GCC Translation Sequence and Gimple IR

Uday Khedker

GCC Resource Center,
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

January 2010
Outline

- GCC Translation Sequence
- An External View of Gimple
- An Internal View of Gimple
- Adding a Pass to GCC
- Working with Gimple API
Part 1

GCC Translation Sequence
A total of 196 unique pass names initialized in `${SOURCE}/gcc/passes.c`

- Some passes are called multiple times in different contexts
  - Conditional constant propagation and dead code elimination are called thrice
- Some passes are only demo passes (e.g., data dependence analysis)
- Some passes have many variations (e.g., special cases for loops)
  - Common subexpression elimination, dead code elimination

The pass sequence can be divided broadly in two parts

- Passes on Gimple
- Passes on RTL

Some passes are organizational passes to group related passes
Basic Transformations in GCC

Target Independent

Parse → Gimplify → Tree SSA Optimize → Generate RTL → Optimize RTL → Generate ASM

Target Dependent

Gimple → RTL

RTL → ASM

Jan 2010

Uday Khedker, IIT Bombay
Basic Transformations in GCC

Target Independent

Parse → Gimplify → Tree SSA Optimize → Generate RTL → Optimize RTL → Generate ASM

Gimple Passes

Gimple → RTL

RTL Passes

RTL → ASM
# Passes On Gimple

<table>
<thead>
<tr>
<th>Pass Group</th>
<th>Examples</th>
<th>Number of passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowering</td>
<td>Gimple IR, CFG Construction</td>
<td>12</td>
</tr>
<tr>
<td>Interprocedural Optimizations</td>
<td>Conditional Constant Propagation, Inlining, SSA Construction</td>
<td>36</td>
</tr>
<tr>
<td>Intraprocedural Optimizations</td>
<td>Constant Propagation, Dead Code Elimination, PRE</td>
<td>40</td>
</tr>
<tr>
<td>Loop Optimizations</td>
<td>Vectorization, Parallelization</td>
<td>24</td>
</tr>
<tr>
<td>Remaining Intraprocedural Optimizations</td>
<td>Value Range Propagation, Rename SSA</td>
<td>23</td>
</tr>
<tr>
<td>Generating RTL</td>
<td></td>
<td>01</td>
</tr>
</tbody>
</table>

Total number of passes on Gimple: 136
# Passes On RTL

<table>
<thead>
<tr>
<th>Pass Group</th>
<th>Examples</th>
<th>Number of passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intraprocedural Optimizations</td>
<td>CSE, Jump Optimization</td>
<td>15</td>
</tr>
<tr>
<td>Loop Optimizations</td>
<td>Loop Invariant Movement, Peeling, Unswitching</td>
<td>7</td>
</tr>
<tr>
<td>Machine Dependent Optimizations</td>
<td>Register Allocation, Instruction Scheduling, Peephole Optimizations</td>
<td>59</td>
</tr>
<tr>
<td>Assembly Emission and Finishing</td>
<td></td>
<td>03</td>
</tr>
<tr>
<td>Total number of passes on RTL</td>
<td></td>
<td>84</td>
</tr>
</tbody>
</table>

Jan 2010 Uday Khedker, IIT Bombay
Finding Out List of Optimizations

Along with the associated flags

- A complete list of optimizations with a brief description
  
  `gcc -c --help=optimizers`

- Optimizations enabled at level 2 (other levels are 0, 1, and 3)
  
  `gcc -c -O2 --help=optimizers -Q`
Dumps Produced by GCC

To see the output after each pass use the option

```
-fdump-<ir>-<pass>
```

where `<ir>` is

- `tree`
  - `<pass>` could be: `gimple`, `cfg` etc.
  - Use `-all` to see all dumps

- `rtl`
  - `<pass>` could be: `expand`, `greg`, `vreg` etc.
  - Use `-all` to see all dumps
  - We can also use `-da` option

Example:

```
gcc -fdump-tree-all -fdump-rtl-all test.c
```
Example Program

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
    {
        a = a+b+c;
    }
}
```

Command used to compile the program

