Straight-Line Strength Reduction in GCC

Bill Schmidt, Ph.D.
PowerLinux Toolchain Team
IBM Linux Technology Center

Motivation

- Why a talk on strength reduction?
  - Missing piece in GCC, other than induction variables
  - Several bug reports lamenting lack of strength reduction in blocks
  - Surprisingly interesting topic (at least to me)
- Ongoing project
  - New pass added a couple of weeks ago
  - Only handles simplest forms so far
  - Rest has been prototyped
  - Input welcome!

Motivation: Examples from Bugzilla (PR22586, 2005)

```
int
foo (int a[], int b[], int i)
{
  a[i] = b[i] + 2;
  i++;
  a[i] = b[i] + 2;
  i++;
  a[i] = b[i] + 2;
  i++;
  a[i] = b[i] + 2;
  i++;
  return i;
}
```

```
D.2001_2 = (long unsigned int) i_1(D);
D.2002_3 = D.2001_2 * 4;
D.2003_5 = a_4(D) + D.2002_3;
D.2004_7 = b_6(D) + D.2002_3;
D.2005_8 = *D.2004_7;
D.2006_9 = D.2005_8 + 2;
*D.2003_5 = D.2006_9;
i_10 = i_1(D) + 1;
D.2001_11 = (long unsigned int) i_10;
D.2002_12 = D.2001_11 * 4;
D.2003_13 = a_4(D) + D.2002_12;
D.2004_14 = b_6(D) + D.2002_12;
D.2005_15 = *D.2004_14;
D.2006_16 = D.2005_15 + 2;
*D.2003_13 = D.2006_16;
i_17 = i_1(D) + 2;
D.2001_18 = (long unsigned int) i_17;
D.2002_19 = D.2001_18 * 4;
D.2003_20 = a_4(D) + D.2002_19;
D.2004_21 = b_6(D) + D.2002_19;
D.2005_22 = *D.2004_21;
D.2006_23 = D.2005_22 + 2;
*D.2003_20 = D.2006_23;
i_24 = i_1(D) + 3;
...
```

Motivation: Examples from Bugzilla (PR32120, 2007)

```
int f(int a)
{
  int c = a+2;
  int d = c*2;
  int e = a*2;
  int f = e+4;
  return d == f;
}
```

```
c_2 = a_1(D) + 2;
d_3 = c_2 * 2;
e_4 = a_1(D) * 2;
f_5 = e_4 + 4;
D.2005_6 = d_3 == f_5;
D.2004_7 = (int) D.2005_6;
return D.2004_7;
```

No optimization at all...

Motivation: Examples from Bugzilla (PR46556, 2010)

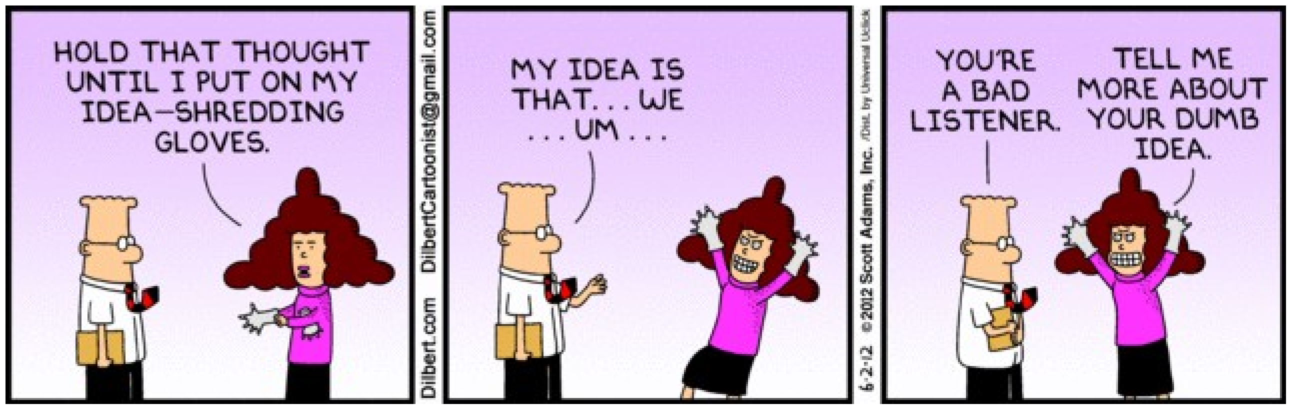
```
struct x
{
  int a[16];
  int b[16];
  int c[16];
};

extern void foo (int, int, int);

void
f (struct x *p, unsigned int n)
{
  foo (p->a[n], p->c[n], p->b[n]);
}
```

```
addi 8,4,32
addi 10,4,16
mr 9,3
sldi 4,4,2
sldi 8,8,2
sldi 10,10,2
std 0,16(1)
stdu 1,-112(1)
lwax 3,3,4
lwax 5,9,10
lwax 4,9,8
bl foo
```

