Workshop on Essential Abstractions in GCC

Gray Box Probing of GCC

GCC Resource Center
(www.cse.iitb.ac.in/grc)

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

Part 1

Preliminaries

What is Gray Box Probing of GCC?

- **Black Box probing:**
  Examining only the input and output relationship of a system

- **White Box probing:**
  Examining internals of a system for a given set of inputs

- **Gray Box probing:**
  Examining input and output of various components/modules
  - Overview of translation sequence in GCC
  - Overview of intermediate representations
  - Intermediate representations of programs across important phases
Basic Transformations in GCC

Transformation from a language to a different language

Target Independent

GIMPLE → RTL

Target Dependent

Tree SSA

Optimize

Generate RTL

Optimize RTL

Generate ASM

RTL → ASM

GIMPLE Passes

RTL Passes

Transformation Passes in GCC 4.6.2

- A total of 207 unique pass names initialized in `{SOURCE}/gcc/passes.c`
- Total number of passes is 241.
  - Some passes are called multiple times in different contexts
    - Conditional constant propagation and dead code elimination are called thrice
  - Some passes are enabled for specific architectures
  - Some passes have many variations (eg. 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

Passes On GIMPLE in GCC 4.6.2

<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>10</td>
</tr>
<tr>
<td>Simple Interprocedural Passes</td>
<td>Conditional Constant Propagation, Inlining, SSA Construction</td>
<td>38</td>
</tr>
<tr>
<td>Regular Interprocedural Passes</td>
<td>Constant Propagation, Inlining, Pointer Analysis</td>
<td>10</td>
</tr>
<tr>
<td>LTO generation passes</td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>Other Intraprocedural Optimizations</td>
<td>Constant Propagation, Dead Code Elimination, PRE Value Range Propagation, Rename SSA</td>
<td>65</td>
</tr>
<tr>
<td>Loop Optimizations</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>Generating RTL</td>
<td></td>
<td>01</td>
</tr>
<tr>
<td>Total number of passes on GIMPLE</td>
<td></td>
<td>154</td>
</tr>
</tbody>
</table>

Passes On RTL in GCC 4.6.2

<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, Dead Code Elimination, Jump Optimization</td>
<td>27</td>
</tr>
<tr>
<td>Loop Optimizations</td>
<td>Loop Invariant Movement, Peeling, Unswitching</td>
<td>07</td>
</tr>
<tr>
<td>Machine Dependent Optimizations</td>
<td>Register Allocation, Instruction Scheduling, Peephole Optimizations</td>
<td>50</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>87</td>
</tr>
</tbody>
</table>
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, 3, and s)
  
  `gcc -c -O2 --help=optimizers -Q`

Producing the Output of GCC Passes

- Use the option `-fdump-<ir>--<passname>`
  
  `<ir>` could be
  - `tree`: Intraprocedural passes on GIMPLE
  - `ipa`: Interprocedural passes on GIMPLE
  - `rtl`: Intraprocedural passes on RTL

- Use `all` in place of `<pass>` to see all dumps
  
  Example: `gcc -fdump-tree-all -fdump-rtl-all test.c`

- Dumping more details:
  - Suffix `raw` for tree passes and `details` or `slim` for RTL passes
  - Individual passes may have more verbosity options (e.g. `-fsched-verbose=5`

- Use `-S` to stop the compilation with assembly generation

- Use `--verbose-asm` to see more detailed assembly dump

Total Number of Dumps

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Number of Dumps</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>47</td>
<td>Fast compilation</td>
</tr>
<tr>
<td>O1</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>174</td>
<td>Optimize for space</td>
</tr>
<tr>
<td>Os</td>
<td>175</td>
<td></td>
</tr>
</tbody>
</table>

Selected Dumps for Our Example Program

- GIMPLE dumps (t)
  
  001t.tu
  003t.original
  004t.gimple
  006t.vcg
  009t.omplower
  010t.lower
  012t.eh
  013t.cfg
  017t.ssa
  018t.veclower
  019t.inline.param1
  020t.inline
  037t.release.ssa
  038t.inline.param2
  044i.whole-program
  048i.inline

- IPA dumps (i)
  
  000i.cgraph
  014i.visibility
  015i.early_local_cleanups
  044i.whole-program
  048i.inline

- RTL dumps (r)
  