```
gcc -fdump-tree-all -da test.c
```
GCC 4.4.2 Dumps for Our Example Program

test.c.001t.tu
test.c.003t.original
test.c.004t.gimple
test.c.006t.vcg
test.c.007t.useless
test.c.010t.lower
test.c.011t.ehopt
test.c.012t.eh
test.c.013t.cfg
test.c.014t.cplxlower0
test.c.015t.veclower
test.c.021t.cleanup_cfg1
test.c.051t.apply_inline
test.c.131r.expand
test.c.132r.sibling
test.c.134r.initvals
test.c.135r.unshare
test.c.136r.vregs
test.c.137r.into_cfglayout
test.c.138r.jump
test.c.157r.regclass
test.c.160r.outof_cfglayout
test.c.166r.split1
test.c.168r.dfininit
test.c.169r.mode-sw
test.c.171r.asmcons
test.c.174r.subregs_of_mode_init
test.c.175r.lreg
test.c.176r.greg
test.c.177r.subregs_of_mode_finish
test.c.180r.split2
test.c.182r.pro_and_epilogue
test.c.196r.stack
test.c.197r.alignments
test.c.200r.mach
test.c.201r.barriers
test.c.204r.eh-ranges
test.c.205r.shorten
test.c.206r.dfinish
test.s

Jan 2010

Uday Khedker, IIT Bombay
# Examples of Gimple and RTL Dumps

<table>
<thead>
<tr>
<th>Gimple (or Tree-SSA) dumps</th>
<th>RTL dumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dump file number ending in <code>t</code></td>
<td>Dump file number ending in <code>r</code></td>
</tr>
<tr>
<td>test.c.003t.original</td>
<td>test.c.166r.split1</td>
</tr>
<tr>
<td>test.c.004t.gimple</td>
<td>test.c.168r.dfinit</td>
</tr>
<tr>
<td>test.c.006t.vcg</td>
<td>test.c.169r.mode-sw</td>
</tr>
<tr>
<td>test.c.007t.useless</td>
<td>test.c.171r.asmcons</td>
</tr>
<tr>
<td>test.c.010t.lower</td>
<td>test.c.175r.lreg</td>
</tr>
<tr>
<td>test.c.013t.cfg</td>
<td>test.c.176r.greg</td>
</tr>
</tbody>
</table>
Dumping Detailed Information of a Pass

- For Gimple passes (dump file numbers ending in t)
  \[ \text{gcc } -f\text{dump-tree-}<\text{name}>-\text{all} \]

- For RTL passes (dump file numbers ending in r)
  \[ \text{gcc } -f\text{dump-rtl-}<\text{name}>-\text{all} \]

- In each case, \(<\text{name}>\) is the dump file name extension of the pass
Selected Dumps for Our Example Program

test.c.001t.tu

test.c.003t.original

**test.c.004t.gimple**
test.c.006t.vcg
test.c.007t.useless

**test.c.010t.lower**
test.c.011t.ehopt
test.c.012t.eh

**test.c.013t.cfg**
test.c.014t.cplxlower0
test.c.015t.veclower
test.c.021t.cleanup_cfg1
test.c.051t.apply_inline

test.c.131r.expand
test.c.132r.sibling
test.c.134r.initvals
test.c.135r.unshare

test.c.136r.vregs
test.c.137r.intoCfgLayout

**test.c.157r.regclass**
test.c.160r.outofCfgLayout
test.c.166r.split1
test.c.168r.dfinit
test.c.169r.mode-sw

test.c.171r.asmcons
test.c.174r.subregs_of_mode_init

test.c.175r.lreg
test.c.176r.greg

test.c.177r.subregs_of_mode_finish

test.c.180r.split2
test.c.182r.pro_and_epilogue

test.c.196r.stack
test.c.197r.alignments

test.c.200r.mach
test.c.201r.barriers

test.c.204r.eh-ranges

test.c.205r.shorten

test.c.206r.dfinish

test.s
Part 2

An External View of Gimple
Important Phases of GCC

1. C Source Code
   - Parser
   - AST
   - Gimplifier
   - Gimple
   - Linearizer
   - Lower
   - CFG Generator
   - CFG
   - RTL Generator
   - RTL expand

2. GRC: An External View of Gimple
   - local reg allocator
   - lregs
   - global reg allocator
   - Gregs
   - pro_epilogue generation
   - prologue-epilogue
   - Pattern Matcher
   - ASM Program

Jan 2010

Uday Khedker, IIT Bombay
Gimplifier

- Three-address language independent representation derived from Generic
  - Computation represented as a sequence of basic operations
  - Temporaries introduced to hold intermediate values
- Control construct are explicated into conditional jumps
Motivation behind Gimple

• Previously, the only common IR was RTL (Register Transfer Language)

• Drawbacks of RTL for performing high-level optimizations:
  ▶ RTL is a low-level IR, works well for optimizations close to machine (e.g., register allocation)
  ▶ Some high level information is difficult to extract from RTL (e.g. array references, data types etc.)
  ▶ Optimizations involving such higher level information are difficult to do using RTL.
  ▶ Introduces stack too soon, even if later optimizations dont demand it.

Notice
Inlining at tree level could partially address the the last limitation of RTL.
Why not ASTs for optimization?

- ASTs contain detailed function information but are not suitable for optimization because
  - Lack of a common representation
    - No single AST shared by all front-ends
    - So each language would have to have a different implementation of the same optimizations
    - Difficult to maintain and upgrade so many optimization frameworks
  - Structural Complexity
    - Lots of complexity due to the syntactic constructs of each language
Need for a new IR

- In the past, compiler would only build up trees for a single statement, and then lower them to RTL before moving on to the next statement.
- For higher level optimizations, entire function needs to be represented in trees in a language-independent way.
- Result of this effort - Generic and Gimple
What is Generic?

What?

- Language independent IR for a complete function in the form of trees
- Obtained by removing language specific constructs from ASTs
- All tree codes defined in $(SOURCE)/gcc/tree.def$

Why?

- Each language frontend can have its own AST
- Once parsing is complete they must emit Generic
What is Gimple?

- Gimple is influenced by SIMPLE IR of McCat compiler
- But Gimple is not same as SIMPLE (Gimple supports GOTO)
- It is a simplified subset of Generic
  - 3 address representation
  - Control flow lowering
  - Cleanups and simplification, restricted grammar
- Benefit: Optimizations become easier
Gimple Phase Sequence in c$c$1 and GCC-4.3.1

c_genericize()
c_gimplify.c
gimplify_function_tree()
gimplify.c
gimplify_body() gimplify.c
gimplify_stmt() gimplify.c
gimplify_expr() gimplify.c
lang_hooks.callgraph.expand_function()
tree_rest_of_compilation() tree-optimize.c
tree_registercfg_hooks() cfghooks.c
execute_pass_list() passes.c

/* TO: Gimple Optimisations passes */

... NEXT_PASS(pass_lower_cf)

May have changed in GCC-4.4.2
Gimple Goals

The Goals of Gimple are

- Lower control flow
  Program = sequenced statements + unrestricted jump
- Simplify expressions
  Typically: two operand assignments!
- Simplify scope
  move local scope to block begin, including temporaries

Notice
Lowered control flow $\rightarrow$ nearer to register machines + Easier SSA!
Dump file: test.c.004t.gimple

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
    {
        D.1199 = a + b;
        a = D.1199 + c;
    }
    else
    {
    }
    a = a+b+c;
}
```
Gimple: Translation of Higher Level Control Constructs

