Induction-variable Optimizations in GCC

程斌

bin.cheng@arm.com

2013.11

Outline

- Background
- Implementation of GCC
- Learned points
- Shortcomings
- Improvements
- Question & Answer
- References

Background – Induction variable

- Induction variable
  - Variables whose successive values form an arithmetic progression over some part of program, usually a loop.

```
int  a[100], i;
for (i = 0; i < 100; i++) {
    a[i] = 202 - 2 * i
}

/*Induction variables:
    i, &a[i], 2*i, 202-2*i
*/
```

Background – Identify induction variables

- Basic/Fundamental induction variable
  - Variables explicitly modified by a same constant amount during each iteration of a loop.
- Derived/General induction variable
  - Variables modified in more complex ways

```
int  a[100], i;
for (i = 0; i < 100; i++) {
    a[i] = 202 - 2 * i
}

/*Induction variables:
    i, &a(i), 2*i, 202-2*i
*/
```

Background – Induction-variable optimizations

- Strength reduction

```
int  a[100], i;
for (i = 0; i < 100; i++) {
    a[i] = 202 - 2 * i
}

//&a[i] ←→ a + 4 * i
//202 – 2 * i


int  a[100], i;
int *iv1 = a, iv2 = 202;

for (i = 0; i < 100; i++) {
    *iv1 = iv2;
    iv1 += 4;
    iv2 -= 2;
}
```

Background – Induction-variable optimizations

- Linear function test replacement

```
int  a[100], i;
int *iv1 = a, iv2 = 202;

for (i = 0; i < 100; i++) {
    *iv1 = iv2;
    iv1 += 4;
    iv2 -= 2;
}
```

```
int  a[100], i;
int *iv1 = a, iv2 = 202;

for (i = 0; iv2 != 2; i++) {
    *iv1 = iv2;
    iv1 += 4;
    iv2 -= 2;
}
```

Background – Induction-variable optimizations

- Removal of induction variables

```
int  a[100], i;
int *iv1 = a, iv2 = 202;

for (i = 0; iv2 != 2; i++) {
    *iv1 = iv2;
    iv1 += 4;
    iv2 -= 2;
}
```

```
int  a[100];
int *iv1 = a, iv2 = 202;

for (; iv2 != 2;) {
    *iv1 = iv2;
    iv1 += 4;
    iv2 -= 2;
}
```

Background – Induction-variable optimizations

- Unnecessary bounds checking elimination
  - refers to determine whether the value of a variable is within specified bounds in all of its uses in a program

```
var b: array[1..100,1..10] of integer;
   i, j, s: integer;
s := 0;
for i = 1 to 50 do
   for j = 1 to 10 do
      s := s + b[i,j]
```

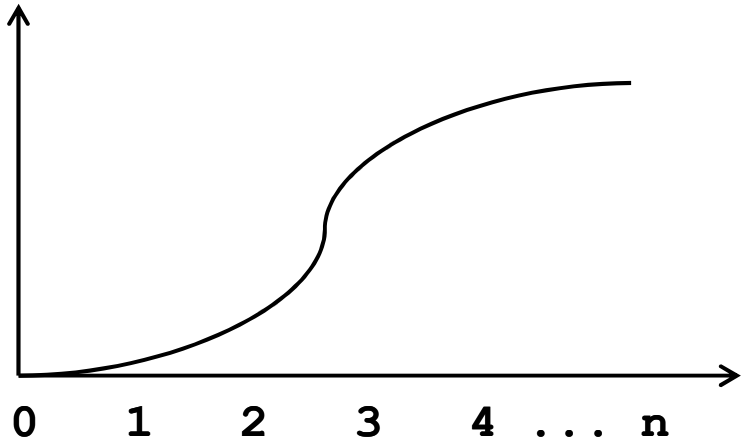
```
      i ← init
L1:   . . .
      if i < lo trap 6
      if i > hi trap 6
      use of i that must
         satisfy lo ≤ i ≤ hi
      . . .
      i ← i + 1
      if i <= fin goto L1
```

```
      if lo > init trap 6
      t1 ← fin min hi
      i ← init
L1:   . . .

      use of i that must
         satisfy lo ≤ i ≤ hi
      . . .
      i ← i + 1
      if i <= t1 goto L1
      if i <= fin trap 6
```

