Global Common Subexpression Elimination *with* Data-flow Analysis

Review

So far, we have seen

- Local Value Numbering
  - → Finds redundancy, constants, & identities in a block
- Superlocal Value Numbering
  - → Extends local value numbering to EBBs
  - → Used SSA-like name space to simplify bookkeeping
- Dominator Value Numbering
  - → Extends scope to "almost" global (no back edges)
  - → Uses dominance information to handle join points in CFG

Today

- Global Common Subexpression Elimination (GCSE)
  - → Applying data-flow analysis to the problem

Today's lecture: computing AVAIL

Using Available Expressions for GCSE

The goal

Find common subexpressions whose range spans basic blocks, *and* eliminate unnecessary re-evaluations

Safety

- Available expressions proves that the replacement value is current
- Transformation must ensure right name→value mapping

Profitability

- Don't add any evaluations
- Add some copy operations

- **Copies are inexpensive**
- **Many copies <u>coalesce</u> away**
- **Copies can shrink or stretch live ranges**

*

Computing Available Expressions

For each block *b*

- Let *AVAIL(b)* be the set of expressions available on entry to *b*
- Let *EXPRKILL(b)* be the set of expression not killed in *b*
- Let *DEEXPR(b)* be the set of expressions defined in *b* and not subsequently killed in *b*

Now, *AVAIL(b)* can be defined as:

$$AVAIL(b) = \cap_{x \in pred(b)} \left(DEEXPR(x) \cup \left(AVAIL(x) \cap EXPRKILL(x) \right) \right)$$

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

This system of simultaneous equations forms a data-flow problem

→ Solve it with a data-flow algorithm

"All paths" ↑ Forward Flow Lattice Boundary condition = f

$\wedge$

$=$

f

|

t ← initial value (condition)

…xpressions for GCSE

Expressions defined in *b* and exposed downward

Expressions killed in *b*

The Method

1. ∀ block *b*, compute *DEExpr(b)* and *ExprKill(b)*
2. ∀ block *b*, compute *Avail(b)*
3. ∀ block *b*, value number the block starting from *Avail(b)*
4. Replace expressions in *Avail(b)* with references

Two key issues

- Computing *Avail(b)*
- Managing the replacement process

We'll look at the replacement issue first

Assume, w.l.og, that we can compute available expressions for a procedure.

This annotates each basic block, *b*, with a set *Avail*(*b*) that contains all expressions that are available on entry to *b*.

*

Global CSE (replacement step)

Managing the name space

Need a unique name $\forall\ e \in AVAIL(b)$

1. Can generate them as replacements are done (Fortran H)
2. Can compute a static mapping
3. Can encode value numbers into names (Briggs 94)

Strategies

1. This works; it is the classic method
2. Fast, but limits replacement to textually identical expressions
3. Requires more analysis (VN), but yields more CSEs

Assume, w.l.o.g., solution 2

Global CSE (*replacement step, strategy two*)

Compute a static mapping from expression to name

- After analysis & before transformation
  - → $\forall b, \forall e \in$ *AVAIL(b)*, assign e a global name by hashing on *e*
- During transformation step
  - → Evaluation of *e* ⇒ insert copy *name(e)* ← *e*
  - → Reference to *e* ⇒ replace *e* with *name(e)*

The major problem with this approac

- Inserts extraneous copies
  - → At all definitions and uses of any
  - → Those extra copies are dead and easy to remove
  - → The useful ones often coalesce away

Common strategy:

- **Insert copies that might be useful**
- **Let DCE sort them out**

Simplifies design & implementation

*

An Aside on Dead Code Elimination

What does "dead" mean?

- Useless code — result is never used
- Unreachable code — code that <u>cannot</u> execute
- Both are lumped together as "dead"

To perform DCE

- Must have a global mechanism to recognize usefulness
- Must have a global mechanism to eliminate unneeded stores
- Must have a global mechanism to simplify control-flow predicates

All of these will come later in the course

Global CSE

Now a three step process

- Compute *AVAIL(b)*, ∀ block *b*
- Assign unique global names to expressions in *AVAIL(b)*
- Perform replacement with local value numbering

Earlier in the lecture, we said

Assume, without loss of generality, that we can compute available expressions for a procedure.

This annotates each basic block, *b*, with a set *AVAIL*(*b*) that contains all expressions that are available on entry to *b*.

