# The GNU Compiler Collection

Diego Novillo
dnovillo@redhat.com

Gelato Federation Meeting
Porto Alegre, Rio Grande do Sul, Brazil
October 3, 2005

# Introduction

- GCC is a popular compiler, freely available and with an open development model.
- However
  - Moderately large code base (2.1 MLOC) and aging (~15 years).
  - Optimization framework based on a single IL (RTL).
  - Monolithic middle-end difficult to maintain and extend.
- Recent architectural changes are making it more attractive for new development.
  - New Intermediate Representations (GENERIC and GIMPLE).
  - New SSA-based global optimization framework.
  - New API for implementing new passes.

# GCC strengths

- One of the most popular compilers.
  - Very wide user base ⇒ lots of test cases.
  - Standard compiler for Linux.
  - Virtually all open/free source projects use it.
- Supports a wide variety of languages: C, C++, Java, Fortran, Ada, ObjC, ObjC++.
- Ported from deeply embedded to mainframes.
- Active and numerous development team.
- Free Software and open development process.

# GCC Development Model - 1

- Three main stages
  - Stage 1 - Big disruptive changes.
  - Stage 2 - Stabilization, minor features.
  - Stage 3 - Bug fixes only (driven by bugzilla, mostly).
- At the end of stage 3, release branch is cut and stage 1 for next version begins.
- Major development that spans multiple releases is done in branches.
- Anyone with CVS access may create a development branch.
- Vendors create own branches from FSF release branches.

# GCC Development Model - 2

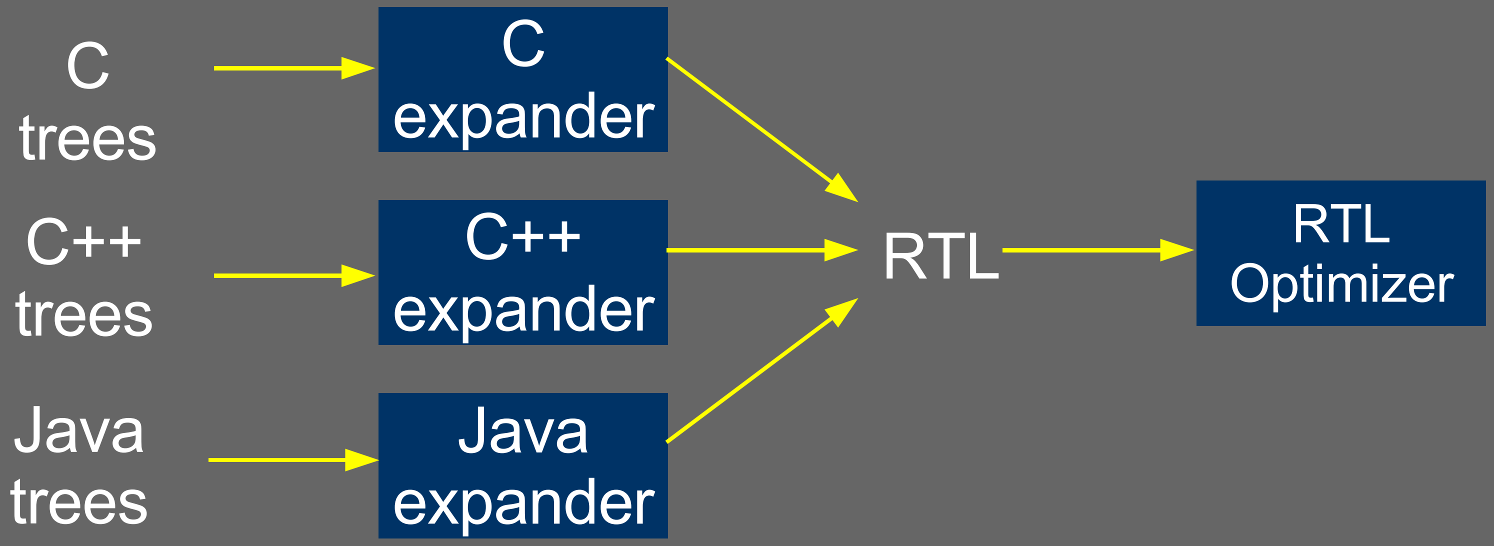
- All contributors must sign FSF copyright release.
  - Even if only working on branches.
- Three levels of access
  - Snapshots (weekly).
  - Anonymous CVS.
  - Read/write CVS.
- Major work on branches encouraged
  - Design/implementation discussion on public lists.
  - Frequent merges from mainline to avoid code drift.
  - Final contribution into mainline only at stage 1 and approved by maintainers.
- Having a thick skin is a definite plus.

# Problem 1 - Modularity

- New ports: ***straightforward***
  - Mostly driven by big embedded demand during 90s.
  - Target description language flexible and well documented.
- Low-level optimizations: ***hard***
  - Too many target dependencies (some are to be expected).
  - Little infrastructure support (no CFG until ~1999).
- New languages: ***very hard***
  - Front ends emit RTL almost directly.
  - No clear separation between FE and BE.
- High-level optimizations: ***sigh***
  - RTL is the only IL available.
  - No infrastructure to manipulate/analyse high-level constructs.

