Tree SSA – A New High-Level Optimization Framework for the GNU Compiler Collection*

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

Abstract

In this paper we introduce Tree SSA, a new optimization framework for the GNU Compiler Collection (GCC) based on the Static Single Assignment form. The paper provides a brief historical perspective on GCC's development, the rationale behind the new framework and its potential applications. We will also discuss some of the analyses and optimizations that are being designed and implemented on top of Tree SSA.

1 Introduction

In its 15 year history, 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.

The compiler is designed around two main components: the *front end*, which handles the processing of the source language, and the *back end*, which handles code optimization and final code generation. These two phases operate on a language-independent representation of the program called RTL (Register Transfer Language). The diagram in Figure 1 provides an overview of the compilation process in GCC.

This separation between language-dependent and language-independent phases has facilitated the creation of different language front ends and of several ports to different architectures. Unfortunately, GCC's optimization framework has not evolved in the same way. The main goal of this project is to develop a new framework for GCC to allow the implementation of high-level analyses and optimizations (that is, closer to the source). We aim to design and implement the new infrastructure around well-known, published techniques. This will improve our ability to add new features and maintain the compiler.

2 Overview of the optimization process

A compiler analyzes an input program written in one language (source code) and transforms it into a semantically equivalent program in another language (object code). During this translation an optimizing compiler applies certain transformations to the input program to improve its efficiency, which usually means improving runtime performance.

*NordU/USENIX 2003, Västerås, Sweden

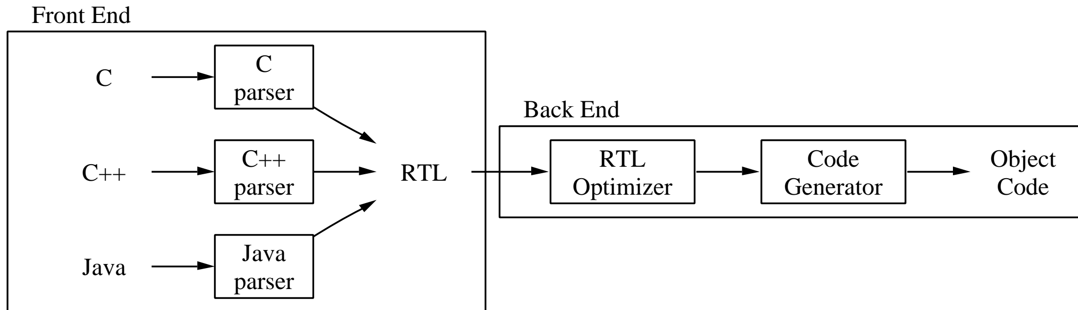

Figure 1: Existing compilation process in GCC.

Once the program syntax has been verified, the compiler generates intermediate code that is a more concise representation of the original program. All the analyses and transformations performed by the compiler are applied to this intermediate representation. The amount of detail provided by the representation depends on the type of optimization being performed. Optimizing compilers typically have more than one intermediate representation, each suited for different transformations. GCC uses two different representations, namely abstract syntax trees (also known as *trees*) and RTL (Register Transfer Language).

The front end knows very little about what the program actually does. Optimization is possible when the compiler understands the flow of control in the program (control-flow analysis) and how the data is transformed as the program executes (data-flow analysis). Analysis of the control and data flow of the program allows the compiler to improve the runtime performance of the code. Many different optimizations are possible once the compiler understands the control and data flow of the program. The following are a few of the most popular optimization techniques used in standard optimizing compilers:

Algebraic simplifications. Expressions are simplified using algebraic properties of their operators and operands. For instance, $i + 1 - i$ is converted to 1. Other properties like associativity, commutativity and distributivity are also used to simplify expressions.

Constant folding. Expressions for which all operands are constant can be evaluated at compile time and replaced with their value. For instance, the expression $a = 4 + 3 - 8$ can be replaced with $a = -1$. This optimization yields best results when combined with *constant propagation.*

Redundancy elimination. There are several techniques that deal with the elimination of redundant computations. Some of the more common ones include:

Loop-invariant code motion. Computations inside loops that produce the same result for every iteration are moved outside the loop.

