An Architectural Overview of GCC

Diego Novillo
dnovillo@redhat.com
Red Hat Canada

Linux Symposium
Ottawa, Canada, July 2006

Topics

1. Overview
2. Development model
3. Compiler infrastructure
4. Intermediate Representation
5. Current status and future work

Overview

- Key strengths
  - Widely popular
  - Freely available almost everywhere
  - Open development model
- However
  - Large code base (2.2 MLOC) and aging (~15 years)
  - Difficult to maintain and enhance
  - Technically demanding
- Recent architectural changes bring hope

Development Model

Development Model

- Project organization
  - Steering Committee → Administrative, political
  - Release Manager → Release coordination
  - Maintainers → Design, implementation
- Three main stages (~2 months each)
  - Stage 1 → Big disruptive changes.
  - Stage 2 → Stabilization, minor features.
  - Stage 3 → Bug fixes only (driven by bugzilla, mostly).

Development Model

- Major development is done in branches
  - Design/implementation discussion on public lists
  - Frequent merges from mainline
  - Final contribution into mainline only at stage 1 and approved by maintainers
- Anyone with SVN access may create a development branch
- Vendors create own branches from FSF release branches

Development Model

- All contributors must sign FSF copyright release
  - Even if only working on branches
- Three levels of access
  - Snapshots (weekly)
  - Anonymous SVN
  - Read/write SVN

