GCC Internals

Diego Novillo
dnovillo@redhat.com
Red Hat Canada

CGO 2007
San Jose, California
March 2007

Outline

1. Overview
2. Source code organization
3. Internal architecture
4. Passes

NOTE: Internal information valid for GCC mainline as of 2007-03-02

1. Overview

- Major features
- Brief history
- Development model

Major Features

Availability

- Free software (GPL)
- Open and distributed development process
- System compiler for popular UNIX variants
- Large number of platforms (deeply embedded to big iron)
- Supports all major languages: C, C++, Java, Fortran 95, Ada, Objective-C, Objective-C++, etc

Major Features

Code quality

- Bootstraps on native platforms
- Warning-free
- Extensive regression testsuite
- Widely deployed in industrial and research projects
- Merit-based maintainership appointed by steering committee
- Peer review by maintainers
- Strict coding standards and patch reversion policy

Major Features

Analysis/Optimization

- SSA-based high-level global optimizer
- Constraint-based points-to alias analysis
- Data dependency analysis based on chains of recurrences
- Feedback directed optimization
- Interprocedural optimization
- Automatic pointer checking instrumentation
- Automatic loop vectorization
- OpenMP support

1. Overview

- Major features
- Brief history
- Development model

Brief History

GCC 1 (1987)

- Inspired on Pastel compiler (Lawrence Livermore Labs)
- Only C
- Translation done one statement at a time

GCC 2 (1992)

- Added C++
- Added RISC architecture support
- Closed development model challenged
- New features difficult to add

Brief History

EGCS (1997)

- Fork from GCC 2.x
- Many new features: Java, Chill, numerous embedded ports, new scheduler, new optimizations, integrated libstdc++

GCC 2.95 (1999)

- EGCS and GCC2 merge into GCC
- Type based alias analysis
- Chill front end
- ISO C99 support

Brief History

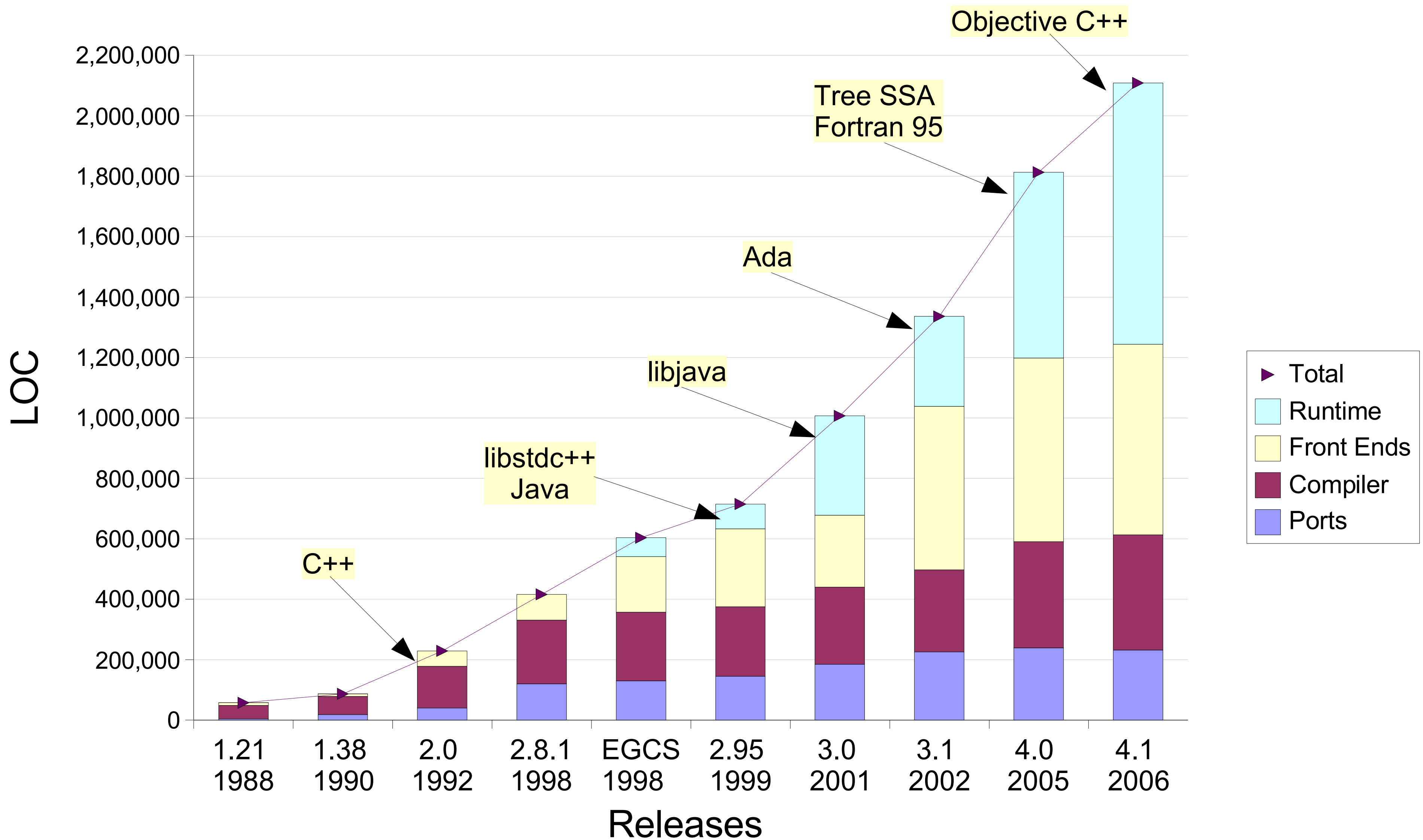
GCC 3 (2001)

- Integrated libjava
- Experimental SSA form on RTL
- Functions as trees

GCC 4 (2005)

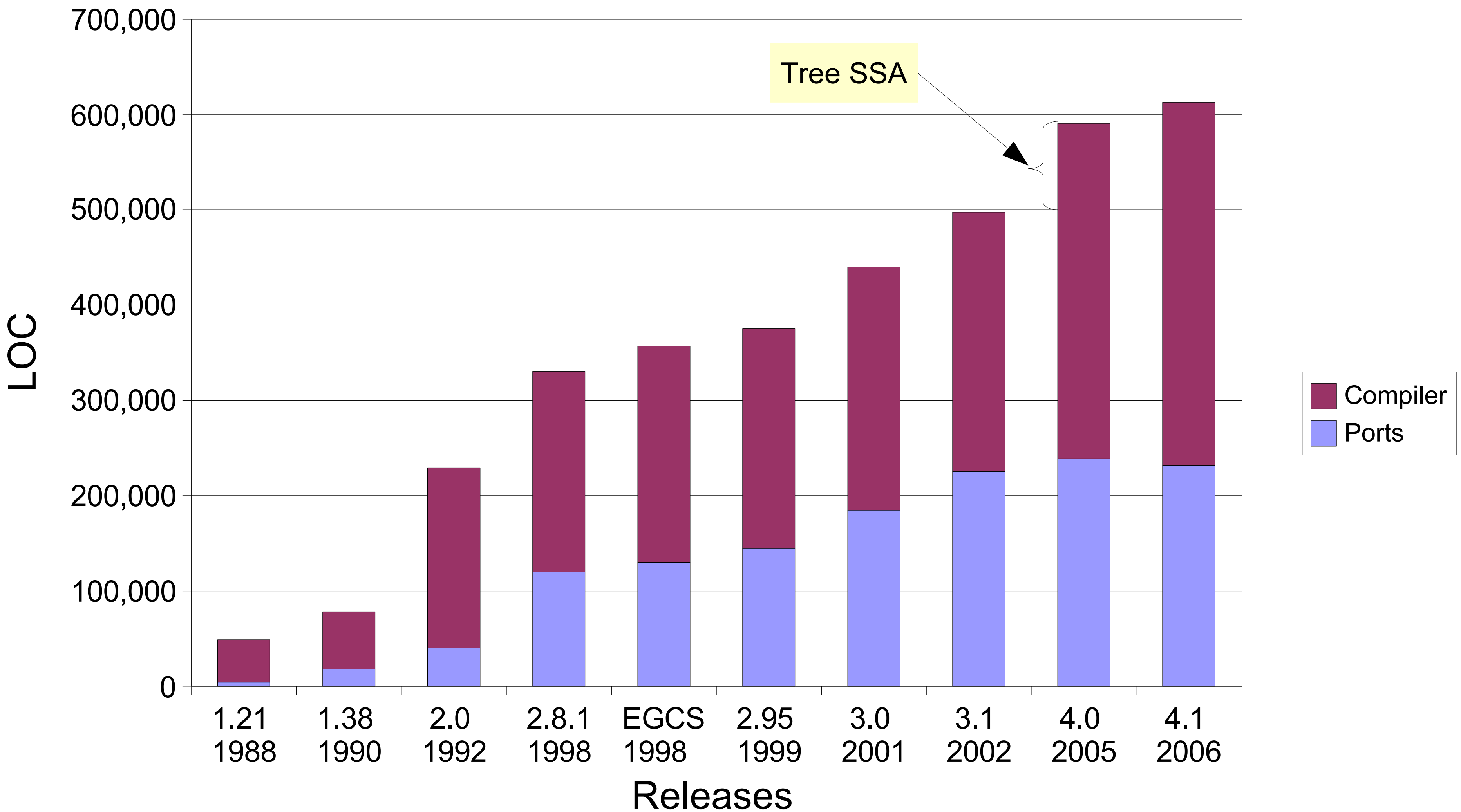
- Internal architecture overhaul (Tree SSA)
- Fortran 95
- Automatic vectorization

