OPTIMIZATION

CMPT 379 Lecture 23b

Lecture Overview

- Introduction
- Local Optimization
  - Local Value Numbering (LVN)
- Regional Optimization
  - Superlocal Value Numbering (SVN)
  - Loop Unrolling
- Global Optimization
  - Live Variables - dataflow

[Chapter 8]

Optimization and Safety

- Two main issues for optimization: safety and profitability.
- Safety means that the program behaves the same with or without the optimization. The compiler (or compiler writer) must have some way of proving that this is the case.
  - Observational Equivalence is the notion that two expressions are equivalent if, in every possible context, they either both produce the same result or both run forever.
  - The safety required of compilers is looser; here we are allowed to substitute expressions if, in the program context, they either both produce the same result or both run forever.

Optimization and Profitability

- Two main issues for optimization: safety and profitability.

- Profitability means that it is expected that the optimization will improve performance.

- Profitability is not easy to estimate in most cases.

- Some program transformations encounter risk to profitability in the form of more register use, fewer cache hits, or less opportunity for future optimization, for example.

Scope of Optimizations

- Local methods: within a single basic block
- Regional methods: bigger than a single basic block but smaller than a full procedure
- Global methods: an entire procedure as context.
- Interprocedural methods: the entire program as context.

- *superlocal, modular, supermodular...*

Local Optimization

- These are among the simplest transformations a compiler can use. The simple execution model of a basic block leads to reasonably precise analysis in support of optimization.

$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$

B
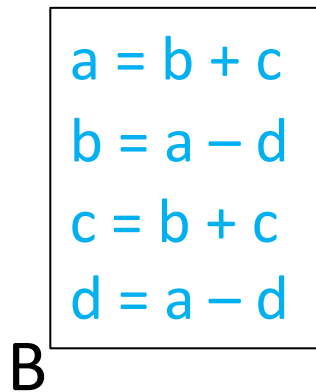
- An expression is redundant in basic block B iff it has been previously computed in B and no intervening operation redefines one of its constituent arguments.

Redundant Expressions

B
```
a = b + c
b = a − d
c = b + c
d = a − d
```

In this example, the occurance of b+c in the third instruction is not redundant, and the occurance of a-d in the fourth instruction is redundant.

We can rewrite this block to compute a − d once.

B'
```
a = b + c
b = a − d
c = b + c
d = b
```

Redundant Expressions

a = b + c

b = a − d

c = b + c

d = b

B'

We replaced the second a − d with a copy of b.

An alternative strategy would replace subsequent uses of d with uses of b, but this would require analysis to see if b is redefined before some use of d.

It is simpler to have the optimizer insert the copy and let a later pass determine which copy operations are necessary and which can be eliminated.

Replacing redundant evaluations with copies is generally profitable, but encounters some risk by increasing the lifetimes of some variables, which increases register pressure.

Redundant Expressions

```
a = b * c
d = b
e = d * c
```

Redundant expressions need not be textually identical.

- Many techniques that find and eliminate redundancies have been developed.

- We next examine one of these methods, Local Value Numbering (LVN).

Local Value Numbering

- Associates a number with each distinct value computed.

- Works on a basic block.   The block has n operations
    $T_i = L_i \, Op_i \, R_i$

- Keeps a counter, initially 0, which is the next value number to assign. Increments the counter whenever a new value number is assigned.

- Keeps a hash table H which has keys L_i or $(Op_i, H(L_i), H(R_i))$ and entries that are value numbers.

- Keeps an array Name in which Name[i] is the name of a variable with value number i.

- To get the value number of a variable V, first look in the hash table to see if the variable is there. If so, the value number is H(V). If not, assign a new value number to V and put it in H() and Name[].

Local Value Numbering

$$T_i = L_i \; Op_i \; R_i$$

for i = 0 **to** n-1
    Get the value numbers for L_i and R_i.
    Construct a key from Op_i and the value numbers
        for L_i and R_i.
    If the key is in the (hash) table H **then**
        let m be the value number H(key)
        replace operation i with a copy of Name(m)
            into T_i
        $H(T_i) = H(key);$
    else
        insert a new value number m into the table at
            the hash key location.
        $H(T_i) = m$
        $Name(m) = T_i$

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

$a^2 = b^0 + c^1$
$b^4 = a^2 - d^3$
$c^5 = b^4 + c^1$
$d^4 = b^4$

Extending LVN

- **Commutative operations**. Commutative operations that differ in only in the order of their operands, such as **a * b** and **b * a**, should receive the same value number. This can be done by sorting the operands before hashing.