Common sub-expression elimination. If an expression is computed more than once on a specific execution path and its operands are never modified, the repeated computations are

replaced with the result computed in the first one.

Partial redundancy elimination. A computation is partially redundant if some execution path computes the expression more than once. This optimization adds and removes computations from execution paths to minimize the number of redundant computations in the program. It encompasses the effects of loop-invariant code motion and common sub-expression elimination.

A final translation phase produces machine or assembly code for the target architecture. Further optimizations are enabled during this translation. Register allocation and code scheduling are typically applied during this phase. Code scheduling refers to a family of instruction reordering techniques that take advantage of specific features of the processor (for example, pipelining, VLIW, super-scalar features, etc).

3 Optimizing the tree representation

After GCC's front end parses the input program, the parse trees created by the parser are transformed into a language-independent representation called Register Transfer Language (RTL), which is ultimately optimized and converted into the target's native code. RTL is an intermediate representation that can be thought of as an assembly language for a machine with an infinite number of registers.

Being a low-level representation, RTL works well for optimizations that are close to the target (for example, register allocation, delay slot optimizations, peepholes, etc). However, many optimizing transformations need higher level information about the program that is not possible (or very difficult) to obtain from RTL (for example, array references, data types, references to program variables, control flow structures). Over time, some of these transformations have been implemented in RTL, but since the data structure is not really prepared for this, the end result is code that is excessively convoluted, hard to maintain and error prone.

To avoid the limitations of RTL with respect to analyses and transformations closer to the source, we decided to start with the other intermediate representation that GCC builds during compilation, namely Abstract Syntax Trees (AST or Trees). GCC's Tree representation contains detailed information about data types, variables and control structures of the original program. Optimizing the tree representation in GCC is appealing because, (a) it facilitates the implementation of new analyses and optimizations closer to the source, (b) it simplifies the work of the RTL optimizers, potentially speeding up the compilation process or improving the generated code.

Although GCC parse trees provide very detailed information about the original program, they are not suitable for optimization because of the following reasons:

1. **Lack of a common representation**. There is no single tree representation shared by all the front ends. This means that each language would require a different implementation of the same infrastructure. This would be a maintenance nightmare and would make it very difficult to add new front ends to

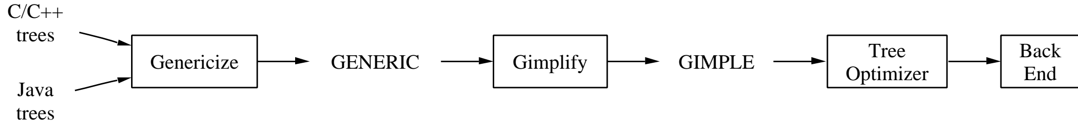

Figure 2: Proposed tree optimization process.

GCC.

2. **Side effects**. Parse trees are allowed to have side effects. For instance, the following expression will have different semantics in GCC's front ends for C and Java

```
b = a + ++a
```

Note that this expression is not valid according to the C language standard. Its behaviour is undefined. However, it helps illustrate the point about the different front ends. GCC's C front end evaluates the pre-increment operator before the assignment. Therefore, the above assignment is equivalent to

```
a = a + 1
b = a + a
```

In Java, however, the expression is evaluated from left to right. Therefore, the above assignment is equivalent to

```
tmp = a
a = a + 1
b = tmp + a
```

This means that the tree analysis and optimization phases would have to understand the semantics of every source language, which takes us to the multiple implementation scenario described above.

3. **Structural complexity**. Parse trees may combine in arbitrarily complex patterns, which may obfuscate the control flow of the program. For instance, the following expression is represented in a single parse tree

```
if ((a = (b > 5) ?  c :  d) >
10)
    ...
```

When building the control flow graph for this code fragment, the compiler will need to realize that the predicate for the `if()` statement contains different flows of control of its own. Furthermore, this expression requires more than one basic block to be represented, which is a complete impossibility.

