Tree SSA
A New Optimization Infrastructure for GCC

Diego Novillo
Red Hat Canada, Ltd.
`dnovillo@redhat.com`

Abstract

Tree SSA is a new optimization framework based on the Static Single Assignment (SSA) form that operates on GCC's tree representation. Tree SSA is designed to be both language and target independent and allow high-level analyses and transformations that are difficult or impossible to implement with RTL. One of the main goals of the project is to produce an analysis and optimization infrastructure based on proven algorithms and techniques available in the literature. In this paper we describe the design and implementation of the Tree SSA framework, provide preliminary results and discuss possible applications and future work.

1 Introduction

Currently, optimizing transformations in GCC operate on a single intermediate representation, namely RTL (Register Transfer Language). Parse trees generated by the front end are almost immediately converted into RTL and passed on to the optimizer (Figure 1).

Being a low-level representation, RTL works well for optimizations that are close to the target (e.g., register allocation, delay slot optimizations, peepholes, etc). However, many optimizing transformations need higher level information about the program that is difficult (or even impossible) to obtain from RTL (e.g., 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 suited for this, the end result is code that is excessively convoluted, hard to maintain and error prone.

In this paper we describe an optimization framework based on GENERIC and GIMPLE, two high-level intermediate representations (IR) derived from GCC parse trees [5]. Language-specific constructs are removed from the input parse trees to obtain GENERIC. In turn, GENERIC trees are broken down into a simpler three address representation called GIMPLE which is used for optimization.

Optimizing GIMPLE 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, and (c) it can be done in a largely language and target-independent way. The latter is an important feature for a compiler like GCC that targets many different architectures and languages.

We believe that modularizing the compiler and using well-known published algorithms will help developers maintain and improve GCC, and flatten the learning curve required for ex-

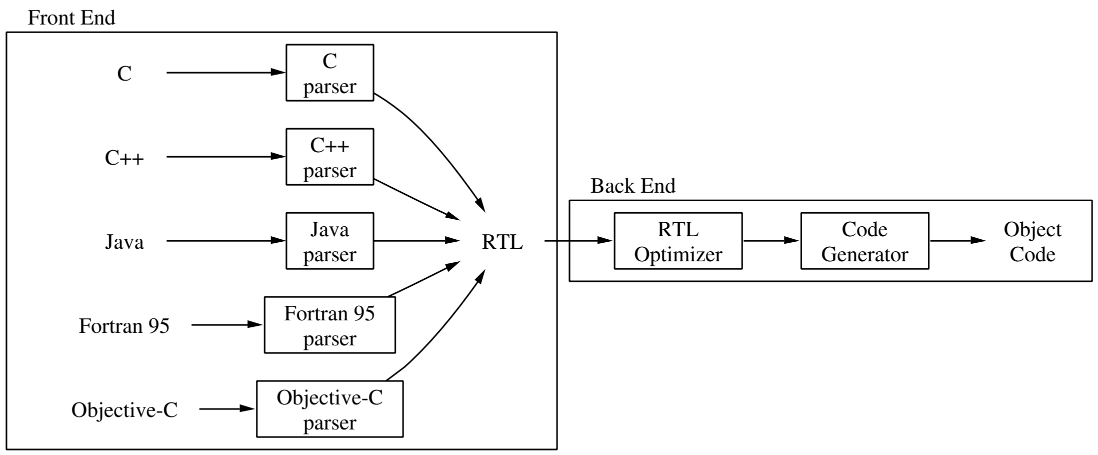

Figure 1: Existing compilation process in GCC.

ternal developers to contribute optimization passes. Furthermore, by reducing the amount of RTL code generated, we also expect to reduce compilation times and improve the quality of the final code.

2 Overview

There are three main components to the basic infrastructure: the gimplifier, the control flow graph (CFG) and the SSA module (Figure 2).

