


CSE 582 – Compilers

Introduction to Optimization
Hal Perkins
Autumn 2002


12/3/2002 © 2002 Hal Perkins & UW CSE Q-1



Agenda

- Optimization
 - Goals
 - Scope: local, superlocal, regional, global, interprocedural
- Control flow graphs
- Value numbering
- Dominators


12/3/2002 © 2002 Hal Perkins & UW CSE Q-2



Code Improvement – How?

- Pick a better algorithm(!)
- Use machine resources effectively
 - Instruction selection & scheduling
 - Register allocation


12/3/2002 © 2002 Hal Perkins & UW CSE Q-3



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


12/3/2002 © 2002 Hal Perkins & UW CSE Q-4



Code Improvement (3)

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


12/3/2002 © 2002 Hal Perkins & UW CSE Q-5



“Optimization”

- None of these improvements are truly “optimal”
 - Hard problems
 - Proofs of optimality assume artificial restrictions
- Best we can do is to improve things


12/3/2002 © 2002 Hal Perkins & UW CSE Q-6



Example: $A[i,j]$

- Without any surrounding context, need to generate code to calculate
 - address(A)
 - + $(i - \text{low}_1(A)) * (\text{high}_2(A) - \text{low}_2(a) + 1) * \text{size}(A)$
 - + $(j - \text{low}_2(A)) * \text{size}(A)$
 - low_i and high_i are subscript bounds in dimension i
 - $\text{address}(A)$ is the runtime address of first element of A


12/3/2002 © 2002 Hal Perkins & UW CSE Q-7



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, probably can replace multiplications with additions each time around the loop to reference successive rows/columns

12/3/2002 © 2002 Hal Perkins & UW CSE Q-8



Optimization Phase

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

12/3/2002 © 2002 Hal Perkins & UW CSE Q-9

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 & 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

12/3/2002 © 2002 Hal Perkins & UW CSE Q-10

Common Problems in Code Improvement


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

12/3/2002 © 2002 Hal Perkins & UW CSE Q-11

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


12/3/2002 © 2002 Hal Perkins & UW CSE Q-12



Issues (2)

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


12/3/2002 © 2002 Hal Perkins & UW CSE Q-13



Issues (3)

- n Downside risks
 - n Even if a transformation is generally worthwhile, need to factor in potential problems
 - n Sample issues
 - n Transformation might need more temporaries, putting additional pressure on registers
 - n Increased code size could cause cache misses


12/3/2002 © 2002 Hal Perkins & UW CSE Q-14



Value Numbering

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


12/3/2002 © 2002 Hal Perkins & UW CSE Q-15



Uses of Value Numbers

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


12/3/2002 © 2002 Hal Perkins & UW CSE Q-16



Local Value Numbering

- n Algorithm
 - n For each operation $o = \langle op, o1, o2 \rangle$ in the block
 1. Get value numbers for operands from hash lookup
 2. Hash $\langle op, VN(o1), VN(o2) \rangle$ 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
 4. If $o1$ and $o2$ are constant, evaluate o at compile time and replace with an immediate load
 - n If hashing behaves, this runs in linear time

12/3/2002 © 2002 Hal Perkins & UW CSE Q-17



Example

Code	Rewritten
a = x + y	
b = x + y	
a = 17	
c = x + y	

12/3/2002 © 2002 Hal Perkins & UW CSE Q-18

Bug in Simple Example

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

12/3/2002 © 2002 Hal Perkins & UW CSE Q-19

Renaming


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

12/3/2002 © 2002 Hal Perkins & UW CSE Q-20

Example Revisited

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


12/3/2002 © 2002 Hal Perkins & UW CSE Q-21



Simple Extensions to Value Numbering

- n Constant folding
 - n Add a bit that records when a value is constant
 - n Evaluate constant values at compile time
 - n Replace op with load immediate
- n Algebraic identities: $x+0$, $x*1$, $x-x$, ...
 - n Many special cases
 - n Switch on op to narrow down checks needed
 - n Replace result with input VN


12/3/2002 © 2002 Hal Perkins & UW CSE Q-22



Larger Scopes

- n The given algorithm works on straight-line blocks of code (basic blocks)
 - n Best possible results for single basic blocks
 - n Loses all information when control flows to another block
- n To go further we need to represent multiple blocks of code and the control flow between them


12/3/2002 © 2002 Hal Perkins & UW CSE Q-23



Basic Blocks

- n Definition: A *basic block* is a maximal length sequence of straight-line code
- n Properties
 - n Statements are executed sequentially
 - n If any statement executes, they all do (barring exceptions)
- n In a linear IR, the first statement of a basic block is often called the *leader*


12/3/2002 © 2002 Hal Perkins & UW CSE Q-24



Control Flow Graph (CFG)

