# CS 4120 Lecture 27 — Redundancy Elimination — 6 November 2009

Lecturer: Andrew Myers

An optimization aims at *redundancy elimination* if it prevents the same computation from being done twice.

## 1 Local value numbering

Local value numbering is a redundancy elimination optimization. It is called a "local" optimization because it is performed on a single basic block. It can as easily be applied to an *extended basic block* (EBB), which is a sequence of nodes in which some nodes are allowed to have exit edges, as depicted in Figure 1. More generally, it can be applied to any tree-like subgraph in which all nodes have only one predecessor and there is a single node that dominates all other nodes. Each path through such a subgraph forms an EBB.

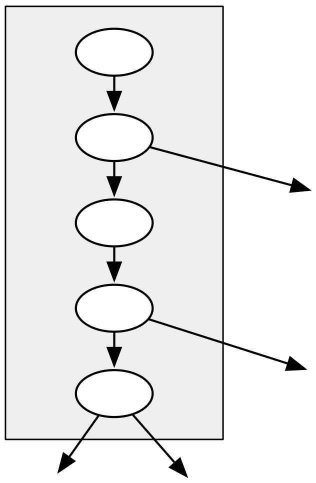

Figure 1: Extended basic block

The idea of value numbering is to label each computed expression a distinct identifier (a "number"), such that a recomputation of the same expression is assigned the same number. If an expression is labeled with the same number as a variable, the variable can be used in place of the expression. Value numbering overlaps but does not completely subsume constant propagation and common subexpression elimination.

For example, every expression and variable in the following the code has been given a number. Expressions with the same number must compute the same value.

$a_2$ = $(i_1 + 5)_2$
$j_1$ = $i_1$
$b_2$ = $5 + j_1$
$i_2$ = $i_1 + 5$
if $c_4$ = $(i_2 + 1)_3$ goto $L_1$
$d_3$ = $(i_2 + 1)_3$

To do this numbering, we start at the beginning of the EBB and assign numbers to expressions as they are encountered. As expressions are assigned numbers, the analysis remembers the mapping between that expression, expressed in terms of the value numbers it uses, and the new value number. If the same expression is seen again, the same number is assigned to it.

In the example, the variable `i` is given number 1. Then the sum `i+1` is given number 2, and the analysis records that the expression (*value*(1) + 1) is the same as *value*(2). The assignment to `a` means that variable has value 2 at that program point. Assignment `j=i` puts value 1 into `j`, so `j` is also given number 1 at that program point. The expression `5 + j` computes (*value*(1) + 1), so it is given number 2. This means we can replace `5+j` with `a`, eliminating a redundant computation.

After doing the analysis, we look for value numbers that appear multiple times. An expression that uses a value computed previously is replaced with a variable with the same value number. If no such variable exists, a new variable is introduced at the first time the value was computed, to be used to replace later occurrences of the value.

For example, the above code can be optimized as follows:

$$
\begin{aligned}
&\mathtt{a}_2 = (\mathtt{i}_1 + 5)_2\\
&\mathtt{j}_1 = \mathtt{i}_1\\
&\mathtt{b}_2 = \mathtt{a}_2\\
&\mathtt{i}_2 = \mathtt{a}_2\\
&\mathtt{t}_3 = (\mathtt{a}_2 + 1)_3\\
&\mathtt{if}\ \mathtt{c}_4 = \mathtt{t}_3\ \mathtt{goto}\ \mathtt{L}_1\\
&\mathtt{d}_3 = \mathtt{t}_3
\end{aligned}
$$

The mapping from previously computed expressions to value numbers can be maintained implicitly by generating value numbers with a strong hash function. For example, we could generate the actual representation of value 2 by hashing the string "$\mathtt{plus(v_1,\ const(1))}$", where $v_1$ represents the hash value assigned to value 1. Note that to make sure that we get the same value number for the computation `5+j`, it will be necessary to order the arguments to commutative operators (e.g., +) in some canonical ordering (e.g., sorting in dictionary order).

