

CS3300 - Compiler Design

Basic block optimizations

V. Krishna Nandivada

IIT Madras

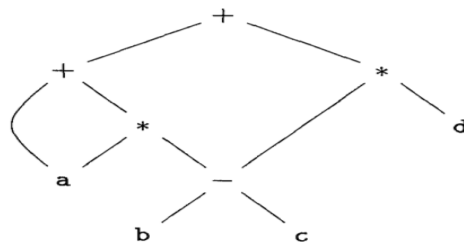
- It is a linear piece of code.
- Analyzing and optimizing is easier.
- Has local scope - and hence effect is limited.
- Substantial enough, not to ignore it.
- Can be seen as part of a larger (global) optimization problem.



DAG representation of basic blocks

Recall: DAG representation of expressions

- leaves corresponding to atomic operands, and interior nodes corresponding to operators.
- A node N has multiple parents - N is a common subexpression.
- Example: $(a + a * (b - c)) + ((b - c) * d)$



DAG construction for a basic block

- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s , of the operands used by s .
- Node N is labeled by the operator applied at s , and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block;



Optimizations on the DAG

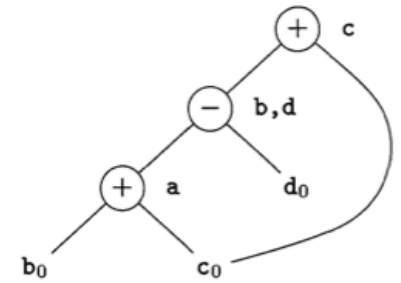
- Common subexpression elimination.
- Eliminate dead code.
- Code reordering.
- Algebraic optimizations.



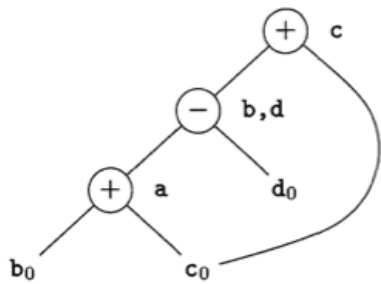
Construct the DAG. Example

```

a = b + c
b = a - d
c = b + c
d = a - d
    
```



Example (contd)



```

a = b + c
d = a - d
c = d + c
    
```

```

// if b is live
b = d
    
```

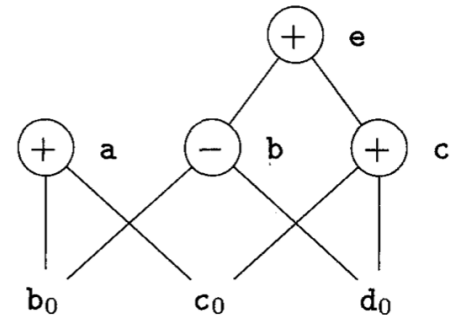
Q: How to know if b is live after the basic block?



Limitations of the DAG based CSE

```

a = b + c
b = b - d
c = c + d
e = b + c
    
```

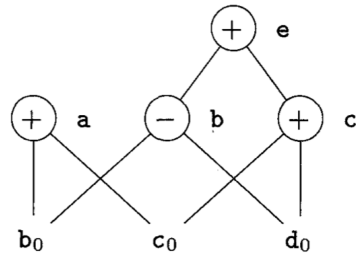


- The two occurrences of the sub-expressions $b + c$ computes the same value.
- Value computed by a and e are the same.
- How to handle the algebraic identities?
- Q: Do the sub-expressions always compute the same value?



Dead code elimination

- Delete any root from DAG that has no ancestors and is not live out (has no live out variable associated).
- Repeat previous step till no change.



- Assume a and b are live out.
- Remove first e and then c .
- a and b remain.



CSE via Algebraic identities

- Recall: In common sub-expression elimination, we want to reuse nodes that compute the same value.
- Recall: We mainly focussed on syntactic similarities.
- Q: Can we go beyond that?



Similarities in the semantics - identity, inverse, zero

$$x + 0 = 0 + x = x$$

$$x * 1 = 1 * x = x \quad \text{identity, examples?}$$

$$a \ \&\& \ \text{true} = \text{true} \ \&\& \ a = a$$

$$a \ || \ \text{false} = \text{false} \ || \ a = a$$

$$x * 0 = 0 * x = 0$$

$$0 / x = 0$$

Goal: apply arithmetic identities to eliminate computation.



Similarities in the semantics - strength reduction

$$x^2 = x * x$$

$$2 * x = x + x = x \ll 1 \quad (?)$$

$$x/2 = x * 0.5 = x \gg 1 \quad (?)$$

Constant folding

$$2 * 0.123456789101112131415 = 0.246913578202224262830$$

Chapernowne's constant

Goal: identify equivalence module strength reduction operations.



