


CSE 582 – Compilers

Data-flow Analysis
Hal Perkins
Autumn 2002

12/5/2002 © 2002 Hal Perkins & UW CSE R-1




Agenda

- n Initial example: data-flow analysis for common subexpression elimination
- n Other analysis problems that work in the same framework

- n Credits: Largely based on Keith Cooper's slides from Rice University

12/5/2002 © 2002 Hal Perkins & UW CSE R-2



The Story So Far...

- n Redundant expression elimination
 - n Local Value Numbering
 - n Superlocal Value Numbering
 - n Extends VN to EBBs
 - n SSA-like namespace
 - n Dominator VN Technique (DVNT)
- n All of these propagate along forward edges
- n None are global
 - n In particular, can't handle back edges (loops)

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

Dominator Value Numbering

- Most sophisticated algorithm so far
- Still misses some opportunities
- Can't handle loops

12/5/2002 © 2002 Hal Perkins & UW CSE R-4

Available Expressions


- Goal: use data-flow analysis to find common subexpressions whose range spans basic blocks
- Idea: calculate *available expressions* at beginning of each basic block
 - Data-flow analysis
- Avoid re-evaluation of an available expression – use a copy operation

12/5/2002 © 2002 Hal Perkins & UW CSE R-5

“Available” and Other Terms

- An expression e is *defined* at point p in the CFG if its value is computed at p
 - Sometimes called *definition site*
- An expression e is *killed* at point p if one of its operands is defined at p
 - Sometimes called *kill site*
- An expression e is *available* at point p if every path leading to p contains a prior definition of e and e is not killed between that definition and p


12/5/2002 © 2002 Hal Perkins & UW CSE R-6



Available Expression Sets

- For each block b , define
 - AVAIL(b) – the set of expressions available on entry to b
 - NKILL(b) – the set of expressions not killed in b
 - DEF(b) – the set of expressions defined in b and not subsequently killed in b

12/5/2002 © 2002 Hal Perkins & UW CSE R-7




Computing Available Expressions

- AVAIL(b) is the set

$$AVAIL(b) = \bigcap_{x \in \text{preds}(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$
 - preds(b) is the set of b 's predecessors in the control flow graph
- This gives a system of simultaneous equations – a data-flow problem


12/5/2002 © 2002 Hal Perkins & UW CSE R-8



Name Space Issues

- In previous value-numbering algorithms, we used a SSA-like renaming to keep track of versions
- In global data-flow problems, we use the original namespace
 - The KILL information captures when a value is no longer available


12/5/2002 © 2002 Hal Perkins & UW CSE R-9



GCSE with Available Expressions

- For each block b , compute $DEF(b)$ and $NKILL(b)$
- For each block b , compute $AVAIL(b)$
- For each block b , value number the block starting with $AVAIL(b)$
- Replace expressions in $AVAIL(b)$ with references


12/5/2002 © 2002 Hal Perkins & UW CSE R-10



Replacement Issues

- Need a unique name for each expression in $AVAIL(b)$
- Several possibilities; all workable


12/5/2002 © 2002 Hal Perkins & UW CSE R-11



Global CSE Replacement

- After analysis and before transformation, assign a global name to each expression e by hashing on e
- During transformation step
 - At each evaluation of e , insert copy $name(e) = e$
 - At each reference to e , replace e with $name(e)$


12/5/2002 © 2002 Hal Perkins & UW CSE R-12



Analysis

- n Main problem – inserts extraneous copies at all definitions and uses of every e that appears in any $AVAIL(b)$
 - n But the extra copies are dead and easy to remove
 - n Useful copies often coalesce away when registers and temporaries are assigned
- n Common strategy
 - n Insert copies that might be useful
 - n Let dead code elimination sort it out later


12/5/2002 © 2002 Hal Perkins & UW CSE R-13



Computing Available Expressions

- n Big Picture
 - n Build control-flow graph
 - n Calculate initial local data – $DEF(b)$ and $NKILL(b)$
 - n This only needs to be done once
 - n Iteratively calculate $AVAIL(b)$ by repeatedly evaluating equations until nothing changes
 - n Another fixed-point algorithm

12/5/2002 © 2002 Hal Perkins & UW CSE R-14



Computing DEF and NKILL (1)

- n For each block b with operations o_1, o_2, \dots, o_k
 - $KILLED = \emptyset$
 - $DEF(b) = \emptyset$
 - for $i = k$ to 1
 - assume o_i is " $x = y + z$ "
 - if ($y \notin KILLED$ and $z \notin KILLED$)
 - add " $y + z$ " to $DEF(b)$
 - add x to $KILLED$
 - ...

