Suppression of Partial Redundancies

Max Hailperin

May 2, 2003

If any path through a program's control flow graph includes two or more computations of an expression without any intervening assignments to the variables used in that expression, then we have an opportunity to improve the program. This general notion subsumes the classical optimizations of loop invariant motion and global common subexpression elimination. For example, in the flow graph fragments shown in figures 1 and 2, we can arrange to only compute $b + c$ once. In addition to these cases, there are some other circumstances in which an expression is needlessly calculated more than once along some path. For example, in the fragment shown in figure 3, the left-hand path calculates $b + c$ twice. This is called a *partial redundancy*, and can be remedied as shown in that figure.

Interestingly, we can use a single optimization technique to uniformly handle all three of these cases. This is because all three are forms of partial redundancy, in that in each case a node computes a value that is already available along at least one of the paths leading to that node. Therefore, an algorithm for suppressing partial redundancies also serves to move invariants out of loops, even though the algorithm embodies no notion of loop. A slight variant on the algorithm can also do strength reduction, but we won't consider that yet, in order to keep the presentation simple.

To simplify matters, we'll focus exclusively on eliminating redundant computations of the expression $b+c$. This leads to boolean-valued dataflow problems to decide questions such as "Can $b + c$ be safely computed here?" in place of set-valued problems such as "Which expressions can be safely