- n Nodes: basic blocks
 - n Possible representations: linear 3-address code, expression-level AST, DAG
- n Edges: include a directed edge from n1 to n2 if there is any possible way for control to transfer from block n1 to n2 during execution


12/3/2002 © 2002 Hal Perkins & UW CSE Q-25



Constructing Control Flow Graphs from Linear IRs

- n Algorithm
 - n Pass 1: Identify basic block leaders with a linear scan of the IR
 - n Pass 2: Identify operations that end a block and add appropriate edges to the CFG to all possible successors
 - n Details: Ch. 9 in the textbook
- n For convenience, ensure that every block ends with conditional or unconditional jump
 - n Code generator can pick the most convenient "fall-through" case later


12/3/2002 © 2002 Hal Perkins & UW CSE Q-26



Scope of Optimizations

- n Optimization algorithms can work on units as small as a basic block or as large as a whole program
- n Local information is generally more precise and can lead to locally optimal results
- n Global information is less precise (lose information at join points in the graph), but exposes opportunities for improvements across basic blocks


12/3/2002 © 2002 Hal Perkins & UW CSE Q-27



Optimization Categories (1)

- *Local methods*
 - Usually confined to basic blocks
 - Simplest to analyze and understand
 - Most precise information


12/3/2002 © 2002 Hal Perkins & UW CSE Q-28



Optimization Categories (2)

- *Superlocal methods*
 - Operate over *Extended Basic Blocks* (EBBs)
 - An EBB is a set of blocks b_1, b_2, \dots, b_n where b_1 has multiple predecessors and each of the remaining blocks b_i ($2 \leq i \leq n$) have only b_1 as its unique predecessor
 - The EBB is entered only at b_1 , but may have multiple exits
 - A single block b_1 can be the head of multiple EBBs (these EBBs form a tree rooted at b_1)
 - Use information discovered in earlier blocks to improve code in successors

12/3/2002 © 2002 Hal Perkins & UW CSE Q-29



Optimization Categories (3)

- *Regional methods*
 - Operate over scopes larger than an EBB but smaller than an entire procedure/function/method
 - Typical example: loop body
 - Difference from superlocal methods is that there may be merge points in the graph (i.e., a block with two or more predecessors)

12/3/2002 © 2002 Hal Perkins & UW CSE Q-30

Optimization Categories (4)

- Global methods
 - Operate over entire procedures
 - Sometimes called *intraprocedural* methods
 - Motivation is that local optimizations sometimes have bad consequences in larger context
 - Procedure/method/function is a natural unit for analysis, separate compilation, etc.
 - Almost always need global *data-flow* analysis information for these

12/3/2002 © 2002 Hal Perkins & UW CSE Q-31

Optimization Categories (5)

- Whole-program methods
 - Operate over more than one procedure
 - Sometimes called *interprocedural* methods
 - Challenges: name scoping and parameter binding issues at procedure boundaries
 - Classic examples: inline method substitution, interprocedural constant propagation
 - Fairly common in aggressive JIT compilers and optimizing compilers for object-oriented languages

12/3/2002 © 2002 Hal Perkins & UW CSE Q-32

Value Numbering Revisited

- Local Value Numbering
 - 1 block at a time
 - Strong local results
 - No cross-block effects
- Missed opportunities

12/3/2002 © 2002 Hal Perkins & UW CSE Q-33

Superlocal Value Numbering

- Idea: apply local method to EBBs
 - {A,B}, {A,C,D}, {A,C,E}
- Final info from A is initial info for B, C; final info from C is initial for D, E
- Gets reuse from ancestors
- Avoid reanalyzing A, C
- Doesn't help with F, G

12/3/2002 © 2002 Hal Perkins & UW CSE Q-34

SSA Name Space (from before)

Code	Rewritten
$a_0^3 = x_0^1 + y_0^2$	$a_0^3 = x_0^1 + y_0^2$
$b_0^3 = x_0^1 + y_0^2$	$b_0^3 = a_0^3$
$a_1^4 = 17$	$a_1^4 = 17$
$c_0^3 = x_0^1 + y_0^2$	$c_0^3 = a_0^3$

- Unique name for each definition
- Name VN
- a_0^3 is available to assign to c_0^3

12/3/2002 © 2002 Hal Perkins & UW CSE Q-35

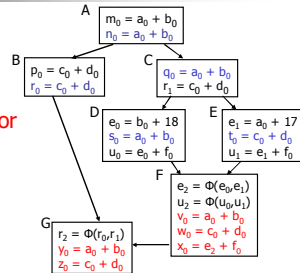
SSA Name Space

- Two Principles
 - Each name is defined by exactly one operation
 - Each operand refers to exactly one definition
- Need to deal with merge points
 - Add Φ functions at merge points to reconcile names
 - Use subscripts on variable names for uniqueness