12/5/2002 © 2002 Hal Perkins & UW CSE R-15

Computing DEF and NKILL (2)

- n After computing DEF and KILLED for a block b ,

$$\text{NKILL}(b) = \{ \text{all expressions} \}$$
 for each expression e
 for each variable $v \in e$
 if $v \in \text{KILLED}$ then

$$\text{NKILL}(b) = \text{NKILL}(b) - e$$

12/5/2002

© 2002 Hal Perkins & UW CSE

R-16

Computing Available Expressions

- n Once $\text{DEF}(b)$ and $\text{NKILL}(b)$ are computed for all blocks b

$$\text{Worklist} = \{ \text{all blocks } b_i \}$$
 while ($\text{Worklist} \neq \emptyset$)
 remove a block b from Worklist
 recompute $\text{AVAIL}(b)$
 if $\text{AVAIL}(b)$ changed

$$\text{Worklist} = \text{Worklist} \cup \text{successors}(b)$$

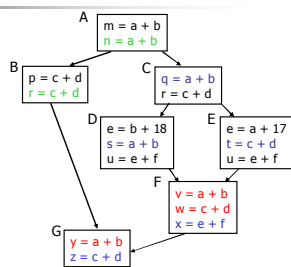
12/5/2002

© 2002 Hal Perkins & UW CSE

R-17

Comparing Algorithms


- n LVN – Local Value Numbering
- n SVN – Superlocal Value Numbering
- n DVN – Dominator-based Value Numbering
- n GRE – Global Redundancy Elimination



12/5/2002

© 2002 Hal Perkins & UW CSE


R-18



Comparing Algorithms (2)

- LVN => SVN => DVN form a strict hierarchy
 - later algorithms find a superset of previous information
- Global RE finds a somewhat different set
 - Discovers e+f in F (computed in both D and E)
 - Misses identical values if they have different names (e.g., a+b and c+d when a=c and b=d)
 - Value Numbering catches this


12/5/2002 © 2002 Hal Perkins & UW CSE R-19



Data-flow Analysis (1)

- A collection of techniques for compile-time reasoning about run-time values
- Almost always involves building a graph
 - Trivial for basic blocks
 - Control-flow graph or derivative for global problems
 - Call graph or derivative for whole-program problems


12/5/2002 © 2002 Hal Perkins & UW CSE R-20



Data-flow Analysis (2)

- Usually formulated as a set of *simultaneous equations* (data-flow problem)
 - Sets attached to nodes and edges
 - Need a lattice (or semilattice) to describe values
 - In particular, has an appropriate operator to combine values and an appropriate "bottom" or minimal value


12/5/2002 © 2002 Hal Perkins & UW CSE R-21



Data-flow Analysis (3)

- Desired solution is usually a *meet over all paths* (MOP) solution
 - "What is true on every path from entry"
 - "What can happen on any path from entry"
 - Usually relates to safety of optimization


12/5/2002 © 2002 Hal Perkins & UW CSE R-22



Data-flow Analysis (4)

- Limitations
 - Precision – "up to symbolic execution"
 - Assumes all paths taken
 - Sometimes cannot afford to compute full solution
 - Arrays – classic analysis treats each array as a single fact
 - Pointers – difficult, expensive to analyze
 - Imprecision rapidly adds up
- Summary: for scalar values we can quickly solve simple problems


12/5/2002 © 2002 Hal Perkins & UW CSE R-23



Scope of Analysis

- Larger context (EBBs, regions, global, interprocedural) sometimes helps
 - More opportunities for optimizations
- But not always
 - Introduces uncertainties about flow of control
 - Usually only allows weaker analysis
 - Sometimes has unwanted side effects
 - Can create additional pressure on registers, for example


12/5/2002 © 2002 Hal Perkins & UW CSE R-24



Some Problems (1)

