GCC—An Architectural Overview, Current Status, and Future Directions

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

Abstract

The GNU Compiler Collection (GCC) is one of the most popular compilers available and it is the de facto system compiler for Linux systems. Despite its popularity, the internal workings of GCC are relatively unknown outside of the immediate developer community.

This paper provides a high-level architectural overview of GCC, its internal modules, and how it manages to support a wide variety of hardware architectures and languages. Special emphasis is placed on high-level descriptions of the different modules to provide a roadmap to GCC.

Finally, the paper also describes recent technological improvements that have been added to GCC and discusses some of the major features that the developer community is thinking for future versions of the compiler.

1 Introduction

The GNU Compiler Collection (GCC) has evolved from a relatively modest C compiler to a multi-language compiler that can generate code for more than 30 architectures. This diversity of languages and architectures has made GCC one of the most popular compilers in use today. It serves as the system compiler for every Linux distribution and it is also fairly popular in academic circles, where it is used for compiler research. Despite this popularity, GCC has traditionally proven difficult to maintain and enhance, to the extreme that some features were almost impossible to implement. And as a result GCC was starting to lag behind the competition.

Part of the problem is the size of its code base. While GCC is not huge by industry standards, it still is a fairly large project, with a core of about 1.3 MLOC. Including the runtime libraries needed for all the language support, GCC comes to about 2.2 MLOC.[1]

Size is not the only hurdle presented by GCC. Compilers are inherently complex and very demanding in terms of the theoretical knowledge required, particularly in the area of optimization and analysis. Additionally, compilers are dull, dreary and rarely provide immediate gratification, as interesting features often take weeks or months to implement.

Over the last few releases, GCC's internal infrastructure has been overhauled to address these problems. This has facilitated the im-

[1] Data generated using David A. Wheeler's 'SLOCCount'.

plementation of a new SSA based global optimizer, sophisticated data dependency analyses, a multi-platform vectoriser, a memory bounds checker (mudflap) and several other new features.

This paper describes the major components in GCC and their internal organization. Note that this is not intended to be a replacement for GCC's internal documentation. Many modules are overlooked or described only briefly. The intent of this document is to serve as introductory material for anyone interested in extending or maintaining GCC.

2 Overview of GCC

GCC is essentially a big pipeline that converts one program representation into another. There are three main components: *front end* (FE), *middle end* (ME)[2] and *back end* (BE). Source code enters the front end and flows through the pipeline, being converted at each stage into successively lower-level representation forms until final code generation in the form of assembly code that is then fed into the assembler.

Figure 1 shows a bird's eye view of the compiler. Notice that the different phases are sequenced by the Call Graph and Pass managers. The call graph manager builds a call graph for the compilation unit and decides in which order to process each function. It also drives the inter-procedural optimizations (IPO) such as inlining. The pass manager is responsible for sequencing the individual transformations and handling pre and post cleanup actions as needed by each pass.

The source code is organized in three major groups: core, runtime and support. In what follows all directory names are assumed to be relative to the root directory where GCC sources live.

[2]Consistency in naming conventions led to this unfortunate term.

2.1 Core

The `gcc` directory contains the C front end, middle end, target-independent back end components, and a host of other modules needed by various parts of the compiler. This includes diagnostic and error machinery, the driver program, option handling, and data structures such as bitmaps, sets, etc.

The other front ends are contained in their own subdirectories: `gcc/ada`, `gcc/cp` (C++), `gcc/fortran` (Fortran 95), `gcc/java`, `gcc/objc` (Objective-C), `gcc/objcp` (Objective C++), and `gcc/treelang`, which is a small toy language used as an example of how to implement front ends.

Directories inside `gcc/config` contain all the target-dependent back end components. This includes the machine description (MD) files that describe code generation patterns and support functions used by the target-independent back end functions.

2.2 Runtime

Most languages and some GCC features require a runtime component, which can be found at the top of the directory tree:

- The Java runtime is in `boehm-gc` (garbage collection), `libffi` (foreign function interface), `libjava` and `zlib`.
- The Ada, C++, Fortran 95 and Objective-C runtime are in `libada`, `libstdc++-v3`, `libgfortran` and `libobjc` respectively.