  144r.expand
  145r.sibling
  147r.initials
  148r.unshare
  149r.vregs
  150r.into_cfglayout
  151r.jump

- Other dumps
  
  163r.reginfo
  183r.out_of_cfglayout
  184r.split1
  186r.dfini
  187r.mode
  188r.asmcons
  191r.ira
  194r.split2
  198r.pro_and_epilogue
  211r.stack
  212r.alignments
  215r.mach
  216r.barriers
  220r.shorten
  221r.nothrow
  222r.final
  223r.dfinish
  assembly
Passes for First Level Graybox Probing of GCC

C Source Code

Parser

AST

Gimplifier

GIMPLE

CFG Generator

RTL Generator

Reg Allocator

ira

pro, epilogue generation

pattern Matcher

ASM Program

Lowering of abstraction!

Generating Abstract Syntax Tree

$ gcc -fdump-tree-original-raw test.c

Part 2

Examining AST Dump

Abstract Syntax Tree

```
test.c

int a;
int main()
{
    a = 55;
}
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Part 3

Examining GIMPLE Dumps

• About GIMPLE
  ▶ Three-address 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

• Examining GIMPLE Dumps
  ▶ Examining translation of data accesses
  ▶ Examining translation of control flow
  ▶ Examining translation of function calls

GIMPLE: Composite Expressions Involving Local and Global Variables

```c
int a;
int main()
{
  int x = 10;
  int y = 5;
  x = a + x * y;
  y = y - a * x;
}
```

Global variables are treated as “memory locations” and local variables are treated as “registers”

GIMPLE: 1-D Array Accesses

```c
int main()
{
  int a[3], x;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

```c
a[2] = 10;
D.1952 = a[2];
int main()
{
  int a[3], x;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

```c
D.1953 = a[1];
D.1954 = a[2];
```
GIMPLE: 2-D Array Accesses

```c
int main()
{
  int a[3][3], x, y;
  a[0][0] = 7;
  a[1][1] = 8;
  a[2][2] = 9;
  x = a[0][0] / a[1][1];
  y = a[1][1] % a[2][2];
}
```

- No notion of "addressable memory" in GIMPLE.
- Array reference is a single operation in GIMPLE and is linearized in RTL during expansion.

GIMPLE: Use of Pointers

```c
int main()
{
  int * D.1953;
  int * * a;
  int b;
  int c;
  b = &c;
  a = &b;
  **a = 10; /* c = 10 */
}
```

```c
main()
{
  int * D.1953;
  int * * a;
  int b;
  int c;
  b = &c;
  a = &b;
  D.1953 = *a;
  *D.1953 = 10;
}
```

GIMPLE: Use of Structures

```c
typedef struct address
{ char *name; }
ad;
typedef struct student
{ int roll;
  ad *ct; }
st;
int main()
{
  st *s;
  s = malloc(sizeof(st));
  s->roll = 1;
  s->ct = malloc(sizeof(ad));
  s->ct->name = "Mumbai";
  D.1957 = malloc(4);
  s->ct = D.1957;
  D.1958 = s->ct;
  D.1958->name = "Mumbai";
}
```

```c
typedef struct address
{ char *name; }
ad;
typedef struct student
{ int roll;
  ad *ct; }
st;
int main()
{
  st *s;
  s = malloc(sizeof(st));
  s->roll = 1;
  s->ct = malloc(sizeof(ad));
  s->ct->name = "Mumbai";
  D.1957 = malloc(4);
  s->ct = D.1957;
  D.1958 = s->ct;
  D.1958->name = "Mumbai";
  D.1959 = malloc(4);
  s->ct = D.1959;
  D.1959->name = "Mumbai";
}
```

GIMPLE: Pointer to Array

```c
int main()
{
  int * D.2048;
  int * D.2049;
  int * p_a;
  int a[3];
  p_a = &a[0];
  *p_a = 10;
  *(p_a+1) = 20;
  *(p_a+2) = 30;
}
```

```c
main ()
{
  int * D.2048;
  int * D.2049;
  int * p_a;
  int a[3];
  p_a = &a[0];
  *p_a = 10;
  D.2048 = p_a + 4;
  D.2048 = 20;
  D.2049 = p_a + 8;
  *D.2049 = 30;
}
```
**GIMPLE: Translation of Conditional Statements**

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

**GIMPLE: Translation of Loops**

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

**Control Flow Graph: Textual View**

