Workshop on Essential Abstractions in GCC

Manipulating GIMPLE IR

GCC Resource Center

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

29 June 2013

Outline

- An Overview of GIMPLE
- Using GIMPLE API in GCC-4.7.2
- Adding a GIMPLE Pass to GCC

Part 1

An Overview of GIMPLE

GIMPLE: A Recap

- Language independent three address code representation
  - Computation represented as a sequence of basic operations
  - Temporaries introduced to hold intermediate values
- Control construct explicated into conditional and unconditional jumps

Motivation Behind GIMPLE

- Previously, the only common IR was RTL (Register Transfer Language)
- Drawbacks of RTL for performing high-level optimizations
  - Low-level IR, more suitable for machine dependent optimizations (e.g., peephole optimization)
  - High level information is difficult to extract from RTL (e.g. array references, data types etc.)
  - Introduces stack too soon, even if later optimizations do not require it

Why Not Abstract Syntax Trees for Optimization?

- ASTs contain detailed function information but are not suitable for optimization because
  - Lack of a common representation across languages
    - 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
    - Hierarchical structure and not linear structure
      Control flow explication is required

Need for a High Level IR

- Earlier versions of GCC would 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 Goals

The Goals of GIMPLE are

- Lower control flow
  Sequenced statements + conditional and unconditional jumps
- Simplify expressions
  Typically one operator and at most two operands
- Simplify scope
  Move local scope to block begin, including temporaries

Tuple Based GIMPLE Representation

- Earlier implementation of GIMPLE used trees as internal data structure
- Tree data structure was much more general than was required for three address statements
- Now a three address statement is implemented as a tuple
- These tuples contain the following information
  - Type of the statement
  - Result
  - Operator
  - Operands

  The result and operands are still represented using trees

Observing Internal Form of GIMPLE

`test.c.004t.gimple` with compilation option `-fdump-tree-all`

```
x = 10;
y = 5;
D.1954 = x * y;
a.0 = a;
x = D.1954 + a.0;
a.1 = a;
D.1957 = a.1 * x;
y = y - D.1957;
```

`test.c.004t.gimple` with compilation option `-fdump-tree-all-raw`

```
gimple_assign <integer_cst, x, 10, NULL>
gimple_assign <integer_cst, y, 5, NULL>
gimple_assign <mult_expr, D.1954, x, y>
gimple_assign <var_decl, a.0, a, NULL>
gimple_assign <plus_expr, x, D.1954, a.0>
gimple_assign <var_decl, a.1, a, NULL>
gimple_assign <mult_expr, D.1957, a.1, x>
gimple_assign <minus_expr, y, y, D.1957>
```

Observing Internal Form of GIMPLE

`test.c.004t.gimple` with compilation option `-fdump-tree-all`

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

`test.c.004t.gimple` with compilation option `-fdump-tree-all-raw`

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```

Observing Internal Form of GIMPLE

`test.c.004t.gimple` with compilation option `-fdump-tree-all`

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

`test.c.004t.gimple` with compilation option `-fdump-tree-all-raw`

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```

Observing Internal Form of GIMPLE

test.c.004t.gimple with compilation option -fdump-tree-all

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

test.c.004t.gimple with compilation option -fdump-tree-all-raw

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```

Observing Internal Form of GIMPLE

test.c.004t.gimple with compilation option -fdump-tree-all

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

test.c.004t.gimple with compilation option -fdump-tree-all-raw

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```

Part 2

Manipulating GIMPLE

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{  %
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); %
                                     gsi_next (&gsi))
         find_pointer_assignmentsgsi_stmt (gsi));
}
```

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{  %
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); %
                                         gsi_next (&gsi))
         find_pointer_assignmentsgsi_stmt (gsi));
}
```

Basic block iterator

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{  %
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); %
                                     gsi_next (&gsi))
         find_pointer_assignmentsgsi_stmt (gsi));
}
```

GIMPLE statement iterator

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{  %
    for (gsi =gsi_start_bb (bb); !gsi_end_p (gsi); %
                                     gsi_next (&gsi))
         find_pointer_assignmentsgsi_stmt (gsi));
}
```

Get the first statement of bb

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{  %
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); %
                                     gsi_next (&gsi))
         find_pointer_assignmentsgsi_stmt (gsi));
}
```

True if end reached

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{  %
    for (gsi =gsi_start_bb (bb); !gsi_end_p (gsi); %
                                     gsi_next (&gsi))
         find_pointer_assignmentsgsi_stmt (gsi));
}
```

Advance iterator to the next GIMPLE stmt

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{  %
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); %
                                     gsi_next (&gsi))
         find_pointer_assignmentsgsi_stmt (gsi));
}
```

Return the current statement

Printing Successors of a Basic Block

```
edge e;
edge_iterator ei;
basic_block bb;

FOR_EACH_BB_FN (bb, cfun)
{
   fprintf(dump_file, "\n Successor(s) of basic block bb%d: ",
                        bb->index);
   FOR_EACH_EDGE (e, ei, bb->succs)
   {
        basic_block succ_bb = e->dest;
        fprintf(dump_file, "bb%d\t ", succ_bb->index);
   }
   fprintf(dump_file, "\n");
}
```

Printing Successors of a Basic Block

```
edge e;
edge_iterator ei;
basic_block bb;

FOR_EACH_BB_FN (bb, cfun)
{
   fprintf(dump_file, "\n Successor(s) of basic block bb%d: ",
                        bb->index);
   FOR_EACH_EDGE (e, ei, bb->succs)
   {
        basic_block succ_bb = e->dest;
        fprintf(dump_file, "bb%d\t ", succ_bb->index);
   }
   fprintf(dump_file, "\n");
}
```

Basic block iterator for current function represented by cfun

Printing Successors of a Basic Block

