Tree SSA – A New Optimization Framework for GCC

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

NordU/USENIX 2003
Västerås, Sweden
February 2003

Goals of the Project

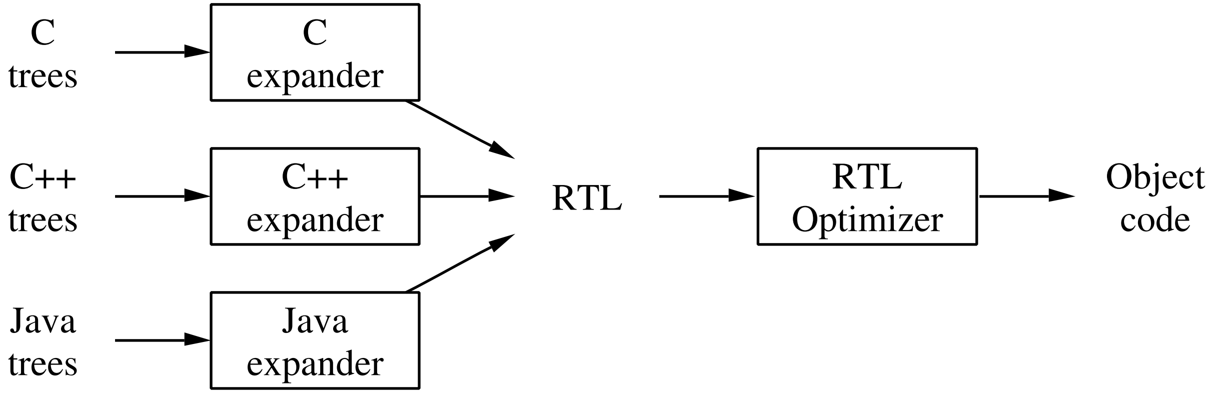
- Technical

  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

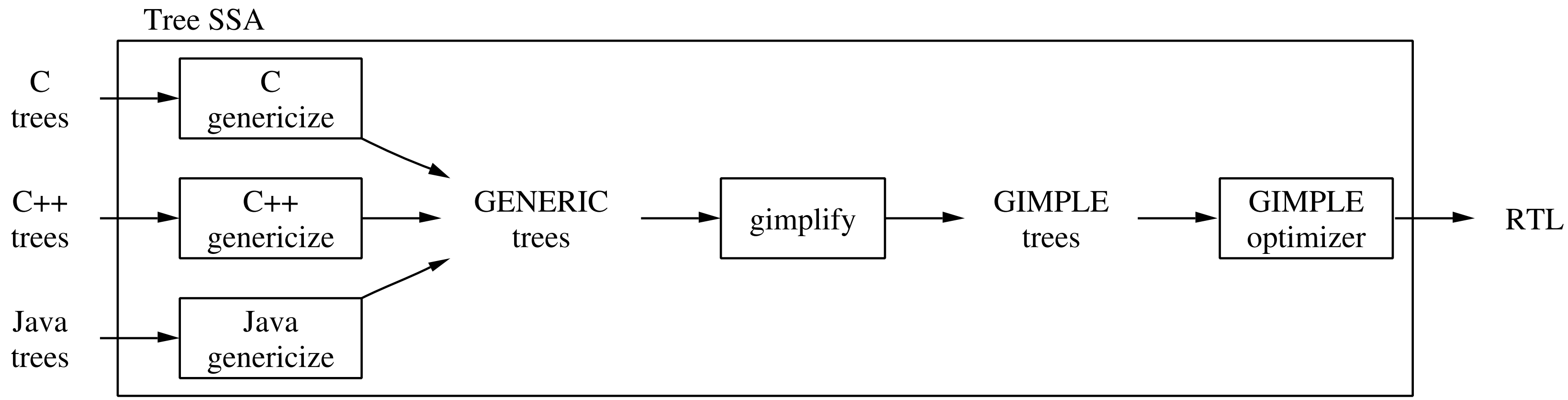


- RTL is not suited for high-level transformations.
- Too many target features have crept in.
- Lost original data type information and control structures.
- Addressing modes have replaced variable references.

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

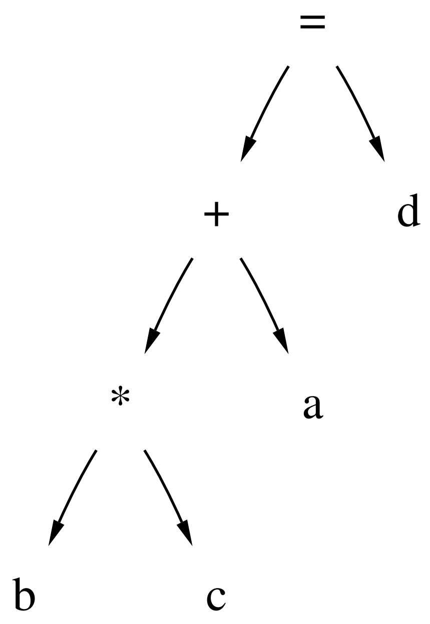


- GIMPLE trees can only be combined in a limited number of ways.
- They have no implicit side-effects.
- Full type information is preserved.
- GIMPLE trees are language/target independent.