- n Merge points often cause loss of information
 - n Sometimes worthwhile to clone the code at the merge points to yield two straight-line sequences


12/5/2002 © 2002 Hal Perkins & UW CSE R-25



Some Problems (2)

- n Procedure/function/method calls are problematic
 - n Have to assume anything could happen, which kills local assumptions
 - n Calling sequence and register conventions are often more general than needed
- n One technique – inline substitution
 - n Allows caller and called code to be analyzed together; more precise information
 - n Can eliminate overhead of function call, parameter passing, register save/restore

12/5/2002 © 2002 Hal Perkins & UW CSE R-26



Other Data-Flow Problems

- n The basic data-flow analysis framework can be applied to many other problems beyond redundant expressions
- n Different kinds of analysis enable different optimizations

12/5/2002 © 2002 Hal Perkins & UW CSE R-27

Characterizing Data-flow Analysis

- All of these involve sets of facts about each basic block b
 - $IN(b)$ – facts true on entry to b
 - $OUT(b)$ – facts true on exit from b
 - $GEN(b)$ – facts created and not killed in b
 - $KILL(b)$ – facts killed in b
- These are related by the equation
 - $OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$
 - Solve this iteratively for all blocks
 - Sometimes information propagates forward; sometimes backward

12/5/2002 © 2002 Hal Perkins & UW CSE R-28

Efficiency of Data-flow Analysis


- The algorithms eventually terminate, but the expected time needed can be reduced by picking a good order to visit nodes in the CFG
 - Forward problems – reverse postorder
 - Backward problems – postorder

12/5/2002 © 2002 Hal Perkins & UW CSE R-29

Example: Live Variable Analysis

- A variable v is *live* at point p iff there is *any* path from p to a use of v along which v is not redefined
- Uses
 - Register allocation – only live variables need a register (or temporary)
 - Eliminating useless stores
 - Detecting uses of uninitialized variables
 - Improve SSA construction – only need Φ -function for variables that are live in a block

12/5/2002 © 2002 Hal Perkins & UW CSE R-30




Equations for Live Variables

- n Sets
 - n USED(b) – variables used in b before being defined in b
 - n NOTDEF(b) – variables not defined in b
 - n LIVE(b) – variables live on *exit* from b
- n Equation

$$\text{LIVE}(b) = \bigcup_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{LIVE}(s) \cap \text{NOTDEF}(s))$$


12/5/2002 © 2002 Hal Perkins & UW CSE R-31



Example: Available Expressions

- n This is the analysis we did earlier to eliminate redundant expression evaluation (i.e., compute AVAIL(b))

12/5/2002 © 2002 Hal Perkins & UW CSE R-32



Example: Reaching Definitions

- n A definition d of some variable v *reaches* operation i iff i reads the value of v and there is a path from d to i that does not define v
- n Uses
 - n Find all of the possible definition points for a variable in an expression

12/5/2002 © 2002 Hal Perkins & UW CSE R-33

Equations for Reaching Definitions

- n Sets
 - DEFOUT(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
 - SURVIVED(b) – set of all definitions not obscured by a definition in b
 - REACHES(b) – set of definitions that reach b
- n Equation

$$\text{REACHES}(b) = \bigcup_{p \in \text{preds}(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))$$

12/5/2002 © 2002 Hal Perkins & UW CSE R-34

Example: Very Busy Expressions

- n An expression e is considered *very busy* at some point p if e is evaluated and used along every path that leaves p , and evaluating e at p would produce the same result as evaluating it at the original locations
- n Uses
 - Code hoisting – move e to p (reduces code size; no effect on execution time)


12/5/2002 © 2002 Hal Perkins & UW CSE R-35

Equations for Very Busy Expressions

- n Sets
 - USED(b) – expressions used in b before they are killed
 - KILLED(b) – expressions redefined in b before they are used
 - VERYBUSY(b) – expressions very busy on exit from b
- n Equation

$$\text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s))$$

12/5/2002 © 2002 Hal Perkins & UW CSE R-36



Summary

- n Dataflow analysis gives a framework for finding global information
- n Key to enabling most optimizing transformations

12/5/2002 © 2002 Hal Perkins & UW CSE R-37