GCC Growth[1]



[1] generated using David A. Wheeler's 'SLOCCount'.

Core Compiler Growth[1]



[1] generated using David A. Wheeler's 'SLOCCount'.

1. Overview

- Major features
- Brief history
- 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 write-access may create a development branch
- Vendors create own branches from FSF release branches

Development Model

- All contributors **must** sign FSF copyright release
  - Even when working on branches
- Three levels of access
  - Snapshots (weekly)
  - Anonymous SVN
  - Read/write SVN
- Two main discussion lists
  - gcc@gcc.gnu.org
  - gcc-patches@gcc.gnu.org

Development Model

- Home page
  - http://gcc.gnu.org/
- Real time collaboration
  - IRC irc://irc.oftc.net/#gcc
  - Wiki http://gcc.gnu.org/wiki/
- Bug tracking
  - http://gcc.gnu.org/bugzilla/
- Patch tracking
  - http://gcc.gnu.org/wiki/GCC_Patch_Tracking/

2. Source code

- Source tree organization
- Configure, build, test
- Patch submission

Source code

- Getting the code for mainline (or trunk)

```
$ svn co svn://gcc.gnu.org/svn/gcc/trunk
```

- Build requirements (http://gcc.gnu.org/install)
  - ISO C90 compiler
  - GMP library
  - MPFR library
  - GNAT (only if building Ada)

Multiple precision floating point libraries

- Source code includes runtimes for all languages and extensive regression testsuite.

Source code

<src>

gcc	Front/middle/back ends
libgcc	Internal library for missing target features
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

Source code

<src>

► **gcc**	Core compiler and C front end
► ada	Ada front end
► **config**	Architecture-specific codegen (ports)
► cp	C++ front end
► doc	User manual and internal documentation
► fortran	Fortran front end
► ginclude	System headers (mainly freestanding support)
► java	Java front end
► objc	Objective C front end
► objcp	Objective C++ front end
► po	Portable object files for I18N
► **testsuite**	Regression tests
► treelang	Toy language front end

Core compiler files (<src>/gcc)

- Alias analysis
- Build support
- C front end
- CFG and callgraph
- Code generation
- Diagnostics
- Driver
- Profiling
- Internal data structures
- Mudflap
- OpenMP
- Option handling
- RTL optimizations
- Tree SSA optimizations

2. Source code

- Source tree organization
- Configure, build, test
- Patch submission

Configuring and Building

```
$ svn co svn://gcc.gnu.org/svn/gcc/trunk
$ mkdir bld && cd bld
$ ../trunk/configure --prefix=`pwd`
$ make all install
```

- Bootstrap is a 3 stage process
  - Stage 0 (host) compiler builds Stage 1 compiler
  - Stage 1 compiler builds Stage 2 compiler
  - Stage 2 compiler builds Stage 3 compiler
  - Stage 2 and Stage 3 compilers must be binary identical

Common configuration options

`--prefix`

- Installation root directory

`--enable-languages`

- Comma-separated list of language front ends to build
- Possible values

  `ada,c,c++,fortran,java,objc,obj-c++,treelang`
- Default values

  `c,c++,fortran,java,objc`

Common configuration options

`--disable-bootstrap`

– Build stage 1 compiler only

`--target`

– Specify target architecture for building a cross-compiler

– Target specification form is (roughly)

```
    cpu-manufacturer-os
    cpu-manufacturer-kernel-os
e.g.  x86_64-unknown-linux-gnu
      arm-unknown-elf
```

– All possible values in `<src>/config.sub`

Common configuration options

`--enable-checking=list`

- Perform compile-time consistency checks
- List of checks: `assert fold gc gcac misc rtl rtlflag runtime tree valgrind`
- Global values:

`yes`	→	`assert,misc,tree,gc,rtlflag,runtime`
`no`	→	Same as `--disable-checking`
`release`	→	Cheap checks `assert,runtime`
`all`	→	Everything except `valgrind`

SLOW!

Common build options

`-j N`

- Usually scales up to 1.5x to 2x number of processors

`all`

- Default `make` target. Knows whether to bootstrap or not

`install`

- Not necessary but useful to test installed compiler
- Set `LD_LIBRARY_PATH` afterward

`check`

- Use with `-k` to prevent stopping when some tests fail

Build results

- Staged compiler binaries

❶ `<bld>/stage1-{gcc,intl,libcpp,libdecnumber,libiberty}`

❷ `<bld>/prev-{gcc,intl,libcpp,libdecnumber,libiberty}`

❸ `<bld>/{gcc,intl,libcpp,libdecnumber,libiberty}`

- Runtime libraries are not staged, except `libgcc`

  `<bld>/<target-triplet>/lib*`

- Testsuite results

  `<bld>/gcc/testsuite/*.{log,sum}`

  `<bld>/<target-triplet>/lib*/testsuite/*.{log,sum}`

Build results

- Compiler is split in several binaries

`<bld>/gcc/xgcc`	Main driver
`<bld>/gcc/cc1`	C compiler
`<bld>/gcc/cc1plus`	C++ compiler
`<bld>/gcc/jc1`	Java compiler
`<bld>/gcc/f951`	Fortran compiler
`<bld>/gcc/gnat1`	Ada compiler

- Main driver forks one of the `*1` binaries
- `<bld>/gcc/xgcc -v` shows what compiler is used

Analyzing test results

- The best way is to have two trees built
  - pristine
  - pristine + patch
- Pristine tree can be recreated with

```
$ cp -a trunk trunk.pristine
$ cd trunk.pristine
$ svn revert -R .
```

- Configure and build both compilers with the exact same flags

Analyzing test results

- Use `<src>/trunk/contrib/compare_tests` to compare individual `.sum` files

```
$ cd <bld>/gcc/testsuite/gcc

$ compare_tests <bld.pristine>/gcc/testsuite/gcc/gcc.sum gcc.sum

  Tests that now fail, but worked before:
  gcc.c-torture/compile/20000403-2.c  -Os  (test for excess errors)

  Tests that now work, but didn't before:
  gcc.c-torture/compile/20000120-2.c  -O0  (test for excess errors)
  gcc.c-torture/compile/20000405-2.c  -Os  (test for excess errors)
```

2. Source code

- Source tree organization
- Configure, build, test
- Patch submission

Patch submission

- Non-trivial contributions require copyright assignment
- Code should follow the GNU coding conventions
  - http://www.gnu.org/prep/standards_toc.html
  - http://gcc.gnu.org/codingconventions.html
- Submission should include
  - `ChangeLog` describing what changed (not how nor why)
  - Test case (if applicable)
  - Patch itself generated with `svn diff` (context or unified)