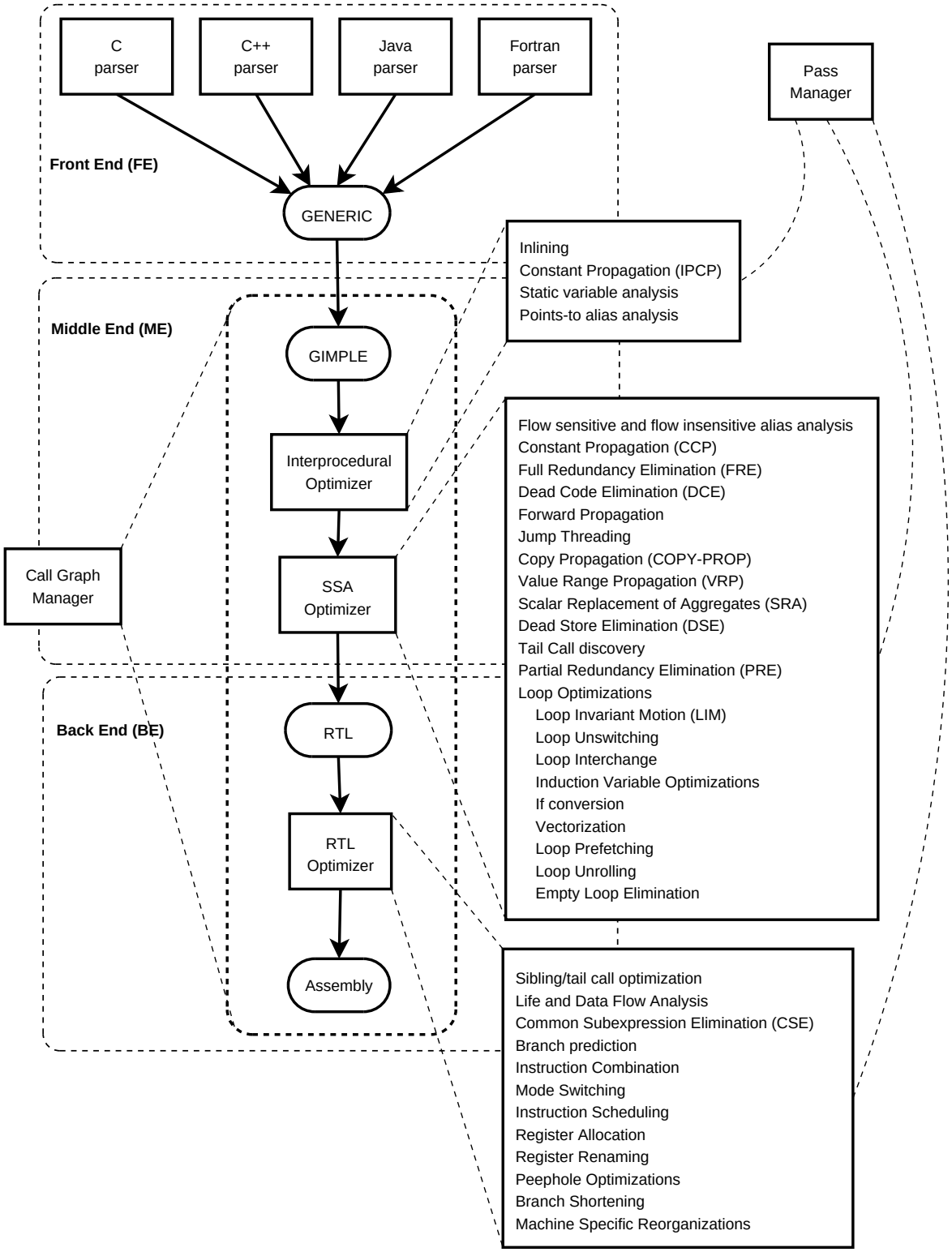

Figure 1: An Overview of GCC

The preprocessor is implemented as a separate library in `libcpp`.

A decimal arithmetic library is included in `libdecnumber`.

The OpenMP [14] runtime is in `libgomp`.

Mudflap [6], the pointer and memory check facility, has its runtime component in `libmudflap`.

The library functions for SSP (Stack Smash Protection) are in `libssp`.

2.3 Support

Various utility functions and generic data structures, such as bitmaps, sets, queues, etc. are implemented in `libiberty`. The configuration and build machinery live in `build` and various scripts useful for developers are stored in `contrib`.

2.4 Development Model

All the major decisions in GCC are taken by the GCC Steering Committee. This usually includes determining maintainership rights for contributors, interfacing with the FSF, approving the inclusion of major features and other administrative and political decisions regarding the project. All these decisions are guided by GCC's mission statement (`http://gcc.gnu.org/gccmission.html`).

