GCC Internals
Internal Representations

Diego Novillo
dnovillo@google.com

November 2007

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)

GIMPLE statements

- GIMPLE statements are instances of type `tree`

For now

- 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

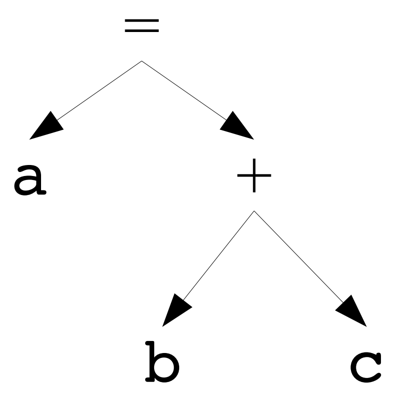
```
use_operand_p use;
ssa_op_iter i;
FOR_EACH_SSA_USE_OPERAND (use, stmt, i, SSA_OP_ALL_USES)
  {
    tree op = USE_FROM_PTR (use);
    print_generic_expr (stderr, op, 0);
  }
```

- 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

GIMPLE tuples

- More compact data structure than `tree`
- Statements no longer an expression tree

```
a = b + c
```



```
  tree_code_size (=) =  64
+ tree_code_size (+) =  64
+ sizeof (annotation)=  96
                     = 224
```

code	=
subcode	+
next	
prev	
bb	
...	...
op0	a
op1	b
op2	c

```
gimple_size (=) = 128
```

64 bit host

GIMPLE tuples

- Fewer pointers
  - Less scattered allocation
  - Simplified pickling for streaming
  - Potentially improved cache behaviour
- Currently only statements are converted
- Symbols and memory expressions are still represented with `tree`
- Expect modest overall memory savings (5% to 15%)
- Bigger memory consumption: declarations, types, debug info

GIMPLE tuples

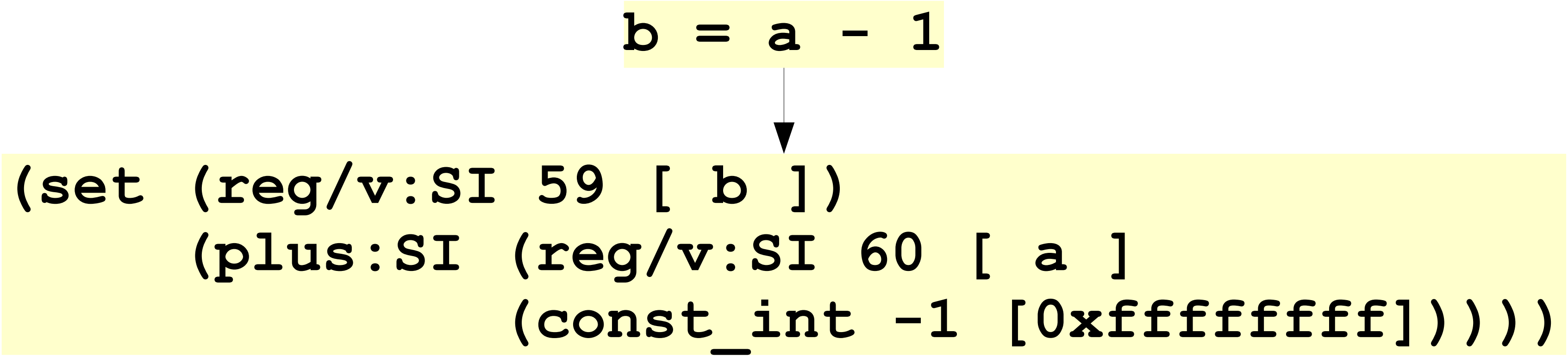
- Challenges
  - Pervasive use of `tree` data structure
  - New APIs are needed
  - RTL expansion tuned to fat expression trees (codegen differences)
- Status
  - Basic lowering, CFG and cgraph working
  - RTL expansion in progress
  - All analysis and optimization passes need to be converted

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)

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 information

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`