Patch submission

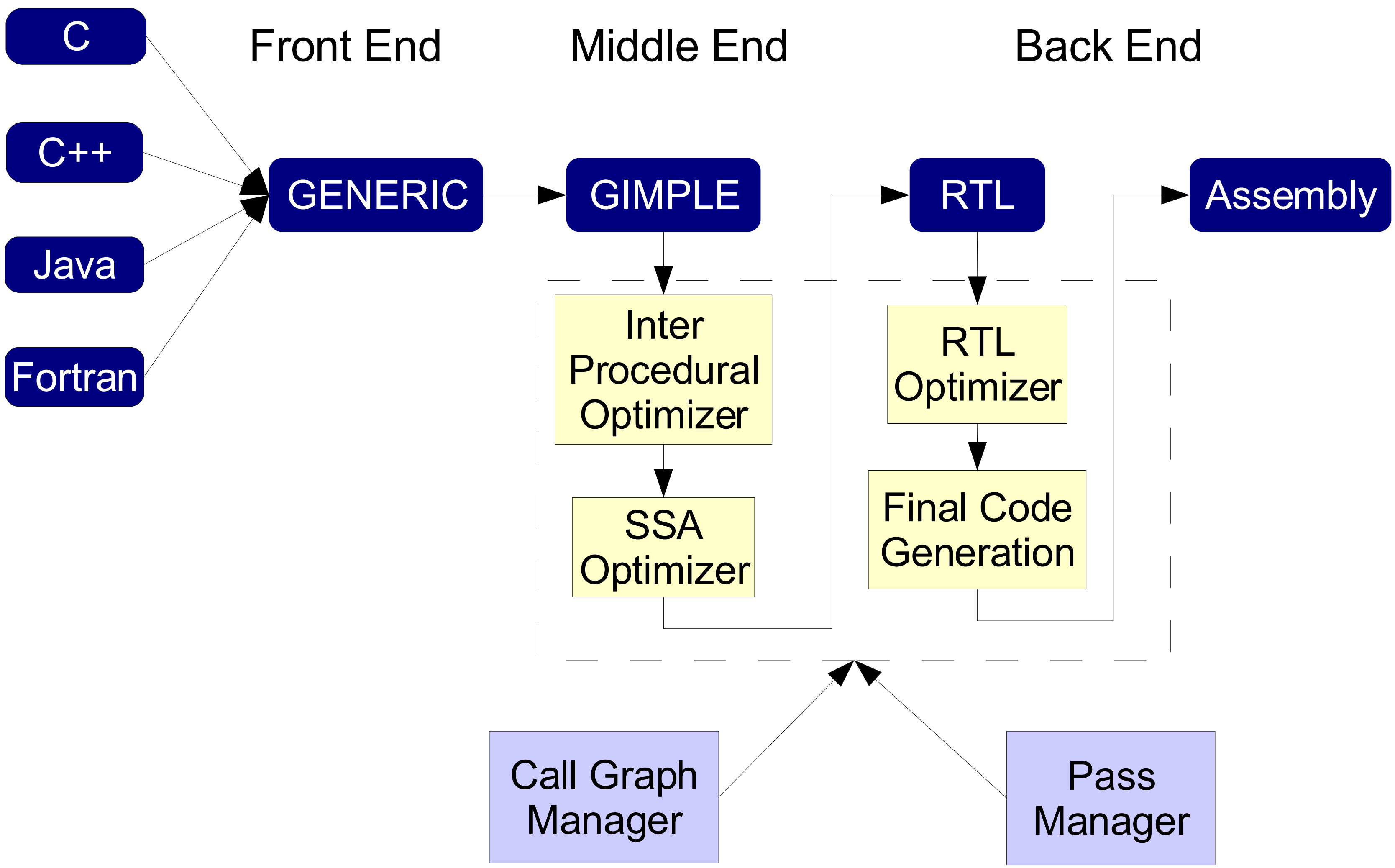
- When testing a patch
  1. Disable bootstrap
  2. Build C front end only
  3. Run regression testsuite
  4. Once all failures have been fixed
     - Enable all languages
     - Run regression testsuite again
  5. Enable bootstrap
  6. Run regression testsuite
- Patches are only accepted after #5 and #6 work

Not strictly necessary, but recommended

3. Internal architecture

- Compiler pipeline
- Intermediate representations
- CFG, statements, operands
- Alias analysis
- SSA forms
- Code generation

Compiler pipeline



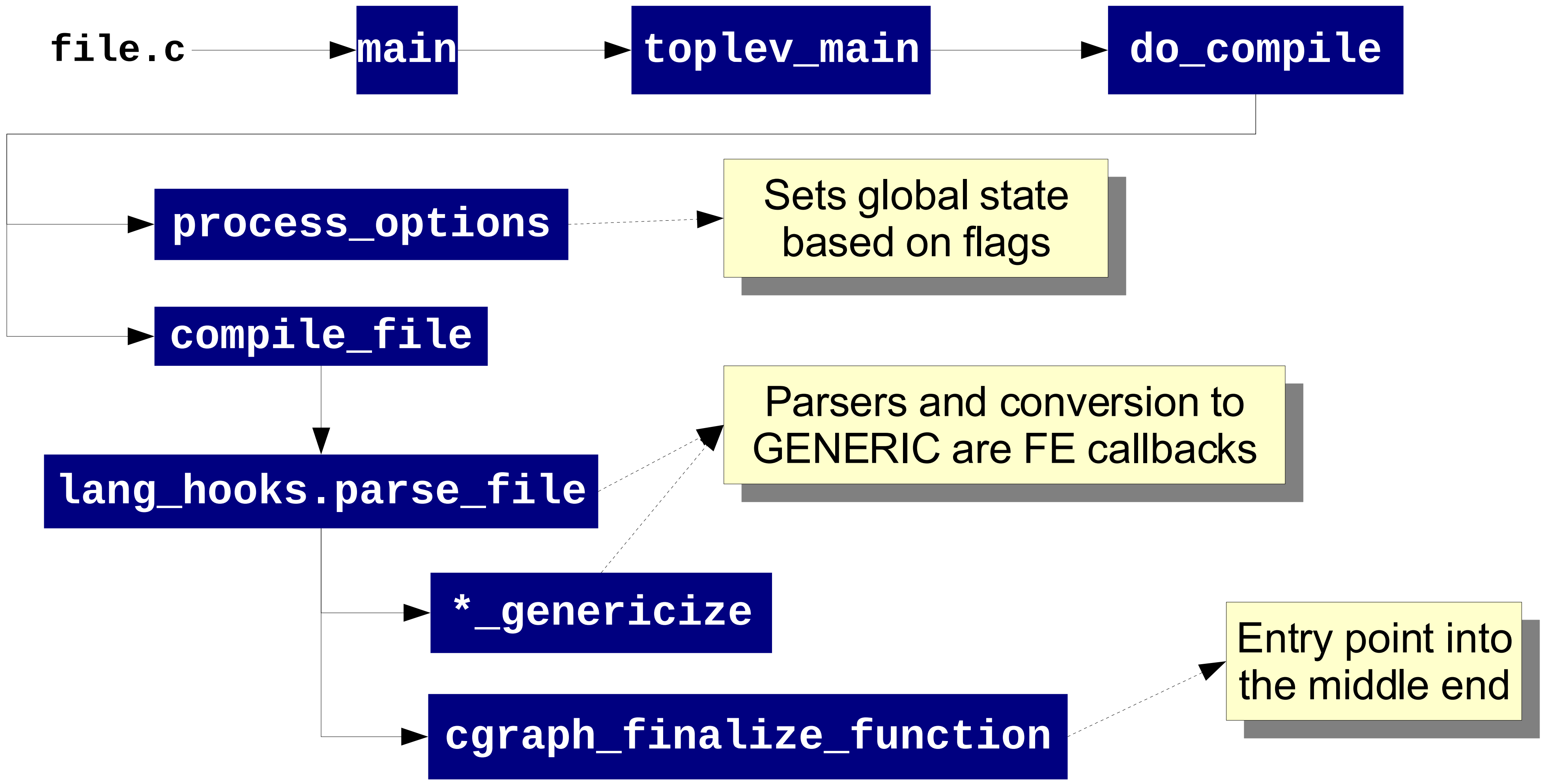
SSA Optimizers

- Operate on GIMPLE
- 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++
  - Interprocedural analysis and optimizations: CCP, inlining, points-to analysis, pure/const and type escape analysis