# Problem 2 – Lack of abstraction

- Single IL used for all optimization
  - RTL not suited for high-level analyses/transformations.
  - Original data type information mostly lost
  - Addressing modes replace variable references



# Problem 3 – Too much abstraction

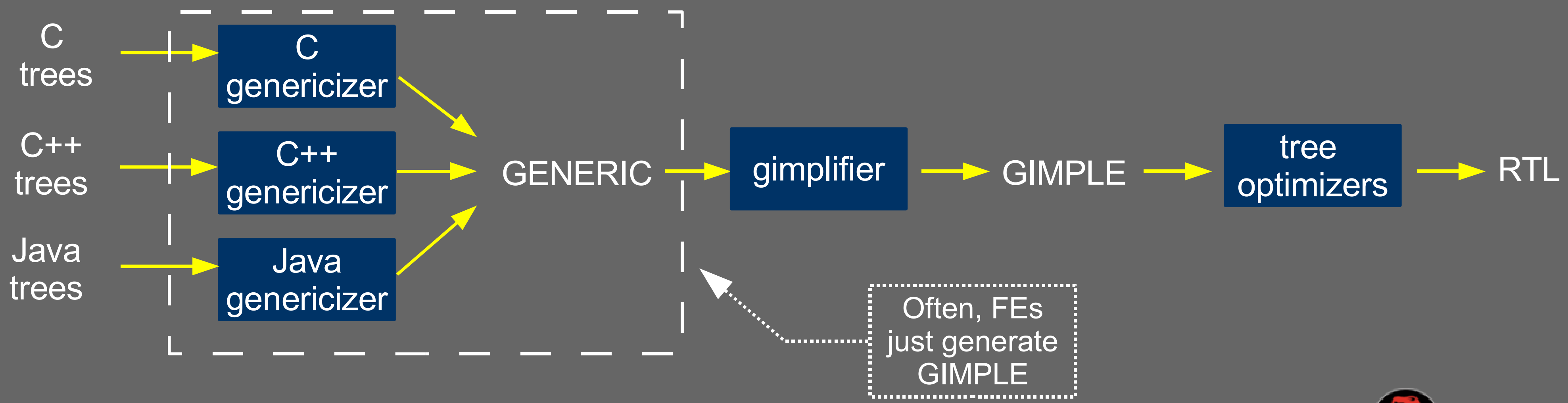
- Parse trees contain complete control/data/type information.
- In principle, well suited for transformations closer to the source
  - Scalar cleanups.
  - Instrumentation.
  - Loop transformations.
- However
  - No common representation across all front ends.
  - Side effects are allowed.
  - Structurally complex.

# Tree SSA

- Project started late 2000 as weekend hobby.
- Goal: SSA framework for high-level optimization.
- Approach: Evolution, not revolution → immediate integration.
- Features
  - Clear separation between FE and BE.
  - FEs generate common high-level IL that is both language and target independent.
  - Gradual lowering of IL.
  - Common API for CFG, statements, operands, aliasing.
  - Optimization framework: dom-tree walker, generic propagator, use-def chain walker, loop discovery, etc.
  - 30+ passes implemented so far.

# GENERIC and GIMPLE - 1

- GENERIC is a common representation shared by all front ends
  - Parsers may build their own representation for convenience.
  - Once parsing is complete, they emit GENERIC.
- GIMPLE is a simplified version of GENERIC.
  - 3-address representation.
  - Restricted grammar to facilitate the job of optimizers.



# GENERIC and GIMPLE - 2

## GENERIC

```
if (foo (a + b, c))
  c = b++ / a
endif
return c
```

## High GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0)
  t3 = b
  b = b + 1
  c = t3 / a
endif
return c
```

## Low GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0) <L1,L2>
L1:
t3 = b
b = b + 1
c = t3 / a
goto L3
L2:
L3:
return c
```