There are "global" versions of value numbering that operate on a general CFG. However, these are awkward unless the CFG is converted to single static assignment (SSA) form.

## 2 Common subexpression elimination

Common subexpression elimination (CSE) is a classic optimization that replaces redundantly computed expressions with a variable containing the value of the expression. It works on a general CFG. An expression is a *common subexpression* at a given node if it is computed in another node that dominates this one, and none of its operands have been changed on any path from the dominating node.

For example, in Figure 2, the expression `a+1` is a common subexpression at the bottom node. Therefore, it can be saved into a new temporary `t` in the top node, and this temporary can be used in the bottom one.

It is worth noting that CSE can make code slower, because it may increase the number of live variables, causing spilling. If there is a lot of register pressure, the reverse transformation, *forward substitution*, may improve performance. Forward substitution copies expressions forward when it is cheaper to recompute them than to save them in a variable.

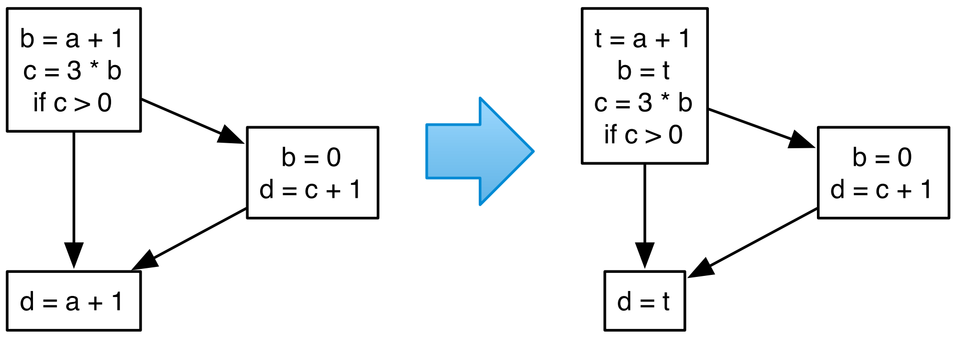

Figure 2: Common subexpression elimination

## 2.1 Available expressions analysis

An expression is *available* if it has been computed in a dominating node and its operands have not been redefined. The *available expressions* analysis finds the set of such expressions. Implicitly, each such expression is tagged with the location in the CFG that it comes from, to allow the CSE transformation to be done.

Available expressions is a forward analysis. We define $out(n)$ to be the set of available expressions on edges leaving node $n$. An expression is available if it was evaluated at $n$, or was available on all edges entering $n$, and it was not killed by $n$:

$$in(n) = \bigcap_{n' < n} out(n)$$
$$out(n) = in(n) \cup eval(n) - kill(n)$$

Therefore, dataflow values are sets of expressions ordered by $\subseteq$; the meet operator is set intersection ($\cap$), and the top value is the set of all expressions, usually implemented as a special value that acts as the identity for $\cap$.

The expressions evaluated and killed by a node, $eval(n)$, are summarized in the following table. Note that we try to include memory operands as expressions subject to CSE, because replacing memory accesses with register accesses is a useful optimization.

| $n$ | $eval(n)$ | $kill(n)$ |
|---|---|---|
| $x = e$ | $e$ and all subexpressions of $e$ | all expressions containing $x$ |
| $[e_1] = [e_2]$ | $[e_2]$, $[e_1]$, and subexpressions | all expressions $[e']$ that can alias $[e_1]$ |
| $x = f(\vec{e})$ | $\vec{e}$ and subexpressions | expressions containing $x$ and expressions $[e']$ that could be changed by function call to $f$ |
| `if` $e$ | $e$ and subexpressions | $\emptyset$ |

If a node $n$ computes an expression $e$ that is used and available in other nodes, the optimization proceeds as follows:

1. In the node that the available expression came from, add a computation $t = e$, and replace the use of $e$ with $t$.

2. Replace expression $e$ in other nodes where it is used and available with $t$.

CSE can work well with copy propagation. For example, the variable `b` in the above code may become dead after copy propagation.