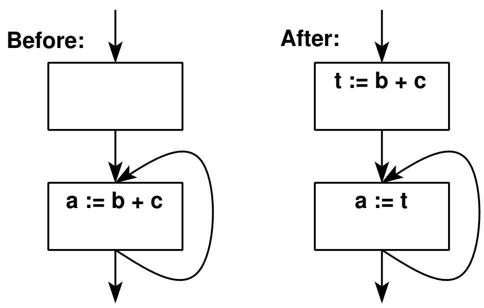

Figure 1: Loop invariant motion

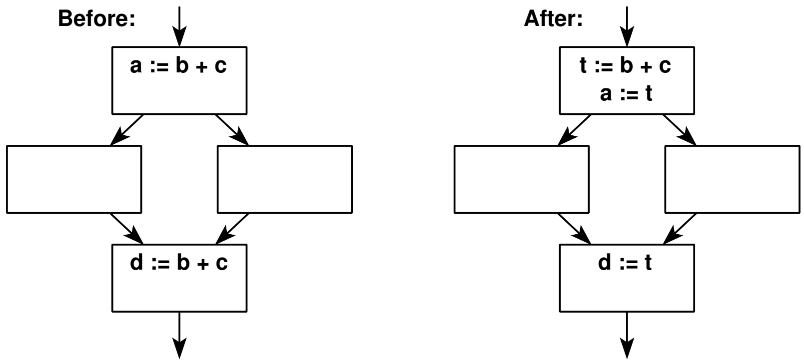

Figure 2: Global common subexpression elimination

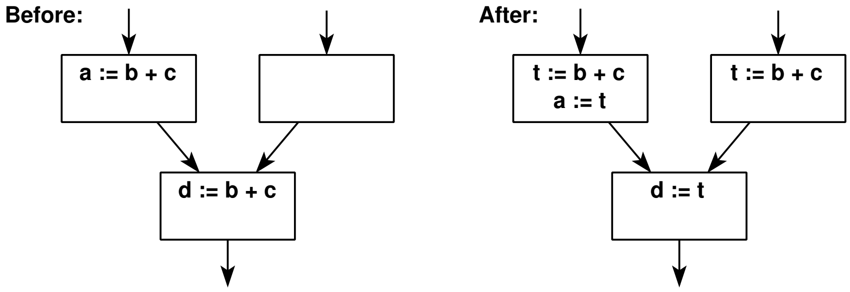

Figure 3: Suppression of a partial redundancy

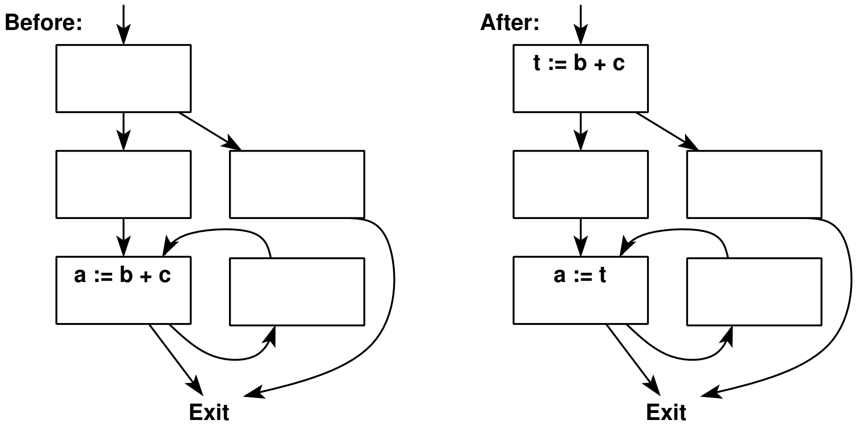

Figure 4: Too early computation; unused on one path

computed here?" Further, we'll assume that all nodes in the flow graph initially contain at most one assignment and that the predecessors of a join node are empty. These restrictions allows us to place the new computations uniformly at the top of nodes. None of these simplifications is difficult to remove once the basic process is understood. One other restriction, however, can not be removed (in general) without impairing optimization: there must never be an edge directly from a branch node to a join node. However, any such edge can easily be eliminated by adding an intermediate empty node.

Down-safe nodes

In general we'll need to move computations earlier in the flow graph, as we did in figures 1 and 3. However, we want to be sure not to move a computation to such an early point that it might wind up unused. For example, in figure 4, the computation of $b+c$ has been moved to too early a point, since if execution proceeds along the right-hand path, the value will be unused. Even if the expression is used along all paths from a node to the program exit, we may be in trouble if one of the variables used in the expression gets re-assigned first. For example, in figure 5, the expression $b+c$ is used on all paths from the top node to the exit; yet computing $b+c$ in the top node might be a waste, because along the right-hand path b is changed before $b+c$ is used.

To make this notion precise, we will call a node of the flow graph *down-safe* if a computation of $b+c$ at the beginning of that node would definitely be used before either b or c is changed. (Some other authors instead say the expression is "anticipatable" or "very busy" at the node.) We can use a dataflow analysis to determine which nodes are down-safe. In order to do this, we'll define two other predicates on nodes, namely *Used* and *Transparent*.

Used is true of a node if $b+c$ is used in that node; in this case, the node

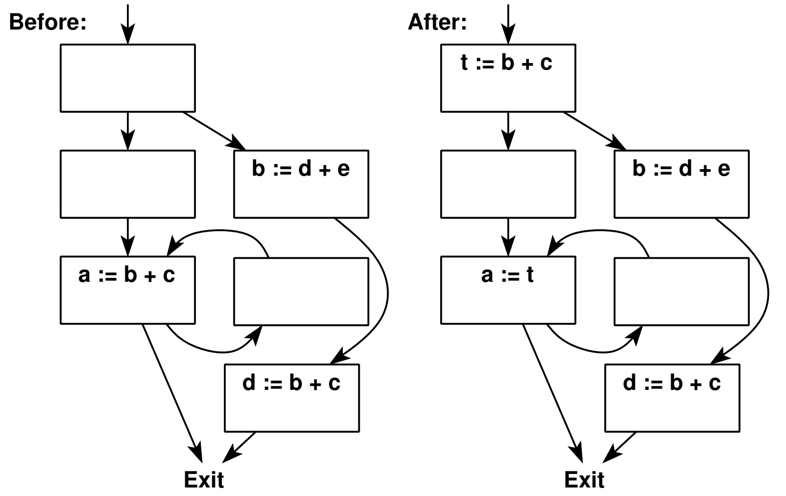

Figure 5: Too early computation; killed on one path

is definitely down-safe. A node is transparent provided neither b nor c is assigned in that node. If a node is transparent, and all its successors are down-safe, then it is down-safe itself. However, the special exit node of the program is definitely not down-safe.

Writing these facts in symbols we have for each node n in the flow graph

$$\textit{D-safe}(n) = \begin{cases} \textit{false} & \text{if } n = \textit{exit} \\ \textit{Used}(n) \vee (\textit{Transp}(n) \wedge \bigwedge_{s \in \textit{Succ}(n)} \textit{D-safe}(s)) & \text{otherwise} \end{cases}$$

Since the flow graph is in general cyclic, this system of equations defines down-safety only implicitly as a fixed point. In particular, the desired solution is the greatest fixed point, which can be found by iteration starting from the initial assumption that all nodes except the exit are down-safe.