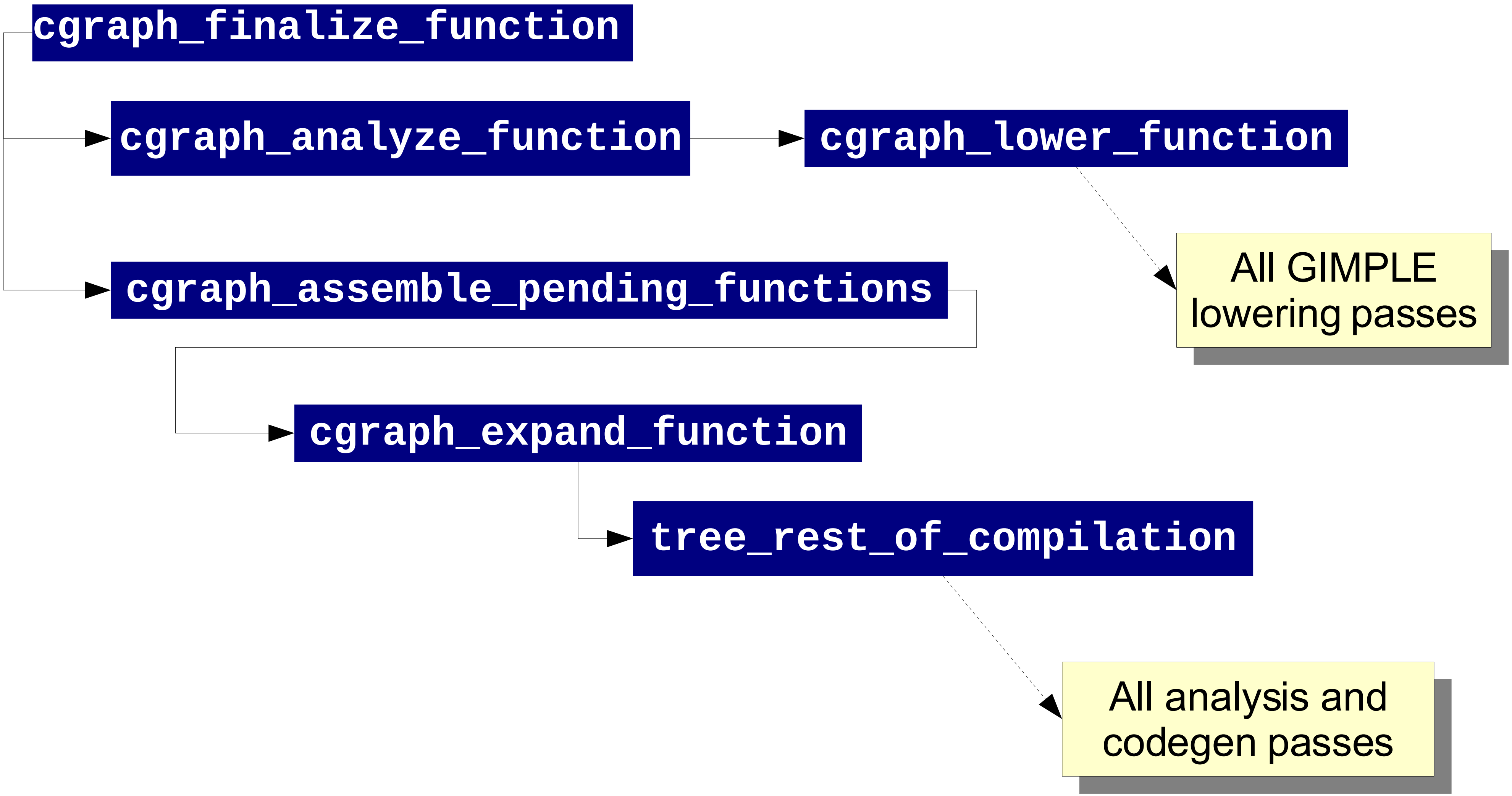
RTL Optimizers

- Around 70 passes
- Operate closer to the target
  - Register allocation
  - Scheduling
  - Software pipelining
  - Common subexpression elimination
  - Instruction recombination
  - Mode switching reduction
  - Peephole optimizations
  - Machine specific reorganization

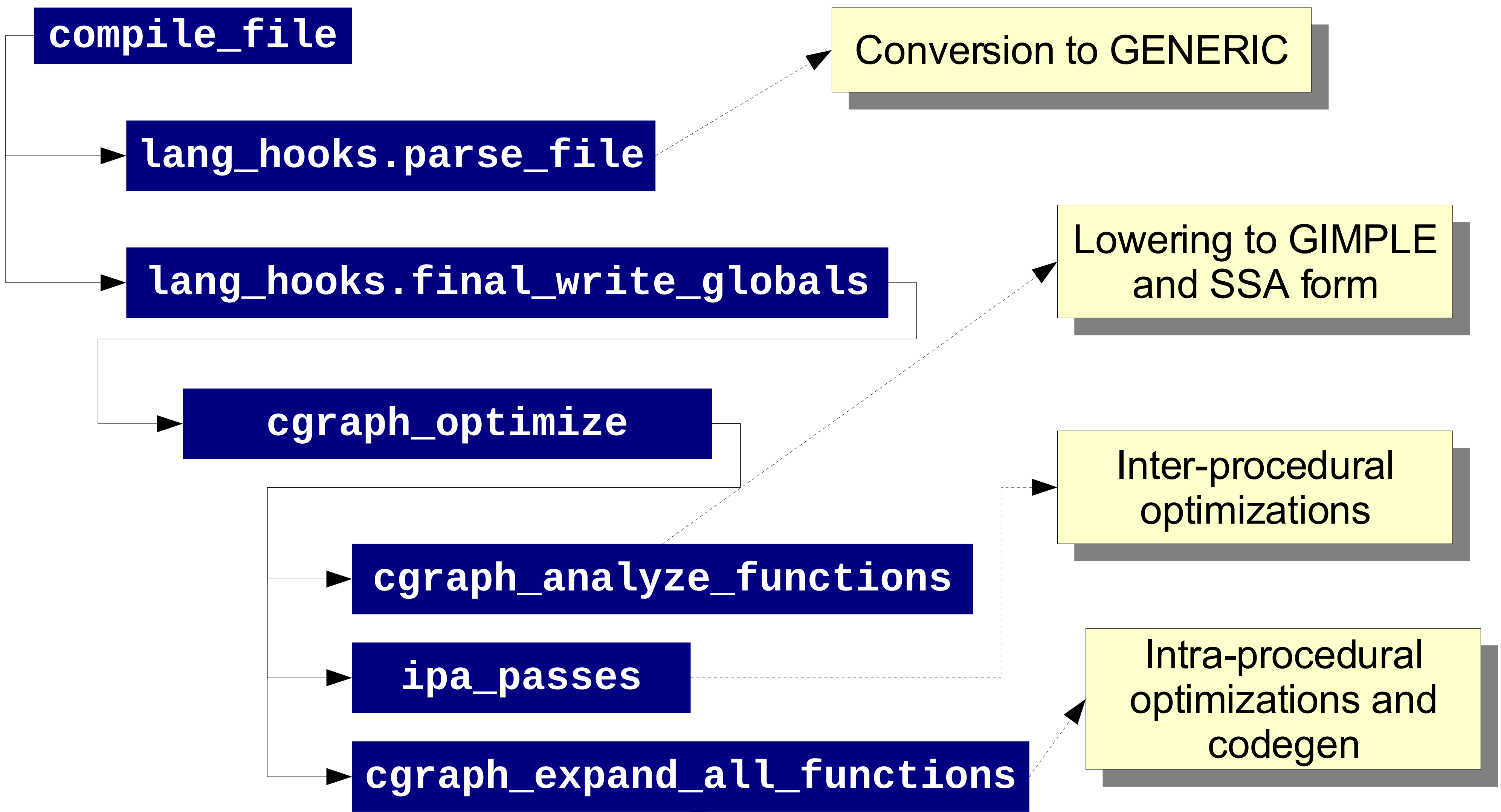
Simplified compilation flow (O0)



Simplified compilation flow (O0)



Simplified compilation flow (O1+)



3. Internal architecture

- Compiler pipeline
- Intermediate representations
- CFG, statements, operands
- Alias analysis
- SSA forms
- Code generation

