Local optimizations

- performed strictly within a basic sequential block
- simpler analysis: every statement in the block runs in one specific order every time the block is invoked
- can't use wider context (e.g. which variables within block are also used before/after the block)
- can resolve constant expressions
- can re-use previously computed values
- can simplify expressions through by applying identities
- can rewrite expressions for greater parallelization

Local value renumbering (LVN)

- assign unique integer id to each value computed in block
- will use to refer to previously computed values
- expressions can be given the same id # if they have provably equal values for all operands of the expressions
- go through list of statements of form T = L Op R, and if expression previously computed then replace RHS with the LVN from the previous instance

Numbering algorithm

- Given block with n operations $T_i = L_i \ OP_i \ R_i$
- Maintain table of known LVNs so far

```
for i = 0..n-1:
    lookup LVN for Li, Ri
        new LVNs assigned on first use of vars, literals
    if expr "Li OPi Ri" already in table
        replace Ti = expr with Ti = expr's LVN from table
    else Get new LVN, v, and insert expr,v in table
    record Ti's LVN in table
```

Example

- first statement
  - first time for x, y, new expr x-y, assign LVNs
- second statement
  - looks up w, new z, new expr
- third statement
  - new expr (different x LVN)
- fourth statement
  - expr is same as second statement, re-use LVN

Original code

```
w = x - y
x = w + z
y = x - y
z = w + z
```

Showing LVNs

```
w(2) = x(0) - y(1)
x(4) = w(2) + z(3)
y(5) = x(4) - y(1)
z(4) = w(2) + z(3)
```

Table lookups, commutative ops

- use hash table, key being combination L(LVN), OP, R(LVN)
- hash function can be tweaked to recognize expressions on commutative ops as the same, e.g. L,+,R hashes the same as R,+,L
- thus LVNs automatically recognizing that x=y+z can be treated the same as x=z+y

Constant folding

- during LVN numbering, can also evaluate constant expressions
  - x = 3 + 5
- gets re-written as
  - x = 8
- LVN for x gets an annotation that it is a constant
- thus can also “fold” later constant expressions that use x
- folding would take place right after the L/R lookup at the beginning of the algorithm

Applying identities

- during LVN algorithm can also apply any identities involving specific literals, simplifying expressions
  - x + 0, 0 + x becomes x
  - x * 1, 1 * x becomes x
  - x * 0, 0 * x becomes 0
  - x^1 becomes x
  - x^0 becomes 1
  - x << 0, x >> 0 becomes x
  - $x * 2^n$ becomes x << n
  - x AND x becomes x
  - etc

Naming implications

- simplest approach is SSA-style, use a new variable name for each LHS target, consider problem below:
  - v = w + x
  - y = w + x // can re-write as y = v
  - v = 0
  - z = w + x // could have re-written as z = v, but over-wrote v
- otherwise need to be very careful about use of our hash table records, since LVN values associated with variables can change over time

Indirection and ambiguity (again!)

- as discussed earlier, pointers and array indexing can complicate attempts to identify LVNs, e.g.
  - a = b + c
  - *ptr = 5
  - d = a
  - e = b
  - f = c
- the pointer use *could* modify a, b, or c
- compiler would have to proceed as if each of them has changed, even though *at most* one of them actually would

Parallelization opportunities

- many processors have multiple adders, and can carry out two (or more) arithmetic operations in a single cycle
- permits parallelization of parts of expression evaluation
  - w + x + y + d // usually 3 cycles, for the 3 additions
- could parallelize:
  - adder1 does tmp1 = (w+x) while adder2 does tmp2 = (y+d)
  - then one of them does tmp1+tmp2
- thus just 2 cycles
- bigger parallelization potential for larger expressions

Rewriting and parallelizing

- For expressions that use a single operator type, that is both commutative and associative, we can rewrite operands in any order
- provides lots of opportunities to improve
  - x + 3 + y + 9 + z + 200
  - default sequential handling left-to-right, 5 cycles
- group the constants, apply identities, and restructure
  - (x + y) + (z + 212)
  - done in 2 cycles using 2 adders

Tree-balancing

- think of expression in abstract syntax tree form
- we want to balance the tree, minimizing its height
- expressions represented as sequences of T = L op R
- need to know where a value computed earlier is used later (i.e. LVNs), so build these dependencies into the tree
- will try to parallelize across instruction sequences

Tree balancing process

- don't want tree revision to change any observable values, to be any longer than original, nor to look outside the block
- build the dependency tree
- try to re-balance it
  - find roots of relevant subtrees, whose operations consist of just one form of associative, commutative operator
- re-write the transformed code
  - apply constant folding, identities as we re-write

Tree balancing approach

- assuming we've identified roots of target trees, and ordered by precedence of the tree operators (highest first)
- as we process statements, we'll queue up values (variable names and literals) for later processing
- as we re-write the trees, we rank elements to ensure code that calculates value X is output before code that uses X
  - constants get rank 0, variables get rank 1, rank for expressions is the sum of their subtree ranks
  - lower-ranked terms get produced before higher-ranked terms

