés principales

- comprendre vos besoins et imaginer les possibilités des greffons de GCC (ou modules MELT)
- appréhender[23] les représentations internes et les passes de GCC
- apprendre par l'exemple à coder en MELT

NB : MELT n'est pas adapté pour des greffons qui ne se plongent pas dans les représentations internes “arborescentes” (Gimple, Generic, Tree) médianes de GCC.

23. Sans devenir un expert de GCC

Facilités de MELT

- concision et confort du langage MELT (traits essentiels : filtrage, programmation applicative/fonctionnelle/objets, adaptabilité à GCC...)
- faculté d'incorporer du code C si besoin
- disponibilité du greffon MELT et de son auteur : je peux venir exposer sur MELT, vous conseiller ponctuellement, etc.

Conclusions

- Compilez et faites compiler GCC **4.5**[24] dès sa parution !
- **développez** et faites développer **vos greffons** en C ou **vos modules** en MELT pour *vos besoins*

Questions

A quoi serviront vos :

- greffons de GCC (en C), ou
- modules (en MELT) ? ?

24. Ou bien un trunk récent actuel

Évolution de GCC en chiffres

GCC[25] = *un **énorme** logiciel **patrimonial** toujours **croissant***

version	4.2.1	4.4.1	*trunk* r.157088
date	juillet 2007	juillet 2009	février 2010
code source	2956KLOC	3844KLOC (+30%)	4021KLOC (+36%)
Fichiers	36,8K	66,0K (+79%)	68 ?K (+85%)

Avec une *grande communauté de développeurs*[26] obéissant à des règles sociales strictes[27].

25. **Gnu Compiler Collection** : `gcc.gnu.org`
26. ≈ 400 *"maintainers"* sans aucun dictateur bénévole.
27. Chaque contribution (*"patch"*) est revue et acceptée par quelqu'un d'autre

Historique de GCC

- Autrefois (1987 !) Gnu C Compiler
  1. **Langage dominant** sous Unix™ : *C* (Kernighan&Ritchie)
  2. Les *compilateurs* étaient *propriétaires*
     (mais gratuits sur SunOS)
  3. *Unix™* était *écrit en C*
  4. **Le logiciel libre exige un compilateur libre**
     motivation initiale de GCC : libre et *simple*
  5. architecture *homogène* des machines (68020 $\rightarrow$ Sparc)
     - *mémoire* RAM *petite* (quelques Mo)
     - d'accès *uniforme* et rapide (quelques cycles CPU $\equiv \mu s$)
     - disque limité (50 Mo)
     - performance *prévisible* des instructions machine
     - processeur séquentiel et deterministe
  6. un compilateur *simple* convenait.

GCC 1 : simple compilateur C (1987)

- mars 1987 : première annonce de GCC 0.9 (R.M.Stallman)
- cible 68020 + vax
- capable de compiler Emacs
- GCC 1.8 : août 1987
- `gcc-1.42.tar.gz` septembre 1992, 1818Ko

Langage C proche des machines de l'époque :

- mot-clé **register**
- traduction "syntaxique" bloc par bloc - en C { ... }
- programmes séparés :
  1. préprocesseur `cpp`
  2. traducteur `cc1`

  (puis [BinUtils] assembleur `as` et édition de liens `ld`)

GCC 2 : Cygnus, C++ & RISC (1999) ; crise EGCS

1. ajout d'un compilateur *natif* C++ (Michaal Tiemann, Cygnus)
   - C++ est encore experimental
   - C++ originel (ATT) = un traducteur `cfront` C++ $\rightarrow$ C
   - `g++` compile nativement (donc plus vite que `cfront`)
2. ajout de nombreuses cibles RISC. Âge d'or des stations de travail (RISC = Reduced Instruction Set Computer). Sparc, PA-RISC, PowerPC
3. crise EGCS (Experimental Gnu Compiler System) - “fork” de GCC (août 1997)
4. réunification : EGCS rejoint GCC : `gcc-2.95` (juillet 1999) compile C,C++,Fortran,Chill,Java

GCC 3 et 4 : la maturité

GCC 3.0 (juin 2001)

- compile et optimise une fonction toute entière
- représentation[s] interne[s] intermédiaire[s] du code

GCC 4.0 (avril 2005)

- plus de 2.2MLOC
- multi-source ; C, C++, ObjectiveC, Java, Fortan 77&95, Ada, . . .
- multi-cible multi-plateforme
- représentations internes Gimple Tree et Tree SSA (indépendantes des langage source et cible)
- compile et optimise une unité de compilation

GCCs récents

GCC 4.3 (mars 2008) - près de 4MLOC

- compétitif par rapport à la concurrence
- plusieurs équipes de compilation migrent vers GCC
- représentations internes Generic + Gimple Tuple + SSA

GCC 4.5 (à paraître au 1er trimestre 2010)

- toujours compétitif, même vis à vis de LLVM qui progresse beaucoup
- optimisations majeures (LTO)
- greffons
- monumental

Outils, méthodes et coutumes de développement

- les mailing-lists (messageries *publiques et archivées*)
  - `gcc@gcc.gnu.org` *débats techniques*
    [517 messages en septembre 2008]
  - `gcc-patches@gcc.gnu.org` *propositions de code*
    [2000 messages en septembre 2008]
  - etc. . .
- le chat `irc://irc.oftc.net/#gcc`
- documentation interne en Texinfo `*.texi`
- wiki `http://gcc.gnu.org/wiki/`
- base de bogues `http://gcc.gnu.org/bugzilla/`
- GCC Summit (Canada, 2 jours chaque été)
- workshops, conférences, . . .

