Notes on GIMPLE representation and code generation

Date: 2011-06-25
Author: Nikolaos Kavvadias
Email: nikolaos.kavvadias@gmail.com
Revision: 0.0.1 (2011-06-02), 0.0.2 (2011-06-25)
Web site: http://www.nkavvadias.com
Copyright: Nikolaos Kavvadias (C) 2011

Contents

1 Introduction

GIMPLE is the machine-independent intermediate representation used in modern GCC releases (post version 4.0). While the GIMPLE API for code generation and manipulation has matured over time, the corresponding textual representation is yet to be in a stable form. This issue hampers many serious efforts for code generation from and to GIMPLE.

Good candidates for the textual representation seem to have been in play for some time. The wiki site http://gcc.gnu.org/wiki/GIMPLE sketches the textual IR that is expected to be implemented by the GIMPLE frontend and backend (both under development). Another case of a textual GIMPLE are the formats generated as a *GIMPLE dumps*. We distignuish here two formats: the format in *.t004.gimple files (*tagged* GIMPLE) and the one e.g. in *.t140.optimized files (*GIMPLE-C*). Both formats express the low-level GIMPLE representation, which is closer to classic three-address code than high-level GIMPLE; the latter is closer to the GENERIC AST representation. An extended form of GIMPLE dumps are expected to be established as the GIMPLE language semantics.

For reference tagged GIMPLE is generated by the command-line option:

```
-fdump-tree-all-raw
```

while GIMPLE-C is generated by:

`-fdump-tree-all` and `-fdump-tree-gimple`

the latter only emitting the *.t004.gimple file.

The rest of this document discusses open issues with the textual GIMPLE IRs, focusing on the tagged GIMPLE format. It will not cover extended semantics issues that are covered much better at http://gcc.gnu.org/wiki/GIMPLEFrontEnd

2 Target audience

This document is expected to be of interest to compiler/translator implementors using GIMPLE as either a source or target language.

3 Issues with tagged GIMPLE

Here follows a listing of some issues that can be identified with using tagged GIMPLE. Most of them apply to the GIMPLE-C format as well.

3.1 Losing the original semantics of the source program

One such example is the omission of emitting global variables. A workaround for some cases is the definition of static variables in the original source. This approach provides only specific file/translation unit scope to globals but it is not certain whether the extern specifier is handled properly for referencing these globals from external scope.

3.2 Inconsistency in handling labels

Automatically generated labels (by the gimplifier) and labels defined in the source program are represented differently. The first category are enclosed in single wedges `< >` while the later are explicitly defined and are used without wedges.

For example, this is the definition of an automatically generated label:

```
gimple_label <<D.1983>>
```

while the following is the (redundant) definition:

```
void V1 = <<< error >>>; ...  ‘‘gimple_label <V1>
```

and the use of a source label:

```
gimple_cond <eq_expr, D.1985, 1, V5, <D.1986>>
```

3.3 Destroyed interfaces

Function interfaces are not maintained appropriately since the original argument types in a function definition may be replaced. This is the case with array arguments (with static sizes) in the definition of non-root procedures.

Here follows an example. This is the a partial view of the source program:

```
void evalcoins(int n, int amount, int C[], int *ncoins, int D[])
{
  ...
}

void coins(int n_eurocents, int *n_coins_used)
{
  int C_euro[15], D_euro[15];
  ...
  evalcoins(15, n_eurocents, C_euro, &n_items, D_euro);
  ...
}

However, this is the tagged GIMPLE form of the same program:

evalcoins (int n, int amount, int * C, int * ncoins, int * D)
gimple_bind <
  ...
>

coins (int n_eurocents, int * n_coins_used)
gimple_bind <
  int C_euro[15];
  int D_euro[15];
  ...
  gimple_call <eval-
coins, NULL, 15, n_eurocents, &C_euro[0], &n_items, &D_euro[0]>
  ...
>
```

3.4 Pointer expressions

Low-level GIMPLE (tagged and GIMPLE-C) use two basic specific operations for dealing with pointer expressions and indirect references, the `pointer_plus_expr` and `indirect_ref`.

A `pointer_plus_expr` in tagged GIMPLE appears as follows:

```
gimple_assign<pointer_plus_expr, D.1986, D, D.1985>
```

where D is an array, D.1985 a temporary int variable and D.1986 a temporary variable defined as pointer to int.

The same in GIMPLE-C is the following:

```
D.1986 = D + D.1985
```

This operation adds the offset determined by D.1985 to the base address of array D, expressed as D. Pointer D.1986 then can be used for intexing the array.