To overcome these limitations, we have introduced two new tree-based intermediate representations called GENERIC and GIMPLE. GENERIC addresses the lack of a common tree representation among the various front ends. GIMPLE solves the side-effect and complexity problems that facilitate the discovery of data and control flow in the program. Both are described in the next section.

3.1 GENERIC and GIMPLE

Although some front ends share the same tree representation, there is no single representation used by all GCC front ends. Every front end is responsible

```
1 a = foo ()
2 b = a + 10
3 c = 5
4 if (a > b + c)
5   c = b++ / a + (b * a)
6 bar (a, b, c)
```

(a) GENERIC form.

```
1  a = foo ()
2  b = a + 10
3  c = 5
4  T1 = b + c
5  if (a > T1)
6    {
7      T2 = b / a
8      T3 = b * a
9      c = T2 + T3
10     b = b + 1
11   }
12 bar (a, b, c)
```

(b) GIMPLE form.

Figure 3: First stage of analysis. Conversion to GIMPLE form.

from translating its own parse trees directly into RTL. To address this problem, we have introduced a new representation, named GENERIC, that is merely a superset of all the existing tree representations in GCC. Instead of generating RTL, every front end is responsible for converting its own parse trees into GENERIC trees. Since GENERIC trees are language-independent, all the semantics of the input language must be explicitly described by each front end. Some of these semantics are translated in the "genericize" pass, while others are translated in the "gimplification" pass.

The conversion to GENERIC removes language dependencies from the program representation. However, it does not address the structural complexity problem described in the previous section. Before manipulating the representation, the compiler breaks down GENERIC trees into a simpler representation that is ultimately analyzed and optimized (Figure 2). This new representation, called GIMPLE, is lexically identical to GENERIC but it has a different grammar that is derived from the SIMPLE representation used by McGill University's McCAT compiler [3].

To illustrate the differences between GENERIC and GIMPLE, consider the following program. Figure 3(a) shows the original program in GENERIC form, while Figure 3(b) shows the same program in GIMPLE form. Notice how the GIMPLE version is equivalent, but individual expressions are simpler and more regular in structure. For instance, a statement in GIMPLE form is guaranteed to have no more than three variable references. GIMPLE expressions are also guaranteed to contain no side-effects (for example, the post-increment operation in line 5 of Figure 3(a) has been explicitly exposed by the conversion to GIMPLE form).

3.2 The Control Flow Graph

The *Control Flow Graph* (CFG) is a directed graph that models the execution of the program. Each node in the CFG,

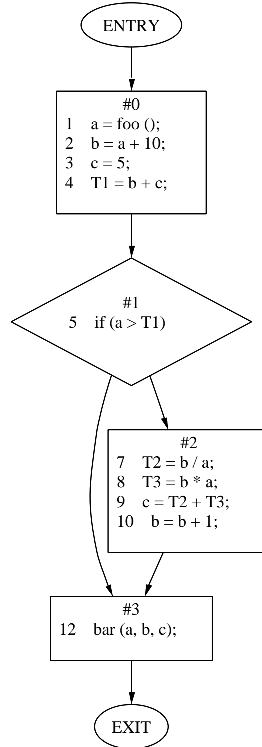

Figure 4: CFG for the program in Figure 3(b).

called a *basic block*, represents a non-branching sequence of statements (execution starts with the first instruction in the group and it leaves the block only after the last instruction has been executed). The edges of the graph represent possible execution paths in the flow of control (conditionals, loops, etc.).

The control flow graph for the example program in Figure 3(b) is shown in Figure 4. As part of the optimization infrastructure, we are implementing the necessary support data structures and functions needed to manipulate the CFG and the statements contained in basic blocks.