GCC goes through three distinct development stages, which are coordinated by GCC's release manager and its maintainers. Each stage usually lasts between 3 and 5 months. During Stage 1, big and disruptive changes are allowed. This is where all the major features are incorporated into the compiler. Stage 2 is the stabilization phase, only minor features are allowed and bug fixes that maintainers consider safe to include. Stage 3 marks the preparation for release. During this phase only bug and documentation fixes are allowed. In particular, these bug fixes are usually required to have a corresponding entry in GCC's bug tracking database (`http://gcc.gnu.org/bugzilla`).

At the end of stage 3, the release manager will cut a release branch. Stabilization work continues on the release branch and a release criteria is agreed by consensus between the release manager and the maintainers. Release blocking bugs are identified in the bugzilla database and the release is done once all the critical bugs have been fixed.[3] Once the release branch is created, Stage 1 for the next release begins.

Using this system, GCC is averaging about a couple of releases a year. Once version *X.Y* is released, subsequent releases in the *X.Y* series continues. In this case, another release manager takes over the *X.Y* series, which accepts no new features, just bug fixes.

Major development that spans multiple releases is done in branches. Anyone with write access to the GCC repository may create a development branch and develop the new feature on the branch. When that feature is ready, they can propose including it at the next Stage 1. Vendors usually create their own branches from FSF release branches.

All contributors must sign an FSF copyright release to be able to contribute to GCC. If the work is done as part of their employment, their employer must also sign a copyright release form to the FSF.

[3] It may also happen that some of these bugs are simply moved over to the next release, if they are not deemed to be as critical as initially thought.

3 GENERIC Representation

Every language front end is responsible for all the syntactic and semantic processing for the corresponding input language. The main interface between an FE and the rest of the compiler is via the GENERIC representation [12]. Every front end is free to use its own internal data structures for parsing and validation. Once the compilation unit is parsed and validated, the FE converts its parse trees into GENERIC, a high-level tree representation where all the language-specific features are explicitly represented (e.g., exception handling, vtable lookups).

Due to historic reasons, most FEs use the `tree` data structure for representing their parse trees. However, the Fortran 95 FE uses its own data structures. This is a desirable property because it shields the FE from the rest of the compiler, providing a clean hand-off interface to the middle end via GENERIC.

While GENERIC provides a mechanism for a language front end to represent entire functions in a language-independent way, there are some features that are not representable in GENERIC. For instance, during alias analysis it is often necessary to determine whether two symbols of different types may occupy the same memory location. Each language has its own rules regarding type conflicts, so the compiler provides a call-back mechanism to query the front end. This mechanism is known as *language hook* or *langhook* and it is used whenever the compiler needs to involve the front end in some transformation or analysis.

All the language semantics must be explicitly represented in GENERIC, but there are no restrictions in how expressions are combined and/or nested. If necessary, a front end can use language-dependent trees in its GENERIC representation, so long as it provides a hook for converting them to GIMPLE. In particular, a front end need not emit GENERIC at all. For instance, in the current implementation, the C and C++ parsers do not actually emit GENERIC during parsing.

```
f(int a, int b, int c)
{
  if (g (a + b, c))
    c = b++ / a
  return c
}
```

Figure 2: A program in GENERIC form.

In practical terms GENERIC is a C-like language. A front end that wants to integrate with GCC can emit any of the tree codes defined in `tree.def` and implement the language hooks in `langhooks.h`. Figure 2 shows a code fragment in GENERIC.

4 GIMPLE Representation

GIMPLE is a subset of GENERIC used for optimization. Both its name and the basic grammar are based on the SIMPLE IR used by the McCAT compiler at McGill University [8]. Essentially, GIMPLE is a 3 address language with no high-level control flow structures:

1. Each GIMPLE statement contains no more than 3 operands (except function calls) and has no implicit side effects. Temporaries are used to hold intermediate values as necessary.

2. There are no lexical scopes.

3. Control structures are lowered to conditional gotos.

```
f (int a, int b, int c)
{
  t1 = a + b
  t2 = g (t1, c)
  if (t2 != 0)
    {
      c = b / a
      b = b + 1
    }
  else
    {

    }
  t3 = c
  return t3
}
```

(a) High GIMPLE.

```
f (int a, int b, int c)
{
  t1 = a + b
  t2 = g (t1, c)
  if (t2 != 0) <D1530> else <D1531>
<D1530>:
  c = b / a
  b = b + 1
<D1531>:
  t3 = c
  return t3
}
```

(b) Low GIMPLE.

