Global Common Subexpression Elimination

with Data-flow Analysis

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
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 coalesce away
  • Copies can shrink or stretch live ranges

*
Computing Available Expressions

For each block $b$

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

Now, $\text{AVAIL}(b)$ can be defined as:

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

$\text{preds}(b)$ 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 Method

1. \( \forall \) block \( b \), compute \( DEEXPR(b) \) and \( EXPRKILL(b) \)
2. \( \forall \) block \( b \), compute \( AVAIL(b) \)
3. \( \forall \) 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.o.g, 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 $\forall e \in \text{AVAIL}(b)$
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
  \[\forall b, \forall e \in \text{AVAIL}(b), \text{assign } e \text{ a global name by hashing on } e\]

- During transformation step
  \[\text{Evaluation of } e \Rightarrow \text{insert copy } \text{name}(e) \leftarrow e\]
  \[\text{Reference to } e \Rightarrow \text{replace } e \text{ with } \text{name}(e)\]

The major problem with this approach:

- Inserts extraneous copies
  \[\text{At all definitions and uses of any } e \in \text{AVAIL}(b), \text{assign } e \text{ a global name by hashing on } e\]

Common strategy:
- Insert copies that might be useful
- Let DCE sort them out
  Simplifies design & implementation

\[\forall b, \forall e \in \text{AVAIL}(b), \text{assign } e \text{ a global name by hashing on } e\]
An Aside on Dead Code Elimination

What does “dead” mean?
- Useless code — result is never used
- Unreachable code — code that cannot 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 $AVAIL(b)$, \( \forall \) 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 \( b \).
Computing Available Expressions

The Big Picture

1. Build a control-flow graph
2. Gather the initial (local) data — \(\text{DEExpr}(b) \& \text{EXPRKill}(b)\)
3. Propagate information around the graph, evaluating the equation
4. Post-process the information to make it useful \(\text{(if needed)}\)

All data-flow problems are solved, essentially, this way
Computing Available Expressions

For each block $b$

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

Now, $\text{AVAIL}(b)$ can be defined as:

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

$\text{preds}(b)$ 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. \( \forall \) block \( b \), compute \( DEExpr(b) \) and \( ExprKill(b) \)
2. \( \forall \) block \( b \), compute \( Avail(b) \)
3. \( \forall \) 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 \( \text{DEExpr} \) & \( \text{EXPRKILL} \)

\begin{itemize}
	\item Assume a block \( b \) with operations \( o_1, o_2, \ldots, o_k \)
	\item \( \text{VARKILL} \leftarrow \emptyset \)
	\item \( \text{DEExpr}(b) \leftarrow \emptyset \)
	\item Backward through block
	\item For \( i = k \) to 1
	\item Assume \( o_i \) is “\( x \leftarrow y + z \)"
	\item Add \( x \) to \( \text{VARKILL} \)
	\item If \( (y \notin \text{VARKILL}) \) and \( (z \notin \text{VARKILL}) \) then
	\item Add “\( y + z \)” to \( \text{DEExpr}(b) \)
	\end{itemize}

\begin{itemize}
	\item \( \text{EXPRKILL}(b) \leftarrow \emptyset \)
	\item \( \text{O}(k) \) steps
	\item For each expression \( e \)
	\item For each variable \( v \in e \)
	\item If \( v \in \text{VARKILL}(b) \) then
	\item \( \text{EXPRKILL}(b) \leftarrow \text{EXPRKILL}(b) \cup \{ e \} \)
	\end{itemize}

\( \text{O}(N) \) steps

\( N \) is \# operations

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

*
Computing Available Expressions

The worklist iterative algorithm

$Worklist \leftarrow \{ \text{all blocks, } b_i \}$

while $(Worklist \neq \emptyset)$
    remove a block $b$ from $Worklist$
    recompute $AVAIL(b)$ as
    
    $AVAIL(b) = \bigcap_{x \in \text{pred}(b)} (DEEXPR(x) \cup (AVAIL(x) \cap EXPRKILL(x)))$
    
    if $AVAIL(b)$ changed then
        $Worklist \leftarrow Worklist \cup \text{successors}(b)$

• 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
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
Computing Available Expressions

\[ \text{AVAIL}(b) = \cap_{x \in \text{pred}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) \cap \text{EXPRKILL}(x))) \]

where

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

Initial condition

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

The other node’s \( \text{AVAIL} \) sets will be computed over their \( \text{preds} \). \( n_0 \) has no predecessor.
**Making Theory Concrete**

**Computing AVAIL for the example**

AVAIL(A) = ∅
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

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

... 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
- \( \text{LVN} \leq \text{SVN} \leq \text{DVN} \leq \text{SCCVN} \)
- GRE is different
  - Based on names, not value
  - Two phase algorithm
    - Analysis
    - Replacement
# Redundancy Elimination Wrap-up

## Comparisons

<table>
<thead>
<tr>
<th>Name</th>
<th>Scope</th>
<th>Line</th>
<th>Operates</th>
<th>Basis of Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVN</td>
<td>local</td>
<td>online</td>
<td>blocks</td>
<td>value</td>
</tr>
<tr>
<td>SVN</td>
<td>superlocal</td>
<td>online</td>
<td>EBBs</td>
<td>value</td>
</tr>
<tr>
<td>DVNT</td>
<td>regional</td>
<td>online</td>
<td>dom. Tree</td>
<td>value</td>
</tr>
<tr>
<td>GCSE</td>
<td>global</td>
<td>offline</td>
<td>CFG</td>
<td>lexical</td>
</tr>
</tbody>
</table>

Better results in loops

<table>
<thead>
<tr>
<th>Name</th>
<th>Visits per Node</th>
<th>Commutes</th>
<th>Algebraic Identities</th>
<th>Constants</th>
<th>Optimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVN</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>n/a</td>
</tr>
<tr>
<td>SVN</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>n/a</td>
</tr>
<tr>
<td>DVNT</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>n/a</td>
</tr>
<tr>
<td>GCSE</td>
<td>D(CFG) + 3</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
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

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

Data-flow problem: Live variables
\[
\text{LIVE}(b) = \bigcup_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{LIVE}(s) \cap \text{NOTDEF}(s))
\]
• $\text{LIVE}(b)$ is the set of variables live on exit from $b$
• $\text{NOTDEF}(b)$ is the set of variables that are not redefined in $b$
• $\text{USED}(b)$ is the set of variables used before redefinition in $b$

Live analysis is a backward flow problem

\[
|\text{LIVE}| = |\text{variables}|
\]

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