Tree balancing algorithm

- assuming we've identified the roots of relevant subtrees (i.e. ops of a single type, commutative and associative)
- root nodes initially assigned rank -1
  - for each root node, R, call Balance(R)
- Balance(node n) // n represents T = L op R
  - if n's rank is -1 (i.e. not yet processed):
    - Q = { }   // queue of values used in the subtree
    - rank(T) = flatten(L,Q) + flatten(R,Q)
    - rebuild(n, Q, op) // writes balanced version of the operands

Flatten(n,Q)

- flatten adds operands from the subtree to the queue
- flatten returns the rank of the subtree
- flatten(node n, queue Q) // node is a value or an op
  - if n is a constant: assign rank 0, enqueue
  - elseif n is a previously assigned var: assign rank 1, enqueue
  - elseif n is a root: call Balance(n), enqueue
  - else n is operator node, with L and R operands
    - call flatten(L,Q), flatten(R,Q), rank is sum
  - return rank of n

Rebuild(root,Q,op)

- called after Balance has put the operands for op into Q

```
while Q not empty
  pull next 2 args, L, R, from Q // just binary operators so far
  if both are constants:
     calculate result
     if Q is now empty
          emit code: “root = result”, assign 0 as root's rank
     else enqueue result with a rank of 0
  // else case on next slide
```

Rebuild(n,Q,op) continued

```
else // at least one is not a constant
   if Q is now empty result = n,
   else result = generateNewName()
   emit code “result = L op R”, rank is rank(L) + rank(R)
   if Q isn't empty yet then enqueue result
      // n is a subtree, so its computed result must be getting used later
```

Example

c = a + b

e = b + d

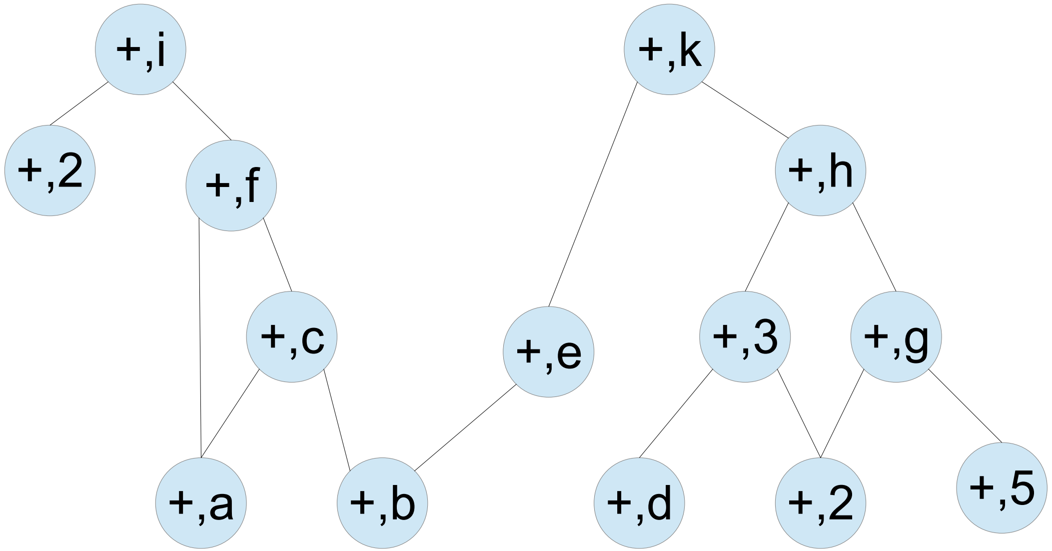
f = a + c

g = 2 + 5

h = 3 + g

i = 2 + f

k = e + f



ideally: re-group k's 3,2,5 into a single constant 10,
and restructure trees to be height 3 instead of 4

Balance(i)

- flatten(2) + flatten(f)
  - 2 is const, rank 0, gets enqueued
- flatten(f) calls flatten(a) + flatten(c)
  - var a, rank 1, gets enqueued
- flatten(c) calls flatten(a) + flatten(b)
  - vars a and b, each rank 1, get enqueued
- rebuild(i, [b,a,a,2], +) emits
  - `tmp1 = b + a`
  - `tmp2 = a + 2`
  - `i = tmp1 + tmp2`

enqueue enqueues by rank,
lower ranks go in front of higher
new values in front of old (of equal rank)

Balance(k)

- flatten(e) + flatten(h)
- flatten(e) calls flatten(b), flatten(d)
  - vars b, d get rank 1, enqueued
- flatten(h) calls flatten(3), flatten(g)
  - const 3 gets rank 0, enqueued
  - flatten(g) calls flatten(2), flatten(5)
  - consts 2,5 get rank 0, enqueued
- rebuild(k, [5,2,3,d,b], +) folds the constants and emits
  - `tmp3 = 10 + d`
  - `k = b + tmp3`