Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Gimplification Improvements

Zdeněk Dvořák
SuSE CR
dvorakz@suse.cz

Abstract

This paper addresses the expansion of the front-end specific representation to the common intermediate representation (GIMPLE). Currently, this process of "gimplification" is based on the tree rewriting, i.e., the code is matched against a pattern and rewritten to a simpler form. The result of a single step of the rewriting does not have to be a valid GIMPLE—a single idiom may need to be rewritten several times. This approach has several drawbacks; in particular, none of the improvements described below can be reasonably implemented in the current framework.

The paper describes a different approach. The main difference is that valid gimple statements are produced directly. This should be more efficient because of the lack of the repeated matching. The well-defined structure of the produced code makes it possible to incorporate simple cleanup passes (local versions of CCP, DCE and CSE). According to literature, this should improve the compiler performance by decreasing the amount of the code to process later. Additionally, this approach simplifies making changes in the representation of GIMPLE (e.g., changing the type used to represent GIMPLE and avoiding the usage of labels) and preserving information passed by the front-end (e.g., loop specific `#pragma`'s).

Most of the compilers work internally over some intermediate representation of the code, instead of the source language itself. This has many advantages—such intermediate representation is usually much simpler and thus easier to work with, and it also enables using the same set of optimizers for many different languages. Often, a single compiler uses several intermediate representations, differing in the level of description. The level may range from very high, mapping almost 1-1 to the original source code, middle levels where some information about the original source code is still preserved (types, structure of memory references), up to very low levels that map almost directly to the target architecture assembler. The physical realization of the different representation levels may be either completely different, or obtained just by restricting the amount of allowed idioms in a common language.

In GCC, we can recognize several intermediate representations. Physically there are two realizations: trees and Register Transfer Language (RTL). Trees originated from the syntax trees used to describe the source languages, while RTL is more suited to description of the target assembler. While in trees we deal with original variables that still preserve the complete information about their types, at RTL we work with pseudo-registers, for whose the type information is reduced to "mode" that only describes the size of the stored information. Even the information about the signedness is lost in RTL, and there exist separate instructions for the signed and the unsigned case for the operations where the signedness matters (division, extending). Logically, there are the following levels of abstraction:

language-specific trees Each front-end may define the language using specific set of trees. Middle-end does not perform any optimizations over this representation, and it is immediately lowered to one of the following forms.

GENERIC A not very precisely specified level, basically trees using the common set of supported operations. GENERIC should be language independent. Theoretically, front-ends could produce GENERIC directly, and thus avoid need to support translating of the language specific tree nodes to the lower forms of representation. This form however does not really exist at the moment—although the amount of language specific tree nodes was decreased, some still remain, and in some areas (representation of the language type system), their complete elimination is not very likely to happen.

GIMPLE Trees using the common set of source-language independent operations (but the type system is still partially language specific). The complex expressions are split into two-operand operations, and some other simplifications are performed, but control flow instructions and exception handling still remains in the original structured form. At the moment, only very simple cleanups and function inlining is performed at this level, and the function inlining is going to be moved to the lower level soon.

unstructured GIMPLE GIMPLE with structured exception handling lowered and the control flow control reduced to simple jump instructions. This makes it suitable for building and work with CFG, as well as for creation of SSA form, and most of the middle-end optimizations are performed at this level.

RTL Low-level language that maps almost directly to the target architecture assembler.

There are some proposals for narrowing the gap between unstructured GIMPLE and RTL and introducing some form of even lower level GIMPLE, where for example address arithmetics would be explicit and exposed to the optimizers.

By the unofficial general consensus of GCC developers, the process of lowering the language specific trees or GENERIC to GIMPLE is called "gimplification" and the pass performing this operation is "gimplifier." This paper deals with a work on improving the gimplification process. In the next section, we provide some more detailed description of the GIMPLE language and its evolution. In Section 2, we describe the current gimplifier. In Section 3 we point out some deficiencies in the gimplifier, and discuss the possible improvements. In the following Section 4, we describe a design of the new gimplifier we are currently working on. Finally, in Section 5 we describe the current state of the project and present the results obtained so far.