Up-safe nodes

Although any down-safe node is an acceptable place to evaluate the expression, some may be preferable to others. In particular, our goal is to reduce the number of evaluations by carefully choosing among the down-safe nodes. The next section will address this problem; for now, we tackle a sub-problem, namely identifying the *up-safe* nodes.

Just as a node is down-safe for an expression if that expression will definitely be used thereafter without being killed first, a node is up-safe for an expression if that expression definitely has already been used, without being killed in the meantime. (Many other authors instead say the expression is "available" at the node.)

It should be clear that it is undesirable to re-compute the expression at an up-safe node, since along all paths leading to that node, the value

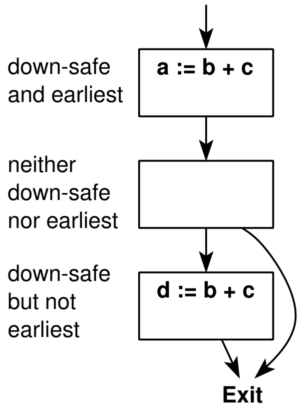

Figure 6: An example of *Earliest*

could have be saved the last time it was computed and then at the up-safe node re-used instead of being re-computed. It is possible to show that the program transformation described in this handout will in fact never call for the computation of the expression at an up-safe node. (This is left as an exercise for the reader.)

We will denote the up-safety predicate by *U-safe*. It can be found as the greatest fixed point of

$$\mathit{U\text{-}safe}(n) = \begin{cases} \mathit{false} & \text{if } n = \mathit{start} \\ \bigwedge_{p \in \mathit{Pred}(n)} (\mathit{Transp}(p) \wedge (\mathit{Used}(p) \vee \mathit{U\text{-}safe}(p))) & \text{otherwise} \end{cases}$$

Earliest down-safe nodes

As noted above, we want to choose among the down-safe nodes those to compute the expression at, in a way that will minimize the number of computations. It can be shown that the number of evaluations is minimized by choosing those down-safe nodes that are as early as possible in the flow graph. Precisely, we want those down-safe nodes that are either the start node or have an opaque predecessor or have a predecessor that is neither up-safe nor down-safe:

$$\mathit{Earliest}(n) = \mathit{D\text{-}safe}(n) \wedge \begin{cases} \mathit{true} & \text{if } n = \mathit{start} \\ \bigvee_{p \in \mathit{Pred}(n)} (\overline{\mathit{Transp}(p)} \vee (\overline{\mathit{U\text{-}safe}(p) \vee \mathit{D\text{-}safe}(p)})) & \text{otherwise} \end{cases}$$

If you are unsure why *U-safe* has to appear on the right of the equation, consider figure 6, in which the bottom node's predecessor is up-safe. Thus, even though that predecessor node isn't down-safe, the bottom node isn't earliest.

We now know how to minimize the number of computations of $b + c$. We can insert $t \leftarrow b + c$ at the beginning of each node that is earliest and

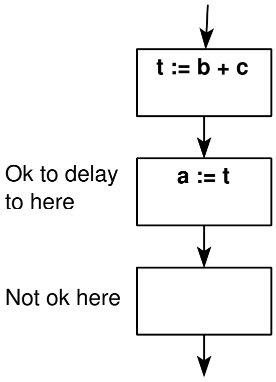

Figure 7: A computation can't be delayed past where it is used

replace all the original instances of $b + c$ by t. (This presumes that t is a new name not otherwise in use.)

Latest nodes as good as earliest safe nodes

Unfortunately, the placement choice described above (the earliest safe nodes) may be overkill. By doing all computations as early as possible, we maximize the lifetime of the stored values and thus maximize the demand for registers. This could cause some registers to be spilled to slower memory. Meanwhile, it is possible that a later placement of the expression might not be executed any more frequently—all we know is that it wouldn't be any less frequent.

This leads us to consider how much later the expressions can be placed without increasing the number of evaluations. One reason why we might not be able to move a computation any later is because the value is used in the node in which it is computed. For example, the computation of $b + c$ in figure 7 can be moved from the first node to the second, but not to the third.