Now, we n

Computing Available Expressions

The Big Picture

1. Build a control-flow graph
2. Gather the initial (local) data — *DEEXPR(b)* & *EXPRKILL(b)*
3. Propagate information around the graph, evaluating the equation
4. Post-process the information to make it useful (*if needed*)

All data-flow problems are solved, essentially, this way

Computing Available Expressions

For each block *b*

- Let *AVAIL(b)* be the set of expressions available on entry to *b*
- Let *EXPRKILL(b)* be the set of expression <u>not killed</u> in *b*
- Let *DEEXPR(b)* be the set of expressions defined in *b* and not subsequently killed in *b*

Now, *AVAIL(b)* can be defined as:

$$AVAIL(b) = \cap_{x \in pred(b)} \left(DEEXPR(x) \cup \left(AVAIL(x) \cap \underline{EXPRKILL(x)}\,\right)\right)$$

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

This system of simultaneous equations forms a data-flow problem

→ Solve it with a data-flow algorithm

Using Available Expressions for GCSE

The Big Picture

1. ∀ block *b*, compute *DEExpr(b)* and *ExprKill(b)*
2. ∀ block *b*, compute *Avail(b)*
3. ∀ block *b*, value number the block starting from *Avail(b)*
4. Replace expressions in *Avail(b)* with references

Computing Available Expressions

First step is to compute *DEExpr* & *ExprKill*

assume a block b with operations $o_1, o_2, \ldots, o_k$

VarKill ← Ø
DEExpr(*b*) ← Ø

for *i* = *k* to *1* — Backward through block

> *assume o_i is "x ← y + z"*
> add *x* to VarKill
> if (*y* ∉ VarKill) and (z ∉ VarKill) then
> add "*y + z*" to DEExpr(*b*)

O(k) steps

ExprKill(*b*) ← Ø

For each expression *e*

> for each variable *v* ∈ *e*
> if v ∈ VarKill(*b*) then
> ExprKill(*b*) ← ExprKill(*b*) ∪ {*e*}

O(N) steps

N is # operations

Many data-flow problems have initial information that costs less to compute

Computing Available Expressions

The worklist iterative algorithm

Worklist $\leftarrow$ { all blocks, b_i}

while (*Worklist* $\neq$ Ø)
remove a block *b* from *Worklist*
recompute AVAIL(*b*) as

$$\textit{AVAIL}(b) = \cap_{x \in pred(b)} (\textit{DEEXPR}(x) \cup (\textit{AVAIL}(x) \cap \overline{\textit{EXPRKILL}(x)}))$$

if AVAIL(*b*) changed then
Worklist $\leftarrow$ Worklist $\cup$ *successors*(*b*)

- Finds fixed point solution to equation for AVAIL
- That solution is unique
- Identical to "meet over all paths" solution

How do we know these things?

*

Data-flow Analysis

Data-flow analysis is a collection of techniques for *compile-time* reasoning about the *run-time* flow of values

- Almost always involves building a graph
  - → Problems are trivial on a basic block
  - → Global problems ⇒ control-flow graph (or derivative)
  - → Whole program problems ⇒ call graph (or derivative)
- Usually formulated as a set of *simultaneous equations*
  - → Sets attached to nodes and edges
  - → Lattice (or semilattice) to describe values
- Desired result is usually *meet over all paths* solution
  - → "What is true on every path from the entry?"
  - → "Can this happen on any path from the entry?"
  - → Related to the safety of optimization

Flow graph

Data-flow problem

Data-flow Analysis

Limitations

1. Precision – *"up to symbolic execution"*
   - → Assume all paths are taken
2. Solution – cannot afford to compute MOP solution
   - → Large class of problems where MOP = MFP= LFP
   - → Not all problems of interest are in this class
3. Arrays – treated naively in classical analysis
   - → Represent whole array with a single fact
4. Pointers – difficult (*and expensive*) to analyze
   - → Imprecision rapidly adds up
   - → Need to ask the right questions

Summary

For scalar values, we can quickly solve simple problems

Good news:

Simple problems can carry us pretty far

*

Computing Available Expressions

$$\textit{AVAIL}(b) = \bigcap_{x \in pred(b)} (\textit{DEEXPR}(x) \cup (\textit{AVAIL}(x) \cap \overline{\textit{EXPRKILL}(x)}))$$

where

- *EXPRKILL(b)* is the set of expression not killed in *b*, and
- *DEEXPR(b)* is the set of downward exposed expressions in *b* (defined and not subsequently killed in *b*)

