Using GCC as a Research Compiler

Diego Novillo
dnovillo@redhat.com

Computer Engineering Seminar Series
North Carolina State University – October 25, 2004

Introduction

- GCC is a popular compiler, freely available and with an open development model.
- However
  - Code base large (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 *"dragging GCC kicking and screaming into the 90s"*.
  - 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.

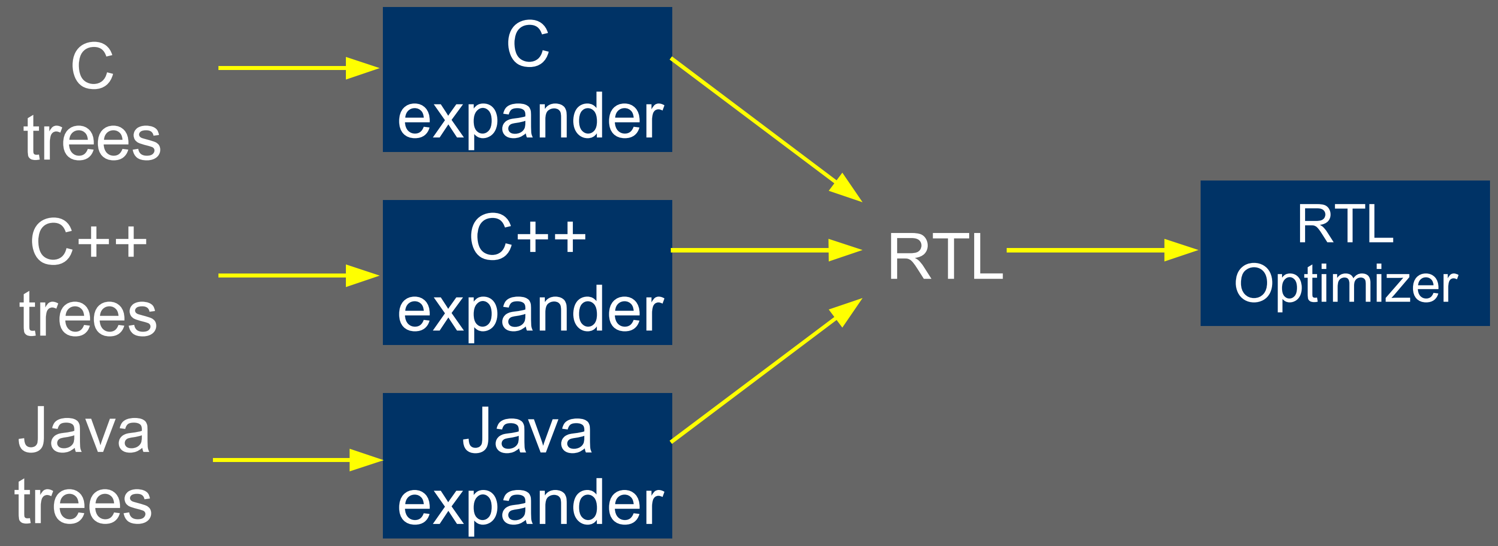
So, what's wrong with it?

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.

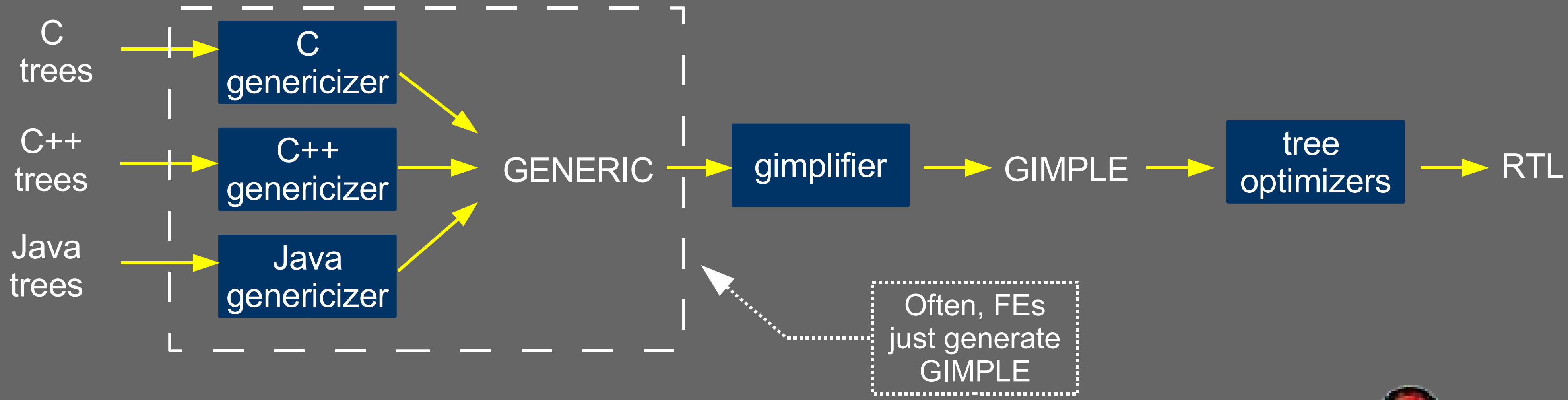
redhat

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.



redhat

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
```


redhat

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).

Statement Operands - 1

- Real operands
  - Non-aliased, scalar, local variables.
  - Atomic references to the whole object.
  - GIMPLE “registers” (may not fit in a physical register).

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

- Virtual operands
  - Globals, aliased, structures, arrays, pointer dereferences.
  - Potential and/or partial references to the object.
  - Distinction becomes important when building SSA form.