1 GIMPLE

The original GIMPLE was influenced by the SIMPLE intermediate language used by the McCAT compiler developed at McGill University [1]. SIMPLE preserves the control flow structures, and in fact any `goto` statements are disallowed. To enable representation of real-world programs, a goto elimination pass is necessary [2]. This approach is highly impractical, requiring introduction of superfluous control structures and temporary variables, which is not desirable in a production compiler. Therefore, `goto` statements were allowed in GIMPLE from the beginning. Also, some other types of statements had to be introduced to the language in order to suit the needs of GCC.

The original attempts to preserve structured control flow statements did not show up to be profitable. While it was possible to perform many optimizations over the structured GIMPLE, any control flow related transformations were exceedingly difficult to implement—just the code to redirect the edge in CFG had more than 500 lines, required creation of up to two new basic blocks in order to redirect a single edge, and actually it still did not handle all the corner cases. Even just traversing the statements in the code was non-trivial and had to be handled by complicated functions. And it turned out that no optimization significantly used the structured control flow. After some discussions, it was decided to transform the code to the unstructured form before optimizations. In unstructured GIMPLE, the branches of `if` and `switch` statements may only contain `goto`'s, thus making CFG manipulation trivial.

The program in GIMPLE consists of list of statements. The expressions in GIMPLE never directly modify values of their operands, i.e., such modifications are always expressed using statements. The operands of the statements are kept as simple as possible, in particular nested expressions are not allowed and the operands of expressions thus must be simple variables or constants.

For full description and grammar of GIMPLE, see the documentation of GCC [4]. Here we present just a simplified version of the grammar (Figure 1), sufficient to demonstrate the results of this paper. The real GIMPLE is a bit richer, and there are some additional invariants not expressed in the grammar. For example modify statement does not contain two references to memory, if the type of assigned values is scalar—this restriction simplifies the optimizers. The unstructured version of GIMPLE is obtained by removing the block statement and replacing `compound-stmt` in the clause for if statement with `goto-stmt`, thus making the `compound-stmt` only possible on the top level of the program.

2 The Current Gimplifier Outline

The current gimplifier uses the fact that the output GIMPLE language is a subset of the input GENERIC language, and works by rewriting the trees. Basically, it recursively traverses the input tree, and when it encounters an operation that either is not part of GIMPLE, or whose operands do not match the constraints, it rewrites the operation using the GIMPLE statements and enforces the constraints by possibly assigning the values to temporary variables.

The core part of the gimplification is contained in `gimplify_expr` function. As one of the parameters, this function takes a predicate that specifies the form of the output (`rhs`, `lhs`, etc.). The `gimplify_expr` function first calls the front-end to gimplify any language specific tree nodes, and then repeatedly calls an appropriate function for gimplification of the currently processed tree node until the current node satisfies the required constraints.

The gimplification functions return the status, which may be one of the following:

GS_ERROR An error occurred during gimplification, we may need to abort it.

GS_UNHANDLED The function did not know what to do with the tree node. This return value is only valid for the front-end gimplifier, or when used internally by the gimplification functions—it cannot be returned to `gimplify_expr`, since this would lead to an infinite loop.

GS_OK The function transformed the node somehow, but it still does not necessarily satisfy the required constraints. The `gimplify_expr` function needs to re-gimplify the node.

GS_ALL_DONE The node was gimplified to the required form.

To accumulate the statements produced by gimplification, two lists are passed to `gimplify_expr`—`pre` and `post` lists. Normally, the statements are stored to the `pre` list. For operations whose side effects need to be performed after the evaluation of the returned expression, `post` list is used. This makes it possible to avoid creating a temporary variable and increasing a register pressure in some cases, for example

```
x = a++ + b;
```

becomes

```
x = a + b;
a = a + 1;
```

instead of the straightforward code

```
tmp = a;
a = a + 1;
x = tmp + b;
```

3 Drawbacks and Improvements

The approach described in the previous section has many advantages. It is quite simple and flexible, and by rewriting the trees in-place, it saves memory. However, there are some drawbacks.

