CS 4120/5120 Lecture 28 More loop optimizations 8 April 2016
Lecturer: Andrew Myers

1 Bounds check removal

In type-safe languages, accesses to array elements generally incur a bounds check. Before accessing `a[i]`, the language implementation must ensure that `i` is a legal index. For example, in Java array indices start at zero, so the language must test that $0 \leq$ `i` $<$ `a.length`.

Returning to our example from earlier, after strength reduction we can expect the code to look more like the following (or the equivalent CFG):

```
s = a + 3*i;
while (i < a.length) {
    j = s;
    if (i < 0 | i ≥ a.length) goto L_err;
    [j] = [j] + 1;
    i = i + 2;
    s = s + 6;
}
```

This extra branch inside the loop is likely to add overhead. Furthermore, it prevents the induction variable elimination operation just discussed.

One simple improvement we can make is to implement the check $0 \leq$ `i` $<$ `n` in a single test. Assuming that `n` is a signed positive integer, and `i` is a signed integer, this test can be implemented by doing an *unsigned* comparison `i<n`. If `i` is negative, it will look like a large unsigned integer that will fail the unsigned comparison, as desired. Processor architectures have a unsigned comparison mode that supports this. For example, the `jae` instruction ("jump above or equal") on the Intel architecture implements unsigned comparison.

Even better would be to eliminate the test entirely. The key insight is that the loop guard (in this case, `i` $<$ `a.length`) often ensures that the bounds check succeeds. If this can be determined statically, the bounds check can be removed. If it can be tested dynamically, the loop can be split into two versions, a fast version that does not do the bounds check and a slow one that does.

The bounds-check elimination works under the following conditions:

1. Induction variable j has a test ($j < u$) vs. a loop-invariant expression u, where the loop is exited if the test failed.

2. Induction variable k in the same family as j has a test equivalent to $k < n$ vs. a loop-invariant expression n, where $j < u$ implies $k < n$, again exiting the loop on failure. The test on k must be dominated by the test on j, and k and j must go in the same direction (both increase or both decrease).

Under these conditions, the bound checks on k is superfluous and can be eliminated. But when does $j < u$ imply $k < n$? Suppose that $j = \langle i, a_j, b_j \rangle$ and $k = \langle i, a_k, b_k \rangle$. If the j test succeeds, then $a_j i + b_j < u$. Without loss of generality, assume $a_j > 0$. Then this implies that $i < (u - b_j)/a_j$. Therefore, $k = a_k i + b_k < a_k(u - b_j)/a_j + b_k$. If we can show statically or dynamically that this right-hand side is less than or equal to n, then we know $k < n$. So the goal is to show that $a_k(u - b_j)/a_j + b_k \leq n$. This can be done either at compile time or by hoisting a test before the loop. In our example, the test for `i` $<$ `a.length` is that $1 * (\texttt{a.length} - 0)/1 + 0 \leq \texttt{a.length}$, which can be determined statically. The compiler does still need to insert a test that $i \geq 0$ before the loop:

```
s = a + 3*i;
if (i < 0) goto L_err;
while (i < a.length) {
    j = s;
    [j] = [j] + 1;
    i = i + 2;
    s = s + 6;
}
```

After linear-function test replacement, this code example will become subject to induction variable elimination.

2 Loop unrolling

Loop guards and induction variable updates add significant overhead to short loops. The cost of loop guards involving induction variables can often be reduced by *unrolling* loops to form multiple consecutive copies. Multiple loop guard tests can be then be combined into a single conservative test. If the loop is unrolled to make n copies, and the loop guard has the form $i < u$ where u is a loop-invariant expression and i is an induction variable with stride c that is updated at most once per loop iteration, the test $i + c(n-1) < u$ conservatively ensures that all n loop copies can be run without having any guard fail.

Since this guard is conservative, there still may be $< n$ iterations left to be performed. So the unrolled loop has to be followed by another copy of the original loop, the *loop epilogue*, which performs any remaining iterations one by one. Since loop unrolling therefore results in $n + 1$ copies of the original loop, it trades off code size for speed. If the original loop was only going to be executed for a small number of iterations, loop unrolling could hurt performance rather than improve it.

Updates to basic linear induction variables inside the unrolled loop can also be combined. If a variable i is updated with the statement $i = i + c$, then n copies of the update can be replaced with a single update $i = i + nc$. However, any uses of i within the copies of the loop must be changed as well – in the second loop copy, to $i + c$, in the third copy (if any), to $i + 2c$, and so on. These additions can often be folded into existing arithmetic computations.

3 Redundancy elimination

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

4 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.

The idea of value numbering is to label each computed expression with a distinct identifier (a "number"), such that recomputations of the same expression are 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 code has been given a number, written as a subscript. Expressions with the same number always compute the same value.

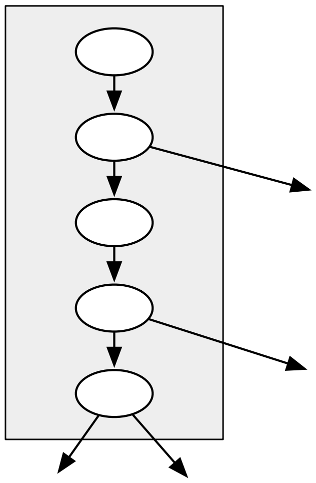

Figure 1: Extended basic block

```
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:

```
a_2 = (i_1 + 5)_2
j_1 = i_1
b_2 = a_2
i_2 = a_2
t_3 = (a_2 + 1)_3
if c_4 = t_3 goto L_1
d_3 = t_3
```

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 "`plus(v1, 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.

5 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.

5.1 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:

$$in(n) = \bigcap_{n' \prec n} out(n)$$
$$out(n) = in(n) \cup exprs(n) - kill(n)$$

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, $exprs(n)$, 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.

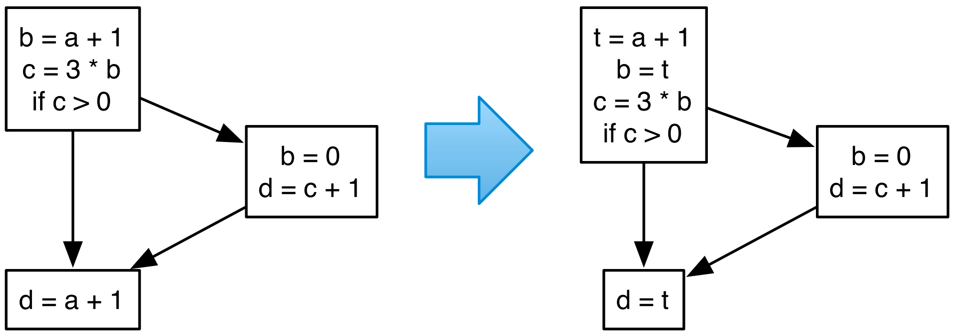

Figure 2: Common subexpression elimination

n	$exprs(n)$	$kill(n)$
$x = e$	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. However, CSE plus copy propagation can enable more CSE, because CSE only recognizes syntactically identical expressions as the same. Copy propagation can make semantically identical expressions look the same through its renaming of variables. It is possible to generalize Available Expressions to keep track of equalities more semantically, though this makes the analysis much more complex.