Why develop a separate pass?



- Common practice: strength reduction as part of PRE
  – Make use of existing value numbering
- Disadvantages
  – Lookup problem
  – Advantageous to view chains of related candidates together
  – Strength reduction needs special handling for casts
  – PRE insertion is complex

The value number lookup problem

One form of strength reduction candidate:

S1: $X = (B + i) * S$

B = base variable, i = constant index, S = stride (assume constant for now)

Need a previous statement of the form:

S0: $Y = (B + i') * S$

i' = possibly different constant index

Can replace S1 with:

S1': $X = Y + c$, $c = (i – i') * S$

<u>Problem</u>: Any arbitrary value of i' will do. What avail expr should we look up?

Common case: $(B + (i - 1)) * S$

but very limited results.

Kinds of strength reduction opportunities

- Opportunities where multiplies are explicit in GIMPLE
  - Stride is a known constant value
  - Stride is an unknown, but still constant, value
- Implicit multiplies in addressing expressions
  - Recall the case of multiple array fields in a structure
- Conditional increments
  - Variant of both forms of explicit multiplies
  - From PR35308:

```
a[i] = i*g;
if (...)
  i++;
a[i] += ...
... = i*g;
```

Goals of this work

- Single framework flexible enough to handle all these types
  - Common features of all abstracted into the candidate table
- Don't disturb existing strength reduction of ivars
  - Late pass following loop optimizations
- Efficiency
  - Single forward pass over GIMPLE representation
- Profitability
  - Use cost model that reflects target
- Efficacy
  - As many candidates as possible (dominator paths)
  - Catch opportunities exposed by earlier transformations

Background concepts

- Basic block: maximal chunk of branch-free code (in or out)
- Control flow graph: Blocks connected by possible execution paths
- Dominance: Block A <u>dominates</u> block B iff executing block B implies that block A has already been executed at least once.
  - Dominance relation is a tree; <u>immediate dominator</u> is immediate predecessor in this tree.
- Static Single Assignment (SSA) form:
  - Each "name" is assigned to exactly once (X ==> X_1, X_2, …)
  - Join points handled specially (more later)
  - Greatly simplifies available expressions over dominator paths

Candidate table

- Central data structure for the pass
  - Essentially a specialized kind of value numbering
- Four kinds of candidates
  - CAND_MULT: Value of the form (B + i) * S
  - CAND_ADD: Value of the form B + (i * S)
  - CAND_REF: For implicit multiplies in addressing expressions
  - CAND_PHI: For conditional candidates
- All use B = base, i = index, S = stride (known or unknown)
- Type: For when casts intervene
- Basis: A previous candidate allowing strength reduction
- Dependent: Later candidate for which this is a basis
- Sibling: Candidate having the same basis

Explicit multiplies

- For CAND_MULT we seek:
  - S0: Y = (B + i') * S
  - S1: X = (B + i) * S
- So we can replace S1 by:
  - S1': X = Y + (i – i') * S (folded to extent possible)
- For CAND_ADD:
  - S0: Y = B + (i' * S)
  - S1: X = B + (i * S)
- So we can replace S1 by:
  - S1': X = Y + (i – i') * S (ah, the same thing!)
- In both cases, S0 is the basis for S1 (and S1 is a dependent of S0).
- We call (i – i') the <u>increment</u> of S1 with respect to its basis S0.