- The approach uses the fact that the physical representation of the input and the output language is the same. It is hard to determine the point where the original trees become GIMPLE. The rewritten result of `gimplify_expr` may be of any type depending on what predicate was passed to it—`lhs`, `rhs`, even the list of GIMPLE statements.

  However, representing (especially unstructured) GIMPLE with trees is inefficient. The representation of the common `a = b + c` statement requires two nodes—one for `MODIFY_EXPR`, one for `PLUS_EXPR`. Each of the nodes carries some redundant information (reference to type, `TREE_CHAIN` pointer, annotation pointer), as well as pointers to the operands. This wastes memory and decreases performance of caches. Also, it makes some operations with the statements more complicated, for example traversing the operands is nontrivial and the pointers to operands have to be extracted to separate structures, thus wasting even more memory. A flat representation, where the statement would be represented just by its code, type, and a simple array of operands would seem to be more efficient.

  Changing the GIMPLE representation without significant changes to gimplifier would be quite difficult. The simple approach would be to take the output of the gimplification and copy it to the new structures, however this somewhat beats the purpose of getting more memory efficient representation.

- The fact that the input and output representations are not clearly separated also makes it more likely to introduce bugs where non-GIMPLE statements from the input leak to the output.

- During code expansion, it is possible to perform some simple optimizations, like unreachable code removal, CSE, and copy and constant propagation, see for example [3]. While these optimizations are not very powerful, they are fast and they reduce the amount of code passed to the more powerful but slower optimizers, thus improving the compile time and memory consumption.

  However, the description of the results of gimplification by just the `pre` and `post` lists is too limited to enable these optimizations. We would like to include more information into the results, like the values of variables that are known to be constants, the fact whether the current point in code is reachable, etc.

  In fact the separation into `pre` and `post` lists is not sufficient even for their original purposes. For example in

  ```
  if (a++ + b != 0) goto L;
  ```

  we produce

  ```
  tmp1 = a + b;
  tmp2 = tmp1 != 0;
  a = a + 1;
  if (tmp2) goto L;
  ```

  Instead of

  ```
  tmp1 = a + b;
  a = a + 1;
  if (tmp1 != 0) goto L;
  ```

- The structured GIMPLE is produced, but (after a trivial cleanup pass that is much weaker than what we can do with a proper infrastructure directly during the expansion) it is lowered to an unstructured GIMPLE. This is a waste of compile time and memory.

- If the unstructured GIMPLE were generated directly, it would be possible to construct CFG directly during the code expansion. This would enable further cleanups, and also if the control flow would be represented implicitly by CFG, it would not be necessary to generate new label statements on tree-level, and also the control flow statements would not need to contain references to these labels, thus making them smaller. The labels would only need to be added during expansion to RTL, which would save some memory.

- The iterative nature of the gimplification makes it more difficult to pass the information from the front-end. The information may need to be preserved during several rounds of rewriting of the same tree.

- The fact that `gimplify_expr` may produce many different types of output depending on the input predicate makes the code harder to read and maintain than necessary.

4 The New Gimplifier Design

Here we describe the changes to the old gimplifier necessary to allow the improvements described in the previous section. We decided to extend and modify the code of the old gimplifier instead of writing a new gimplifier from the scratch, because the gimplifier contains a great amount of knowledge of the semantics of the trees that would be hard to preserve otherwise.

- Separate types for the various elements of GIMPLE grammar are introduced—`gimple_stmt`, `gimple_rhs`, `gimple_lhs`, etc. For now these are just aliases for the `tree` type, but this makes it easier to change the types in the future, as well as making the code easier to read by making the constraints on the operands easier to see. At the moment, we are not very strict with the casts between the types—we often use the `gimple_lhs` values as `gimple_rhs` if necessary, for example—but if we later change the types to the specialized ones, the type checking should make all the necessary changes a straightforward mechanical task.