Compiler Infrastructure

Source code

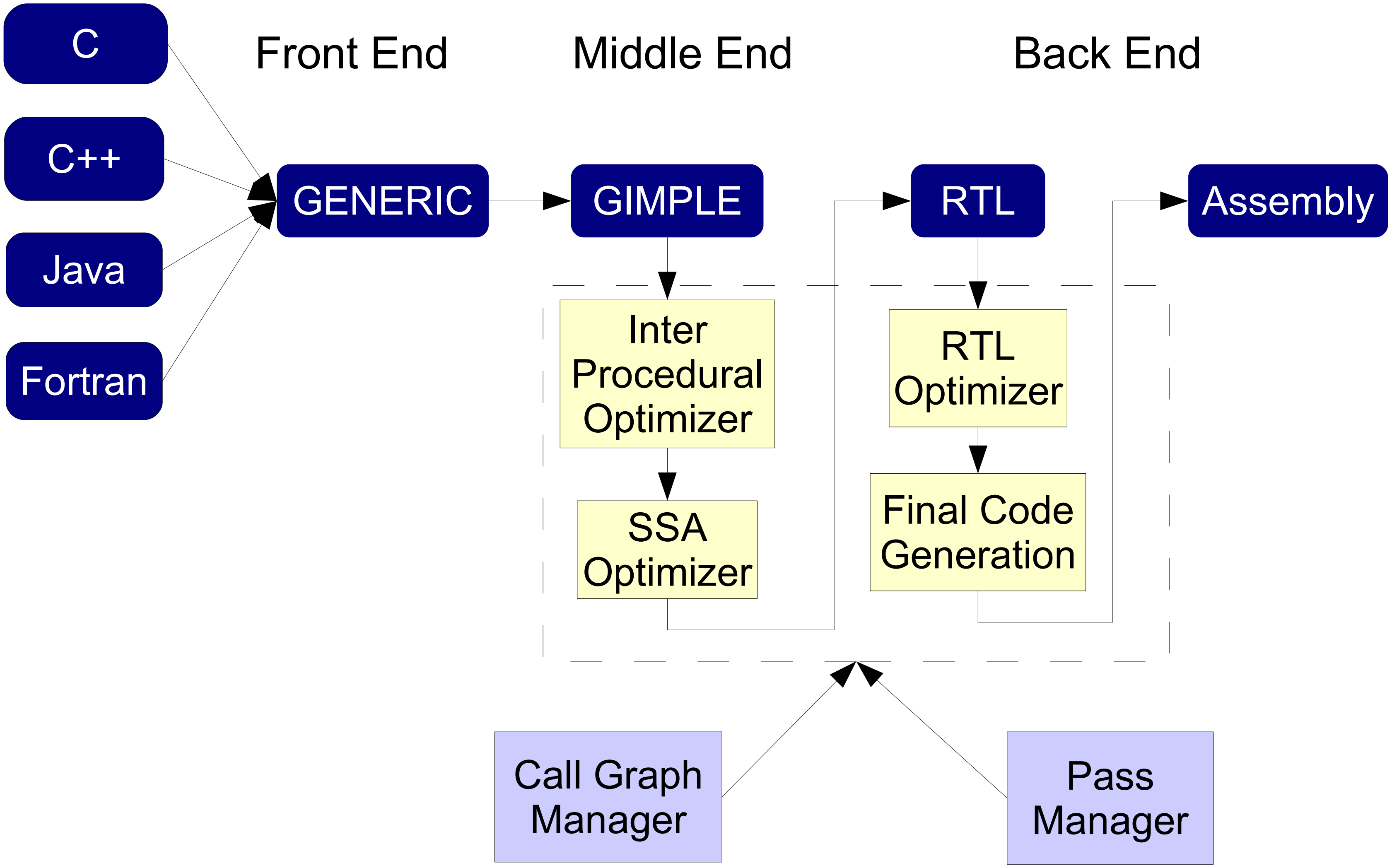
`<src>`

`gcc`	Front/middle/back end
`libcpp`	Pre-processor
`libada`	Ada runtime
`libstdc++-v3`	C++ runtime
`libgfortran`	Fortran runtime
`libobjc`	Objective-C runtime

`boehm-gc`
`libffi`
`libjava`
`zlib`
	Java runtime

`libiberty`	Utility functions and generic data structures
`libgomp`	OpenMP runtime
`libssp`	Stack Smash Protection runtime
`libmudflap`	Pointer/memory check runtime
`libdecnumber`	Decimal arithmetic library

Compiler pipeline



SSA Optimizers

- Operate on GIMPLE IL
- Around 100 passes
  - Vectorization
  - Various loop optimizations
  - Traditional scalar optimizations: CCP, DCE, DSE, FRE, PRE, VRP, SRA, jump threading, forward propagation
  - Field-sensitive, points-to alias analysis
  - Pointer checking instrumentation for C/C++

RTL Optimizers

- Operate closer to the hardware
  - Register allocation
  - Scheduling
  - Software pipelining
  - Common subexpression elimination
  - Instruction recombination
  - Mode switching reduction
  - Peephole optimizations
  - Machine specific reorganization