Constructing the candidate table

- Statements are visited in dominance order
- Any statement that <u>can contribute</u> to a CAND_MULT or CAND_ADD gets at least one entry
  - Add, ptr add, subtract, multiply, negate, copy, type conversion
- Multiple interpretations per statement
  - Separate entries, chained together
  - Add of two SSA names: either could be the base
  - Multiply of two SSA names: either could be the base or stride
  - Copy or conversion:
    - CAND_MULT: X = (B + 0) * 1
    - CAND_ADD: X = B + (0 * 1)
- Potential dead-code savings identified

Constructing the candidate table (cont.)

- Algebraic rules propagate values forward to produce more complex candidates.
  - Suppose we encounter X = Y * c.
  - We find Y represented in the candidate table by:
    - Y = (B + i') * S, S constant
  - We can add this representation of X to the table:
    - $X = (B + i') * (c * S)_{fold}$
- Or an “undistribution” example:
  - X = Y * c
  - Y = (B + i') * S, S constant, c = kS for integer k
  - => $X = (B + (i' + k)_{fold}) * S$

Finding a basis

- Maintain mapping from base names to chains of candidates
  - I.e., “B” in all our examples
- Basis must satisfy these properties:
  - Same kind (CAND_MULT, etc.)
  - Same base name B and stride S
  - Compatible types (see slide 3)
  - Dominates the candidate
- More than one basis may exist
  - For now, most immediately dominating basis is chosen
  - Limits introduced lifetimes
    - But subsequent optimizations often defeat this :(
  - Other bases can be found transitively

Profitability: The easy cases

- Trees of related candidates are considered together.
- Recall these all have same base B and stride S.
- Known stride == always profitable (well, not unprofitable)
  - CAND_ADD doesn't apply: $B + (i * S)_{fold}$ is already simplified
  - CAND_MULT:

M = B + i';	M = B + i';
Y = M * S;	Y = M * S;
N = B + i;	[N = B + i;]
X = N * S;	X = Y + c;

where c = (i – i') * S.

- When S is unknown, we are still ok if the increment (i – i') is -1, 0, or 1.

Profitability: Unknown strides with nontrivial increments

- Increment = (i – i'). Assume | i – i' | > 1.
- Candidate will be replaced with B + $((i - i')_{fold} * S)$.
  - The second addend may not yet exist, in which case it must be inserted ("initializer").
  - Without savings from dead code, this can worsen performance.
  - If same initializer is needed by several related candidates, the cost is reduced.
- Table of increments created for chain of related candidates
  - Each increment checked for total profitability (cost of inserting initializer, dead code savings, replacement savings)
  - All-or-nothing decision for related candidates with same increment
  - Initializer inserted at nearest common dominator position

