CSE 582 – Compilers

Data-flow Analysis
Hal Perkins
Autumn 2002

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

Credits: Largely based on Keith Cooper’s slides from Rice University

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

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

“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
Available Expression Sets
- For each block $b$, define
  - $\text{AVAIL}(b)$ – the set of expressions available on entry to $b$
  - $\text{NKILL}(b)$ – the set of expressions not killed in $b$
  - $\text{DEF}(b)$ – the set of expressions defined in $b$ and not subsequently killed in $b$

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

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

GCSE with Available Expressions
- For each block $b$, compute $\text{DEF}(b)$ and $\text{NKILL}(b)$
- For each block $b$, compute $\text{AVAIL}(b)$
- For each block $b$, value number the block starting with $\text{AVAIL}(b)$
- Replace expressions in $\text{AVAIL}(b)$ with references

Replacement Issues
- Need a unique name for each expression in $\text{AVAIL}(b)$
- Several possibilities; all workable

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 $\text{name}(e) = e$
  - At each reference to $e$, replace $e$ with $\text{name}(e)$
Analysis

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

Computing Available Expressions

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

Computing DEF and NKILL (1)

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

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

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

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

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

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
Some Problems (1)
- Merge points often cause loss of information
- Sometimes worthwhile to clone the code at the merge points to yield two straight-line sequences

Some Problems (2)
- Procedure/function/method calls are problematic
  - Have to assume anything could happen, which kills local assumptions
  - Calling sequence and register conventions are often more general than needed
- One technique – inline substitution
  - Allows caller and called code to be analyzed together; more precise information
  - Can eliminate overhead of function call, parameter passing, register save/restore

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

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

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

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 $\Phi$-function for variables that are live in a block
**Equations for Live Variables**

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

- **Equation**
  \[
  \text{LIVE}(b) = \bigcup_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{LIVE}(s) \cap \text{NOTDEF}(s))
  \]

**Example: Available Expressions**

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

**Example: Reaching Definitions**

- 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 \(v\)

- **Uses**
  - Find all of the possible definition points for a variable in an expression

- **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

- **Equation**
  \[
  \text{REACHES}(b) = \bigcup_{p \in \text{preds}(b)} \text{DEFOUT}(p)
  \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))
  \]

**Example: Very Busy Expressions**

- 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

- **Uses**
  - Code hoisting – move \(e\) to \(p\) (reduces code size; no effect on execution time)

- **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

- **Equation**
  \[
  \text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) \cap \text{KILLED}(s))
  \]
Summary

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