Figure 3: High and Low GIMPLE versions for the code in 2.

4. Variables that need to live in memory are never used in expressions. They are first loaded into a temporary and the temporary is used in the expression.

There are two slightly different versions of GIMPLE used in GCC, namely High GIMPLE and Low GIMPLE. The main difference is that in High GIMPLE binding scopes like the body of an `if-then-else` construct are nested with the parent construct, while in Low GIMPLE binding scopes are completely linearized using labels and jumps. Figure 3(a) shows the High GIMPLE form for the code in Figure 2. The Low GIMPLE version is shown in Figure 3(b). The differences between Low and High GIMPLE are more noticeable when lowering other constructs like exception handling and OpenMP directives.

The process of lowering GENERIC into GIMPLE is known as *gimplification*. It recursively replaces complex statements with sequences of statements in GIMPLE form. The *gimplifier* lives in `gimplify.c` and `tree-gimple.c`. The lowering between High and Low GIMPLE is in `gimple-low.c`.

5 Call Graph

In order to perform interprocedural analyses and optimizations, GCC builds a call graph for the whole compilation unit.[4] This static call graph is used in two ways: to drive the sequence in which functions are optimized and to perform interprocedural optimizations. Each node in the call graph represents a function or procedure in the compilation unit and edges represent call operations. Data and attributes are stored both on nodes and edges.

After the front end is done parsing a function and producing the GENERIC form for it, the middle end is invoked to create the High GIMPLE form with a call to `gimplify_function_tree`. The gimplified function is then added to the call graph. Once all the functions have been parsed and added to the

[4] Only at `-O2` and higher.

call graph, the middle end invokes `cgraph_optimize`, which will:

1. Perform interprocedural optimizations with a call to `ipa_passes`.

2. Optimize every reachable function in the call graph with a call to `cgraph_expand_function` which in turn calls `tree_rest_of_compilation` to execute all the intraprocedural transformations by calling the pass manager (`execute_pass_list`).

The implementation of the call graph lives in `cgraph.c` and `cgraphunit.c`. All passes executed by the pass manager are defined and registered by `init_optimization_passes` in `passes.c`.

6 SSA Form

After a function is in Low GIMPLE form, the pass manager will build the Control Flow Graph (CFG) and rewrite the function in Static Single Assignment (SSA) form [3], which is the main data structure used for analysis and optimization in the middle end.

The SSA form is a representation that exposes data flow by linking read and write operations using a static versioning scheme. When a variable V is the target of a write operation, a new version number is created for V and labeled V_i. Subsequent read operations, without an intervening assignment to V, are modified to use V_i. For example,

```
foo (int a, int b, int c)
{
  a_1 = 3;
  b_2 = 5;
  c_3 = a_1 + b_2;
  a_4 = c_3;
  return a_4;
}
```

When analyzing the statement $c_3 = a_1 + b_2$, the optimizer can easily determine that a_1 can be replaced with 3 and b_2 with 5 because both SSA names are guaranteed to be assigned exactly once. But, in many cases, conditional control flow makes it impossible to statically determine the most recent version of a variable. For instance,

```
foo (int a, int b, int c)
{
  if (c_3 > 10)
    c_4 = a_1 + b_2;
  else
    c_5 = a_1 - b_2;
  c_6 = PHI <c_4, c_5>
  return c_6;
}
```

Since it is not possible to statically determine which branch will be taken at runtime, we don't know which of c_4 or c_5 to use at the return statement. So, the SSA renamer creates a new version, c_6, which is assigned the result of "merging" c_4 and c_5. This PHI function tells the compiler that at runtime c_6 will be either c_4 or c_5.

GCC uses two different SSA variants, a rewriting and a non-rewriting form. In the rewriting form, symbols are replaced with their corresponding SSA names. Each SSA name is considered a distinct object and, as such, code motion transformations are allowed to cause two or more SSA versions for the same symbol to

be simultaneously live. When the function is taken out of SSA form, the SSA names are converted into regular symbols and new artificial symbols are created as needed to satisfy live range requirements [13]. In the non-rewriting form, SSA names are only used to connect variable uses to their definition sites, distinct SSA names for the same symbol are not allowed to have overlapping live ranges, and so, when taking the function out of SSA form the SSA names are simply discarded.