- **Constant folding**. If all operands have known constant values, perform the operation and fold the answer directly into the code, rather than doing the hashtable lookup. Must keep information about which values are constant (and what their values are).

Extending LVN

- **Algebraic identities.** LVN Can apply algebraic identities to simplify the code. For example, **x + 0** and **x** should get the same value number. Unfortunately, special-case code is required for each identity. Too many identities can cause a slowdown, so organize them into operator-specific decision trees. Some identities that can be handled this way:

$a + 0 = a$	$a - 0 = a$	$a - a = 0$	$2 * a = a + a$
$a * 1 = a$	$a * 0 = 0$	$a / 1 = a$	$a / a = 1$ (a != 0)
$a^1 = a$	$a^2 = a * a$	$a >> 0 = a$	$a << 0 = a$
a AND $a = a$	a OR $a = a$	$MAX(a,a) = a$	$MIN(a, a) = a$

The Role of Naming

- The choice of names for variables and values can limit the effectiveness of LVN.

<div style="display:flex">

$$a = x + y$$
$$b = x + y$$
$$a = 17$$
$$c = x + y$$

$$a^3 = x^1 + y^2$$
$$b^3 = a^3$$
$$a^4 = 17^4$$
$$c^3 = x^1 + y^2$$

</div>

- We can use Static Single-Assignment form to eliminate this problem:

<div style="display:flex">

$$a = x + y$$
$$b = x + y$$
$$a' = 17$$
$$c = x + y$$

$$a^3 = x^1 + y^2$$
$$b^3 = a^3$$
$$a'^4 = 17^4$$
$$c^3 = a^3$$

</div>

Indirect Assignments

- Indirect assignments cause problems for analysis. Indirect assignments include assignment through a pointer, such as *p = 0, and assignment to an unknown array element, such as a(i, j) = 0.

- Without specific knowledge of the memory locations to which p can refer, the compiler must treat *p = 0 as if it modifies every variable. Similarly, an unknown i and j in a(i, j) = 0 means the compiler must treat every value in the entire array as changed.

- Compilers invest a lot of energy in narrowing the scope of indirect references so that other optimizations such as LVN are effective.

Regional Optimization

- Most optimizations examine a larger context than a single block. The main complication this introduces is the need to consider more than one possibility for the flow of control.

  - **Extended Basic Blocks (EBBs).** An extended basic block is a set of basic blocks B_1, B_2, … B_n in the control flow graph (CFG), where B_1 may have multiple CFG predecessors and each other B_i has just one, which is some B_j in the set.
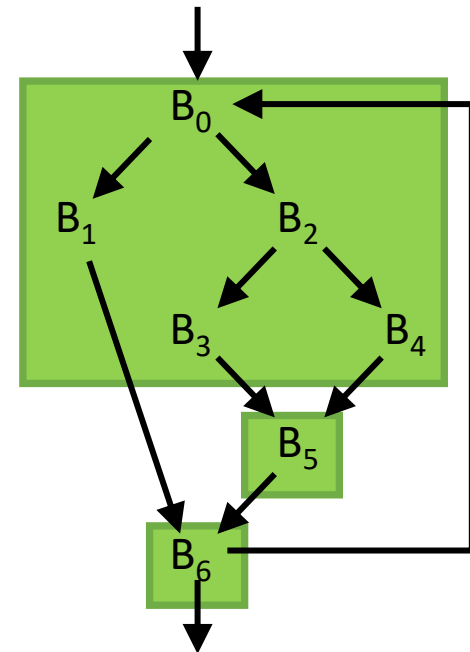
Superlocal Value Numbering (SVN)

- Uses EBBs as a starting structure. In a large EBB, each path from root to leaf is analyzed as a single basic block. In the above example, B_0B_1, $B_0B_2B_3$, and $B_0B_2B_4$ are the paths in the large EBB. To analyze B_0B_1 as a single block, analyze B_0 with LVN and use resulting hash table and value numbers to start the analysis of B_1.

