Tree SSA – A New Optimization Framework for GCC

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

GCC Developers' Summit
Ottawa, Canada
May 2003

Goals of the Project

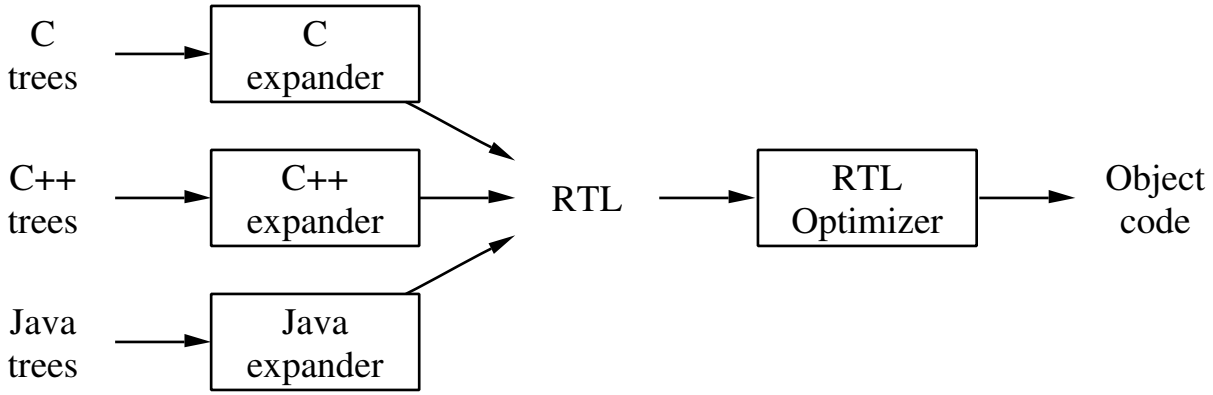
- Technical

  1. Internal infrastructure overhaul.
  2. Add new optimization features: vectorization.
  3. Add new analysis features: mudflap.

- Non-technical

  1. Improve maintainability.
  2. Improve our ability to add new features to the optimizer.
  3. Allow external groups to get interested in GCC.

RTL based optimizers



- RTL is not suited for high-level transformations.
- Too many target features have crept in.
- Lost original data type information and control structures.
- Addressing modes have replaced variable references.

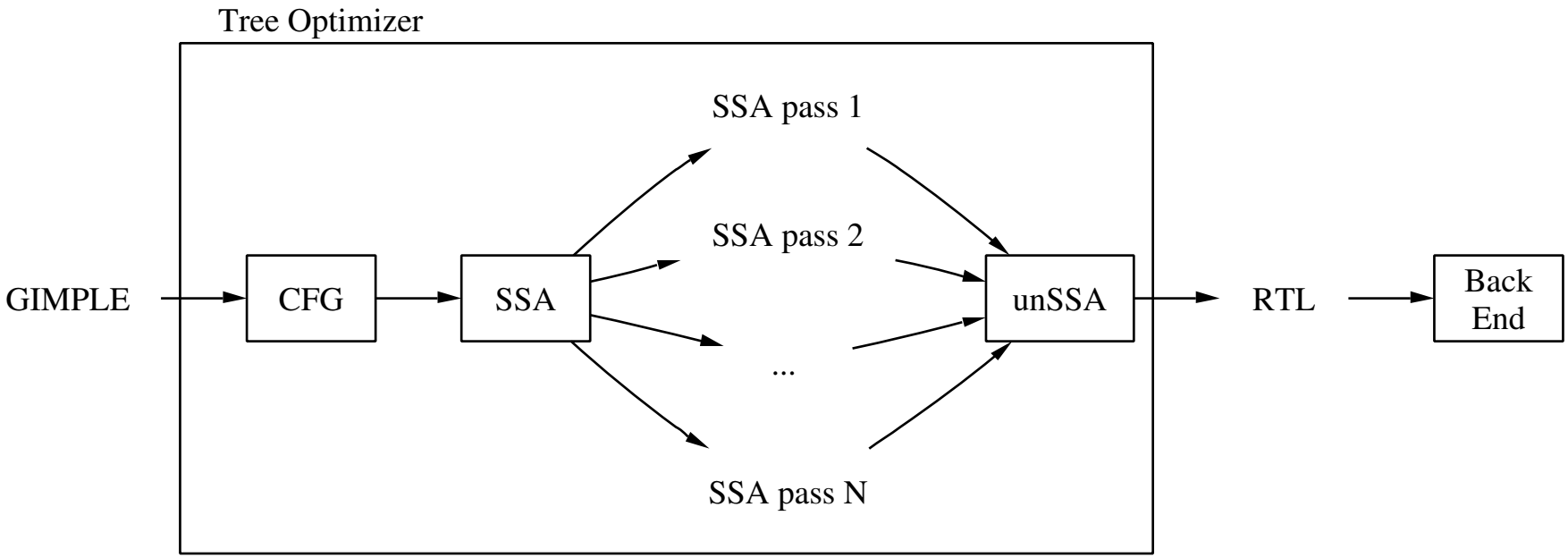
Tree based optimizers

- GCC trees contain complete control, data and type information for the original program.

- Suited for transformations closer the source.
  - Control flow restructuring.
  - Scalar cleanups.
  - Data dependency analysis on arrays.
  - Instrumentation.

- Problems.
  - Each front end generates its own "flavor" of trees.
  - Trees are complex to analyze. They can be freely combined and carry a lot of semantic information and side-effects.

