CSE P 501 – Compilers

Introduction to Optimization

Hal Perkins

Autumn 2011

Agenda

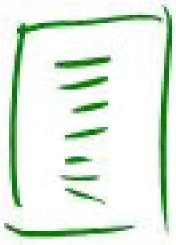
- Optimization
  - Goals
  - Scope: local, superlocal, regional, global (intraprocedural), interprocedural
- Control flow graphs
- Value numbering
- Dominators
- Ref.: Cooper/Torczon ch. 8 + 9

Code Improvement (1)

- Pick a better algorithm(!)
- Use machine resources effectively
  - Instruction selection & scheduling
  - Register allocation
  - More about these later…

Code Improvement (2)

- Local optimizations – basic blocks
  - Algebraic simplifications
  - Constant folding
  - Common subexpression elimination (i.e., redundancy elimination)
  - Dead code elimination
  - Specialize computation based on context
  - etc., etc., …



Code Improvement (3)

- Global optimizations
  - Code motion
  - Moving invariant computations out of loops
  - Strength reduction (replace multiplications by repeated additions, for example)
  - Global common subexpression elimination
  - Global register allocation
  - Many others...

“Optimization”

- None of these improvements are truly “optimal”
  - Hard problems
  - Proofs of optimality assume artificial restrictions
- Best we can do is to improve things
      - Most (much?) (some?) of the time
  - Realistically: try to do better for common idioms both in the code and on the machine

Example: A[i,j]

- Without any surrounding context, need to generate code to calculate

  address(A)
  + (i-low_1(A)) * ($high_2$(A)-low_2(a)+1) * size(A)
  + (j-low_2(A)) * size(A)

  - low_i and $high_i$ are subscript bounds in dimension i
  - address(A) is the runtime address of first element of A
- ... And we really should be checking that i, j are in bounds

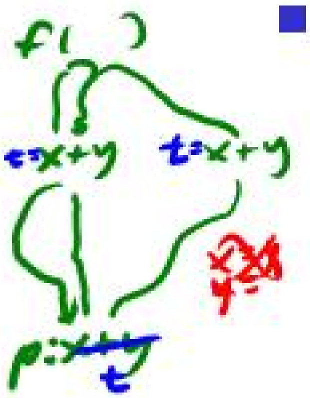
Some Optimizations for A[i,j]

- With more context, we can do better
- Examples
  - If A is local, with known bounds, much of the computation can be done at compile time
  - If A[i,j] is in a loop where i and j change systematically, we probably can replace multiplications with additions each time around the loop to reference successive rows/columns
    - Even if not, we can move "loop-invariant" parts of the calculation outside the loop

Optimization Phase

- Goal
  - Discover, at compile time, information about the runtime behavior of the program, and use that information to improve the generated code

A First Running Example: Redundancy Elimination



- An expression x+y is *redundant* at a program point iff, along every path from the procedure's entry, it has been evaluated and its constituent subexpressions (x and y) have not been redefined
- If the compiler can prove the expression is redundant:
  - Can store the result of the earlier evaluation
  - Can replace the redundant computation with a reference to the earlier (stored) result

Common Problems in Code Improvement

- This strategy is typical of most compiler optimizations
  - First, discover opportunities through program analysis
  - Then, modify the IR to take advantage of the opportunities
    - Historically, goal usually was to decrease execution time
    - Other possibilities: reduce space, power, …

Issues (1)

- Safety – transformation must not change program meaning
  - Must generate correct results
  - Can't generate spurious errors
  - Optimizations must be conservative
  - Large part of analysis goes towards proving safety
  - Can pay off to speculate (be optimistic) but then need to recover if reality is different

x.m(—)
if (ok)
call T::m
else
vtable

Issues (2)

- Profitibility
  - If a transformation is possible, is it profitable?
  - Example: loop unrolling
    - Can increase amount of work done on each iteration, i.e., reduce loop overhead
    - Can eliminate duplicate operations done on separate iterations

Issues (3)

- Downside risks
  - Even if a transformation is generally worthwhile, need to think about potential problems
  - For example:
    - Transformation might need more temporaries, putting additional pressure on registers
    - Increased code size could cause cache misses, or, in bad cases, increase page working set

$x+y^2$ 3

Example: Value Numbering

- Technique for eliminating redundant expressions: assign an identifying number VN(n) to each expression
  - VN(x+y)=VN(j) if x+y and j have the same value
  - Use hashing over value numbers for effeciency
- Old idea (Balke 1968, Ershov 1954)
  - Invented for low-level, linear IRs
  - Equivalent methods exist for tree IRs, e.g., build a DAG

Uses of Value Numbers

- Improve the code
  - Replace redundant expressions
  - Simplify algebraic identities
  - Discover, fold, and propagate constant valued expressions

$x = y + z$

Local Value Numbering

- Algorithm
  - For each operation o = <op, o1,o2> in a block
    1. Get value numbers for operands from hash lookup
    2. Hash <op, VN(o1), VN(o2)> to get a value number for o (If op is commutative, sort VN(o1), VN(o2) first)
    3. If o already has a value number, replace o with a reference to the value
    4. If o1 and o2 are constant, evaluate o at compile time and replace with an immediate load
- If hashing behaves well, this runs in linear time

Example

Code

$a^3 = x^1 + y^2$

$b^3 = x^1 + y^2$

$a^4 = 17^4$

$c^3 = x^1 + y^2$

Rewritten

$a^3 = x^1 + y^2$ $t = a^3$

$b^3 = a^3$

$a^4 = 17^4$

$c^3 = a^3$ $c^3 = t$

Bug in Simple Example

- If we use the original names, we get in trouble when a name is reused
- Solutions
  - Be clever about which copy of the value to use (e.g., use c=b in last statement)
  - Create an extra temporary
  - Rename around it (best!)

Renaming

V_n^{vn}

- Idea: give each value a unique name
  - a_i^j means i^{th} definition of a with VN = j
- Somewhat complex notation, but meaning is clear
- This is the idea behind SSA (Static Single Assignment)
  - Popular modern IR – exposes many opportunities for optimizations

Example Revisited

Code

$a_0^3 = x_0^1 + y_0^2$

$b_0^3 = x_0^1 + y_0^2$

$a_1^4 = 17^4$

$c_0^3 = x_0^1 + y_0^2$

Rewritten

$a_0^3 = x_0^1 + y_0^2$

$b_0^3 = a_0^3$

$a_1^4 = 17^4$

$c_0^3 = a_0^3$