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)
  - High GIMPLE → lexical scopes and inline parallel regions
  - Low GIMPLE → no scopes and out-of-line parallel regions
- It contains extensions to represent explicit parallelism (OpenMP)

RTL

- Register Transfer Language ≈ assembler for an abstract machine with infinite registers
- It represents low level features
  - Register classes
  - Memory addressing modes
  - Word sizes and types
  - Compare-and-branch instructions
  - Calling conventions
  - Bitfield operations
  - Type and sign conversions

RTL

```
b = a - 1
```

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

- It is commonly represented in LISP-like form
- Operands do not have types, but type modes
- In this case they are all `SImode` (4-byte integers)

3. Internal architecture

- Compiler pipeline
- Intermediate representations
- Control/data structures
- Alias analysis
- SSA forms
- Code generation

Callgraph

- Every internal/external function is a node of type `struct cgraph_node`
- Call sites represented with edges of type `struct cgraph_edge`
- Every cgraph node contains
  - Pointer to function declaration
  - List of callers
  - List of callees
  - Nested functions (if any)
- Indirect calls are not represented

Callgraph

- Callgraph manager drives intraprocedural optimization passes
- For every node in the callgraph, it sets `cfun` and `current_function_decl`
- IPA passes must traverse callgraph on their own
- Given a cgraph node

```
DECL_STRUCT_FUNCTION (node->decl)
```

points to the `struct function` instance that contains all the necessary control and data flow information for the function

Control Flow Graph

- Built early during lowering
- Survives until late in RTL
  - Right before machine dependent transformations (`pass_machine_reorg`)
- In GIMPLE, instruction stream is physically split into blocks
  - All jump instructions replaced with edges
- In RTL, the CFG is laid out over the double-linked instruction stream
  - Jump instructions preserved

Using the CFG

- Every CFG accessor requires a `struct function` argument
- In intraprocedural mode, accessors have shorthand aliases that use `cfun` by default
- CFG is an array of double-linked blocks
- The same data structures are used for GIMPLE and RTL
- Manipulation functions are callbacks that point to the appropriate RTL or GIMPLE versions

Using the CFG - Callbacks

- Declared in `struct cfg_hooks`

```
create_basic_block
redirect_edge_and_branch
delete_basic_block
can_merge_blocks_p
merge_blocks
can_duplicate_block_p
duplicate_block
split_edge
...
```

- Mostly used by generic CFG cleanup code
- Passes working with one IL may make direct calls

Using the CFG - Accessors

`basic_block_info_for_function(fn)` `basic_block_info`	Sparse array of basic blocks
`BASIC_BLOCK_FOR_FUNCTION(fn, n)` `BASIC_BLOCK (n)`	Get basic block N
`n_basic_blocks_for_function(fn)` `n_basic_blocks`	Number of blocks
`n_edges_for_function(fn)` `n_edges`	Number of edges
`last_basic_block_for_function(fn)` `last_basic_block`	First free slot in array of blocks (≠ `n_basic_blocks`)
`ENTRY_BLOCK_PTR_FOR_FUNCTION(fn)` `ENTRY_BLOCK_PTR`	Entry point
`EXIT_BLOCK_PTR_FOR_FUNCTION(fn)` `EXIT_BLOCK_PTR`	Exit point

Using the CFG - Traversals

- The block array is sparse, never iterate with

  ~~`for (i = 0; i < n_basic_blocks; i++)`~~

- Basic blocks are of type `basic_block`
- Edges are of type `edge`
- Linear traversals

```
FOR_EACH_BB_FN (bb, fn)
  FOR_EACH_BB (bb)

FOR_EACH_BB_REVERSE_FN (bb, fn)
  FOR_EACH_BB_REVERSE (bb)

FOR_BB_BETWEEN (bb, from, to, {next_bb|prev_bb})
```

Using the CFG - Traversals

- Traversing successors/predecessors of block bb

```
edge e;
edge_iterator ei;
FOR_EACH_EDGE (e, ei, bb->{succs|preds})
   do_something (e);
```

- Linear CFG traversals are essentially random
- Ordered walks possible with dominator traversals
  - Direct dominator traversals
  - Indirect dominator traversals via walker w/ callbacks

Using the CFG - Traversals

- Direct dominator traversals
  - Walking all blocks dominated by **bb**

```
for (son = first_dom_son (CDI_DOMINATORS, bb);
     son;
     son = next_dom_son (CDI_DOMINATORS, son))
```

  - Walking all blocks post-dominated by **bb**

```
for (son = first_dom_son (CDI_POST_DOMINATORS, bb);
     son;
     son = next_dom_son (CDI_POST_DOMINATORS, son)
```

  - To start at the top of the CFG

```
FOR_EACH_EDGE (e, ei, ENTRY_BLOCK_PTR->succs)
  dom_traversal (e->dest);
```

Using the CFG - Traversals

- `walk_dominator_tree()`
- Dominator tree walker with callbacks
- Walks blocks and statements in either direction
- Up to six walker callbacks supported

  Before **and** after dominator children

x2

  1. Before walking statements
  2. Called for every GIMPLE statement in the block
  3. After walking statements

- Walker can also provide block-local data to keep pass-specific information during traversal

GIMPLE statements

- GIMPLE statements are instances of type `tree`
- Every block contains a double-linked list of statements
- Manipulation done through iterators

```
block_statement_iterator si;
basic_block bb;
FOR_EACH_BB(bb)
  for (si = bsi_start(bb); !bsi_end_p(si); bsi_next(&si))
   print_generic_stmt (stderr, bsi_stmt(si), 0);
```

- Statements can be inserted and removed inside the block or on edges

GIMPLE statement operands

- Real operands (DEF, USE)
  - Non-aliased, scalar, local variables
  - Atomic references to the whole object
  - GIMPLE “registers” (may not fit in a physical register)
- Virtual or memory operands (VDEF, VUSE)
  - Globals, aliased, structures, arrays, pointer dereferences
  - Potential and/or partial references to the object
  - Distinction becomes important when building SSA form

GIMPLE statement operands

- Real operands are part of the statement

```
int a, b, c
c = a + b
```

- Virtual operands are represented by two operators VDEF and VUSE

```
int c[100]
int *p = (i > 10) ? &a : &b
# a = VDEF <a>
# b = VDEF <b>
# VUSE <c>
*p = c[i]
```

a or b *may* be defined

`c[i]` is a *partial* load from c

Accessing GIMPLE operands

```
use_operand_p use;
ssa_op_iter i;
FOR_EACH_SSA_USE_OPERAND (use, stmt, i, SSA_OP_ALL_USES)
    {
        tree op = USE_FROM_PTR (use);
        print_generic_expr (stderr, op, 0);
    }
```

- Prints all USE and VUSE operands from `stmt`
- SSA_OP_ALL_USES filters which operands are of interest during iteration
- For DEF and VDEF operands, replace “use” with “def” above

RTL statements

- RTL statements (insns) are instances of type `rtx`
- Unlike GIMPLE statements, RTL insns contain embedded links
- Six types of RTL insns

`INSN`	Regular, non-jumping instruction
`JUMP_INSN`	Conditional and unconditional jumps
`CALL_INSN`	Function calls
`CODE_LABEL`	Target label for `JUMP_INSN`
`BARRIER`	Control flow stops here
`NOTE`	Debugging infomation

RTL statements

- Some elements of an RTL insn

`PREV_INSN`	Previous statement
`NEXT_INSN`	Next statement
`PATTERN`	Body of the statement
`INSN_CODE`	Number for the matching machine description pattern (-1 if not yet recog'd)
`LOG_LINKS`	Links dependent insns in the same block Used for instruction combination
`REG_NOTES`	Annotations regarding register usage

RTL statements

- Traversing all RTL statements

```
basic_block bb;
FOR_EACH_BB (bb)
   {
      rtx insn = BB_HEAD (bb);
      while (insn != BB_END (bb))
         {
            print_rtl_single (stderr, insn);
            insn = NEXT_INSN (insn);
         }
   }
```

RTL operands

- No operand iterators, but RTL expressions are very regular
- Number of operands and their types are defined in `rtl.def`

`GET_RTX_LENGTH`	Number of operands
`GET_RTX_FORMAT`	Format string describing operand types
`XEXP/XINT/XSTR/...`	Operand accessors
`GET_RTX_CLASS`	Similar expressions are categorized in classes

RTL operands

- Operands and expressions have modes, not types
- Supported modes will depend on target capabilities
- Some common modes

`QImode`	Quarter Integer (single byte)
`HImode`	Half Integer (two bytes)
`SImode`	Single Integer (four bytes)
`DImode`	Double Integer (eight bytes)
...	