Initial condition

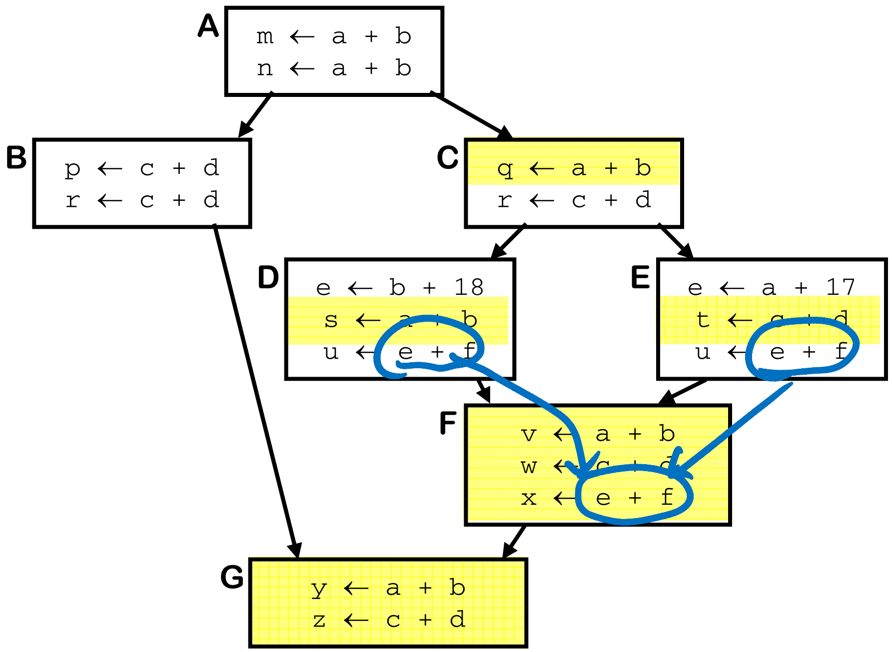
$\textit{AVAIL}(n_0) = \emptyset$, because nothing is computed before n_0

The other node's *AVAIL* sets will be computed over their *preds*.

n_0 has no predecessor.

Making Theory Concrete

Computing AVAIL for the example



	A	B	C	D	E	F	G
DEEXPR	a+b	c+d	a+b,c+d	b+18,a+b,e+f	a+17,c+d,e+f	a+b,c+d,e+f	a+b,c+d
EXPRKILL	{ }	{ }	{ }	e+f	e+f	{ }	{ }

AVAIL(A) = Ø

AVAIL(B) = {a+b} ∪ (Ø ∩ all)
= {a+b}

AVAIL(C) = {a+b}

AVAIL(D) = {a+b, c+d} ∪ ({a+b} ∩ all)
= {a+b, c+d}

AVAIL(E) = {a+b, c+d}

AVAIL(F) = [{b+18, a+b, e+f} ∪
({a+b, c+d} ∩ {all - e+f})]
∩ [{a+17, c+d, e+f} ∪
({a+b, c+d} ∩ {all - e+f})]
= {a+b, c+d, e+f}

AVAIL(G) = [{c+d} ∪ ({a+b} ∩ all)]
∩ [{a+b, c+d, e+f} ∪
({a+b, c+d, e+f} ∩ all)]
= {a+b, c+d}

Redundancy Elimination Wrap-up

Algorithm	Acronym	Credits
Local Value Numbering	LVN	Balke, 1967
Superlocal Value Numbering	SVN	Many
Dominator-based Value Num'g	DVNT	Simpson, 1996
Global CSE (with AVAIL)	GCSE	Cocke, 1970
SCC-based Value Numbering†	SCCVN/VDCM	Simpson, 1996
Partitioning Algorithm†	AWZ	Alpern et al, 1988