Tree SSA Overview



- GIMPLE trees are language/target independent.
- Full type information is preserved.

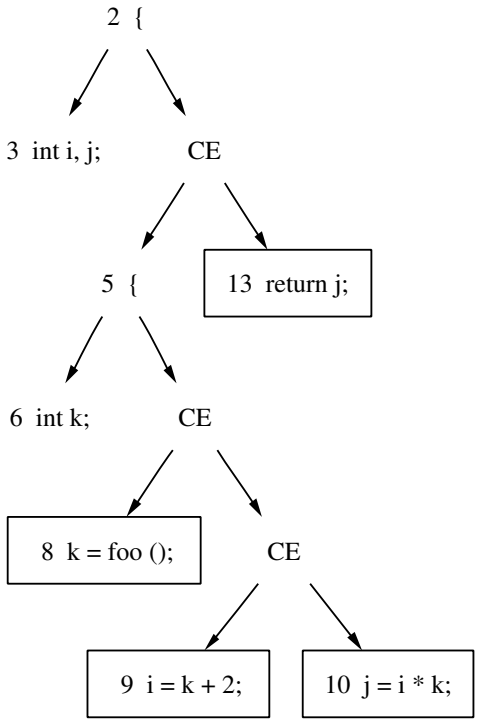
GIMPLE trees

```
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 = 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);
```

Statement manipulation

```
 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  }
```



- Two kind of iterators: block (BSI) and tree (TSI).

Control Flow Graph

① Share same flowgraph data structures and code from RTL flowgraph.

② IL-specific information is replicated or use langhooks.

③ Basic cleanup passes: linearization, unreachable code elimination.

SSA form

- A program is in SSA form iff every USE of a variable is reached by no more than **one** DEF.

```
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);
```

```
a_1 = foo ();
b_1 = a_1 + 10;
c_1 = 5;
T1_1 = b_1 + c_1;
if (a_1 > T1_1)
  {
    T2_1 = b_1 / a_1;
    T3_1 = b_1 * a_1;
    c_2 = T2_1 + T3_1;
    b_2 = b_1 + 1;
  }
b_3 = φ(b_1, b_2);
c_3 = φ(c_1, c_2);
bar (a_1, b_3, c_3);
```

SSA form

Most programs are not in SSA form and need to be converted

① Every time a variable is defined, it receives a new version number.

② Variable uses get the version number of their immediately reaching definition.

③ Ambiguities (i.e., more than one immediately reaching definition) are solved by inserting artificial variables called ϕ-nodes (or ϕ-terms).

ϕ-nodes are functions with N arguments. One argument for each incoming edge.

Handling non-scalar variables and aliasing

```
foo (i, j, k, l)
{
  # M_2 = VDEF <M_1>
  M[i][j] = ...

  # M_3 = VDEF <M_2>
  M[k][l] = ...

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

  f_6 = T1_4 + T2_5;

  return f_6;
}
```

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

  if (i_1 > j_2)
    p_5 = &a;

  # MT.1_7 = VDEF <MT.1_4>
  a = i_1 + j_2;

  # VUSE MT.1_7
  return *p;
}
```

Conversion into SSA form

1. May-alias computation.

2. Insertion of ϕ nodes.

   - Minimal
   - Semi-pruned
   - Pruned

3. Statement renaming. Dominator-based optimizations:

   - constant propagation
   - redundancy elimination
   - propagation of predicate expressions.

Conversion out of SSA form

1. Remove ϕ nodes by converting them into copies.

2. Coalesce as many copies as possible.

3. Deal with overlapping live ranges of different SSA names for the same variable.

4. Assign SSA names to real variables.

Current Status

- C and C++ front ends emit GIMPLE trees.
- SSA based constant propagation and dead code elimination working.
- Copy propagation, partial redundancy elimination, global value numbering and value range propagation being implemented.
- Plan to merge infrastructure for GCC 3.5, provided we keep making the same progress.
- Performance w.r.t. mainline still lagging, but making steady progress.

Implementation Details

- Main entry points.

  `c-decl.c` calls the gimplification and optimization passes before RTL expansion.
  `gimplify.c` converts the function into GIMPLE form.
  `tree-cfg.c` builds the CFG.
  `tree-dfa.c` finds all variable references in the function.
  `tree-ssa.c` builds the SSA web.
  `tree-simple.c` validates statements and expressions in GIMPLE form.
  `tree-pretty-print.c` unparses GENERIC trees.

TODO List

- Optimizations.
  - Value Numbering (VN), Value Range Propagation (VRP).
  - Mudflap-specific optimizations.
  - Loop transformations
    loop canonicalization.
    loop unswitching.
    loop unrolling.
  - Vectorization: Super-word level parallelism (SLP).

- Performance evaluation: profile, remove superfluous RTL passes, improve tree→RTL conversion.

Conclusions

- Tree SSA provides a new optimization framework to implement high-level analyses and optimizations in GCC.

- Goals:

  1. Provide a basic data and control flow API for optimizers.
  2. Simplify and/or replace RTL optimizations. Improve compile times and code quality.
  3. Implement new optimizations and analyses that are either difficult or impossible to implement in RTL.

- Currently implemented in the C and C++ front ends.

- Code lives in the FSF branch `tree-ssa-20020619-branch`.

- Project page `http://gcc.gnu.org/projects/tree-ssa/`