Dump file: test.c.004t.gimple

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
    {
        a = a+b+c;
    }
    goto <D.1197>;
    <D.1196>:
    a = a + 1;
    <D.1197>:
    if (a <= 7)
    {
        goto <D.1196>;
    }
    else
    {
        goto <D.1198>;
    }
    <D.1198>;
```
Dump file: test.c.004t.gimple

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
    {
        a = a+b+c;
    }
}
```

```c
goto <D.1197>;
<D.1196>:
    a = a + 1;
    <D.1197>:
    if (a <= 7)
    {
        goto <D.1196>;
    }
    else
    {
        goto <D.1198>;
    }
    <D.1198>:
```
Gimple: Translation of Higher Level Control Constructs

Dump file: test.c.004t.gimple

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
    {
        a = a+b+c;
    }
    goto <D.1197>;
    <D.1197>:;
    a = a + 1;
    <D.1196>:;
    if (a <= 7)
    {
        goto <D.1196>;
    }
    else
    {
        goto <D.1198>;
    }
    <D.1198>:;
    <D.1198>:;
```
Dump file: test.c.004t.gimple

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
    {
        a = a+b+c;
    }
}
```

goto <D.1197>;
<D.1196>:;
```
a = a + 1;
<D.1197>:;
if (a <= 7)
{
    goto <D.1196>;
} else
{
    goto <D.1198>;
}
```

