Workshop on Essential Abstractions in GCC

Graybox Probing for Machine Independent Optimizations

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

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

30 June 2011

Outline

• Example 1
  ▶ Constant Propagation
  ▶ Copy Propagation
  ▶ Dead Code Elimination
  ▶ Loop unrolling

• Example 2
  ▶ Partial Redundancy Elimination
  ▶ Copy Propagation
  ▶ Dead Code Elimination

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

30 June 2011
Machine Independent Optimizations: Outline
1/29

Outline

• Example 1
  ▶ Constant Propagation
  ▶ Copy Propagation
  ▶ Dead Code Elimination
  ▶ Loop unrolling

• Example 2
  ▶ Partial Redundancy Elimination
  ▶ Copy Propagation
  ▶ Dead Code Elimination

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

30 June 2011
Machine Independent Optimizations: Outline
1/29

Example Program 1

```
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

```c
#include <stdio.h>

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

Diagram showing the control flow of the program, with nodes and edges indicating the logic flow. The diagram highlights the conditional statements and loops in the program, along with the variables and their values at different points in the program flow.

Dump file ccp.c.013t.cfg

The dump file contains the control flow graph in text format, which can be used to understand the program's logic flow in a textual manner.

Control Flow Graph: Pictorial and Textual View

The control flow graph is depicted in two different ways: pictorially and textually. The pictorial view shows the flow of control through the program, while the textual view provides a line-by-line description of the program's logic.

Control Flow Graph: Pictorial and Textual View

The control flow graph is shown in both pictorial and textual formats, allowing for a comprehensive understanding of the program's control flow. The textual view provides a detailed description of each step in the program, while the pictorial view offers a visual representation of the flow.

Dump file ccp.c.013t.cfg

The dump file contains the control flow graph in a structured format, which can be parsed and used to debug or analyze the program's logic.
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>;

D.1201 = a
return D.1201

Control Flow Graph: Pictorial and Textual View

Control flow graph

Dump file ccp.c.013t.cfg

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

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

Single Static Assignment (SSA) Form

Control flow graph

SSA Form

a = 1; b = 2
c = 3; n = c + 2

a = φ(a, a)

D.1200 = a + b
a = D.1200 + c
Single Static Assignment (SSA) Form

Control flow graph

SSA Form

B2
\[ a = 1; b = 2 \]
B3
\[ c = 3; n = c + 2 \]
B4
\[ if a \leq n \]
B5
\[ j = a + 1 \]
B6
\[ D.1200 = a + b \]
B7
\[ a = D.1200 + c \]
B7
\[ D.1201 = a \]
return D.1201

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

Control flow graph

SSA Form

B2
\[ a = 1; \ b = 2 \]
B4
\[ c = 3; \ n = c \ast 2 \]
B1
\[ a_1 = \phi (a_3, a_7) \]
\[ \text{if } a_1 \leq n \]
B3
\[ a_7 = a_1 + 1 \]
B5
\[ \text{if } a_1 \leq 11 \]
B6
\[ D.1200 = a_1 + b_4 \]
\[ a_9 = D.1200 \ast c_5 \]
B7
\[ a_2 = \phi (a_1, a_9) \]
\[ \text{return } D.1201 \]

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.

SA construction algorithm is expected to insert as few \( \phi \) functions as possible to ensure the above properties.

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

**Dump file ccp.c.013t.cfg**

```plaintext
<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**

```plaintext
<bb 2>:
# 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>;
```

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

**Dump file ccp.c.013t.cfg**

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

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

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

```plaintext
<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>;
```

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

Dump file ccp.c.013t.cfg

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

Dump file ccp.c.017t.ssa

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

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

Copy Renaming

Input dump: ccp.c.017t.ssa

<bb 7>:
D.1201_10 = a_2;
return D.1201_10;

Output dump: ccp.c.022t.copyrename1

<bb 7>:
D.1201_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

Output dump: ccp.c.027t.copyprop1

<bb 6>:
D.1200,8 = a.1 + 2;
  a.9 = 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
  c.5 = 3; n.6 = c.5 + 2
B4
  a.1 = φ (a.3, a.7)
  if a.1 ≤ n.6

B5
  if a.1 ≤ 11

B6
  a.9 = a.1 + 5
B7
  a.2 = φ (a.1, a.9)
  return a.2

• No uses for variables a.3, b.4, c.5, and n.6
• Assignments to these variables can be deleted

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Loop Unrolling

Complete Unrolling of Inner Loops

Another Round of Constant Propagation

Second Example Program
Example Program 2