- The gimplifier is responsible for converting the input 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 Control Flow Graph (CFG) is a directed graph that models the execution of the program. Each node in the CFG, 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.).

- Static Single Assignment (SSA) is a relatively new representation that is becoming increasingly popular because it leads to efficient algorithmic implementations of data flow analyzers and optimizing transformations [3].

  The SSA module finds all the variables referenced and builds the SSA form for the function. It provides all the necessary functions and data structures to compute, among other things, aliasing, reaching definitions, and reached-uses information. It is also responsible for converting the function back to normal form right before calling the RTL expanders.

Figure 3 shows the proposed integration between GIMPLE and RTL optimizations as implemented in the `tree-ssa` branch. Notice that the interface between GENERIC and GIMPLE may involve some language-dependent transformations, but those issues are beyond the scope of this paper.

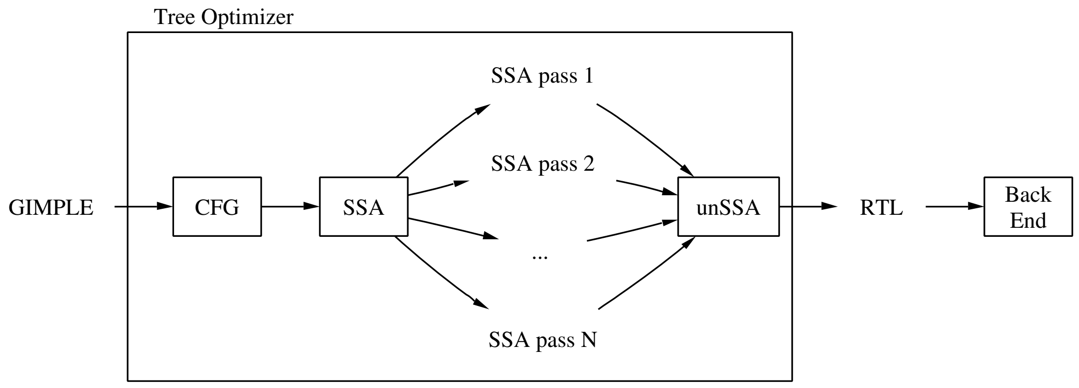

Figure 2: Overview of the tree optimization process.

3 GIMPLE Trees

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

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 GCC.

2. **Side effects**. Parse trees are allowed to have side effects. 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 must 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.

To overcome these limitations, we use 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. All the analyses and optimizations are done on the GIMPLE representation.

Figure 4 illustrates the differences between GENERIC (Figure 4(a)) and GIMPLE (Figure 4(b)). Notice how in the GIMPLE version individual expressions are simpler and more regular in structure. For instance, with the exception of function calls, 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 4(a) has been explicitly exposed by