GIMPLE trees

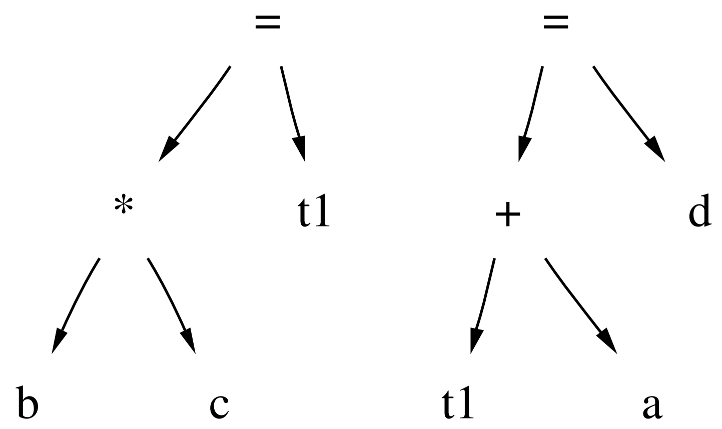
```
d = a + b * c;
```

Original tree



GIMPLE tree

```
t1 = b * c;
d = a + t1;
```

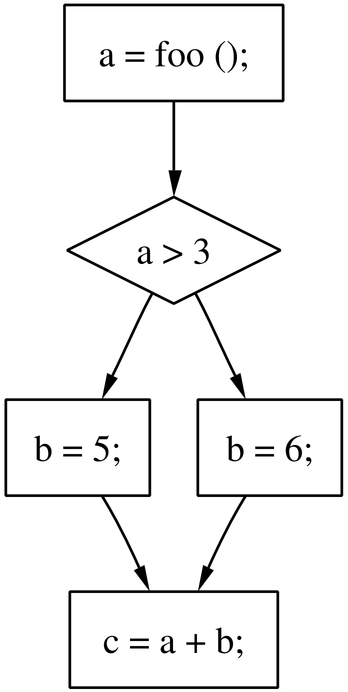


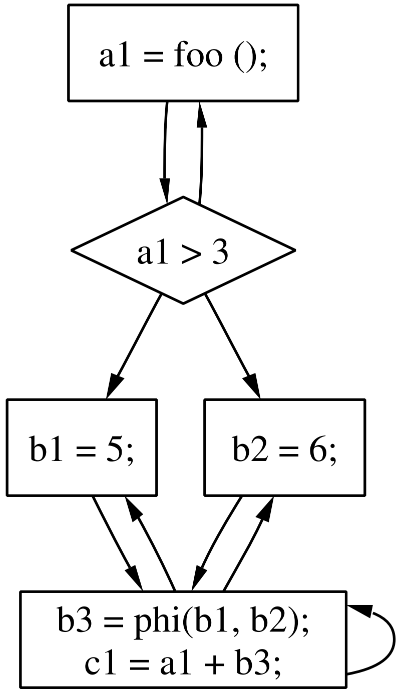
SSA form – 1

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

- Compiler modifies flow graph to model SSA property.

```
a = foo ();
if (a > 3)
  b = 5;
else
  b = 6;
c = a + b;
```





SSA form – 2

① Build CFG.

② Find variable references.

③ Build SSA web.

- Why is SSA so interesting?
  1. It's a sparse data structure that describes all the DEF/USE points in the program.
  2. Some data flow problems are trivial to solve using SSA:
     – Is this definition dead?
     – Is this use reached by any definition?
  3. Several modern transformations are based on SSA.

SSA form – 3

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.

Applications – Optimization

Original

$a_1 = 10;$
$b_1 = 3;$
$c_1 = a_1 + b_1;$
if $(c_1 < V_1)$
  $a_2 = a_1 + 3;$
else
  $a_3 = b_1 + 10;$
$a_4 = \phi(a_2, a_3)$
$c_2 = a_4 - b_1;$
printf ("%d\n", c_2);

Constant Propagation

$a_1 = 10;$
$b_1 = 3;$
$c_1 = 13;$
if $(13 < V_1)$
  $a_2 = 13;$
else
  $a_3 = 13;$
$a_4 = \phi(a_2, a_3)$
$c_2 = 10;$
printf ("%d\n", 10);

Dead Code Elimination

printf ("%d\n", 10);

Applications – Mudflap

- Instruments programs in GIMPLE form to detect memory access errors.
  - Pointer/array dereferences.
  - Addressed static/auto object lifetimes.

- Generates calls into runtime when errors are detected.

- Uses heuristics to work with uninstrumented code.

- Runtime (`libmudflap`)
  - Tracks heap us and provides efficient checked versions of str* and mem* function.
  - Provides efficient checked versions of str* and mem* functions.
  - Maintains live object database with names, bounds and statistics.

Applications – Mudflap

Original

```
{
char a;

int *b = (int *) & a;
b++;




*b = 5;

}
```

Instrumented

```
{
char a;
mf_register (& a, sizeof(char), "file:3 a");
int *b = (int *) & a;
b++;
* ({int *ptr = b;
  if (INLINE_LOOKUP_CACHE_MISS (ptr, sizeof(int)))
    mf_check (ptr, sizeof(int));
  ptr;}) = 5;
mf_unregister (& a, sizeof(char));
}
```

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.
- Several compile time improvements over the last few weeks (20% faster bootstraps since Jan22).
- Plan to merge infrastructure for GCC 3.5, provided we keep making the same progress.
- In some (still rare) cases, tree SSA transformations simplify the program enough to allow RTL optimizers to produce optimal code.

TODO List

- Optimizations.
  - Copy propagation (CP), 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 and remove superfluous RTL passes.

- Explicit parallellism
  - **GOMP**. New project recently started to implement OpenMP in GCC.
  - `http://savannah.nongnu.org/projects/gomp/`

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/`