```c
<bb 5>: if (a <= 12) goto <bb 6>;
else goto <bb 7>;
D.1199 = a + b;
a = D.1199 + c;
<bb 6>:
D.1199 = a + b;
a = D.1199 + c;
return;
<bb 7>:
return;
```

**Control Flow Graph: Pictorial View**

```
Block 4: If(a<=7)
    True
    Block 5: If(a<=12)
        True
        Block 6: a = b+c;
        False
        Block 7: return;
    False
    Block 3: a = a+1;
```

Control Flow Graph: Pictorial View

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;

False True

Control Flow Graph: Pictorial View

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;

False True

GIMPLE: Function Calls and Call Graph

extern int divide(int, int);int multiply(int a, int b){
    return a*b;
}

int main(){
    int x,y;
x = divide(20,5);
y = multiply(x,2);
printf("%d\n", y);
}

test.c.000i.cgraph

printf/3(-1) @0xb73c7ac8 availability:not availa
called by: main/1 (1.00 per call)
calls:
divide/2(-1) @0xb73c7a10 availability:not availa
called by: main/1 (1.00 per call)
calls:
main/1(1) @0xb73c7958 availability:available 38
called by:
calls: printf/3 (1.00 per call)
multiply/0 (1.00 per call)
divide/2 (1.00 per call)
multiply/0(0) @0xb73c78a0 availability:available
called by: main/1 (1.00 per call)
calls:

GIMPLE: Function Calls and Call Graph

test.c.000i.cgraph

call graph

printf/3(-1) called by: main/1
calls:
divide/2(-1)
called by: main/1
calls:
main/1(1)
called by:
calls: printf/3
calls: multiply/0
calls: divide/2

called by: main/1
calls:
GIMPLE: Call Graphs for Recursive Functions

```
int even(int n)
{ if (n == 0) return 1;
   else return (!odd(n-1));
}

int odd(int n)
{ if (n == 1) return 1;
   else return (!even(n-1));
}

main()
{ int n;
   n = abs(readNumber());
   if (even(n))
      printf("n is even\n");
   else printf("n is odd\n");
}
```

Inspect GIMPLE When in Doubt (1)

```
int x=2,y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?
x = 10, y = 5

```
x 3
y 3
(y + x) 6
(y + x) + y
```
### Inspect GIMPLE When in Doubt (1)

```c
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of `x` and `y`?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>3</td>
</tr>
<tr>
<td><code>y</code></td>
<td>5</td>
</tr>
<tr>
<td><code>(y + x)</code></td>
<td>6</td>
</tr>
<tr>
<td><code>(y + x) + y</code></td>
<td>11</td>
</tr>
</tbody>
</table>

\[ x = 10, y = 5 \]

### Inspect GIMPLE When in Doubt (2)

- How is `a[i] = i++` handled?
  - This is an undefined behaviour as per C standards.

- What is the order of parameter evaluation?
  - For a call `f(getX(), getY())`, is the order left to right? arbitrary?
  - Is the evaluation order in GCC consistent?

- Understanding complicated declarations in C can be difficult
  - What does the following declaration mean:

```c
int * (* (*MYVAR) (int)) [10];
```

Hint: Use `-fdump-tree-original-raw-verbose` option. The dump to see is `003t.original`
Translation of $a = a + 1$

**Dump file:** test.c.144r.expand

(insn 12 11 13 4 (parallel [ 
  (set (mem/c/i:SI 
    (plus:SI 
      (reg/f:SI 54 virtual-stack-vars) 
      (const_int -4 [0xffffffff])) [0 a+0 S4 A32]) 
    (plus:SI 
      (mem/c/i:SI 
        (plus:SI 
          (reg/f:SI 54 virtual-stack-vars) 
          (const_int -4 [0xffffffff])) [0 a+0 S4 A32]) 
        (const_int 1 [0x1]))) 
  (clobber (reg:CC 17 flags)) 
]) t.c:24 -1 (nil))

---

**RTL for i386:** Arithmetic Operations (1)

Translation of $a = a + 1$

**Dump file:** test.c.144r.expand

(insn 12 11 13 4 (parallel [ 
  (set (mem/c/i:SI 
    (plus:SI 
      (reg/f:SI 54 virtual-stack-vars) 
      (const_int -4 [0xffffffff])) [0 a+0 S4 A32]) 
    (plus:SI 
      (mem/c/i:SI 
        (plus:SI 
          (reg/f:SI 54 virtual-stack-vars) 
          (const_int -4 [0xffffffff])) [0 a+0 S4 A32]) 
        (const_int 1 [0x1]))) 
  (clobber (reg:CC 17 flags)) 
]) t.c:24 -1 (nil))
RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$

Dump file: test.c.144r.expand

(a is a local variable allocated on stack)

Dump file: test.c.144r.expand

Output with slim suffix

Current Instruction

Side-effect of plus may modify condition code register non-deterministically

Additional Information in RTL

Translation of $a = a + 1$

Dump file: test.c.144r.expand

Output with slim suffix

Current Instruction
Additional Information in RTL

(insn 12 11 13 4 (parallel [
  (set (mem/c/i:SI (plus:SI
    (reg/f:SI 54 virtual-stack-vars)
    (const_int -4 [0xfffffffc])) [0 a+0 S4 A32])
  (plus:SI
    (mem/c/i:SI
      (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffffc])) [0 a+0 S4 A32])
  (const_int 1 [0x1]))
  (clobber (reg:CC 17 flags))
]) t.c:24 -1 (nil))

Previous Instruction

Basic Block

File name: Line number

Next Instruction
Additional Information in RTL

(insn 12 11 13 4 (parallel [
  (set (mem/c/i:SI (plus:SI
    (reg/f:SI 54 virtual-stack-vars)
    (const_int -4 [0xfffffffc])) [0 a+0 S4 A32]))
  (plus:SI
    (mem/c/i:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffffc])) [0 a+0 S4 A32]))
  (const_int 1 [0x1]))
  (clobber (reg:CC 17 flags)))
]) t.c:24 -1 (nil))

memory reference that does not trap
RTL for i386: Arithmetic Operations (2)

Translation of $a = a + 1$ when $a$ is a global variable

Dump file: test.c.144r.expand

(insn 11 10 12 4 (set
  (reg:SI 64 [ a.0 ])
  (mem/c:1:SI (symbol_ref:SI ("a")
      <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])) t.c:26 -1 (nil))

(insn 12 11 13 4 (parallel [
   (set (reg:SI 63 [ a.1 ])
      (plus:SI (reg:SI 64 [ a.0 ])
         (const_int 1 [0x1])))
   (clobber (reg:CC 17 flags))]) t.c:26 -1 (nil))

(insn 13 12 14 4 (set
  (mem/c:1:SI (symbol_ref:SI ("a")
      <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])
  (reg:SI 63 [ a.1 ])) t.c:26 -1 (nil))

Load a into reg64
reg63 = reg64 + 1

store reg63 into a

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
RTL for i386: Arithmetic Operations (2)

Translation of $a = a + 1$ when $a$ is a global variable

Dump file: test.c.144r.expand

Load a into reg64
reg63 = reg64 + 1
store reg63 into a

Output with slim suffix
r64:SI=[‘a’]
{r63:SI=r64:SI+0x1;
clobber flags:CC;
}
[‘a’]=r63:SI

RTL for i386: Arithmetic Operations (3)

Translation of $a = a + 1$ when $a$ is a formal parameter

Dump file: test.c.144r.expand

Access through argument pointer register instead of frame pointer register
No offset required?

Access through argument pointer register instead of frame pointer register
No offset required?

Access through argument pointer register instead of frame pointer register
No offset required?

Access through argument pointer register instead of frame pointer register
No offset required?

Access through argument pointer register instead of frame pointer register
No offset required?
### RTL for i386: Arithmetic Operation (4)

Translation of \(a = a + 1\) when \(a\) is the second formal parameter

**Dump file:** test.c.144r.expand

\[
\text{(insn 10 9 11 4 (parallel [}
\text{(set (mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
(const_int 4 [0x4])) [0 a+0 S4 A32])
(const_int 4 [0x4])) [0 a+0 S4 A32])
(const_int 1 [0x1]))
(clobber (reg:CC 17 flags)))
}) t1.c:25 -1 (nil))
\]

### RTL for i386: Control Flow

What does this represent?

\[
\text{(jump_insn 15 14 16 4 (set (pc)}
\text{ (if \_then_else (lt (reg:CCGC 17 flags)
(const_int 0 [0x0])))
(label_ref 12)
(pc)) p1.c:6 -1 (nil)
(nil)
-> 12)
\]

\[
\text{pc = r17 <0 ? label(12) : pc}
\]

---

### RTL for spim: Arithmetic Operations

Translation of \(a = a + 1\) when \(a\) is a local variable

**Dump file:** test.c.144r.expand

\[
\text{(insn 7 6 8 4 (set (reg:SI 39)
(mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
(const_int -4 [...] )) [...] )) -1 (nil))
\]

\[
\text{(insn 8 7 9 4 test.c:6 (set (reg:SI 40)
(plus:SI (reg:SI 39)
(const_int 1 [...] )) -1 (nil))}
\]

\[
\text{(insn 9 8 10 4 test.c:6 (set}
\text{ (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
(const_int -4 [...] ]) [...] ))
\text{(reg:SI 40)) test.c:6 -1 (nil))}
\]

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated.

---

### RTL for spim: Control Flow

What does this represent?

\[
\text{(jump_insn 15 14 16 4 (set (pc)}
\text{ (if \_then_else (lt (reg:CCGC 17 flags)
(const_int 0 [0x0])))
(label_ref 12)
(pc)) p1.c:6 -1 (nil)
(nil)
-> 12)
\]

\[
\text{pc = r17 <0 ? label(12) : pc}
\]
RTL for i386: Control Flow

Translation of if (a > b) { /* something */ }

**Dump file:** test.c.144r.expand