Jan 2010 Uday Khedker, IIT Bombay
Important Phases of GCC

C Source Code

- Parser
- AST
- Gimplifier
- Gimple
- Linearizer
- Lower
- CFG Generator
- CFG
- RTL Generator
- RTL expand

local reg allocator
- lregs
- global reg allocator
- Gregs
- pro_epilogue_generation
- prologue-epilogue
- Pattern Matcher
- ASM Program

Jan 2010 Uday Khedker, IIT Bombay
Dump file: test.c.010t.lower

```c
if (a <= 12)
{
    D.1199 = a + b;
    a = D.1199 + c;
}
```

```c
if (a <= 12) goto <D.1200>;
else goto <D.1201>;
```

```c
<D.1200>:
    D.1199 = a + b;
    a = D.1199 + c;
    <D.1201>:
    return;
```
Dump file: test.c.010t.lower

if (a <= 12)
{
    D.1199 = a + b;
    a = D.1199 + c;
}

if (a <= 12) goto <D.1200>;
else goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
return;

if–then translated in terms of conditional and unconditional gotos
Lowering Gimple

Dump file: test.c.010t.lower

```c
if (a <= 12)
{
    D.1199 = a + b;
    a = D.1199 + c;
}
else goto <D.1201>;
return;
```

if-then translated in terms of conditional and unconditional gotos

```c
if (a <= 12) goto <D.1200>;
else goto <D.1201>;
return;
```
Important Phases of GCC

C Source Code

- Parser
- AST
- Gimplifier
- Gimple
- Linearizer
- Lower
- CFG Generator
- CFG
- RTL Generator
- RTL expand

local reg allocator
- Iregs
- global reg allocator
- Gregs
- pro_epilogue generation
- prologue-epilogue
- Pattern Matcher
- ASM Program

Jan 2010
Constructing the Control Flow Graph

Dump file: test.c.013t.cfg

```c
if (a <= 12) goto <D.1200>;
else goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
return;
```

```c
# BLOCK 5
# PRED: 4 (false)
if (a <= 12)
    goto <bb 6>;
else
    goto <bb 7>;
# SUCC: 6 (true) 7 (false)
# BLOCK 6
# PRED: 5 (true)
D.1199 = a + b;
a = D.1199 + c;
# SUCC: 7 (fallthru)
# BLOCK 7
# PRED: 5 (false) 6 (fallthru)
return;
# SUCC: EXIT
```
Constructing the Control Flow Graph

Dump file: test.c.013t.cfg

if (a <= 12) goto <D.1200>;
else goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
return;

# BLOCK 5
# PRED: 4 (false)
if (a <= 12)
    goto <bb 6>;
else
    goto <bb 7>;
# SUCC: 6 (true) 7 (false)
# BLOCK 6
# PRED: 5 (true)
D.1199 = a + b;
a = D.1199 + c;
# SUCC: 7 (fallthru)
# BLOCK 7
# PRED: 5 (false) 6 (fallthru)
return;
# SUCC: EXIT
### Constructing the Control Flow Graph

**Dump file:** test.c.013t.cfg

```c
if (a <= 12) goto <D.1200>;
else goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
return;
```

```c
# BLOCK 5
# PRED: 4 (false)
if (a <= 12)
   goto <bb 6>;
else
   goto <bb 7>;
# SUCC: 6 (true) 7 (false)
# BLOCK 6
# PRED: 5 (true)
D.1199 = a + b;
a = D.1199 + c;
# SUCC: 7 (fallthru)
# BLOCK 7
# PRED: 5 (false) 6 (fallthru)
return;
# SUCC: EXIT
```
Constructing the Control Flow Graph

Dump file: test.c.013t.cfg

```c
if (a <= 12) goto <D.1200>;
else goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
return;
```

```c
# BLOCK 5
# PRED: 4 (false)
if (a <= 12)
    goto <bb 6>;
else
    goto <bb 7>;
# SUCC: 6 (true) 7 (false)
# BLOCK 6
# PRED: 5 (true)
D.1199 = a + b;
a = D.1199 + c;
# SUCC: 7 (fallthru)
# BLOCK 7
# PRED: 5 (false) 6 (fallthru)
return;
# SUCC: EXIT
```
## Constructing the Control Flow Graph