- To conserve the memory, we still rewrite the trees in-place. However, we provide separate functions to generate most of the elements of GIMPLE grammar, called `gen_gimple_rhs_unary`, `gen_gimple_rhs_binary`, etc. The functions take the node to reuse as a separate parameters in addition to the operands and the code of the operation. This makes it possible to change the physical representation of the elements of the grammar easily, as well as to create different expressions in case constant propagation or CSE determines that it is possible to simplify the produced one.

- We preserve the `gimplify_expr` function, but in order to match its new semantics, we rename the function to `gimplify_to_void_or_rhs`. This function no longer takes a predicate to specify what its output is. This function returns (in addition to gimple statements realizing the side effects of the gimplified computation) a `rhs` expression to that the input evaluates if the caller asks for it. The caller may specify that the value is not used, in that case only the side effects are produced. Additionally, it is guaranteed that if the input to `gimplify_to_void_or_rhs` is `lhs`, the output is `lhs` as well—this is always possible to achieve, since `lhs` is a subset of `rhs` in the GIMPLE grammar.

  The functions `gimplify_to_void`, `gimplify_to_rhs`, `gimplify_to_lhs`, `gimplify_to_operand`, etc., are provided. These functions are used to transform its input to the prescribed form. They use the `gimplify_to_void_or_rhs` function internally and then process its output.

  In case some more precise subtype of the result is needed (for example, there are various forms of `rhs` allowed in the modify statement depending on the `lhs` and type of the expression), which the current gimplifier enables to express using various predicates (`is_gimple_formal_tmp_rhs`, `is_gimple_reg_rhs` and `is_gimple_mem_rhs` for the `rhs` element of the grammar, for example), the gimplification function needs to enforce such restriction itself by processing the result of `gimplify_to_rhs`. Functions performing this are implemented for the common cases.

- The `gimplify_to_void_or_rhs` function does not iterate. The gimplification functions for the elements of the GENERIC grammar are required to produce directly the GIMPLE statements and possibly the `gimple_rhs` result. In (very rare) cases when the iteration is the only practical way of achieving the required result, they must call `gimplify_to_void_or_rhs` recursively.

- Front-ends may define two language hooks—`genericize`, that should rewrite the passed language-specific tree node to GENERIC, and `gimplify_to_void_or_rhs`, that may produce the GIMPLE statements for the node directly. The transition to `genericize` hooks should be easier to perform, but longer term, the `gimplify_to_void_or_rhs` hooks should be preferred.

- Mostly unstructured GIMPLE is produced directly. The only statements left in structured form are the exception handling constructs (`try`, `finally` and `catch` blocks) that are lowered later by a separate (and quite complicated) pass. It might be possible to perform full lowering to unstructured GIMPLE, but it is not clear whether the increased complexity of the code obtained by merging two nontrivial tree rewriting passes is worth that.
- Output of the gimplification (representing basically the list of gimple statements) is represented by a `goutput` object that enables more precise information to be recorded with the output. A set of functions to manipulate `goutput` objects are provided, including adding statements of various types to the object and joining the objects one after another. These functions automatically gather and propagate the information from the statements.

The `goutput` object carries the following information:

stmts The list of GIMPLE statements represented by the object.

side_effects The list of postponed side effects, roughly corresponding to the `post` list of the current gimplifier. The function `gimplify_to_void_or_rhs` automatically adds those side effects that do not affect the value returned by it to the `stmts` list. Additionally, `goutput_sequence_point` function may be used to force evaluation of side effects—this function also replaces the references to the changed variables inside the currently processed expression by temporary variables containing the original values of the changed variable, in order to preserve the semantics. For example

```
if (a++ + b != 0) goto L;
```

becomes

```
tmp1 = a + b;
a = a + 1;
if (tmp1 != 0) goto L;
```

as we wanted, but

```
if (a++ != 0) goto L;
```

is transformed to

```
tmp1 = a;
a = a + 1;
if (tmp1 != 0) goto L;
```

as necessary to preserve the value of the comparison.

may_fallthru Specifies whether the control flow may continue directly after the object. Statements like `goto` set this flag to false, while labels set it to true. This flag is used for unreachable code elimination—when `may_fallthru` is false, we do not emit new statements up till the first label.