- Modes are defined in `machmode.def`

3. Internal architecture

- Compiler pipeline
- Intermediate representations
- Control/data structures
- Alias analysis
- SSA forms
- Code generation

Overview

- GIMPLE represents alias information explicitly
- Alias analysis is just another pass
  - Artificial symbols represent memory expressions (virtual operands)
  - FUD-chains computed on virtual operands → Virtual SSA
- Transformations may prove a symbol non-addressable
  - Promoted to GIMPLE register
  - Requires another aliasing pass

Memory expressions in GIMPLE

- At most one memory load and one memory store per statement
  - Loads only allowed on RHS of assignments
  - Stores only allowed on LHS of assignments
- Gimplifier will enforce this property
- Dataflow on memory represented explictly
  - Factored Use-Def (FUD) chains or “Virtual SSA”
  - Requires a symbolic representation of memory

Symbolic Representation of Memory

- Aliased memory referenced via pointers
- GIMPLE only allows single-level pointers

Invalid	Valid
`**p`	`t.1 = *p` `*t.1`
`*(a[3].ptr)`	`t.1 = a[3].ptr` `*t.1`

Symbolic Representation of Memory

- Pointer `P` is associated with memory tag `MT`
  - `MT` represents the set of variables pointed-to by `P`
- So `*P` is a reference to `MT`

```
if (...)
  p = &a
else
  p = &b
*p = 5
```

p points-to `{a, b}`
p has memory tag `MT`

Interpreted as `MT = 5`

Associating Memory with Symbols

- Alias analysis
  - Builds points-to sets and memory tags
- Structural analysis
  - Builds field tags (sub-variables)
- Operand scanner
  - Scans memory expressions to extract tags
  - Prunes alias sets based on expression structure

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

- Since MT is aliased with x:

```
# x = VDEF <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

Alias analysis in RTL

- Pure query system
- Pairwise disambiguation of memory references
  - Does store to A affect load from B?
  - Mostly type-based (same predicates used in GIMPLE's TBAA)
- Very little information passed on from GIMPLE

Alias analysis in RTL

- Some symbolic information preserved in RTL memory expressions
  - Base + offset associated to aggregate refs
  - Memory symbols
- Tracking of memory addresses by propagating values through registers
- Each pass is responsible for querying the alias system with pairs of addresses

Alias analysis in RTL – Problems

- Big impedance between GIMPLE and RTL
  - No/little information transfer
  - Producers and consumers use different models
  - GIMPLE → explicit representation in IL
  - RTL → query-based disambiguation
- Work underway to resolve this mismatch
  - Results of alias analysis exported from GIMPLE
  - Adapt explicit representation to query system

Alias Analysis

- Points-to alias analysis (PTAA)
  - Based on constraint graphs
  - Field and flow sensitive, context insensitive
  - Intra-procedural (inter-procedural in 4.2)
  - Fairly precise
- Type-based analysis (TBAA)
  - Based on input language rules
  - Field sensitive, flow insensitive
  - Very imprecise

Alias Analysis

- Two kinds of pointers are considered
  - Symbols: Points-to is flow-insensitive
    - Associated to Symbol Memory Tags (SMT)
  - SSA names: Points-to is flow-sensitive
    - Associated to Name Memory Tags (NMT)
- Given pointer dereference `*ptr`$_{42}$
  - If `ptr`$_{42}$ has NMT, use it
  - If not, fall back to SMT associated with `ptr`

Structural Analysis

- Separate structure fields are assigned distinct symbols

```
struct A
{
  int x;
  int y;
  int z;
};

struct A a;
```

- Variable a will have 3 sub-variables
  `{ SFT.1, SFT.2, SFT.3 }`

- References to each field are mapped to the corresponding sub-variable

redhat

IL Representation

```
foo (i, a, b, *p)
{
  p =(i > 10) ? &a : &b
  *p = 3
  return a + b
}
```

```
foo (i, a, b, *p)
{
  p = (i > 10) ? &a : &b

  # a = VDEF <a>
  # b = VDEF <b>
  *p = 3

  # VUSE <a>
  t1 = a

  # VUSE <b>
  t2 = b

  t3 = t1 + t2
  return t3
}
```

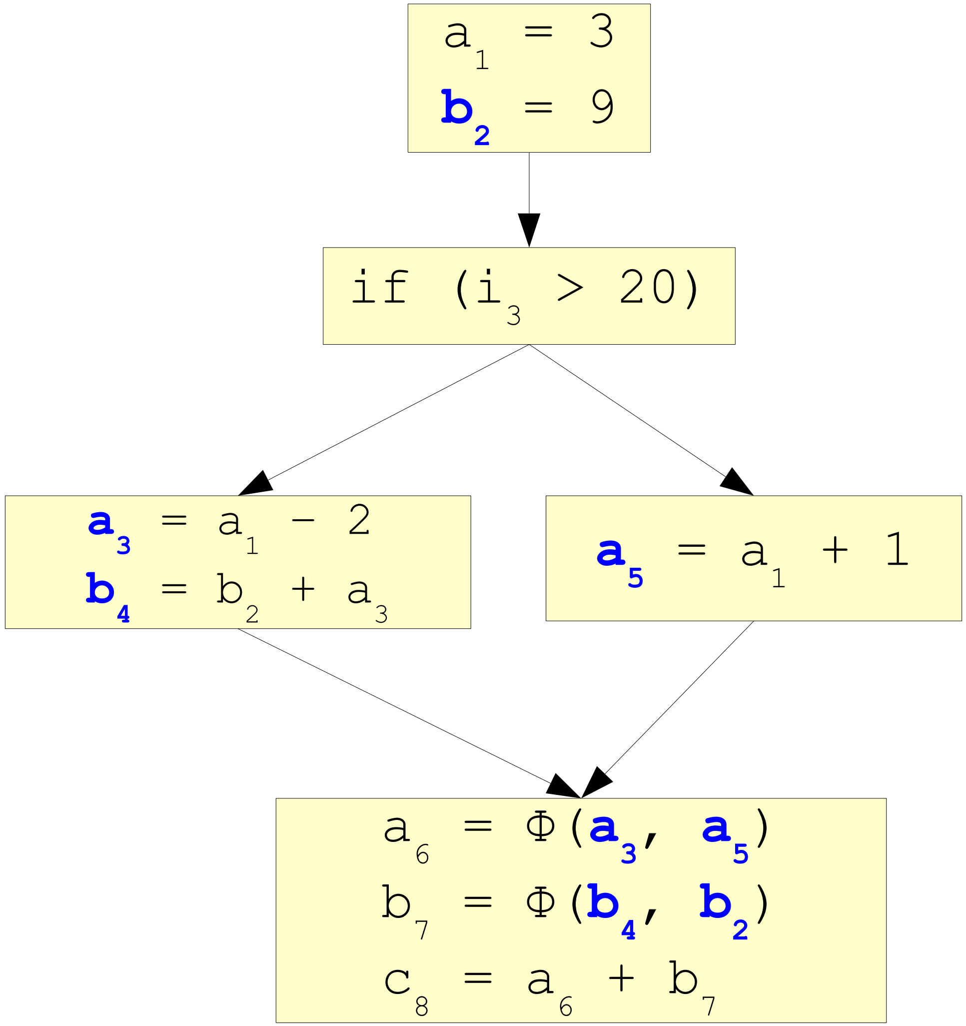
3. Internal architecture

- Compiler pipeline
- Intermediate representations
- Control/data structures
- Alias analysis
- SSA forms
- Code generation

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)



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

$$\texttt{if } (x_2 > 4)$$
$$z_5 = x_3 - 1$$

  - 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
- Both SSA forms can be updated incrementally
  - Name→name mappings
  - Individual symbols marked for renaming

Virtual SSA Form

- VDEF operand needed to maintain DEF-DEF links
- They also prevent code movement that would cross stores after loads
- When alias sets grow too big, static grouping heuristic reduces number of virtual operators in aliased references