Algebraic properties

- Commutative: Say the operator * is commutative. $x * y = y * x$
- Associative: $a + (b - c) = (a + b) - c$
 - $a = b + c$
 - $e = c + d + b$
 - >
 - $a = b + c$
 - $t = c + d$
 - $a = t + b$
 - > (assuming t is not used anywhere else)
 - $a = b + c$
 - $e = a + d$
- $a = b - 1; c = a + 1 \rightarrow c = b$



How to?

In general the problem is that of checking equivalence of two expressions – Undecidable!

A rough idea:

- When creating the DAG, create the node for expression that has the most reduced strength.
- For each expression e ,
 - Take all “sub-expressions” that “build” the operands of e .
 - Build a new large expression using these sub-expressions.
 - Simplify the large expression.
 - Check if the simplified expression (or part thereof) or any variations thereof can be found in the tree.
 - Build sub-tree for the rest.



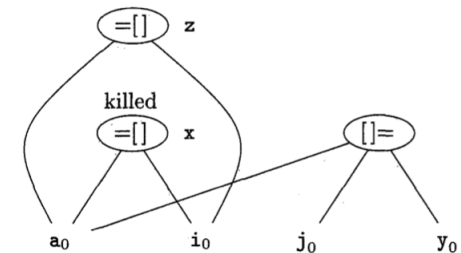
Restrictions

- The language manual may restrict.
 - Fortran: you can evaluate any equivalent expression, but cannot violate the integrity of paranthesis.
 - Thus $x * y - x * z \rightarrow x * (y - z)$
 - But $a + (b - c) \neq (a + b) - c$
- Keep a language manual handy if you are writing a compiler!



Representing Array accesses in the DAG

$x = a[i]$
 $a[j] = y$
 $z = a[i]$



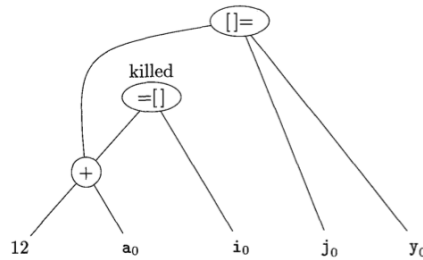
Q: Is $a[i]$ a common sub-expression?



Array representation (2)

```
b = a + 12
x = b[i]
a[j] = y
```

Q: Say, elements of 'a' are 4bytes size



Home reading: How to handle pointers.



Peephole optimization

- A local optimization technique.
- Simplistic in nature, but effective in practise.
- Idea:
 - Keep a sliding window (called peephole)
 - Replace instruction sequences within the peephole by a by an efficient (shorter / faster / ...) sequence.



Peephole optimization

- The “peephole” is typically small. Why?
- The code in the peephole need not be contiguous.
- Each improvement may lead to additional improvements.
- In general, we may have to make multiple passes.



Eliminating redundant loads and stores

```
Load a, R0
Store R0, a
```

Delete the pair of instructions. Always?

What if there is a label on the store instruction?

We need to be sure that the `Store` instruction and `Load` are executed as a pair.

Why would we have such stupid code?



Eliminating unreachable code

- An unlabelled statement after an unconditional jump – can be removed.

```
goto L2
INCR R0
L2:
```

- Eliminating jumps over jumps:

```
    if class == 2010 goto L1
    goto L2
L1: print 22
L2:
→
if class != 2010 goto L2
print 22
L2:
```

- What can constant propagation do?



Flow-of-control optimizations

- Naive code generation creates many jumps.
- Jumps to jumps can be short circuited!

```
goto L1
...
L1: goto L2
```

Can be replaced with

```
goto L2
...
L1: goto L2
```

Further optimizations on L1 are possible.

Similar situation with conditional jumps

```
if (cond) goto L1
...
L1: goto L2
```



Algebraic simplification and strength reduction

- Eliminate identity operations.
- Replace x^2 by $x*x$, and so on.
- Replace fixed-point mult by a power of two (by left-shift) and division by a power of two (by right shift).
- Replace floating-point division by multiplication!



Machine specific peephole optimization

- Use auto-increment / auto-decrement if available.
add r1, (r2)+ → r1 = r1 + M[r2]; r2 = r2+d
- A cool PA-RISC instruction called sh2add
r2 = r1 * 5 → sh2add r1, r1, r2
- PA-RISC instruction ADDBT, <= r2, r1, L1



Peephole procedure

- First make a list of patterns that you want to replace with a list of target patterns.
- Identify the pattern in the code and do the replacement.
- Iterate till you are done.
- Can be efficiently done on an DAG.
- No guarantees about optimality.
- Most of the peephole optimizations guarantee improvement.