## 3 Partial redundancy elimination

CSE eliminates computation of *fully redundant* expressions: those computed on all paths leading to a node. Partially redundant expressions are those computed at least twice along some path, but not necessarily all paths. Partial redundancy elimination (PRE) eliminates these partially redundant expressions. PRE subsumes CSE and loop-invariant code motion.

Figure 3 shows an example of PRE. The computation `b+c` is redundant along some paths but not others. To make it fully redundant, we place computation of `b+c` onto earlier edges so that it has always been computed at each point where it is needed.

### 3.1 Lazy code motion

The idea of lazy code motion is to eliminate all redundant computations while avoid creating any unnecessary computation: computations are moved earlier in the CFG. Further, we want to make sure that although the computations are moved earlier in the CFG, they are postponed as long as possible, to avoid creating register pressure.

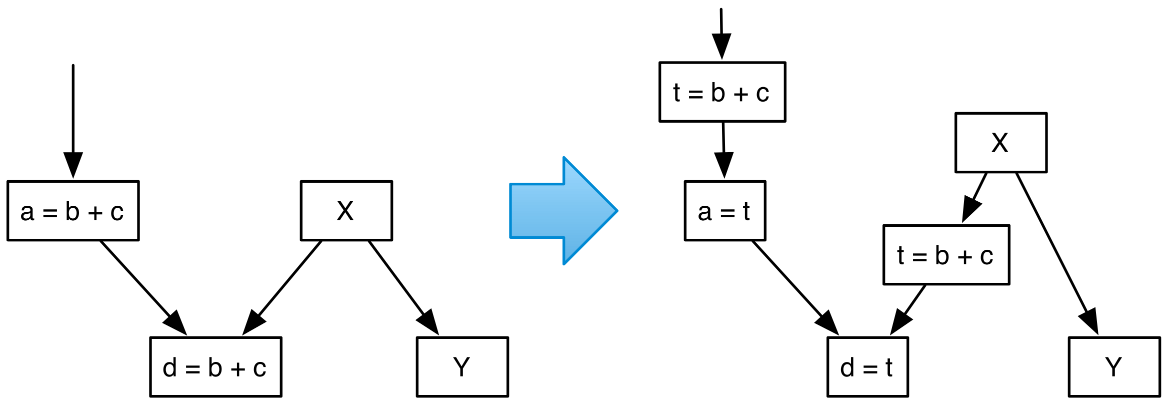

Figure 3: Partial redundancy elimination

The approach is to first identify candidate locations where the partially redundant expression could have been moved in order to make it fully redundant, without creating extra computations. Then among these candidates we choose the one that comes latest along each path that needs it.

## 3.2 Anticipated expressions

The *anticipated expressions* analysis (also known as *very busy expressions*) find expressions that are needed along every path leaving a given node. If an expression is needed along every path leaving the node, then there can be no wasted computation if the expression is moved to that node.

This is a backward analysis, in which the dataflow values are sets of expressions and the meet operator is $\cap$.

Once we know the anticipated expressions at each node, we tentatively place computations of these expressions and use an available expressions analysis to find expressions that are fully redundant under the assumption that the anticipated expressions are computed everywhere anticipated. These fully redundant expressions are the expressions to which we can apply the PRE optimization.

## 3.3 Postponable expressions

At this point we know some set of nodes where the expression can be moved, and we know where it is used. We need to pick a set of edges that separate these two parts of the CFG, and put the computation of the expression on those edges. We want to postpone the computation as long as possible. The *postponable expressions* analysis finds expressions $e$ that are are anticipated at program point $p$ but not yet used: every path from the start to $p$ contains an anticipation of $e$ and no use before $p$. This is a forward analysis with meet operator $\cap$.

Once postponable expressions have been computed, certain edges form a *frontier* where the expression transitions from postponable to not postponable. It is on these edges that the new node computing the expressions is placed.