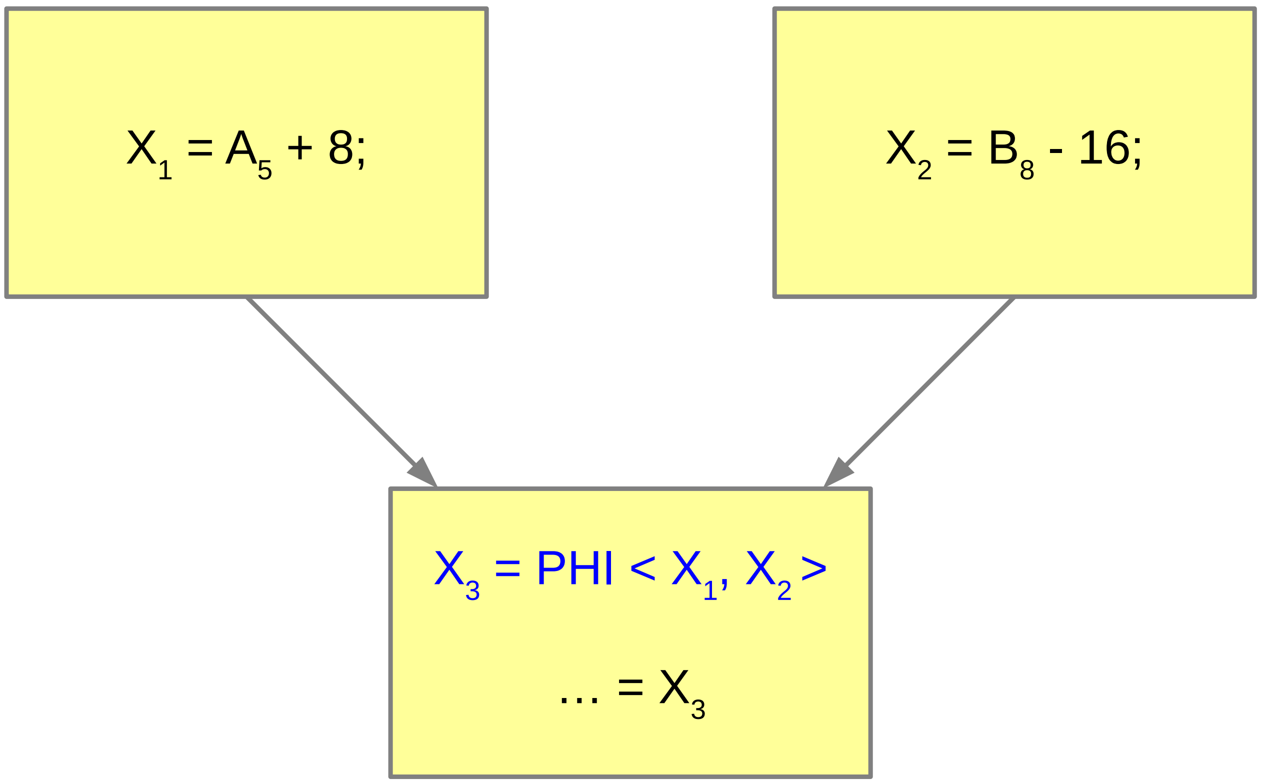
Measures of profitability

- Optimizing for size
  - Cost function is the delta of the size of instructions
  - Total over all candidates with same increment
- Optimizing for speed
  - Cost function is target-specific model of execution time
  - Must improve performance along at least one path
  - More conservative requirement: must improve performance along all paths
- Problem: size/speed is a per-block decision
  - For now: Block containing ultimate basis determines which

Limitations of increment model

- Current prototype uses model just described
- More involved analysis can provide improvements
  - Increment for each candidate is determined by its chosen basis
  - Chains of related candidates enable multiple choices for potential increments (choose different basis)
  - Best would be to search all potential increment spaces for a group of related candidates and choose most profitable
- Target capabilities change profitability
  - Targets having a shift-add instruction should be biased towards power-of-2 increments (thanks to Richard Earnshaw)

More on SSA form: PHI statements



Conditional candidates

$X_0 = \ldots;$
$A_4 = X_0 * 5;$ (MULT B:X_0, i:0, S:5)
if (…)
  $X_1 = X_0 + 1;$ (ADD B:X_0, i:1, S:1)
$X_2 = PHI < X_0, X_1 >;$ (PHI B:X_0, i:0, S:1)
$X_3 = X_2 + 1;$ (ADD B:X_2, i:1, S:1)
$A_5 = X_3 * 5;$ (MULT B:X_2, i:1, S:5)

$X_0 = \ldots;$
$A_4 = X_0 * 5;$
if (…) {
  $X_1 = X_0 + 1;$
  $T_6 = A_4 + 5;$
}
$X_2 = PHI < X_0, X_1 >;$
$T_7 = PHI < A_4, T_6 >;$
$X_3 = X_2 + 1;$
$A_5 = T_7 + 5;$

Note: A_5 can't use A_4 as a basis with our current definitions, as they don't have the same base name. We will call it a “hidden basis.”