**Dump file:** test.c.013t.cfg

<table>
<thead>
<tr>
<th>BLOCK 5</th>
<th>PRED: 4 (false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &lt;= 12) goto &lt;bb 6&gt;;</td>
<td></td>
</tr>
<tr>
<td>else goto &lt;bb 7&gt;;</td>
<td></td>
</tr>
<tr>
<td># SUCC: 6 (true) 7 (false)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLOCK 6</th>
<th>PRED: 5 (true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1199 = a + b;</td>
<td></td>
</tr>
<tr>
<td>a = D.1199 + c;</td>
<td></td>
</tr>
<tr>
<td># SUCC: 7 (fallthru)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BLOCK 7</th>
<th>PRED: 5 (false) 6 (fallthru)</th>
</tr>
</thead>
<tbody>
<tr>
<td>return;</td>
<td></td>
</tr>
<tr>
<td># SUCC: EXIT</td>
<td></td>
</tr>
</tbody>
</table>
Dump file: test.c.013t.cfg

Block 4:
\[
\text{if}(a \leq 7)
\]

Block 5:
\[
\text{if}(a \leq 12)
\]

Block 3:
\[
a = a + 1;
\]

Block 6:
\[
D.1199 = a + b;
a = D.1199 + c;
\]

Block 7:
\[
\text{return};
\]
Control Flow Graph

Dump file: test.c.013t.cfg

Block 4:
\[
\text{if}(a\leq 7)
\]
Block 5:
\[
\text{if}(a\leq 12)
\]
Block 3:
\[
a = a + 1;
\]
Block 6:
\[
D.1199= a + b;
a= D.1199 + c;
\]
Block 7:
\[
\text{return};
\]
Control Flow Graph

**Dump file:** test.c.013t.cfg

Block 4:
if(a <= 7)

Block 5:
if(a <= 12)
Block 3:
a = a + 1;

Block 6:
D.1199 = a + b;
a = D.1199 + c;

Block 7:
return;

Jan 2010 Uday Khedker, IIT Bombay
Inspect Gimple when in doubt

```c
int main(void)
{
    int x=2, y=3;
    x = y++ + ++x + ++y;
    printf("\nx = \%d\n", x);
    printf("\ny = \%d\n", y);
    return 0;
}
```
Resolving doubts by inspecting Gimple

Inspect Gimple when in doubt

```c
int main(void)
{
    int x=2,y=3;
    x= y++ + ++x + ++y ;
    printf("\nx = %d", x);
    printf("\ny = %d", y);
    return 0;
}
```

```c
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
printf (&"\nx = %d"[0], x);
printf (&"\ny = %d"[0], y);
```
Resolving doubts by inspecting Gimple

Inspect Gimple when in doubt

```c
int main(void)
{
    int x=2,y=3;
    x = y++ + ++x + ++y ;
    printf("\nx = %d\n", x);
    printf("\ny = %d\n", y);
    return 0;
}
```

```c
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
printf (&"\nx = %d"[0], x);
printf (&"\ny = %d"[0], y);
```

```
x = 10 , y =5
```

x = 10 , y =5
Decisions that have been taken

- Three-address representation is generated
- All high level control flow structures are made explicit.
- Source code divided into interconnected blocks of sequential statements.
- This is a convenient structure for later analysis.
Part 3

An Internal View of Gimple in GCC-4.3.1
High Gimple in GCC-4.3.1

Gimple is based on tree data structure.

- Gimple that is not fully lowered.
- Consists of Intermediate Language before the pass `pass_lower Cf`.
- Contains some container statements like lexical scopes and nested expressions.

- **High Gimple Instruction Set**: `GIMPLE_BIND`, `GIMPLE_CALL`, `GIMPLE_CATCH`, `GIMPLE_GOTO`, `GIMPLE_EH_FILTER`, `GIMPLE_RETURN`, `GIMPLE_SWITCH`, `GIMPLE_TRY`, `GIMPLE_ASSIGN`
Low Gimple in GCC-4.3.1

Gimple is based on *tree* data structure.

