Tree SSA
Design and Implementation

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

GCC & GNU Toolchain Developers' Summit
June 2004 – Ottawa, Canada

Project History - 1

Late 2000	Project starts.
Mar 2001	CFG/Factored UD chains on C trees.
Jul 2001	Added to `ast-optimizer-branch`.
Jan 2002	Pretty printing and SIMPLE for C.
May 2002	SSA-PRE.
Jun 2002	Move to `tree-ssa-20020619-branch`.
	SIMPLE for C++.

Project History - 2

Jul	2002	SSA-CCP.
		Flow insensitive points-to analysis.
Aug	2002	Mudflap and SSA-DCE.
Oct	2002	GIMPLE and GENERIC.
Nov	2002	Tree browser.
Jan	2003	Replace FUD chains with rewriting SSA form.

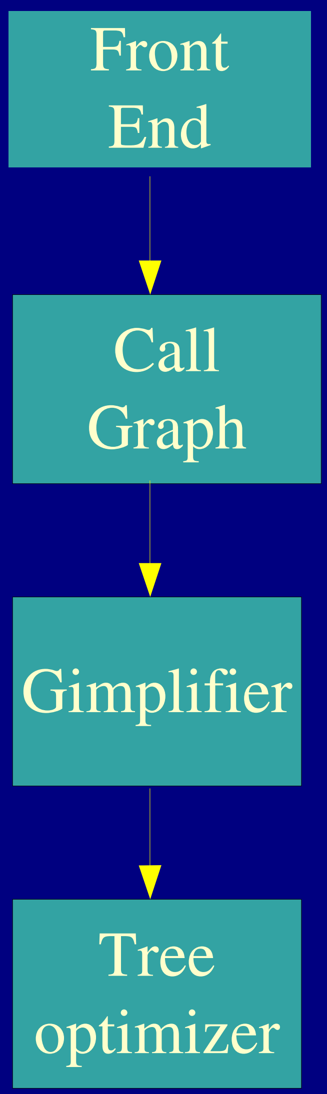
Project History - 3

Feb	2003	Statement iterators.
Apr	2003	Out of SSA pass.
Jun	2003	Dominator-based optimizations.
		GIMPLE for Java.
Jul	2003	Fortran 95 front end.
Sep	2003	EH lowering.
Nov	2003	Memory management for SSA names and PHI nodes.

Project History - 4

Nov 2003	Scalar Replacement of Aggregates.
Dec 2003	Statement operands API.
	Pass manager.
Jan 2004	Complex numbers lowering.
Feb 2004	Flow-sensitive and escape analysis, PHI optimization, forward propagation, function unnesting, tree profiling, DSE, NRV.

Compile Process



Generates a whole function representation.
Emits GENERIC and/or provides GIMPLE hook.

Makes inlining decisions.
Manages optimization ordering.

GIMPLE: a 3-address representation.
Lowers control structures to conditional jumps.

Renames GIMPLE into SSA.
SSA form includes aliasing and memory objects.
Pass manager sequences all analyses/optimizations.

Statement Operands - 1

- **Real operands** are for non-aliased scalars

  ```
  int x, y, z;
  x = y + z;
  ```

  Whole object reference

- **Virtual operands** are for aliased or aggregates

  ```
  int a[10], *p;
  *p = a[2] + 5;
  ```

  Partial, global or potential references.

Statement Operands - 2

- Real operands are part of the statement.

```
int x, y
x_5 = y_3 + 2
```

- Virtual operands are not.

```
int x[10], y[10]
# x_5 = VDEF <x_4>
# VUSE <y_3>
x[0] = y[0] + 2
```

Statement Operands - 3

```
int x, y
y_2 = 3      <-- DEAD
y_3 = 10
x_5 = y_3 + 2
```

```
int a, b, c, *y
y_2 = φ(&a,&b)
# a_6 = VDEF <a_1>
# b_7 = VDEF <b_3>
*y_2 = 3
# b_8 = VDEF <b_7>
b = 10       <-- NOT DEAD
# VUSE <a_6> <b_8>
c_5 = *y_2 + 2
```

Alias Analysis - 1

- GIMPLE only has single level pointers.
- Pointer dereferences represented by artificial symbols ⇒ ***memory tags*** (MT).
- If `p` points-to `x` ⇒ `p`'s tag is aliased with `x`.

```
# MT_2 = VDEF <MT_1>
*p_3 = ...
```

- Since `MT` is aliased with `x`:

```
# x_2 = VDEF <x_1>
*p_3 = ...
```

Alias Analysis - 2

- Type Memory Tags (TMT)
  - Used in type-based and flow-insensitive points-to analyses.
  - Tags are associated with symbols.
- Name Memory Tags (NMT)
  - Used in flow-sensitive points-to analysis.
  - Tags are associated with SSA names.
- Compiler tries to use name tags first.

Implementing SSA passes - 1

1. Add entry in `struct tree_opt_pass`
2. Declare it in `tree-pass.h`
3. Sequence it in `init_tree_optimization_passes`

- Access CFG with `FOR_EACH_BB`
- Use `block_stmt_iterator` to access statements
- Use `get_stmt_operands` and `{USE, DEF, VUSE, VDEF}_OPS` to access operands

Implementing SSA passes - 2

```
basic_block bb;
block_stmt_iterator si;

FOR_EACH_BB (bb)
  for (si = bsi_start (bb);
       !bsi_end_p (si);
       bsi_next (&si))
    {
      tree stmt = bsi_stmt (si);
      print_generic_stmt (stderr, stmt, 0);
    }
```

Implementation Status

- Infrastructure
  - Pass manager
  - CFG, statement and operand iteration/manipulation
  - SSA renaming and verification
  - Alias analysis built in the representation
  - Pointer and array bound checking (*mudflap*)
- Optimizations
  - Most traditional scalar passes: DCE, CCP, DSE, SRA, tail call, etc.

Future Work - 1

- Short term
  - Split up DOM
  - GVN PRE
  - Range propagation
  - Must-def for aggregates and globals
  - Not go out of SSA form for some transformations
  - Make C / C++ front ends emit GENERIC
- Medium term
  - Stabilization and speedup (Bugzilla)
  - Make RTL expanders work directly on SSA form

Future Work - 2

- LNO
- OpenMP
- Code factoring/hoisting for size
- Various type-based optimizations
  - Devirtualization
  - Redundant type checking elimination
  - Escape analysis for Java