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

1 July 2012

Outline

• Introduction to Graybox Probing of GCC
• Examining AST
• Examining GIMPLE Dumps
  ▶ Translation of data accesses
  ▶ Translation of intraprocedural control flow
  ▶ Translation of interprocedural control flow
• Examining RTL Dumps
• Examining Assembly Dumps
• Examining GIMPLE Optimizations
• Conclusions
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

<table>
<thead>
<tr>
<th>Target Independent</th>
<th>Target Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse → Simplify</td>
<td>Generate RTL</td>
</tr>
<tr>
<td>Tree SSA</td>
<td>Optimize RTL</td>
</tr>
<tr>
<td>Optimize</td>
<td>Generate ASM</td>
</tr>
</tbody>
</table>

GIMPLE → RTL

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 (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
### 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>Passes (Non-LTO)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regular Interprocedural Passes</td>
<td>Constant Propagation, Inlining, Pointer Analysis</td>
<td>10</td>
</tr>
<tr>
<td>(LTO)</td>
<td></td>
<td></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>Vectorization, Parallelization, Copy Propagation, Dead Code Elimination</td>
<td>28</td>
</tr>
<tr>
<td>Generating RTL</td>
<td></td>
<td>01</td>
</tr>
<tr>
<td><strong>Total number of passes on GIMPLE</strong></td>
<td></td>
<td><strong>154</strong></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><strong>Total number of passes on RTL</strong></td>
<td></td>
<td><strong>87</strong></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></td>
</tr>
<tr>
<td>Os</td>
<td>175</td>
<td>Optimize for space</td>
</tr>
</tbody>
</table>

### Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

### 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

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

**rtl dumps (r)**
- 138t.cplxlower
- 143t.optimized
- 224t.statistics
- ipa dumps (i)
- 000i.cgraph
- 014i.visibility
- 015i.early_local_cleanups
- 044i.whole-program
- 048i.inline
- rtl dumps (r)
- 144r.expand
- 145r.sibling
- 147r.initvals
- 148r.unshare
- 149r.vregs
- 150r.into_cflayout
- 151r.jump
- 163r.reginfo
- 183r.outof_cflayout
- 184r.split1
- 186r.definit
- 187r.mode_sw
- 188r.asmcons
- 191rира
- 194r.split2
- 198r.pro_and_epilogue
- 211r.stack
- 212r.alignments
- 215r.mach
- 216r.barriers
- 220r.shorten
- 221r.nothrow
- 222r.final
- 223r.dfinish
- assembly
Essential Abstractions in GCC

Part 2

Examining AST Dump
```bash
$ gcc -fdump-tree-original-raw test.c
```

```c
int a;
int main()
{
    a = 55;
}
```
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
int a[3], x;
int main()
{
    a[2] = 10;
    a[1] = 10;
    a[0] = a[2] = 10;
    x = a[1] + a[2];
    a[0] = a[1] + a[1]*x;
}
```
GIMPLE: 2-D Array Accesses

```
int main()
{
    int a[3][3], x, y;
    a[0][0] = 7;
    a[1][1] = 8;
    a[2][2] = 9;
    D.1953 = a[0][0];
    D.1954 = a[1][1];
    x = D.1953 / D.1954;
    D.1955 = a[1][1];
    D.1956 = a[2][2];
    y = D.1955 % D.1956;
}
```

- 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

```
int main()
{
    int * D.1953; int * * a; int b, c;
    b = &c;
    a = &b;
    **a = 10; /* c = 10 */
}
```
### GIMPLE: Use of Structures

**test.c**

```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";
}
```

### Notes

- **Essential Abstractions in GCC**
- **GCC Resource Center, IIT Bombay**

### GIMPLE: Pointer to Array

**test.c**

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

### Notes

- **Essential Abstractions in GCC**
- **GCC Resource Center, IIT Bombay**
### GIMPLE: Translation of Conditional Statements

**test.c**

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

**test.c.004t.gimple**

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

### GIMPLE: Translation of Loops

**test.c**

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

**test.c.004t.gimple**

