Customizing the GCC compiler with MELT

Basile STARYNKEVITCH

gcc-melt.org

basile@starynkevitch.net or basile.starynkevitch@cea.fr

CEA, LIST (Labo Sûreté du Logiciel), France

October 4th, 2013, Open World Forum (Paris, Montrouge)

Caveat

All opinions are mine only

- I (Basile) don't speak for my employer, CEA (or my institute LIST)
- I don't speak for GCC community
- I don't speak for anyone else (e.g. funding agencies)
- My opinions may be highly controversial
- My opinions may change

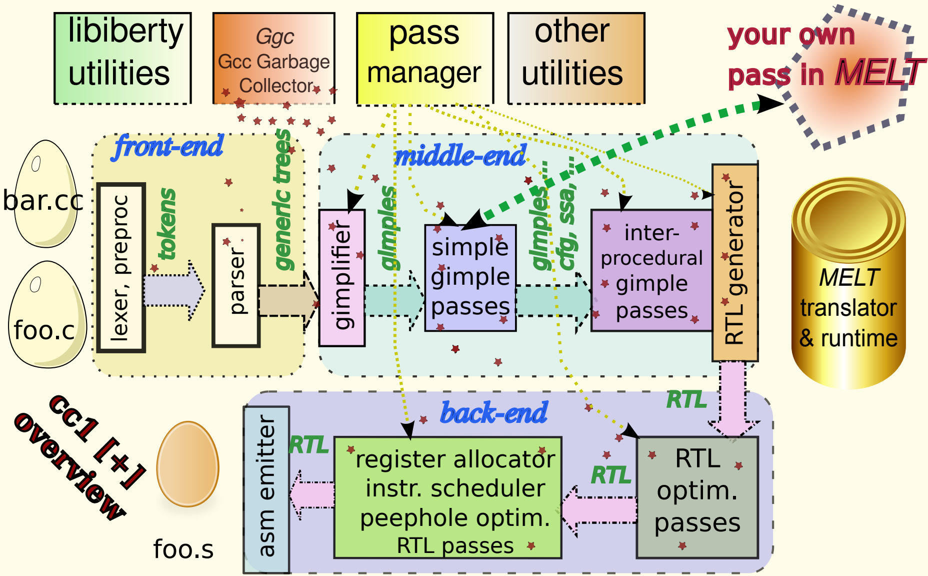
Contents

About GCC

gcc.gnu.org : Gnu Compiler Collection Gcc

- **free GNU software** (GPLv3+ licensed, FSF copyrighted)
- related **collection** of ***optimizing*** **compilers** for **many source languages :** *C*, *C++* [2011], *Ada*, *Fortran*, *Objective-C*, [soon] *D* , *Go*, . . .
- **hosted** on **many** systems : GNU/Linux, MacOSX, Android, other Unixes, Hurd, Windows, . . .
- **targetting many processors** (x86, ARM, Sparc, PowerPC, MIPS, Cris, Xtensa, Mmix, . . .) and systems
- main compiler on GNU/Linux; often used as **cross-compiler**
- since **1985**; current version **4.8** [1]; still *growing* (+6% in 2 years)
- more than **ten millions** of source lines of code; ≈ **400 developers**
- **customizable and extensible** [2] **thru** ***plugins*** (e.g. MELT)

[1] Release 4.8.1: may 2013, 4.8.0: march 2013; 4.7.0: march 2012, 4.7.3: april 2013
[2] Since GCC 4.5 (april 2010) experimentally or 4.6 (march 2011) !



libiberty utilities
Ggc
Gcc Garbage Collector
pass manager
other utilities
your own pass in MELT
front-end
bar.cc
foo.c
lexer, preproc
tokens
parser
generic trees
middle-end
gimplifier
gimples
simple gimple passes
gimples, cfg, ssa,...
inter-procedural gimple passes
RTL generator
MELT translator & runtime
cc1 [+] overview
back-end
RTL
asm emitter
RTL
register allocator
instr. scheduler
peephole optim.
RTL passes
RTL
RTL optim. passes
foo.s

Why and when customize Gcc ?