Traits principaux de GCC

quelques éléments copiés de l'exposé de Laurent Guerby à Toulibre

- **logiciel libre, sous licence GPLv3**, ©FSF, en C (C++, Ada)
- multi-langage source : *C, C++, Java, Objective C[++], Fortran, Ada* (et aussi Modula 3, Pascal, PL/1, D, Mercury, VHDL en dehors)
- *plus de 30 machines/systèmes cibles* dont x86/AMD64, ARM, PowerPC, IA-64, MIPS, Sparc, CRIS, PowerPC, M68K, Xtensa, RX, Alpha, ...
- *compilateur croisé* (cible $\neq$ hôte) si besoin - important pour l'embarqué
- assez facile à construire puis installer en mode natif

```
apt-get build-dep gcc #installer les dépendances: GMP ...
tar xfj gcc-4.5.tar.bz2; mkdir Build; cd Build
../gcc-4.5/configure #avec les options, voir --help
make bootstrap; sudo make install
```

- compilateur amorcé : *make* `stage1` avec le compilateur système, puis `stage2` avec ce gcc, puis `stage3` pour comparer.

Options habituelles de GCC

Quelques options habituelles [28] (parmi plus d'une centaine) :

- `-v -time` affiche les programmes lancés (dont `cc1`) et leur temps
- `-O0` sans optimisation (compile vite, code médiocre), par défaut
- `-O1` ou `-O2` avec optimisations simples ou coûteuses
- `-O3` avec optimisations très coûteuses *“auto-inlining, vectorization”*
- `-Os` avec optimisation de la taille
- `-g` avec information de déboguage
- `--help` pour l'aide (ou même `gcc -Q --help=optimizers -O1`)
- `-m32` (x86 32bits) ou `-m64` (x86-64) (ou même `-march=` ou `-mtune=`)
- `-Wall` : tous les avertissements - *très conseillé*
- beaucoup d'options `-f`Ω, y compris des optimisations supplémentaires ; `-fno-inline` `-fprofile-generate` `-fprofile-use`

28. options communes à plusieurs versions de GCC

Options nouvelles de GCC 4.5

La prochaine version 4.5 apporte plusieurs nouveautés importantes.

- **optimisation à l'éditions de liens** *"link-time optimization"*.
  Il faut passer par exemple `-flto -O2` *à la compilation* ***et*** *à l'édition de liens*.
  Il est préférable d'avoir le linker GOLD (paquet `binutils-gold`).
- **mécanisme de greffons** *"plugins"*.
  Le compilateur GCC peut être étendu `-fplugin=./foo.so` par des greffons extérieurs, qui pourraient par exemple :
  - organiser différemment les phases de compilation
  - étendre légèrement le langage, par des pragmas ou attributs
  - optimiser pour une bibliothèque *"library"* particulière
  - fournir un diagnostic spécifique à une application
  - tout ce que vous pouvez imaginer *"everything but the kitchen sink"* !

  Les greffons doivent être libres (compatibles GPLv3) !
  Ecrivez vos propres greffons (par exemple avec MELT)

Autres nouveautés de GCC 4.5

`http://gcc.gnu.org/gcc-4.5/changes.html` :

- support du format de déboguage Dwarf3 pour Gdb 7.0
- diagnostics affinés, notamment avec numéro de colonne
- support des dialectes expériementaux de C, C++0x, Fortran, ...
- encore meilleur respect des standards
- optimisations avancées, dont Graphite (optimisation polyhedrique) avec `-floop-parallelize-all`
- mode de profilage pour conseils sur conteneurs standards C++
- plusieurs cibles ajoutées (MeP, RX) ou ôtées (Itanium1)
- `-fwhopr` *"whole program analysis & optimization"* *expérimental*
- etc.

Conseils de travail dans GCC

Contribuer à GCC ou *coder son greffon*, c'est travailler dans GCC.

GCC est trop énorme pour être totalement compris :

- **ne pas chercher à tout comprendre**
- *lire la documentation interne*
- ***demander de l'aide** sur la liste* `gcc@gcc.gnu.org`
  après avoir cherché un petit peu tout seul
- *s'inspirer de code existant*
  mais lire seulement le code proche de ce qu'on veut faire.
- programmer défensivement
- **tester** *très fréquemment*
- *utiliser les bons outils* (par exemple MELT)
- ne pas trop bien faire ; publier son code très vite
- n'ayez pas peur !