no_exit Specifies whether the last statement in the object postdominates the first one, i.e., whether there is no other exit from a block other than falling though it. This flag is used for unnecessary exception handling statements removal—if the `try` part of a `try-finally` block has `no_exit` set, the `try-finally` construction is not needed and we may just join the `finally` block directly after the `try` one.

consts A table of variables known to have constant value. Used by a constant propagation. The `gen_gimple_...` functions automatically try to substitute values from this table to their results. The functions that add statements to the object automatically update or invalidate entries in the table.

copies A table of variables known to be just copies of other variables, handled similarly to the `consts` table.

values A table of values of variables used by local value numbering and CSE. While the `consts` and `copies` tables are updated over simple control flow (structured `if` statements, etc.), this table is purely local to a single basic block in order to keep the optimization fast—the non-local optimization opportunities are left for the real optimization passes.

5 State of the Project

At the moment this paper is finished, we have implemented the core of the new gimplifier outlined in the previous section. This core is sufficient to gimplify most of the C language, and we are able to to compile libgcc and to pass most of the tests in the testsuite. None of the optimizations described above is implemented yet, except for the unreachable code removal and better handling of post-modify side effects. We hope to get the gimplifier to the state when some reasonable comparison with the old gimplifier is possible before the GCC summit, and present the results at the conference.

Our plans are:

- Fix up the corner cases by making GCC with the new gimplifier pass C testsuite and bootstrap.
- Implement the simple optimizations and cleanups described in this paper, and get comparison with the old gimplifier. If the results turn out to be positive,
- open the branch to encourage testing and contributions to the project,
- implement gimplification for the remaining languages and update the target gimplification hooks, and
- merge the changes to mainline.
- Proceed with the changes of the representation of GIMPLE.

References

[1] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, and Bhama Sridharan, *Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations*, In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, Number 757 in Lecture Notes in Computer Science, pp. 406–420, August 1992 (Proceedings published in 1993).

```
program        :  compound-stmt
compound-stmt  :  stmt
               |  STATEMENT_LIST members -> stmt
stmt           :  block
               |  if-stmt
               |  switch-stmt
               |  goto-stmt
               |  return-stmt
               |  label-stmt
               |  modify-stmt
               |  call-stmt
block          :  BIND_EXPR compound-stmt
if-stmt        :  COND_EXPR condition compound-stmt compound-stmt
switch-stmt    :  SWITCH_EXPR val labels
goto-stmt      :  GOTO_EXPR (LABEL_DECL | val)
return-stmt    :  RETURN_EXPR return-value
return-value   :  NULL
               |  RESULT_DECL
               |  MODIFY_EXPR RESULT_DECL lhs
label-stmt     :  LABEL_EXPR LABEL_DECL
modify-stmt    :  MODIFY_EXPR lhs rhs
call-stmt      :  CALL_EXPR (val | OBJ_TYPE_REF) call-arg-list
call-arg-list  :  TREE_LIST members -> lhs | CONST
lhs            :  addr-expr-arg | indirectref
addr-expr-arg  :  ID | compref
indirectref    :  INDIRECT_REF val
compref        :  min-lval
               |  COMPONENT_REF compref FIELD_DECL
               |  ARRAY_REF compref val
min-lval       :  ID | indirectref
condition      :  val | RELOP val val
val            :  ID | CONST
rhs            :  lhs
               |  CONST
               |  call-stmt
               |  ADDR_EXPR addr-expr-arg
               |  UNOP val
               |  BINOP val val
               |  RELOP val val
```

Figure 1: The simplified GIMPLE grammar

[2] Ana M. Erosa and Laurie J. Hendren, *Taming Control Flow: A Structured Approach to Eliminating Goto Statements*, in Proceedings of the 1994 International Conference on Computer Languages, Toulouse, France, pp. 229–240, May 1994.

[3] S. Muchnik, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[4] *GCC GIMPLE documentation*, `http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html#GIMPLE`.