3.3 Static Single Assignment form

The CFG summarizes the possible execution paths of the program, but optimizers also need information about data flow (where variables are modified and used). To represent the flow of data in the program, we use a representation called Static Single Assignment (SSA). This is a relatively new intermediate representation that is becoming increasingly popular because it leads to efficient algorithmic implementations of data flow analyzers and optimizing transformations [2]. The SSA form is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of the variable. In essence, the SSA form links every use of a

```
1  a = foo ()
2  b = a + 10
3  c = 5
4  T1 = b + c
5  if (a > T1)
6    {
7      T2 = b / a
8      T3 = b * a
9      c = T2 + T3
10     b = b + 1
11   }
12 bar (a, b, c)
```

(a) Original GIMPLE program.

```
1  a_1 = foo ()
2  b_1 = a_1 + 10
3  c_1 = 5
4  T1_1 = b_1 + c_1
5  if (a_1 > T1_1)
6    {
7      T2_1 = b_1 / a_1
8      T3_1 = b_1 * a_1
9      c_2 = T2_1 + T3_1
10     b_2 = b_1 + 1
11   }
12 b_3 = φ(b_1, b_2)
13 c_3 = φ(c_1, c_2)
14 bar (a_1, b_3, c_3)
```

(b) Program in SSA form.

Figure 5: Static Single Assignment form for the program in Figure 3(b).

variable in the program to its corresponding unique definition. This allows very efficient implementations of analyses and optimizing transformations.

Actual programs are seldom in SSA form initially because variables tend to be assigned multiple times; not just once. An SSA-based compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment. Notice that it is not always possible to statically determine what is the most recent assignment for a given use. These ambiguities are the result of branches and loops in the program's flow of control. To solve them, the SSA form introduces a new type of operation called ϕ (phi) functions. ϕ functions merge multiple incoming assignments to generate a new definition; they are placed at points in the program where the flow of control causes more than one assignment to be available.

Figure 5 shows the program from Figure 3(b) and its corresponding SSA form (Figures 5(a) and 5(b) respectively). Notice that every assignment in the program introduces a new version number for the corresponding variable. Every time a variable is used, its name is replaced with the version corresponding to the most recent assignment for the variable. Now consider the use of variable b in the call to $bar()$ (line 12). There are two assignments to b that could reach line 12; the assignment at line 2 and the assignment inside the `if` statement at line 10. To solve this ambiguity, SSA introduces a new artificial definition for b by means of a ϕ function. This new definition merges both assignments to create a new version of b (b_3). The semantics of the ϕ function dictate that b_3 will take the value from one of the function's arguments. The specific argument returned by the ϕ function is not known until runtime.

It is important to note that ϕ functions are not actually emitted in the final code. They are only an analysis artifact used by the compiler to maintain the single assignment property of the program and summarize the presence of multiple definitions available at a specific point in the program. The current SSA implementation is based on the work on factored use-def chains (FUD chains) [4].

4 Implementation

The concepts described in the previous sections are being implemented on a development branch of GCC. The implementation consists of two fundamental components:

1. The **base infrastructure** provides a core of basic functions and data structures to analyze and manipulate the program.
2. The **optimizers** transform the program using the information provided by the base infrastructure.

There are three main components to the basic infrastructure: the gimplifier, the control flow graph module and the SSA module. Each module builds on top of the previous one.

- The gimplifier module is responsible for converting the GENERIC representation into GIMPLE. It also provides functions for generating GIMPLE statements and testing whether a given statement or expression is in GIMPLE form.
- The CFG module builds the flow graph for the program and provides functions for manipulating it. Additionally, it can simplify the flow of the program in the presence of unreachable regions or control expressions made superfluous by an optimization.
- The SSA module finds all the variables referenced in the program and builds the SSA form described in section 3.3. It provides all the necessary functions and data structures to compute, among other things, aliasing, reaching definitions, and reached-uses information. It also provides the functionality needed to update and/or rebuild data flow information after an optimization makes changes to the program.