Another circumstance under which a computation shouldn't be moved later is illustrated in figure 8; delaying the computation in the top left node until the join node would cause a redundant computation when the right-hand path was taken. Formalizing these notions, we'll find the greatest solution to

$$\mathit{Delay}(n) = \mathit{Earliest}(n) \vee \begin{cases} \mathit{false} & \text{if } n = \mathit{start} \\ \bigwedge_{p \in \mathit{Pred}(n)} (\overline{\mathit{Used}(p)} \wedge \mathit{Delay}(p)) & \text{otherwise} \end{cases}$$

Having identified the nodes to which delaying is reasonable, we simply select the latest of those, i.e. those that either include a use of $b + c$ or that

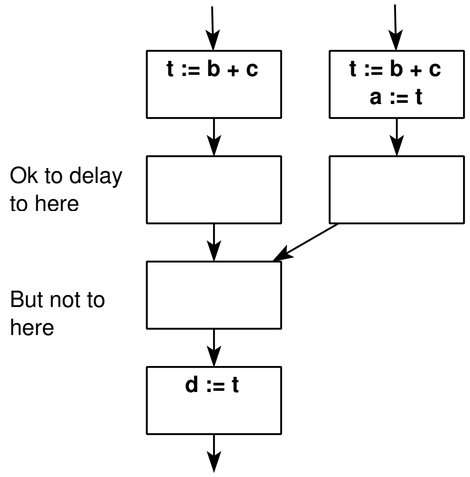

Figure 8: A join can deter delaying

have a successor to which delaying isn't possible:

$$Latest(n) = Delay(n) \wedge (Used(n) \vee \bigvee_{s \in Succ(n)} \overline{Delay(s)})$$

These latest nodes are where the computations of $t \leftarrow b + c$ should be inserted.

Strength reduction

This same algorithm can also do strength reduction in either of two ways:

1. We can broaden our notion of which nodes are transparent, i.e. weaken our notion of which assignments kill a value. Suppose that we are considering computations of the expression $i * n$. Rather than saying that an assignment of the form $i \leftarrow i + 1$ kills this expression, we can say that it merely "injures" $i * n$. An injured value can be healed by addition. In this example, if t holds $i * n$, it can be healed by adding n to it. Simply introducing these notions into the algorithm for suppressing partial redundancies allows some multiplications to be replaced by additions; however, it also can cause some paths to contain both an addition and a multiplication, or to have multiple additions. By doing some additional analysis, these problems can be overcome.

2. Alternatively, we can treat the placement of full-cost computations like $t \leftarrow i * n$ and the placement of lower-cost updates like $t \leftarrow t + n$ as being special cases of the general problem of cost-optimally placing members of a family of equivalent computations. (At the point where we would place $t \leftarrow t + n$, it is equivalent to $t \leftarrow i * n$.) Then the full-cost computations and the updates can both be placed using a unified

algorithm for cost-optimal placement. This unified scheme generalizes the algorithm described earlier by choosing placement points between the earliest and latest.

Notes

Morel and Renvoise [6] originated the notion of suppressing partial redundancies. This paper also showed that loop invariant motion and global common sub-expression elimination are special cases of suppression of partial redundancies.

The particular algorithm described here is a more efficient version due to Knoop, Rüthing, and Steffen [5]. One small difference in the version presented here is that the published version avoids inserting assignments that are only used in their own node. Not only would that have cluttered our presentation, but it also is of questionable utility, since the register allocation phase of the compiler can completely eliminate the apparent cost of these unnecessary assignments.

Joshi and Dhamdhere [2] proposed the first method of incorporating strength reduction into the suppression of partial redundancies. Knoop, Rüthing, and Steffen [4] showed how to implement this in their version of the algorithm (actually, in their earlier variant [3]), including the refinements necessary to combine multiple additions and prevent addition and multiplication from being done on the same path. Hailperin [1] proposed the alternative method using cost-optimal placement.

References

[1] Max Hailperin. Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems*, 20(6):1297–1322, November 1998.

[2] S. M. Joshi and D. M. Dhamdhere. A composite hosting-strength reduction transformation for global program optimization. *International Journal of Computer Mathematics*, 11:21–41, 111–126, 1982.

[3] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234, 1992.

[4] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, 1993.

[5] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[6] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.