```c
if (a > b)
{
    /* something */
}
```

**Dump file:** test.c.144r.expand

```c
insn 8 7 9 (set (reg:SI 61)
    (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
        (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])) test.c:7 -1 (nil))

insn 9 8 10 (set (reg:CCGC 17 flags)
    (compare:CCGC (reg:SI 61)
        (const_int -4 [0xffffffffc]) [0 b+0 S4 A32]))) test.c:7 -1 (nil))

(jump_insn 10 9 0 (set (pc)
    (if_then_else (le (reg:CCGC 17 flags)
        (const_int 0 [0x0])
        (label_ref 13)
        (pc))) test.c:7 -1 (nil)
    -> 13)
```

Examining Assembly Dumps

### i386 Assembly

**Dump file:** test.s

```assembly
jmp .L2

.L3:
    addl $1, -4(%ebp)
.L2:
    cmpl $7, -4(%ebp)
    jle .L3
    cmpl $12, -4(%ebp)
    jg .L6
    movl -8(%ebp), %edx
    movl -4(%ebp), %eax
    addl %edx, %eax
    addl -12(%ebp), %eax
    movl %eax, -4(%ebp)

.L6:
```

```assembly
while (a <= 7)
{
    a = a+1;
}
```

```assembly
jmp .L2

.L3:
    addl $1, -4(%ebp)
.L2:
    cmpl $7, -4(%ebp)
    jle .L3
    cmpl $12, -4(%ebp)
    jg .L6
    movl -8(%ebp), %edx
    movl -4(%ebp), %eax
    addl %edx, %eax
    addl -12(%ebp), %eax
    movl %eax, -4(%ebp)

.L6:
```

```assembly
while (a <= 7)
{
    a = a+1;
}
```

```assembly
jmp .L2

.L3:
    addl $1, -4(%ebp)
.L2:
    cmpl $7, -4(%ebp)
    jle .L3
    cmpl $12, -4(%ebp)
    jg .L6
    movl -8(%ebp), %edx
    movl -4(%ebp), %eax
    addl %edx, %eax
    addl -12(%ebp), %eax
    movl %eax, -4(%ebp)

.L6:
```
1 July 2012

i386 Assembly

Dump file: test.s

jmp .L2
.L3:
    addl $1, -4(%ebp)
.L2:
    cmpl $7, -4(%ebp)jle .L3
    cmpl $12, -4(%ebp)jg .L6
    movl -8(%ebp), %edx
    movl -4(%ebp), %eax
    addl %edx, %eax
    addl -12(%ebp), %eax
    movl %eax, -4(%ebp)
.L6:

Part 6

Examining GIMPLE Optimization

Example Program for Observing Optimizations

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

• What does this program return?
  • 12
• We use this program to illustrate various shades of the following optimizations:
  Constant propagation, Copy propagation, Loop unrolling, Dead code elimination
Compilation Command