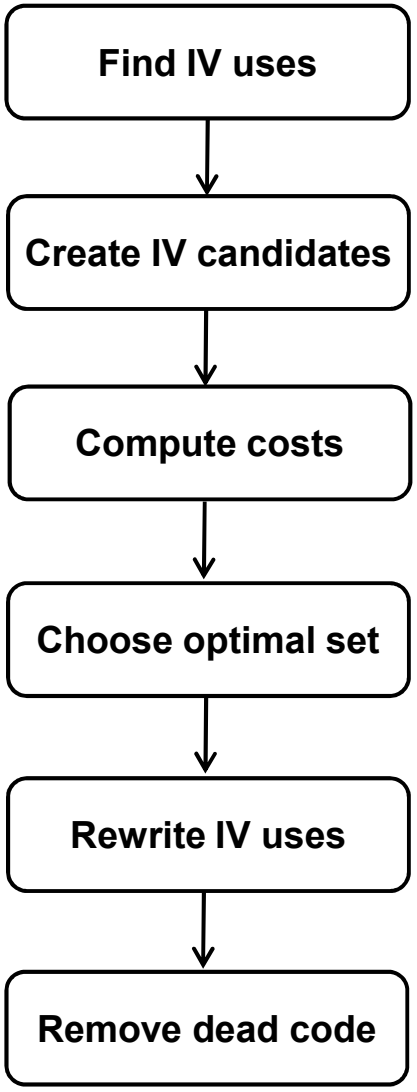
Implementation of GCC

- Identify induction variable
  - SSA form
  - Scalar evolution
    - implemented in tree-scalar-evolution.[ch]
    - analyzes the evolution of scalar variables in loop
  - Chain of recurrence
    - is a canonical representation of polyn1mials functions
    - can evaluate polynomials function at a number of points in an interval effectively
    - models induction variable analysis

f(x) = x^a + x^(a-1) + c
f(0), f(1), f(2), ...f(n)



Implementation of GCC – Unified algorithm




```
int  a[100], i;
for (i = 0; i < 100; i++) {
    a[i] = 202 - 2 * i
}
```

```
use 0: 202-2*i

use 1: &a[i]

use 2: i < 100
```

```
cand 0: <0, 1>

cand 1: <202, -2>

cand 2: <a, 4>

cand 3: <a-4, 4>
```

```
int  a[100];
int *iv1 = a, iv2 = 202;

for (; iv2 != 2;) {
    *iv1 = iv2;
    iv1 += 4;
    iv2 -= 2;
}
```

Implementation of GCC – Example

- IR before IVOPT

```
<bb 3>:
# i_14 = PHI <i_11(4), 0(2)>
i.0_4 = (unsigned int) i_14;
_5 = i.0_4 * 4;
_7 = a_6(D) + _5;
_8 = 101 - i_14;
_9 = _8 * 2;
*_7 = _9;
i_11 = i_14 + 1;
if (i_11 != 100)
  goto <bb 3>;
else
  goto <bb 5>;
```

- IV uses

```
use 0: <202, -2, _9>
use 1: <a, 4, _7>
use 2: <0, 1, i_11>
```

- IV candidates

```
cand 0: <0, 1, normal>
cand 1: <202, -2, normal>
cand 2: <a, 4, normal>
cand 3: <a-4, 4, before>
cand 4: <a, 4, after>
```

Implementation of GCC – Example

- Representation

	use 0	use 1	use 2
cand 0	202-2*iv_cand	a + 4 * iv_cand	iv_cand != 100
cand 1	iv_cand	a+2*(202-iv_cand)	iv_cand != 2
cand 2	NA	MEM[iv_cand]	iv_cand != a+400
cand 3	NA	MEM[iv_cand]	iv_cand != a+400
cand 4	NA	MEM[iv_cand]	iv_cand != a+400

- Costs

	use 0	use 1	use 2
cand 0	8	8	0
cand 1	0	26	0
cand 2	NA	6	0
cand 3	NA	2	0
cand 4	NA	2	0