- **Gcc customization** (with MELT or some other plugin [3], or even your own plugin in *C++*) is *worthwhile* for **advanced Gcc users** (not for "hello world" programs!)
- Compiler customization possible thru the *GCC Runtime Library Exception* : plugins should be "GPLv3 compatible".
- **work on** and *take advantage of* some Gcc **internal representations**
- profit of *existing* Gcc **optimizations**
- when external textual approaches (`grep`, `perl`, `awk` ...) are inadequate
- examples
  - **find all calls to `malloc` with a constant argument** > 100 (generally, `malloc(sizeof` τ`)` or `malloc (2*sizeof` τ`)` is not easily `grep`-able and may appear after inlining and constant folding)
  - find all assignments to the `next` field of some `struct packet_st`
  - **optimize** `fprintf (stdout,` $\phi, \alpha_1 \ldots$`)` $\Rightarrow$ `printf(`$\phi, \alpha_1 \ldots$`)`
  - semi-automatic **validation of** some **industry-specific coding rules** [4] (every call to `fork` is tested for < 0 in the same function doing the `fork`)

[3] Like D.Malcom's Gcc Python Plugin.

[4] or validation of API-specific coding rules

Contents

Importance of GCC optimizations

Gcc [5] is doing many important optimizations, because:

1. users want runtime performance (of their compiled code).
2. hardware is much farther from the low-level *C* language than it was in the 1980s (super-scalar out-of-order multi-core heterogeneous processors today!).
3. predicting hardware behavior (timing? energy consumption?) is impossible today (how much nanoseconds cost this `i++` in your *C* code ???).
4. languages standards are slowing raising the abstraction level.

```
int sumarrayof10(std::array<int,10> &t) // C++2011
{ int s=0;
  std::for_each(t.begin(),t.end(),[&s](int e){s+=e;});
  return s; }
```

same optimized code (but *very different* unoptimized code) as

```
int sumarrayof10(int *t) { int s=0; /* C99 */
    for(int ix=0; ix<10; ix++) s+= t[ix];
    return s; }
```

[5] Other industrial-strength compilers, e.g. Clang/LLVM, also have important optimizations...

find **malloc**-s of constant size > 100 with MELT

Just in one [long] command line with MELT 1.0 [6] and GCC 4.7 or better

```
gcc -fplugin=melt -fplugin-arg-melt-mode=findgimple
 -fplugin-arg-melt-arg='
 ?(gimple_call_1
     ?(tree_function_decl_of_name "malloc" ?_ ?_)
     ?(tree_integer_cst  ?(some_integer_greater_than 100))
 )' -O2 -c yourcode.c
```

That has to be done **inside** the compiler (because of inlining, constant folding, `sizeof`, ...) and *cannot be done textually* (e.g. using **grep**). It works by **pattern-matching** on GCC **internal representations** :

- **gimple**-s : elementary abstract instructions (e.g. function calls)
- **tree**-s: abstract syntax trees (for declarations and operands)

Patterns are explained in a few slides!

[6] today oct.04 2013, rc1; real release of MELT 1.0 before october 15th, 2013.

What is happening underneath?

- `gcc` is a driver program which starts the compiler proper `cc1`
- `cc1` is loading the **MELT plugin** before proceeding (preprocessing, parsing, front-end, middle-end, back-end, emission of assembler code)
- MELT needs one (or more) **mode**[s] (otherwise, won't do anything). List them with `-fplugin-arg-melt-mode=help` or add your own.
- the `findgimple` mode:
  1. **needs a pattern** on Gimples as its argument
  2. **translates that pattern** (using MELT macro system ...) into **generated *C++* code** suitable for GCC
  3. **forks** an internal `make` to compile that code into a shared object module [7]
  4. dynamically **loads** with `dlopen` that shared object **module**
  5. runs the generated code which **inserts a new GCC pass** which
     - scan every compiled function for its Gimples
     - pattern-match each Gimple
     - shows a notice on success (with location in your source code)
     - give a summary (various counts) at end of compilation