Currently, we have three optimizations at various degrees of development:

1. Sparse Conditional Constant Propagation (CCP) [5] is an efficient formulation of the constant propagation problem that is also capable of finding constant conditionals and unreachable code.
2. Partial Redundancy Elimination (PRE) [1] finds expressions that are computed more than once and re-writes them so that their values are computed once and re-used as necessary. In addition to removing completely redundant computations, PRE has the ability to make partially redundant computations fully redundant, thus combining the effects of global common subexpression elimination and loop invariant code motion.
3. Dead Code Elimination (DCE) [2] removes all statements in the program that have no effect on its output (assignments to variables that

```
bar()
{
  T1 = "a = %d, b = %d, c = %d\n";
  T2 = (char *)T1;
  T3 = (const char *)T2;
  printf (T3, a, b, c)
}

foo()
{
  return 0;
}

main()
{
  a = foo ();
  b = 19;
  c = 5;
  T4 = b + c;
  if (a > T4)
    {
      T5 = b / a;
      T6 = b * a;
      c = T5 + T6;
      b = b + 1
    };
  bar (a, b, c)
}
```

Figure 6: Example program in GIMPLE form before optimization.

are never used again, conditional expressions with empty bodies, etc).

We are also developing a memory checker called *mudflap*. This pass instruments every pointer and array reference in the program with boundary checks. Mudflap is a combination of compile-time instrumentation and run-time library. The instrumented code contains calls to the run-time library that will be triggered when the program attempts one of several illegal operations, such as accessing an array out of bounds, freeing the same block of memory more than once, accessing unallocated memory, leaking memory, etc.

To illustrate some of the capabilities of the current implementation, we will show the effects of constant propagation and dead-code elimination with the full version of the running example we have been using (Figure 6)[1]. Figure 7 shows the effects of constant propagation on function `main()`. Notice that few constants have been propagated, mainly due to the presence of the calls to `foo()` and `bar()`.

Now, we will combine the effects of function inlining with constant propagation. Notice that now the constant propagator can discover more constants and is able to drastically reduce the code generated for `main`. Because the call to `foo` always returns `0`, the CCP pass can prove

[1]The code has been partially redacted to improve legibility.

```
main()
{
  a = foo ();
  b = 19;
  c = 5;
  if (a > 24)
    {
      T5 = 19 / a;
      T6 = a * 19;
      c = T5 + T6;
      b = 20
    };
  bar (a, b, c)
}
```

Figure 7: Function `main` after CCP and DCE (no inlining).

that the predicate for the `if` statement is always false, rendering the body of the `if` unreachable. When the control flow graph clean-up routines and dead-code elimination run, they reduce the function to the code shown in Figure 8.

5 Conclusions

The tree SSA project provides a new optimization framework to make it possible for GCC to implement high-level analyses and optimizations. Currently, the framework is in active development and some optimizations have already been implemented. The goals of this project include:

- Provide a basic set of data structures and functions for optimizers to query and manipulate the tree representation.
- Simplify and, in some cases, replace existing optimizations that work on the RTL representation but are not really suited for it. By simplifying the work for the RTL optimizers we aim to improve compile times and code quality.
- Implement new optimizations and analyses that are either difficult or impossible to implement in RTL.

By basing all the analyses and transformations on widely known published algorithms, we are also trying to improve our ability to maintain and add new features to GCC. Furthermore, the use of standard techniques will encourage external participation from groups in the compiler community that are not necessarily familiar with GCC.

We are also tracking the effects of the tree SSA optimizers with periodic runs of the SPEC benchmark suite (`http://www.spec.org/`). Daily results of these experiments can be found at `http://gcc.gnu.org/benchmarks/`. Readers interested in testing the current implementation or contributing to its development are invited to visit the tree SSA web page at `http://gcc.gnu.org/projects/tree-ssa/`. This page contains information for retrieving a copy of the development branch in CVS, status of the implementation and a list of "to-do"

```
main()
{
  printf ("a = %d, b = %d, c = %d\n", 0, 19, 5);
}
```

Figure 8: Function `main` after constant propagation and dead-code elimination (with inlining).

items.

References

[1] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, 1997.

[2] 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.

[3] 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.

[4] E. Stoltz, M. P. Gerlek, and M. Wolfe. Extended SSA with Factored Use-Def Chains to Support Optimization and Parallelism. In *Proc. Hawaii International Conference on Systems Sciences*, Maui, Hawaii, January 1994.

[5] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.