- Gimple that is fully lowered after the pass `pass_lower_cf`.
- Exposes all of the implicit jumps for control and exception expressions.

- **Low Gimple Instruction Set**: `GIMPLE_CALL`, `GIMPLE_GOTO`, `GIMPLE_RETURN`, `GIMPLE_SWITCH`, `GIMPLE_ASSIGN`

- **Lowered Instruction Set**: `GIMPLE_BIND`, `GIMPLE_CATCH`, `GIMPLE_EH_FILTER`, `GIMPLE_TRY`
Some Gimple Node types in GCC-4.3.1

Binary Operator  MAX_EXPR
Comparison       EQ_EXPR, LT_EXPR
Constants        INTEGER_CST, STRING_CST
Declaration      FUNCTION_DECL, LABELDECL, VARDECL
Expression       PLUS_EXPR, ADDR_EXPR
Reference        COMPONENT_REF, ARRAY_RANGE_REF
Statement        GIMPLE_MODIFY_STMT, RETURN_EXPR, COND_EXPR,
                 INIT_EXPR
Type             BOOLEAN_TYPE, INTEGER_TYPE
Unary            ABS_EXPR, NEGATE_EXPR

Tip:
All tree nodes (~ 152) in GCC are listed in: $(SOURCE)/gcc/tree.def
(In GCC-4.4.2, the file is $(SOURCE)/gcc/gimple.def)
Part 4

Adding a Pass to GCC
Adding a Pass on Gimple IR in GCC-4.3.1

- Step 0. Write function gccwk09_main() in file gccwk09.c.
- Step 1. Create the following data structure in file gccwk09.c.

```c
struct tree_opt_pass pass_gccwk09 =
{
    "gccwk09", /* name */
    NULL,    /* gate, for conditional entry to this pass */
    gccwk09_main,  /* execute, main entry point */
    NULL,    /* sub-passes, depending on the gate predicate */
    NULL,    /* next sub-passes, independ of the gate predicate */
    0,    /* static_pass_number , used for dump file name*/
    0,    /* tv_id */
    0,    /* properties_required, indicated by bit position */
    0,    /* properties_provided , indicated by bit position*/
    0,    /* properties_destroyed , indicated by bit position*/
    0,    /* todo_flags_start */
    0,    /* todo_flags_finish */
    0,    /* character for RTL dump */
};
```
Adding a Pass on Gimple IR in GCC-4.4.2

- Step 0. Write function gccwk09_main() in file gccwk09.c.
- Step 1. Create the following data structure in file gccwk09.c.

```c
struct gimple_opt_pass pass_gccwk09 = {
    {GIMPLE_PASS,
     "gccwk09", /* name */
     NULL, /* gate, for conditional entry to this pass */
     gccwk09_main, /* execute, main entry point */
     NULL, /* sub-passes, depending on the gate predicate */
     NULL, /* next sub-passes, independ of the gate predicate */
     0, /* static_pass_number , used for dump file name*/
     0, /* tv_id */
     0, /* properties_required, indicated by bit position */
     0, /* properties_provided , indicated by bit position*/
     0, /* properties_destroyed , indicated by bit position*/
     0, /* todo_flags_start */
     0, /* todo_flags_finish */
    }
};
```
Adding a Pass on Gimple IR

- Step 2. Add the following line to tree-pass.h
  extern struct gimple_opt_pass passgccwk09;
Adding a Pass on Gimple IR

- Step 2. Add the following line to tree-pass.h:
  ```c
  extern struct gimple_opt_pass pass_gccwk09;
  ```
- Step 3. Include the following call at an appropriate place in the function `init_optimization_passes()` in the file passes.c:
  ```c
  NEXT_PASS (pass_gccwk09);
  ```
• Step 2. Add the following line to `tree-pass.h`:
  
  ```
  extern struct gimple_opt_pass pass_gccwk09;
  ```

• Step 3. Include the following call at an appropriate place in the function `init_optimization_passes()` in the file `passes.c`:
  
  ```
  NEXT_PASS (pass_gccwk09);
  ```

• Step 4. Add the file name in the Makefile:
  
  - Either in `$SOURCE/gcc/Makefile.in`
    Reconfigure and remake
  - Or in `$BUILD/gcc/Makefile`
    Remake