Implementation of GCC – Example

- Choose optimal iv set

```
use:0  → cand:0  → cand:1  → cand:1
use:1  → cand:0  → cand:0  → cand:4
use:2  → cand:0  → cand:0  → cand:1
```

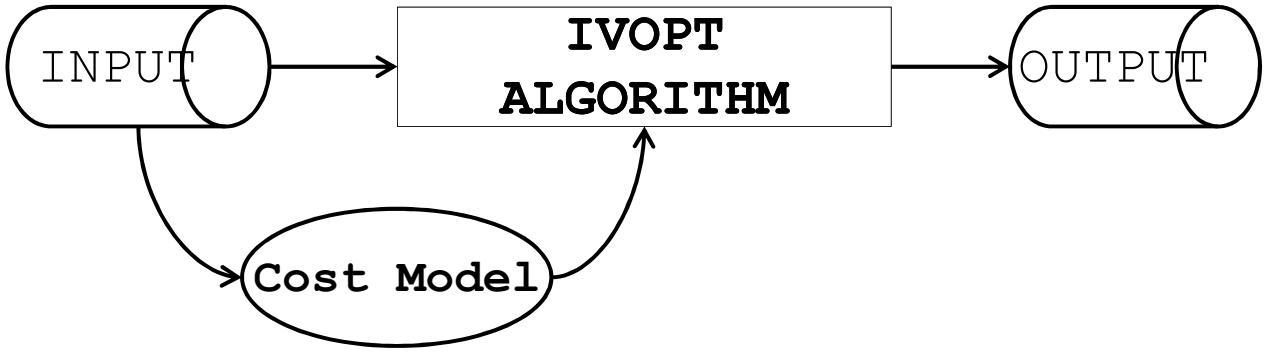
- Rewrite iv uses and remove dead code

```
<bb 3>:
# i_14 = PHI <i_11(4), 0(2)>
i.0_4 = (unsigned int) i_14;
_5 = i_14 * 4;
_7 = a_6(D) + _5;
_8 = 101 - i_14;
_9 = _8 * 2;
*_7 = _9;
i_11 = i_14 + 1;
if (i_11 != 100)
  goto <bb 3>;
else
  goto <bb 5>;
```

```
<bb 2>:
iv.9_2 = (unsigned int) a_6(D);
<bb 3>:
# i_14 = PHI <i_11(4), 0(2)>
# iv.6_3 = PHI <iv.6_2(4), 202(2)>
# iv.9_1 = PHI <iv.9_3(4),iv.9_2(2)>
_5 = i_14 * 4;
_7 = a_6(D) + _5;
_8 = 101 - i_14;
_9 = (int) iv.6_3;
_16 = (void *) iv.9_1;
MEM[base: _16, offset: 0B] = _9;
iv.9_3 = iv.9_1 + 4;
iv.6_2 = iv.6_3 - 2;
i_11 = i_14 + 1;
if (iv.6_2 != 2)
  goto <bb 3>;
else
  goto <bb 5>;
```

Implementation of GCC – Learned points

- Infrastructure



- Unified algorithm
- Cost Model
  - rtx cost, like “+,-,*,/,%,<<,...”
  - address cost
- Context information
  - Loop invariant
  - Register pressure
- Interact with other optimizations

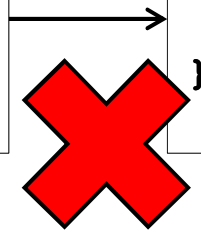
addressing mode
[reg]
[reg + reg]
[reg + reg<<const]
[reg + const]!
[reg], #const

Shortcomings – implementation

- fine tuned only for x86/x86_64
  - scaled addressing mode not supported for ARM, etc.
  - support of auto-increment addressing mode is weak for ARM

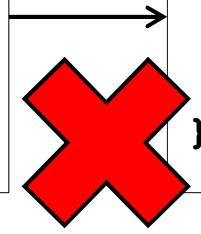
addressing mode
[reg]
[reg + reg]
[reg + reg<<const]
[reg + const]!
[reg], #const

```
int  a[100], i;
for (i = 0; i < 100; i++){
    tmp = a + i*4
    MEM[tmp] = 202 - 2*i
}
```



```
int  a[100], i;
for (i = 0; i < 100; i++){
    MEM[a+i<<2] = 202 - 2*i
}
```