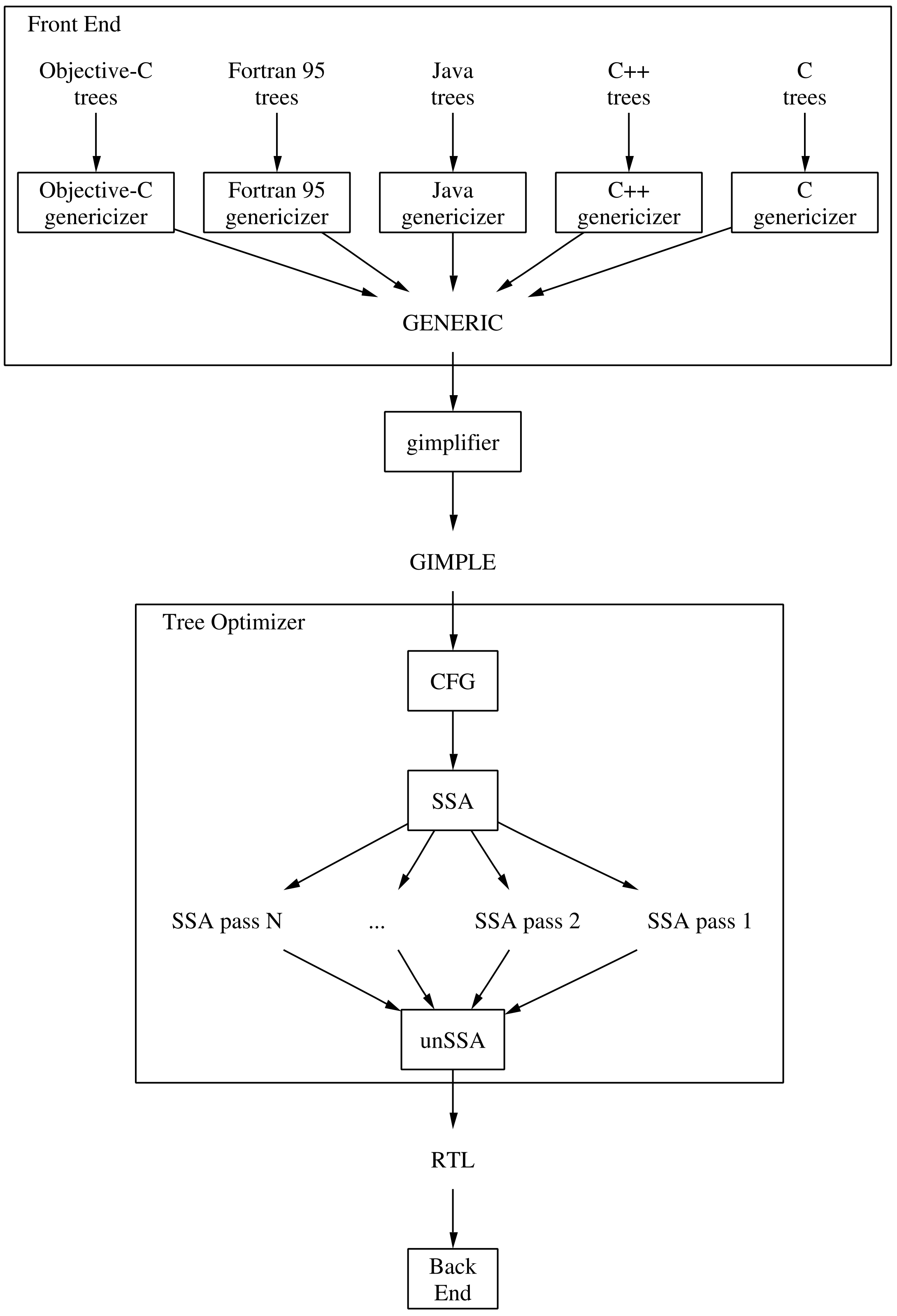

Figure 3: Proposed integration of GIMPLE and RTL optimizers.

```
a = foo ();
b = a + 10;
c = 5;
if (a > b + c)
  c = b++ / a + (b * a);
bar (a, b, c);
```

(a) GENERIC form.

```
a = foo ();
b = a + 10;
c = 5;
T1 = b + c;
if (a > T1)
  {
    T2 = b / a;
    T3 = b * a;
    c = T2 + T3;
    b = b + 1;
  }
bar (a, b, c);
```

(b) GIMPLE form.

Figure 4: A program in GENERIC and GIMPLE forms.

the conversion to GIMPLE form).

4 The Control Flow Graph

To take advantage of the existing flow graph code for RTL, the GIMPLE flow graph uses the same data structures for basic blocks and edges. This allows the GIMPLE CFG to use all the functions that operate on the flow graph independently of the underlying IR (e.g., dominance information, edge placement, reachability analysis). For the cases where IR information is necessary, we either replicate functionality (e.g., flow graph cleanup) or have introduced hooks (e.g., loop discovery).