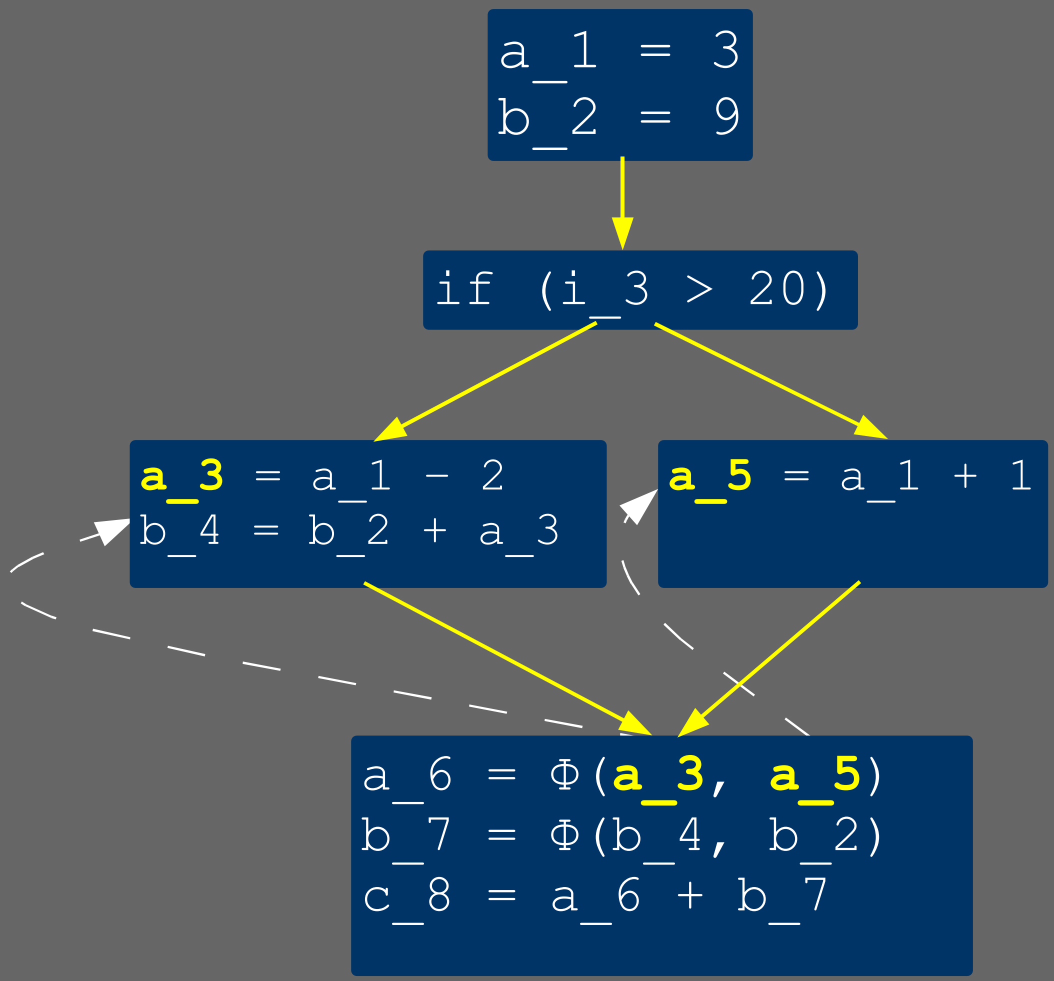
# Properties of GIMPLE form

- No hidden/implicit side-effects.
- Simplified control flow
  - Loops represented with `if/goto`.
  - Lexical scopes removed (low-GIMPLE).
- Locals of scalar types are treated as “registers”.
- Globals, aliased variables and non-scalar types treated as “memory”.
- At most one memory load/store operation per statement.
  - Memory loads only on RHS of assignments.
  - Stores only on LHS of assignments.
- Can be incrementally lowered (2 levels currently).

# SSA form - 1

Static Single Assignment (SSA)

- Versioning representation to expose data flow explicitly.
- Assignments generate new versions of symbols.
- Convergence of multiple versions generates new one (Φ functions).
- Two kinds of SSA forms, one for real another for virtual operands.



# SSA Form - 2

- Rewriting (or standard) SSA form
  - Used for real operands.
  - Different names for the same symbol are *distinct objects*.
  - Optimizations may produce overlapping live ranges (OLR).

```
x_3 = y_2
if (x_2 > 4)
  z_5 = x_3 - 1
```

  - Currently, program is taken out of SSA form for RTL generation (new symbols are created to fix OLR).
- Factored Use-Def Chains (FUD Chains)
  - Used for virtual operands.
  - All names refer to the *same object*.
  - Optimizers may not produce OLR for virtual operands.

# Implementation Status

- Infrastructure
  - Pass manager.
  - CFG, statement and operand iteration/manipulation.
  - SSA renaming and verification.
  - Alias analysis built into the representation.
  - Pointer and array bound checking (*mudflap*).
  - Generic value propagation support.
- Optimizations
  - Most traditional scalar passes: DCE, CCP, DSE, SRA, tail call, etc.
  - Some loop optimizations (loop invariant motion, loop unswitching, if-conversion, loop vectorization).

# Future Work - 1

- Short term
  - Remove dominator-based optimizations.
  - Stabilization and speedup (Bugzilla).
  - Alias analysis improvements.
    - Reduce IR size, alias queries, improve escape/clobbering analysis.
  - OpenMP (`gomp-20050608-branch`).

- Medium term
  - Unify Tree and RTL alias analysis.
  - Documentation.
  - Tie into fledgling IPA framework.
  - More loop optimizers (LNO branch).

# Future Work - 2

- Long term
  - Memory hierarchy optimizations.
  - Reimplement register allocation.
  - Code factoring/hoisting for size.
  - Slim down IR data structures.
  - Various type-based optimizations
    - Devirtualization.
    - Redundant type checking elimination.
    - Escape analysis for Java.
  - Analysis/optimization of OpenMP programs
    - Static analysis of synchronization constructs.
    - Cross thread optimization.

# Questions?