```
int x[10];
struct A y;
x[3] = y.f;
```


redhat

Statement Operands - 2

- Types of virtual operands:
  - Partial, potential and/or aliased stores (**V_MAY_DEF**)

```
p = (cond) ? &a : &b
# a = V_MAY_DEF <a>
# b = V_MAY_DEF <b>
*p = x + 1
```

```
# a = V_MAY_DEF <a>
# b = V_MAY_DEF <b>
foo (p)
# s = V_MAY_DEF <s>
s.f = y
```

  - Partial, total and/or aliased loads (**V_USE**)

```
# V_USE <s>
y = s.f
```

```
# V_USE <a>
# V_USE <b>
x = *p
```

  - Killing definitions of aggregates and globals (**V_MUST_DEF**)

```
# s = V_MUST_DEF <s>
s = u
```


redhat

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 = V_MAY_DEF <MT>
*p = ...
```

- Since MT is aliased with `x`:

```
# x = V_MAY_DEF <x>
*p = ...
```

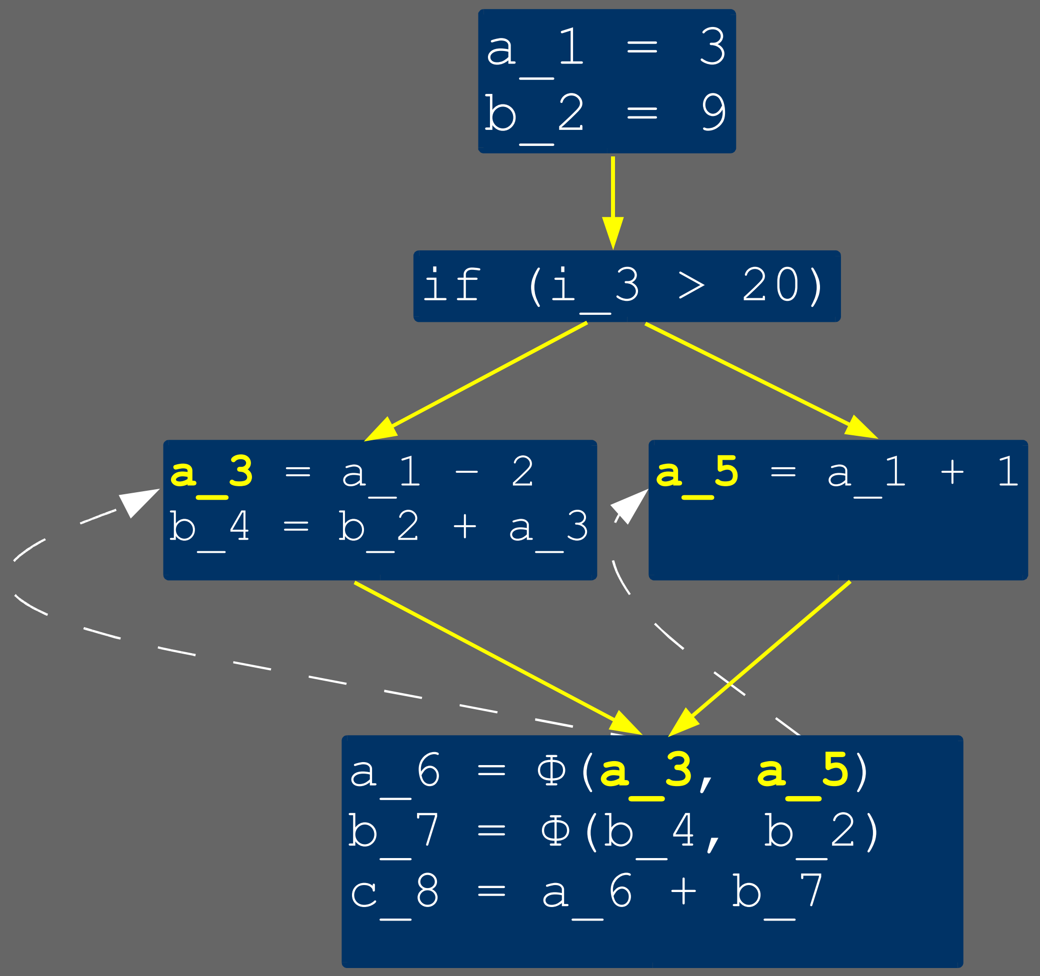
Alias Analysis - 2

- 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.

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.

redhat

Implementing SSA passes - 1

- To implement a new pass
  1. Create an instance of `struct tree_opt_pass`
  2. Declare it in `tree-pass.h`
  3. Sequence it in `init_tree_optimization_passes`
- APIs available for
  - CFG: block/edge insertion, removal, dominance information, block iterators, dominance tree walker.
  - Statements: insertion in block and edge, removal, iterators, replacement.
  - Operands: iterators, replacement.
  - Loop discovery and manipulation.
  - Data dependency information (scalar evolutions framework).

redhat

Implementing SSA passes - 2

- Other available infrastructure
  - Debugging dumps (`-fdump-tree-...`)
  - Timers for profiling passes (`-ftime-report`)
  - CFG/GIMPLE/SSA verification (`--enable-checking`)
  - Generic value propagation engine with callbacks for statement and Φ node visits.
  - Generic use-def chain walker.
  - Support in test harness for scanning dump files looking for specific transformations.
  - Pass manager for scheduling passes and describing interdependencies, attributes required and attributes provided.

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*).
  - 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).

redhat

Future Work - 1

- Short term
  - Remove dominator-based optimizations.
  - GVN PRE.
  - Value range propagation.
  - Conditional copy propagation.
  - Copy and constant propagation of loads and stores.
- Medium term
  - Stabilization and speedup (Bugzilla).
  - Documentation.
  - Tie into fledgling IPA framework.
  - More loop optimizers (LNO branch).

Future Work - 2

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

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.

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.