The flow graph builder will also remove superfluous control expressions of the form `if (0)`, `if (1)` and `switch (CST)`. The edges leading to the unreachable arms of the conditionals are removed, which in turn causes the unreachable arms to be removed. These statements are also completely linearized by replacing the conditional with the clause that is always executed.

4.1 Statement manipulation

Although GIMPLE trees have a much simpler structure than GENERIC and the original parse trees, they still contain certain elements that are of no interest to a typical optimization pass. GIMPLE is a container-based data structure. As such, statements inside constructs like loops, conditionals and lexical scopes are contained in sub-trees. Within each lexical scope, individual statement nodes are chained together using compound expression (CE) nodes. For instance, the body of function `baz` in Figure 5 contains two statements, the lexical scope starting at line 5 and the `return` statement at line 13. In turn, the lexical scope at line 5 contains 3 statements (lines 8, 9 and 10). Notice how all the statements in each lexical scope are joined using CE nodes.

One way to traverse this function is to use the traditional call to `walk_tree` with a callback function to do the processing. However, this approach not only visits more nodes than necessary, but it also makes it difficult to distinguish a statement from an expression contained

```
1   baz ()
2   {
3     int i, j;
4
5     {
6       int k;
7
8       k = foo ();
9       i = k + 2;
10      j = i * k;
11    }
12
13    return j;
14  }
```

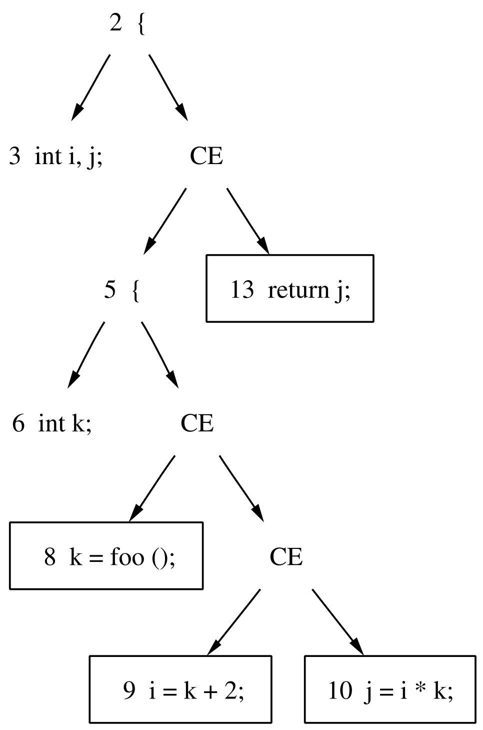

Figure 5: A GIMPLE program and its tree representation.

in a statement[1].

To traverse the statements of a function in GIMPLE, one must follow the compound expression nodes in the body of the function. We have implemented an iterator data structure, called *tree statement iterator* (TSI), to facilitate this process. Note that TSIs don't guarantee that every single statement will be visited. A traversal starting at line 5 in Figure 5 will only visit lines 5 and 13. It is up to the caller to detect when a lexical scope or control statement is being visited and recursively visit its body.

While TSIs are convenient for traversing lexical scopes, they are not suited for traversing statements inside basic blocks. Notice how function `baz()` contains a single basic block. A proper traversal should visit lines 8, 9, 10 and 13, which could be done using TSIs, but the caller would have to be responsible for handling lexical scopes and determining basic block boundaries. This is provided by *block statement iterators* (BSI). Thus, once the flow graph for the function has been built, traversing all the statements in the function can be done with the double nested loop:

```
FOR_EACH_BB (bb)
  for (i = bsi_start (bb); !bsi_end_p (i); bsi_next (&i))
    process_stmt (bsi_stmt (i));
```

BSIs can also be used for backward traversals as well as statement manipulation. Currently, statements can be removed, inserted inside blocks (before and after other statements) and inserted on edges.