```c
goto <D.1197>;
<D.1196>:
    a = a + 1;
<D.1197>:
if (a <= 7) goto <D.1196>;
else goto <D.1198>;
<D.1198>:
```
### Control Flow Graph: Textual View

#### test.c.004t.gimple

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

#### test.c.013t.cfg

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

### Control Flow Graph: Pictorial View

#### test.c.013t.cfg

```
Block 4:
If(a<=7)
    False
    Block 5:
    If(a<=12)
        True
        Block 3:
        a = a +1;
    False
    Block 6:
    D.1199= a + b;
a= D.1199 + c;
```

```
    True
    Block 7:
    return;
```
Control Flow Graph: Pictorial View

test.c.013t.cfg

while(a <= 7)
a = a + 1;

Block 4:
if(a<=7)
False
True

Block 5:
if(a<=12)

Block 3:
a = a +1;

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

Block 7:
return;

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
# GIMPLE: Function Calls and Call Graph

**test.c**

```c
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
", y);
}
```

**test.c.000i.cgraph**

```c
printf/3(-1) @0xb73c7ac8 availability:not available
called by: main/1 (1.00 per call)
calls:
divide/2(-1) @0xb73c7a10 availability:not available
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 vailability:available
called by: main/1 (1.00 per call)
calls:
```

**Notes**
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");
}
int x=2,y=3;
x = y++ + ++x + ++y;

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
</tr>
<tr>
<td>(y + x)</td>
<td>6</td>
</tr>
<tr>
<td>(y + x) + y</td>
<td></td>
</tr>
</tbody>
</table>

---

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
### 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>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
</tr>
<tr>
<td>(y + x)</td>
<td>6</td>
</tr>
<tr>
<td>(y + x) + y</td>
<td>11</td>
</tr>
</tbody>
</table>
```

x = 10 , y = 5
```

---

**Notes**

- `D.1572 = y + x; /* 6 */`
- `y = y + 1; /* 4 */`
- `x = D.1572 + y; /* 10 */`
- `y = y + 1; /* 5 */`
• 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`
RTL for i386: Arithmetic Operations (1)

Translation of \( a = a + 1 \)

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

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

*a* is a local variable allocated on stack
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)

(side-effect of plus may modify condition code register non-deterministically)

set

reg 54

-4

plus

mem

mem

plus

reg 54

-4

parallel

clobber

reg CC

Notes
Translation of $a = a + 1$

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

```c
{[r54:SI-0x4]=[r54:SI-0x4]+0x1;
clobber flags:CC;
}
```

**Notes**

Additional Information in RTL

```c
{[r54:SI-0x4]=[r54:SI-0x4]+0x1;
clobber flags:CC;
}
```
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 [0xffffffffc]) [0 a+0 S4 A32])
  (plus:SI
   (mem/c/i:SI
    (plus:SI
     (reg/f:SI 54 virtual-stack-vars)
     (const_int -4 [0xffffffffc]) [0 a+0 S4 A32])
     (const_int 1 [0x1]))
    (clobber (reg:CC 17 flags))
   ])
  ])
]) t.c:24 -1 (nil))
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])
       (const_int 1 [0x1]))
     (clobber (reg:CC 17 flags))
]) t.c:24 -1 (nil))
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 [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)

Notes

Scalar that is not a part of an aggregate

Memory reference that does not trap
Additional Information in RTL

\begin{verbatim}
(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
                (plus: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))
\end{verbatim}
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/i: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/i:SI (symbol_ref:SI ("a")
    <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])
  (reg:SI 63 [ a.1 ])) t.c:26 -1 (nil))
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

$reg63 = reg64 + 1$

Dump file: test.c.144r.expand

Load a into reg64

$reg63 = reg64 + 1$

store reg63 into a

Notes

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Translation of $a = a + 1$ when $a$ is a global variable

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

```plaintext
(insn 11 10 12 4 (set
  (reg:SI 64 [ a.0 ])
  (mem/c/i:SI (symbol_ref:SI ("a")
    <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])))