Intermediate Representation

GENERIC and GIMPLE

- 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

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
```

GIMPLE

- 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” (*real operands*)
- Globals, aliased variables and non-scalar types treated as “memory” (*virtual operands*)

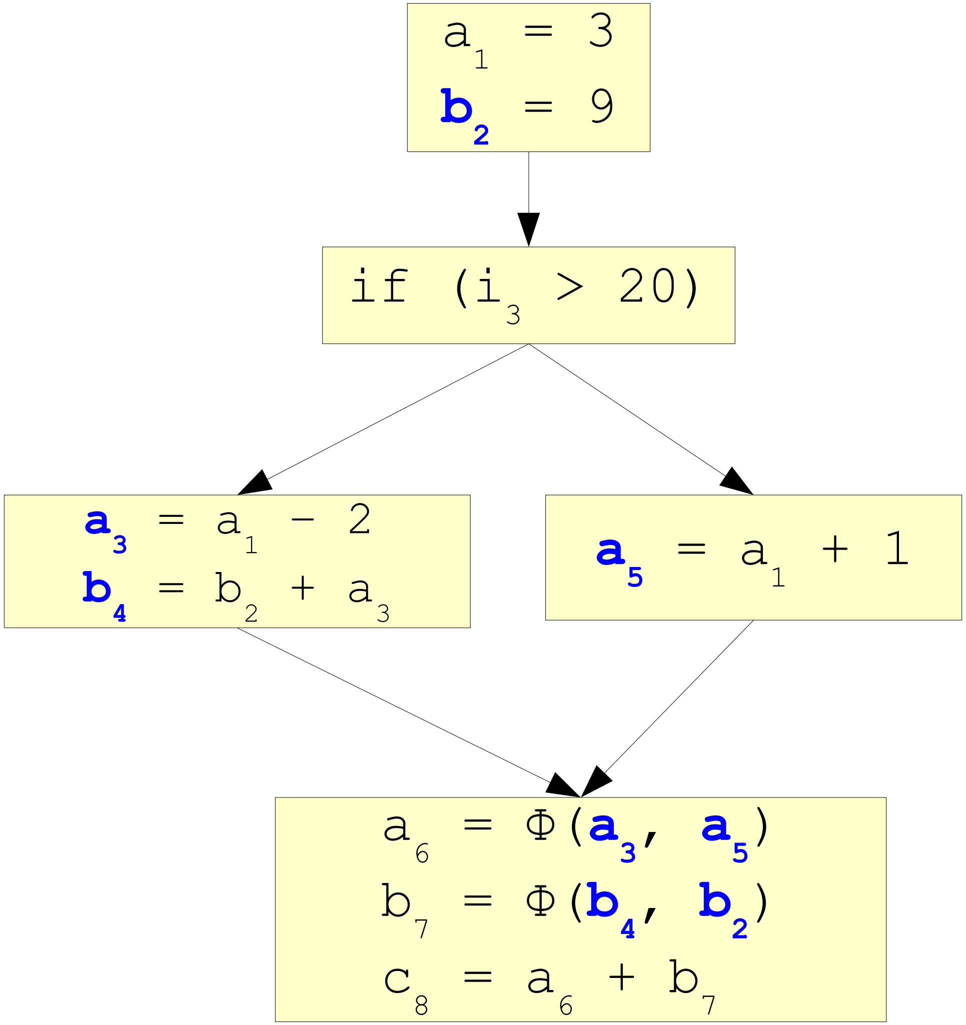
GIMPLE

- 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

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)



$a_1 = 3$
$\mathbf{b_2} = 9$

if $(i_3 > 20)$

$\mathbf{a_3} = a_1 - 2$
$\mathbf{b_4} = b_2 + a_3$

$\mathbf{a_5} = a_1 + 1$

$a_6 = \Phi(\mathbf{a_3}, \mathbf{a_5})$
$b_7 = \Phi(\mathbf{b_4}, \mathbf{b_2})$
$c_8 = a_6 + b_7$

SSA Form

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

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

  - Program is taken out of SSA form for RTL generation (new symbols are created to fix OLR)

SSA Form

- Factored Use-Def Chains (FUD Chains)
  - Also known as Virtual SSA Form
  - Used for virtual operands.
  - All names refer to the *same object*.
  - Optimizers may not produce OLR for virtual operands.

RTL

- Register Transfer Language
- Assembler for abstract machine with infinite registers

```
b = a - 1
```

```
(set (reg/v:SI 59 [ b ])
     (plus:SI (reg/v:SI 60 [ a ]
              (const_int -1 [0xffffffff]))))
```

RTL

- Abstracts
  - Register classes
  - Memory addressing modes
  - Word sizes and types
  - Compare-and-branch instructions
  - Calling conventions
  - Bitfield operations
  - Type and sign conversions

RTL

- Abstractions defined and controlled in *machine description* file

  `gcc/config/<arch>/<arch>.md`

- MD file defines all code generation mappings (instruction templates)
- Target description macros describe hardware capabilities (register classes, calling conventions, type sizes, etc)

Current Status
and
Future Work

Current Status

- New Intermediate Representations decouple Front End and Back End
- Increased internal modularity
- Lots of new features
  - Fortran 95, mudflap, vectorizer, OpenMP, inter/intra procedural optimizers, stack protection, profiling, etc.
- Easier to modify

Future Work

- Static analysis support
  - Extensibility mechanism to allow 3rd party tools
- Link time optimizations
  - Write intermediate representation
  - Read and combine multiple compilation units
- Dynamic compilation
  - Emit bytecodes
  - Implement virtual machine with optimizing JIT

Contacts

- Home page http://gcc.gnu.org/
- Wiki http://gcc.gnu.org/wiki
- Mailing lists
  - gcc@gcc.gnu.org
  - gcc-patches@gcc.gnu.org
  - gcc-help@gcc.gnu.org
- IRC
  - irc.oft.net/#gcc

Additional Implementation Details

Statement Operands

- 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;
```

Statement Operands

- 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 = V_MAY_DEF <x>
# VUSE <y>
x[3] = y.f
```

Statement Operands

- Partial, potential and/or aliased stores

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

- Partial, total and/or aliased loads

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

Alias Analysis

- 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

- Symbol Memory Tags (SMT)
  - 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 Optimizations

- To implement a new pass
  - Create an instance of `struct tree_opt_pass`
  - Declare it in `tree-pass.h`
  - Sequence it in `init_tree_optimization_passes`

Implementing Optimizations

- 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).

Implementing Optimizations

- 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.