[1]GENERIC and GIMPLE do not distinguish statements from expressions as is done in the C and C++ front ends.

```
 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) Same program in SSA form.

Figure 6: Static Single Assignment form.

5 Static Single Assignment form

The Static Single Assignment (SSA) form [3] 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 that variable. Naturally, actual programs are seldom in SSA form initially because variables tend to be assigned multiple times. The 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.

Figure 6 shows the program from Figure 4(b) and its corresponding SSA form (Figures 6(a) and 6(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` at line 10. To solve this ambiguity, SSA introduces a new artificial definition for `b` by means of a ϕ (phi) function. This new definition merges both assignments to create a new version for `b` ($\mathtt{b}_3$). The semantics of the ϕ function dictate that $\mathtt{b}_3$ will take the value from one of the function's arguments. The specific argument returned by the ϕ function is not known until runtime.

6 Real and virtual operands

The SSA form is not suited for handling non-scalar variable types. For instance, given an array `M[100][100]`, not only would the compiler need to keep track of 10,000 different version numbers for `M`, but it may also be impossible to determine whether two references `M[i][j]` and `M[k][l]` are the same variable or not. Structures, unions and aliased variables present similar problems.

One alternative to handling non-scalar types would be to simply ignore them. After all, if the operands are not converted into SSA form, they would not be considered for optimization. However, that would also mean that statements referencing nothing but non-scalars would appear dead to the optimizers. Also, situations like scalar variables aliased by a structure field would also be missed.

To address this problem, operands referencing non-scalar variables are considered references to the base object for that variable. For instance, references to `M[i][j]` and `M[k][l]` in the previous example would be considered references to `M`. Since these operands need to be treated separately by the optimizers, they are known as *virtual operands*, as opposed to the *real operands* for scalar variables. Therefore, every GIMPLE statement *S* contains four distinct sets of operands:

DEF(S). If *S* is an assignment statement, this is the variable at its left-hand side.

USES(S) is the set of all the variables used (or loaded) by *S*.

VDEFS(S) is the set of all the virtual variables defined (or stored) by *S*. VDEF operators represent non-killing definitions because they may or may not occur at run time. A VDEF operator is of the form `V = VDEF <V>`, which means that a new value for `V` is created from `V`'s old value.

VUSES(S) is the set of all the virtual variables used by *S*.

Virtual operands are also used to handle situations where the program is altering variables in ways that are difficult or impossible to determine statically. In these cases, the data flow framework needs to gather enough information to prevent the optimizers from missing a potential data dependency. In all these cases, virtual operands are used. Some of the more common situations include:

1. **Aliasing**. If two variables `a` and `b` may alias each other, then the compiler selects one of them to serve as the representative for all the aliased references. Every reference to either variable is then considered a virtual operand using the alias representative.

2. **Call clobbering**. Function calls may modify addressable local variables and globals in unknown ways. This is handled using a similar approach. Variables that may be call clobbered are considered alias of an artificial variable called `.global_var`. This variable is considered modified by function calls and by assignments to any of the variables associated with it.

3. **Inline assembly**. Much like function calls, inline assembly may modify local variables in ways that the optimizers do not understand. Variables listed in the *outputs* or *clobbers* list of GCC's `asm` statement, are considered VDEF operands.

The programs in Figures 7, 8 and 9 illustrate how virtual operands are used to handle non-scalar variables, aliasing and call clobbering. All the example functions have been renamed into SSA already. Notice how the VDEF operators link new SSA versions for a variable with its previous version. This creates def-def links that are used in passes like dead-code elimination to determine all the potentially live assignments.

```
double foo (int i, int j, int k, int l)
{
  double T1, T2, f;
  double M[100][100];

  /* References to an element of 'M' are
     considered references to the whole
     matrix.   */
  # M_2 = VDEF <M_1>
  M[i][j] = . . .

  /* VDEFs are non-killing definitions,
     that's why the new definition
     created for M_3 is linked to M_2 in
     the previous assignment.     */
  # M_3 = VDEF <M_2>
  M[k][l] = . . .