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

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

[‘a’]=r63:SI
```

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

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

```plaintext
(insn 10 9 11 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))
```

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

---

1 July 2012
Graybox Probing: Examining RTL Dumps
33/56

RTL for i386: Arithmetic Operations (3)

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

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

```plaintext
(insn 10 9 11 4 (parallel [
  (set
    (mem/c/i:SI
      (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32])
    (plus:SI
      (mem/c/i:SI
        (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32])
        (const_int 1 [0x1])))
      (clobber (reg:CC 17 flags))
    ])
]) t1.c:25 -1 (nil))
```
Translation of $a = a + 1$ when $a$ is a formal parameter

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

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

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

No output with slim suffix

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
RTL for i386: Arithmetic Operation (4)

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

Dump file: test.c.144r.expand

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

Offset 4 added to the argument pointer register

When \( a \) is the first parameter, its offset is 0!

Output with slim suffix

\{[r53:SI+0x4]=[r53:SI+0x4]+0x1; clobber flags:CC; \}
RTL for spim: Arithmetic Operations

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

Dump file: test.c:144r.expand

\[
\begin{align*}
& \text{insn 7 6 8 4 (set (reg:SI 39)} \\
& \quad (\text{mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)} \\
& \quad (\text{const_int -4 [...])) [...]))) -1 (nil))} \\
& \text{insn 8 7 9 4 test.c:6 (set (reg:SI 40)} \\
& \quad (\text{plus:SI (reg:SI 39)} \\
& \quad (\text{const_int 1 [...]))} -1 (nil)) \\
& \text{insn 9 8 10 4 test.c:6 (set} \\
& \quad (\text{mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)} \\
& \quad (\text{const_int -4 [...]))} [...])} \\
& \quad (\text{reg:SI 40}) \text{ test.c:6} -1 (nil))
\end{align*}
\]

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

RTL for i386: Control Flow

What does this represent?

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

(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)
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:
while (a <= 7) {
    a = a+1;
}
if (a <= 12) {
    a = a+b+c;
}
```
Dump file: `test.s`

```assembly
jmp .L2
.L3:
  addl $1, -4(%ebp)
.L2:
  cmpl $7, -4(%ebp)     # while (a <= 7)
  cmpl $12, -4(%ebp)   #   if (a <= 12)
    jle .L3            #     a = a+1;
    jg .L6             #     a = a+b+c;
  movl -8(%ebp), %edx
  movl -4(%ebp), %eax
  addl %edx, %eax
  addl -12(%ebp), %eax
  movl %eax, -4(%ebp)

.L6:
while (a <= 7) {
  a = a+1;
}
if (a <= 12) {
  a = a+b+c;
}
```

Notes
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

Program ccp.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

Dump file ccp.c.013t.cfg

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Control Flow Graph: Pictorial and Textual View

Control flow graph

Dump file ccp.c.013t.cfg

<bb 3>:
  a = a + 1;

<bb 4>:
  if (a <= n)
    goto <bb 3>;
  else
    goto <bb 5>;

<bb 5>:
  if (a <= 11)
    goto <bb 6>;
  else
    goto <bb 7>;

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

<bb 7>:
  D.1201 = a;
  return D.1201;
Control Flow Graph: Pictorial and Textual View

Control flow graph

Dump file ccp.c.013t.cfg

Control Flow Graph: Pictorial and Textual View

Notes

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

SSA Form

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 construction algorithm is expected to insert as few $\phi$ functions as possible to ensure the above properties
Dump file ccp.c.017t.ssa

<bb 2>:
    a_3 = 1;
    b_4 = 2;
    c_5 = 3;
    n_6 = c_5 * 2;
goto <bb 4>;

<bb 3>:
    a_7 = a_1 + 1;

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
### SSA Form: Pictorial and Textual View