```
foo (i, a, b, *p)
{
  p_2 = (i_1 > 10) ? &a : &b

  # a_4 = VDEF <a_11>
  a = 9;

  # a_5 = VDEF <a_4>
  # b_7 = VDEF <b_6>
  *p_2 = 3;

  # VUSE <a_5>
  t1_8 = a;

  t3_10 = t1_8 + 5;
  return t3_10;
}
```

Incremental SSA form

SSA forms are kept up-to-date incrementally

Manually

- As long as SSA property is maintained, passes may introduce new SSA names and PHI nodes on their own
- Often this is the quickest way

Automatically using `update_ssa`

- marking individual symbols (`mark_sym_for_renaming`)
- name→name mappings (`register_new_name_mapping`)
- Passes that invalidate SSA form must set `TODO_update_ssa`

3. Internal architecture

- Compiler pipeline
- Intermediate representations
- Control/data structures
- Alias analysis
- SSA forms
- Code generation

Code generation

- Code is generated using a rewriting system
- Target specific configuration files in

  `gcc/config/<arch>`

- Three main target-specific files
  - `<arch>.md` Code generation patterns for RTL insns
  - `<arch>.h` Definition of target capabilities (register classes, calling conventions, type sizes, etc)
  - `<arch>.c` Support functions for code generation, predicates and target variants

Code generation

- Two main types of rewriting schemes supported
  - Simple mappings from RTL to assembly (`define_insn`)
  - Complex mappings from RTL to RTL (`define_expand`)
- `define_insn` patterns have five elements

```
(define_insn "addsi3"

  [(set (match_operand:SI 0 "integer_register_operand" "=d")
        (plus:SI (match_operand:SI 1 "integer_register_operand" "%d")
                 (match_operand:SI 2 "gpr_or_int12_operand" "dNOPQ")))]

  ""

  "add%I2 %1,%2,%0"

  [(set_attr "length" "4")
   (set_attr "type" "int")])
```

Pattern name (optional)

RTL template

Additional matching predicate

Assembly output template

Attributes associated with this pattern (optional)

Code generation

`define_insn "addsi3"`

- Named patterns
  - Used to generate RTL
  - Some standard names are used by code generator
  - Some missing standard names are replaced with library calls (e.g., `divsi3` for targets with no division operation)
  - Some pattern names are mandatory (e.g. move operations)
- Unnamed (anonymous) patterns do not generate RTL, but can be used in insn combination

Code generation

```
[(set (match_operand:SI 0 "integer_register_operand" "=d,=d")
      (plus:SI (match_operand:SI 1 "integer_register_operand" "%d,m")
               (match_operand:SI 2 "gpr_or_int12_operand" "dNOPQ,m")))]
```

Matching uses
- Machine mode (`SI`, `DI`, `HI`, `SF`, etc)
- Predicate (a C function)
- Both operands and operators can be matched

Constraints provide second level of matching
Select best operand among the set of allowed operands
Letters describe kinds of operands
Multiple alternatives separated by commas

Code generation

```
"add%I2 %1,%2,%0"
```

- Code is generated by emitting strings of target assembly
- Operands in the insn pattern are replaced in the %n placeholders
- If constraints list multiple alternatives, multiple output strings must be used
- Output may be a simple string or a C function that builds the output string

Pattern expansion

- Some standard patterns cannot be used to produce final target code. Two ways to handle it
  - Do nothing. Some patterns can be expanded to libcalls
  - Use `define_expand` to generate matchable RTL
- Four elements
  - The name of a standard insn
  - Vector of RTL expressions to generate for this insn
  - A C expression acting as predicate to express availability of this instruction
  - A C expression used to generate operands or more RTL

Pattern expansion

```
(define_expand "ashlsi3"
  [(set (match_operand:SI 0 "register_operand" "")
        (ashift:SI
           (match_operand:SI 1 "register_operand" "")
           (match_operand:SI 2 "nonmemory_operand" ""))))]
  ""
  "{
     if (GET_CODE (operands[2]) != CONST_INT
         || (unsigned) INTVAL (operands[2]) > 3)
       FAIL;
   }")
```

- Generate a left shift only when the shift count is [0...3]
- **FAIL** indicates that expansion did not succeed and a different expansion should be tried (e.g., a library call)
- **DONE** is used to prevent emitting the RTL pattern. C fragment responsible for emitting all insns.

4. Passes

- Adding a new pass
- Debugging dumps
- Case study: VRP

Adding a new pass

- To implement a new pass
  - Add a new file to `trunk/gcc` or edit an existing pass
  - Add a new target rule in `Makefile.in`
  - If a flag is required to trigger the pass, add it to `common.opt`
  - Create an instance of `struct tree_opt_pass`
  - Declare it in `tree-pass.h`
  - Sequence it in `init_optimization_passes`
  - Add a gate function to read the new flag
  - Document pass in `trunk/gcc/doc/invoke.texi`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (vo

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_req
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_sta
  unsigned int todo_flags_fin
  char letter;
};
```

Extension for dump file is
`.<static_pass_number>[itr].<name>`

e.g., `prog.c.158r.greg`

`i` for IPA passes
`t` for GIMPLE (tree) passes
`r` for RTL passes

`static_pass_number` is automatically assigned by pass manager

Letter used by the -d switch to enable a specific RTL dump (backward compatibility)

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

If function `gate()` returns true, then the pass entry point function `execute()` is called

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_r
  unsigned int properties_p
  unsigned int properties_d
  unsigned int todo_flags_s
  unsigned int todo_flags_f
  char letter;
};
```

Passes may be organized hierarchically
`sub` points to first child pass
`next` points to sibling class
Passes are chained together with
`NEXT_PASS` in `init_optimization_passes`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

Each pass can define its own separate timer

Timers are started/stopped automatically by pass manager

Timer handles (`timevars`) are defined in `timevar.def`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

Properties required, provided and destroyed are defined in `tree-pass.h`

Common properties
`PROP_cfg`
`PROP_ssa`
`PROP_alias`
`PROP_gimple_lcf`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

Cleanup or bookkeeping actions that the pass manager should do before/after the pass

Defined in `tree-pass.h`

Common actions
`TODO_dump_func`
`TODO_verify_ssa`
`TODO_cleanup_cfg`
`TODO_update_ssa`

Available features

- 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).

Available features

- 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.

4. Passes

- Adding a new pass
- Debugging
- Case study: VRP

Debugging dumps

Most passes understand the `-fdump` switches

```
-fdump-<ir>-<pass>[-<flag1>[-<flag2>]...]
```

```
ipa
tree
rtl
```

- `details, stats, blocks,` ...
- `all` enables all flags
- Possible values taken from array `dump_options`

- `inline, dce, alias, combine` ...
- `all` to enable all dumps
- Possible values taken from `name` field in `struct tree_opt_pass`

Debugging dumps

- Adding dumps to your pass
  - Specify a name for the dump in `struct tree_opt_pass`
  - To request a dump at the end of the pass add `TODO_dump_func` in `todo_flags_finish` field
- To emit debugging information during the pass
  - Variable `dump_file` is set if dumps are enabled
  - Variable `dump_flags` is a bitmask that specifies what flags were selected
  - Some common useful flags: `TDF_DETAILS`, `TDF_STATS`

Using gdb

- Never debug the gcc binary, that is only the driver
- The real compiler is one of cc1, jc1, f951, ...

```
$ <bld>/bin/gcc -O2 -v -save-temps -c a.c
  Using built-in specs.
  Target: x86_64-unknown-linux-gnu
  Configured with: [ ... ]
  [ ... ]
  End of search list.
  <path>/cc1 -fpreprocessed a.i -quiet -dumpbase a.c
  -mtune=generic -auxbase a -O2 -version -o a.s
```

```
$ gdb --args <path>/cc1 -fpreprocessed a.i -quiet -dumpbase
  a.c -mtune=generic -auxbase a -O2 -version -o a.s
```

Using gdb

- The build directory contains a `.gdbinit` file with many useful wrappers around debugging functions
- When debugging a bootstrapped compiler, try to use the stage 1 compiler
- The stage 2 and stage 3 compilers are built with optimizations enabled (may confuse debugging)
- To recreate testsuite failures, cut and paste command line from

```
<bld>/gcc/testsuite/{gcc,gfortran,g++,java}/*.log
```

4. Passes

- Adding a new pass
- Debugging
- Case study: VRP

Value Range Propagation

- Based on Patterson's range propagation for jump prediction [PLDI'95]
  - No branch probabilities (only taken/not-taken)
  - Only a single range per SSA name.