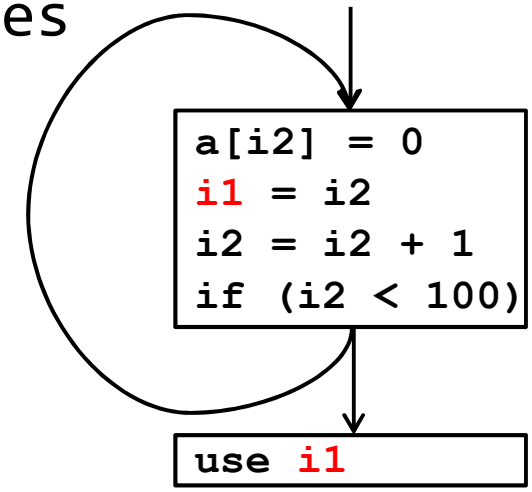
```
int  a[100];
int *iv1 = a, iv2 = 202;
for (; iv2 != 2;) {
    *iv1 = iv2;
    iv1 += 4;
    iv2 -= 2;
}
```

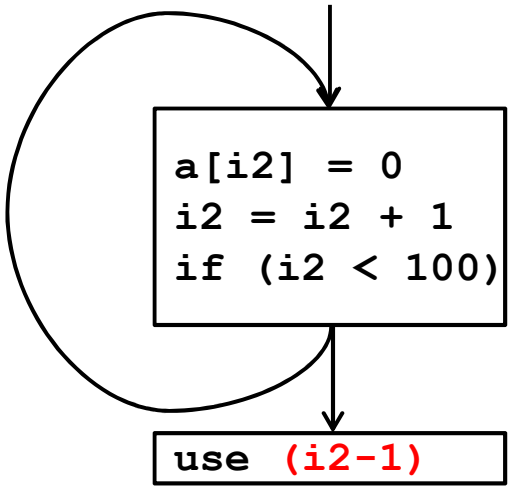


```
int  a[100];
int *iv1 = a, iv2 = 202;
for (; iv2 != 2;) {
    MEM([iv1], #4) = iv2;
    iv2 -= 2;
}
```

Shortcomings – implementation

- iv uses outside of loop are not handled along loop exit edges





- induction variables in both branches of IF-statement are not recognized

```
int  a[100], i = 0;
while (i < 100) {
  if (i % 2 == 0)
    a[i] = 0;  i++;
  else
    a[i] = 1;  i++;
}
```