```
edge e;
edge_iterator ei;
basic_block bb;

FOR_EACH_BB_FN (bb, cfun)
{
   fprintf(dump_file, "\n Successor(s) of basic block bb%d: ",
                          bb->index);
   FOR_EACH_EDGE (e, ei, bb->succs)
   {
        basic_block succ_bb = e->dest;
        fprintf(dump_file, "bb%d\t ", succ_bb->index);
   }
   fprintf(dump_file, "\n");
}
```

Edge iterator

Other Useful APIs for Manipulating GIMPLE

Extracting parts of GIMPLE statements:

- `gimple_assign_lhs`: left hand side
- `gimple_assign_rhs1`: left operand of the right hand side
- `gimple_assign_rhs2`: right operand of the right hand side
- `gimple_assign_rhs_code`: operator on the right hand side

A complete list can be found in the file `gimple.h`

Discovering More Information from GIMPLE

- Discovering local variables
- Discovering global variables
- Discovering pointer variables
- Discovering assignment statements involving pointers
  (i.e. either the result or an operand is a pointer variable)

Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```

Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```

Local variable iterator

Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```

Exclude variables that do not appear in the source

Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```

Find the name from the TREE node

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

List of global variables of the current function

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

Exclude variables that do not appear in the source

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

Find the name from the TREE node

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

Go to the next item in the list

Assignment Statements Involving Pointers

```
int *p, *q;
void callme (int);
int main ()
{
    int  a, b;
    p = &b;
    callme (a);
    return 0;
}
void callme (int a)
{
    a = *(p + 3);
    q = &a;
}
```

```
main ()
{  int D.1965;
    int a;
    int b;

    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   int * p.0;
    int a.1;

    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Discovering Pointers in GIMPLE IR

```
static bool
is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool
is_pointer_type (tree type)
{
     if (POINTER_TYPE_P (type))
         return true;
     if (TREE_CODE (type) == ARRAY_TYPE)
         return is_pointer_var (TREE_TYPE (type));
     /* Return true if it is an aggregate type. */
     return AGGREGATE_TYPE_P (type);
}
```

Discovering Pointers in GIMPLE IR

```
static bool
is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool
is_pointer_type (tree type)
{
     if (POINTER_TYPE_P (type))
         return true;
     if (TREE_CODE (type) == ARRAY_TYPE)
         return is_pointer_var (TREE_TYPE (type));
     /* Return true if it is an aggregate type. */
     return AGGREGATE_TYPE_P (type);
}
```

Data type of the expression

Discovering Pointers in GIMPLE IR

```
static bool
is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool
is_pointer_type (tree type)
{
     if (POINTER_TYPE_P (type))
         return true;
     if (TREE_CODE (type) == ARRAY_TYPE)
         return is_pointer_var (TREE_TYPE (type));
     /* Return true if it is an aggregate type. */
     return AGGREGATE_TYPE_P (type);
}
```

Defines what kind of node it is

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{
    if (is_gimple_assign (stmt))
    {
         tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
                  fprintf (dump_file, "Pointer Statement :");
                  print_gimple_stmt (dump_file, stmt, 0, 0);
                  num_ptr_stmts++;
         }
    }
}
```

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{
    if (is_gimple_assign (stmt))
    {
         tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
                  fprintf (dump_file, "Pointer Statement :");
                  print_gimple_stmt (dump_file, stmt, 0, 0);
                  num_ptr_stmts++;
         }
    }
}
```

Extract the LHS of the assignment statement

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{
    if (is_gimple_assign (stmt))
    {
         tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
                  fprintf (dump_file, "Pointer Statement :");
                  print_gimple_stmt (dump_file, stmt, 0, 0);
                  num_ptr_stmts++;
         }
    }
}
```

Extract the first operand of the RHS

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{
    if (is_gimple_assign (stmt))
    {
         tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
                  fprintf (dump_file, "Pointer Statement :");
                  print_gimple_stmt (dump_file, stmt, 0, 0);
                  num_ptr_stmts++;
         }
    }
}
```

Extract the second operand of the RHS

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{
    if (is_gimple_assign (stmt))
    {
         tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
                  fprintf (dump_file, "Pointer Statement :");
                  print_gimple_stmt (dump_file, stmt, 0, 0);
                  num_ptr_stmts++;
         }
    }
}
```

Pretty print the GIMPLE statement

Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                           gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```

Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                           gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```

Basic block iterator parameterized with function

Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                           gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```

Current function (i.e. function being compiled)

Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                           gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```

GIMPLE statement iterator

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

Intraprocedural Analysis Results

```
main ()
{    ...
     p = &b;
     callme (a);
     D.1965 = 0;
     return D.1965;
}
callme (int a)
{    ...
     p.0 = p;
     a.1 = MEM[(int *)p.0 + 12B];
     a = a.1;
     q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1
- For `callme`: 2

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1
- For `callme`: 2

Why is the pointer in the red statement being missed?

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1
- For `callme`: 2

Why is the pointer in the red statement being missed?

Because it is deeper in the tree and our program does not search deeper in the tree

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
           continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
           continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Iterating over all the callgraph nodes

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
           continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Setting the current function in the context

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
           continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Basic Block Iterator

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
           continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

GIMPLE Statement Iterator

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
           continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Resetting the function context

Interprocedural Results

Number of Pointer Statements $= 3$

Interprocedural Results

Number of Pointer Statements $= 3$

Observation:

- Information can be collected for all the functions in a single pass
- Better scope for optimizations