---

Jan 2010

Uday Khedker, IIT Bombay
Adding a Pass on Gimple IR

- Step 2. Add the following line to tree-pass.h
  ```c
  extern struct gimple_opt_pass pass_gccwk09;
  ```

- Step 3. Include the following call at an appropriate place in the function `init_optimization_passes()` in the file `passes.c`
  ```c
  NEXT_PASS (pass_gccwk09);
  ```

- Step 4. Add the file name in the Makefile
  - Either in `$SOURCE/gcc/Makefile.in`
    Reconfigure and remake
  - Or in `$BUILD/gcc/Makefile`
    Remake

- Step 5. Build the compiler
Adding a Pass on Gimple IR

• Step 2. Add the following line to tree-pass.h
  extern struct gimple_opt_pass pass_gccwk09;

• Step 3. Include the following call at an appropriate place in the
  function init_optimization_passes() in the file passes.c
  NEXT_PASS (pass_gccwk09);

• Step 4. Add the file name in the Makefile
  ▶ Either in $SOURCE/gcc/Makefile.in
     Reconfigure and remake
  ▶ Or in $BUILD/gcc/Makefile
     Remake

• Step 5. Build the compiler

• Step 6. Debug using gdb if need arises
Part 5

Working with the Gimple API in GCC-4.3.1
Gimple Statements

- Gimple Statements are nodes of type `tree`
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through `iterators`
Gimple Statements

- Gimple Statements are nodes of type `tree`
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through `iterators`

```c
block_statement_iterator bsi;
basic_block bb;
```
Gimple Statements

- Gimple Statements are nodes of type `tree`
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through `iterators`

```c
block_statement_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
```

Basic Block Iterator
Gimple Statements

- Gimple Statements are nodes of type `tree`
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through `iterators`

```c
block_statement_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
   for ( bsi = bsi_start(bb); !bsi_end_p(bsi); bsi_next(&bsi))
```

Block Statement Iterator
Gimple Statements

1. Gimple Statements are nodes of type `tree`
2. Every basic block contains a doubly linked-list of statements
3. Processing of statements can be done through `iterators`

```c
block_statement_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
    for (bsi = bsi_start(bb); !bsi_end_p(bsi); bsi_next(&bsi))
        print_generic_stmt (stderr, bsi_stmt(bsi), 0);
```
A simple application

Counting the number of assignment statements in Gimple

```c
#include <stdio.h>
int m,q,p;
int main(void)
{
    int x,y,z,w;
    x = y + 5;
    z = x * m;
    p = m + q + w;
    return 0;
}
```

The statements in blue are the assignments corresponding to the source.
A simple application

Counting the number of assignment statements in Gimple

```c
struct tree_opt_pass pass_gccwk09 =
{
    "gccwk09",
    NULL,
    gccwk09_main,
    NULL,
    NULL,
    0,
    0,
    0,
    0,
    0,
    0,
    0
};
```
A simple application

Counting the number of assignment statements in Gimple

static unsigned int gccwk09_main(void)
{
    basic_block bb;
    block_stmt_iterator si;

    initialize_stats();

    FOR_EACH_BB (bb)
    {
        for (si=bsi_start(bb); !bsi_end_p(si); bsi_next(&si))
        {
            tree stmt = bsi_stmt(si);
            process_statement(stmt);
        }
    }
    return 0;
}
A simple application

Counting the number of assignment statements in Gimple

```c
void process_statement(tree stmt)
{
  tree lval, rval;
  switch (TREE_CODE(stmt))
  {
    case GIMPLE_MODIFY_STMT:
      lval = GIMPLE_STMT_OPERAND(stmt, 0);
      rval = GIMPLE_STMT_OPERAND(stmt, 1);
      if (TREE_CODE(lval) == VAR_DECL)
      {
        if (!DECL_ARTIFICIAL(lval))
        {
          print_generic_stmt(stderr, stmt, 0);
          numassigns++;
        }
        totalassigns++;
      }
    break;
    default :
    break;
  }
}
```
A simple application

Counting the number of assignment statements in Gimple

- Add the following in `$(SOURCE)/gcc/common.opt`:
  - `fpass_gccwk09`
  - `flag_pass_gccwk09`
- Enable pass named `pass_gccwk09`

Compile using `./gcc -fdump-tree-all -fpass_gccwk09 test.c`