```c
int f(int b, int c, int n)
{
    int a;
    do {
        a = b+c;
    } while (a <= n);
    return a;
}
```

We use this program to illustrate the following optimizations:

- Partial Redundancy Elimination,
- Copy Propagation,
- Dead Code Elimination

**Compilation Command**

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

**Dump of Input to PRE Pass**

```
loop.c.091t.crited
<bb 2>:<bb 3>:
    a_3 = b_1(D) + c_2(D);
    if (a_3 <= n_4(D)) goto <bb 5>; else goto <bb 6>;
<bb 5>:
    goto <bb 3>;
<bb 6>:
    # a_6 = PHI a_3(6)
    return a_6;
```
Input and Output of PRE Pass

```c
loop.c.091t.crited

<bb 2>:

<bb 3>:
    a_3 = b_1(D) + c_2(D);
    if (a_3 <= n_4(D))
        goto <bb 5>;
    else goto <bb 6>;

<bb 5>:
    goto <bb 3>;

<bb 6>:
    # a_6 = PHI <a_3(6)>
    return a_6;
```

```c
loop.c.092t.pre

<bb 2>:
    pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
    a_3 = pretmp.2_7;
    if (a_3 <= n_4(D))
        goto <bb 5>;
    else goto <bb 6>;

<bb 5>:
    goto <bb 3>;

<bb 6>:
    # a_6 = PHI <a_3(6)>
    return a_6;
```

Copy Propagation after PRE

```c
loop.c.097t.copyprop4

<bb 2>:
    pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
    a_3 = pretmp.2_7;
    if (n_4(D) >= pretmp.2_7)
        goto <bb 4>;
    else goto <bb 5>;

<bb 4>:
    goto <bb 3>;

<bb 5>:
    # a_8 = PHI <pretmp.2_7(3)>
    a_6 = a_8;
    return a_8;
```

Dead Code Elimination

```c
loop.c.097t.dceloop1

<bb 2>:
    pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
    a_3 = pretmp.2_7;
    if (n_4(D) >= pretmp.2_7)
        goto <bb 4>;
    else goto <bb 5>;

<bb 4>:
    goto <bb 3>;

<bb 5>:
    # a_8 = PHI <pretmp.2_7(3)>
    a_6 = a_8;
    return a_8;
```

Redundant \( \phi \) Function Elimination and Copy Propagation

```c
loop.c.098t.dceloop1

<bb 2>:
    pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
    if (n_4(D) >= pretmp.2_7)
        goto <bb 4>;
    else goto <bb 5>;

<bb 4>:
    goto <bb 4>;

<bb 5>:
    # a_8 = PHI <pretmp.2_7(3)>
    a_6 = a_8;
    return a_8;
```

```c
loop.c.125t.phicprop2

<bb 2>:
    pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
    if (n_4(D) >= pretmp.2_7)
        goto <bb 4>;
    else goto <bb 3>;

<bb 4>:
    goto <bb 3>;

<bb 5>:
    # a_8 = PHI <pretmp.2_7(3)>
    a_6 = a_8;
    return a_8;
```
Final Assembly Program

```assembly
loop.c.125t.phicprop2

<bb 2>:
  pretmp.2_7 = c_2(D) + b_1(D);
  if (n_4(D) >= pretmp.2_7)
    goto <bb 4>;
  else
    goto <bb 3>;

<bb 3>:
  return pretmp.2_7;

<bb 4>:
  goto <bb 4>;
```

```c
loop.s

movl 8(%esp), %eax
addl 4(%esp), %eax
cmpl %eax, 12(%esp)
jge .L2
.repret
.L2:
.L3: jmp .L3
```

Why infinite loop?

Infinite Loop in Example Program 2

```c
int f(int b, int c, int n){ int a;
  do{
    a = b+c;
  } while (a <= n);
  return a;
}
```

The program does not terminate unless a > n

Conclusions

- GCC performs many machine independent optimizations
- The dumps of optimizations are easy to follow, particularly at the GIMPLE level
- 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)