- SSA should be used with SVN so that the lookup of value numbers for arguments is simplified.

Superlocal Value Numbering (SVN)

- To make SVN efficient, the compiler must reuse the results of blocks that occur as prefixes on multiple paths through the EBB. There are several ways to do this, some more efficient than others.

- Lexically scoped hash tables are a natural fit. These hash tables have extra operations of *enterScope()* and *leaveScope();* when a *leaveScope()* is executed, all entries that entered the table since the last *enterScope()* are removed.

Loop Unrolling

- The oldest and best-known loop transformation.
- The loop body is replicated and the logic controlling the loop is adjusted.

```
do 80 i=1, n
      sum = sum + a(i)
80 continue
```

→

```
n1 = mod(n, 2)
if( n1 .eq. 1) then
      sum = sum + a(1)
end if
do 80 i = n1+1, n, 2
      sum = sum + a(i)
      sum = sum + a(i+1)
80 continue
```

- Here, there is an opportunity to optimize the body of the loop, combining the assignments into one:

```
sum = sum + a(i) + a(i-1)
```

Loop Unrolling

- Loop unrolling has direct and indirect benefits.
- The direct benefits are that the number of operations required to complete the loop is reduced. The control-flow changes reduce the number of test-and-branch sequences. Variable reuse can be created, reducing memory traffic.
- As a risk, though, unrolling increases program size. If this causes the loop to overflow the instruction cache, it can cause serious degradation.
- Indirect benefits are a result of having more instructions in the body of the loop. They include:
    - Better instruction scheduling.
    - Better scheduling of memory accesses.
    - Cross-iteration redundancies that can be eliminated.
    - Can change register allocation (for good or bad).

Global Optimization

- Global optimizations operate on an entire procedure.
- Typically global optimizations have an analysis phase followed by a code modification phase.
- *Live variable analysis*. A variable **v** is live at point **p** if and only if there exists a path in the CFG from **p** to a use of **v** along which **v** is not redefined.
- We will compute, for each basic block **b** in the procedure, a set **LiveOut(b)** that contains all the variables that are live on exit from b.
- Computing **LiveOut** is an example of global dataflow analysis, a family of techniques for static reasoning about the flow of dynamic values.

Dataflow and LiveOut

- Problems in dataflow are typically posed as a set of simultaneous equations over sets associated with the nodes and edges of the CFG.

- The defining equation for **LiveOut** is:

$$\text{LiveOut}(n) =$$

$$\bigcup_{m \,\in\, \text{succ}(n)} \left(\text{UEVar}(m) \cup \left(\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}\right)\right)$$

LiveOut

$$\text{LiveOut}(n) =$$

$$\bigcup_{m \in \text{succ}(n)} (\text{UEVar}(m) \cup \left(\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}\right))$$

- UEVar(b) is the set of Upwards-Exposed Variables in block b. These are the variables used in b before any redefinition of them in b.

- VarKill(b) is the set of all variables defined in block b. The overbar represents logical complement, so $\overline{\text{VarKill}(m)}$ is the set of all variables *not* defined in block m.

- The defining equation says that LiveOut of a particular block *n* is the union of the variables that are live at the head of some successor of *n*. The variables that are live at the head of block *m* are the upwards-exposed variables of *m* along with those live-out variables of *m* that do not have definitions in *m*.

LiveOut

- Note that the defining equation for LiveOut is for one block only. For a given procedure, we have a copy/variant of this equation for each block in the procedure. The variation in the equation is because different blocks have different successors, and different blocks have different UEVar and VarKill.

- Thus, LiveOut is actually defined by a system of simultaneous equations, each one of the form

$$\text{LiveOut}(n) =$$

$$\bigcup_{m \in \text{succ}(n)} \left(\text{UEVar}(m) \cup \left(\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}\right)\right)$$

- We will discuss solving this system of equations, as well as how to compute UEVar and VarKill for a block, in the next lecture.