[7] There is a way to keep for re-use that module

Glance inside GCC passes

- GCC runs many ($>$ 200) optimization **passes**, use **`-fdump-passes`** to find out which and the **`justshowpasses`** MELT mode. Several kinds of passes:
  1. plain **`GIMPLE_PASS`** working on a single function
  2. **`SIMPLE_IPA_PASS`** for simple **I**nter-**P**rocedural **A**nalysis
  3. complex **`IPA_PASS`** for link-time or full program or full-compilation unit optimizations
  4. **`RTL_PASS`** for backends (and target-specific optimizations)

  See
  `gcc-python-plugin.readthedocs.org/en/latest/tables-of-passes.html`
  for a nice picture. You can insert your own pass coded in MELT.

Glance inside GCC trees and gimples

Tree-s represent abstract syntax trees of declarations (and operands) :

- see `tree.def` header file for a list ($>$ 200 kind of trees).
- see `melt/xtramelt-ana-tree.melt`

Gimple-s represent elementary abstract instructions :

- see `gimple.def` header file for a list (36 kind of gimples, half for OpenMP support)
- most Gimples are **3-operand assignments** like `x = y + z`
- variadic Gimples for calls, switches
- see `melt/xtramelt-ana-gimple.melt`

Basic blocks contain a **sequence of gimple-s** and are linked by **edges** for the **control flow graph**

Pass `-fdump-tree-all` to `gcc` to get hundreds of dump files. Or use the **MELT probe** (GTK based).

MELT pattern matching

One of the most **exciting feature** of MELT, the ability to digest arbitrary data (either GCC internals or MELT values). **Patterns** are **filtering and extracting data** (a bit like regexp-s are filtering and extracting strings). Patterns may be nested.

- **`?_`** is a joker or **wildcard** pattern (that always matches).
- **`?(some_integer_greater_than n)`** match integers $> n$
- **`?(tree_integer_cst` π`)`** match tree-s representing a constant integer which is matching the pattern π
- **`?(tree_function_decl_of_name` σ ν τ`)`** match a tree for a function declaration naned by the string σ; the name sub-tree should match ν and the result type tree should match τ
- **`?(gimple_call_1` δ α`)`** match a gimple which is a call to a function whose declaration matches δ and with an argument matching α

Contents

MELT as a high-level domain specific language

- simple, orthogonal, Lisp-like syntax **(** *operator operands* ... **)**
- implemented thru the MELT plugin (GPLv3)
- **values** versus **stuff** :
  1. values are first class citizens (lists, closures, boxed trees, . . . boxed integers, objects with reified classes, etc . . .)
  2. stuff is existing GCC data (raw Gimple; raw Trees; . . .)

  MELT values are more sexy to use.
- **garbage collector** (MELT generational copying GC for values above existing GCC mark-and-sweep GC for stuff)
- **pattern matching**
- **macro system** (and run-time **evaluation** by C++ code generation)
- **various** high-level programming **styles**: **functional, reflective, object-oriented**
- **translated to C++** by a bootstrapped MELT translator (coded in MELT)
- ability to **mix C++ code chunks** and MELT code

Your own MELT extension

- coded in MELT (the high-level lispy DSL)
- translated to C++ by MELT
- understand what to do on GCC internals (Gimples, ...)
- define your MELT mode
- usually add your own GCC pass (choose where)
- may modify GCC internal representations (Gimple transformation)

Your own applications

- API specific coding rules
  Industrial API needs specific support in GCC (much like standard C functions like `printf` are known by GCC)
- navigation (at the Gimple level) or metrics on large software base
- specific optimizations

Use MELT

Use MELT (free software, GPLv3+) at your place. See **gcc-melt.org**. Subscribe to `gcc-melt@googlegroups.com` if using MELT.

Or subcontract CEA, LIST for your MELT development and commercial support, or collaborative research projects. Contact **basile.starynkevitch@cea.fr** and **florent.kirchner@cea.fr** [Head of LSL] for more.