  # VUSE <M_3>
  T1_4 = M[i][j];

  # VUSE <M_3>
  T2_5 = M[k][l];
  f_6 = T1_4 + T2_5;

  return f_6;
}
```

Figure 7: Virtual operands for handling non-scalar variables.

7 Representing pointers

In GIMPLE there are no multi-level pointers. This is a very appealing property that allows the compiler to keep track of a pointer p and its dereference *p as two separate, but related, variables. The relation between p and *p is quite straightforward:

1. Every store to p implies a store operation to *p, because now p is pointing to a different memory location.
2. Every store or load of *p implies a load operation from p, because p is read to determine what memory location to use.

```
int foo (int i, int j, int *p)
{
  int a;

  if (i_1 > j_2)
    {
      /* Whenever 'p' changes, '*p' must
         also change.   */
      # (*p)_4 = VDEF <(*p)_3>
      p_5 = &a;
    }

  /* Since '*p' may alias 'a', instead
     of renaming the operand 'a', we
     create a virtual definition for its
     alias '*p'.    */
  # (*p)_7 = VDEF <(*p)_4>
  /* 'p' is needed to access '*p'.      */
  # VUSE <p_5>
  a = i_1 + j_2;

  # VUSE <(*p)_7>
  return *p;
}
```

Figure 8: Virtual operands for handling aliases.

8 Conversion into SSA form

Converting the program into SSA form consists of three main phases:

1. may-alias computation, which determines what variables are referenced in the function and whether they may be aliased or not,
2. insertion of ϕ nodes at basic blocks reached by more than one definition of the same variable, and,
3. statement renaming, which rewrites every operand and virtual operand using the appropriate SSA version numbers.

The following sections highlight the more important aspects of the conversion into SSA

```
float F;

float foo(float f)
{
  /* Since 'F' is call-clobbered,
     instead of renaming 'F' in the
     statement, we rename the virtual
     operand .GLOBAL_VAR. */
  # .GLOBAL_VAR_2 = VDEF <.GLOBAL_VAR_1>
  F = f_3 + 2;

  /* Function calls clobber the variable
     .GLOBAL_VAR which in turn indicates
     that 'F' is also clobbered.       */
  # .GLOBAL_VAR_3 = VDEF <.GLOBAL_VAR_2>
  bar ();