```
for (int i = 0; i < a->len; i++)
  {
    if (i < 0 || i >= a->len)
      throw 5;
    call (a->data[i]);
  }
```

- Conditional inside the loop is unnecessary.

Value Range Propagation

Two main phases

Range assertions

Conditional jumps provide info on value ranges

```
if (a_3 > 10)
  a_4 = ASSERT_EXPR <a_3, a_3 > 10>
  ...
else
  a_5 = ASSERT_EXPR <a_4, a_4 <= 10>
```

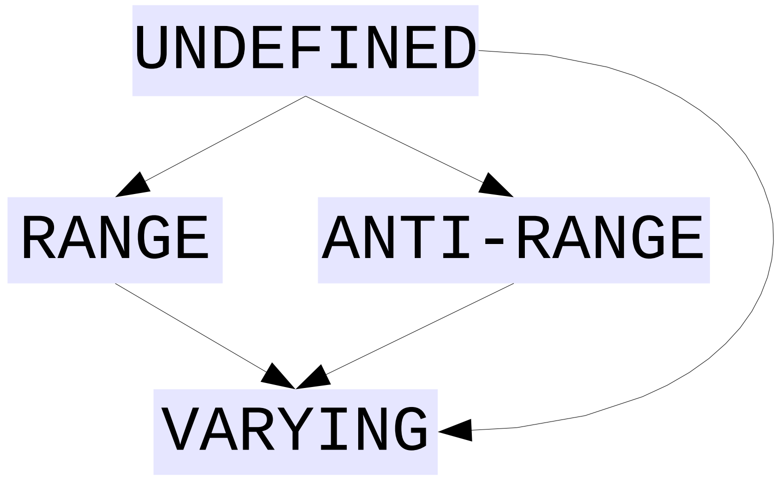
Now we can associate a range value to a_4 and a_5.

Range propagation

Generic propagation engine used to propagate value ranges from ASSERT_EXPR

Value Range Propagation

- Two range representations
  - Range [MIN, MAX] → MIN <= N <= MAX
  - Anti-range ~[MIN, MAX] → N < MIN or N > MAX
- Lattice has 4 states



- No upward transitions

Propagation engine

- Generalization of propagation code in SSA-CCP
- Simulates execution of statements that produce "*interesting*" values
- Flow of control and data are simulated with work lists.
  - CFG work list → control flow edges.
  - SSA work list → def-use edges.
- Engine calls-back into VRP at every statement and PHI node

Propagation engine

Usage

```
ssa_propagate (visit_stmt, visit_phi)
```

Returns 3 possible values for statement **S**

- `SSA_PROP_INTERESTING`
  - **S** produces an interesting value
  - If **S** is not a jump, `visit_stmt` returns name N_i holding the value
  - Def-use edges out of N_i are added to SSA work list
  - If **S** is jump, `visit_stmt` returns edge that will always be taken
- `SSA_PROP_NOT_INTERESTING`
  - No edges added, **S** may be visited again
- `SSA_PROP_VARYING`
  - Edges added, **S** will *not* be visited again

Propagation engine

- `visit_phi` has similar semantics as `visit_stmt`
  - PHI nodes are merging points, so they need to “intersect” all the incoming arguments
- Simulation terminates when both SSA and CFG work lists are drained
- Values should be kept in an array indexed by SSA version number
- After propagation, call `substitute_and_fold` to do final replacement in IL

Implementing VRP

Pass declaration in gcc/tree-vrp.c

```
struct tree_opt_pass pass_vrp =
  {
    "vrp",
    gate_vrp,
    execute_vrp,
    NULL,
    NULL,
    0,
    TV_TREE_VRP,
    PROP_ssa | PROP_alias,
    0,
    0,
    0,
    TODO_cleanup_cfg | TODO_ggc_collect
      | TODO_verify_ssa | TODO_dump_func
      | TODO_update_ssa,
    0
  };
```

Extension for dump file

Gating function

Pass entry point

Timevar handle for timings

Properties required by the pass

Things to do after VRP and before calling the next pass

Implementing VRP

Add -ftree-vrp to common.opt

```
ftree-vrp
Common Report Var(flag_tree_vrp) Init(0) Optimization
Perform Value Range Propagation on trees
```

`Common`	This flag is available for all languages
`Report`	`-fverbose-asm` should print the value of this flag
`Var`	Global variable holding the value of this flag
`Init`	Initial (default) value for this flag
`Optimization`	This flag belongs to the optimization family of flags

Implementing VRP

Add gating function

```
static bool
gate_vrp (void)
{
   return flag_tree_vrp != 0;
}
```

Add new entry in Makefile.in

- Add `tree-vrp.o` to `OBJS-common` variable
- Add rule for `tree-vrp.o` listing **all** dependencies

Implementing VRP

Add entry point function

```
static unsigned int
execute_vrp (void)
{
   insert_range_assertions ();
   ...
   ssa_propagate (vrp_visit_stmt,vrp_visit_phi_node);
   ...
   remove_range_assertions ();

   return 0;
}
```

If the pass needs to add TODO items, it should return them here

Implementing VRP

Schedule VRP in init_optimization_passes

```
init_optimization_passes (void)
{
   ...
   NEXT_PASS (pass_merge_phi);
   NEXT_PASS (pass_vrp);
   ...
   NEXT_PASS (pass_reassoc);
   NEXT_PASS (pass_vrp);
   ...
}
```

Why here?
(good question)

Conclusions

- GCC is large and seemingly scary, but
  - Active and open development community (eager to help)
  - Internal architecture has been recently overhauled
  - Modularization effort still continues
- This was just a flavour of all the available functionality
  - Extensive documentation at http://gcc.gnu.org/onlinedocs/
  - IRC (irc://irc.oftc.net/#gcc) and Wiki (http://gcc.gnu.org/wiki/) highly recommended

Current and Future Projects

Plug-in Support

- Extensibility mechanism to allow 3rd party tools
- Wrap some internal APIs for external use
- Allow loading of external shared modules
  - Loaded module becomes another pass
  - Compiler flag determines location
- Versioning scheme prevents mismatching
- Useful for
  - Static analysis
  - Experimenting with new transformations

Scheduling

- Several concurrent efforts targetting 4.3 and 4.4
  - Schedule over larger regions for increased parallelism
  - Most target IA64, but benefit all architectures
- Enhanced selective scheduling
- Treegion scheduling
- Superblock scheduling
- Improvements to swing modulo scheduling

Register Allocation

- Several efforts over the years
- Complex problem
  - Many different targets to handle
  - Interactions with reload and scheduling
- YARA (Yet Another Register Allocator)
  - Experimented with several algorithms
- IRA (Integrated Register Allocator)
  - Priority coloring, Chaitin-Briggs and region based
  - Expected in 4.4
  - Currently works on x86, x86-64, ppc, IA64, sparc, s390

Register pressure reduction

- SSA may cause excessive register pressure
  - Pathological cases → ~800 live registers
  - RA battle lost before it begins
- Short term project to cope with RA deficiencies
- Implement register pressure reduction in GIMPLE before going to RTL
  - Pre-spilling combined with live range splitting
  - Load rematerialization
  - Tie RTL generation into out-of-ssa to allow better instruction selection for spills and rematerialization

Dynamic compilation

- Delay compilation until runtime (JIT)
  - Emit bytecodes
  - Implement virtual machine with optimizing transformations
- Leverage on existing infrastructure (LLVM, LTO)
- Not appropriate for every case
- Challenges
  - Still active research
  - Different models/costs for static and dynamic compilers

Incremental Compilation

- Speed up edit-compile-debug cycle
- Speeds up ordinary compiles by compiling a given header file “once”
- Incremental changes fed to compiler daemon
- Incremental linking as well
- Side effects
  - Refactoring
  - Cross-referencing
  - Compile-while-you-type (e.g., Eclipse)

Dynamic Optimization Pipeline

- Phase ordering not optimal for every case
- Current static ordering difficult to change
- Allow external re-ordering
  - Ultimate control
  - Allow experimenting with different orderings
  - Define `-On` based on common orderings
- Problems
  - Probability of finding bugs increases
  - Enormous search space