```
$gcc -fdump-tree-all -O2 ccp.c
```

Example Program 1

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

Control Flow Graph: Pictorial and Textual View

```
<bb 2>: a = 1; b = 2; c = 3; n = c * 2; goto <bb 4>;
```

Dump file ccp.c.013t.cfg

```
B2: a = 1
    b = 2
    c = 3
    n = c * 2

B4: if a <= n
    F
    B5: if a <= 11
        T
        a = a + 1
        B3
        F
        D.1200 = a + b
        a = D.1200 + c
        B6
        F
        D.1201 = a
        return D.1201
    T
    B7
```

Dump file ccp.c.013t.cfg

```
<bb 2>: a = 1; b = 2; c = 3; n = c * 2;
```

```
B2: a = 1
    b = 2
    c = 3
    n = c * 2

B4: if a <= n
    F
    B5: if a <= 11
        T
        a = a + 1
        B3
        F
        D.1200 = a + b
        a = D.1200 + c
        B6
        F
        D.1201 = a
        return D.1201
    T
    B7
```
Control Flow Graph: Pictorial and Textual View

Control flow graph

Dump file ccp.c.013t.cfg

1. a = 1
2. b = 2
3. c = 3
4. n = c + 2

B2

B4

if a ≤ n

F

T

B5

if a ≤ 11

a = a + 1

B3

D.1201 = a

return D.1201

B7

<bb 3>:

a = a + 1;

<bb 4>:

if (a ≤ n)

goto <bb 3>;
else

goto <bb 5>;

D.1200 = a + b

a = D.1200 + c

B6

B7

D.1201 = a

return D.1201

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

Single Static Assignment (SSA) Form

Control flow graph

Dump file ccp.c.013t.cfg

a = 1; b = 2

b = 4 = 2

c = 3; n = c + 2

B2

B4

if a ≤ n

F

T

B5

if a ≤ 11

a = a + 1

B3

D.1200 = a + b

a = D.1200 + c

B6

B7

D.1201 = a

return D.1201

<bb 7>:

D.1201 = a;

return D.1201;

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Single Static Assignment (SSA) Form

Control flow graph

SSA Form

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
### Single Static Assignment (SSA) Form

#### Control flow graph

- B2: `a = 1; b = 2; c = 3; n = c * 2`
- B4: if `a ≤ n`
  - B3: `a = a + 1`
  - B5: if `a ≤ 11`
- B6: `D.1200 = a + b; a = D.1200 + c`
- B7: `D.1201 = a; return D.1201`

#### SSA Form

- B2: `a = 1; b = 2; c = 3; n = c * 2`
- B4: if `a ≤ n`
  - B3: `a = a + 1`
  - B5: if `a ≤ 11`
- B6: `D.1200 = a + b; a = D.1200 + c`
- B7: `D.1201 = a; return D.1201`

```
Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
```

### Properties of SSA Form

- A \( \phi \) function is a multiplexer or a selection function
- Every use of a variable corresponds to a unique definition of the variable
- For every use, the definition is guaranteed to appear on every path leading to the use

### SSA Form: Pictorial and Textual View

#### CFG in SSA form

- B2: `\( a = 1; b = 2 \)`
- B4: if `\( a \leq n \)`
  - B3: `\( a = a + 1 \)`
- B5: if `\( a \leq 11 \)`
- B6: `\( D.1200 = a + b; a = D.1200 + c \)`
- B7: `\( D.1201 = a; return D.1201 \)`

```
Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
```

### SSA Form: Dump file ccp.c.017t.ssa

```
<bb 2>:
a = 1; b = 2;
c = 3; n = c * 2;
goto <bb 4>;