  /* Uses of 'F' are converted to
     virtual uses of .GLOBAL_VAR.   In
     this statement we are using the
     value of 'F' potentially
     modified by the call to bar().      */
  # VUSE <.GLOBAL_VAR_3>
  return F;
}
```

Figure 9: Virtual operands for handling call clobbering.

form. A more detailed description of the process can be found in the literature [3, 1, 6].

8.1 Computing may-alias information

This pass collects all the variables referenced in the function and determines may-alias sets for each one. Currently, alias information is type-based. A points-to analyzer is implemented, but it is not fully functional yet.

8.2 Inserting ϕ nodes

This pass inserts ϕ nodes at the dominance frontier of blocks with live variable definitions. The algorithm implements the semi and fully pruned forms suggested by Briggs et. al. [1] to reduce the number of ϕ nodes in the program. The basic idea is that if a variable is not live after being defined in block b, then it is not necessary to insert a ϕ node at the dominance frontier of b.

Since computing global live-in information is more expensive than local live-in, this pass uses a heuristic based on the total number of ϕ arguments. If this is is above a certain threshold[2], the compiler builds a fully pruned form.

8.3 Rewriting statements and dominator-based optimizations

The renaming process is done using a depth-first traversal of the flow graph's dominator tree [3]. During this traversal it is possible to apply very simplistic transformations that take advantage of the order in which basic blocks are visited [6].

These transformations, also known as *dominator-based optimizations*, include constant propagation, redundancy elimination, copy propagation and propagation of predicate expressions. These optimizations are only supposed to do simple cleanup work that catches most of the simple cases. The key property is that they must work fast because they are piggybacked on top of the renaming process (which is linear in the number of statements).

1. Constant propagation. When a constant assignment of the form $a_i = C$ is found, it is stored in a hash table. Successive occurrences of a_i are replaced with C. No folding nor control flow simplification is done, only constant replacements. Copy assignments are similarly optimized.

2. Redundancy elimination uses a similar idea. When an assignment of the form $a_i = b_j \oplus c_k$ is found, the expression $b_j \oplus c_k$

[2]Currently 32.

is stored into a hash table. Successive occurrences of $b_j \oplus c_k$, *within the same sub-tree*, are replaced with a_i. Notice that this transformation is valid only when replacing redundant expressions dominated by the original assignment, otherwise it would be possible to insert a_i in a control flow path where it is never evaluated.

3. Propagation of predicate expressions. When a conditional statement of the form `if (`$a_i == C$`)` is found, the assignment $a_i = C$ is inserted into the hash table for constants and copies when processing the "then" clause of the conditional. This will cause the constant/copy propagator to replace a_i with C in that sub-tree.

8.4 Conversion back to normal form

Once all the SSA optimizations have been applied to the function, all the SSA version numbers and ϕ nodes must be removed to return the code to its original form. This process consists mainly in converting all ϕ nodes into copy operations. Some of the more important aspects of this pass is avoiding superfluous copy operations. We implement the standard conversion into normal form described in the literature [1, 6].

9 Implementation status

Currently, the basic framework is almost finished. Two front ends (C and C++) have been fully converted to emit GIMPLE trees and the regression test suite presents similar results to those of mainline GCC. Readers interested in testing the current implementation and/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" items.

In terms of performance, the branch still lags behind mainline. This is hardly surprising as we have mostly worked on correctness issues. Performance is going to be the focus of the next phase of development. We have been tracking performance using SPEC95 and SPEC2000. Daily results of these experiments can be found at `http://gcc.gnu.org/benchmarks/`.

In addition to the optimizations performed while renaming into SSA form and the flow graph restructuring, there are four optimization passes implemented.

1. Sparse Conditional Constant Propagation (CCP) [7] is an efficient formulation of the constant propagation problem that is also capable of finding constant conditionals and unreachable code. This optimization is currently enabled by default at `-O1` and above.

2. Partial Redundancy Elimination (PRE) [2] 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) [3] removes all statements in the program that have no effect on its output (assignments to variables that are never used again, conditional expressions with empty bodies, etc). This optimization is currently enabled by default at `-O1` and above.

4. Copy Propagation (CP) is the same optimization applied while converting the program into SSA form, but implemented as a separate pass.

We are also implementing a memory checker, called *mudflap*, that instruments every pointer and array reference in the program with boundary checks [4]. It 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.

Mudflap is not yet integrated into the SSA framework, so no static analyses are done to prevent inserting superfluous instrumentation. Optimization of mudflap instrumentation is currently underway.

10 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.

Acknowledgments

I would like to thank Red Hat for funding the Tree SSA project and to all the developers who have contributed to it. In particular, I would like to thank the regular contributors to the project: Jeff Law and Andrew MacLeod for their work on the base infrastructure and optimizers; Jason Merrill for his work on GENERIC and GIMPLE; Frank Eigler for his work on Mudflap; Sebastian Pop for the original expression simplifier, tree unparser, and tree browser; Daniel Berlin for his work on points-to alias analysis and PRE; Steven Bosscher and the G95 team for their work on integrating G95 with GIMPLE; and Andreas Jaeger, Phil Edwards, and Andrew Pinski for testing the branch on a regular basis.

References

[1] P. Briggs, K. D. Cooper, T. J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software—Practice and Experience*, 28(8):859–881, 1998.

[2] 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.

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

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

[6] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

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