A typical idiom in generated GIMPLE suggests that `pointer_plus_expr` is followed by an `indirect_ref`. The indirect reference is used to access the array and loading the contents of a memory position to a variable.

An `indirect_ref` in tagged GIMPLE appears as follows:

```
gimple_assign<indirect_ref, D.1987, *D.1986, NULL>
```

and the same in GIMPLE-C is:

```
D.1987 = *D.1986;
```

In order to avoid a thorough pointer analysis for establishing that D.1986 points to the contents of array D, typical data-dependence analysis can be used to trace that D is the referenced entity by D.1986.

3.5 Function calls

Function calls are represented by `gimple_call` in tagged GIMPLE. Due to the issue 3 (Destroyed interfaces), in some cases calls-by-reference appear when not really needed. This refers to the simulated call-by-reference available in the C programming language, and not the actual kind that can be found, e.g. in Perl.

For example the following call is by-reference:

```
gimple_call <..., &C_euro[0],...>
```

making use of the address of the first element of C_euro (the base address).

With interfaces kept unchanged, the following would suffice:

```
gimple_call <..., C_euro, ...>
```

3.6 Inconsistency in array initialization sequences

An array can be initialized either by a literal initialization list or by emitting a sequence of operations for initializing its contents. From a black-box point of view, it seems that the gimplifier arbitrarily chooses which approach to follow.

For example, in our example, the C_euro is initialized via explicit operations:

```
gimple_assign <integer_cst, C_euro[0], 1, NULL>
gimple_assign <integer_cst, C_euro[1], 2, NULL>
gimple_assign <integer_cst, C_euro[2], 5, NULL>
gimple_assign <integer_cst, C_euro[3], 10, NULL>
gimple_assign <integer_cst, C_euro[4], 20, NULL>
gimple_assign <integer_cst, C_euro[5], 50, NULL>
gimple_assign <integer_cst, C_euro[6], 100, NULL>
gimple_assign <integer_cst, C_euro[7], 200, NULL>
gimple_assign <integer_cst, C_euro[8], 500, NULL>
gimple_assign <integer_cst, C_euro[9], 1000, NULL>
gimple_assign <integer_cst, C_euro[10], 2000, NULL>
gimple_assign <integer_cst, C_euro[11], 5000, NULL>
gimple_assign <integer_cst, C_euro[12], 10000, NULL>
gimple_assign <integer_cst, C_euro[13], 20000, NULL>
gimple_assign <integer_cst, C_euro[14], 50000, NULL>

It is not clear why an initialization list is not used:

int C_euro[15] = {1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000};
```

Automatically generated labels (by the gimplifier) and labels defined in the source program are represented differently. The first category are enclosed in single wedges `< >` while the later are explicitly defined and are used without wedges.

For example, this is the definition of an automatically generated label:

```
gimple_label <<D.1983>>
```

while the following is the (redundant) definition:

```
void V1 = <<< error >>>; ...  ‘‘gimple_label <V1>
```

and the use of a source label:

```
gimple_cond <eq_expr, D.1985, 1, V5, <D.1986>>
```

3.7 Inconsistency of the tagged GIMPLE format

As discussed in the introduction, the tagged GIMPLE format uses alternate syntax for the unoptimized (*.t004.gimple) and certain optimized (e.g. *.t140.gimple) intermediate code dumps. It would be clearer if a single convention throughout all GIMPLE dumps.

3.8 Lack of bit-accurate semantics

The availability of bit-accurate data types is an interesting asset of modern compiler infrastructures such as LLVM: http://www.llvm.org. LLVM uses the LLVM bitcode IR which adheres to such semantics. On the other side, GCC GIMPLE might be too closely coupled with C-like semantics. Especially, implementors of non-conventional backend architectures (e.g. developers of hardware compilers) would be interested in a form of GIMPLE with bit-accurate types.

For example, the following would denote a 14-bit unsigned integer and a 8.16 signed fixed-point representation, respectively.

- `u14`
- `q8.16s`

4 Final notes

This document is a work-in-progress. Several aspects of programming language translation to low-level GIMPLE are not covered:

1. Support for recursion.
2. OMP semantics.
3. `_Bool` data types.
4. Explicit return types (other than void).
5. Semantics expected to be integrated as part of GCC mainline. These reflect the current status of the `gimple-front-end` branch, which adds important capabilities to the GIMPLE infrastructure such as consistently-styled declarations for pointers, arrays, and compound types (structs, unions).