<bb 3>:
a = a + 1;
```

```
Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
```
A Comparison of CFG and SSA Dumps

Dump file ccp.c.013t.cfg

Dump file ccp.c.017t.ssa

A Comparison of CFG and SSA Dumps

<bb 6>:
D.1200 = a + b;
a = D.1200 + c;

<bb 7>:
D.1201 = a;
return D.1201;

Copy Renaming

Input dump: ccp.c.017t.ssa

Output dump: ccp.c.022t.copyrename1

<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
D.1201_10 = a_2;
return D.1201_10;

First Level Constant and Copy Propagation

Input dump: ccp.c.022t.copyrename1

Output dump: ccp.c.023t.ccp1

<bb 2>:
a_3 = 1;
b_4 = 2;
c_5 = 3;
if (a_3 <= n_6)
goto <bb 3>;
else
goto <bb 5>;

<bb 3>:
a_7 = a_1 + 1;

<bb 4>:
# a_1 = PHI <a_1(2), a_7(3)>
if (a_1 <= n_8)
goto <bb 3>;
else
goto <bb 5>;

<bb 6>:
D.1200_8 = a_1 + b_4;
a_9 = D.1200_8 + c_5;

First Level Constant and Copy Propagation

Input dump: ccp.c.022t.copyrename1

Output dump: ccp.c.023t.ccp1

<bb 2>:
a_3 = 1;
b_4 = 2;
c_5 = 3;
if (a_3 <= 6)
goto <bb 3>;
else
goto <bb 4>;

...
Second Level Copy Propagation

Input dump: ccp.c.023t.ccp1

Output dump: ccp.c.027t.copyprop1

<bb 6>:
D.1200_8 = a_1 + 2;
= D.1200_8 + 3;

<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
a_10 = a_2;
return a_10;

The Result of Copy Propagation and Renaming

B2

a_3 = 1; b_4 = 2
a_9 = D.1200_8 + 3;

<bb 6>:

<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
return a_2;

• No uses for variables a_3, b_4, c_5, and n_6
• Assignments to these variables can be deleted

The Result of Copy Propagation and Renaming

B2

a_3 = 1; b_4 = 2
b_5 = a_9 + 5

<bb 6>:

<bb 7>:

Dead Code Elimination Using Control Dependence

Dump file ccp.c.029t.cddce1

<bb 2>:
goto <bb 4>
<bb 3>:
a_7 = a_1 + 1;
<bb 4>:
# a_1 = PHI <1(2), a_7(3)>
if (a_1 <= 6) goto <bb 3>
else goto <bb 5>
<bb 5>:
if (a_1 <= 11) goto <bb 6>
else goto <bb 7>
<bb 6>:
a_9 = a_1 + 5
<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
return a_2
**Loop Unrolling**

\[
\begin{align*}
\text{B2:} & \\
\text{B4:} & \quad a_1 = \phi (1, a_7) \\
& \quad \text{if } a_1 \leq 6 \\
\text{B5:} & \quad a_7 = a_1 + 1 \\
\text{B6:} & \quad \text{if } a_1 \leq 11 \\
\text{B7:} & \quad a_2 = \phi (a_1, a_9) \\
& \quad \text{return } a_2 \\
\end{align*}
\]

**Complete Unrolling of Inner Loops**

**Dump file:** ccp.c.058t.cunrolli

\[
\begin{align*}
\text{<bb 2>:} & \\
& \quad a_{12} = 2; \\
& \quad a_{14} = a_{12} + 1; \\
& \quad a_{16} = a_{14} + 1; \\
& \quad a_{18} = a_{16} + 1; \\
& \quad a_{20} = a_{18} + 1; \\
& \quad a_{22} = a_{20} + 1; \\
& \quad \text{if } (a_{22} \leq 11) \text{ goto } <\text{bb 3}>; \\
& \quad \text{else goto } <\text{bb 4}>; \\

\text{<bb 3>:} & \\
& \quad a_9 = a_{22} + 5; \\

\text{<bb 4>:} & \\
& \quad a_2 = \phi (a_{22}, a_9) \\
& \quad \text{return } a_2; \\
\end{align*}
\]

**Another Round of Constant Propagation**

**Dump file:** ccp.c.059t.ccp2

\[
\begin{align*}
\text{main ():} & \\
& \{ \\
& \quad \text{<bb 2>:} \\
& \quad \quad \text{return } 12; \\
& \} \\
\end{align*}
\]

**Conclusions**
Gray Box Probing of GCC: Conclusions

- Source code is transformed into assembly by lowering the abstraction level step by step to bring it close to the machine.
- This transformation can be understood to a large extent by observing inputs and output of the different steps in the transformation.
- It is easy to prepare interesting test cases and observe the effect of transformations.
- One optimization often leads to another. Hence GCC performs many optimizations repeatedly (eg. copy propagation, dead code elimination).