The rewriting form is applied to local scalar variables that may not be modified in ways unknown to the compiler. That is, they may not be modified as a side-effect to a function call, their address has not been taken by any pointer and every read/write operation references the whole object. GCC labels these variables as *GIMPLE registers*.[5] Operand statements that use GIMPLE registers are referred to as *real operands*, and so the rewriting SSA form in GCC is also known as *real SSA form*. The previous two code fragments are examples of GCC's real SSA form.

In contrast, the non-rewriting SSA form is used on *memory* symbols. These are variables that may be modified in ways unknown to the compiler. That is, they may be clobbered by a function call or their address has been taken or they may be partial references to the object (e.g. symbols of aggregate types like structures and arrays). Since statements may have implicit references to memory symbols, GCC represents them with special *virtual operators* attached to the statement. There are two such operators: V_MAY_DEF to indicate a partial and/or potential store to the object, and VUSE to indicate partial and/or potential load from the object. This non-rewriting form is known in GCC as *virtual SSA form*.

[5]This does not imply that the object will actually be allocated a physical register.

For example, in the following code fragment variable *A* is a global variable that may be clobbered by function *foo*, so when calling *foo*, GCC indicates that *A* may be modified by it with a V_MAY_DEF operator, resulting in the following virtual SSA form for *A*:

```
int A;

bar ()
{
  # A_2 = V_MAY_DEF <A_1>
  A = 9

  # A_3 = V_MAY_DEF <A_2>
  foo ()

  # VUSE <A_3>
  x_4 = A
  return x_4
}
```

Notice that the virtual SSA form not only links uses to definitions (*use-def chains*). It also links definitions to other definitions (*def-def chains*). This is necessary because V_MAY_DEF operators represent partial/potential stores. In the previous code fragment, the store $A = 9$ may or may not reach the $x_4 = A$ statement, so it is necessary to link A_3 and A_2, or transformations like dead-code elimination would eliminate the statement $A = 9$.

The CFG, SSA form and related facilities (such as incremental SSA updates) are implemented in `tree-cfg.c`, `tree-into-ssa.c`, `tree-ssa.c`, and `tree-outof-ssa.c`.

6.1 Aliasing

GCC uses two mechanisms for representing aliasing: *query* or *oracle* based, used during RTL optimization and an explicit representation used during GIMPLE optimization.

The query based mechanism provides functions that, given two memory references will determine whether they may overlap or not. Currently, the analysis done by this mechanism is mostly type and structural based. If the two memory references are to objects whose types may not alias according to the rules of the language, then they may not overlap. The basic mechanism uses the notion of *alias set numbers*, which are associated to the type of the memory location and organized in a hierarchical structure according to the rules of the input language. Given two memory references, the function `alias_sets_conflict_p` determines whether they may occupy the same memory slot based on their alias set numbers. The file `alias.c` contains the implementation of this mechanism.

The explicit representation is used during GIMPLE optimization and it takes advantage of the virtual SSA representation. After the code is put into SSA form, an alias analysis pass (`pass_may_alias`) computes points-to information for all referenced pointers. This is used to build flow-sensitive and flow-insensitive alias sets that are then associated with special symbols called *memory tags* that represent pointer dereferences. When a pointer is dereferenced, the compiler will look-up its associated memory tag, determine what symbols belong to the alias set and insert virtual operators for every symbol in the alias set.

Each mechanism has its strengths and weaknesses. The advantage of an explicit representation is that optimizers need not concern themselves with possible aliasing problems when doing a transformation. On the other hand, an explicit representation takes up memory and compile time (unbearably so in some extreme cases), so it may become an unnecessary burden. For example, consider the function in Figure 4.[6] To illustrate the difference between the explicit representation and the query based mechanism, consider the problem of re-ordering the store operations at lines 6 and 7. With a query based mechanism, the transformation pass should call `alias_sets_conflict_p` on the memory references given by $*p_2$ and $*q_3$.

```
1  foo (int i, float *q)
2  {
3    int a, b, *p;
4
5    p_2 = (i_1 > 10) ? &a : &b
6    *p_2 = 42
7    *q_3 = 0.42
8    return *p_2
9  }
```

Figure 4: Query-based alias analysis requires no additional operators in the IL.

```
1  foo (int i, float *q)
2  {
3    int a, b, *p;
4
5    p_2 = (i_1 > 10) ? &a : &b

     # a_5 = V_MAY_DEF <a_4>
     # b_7 = V_MAY_DEF <b_6>
6    *p_2 = 42

     # MT_9 = V_MAY_DEF <MT_8>
7    *q_3 = 0.42

     # VUSE <a_5>
     # VUSE <b_7>
8    return *p_2
9  }
```

Figure 5: Explicit representation of aliasing using virtual SSA form.