… and there are many others …

Three general approaches

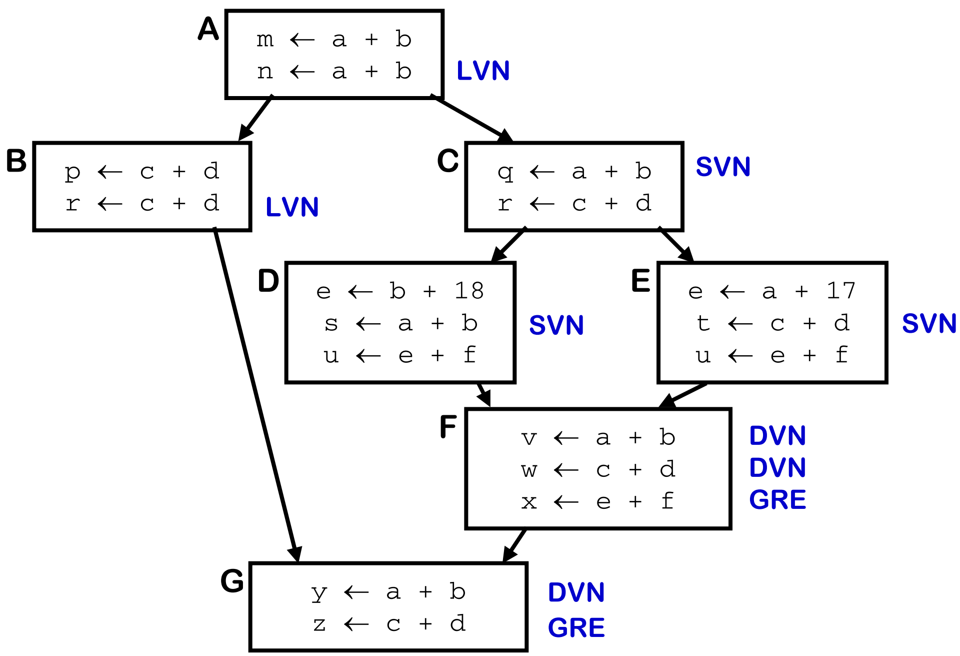
- **Hash-based, bottom-up techniques**
- **Data-flow techniques**
- **Partitioning**

Each has strengths & weaknesses

†We have not seen these ones (yet).

Making Theory Concrete

Comparing the techniques



The VN methods are ordered

- **LVN ≤ SVN ≤ DVN** (≤ SCCVN)
- **GRE is different**
  - **Based on names, not value**
  - **Two phase algorithm**
    - → **Analysis**
    - → **Replacement**

Redundancy Elimination Wrap-up

Comparisons

Name	Scope	On/Off Line	Operates On	Basis of Identity
LVN	local	online	blocks	value
SVN	superlocal	online	EBBs	value
DVNT	regional	online	dom. Tree	value
GCSE	global	offline	CFG	lexical

Name	Visits per Node	Commutes	Algebraic Identities	Constants	Optimistic
LVN	1	yes	yes	yes	n/a
SVN	1	yes	yes	yes	n/a
DVNT	1	yes	yes	yes	n/a
GCSE	D(CFG) + 3	no	no	no	no

Better results in loops

The partitioning method based on DFA minimization

Redundancy Elimination Wrap-up

Generalizations

- Hash-based methods are fastest
- AWZ (& SCCVN) find the most cases
- Expect better results with larger scope

Experimental data

- Ran LVN, SVN, DVNT, AWZ
- Used global name space for DVNT
  - → Requires offline replacement
  - → Exposes more opportunities
- Code was compiled with lots of optimization

How did they do?

- → DVNT beat AWZ
- → Improvements grew with scope
- → DVNT vs. SCCVN was ± 1%
- → DVNT 6x faster than SCCVN
- → SCCVN 2.5x faster than AWZ

*

Redundancy Elimination Wrap-up

Conclusions

- Redundancy elimination has some depth & subtlety
- Variations on names, algorithms & analysis matter
- Compile-time speed does not have to sacrifice code quality

DVNT is probably the method of choice

- Results quite close to the global methods (± 1%)
- Much lower costs than SCCVN or AWZ

Example

|LIVE| = |*variables*|

Transformation: Eliminating unneeded stores

- *e* in a register, have seen last definition, never again used
- The store is dead (*except for debugging*)
- Compiler can eliminate the store

Form of *f* is same as in AVAIL

Data-flow problem: Live variables

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

- LIVE(b) is the set of variables live on exit from b
- NOTDEF(b) is the set of variables that are not redefined in b
- USED(b) is the set of variables used before redefinition in b

Compute as DEF(b)

Live analysis is a backward flow problem

LIVE plays an important role in both register allocation and the pruned-SSA construction.

*