CSE P 501 – Compilers

Dataflow Analysis
Hal Perkins
Winter 2008
Agenda

- Initial example: dataflow analysis for common subexpression elimination
- Other analysis problems that work in the same framework
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)
Dominator Value Numbering

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

\[
\begin{align*}
  \text{A} & : m_0 = a_0 + b_0 \\
  & \quad n_0 = a_0 + b_0 \\
  \text{B} & : p_0 = c_0 + d_0 \\
  & \quad r_0 = c_0 + d_0 \\
  \text{C} & : q_0 = a_0 + b_0 \\
  & \quad r_1 = c_0 + d_0 \\
  \text{D} & : e_0 = b_0 + 18 \\
  & \quad s_0 = a_0 + b_0 \\
  & \quad u_0 = e_0 + f_0 \\
  \text{E} & : e_1 = a_0 + 17 \\
  & \quad t_0 = c_0 + d_0 \\
  & \quad u_1 = e_1 + f_0 \\
  \text{F} & : e_2 = \Phi(e_0, e_1) \\
  & \quad u_2 = \Phi(u_0, u_1) \\
  \text{G} & : r_2 = \Phi(r_0, r_1) \\
  & \quad y_0 = a_0 + b_0 \\
  & \quad z_0 = c_0 + d_0 \\
  & \quad v_0 = a_0 + b_0 \\
  & \quad w_0 = c_0 + d_0 \\
  & \quad x_0 = e_2 + f_0
\end{align*}
\]
Available Expressions

- Goal: use dataflow analysis to find common subexpressions whose range spans basic blocks
- Idea: calculate *available expressions* at beginning of each basic block
- 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

- AVAIL(b) is the set
  \[
  \text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))
  \]

- preds(b) is the set of b’s predecessors in the control flow graph

- This gives a system of simultaneous equations – a dataflow problem
Name Space Issues

- In previous value-numbering algorithms, we used a SSA-like renaming to keep track of versions

- In global dataflow 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 to the previously computed values
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 $name(e)$
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 – DEF(b) and NKILL(b)
    - This only needs to be done once
  - Iteratively calculate AVAIL(b) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm
Computing DEF and NKILL (1)

- For each block $b$ with operations $o_1, o_2, \ldots, o_k$
  
  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
  
  ...

Computing DEF and NKILL (2)

- After computing DEF and KILLED for a block $b$,
  
  $\text{NKILL}(b) = \{ \text{all expressions} \}$
  
  for each expression $e$
  
  for each variable $\nu \in e$
  
  if $\nu \in \text{KILLED}$ then
  
  $\text{NKILL}(b) = \text{NKILL}(b) - e$
Computing Available Expressions

- Once DEF(b) and NKILL(b) are computed for all blocks b
  
  \[ \text{Worklist} = \{ \text{all blocks } b_i \} \]
  
  while (Worklist \neq \emptyset)
  
  remove a block b from Worklist
  
  recompute AVAIL(b)
  
  if AVAIL(b) changed
  
  \[ \text{Worklist} = \text{Worklist} \cup \text{successors}(b) \]
Comparing Algorithms

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

- Sometimes replicating code increases opportunities – modify the code to create larger regions with simple control flow

- Two examples
  - Cloning
  - Inline substitution
Cloning

- Idea: duplicate blocks with multiple predecessors

- Tradeoff
  - More local optimization possibilities – larger blocks, fewer branches
  - But: larger code size, may slow down if it interacts badly with cache
Example with cloning

```
m = a + b
n = a + b

A

p = c + d
r = c + d
y = a + b
z = c + d

B

q = a + b
r = c + d

C

e = b + 18
s = a + b
u = e + f

D

v = a + b
w = c + d
x = e + f

F

y = a + b
z = c + d

G

E

e = a + 17
t = c + d
u = e + f

F

v = a + b
w = c + d
x = e + f

G

y = a + b
z = c + d

G
```
Inline Substitution

- Problem: an optimizer has to treat a procedure call as if it (could have) modified all globally reachable data
  - Plus there is the basic expense of calling the procedure
- Inline Substitution: replace each call site with a copy of the called function body
Inline Substitution Issues

Pro

- More effective optimization – better local context and don’t need to invalidate local assumptions
- Eliminate overhead of normal function call

Con

- Potential code bloat
- Need to manage recompilation when either caller or callee changes
Dataflow analysis

- Global redundancy elimination is the first example of a *dataflow analysis* problem
- Many similar problems can be expressed in a similar framework
- Only the first part of the story – once we’ve discovered facts, we then need to use them to improve code
Dataflow 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
Dataflow Analysis (2)

- Usually formulated as a set of *simultaneous equations* (dataflow 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
Dataflow 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
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

For scalar values we can quickly solve simple problems
Characterizing Dataflow Analysis

- All of these algorithms 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
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 \textit{live} at point $p$ iff there is \textit{any} path from $p$ to a use of $v$ along which $v$ is not redefined.

- \textbf{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**
  \[
  \text{LIVE(b)} = \bigcup_{s \in \text{succ(b)}} \text{USED(s)} \cup \left(\text{LIVE(s)} \cap \text{NOTDEF(s)}\right)
  \]
Example: Available Expressions

- This is the analysis we did earlier to eliminate redundant expression evaluation

- Equation:
  \[
  \text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))
  \]
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
Equations for Reaching Definitions

- **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)
Equations for Very Busy Expressions

- 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
  \[
  \text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup \left( \text{VERYBUSY}(s) - \text{KILLED}(s) \right)
  \]
And so forth...

- General framework for discovering facts about programs
  - Although not the only possible story
- Next: what can we do with that information?