But that relation is made explicit when the program is in virtual SSA form (Figure 5). Since p_2 points to one of a or b, line 6 contains one virtual operator for each variable. On the other hand, pointer q_3 does not really point to any

[6] SSA form redacted for simplicity.

variable in function *foo*, so dereferencing q_3 is represented with a virtual operator to its memory tag (MT). Given this, the stores at line 6 and 7 do not conflict and so the transformation can proceed safely.

While the explicit representation has several advantages over the query mechanism, all the virtual operators required and their PHI nodes will add more bulk to the IR, resulting in increased compile time and memory consumption (we are currently working on a new representation to alleviate this problem).

6.2 SSA Optimizers

In general, transformations at the GIMPLE level are target-independent because the IL does not expose attributes such as word size, address arithmetic, registers, calling conventions, etc. However, there are transformations that need to span multiple IL abstractions to work properly. One of the prime examples is vectorization. The analysis required to determine whether some code may be vectorized requires high-level dataflow information that is available in GIMPLE. However, the actual transformation depends on hardware capabilities, such as size of vector registers and available vector operations. This communication is done via special IL codes and call backs between the middle end and back end.

A variety of SSA-based analyses and optimizations have been implemented on the GIMPLE representation (Figure 1). Together with other cleanup passes and the fact that some of them are executed more than once, the middle end pipeline runs to about 100 stages. Some of the more notable transformations include

Vectorization [15] supporting multiple architectures.

Loop optimizations based on chains of recurrences to recognize scalar evolutions and track induction variables [2].

Traditional scalar optimizations, including constant/copy propagation, dead code elimination, full and partial redundancy elimination, value range propagation, scalar replacement of aggregates, jump threading, forward propagation and dead store elimination.

Flow sensitive and flow insensitive alias analysis, including field-sensitive points-to analysis for aggregates [1].

Automatic instrumentation for pointer checking for C and C++ [6].

7 RTL

Register Transfer Language (RTL) is the original intermediate representation used by GCC. It was developed at the University of Arizona as part of their research in re-targetable compilers in the early 80s [4, 5]. It is also used by VPCC (Very Portable C Compiler). Basically, RTL is an assembler language for an abstract machine with an infinite number of registers. As opposed to the C-like representation used by GENERIC and GIMPLE, RTL resembles Lisp (although it is possible to obtain an assembly-like rendering for debugging purposes). The code fragment in Figure 6(a) shows a GIMPLE code fragment and its conversion to RTL (Figure 6(b)). The exact RTL will contain more detailed information and may vary from one processor to another.

RTL is designed to abstract hardware features such as register classes, memory addressing modes, word sizes and types (machine modes), compare-and-branch instructions, calling conventions, bitfield operations, type and sign

```
if (a > 10) <L1> else <L2>;

<L1>:

b = a - 1;

<L2>:
```

(a) GIMPLE version.

```
(insn 20 18 21 3 (set (reg:CCGC 17 flags)
        (compare:CCGC (reg/v:SI 60 [ a ])
            (const_int 10 [0xa]))))

(jump_insn 21 20 22 3 (set (pc)
        (if_then_else (le (reg:CCGC 17 flags)
                          (const_int 0 [0x0]))
            (label_ref 26)
            (pc))))

(code_label 22 21 23 4 3 "" [0 uses])

(insn 25 23 26 4 (parallel [
            (set (reg/v:SI 59 [ b ])
                (plus:SI (reg/v:SI 60 [ a ])
                    (const_int -1 [0xffffffff])))
            (clobber (reg:CC 17 flags))
        ]))

(code_label 26 25 27 5 2 "" [1 uses])
```

(b) RTL version.

Figure 6: GIMPLE and RTL variants of a conditional branch.

conversions, and generic instruction patterns. These abstractions are defined and controlled by an elaborate pattern matching mechanism defined in a *machine description* (MD) file, which defines all the necessary code generation mappings between the back end and the target processor.