- Base name of CAND_MULT A_5 is defined by a PHI
- That PHI's arguments have the same “derived base name”:
  - Argument is a CAND_ADD with stride 1 => base name
  - Otherwise, argument itself

Conditional candidate transformations

- When a PHI is encountered while building the candidate table, determine if it has a derived base name B.
  - If so, add CAND_PHI (B, 0, 1) to the table.
- During transformation phase:
  - Find the hidden basis as per previous slide.
    - Strides must still be identical.
  - Introduce a new PHI in the same block with the feeding PHI.
    - “Phi basis.”
  - Translate argument from feeding PHI to argument for phi basis.
    - Argument is a CAND_ADD (B, i, 1) => create an add of hidden basis with i * S.
    - Argument is the derived base name => copy the hidden basis.
  - Replace CAND_MULT with an add relative to the phi basis.

Profitability for conditional candidates

- Conditional candidates with known strides are no longer always profitable to replace.
  - Cost of compensation code can offset gains if not enough code goes dead.
- Conditional candidates with unknown strides are considered together with their foregoing related candidates.
  - But a phi breaks the chain, so subsequent candidates are now based off the candidate that had a hidden basis.
    - So phis are at the frontier of the candidate tree.
  - Increment analysis proceeds as before, but a phi is only replaced if every increment needed for an argument is profitable to replace.

Implicit multiplies in addressing expressions

- Recall the code from PR46556 (slide 5):

```
struct x { int a[16];  int b[16]; int c[16]; };

void f (struct x *p, unsigned int n) {
   foo (p->a[n], p->c[n], p->b[n]);
}
```

Implicit multiplies in addressing expressions

- p->a[n] = *(p + offsetof(a) + (n * sizeof(int)))
- General form: *(T1 + C1 + (T2 + C2) * C3 + C4)
  - T1, T2: arbitrary trees
  - C1, C2, C3, C4: arbitrary constants
- We want to distribute the multiply and replace the original reference by:
  - $*(T1 + (T2 * C3), (C1 + (C2 * C3) + C4)_{fold})$
- This moves the [n] into the base address calculation so similar expressions differ only in the offset.
  - p->a[n].x and p->b[n+4].y would differ only in offset
- But only make the transformation if it's profitable.

Reference candidates

- When a component reference expression is encountered in GIMPLE, use get_inner_reference to look for pattern:
  - base = *(T1 + C1), C1 may be zero
  - offset = (T2 + C2) * C3, C2 may be zero
  - bitpos = C4 * BITS_PER_UNIT, C4 may be zero
- If found, create a CAND_REF candidate with:
  - B = T1
  - $i = (C1 + (C2 * C3) + C4)_{fold}$
  - S = T2 * C3
- Note B and S can now be arbitrary trees.
- Basis for candidate again must match B and S.

Transforming reference candidates

- As with other kinds, chains of related CAND_REFs are processed together.
- If there is at least one basis-dependent relation, all candidates are transformed (including the ultimate basis).
- This allows the RTL addressing code to recognize these addresses differ only by a constant address.

```
addi 8,4,32
addi 10,4,16
mr 9,3
sldi 4,4,2
sldi 8,8,2
sldi 10,10,2
std 0,16(1)
stdu 1,-112(1)
lwax 3,3,4
lwax 5,9,10
lwax 4,9,8
bl foo
```

```
sldi 4,4,2
add 9,3,4
std 0,16(1)
stdu 1,-112(1)
lwax 3,3,4
lwa 5,64(9)
lwa 4,128(9)
bl foo
```

Status and future work

- Stage 1 complete: explicit multiplies with known strides
- Handling of reference candidates submitted for approval
- The rest has been implemented in a prototype but not yet submitted.
- Remaining issues and open questions
  - Improve the increment selection for explicit multiplies with unknown strides
    - Wider search
    - Better choices for targets with shift-add
  - Look for more un-distribution opportunities
  - Handle B = &var
  - Register pressure concerns?

Conclusions and acknowledgments

- Conclusions
  - Strength reduction proved more interesting than I thought
  - Goals of project are all on target
  - Open to ideas for improvements!
- Acknowledgments
  - Richard Guenther
  - Richard Earnshaw
  - Andrew Pinski
  - Steven Bosscher
  - Alan Modra
  - Xinliang (David) Li