12/3/2002 © 2002 Hal Perkins & UW CSE Q-36

Superlocal Value Numbering with All Bells & Whistles

- Finds more redundancies
- Little extra cost
- Still does nothing for F and G



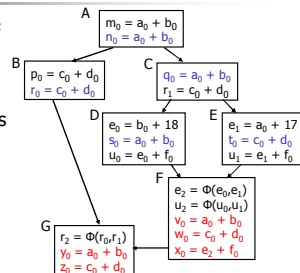
12/3/2002

© 2002 Hal Perkins & UW CSE

Q-37

Larger Scopes

- Still have not helped F and G
- Problem: multiple predecessors
- Must decide what facts hold in F and in G
 - For G, combine B & F?
 - Merging states is expensive
 - Fall back on what we know



12/3/2002

© 2002 Hal Perkins & UW CSE

Q-38

Dominators

- Definition
 - x *dominates* y iff every path from the entry of the control-flow graph to y includes x
- By definition, x dominates x
- Associate a Dom set with each node
 - $|\text{Dom}(X)| \geq 1$
- Many uses in analysis and transformation
 - Finding loops, building SSA form, code motion

12/3/2002

© 2002 Hal Perkins & UW CSE

Q-39



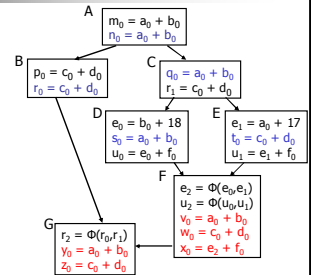
Immediate Dominators

- For any node x , there is a y in $\text{Dom}(x)$ closest to x
- This is the *immediate dominator* of x
 - Notation: $\text{IDom}(x)$



Dominator Sets

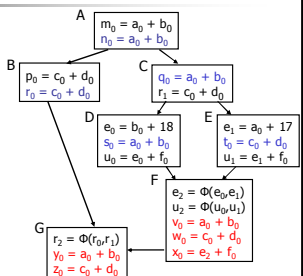
Block Dom IDom





Dominator Value Numbering

- Still looking for a way to handle F and G
- Idea: Use info from $\text{IDom}(x)$ to start analysis of x
 - Use C for F and A for G
- Dominator VN Technique (DVNT)



DVNT algorithm

- n Use superlocal algorithm on extended basic blocks
 - n Use scoped hash tables & SSA name space as before
- n Start each node with table from its IDOM
- n No values flow along back edges (i.e., loops)
- n Constant folding, algebraic identities as before

12/3/2002 © 2002 Hal Perkins & UW CSE Q-43

Dominator Value Numbering

- n Advantages
 - n Finds more redundancy
 - n Little extra cost
- n Shortcomings
 - n Misses some opportunities (common calculations in ancestors that are not IDOMs)
 - n Doesn't handle loops or other back edges

The graph consists of nodes A through G. Node A is the root with $m_0 = a_0 + b_0$ and $n_0 = a_0 + b_0$. Node B has $p_0 = c_0 + d_0$ and $r_0 = c_0 + d_0$. Node C has $q_0 = a_0 + b_0$ and $r_1 = c_0 + d_0$. Node D has $e_0 = b_0 + 18$, $s_0 = a_0 + b_0$, and $u_0 = e_0 + f_0$. Node E has $e_1 = a_0 + 17$, $t_0 = c_0 + d_0$, and $u_1 = e_1 + f_0$. Node F has $e_2 = \Phi(e_0, e_1)$, $u_2 = \Phi(u_0, u_1)$, $v_0 = a_0 + b_0$, $w_0 = c_0 + d_0$, and $x_0 = e_2 + f_0$. Node G has $r_2 = \Phi(r_0, r_1)$, $y_0 = a_0 + b_0$, and $z_0 = c_0 + d_0$. Edges connect A to B and C, B to D, C to E, D to F, E to F, and F to G.

12/3/2002 © 2002 Hal Perkins & UW CSE Q-44

The Story So Far...

- n Local algorithm
- n Superlocal extension
 - n Some local methods extend cleanly to superlocal scopes
- n Dominator VN Technique (DVNT)
- n All of these propagate along forward edges
- n None are global

12/3/2002 © 2002 Hal Perkins & UW CSE Q-45



Coming Attractions

- n Data-flow analysis
 - n Provides global solution to redundant expression analysis
 - n Catches some things missed by DVNT, but misses some others
 - n Generalizes to many other analysis problems, both forward and backward
- n Transformations
 - n A catalog of some of the things a compiler can do with the analysis information