#### Dump file ccp.c.017t.ssa

```
<bb 4>:
    # a_1 = PHI <a_3(2), a_7(3)>
    if (a_1 <= n_6)
        goto <bb 3>;
    else
        goto <bb 5>;

<bb 5>:
    if (a_1 <= 11)
        goto <bb 6>;
    else
        goto <bb 7>;
```

```
# a_2 = PHI <a_2(5), a_9(6)>
D.1201_10 = a_2;
return D.1201_10;
```
A Comparison of CFG and SSA Dumps

Dump file ccp.c.013t.cfg

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

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

Dump file ccp.c.017t.ssa

```c
<bb 2>:
a_3 = 1;
b_4 = 2;
c_5 = 3;
n_6 = c_5 * 2;
goto <bb 4>;

<bb 3>:
a_7 = a_1 + 1;
```

Notes

```
<bb 4>:
if (a <= n)
goto <bb 3>;
else
goto <bb 5>;

<bb 5>:
if (a <= 11)
goto <bb 6>;
else
goto <bb 7>;
```

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

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

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

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

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

Notes

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;

<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
a_10 = a_2;
return a_10;
First Level Constant and Copy Propagation

Input dump: ccp.c.022t.copyrename1

<bb 2>:
  a_3 = 1;
  b_4 = 2;
  c_5 = 3;
  n_6 = c_5 * 2;
  goto <bb 4>;

<bb 3>:
  a_7 = a_1 + 1;

<bb 4>:
# a_1 = PHI < a_3(2), a_7(3)>
  if (a_1 <= n_6)
    goto <bb 3>;
  else
    goto <bb 5>;

Output dump: ccp.c.023t.ccp1

<bb 2>:
  a_3 = 1;
  b_4 = 2;
  c_5 = 3;
  n_6 = 6;
  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>;

...
Second Level Copy Propagation

Input dump: ccp.c.023t.ccp1

\[
\begin{align*}
\text{bb 6}: & \quad D.1200_8 = a_1 + 2; \\
& \quad a_9 = D.1200_8 + 3; \\
\text{bb 7}: & \quad \# a_2 = PHI <a_1(5), a_9(6)> \\
& \quad a_{10} = a_2; \\
& \quad \text{return } a_{10};
\end{align*}
\]

Output dump: ccp.c.027t.copyprop1

\[
\begin{align*}
\text{bb 6}: & \quad a_9 = a_1 + 5; \\
\text{bb 7}: & \quad \# a_2 = PHI <a_1(5), a_9(6)> \\
& \quad \text{return } a_2;
\end{align*}
\]

The Result of Copy Propagation and Renaming
1 July 2012  Graybox Probing: Examining GIMPLE Optimization

The Result of Copy Propagation and Renaming

- No uses for variables \(a_3, b_4, c_5, \) and \(n_6\)
- Assignments to these variables can be deleted

Dump file ccp.c.029t.cddce1

Dead Code Elimination Using Control Dependence

- No uses for variables \(a_3, b_4, c_5, \) and \(n_6\)
- Assignments to these variables can be deleted
Loop Unrolling

B4
\[ a_{12} = \phi (1, a_7) \]
\[ \text{if } a_1 \leq 6 \]
F
B3
\[ a_7 = a_{12} + 1 \]
T
B5
\[ a_9 = a_{12} + 5 \]
B6
\[ a_2 = \phi (a_{12}, a_9) \]
return a_2

Notes

Complete Unrolling of Inner Loops

Dump file: ccp.c.058t.cunrolli

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

<bb 3>:
\[ a_9 = a_{22} + 5; \]

<bb 4>:
\# a_2 = PHI <a_{22}(2), a_9(3)>
return a_2;
Another Round of Constant Propagation

Input

```
a_{12} = 2
a_{14} = a_{12} + 1
a_{16} = a_{14} + 1
a_{18} = a_{16} + 1
a_{20} = a_{18} + 1
a_{22} = a_{20} + 1
if a_{22} <= 11
```

Dump file: ccp.c.059t.cc

```
main ()
{
    <bb 2>:
        return 12;

    if a_{22} <= 11
```

Part 7

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)