Machine description files together with other files needed to generate target code are commonly referred-to as *ports*. They are stored in sub-directories under `gcc/config/`. Currently, GCC contains more than 30 such ports, and while the implementation of a new port is not a trivial task, it is perhaps one of the aspects of GCC with the most extensive available documentation (`http://gcc.gnu.org/onlinedocs/`). There are two main components that make up a port:

Instruction templates define the mappings between generic RTL and the target machine. For instance, the `define_insn` pattern in Figure 7 describes a typical 32-bit addition operation for a RISC processor. The top portion defines the pattern to be matched. In this case, it's looking for $x = y + z$ where all the operands are word-sized (SI), general purpose registers (`"register_operand" "r"`). It also indicates that the x operand is modified by the instruction (`"=r"`). The bottom portion indicates the final assembly code that should be emitted when this pattern is matched (`add x, y, z`).

Target description macros describe hardware capabilities such as register classes, calling conventions, data types and sizes, predicates that validate moves between memory and registers, etc.

7.1 RTL passes

Historically, all the optimization work was done in RTL, but the current trend is to

```
(define_insn "addsi3"
 [(set (match_operand:SI 0 "register_operand" "=r")
       (plus:SI (match_operand:SI "register_operand" "r")
                (match_operand:SI "register_operand" "r")))]
 ""
 "add %0, %1, %2")
```

Figure 7: An RTL code generation pattern for 32-bit addition.

move most of the heavy lifting from RTL into the GIMPLE optimizers (currently there are around 60 RTL passes and more than 100 GIMPLE passes). The final goal is to implement analyses and transformations at the right level of abstraction. RTL is ideally suited for low-level transformations such as register allocation and scheduling, but most of the generic transformations are more efficient to implement in GIMPLE.

RTL and GIMPLE also share common infrastructure code such as the pass and call graph managers, the flowgraph, dominance information and type-based aliasing. Some of the main transformations done in RTL include:

Register allocation. Arguably, one of the more complex passes in GCC. It is organized as a multi-pass allocator: a local pass (`local-alloc.c`) allocates registers within a basic block, and a global pass (`global.c`) which works across basic block boundaries. The actual code modification is done by a third pass known as *reload* (`reload.c` and `reload1.c`). Most of the complexity in register allocation lies in the multitude of targets supported by GCC. Every target will have its own set of register classes and rules for moving values between registers and memory. Several efforts exist to re-implement this pass, but is generally considered to be a fairly difficult problem [9, 10, 11, 16].

Scheduling. This pass tries to take advantage of the implicit parallelism provided by the multiple functional units in modern processors that allow the simultaneous execution of multiple instructions. The scheduler rearranges the instructions according to data dependency restrictions in an effort to increase instruction parallelism. The scheduler is implemented in `haifa-sched.c` and `sched-rgn.c`.

Software pipelining is implemented using Swing Modulo Scheduling (SMS) [7]. This pass complements instruction scheduling by improving parallelism inside loops by overlapping the execution of instructions from different loop iterations.

Other optimizations include common subexpression elimination, instruction recombination, mode switching reduction, peephole optimizations and machine specific reorganizations.

8 Current Status and Future Work

The open development model used by GCC has all the usual advantages of other FOSS projects. It attracts a wide variety of developers and since it is the system compiler of every Linux distribution, it is a fairly stable and robust compiler. Furthermore, the wide variety of supported languages, platforms and flexible architecture makes it a compelling option for both industry and academic compiler projects.

GCC has changed quite significantly in the last 3–4 years. The addition of GENERIC, GIMPLE and the SSA framework allowed the development of features that had traditionally been considered difficult or impossible to implement, including vectorization, OpenMP and advanced loop transformations.

8.1 New languages

The introduction of the GENERIC representation has further simplified the task of introducing a new language front end to GCC. The increased separation between the front end and the rest of the compiler provides a lot of independence to language designers. While we do not claim GENERIC to be the perfect target for all languages, it has proven to be sufficiently flexible for the variety of languages currently supported by GCC, including C, C++, Java, Ada, Objective-C/C++ and Fortran 95.

The addition of new languages may require extending and/or adapting GENERIC. In terms of language-specific analyses and transformations, GCC's strategy is to, as much as possible, implement in GIMPLE where all the data and control-flow information is gathered and use *langhooks* to communicate with the front end when necessary. Most transformations in this area are expected to be in the area of abstraction removal such as method devirtualization in OO languages. There is also interest in more sophisticated escape analysis for languages like Java.