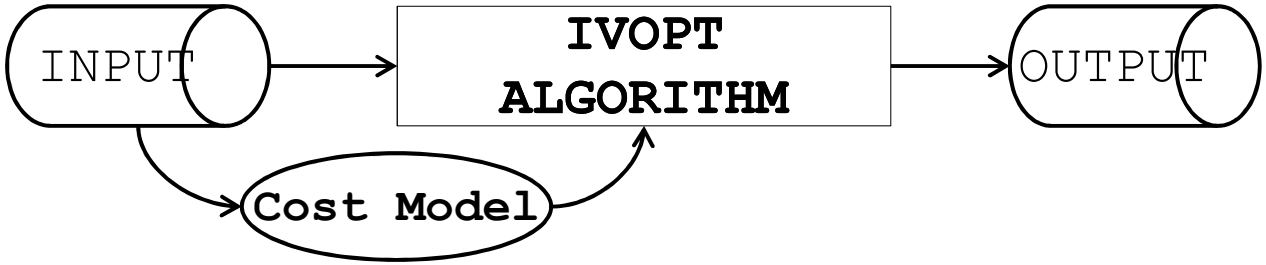
Shortcomings – implementation

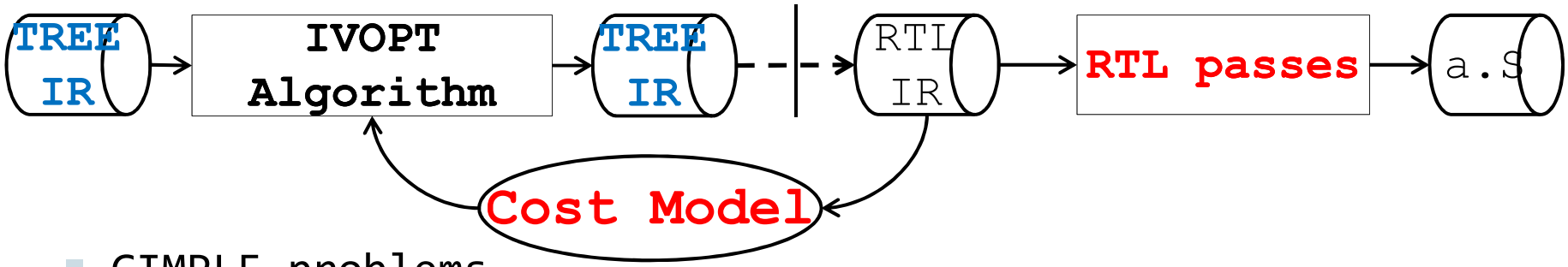
- complex address expression
  - hard to process
  - inaccurate cost

```
&MEM[ptr+offset]
&arr[index].y
&MEM[ptr+offset] +/- step
&arr[index].y    +/- step

ptr + offset
arr + index*obj_size + y_offset
ptr + offset                        +/- step
arr + index*obj_size + y_offset     +/- step

ptr + const_1
arr + const_2
```

Shortcomings – infrastructure

- GCC's two IRs: GIMPLE(IVOPT) & RTL
  - Inaccurate rtx cost
  - "WRONG" address cost model for ARM
  - Conflict with RTL optimizations, like loop unrolling





- GIMPLE problems
  - Conflict with GIMPLE optimizations, like DOM
  - Association of loop invariant in IVOPT

```
x=a+b+c → t=a+b; x=t+c
    or
→ t=a+c; x=t+b
```

Improvements – Work & patches

- Support scaled addressing mode for architectures other than x86/x86_64.
  - http://gcc.gnu.org/ml/gcc-patches/2013-08/msg01642.html
  - http://gcc.gnu.org/ml/gcc-patches/2013-09/msg01927.html
- Simplify address expressions for IVOPT
  - http://gcc.gnu.org/ml/gcc-patches/2013-11/msg00537.html
  - http://gcc.gnu.org/ml/gcc-patches/2013-11/msg01075.html
- Fix wrong address cost for auto-increment addressing mode
  - http://gcc.gnu.org/ml/gcc-patches/2013-11/msg00156.html
- Expedite the use of auto-increment addressing mode
  - http://gcc.gnu.org/ml/gcc-patches/2013-09/msg00034.html
  - more patches coming...
- Compute outside loop iv uses along exit edges
  - http://gcc.gnu.org/ml/gcc-patches/2013-11/msg00535.html
- Improve the optimal iv set choosing algorithm
  - patches coming...
- Miscellaneous fixes for IVOPT
  - ...

Improvements – Work & patches

- Following work
  - Fix address cost mode and RTL passes, e.g. fwprop_addr

```
arm_arm_address_cost (rtx x)
{
  enum rtx_code c  = GET_CODE (x);

  if (c == PRE_INC || c == PRE_DEC || c == POST_INC || c == POST_DEC)
    return 0;
  if (c == MEM || c == LABEL_REF || c == SYMBOL_REF)
    return 10;

  if (c == PLUS)
    {
      if (GET_CODE (XEXP (x, 1)) == CONST_INT)
        return 2;

      if (ARITHMETIC_P (XEXP (x, 0)) || ARITHMETIC_P (XEXP (x, 1)))
        return 3;

      return 4;
    }

  return 6;
}
```

addr mode	[reg]	[reg+reg]	[reg+reg<<i]	[reg+offset]	[reg], #off
cost	6	4	3	2	0

Improvements – Benchmark data

- Performance
  - Coremark
  - EEMBC_v1
    - automotive, office, consumer, telecom
  - Spec2000
- Code size
  - CSIBE

Question and Answer

Thank You!

References

- source code & internal & mailing list
  - gcc.gnu.org
- Advanced compiler design and implementation
- Symbolic Evaluation of Chains of Recurrences for Loop Optimization, etc.