8.2 Internal modularity

The basic compiler infrastructure is encapsulated as much as C allows. While this remains one of the weakest points in the implementation, we try to draw strict API boundaries to abstract the major conceptual modules, such as call graph, control flow graph, intermediate representation, fundamental data types (bitmaps, hash tables), SSA form, etc.

GCC would probably benefit from switching to an implementation language with more capabilities, such as C++. In fact, the topic comes up every now and again on the development lists. The consensus seems to be in favor of switching, but 1+ million lines of code represent a lot of inertia to overcome, and implementation discipline can go a long way. Not every module of the compiler is implemented in C, however. Front ends, for instance, are free-to-use different implementation languages (Ada being the prime example).

8.3 High performance computing

With the advent of multi-core processors, optimizations and languages that take advantage of task/data parallelism will become increasingly important. GCC includes a multi-platform vectoriser that is unique in its class and starting with version 4.2, it will include support for OpenMP (`http://www.openmp.org`), which provides compiler directives for specifying parallelism in C, C++ and Fortran.

Other important features for future releases include an automatic parallelization option built on top of the OpenMP framework, memory locality optimizations, more advanced loop optimizations and additional vectorization improvements.

8.4 Static analysis tools

This is another area that is starting to gain widespread interest. Compilers are in a unique position to provide this facility because they already have a synthetic representation of the

input program. However, the set of interesting analyses to perform may vary widely, some people will want to check for security problems, others may want to enforce coding guidelines, others may want to check for buffer overflows, etc.

GCC already includes some of the more commonly requested features such as pointer checking and buffer overflow prevention. But there are other types of checks specific enough that it usually does not make sense to include in the compiler. At the same time, people interested in them may not have the interest nor the time to invest in delving inside the compiler to implement their analysis.

There are plans to provide some form of extensibility mechanism so that external developers would be able to connect ad-hoc analysis code by interfacing with GCC.

8.5 Dynamic Compilation

Static compilation techniques are generally believed to have reached a saturation point. Compilers do not have a sufficiently global view of the program to perform more aggressive optimizations. Modern software is usually spread over many files and it likely uses many services from shared libraries, all of which is hidden from the compiler at compile time. And since most libraries use dynamic linking. the code may even be hidden from the compiler at link time.

To compound this problem, languages like Java and C# have fairly powerful dynamic properties, such as class loading. Therefore, static compilers may only see a small portion of the whole program. All this provides a big incentive to move parts of the compilation process into the runtime system. This is generally known as Just-In-Time compilation (JIT). These systems work on top of a bytecode language and virtual machine which converts the bytecodes into native form at runtime. While this provides a lot of flexibility in terms of portability and dynamic features, the runtime overhead can be pretty significant, so hiding compile time latencies becomes fundamentally important in these environments.

We are currently planning to extend GCC to support such dynamic compilation schemes. At the time of this writing, we still do not have any concrete plans, but it is certainly an area in which we plan to take GCC in the medium to long term.

References

[1] D. Berlin. Structure Aliasing in GCC. In *Proceedings of the 2005 GCC Summit*, Ottawa, Canada, June 2005.

[2] D. Berlin, D. Edelsohn, and S. Pop. High-Level Loop Optimizations for GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] J. W. Davidson and C. W.Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[5] J. W. Davidson and D. B. Whalley. Quick Compilers Using Peephole

Optimization. *Software - Practice and Experience*, 19(1):79–97, 1989.

[6] F. Ch. Eigler. Mudflap: Pointer Use Checking for C/C++. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[7] M. Hagog and A. Zaks. Swing Modulo Scheduling for GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[8] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. 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*, pages 406–420. Lecture Notes in Computer Science, no. 457, Springer-Verlag, August 1992.

[9] V. Makarov. Fighting Register Pressure in GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[10] V. Makarov. Yet Another GCC Register Allocator. In *Proceedings of the 2005 GCC Summit*, Ottawa, Canada, June 2005.

[11] M. Matz. Design and Implementation of the Graph Coloring Register Allocator for GCC. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, June 2003.

[12] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[13] D. Novillo. Design and Implementation of Tree SSA. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[14] D. Novillo. OpenMP and automatic parallelization in GCC. In *Proceedings of the 2006 GCC Summit*, Ottawa, Canada, June 2006.

[15] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *The 4th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2006.

[16] M. Punjani. Register Rematerialization in GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.