Corner cases in value-based partial redundancy elimination CSD TR# 03-032

Thomas VanDrunen Antony L Hosking

November 4, 2003

In this technical report, we discuss extensions to our GVN-PRE algorithm [2] needed for full optimality. In terms of Knoop et al's categories [1], we need partial anticipation to fully cover *code motion*; We need a system of predicates to analyze hypothetical insertions in order for this optimization to be *code placement*.

The basic algorithm uses the following flow equations.

$$\begin{aligned}
\mathsf{AVAIL_IN}[b] &= \mathsf{AVAIL_OUT}[\mathsf{dom}(b)] && (1)\\
\mathsf{AVAIL_OUT}[b] &= \mathsf{canon}(\mathsf{AVAIL_IN}[b] \cup \mathsf{PHI_GEN}(b) && (2)\\
&\qquad \cup \mathsf{TMP_GEN}(b))
\end{aligned}$$

$$\begin{aligned}
\mathsf{ANTIC_OUT}[b] &= \begin{cases} \{e \mid e \in \mathsf{ANTIC_IN}[\mathsf{succ}_0(b)] \wedge \\ \quad \forall\ b' \in \mathsf{succ}(b), \exists e' \in \mathsf{ANTIC_IN}[b'] \mid \\ \quad \mathsf{lookup}(e) = \mathsf{lookup}(e')\} & \text{if } |\mathsf{succ}(b)| > 1 \\ \\ \mathsf{phi_translate}(A[\mathsf{succ}(b)], b, \mathsf{succ}(b)) & \text{if } |\mathsf{succ}(b)| = 1 \end{cases} && (3)\\
\mathsf{ANTIC_IN}[b] &= \mathsf{clean}(\mathsf{canon}_e(\mathsf{ANTIC_OUT}[b] \cup \mathsf{EXP_GEN}[b] && (4)\\
&\qquad -\mathsf{TMP_GEN}(b)))
\end{aligned}$$

$$\begin{aligned}
\mathsf{phi_translate}(S, b, b') &= \{\mu(e, S, b, b') \mid e \in S\} && (5)\\
\mu(t, S, b, b') &= \begin{cases} t_i \text{ if } \overline{b} = \mathsf{pred}(b') \wedge b = b_i \wedge t \leftarrow \phi(\overline{t}) \in b' \\ t \text{ otherwise} \end{cases} && (6)\\
\mu(v_1 \ \mathsf{op}\ v_2, S, b, b') &= \mathsf{lookup}(\mu(e_1, S, b, b'))\ \mathsf{op}\ \mathsf{lookup}(\mu(e_2, S, b, b')), && (7)\\
&\quad \text{where } e_1, e_2 \in S, \mathsf{lookup}(e_1) = v_1, \mathsf{lookup}(e_2) = v_2\\
\mathsf{clean}(S) &= \{e \mid e \in S, \mathsf{live}(e, S)\} && (8)\\
\mathsf{live}(v_1\ \mathsf{op}\ v_2, S) &= (\exists e \in S \mid \mathsf{live}(e, S), \mathsf{lookup}(e) = v_1) && (9)\\
&\quad \wedge (\exists e \in S \mid \mathsf{live}(e, S), \mathsf{lookup}(e) = v_2)
\end{aligned}$$

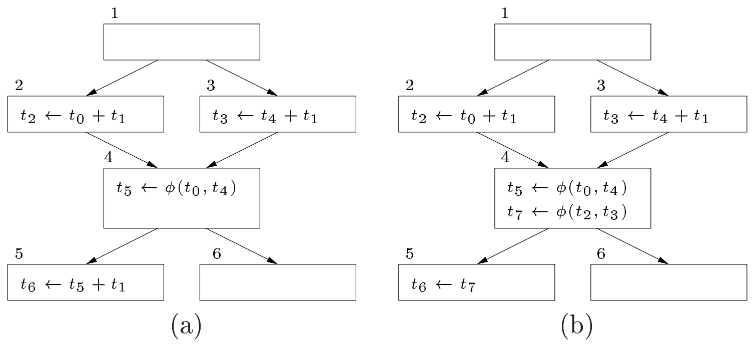

Figure 1: The need for partial anticipation

1 Improving precision: partial anticipation

Consider the program in Figure 1(a). The instruction $t_6 \leftarrow t_5 + t_1$ in block 5 is fully redundant, although there is no available expression at that point. A no-cost phi inserted at block 4, however, would allow us to remove that computation. See Figure 1(b). However, our algorithm would not perform this insertion, because the expression is not anticipated at block 4.

To improve on this, we define *partial anticipation*. A value is partially anticipated at a program point if it is computed on at least one but not all path to program exit. We will define flow equations for calculating $\mathsf{PA\_IN}$ and $\mathsf{PA\_OUT}$ sets for block which are similar to those for $\mathsf{ANTIC\_IN}$ and $\mathsf{ANTIC\_OUT}$, except that they also depend on on $\mathsf{ANTIC\_IN}$ and $\mathsf{ANTIC\_OUT}$, and they require a value-wise union instead of intersection. The versions of $\mathsf{phi\_translate}$ and clean, $\mathsf{dep\_phi\_trans}$ and $\mathsf{dep\_clean}$ respecrively, rely on $\mathsf{ANTIC\_IN}$.

$$
\begin{aligned}
\mathsf{PA\_OUT} &= \begin{cases} \mathsf{canon}_e\Big(\bigcup\limits_{b' \in \mathsf{succ}(b), \neg b' \gg b} (\mathsf{ANTIC\_IN}[b'] \cup \mathsf{PA\_IN}[b'])\Big) & \\ \quad -\mathsf{ANTIC\_OUT}[b] & |\mathsf{succ}(b)| > 1 \\ \mathsf{dep\_phi\_trans}(\mathsf{PA\_IN}[\mathsf{succ}(b)], & \\ \qquad \mathsf{ANTIC\_IN}[\mathsf{succ}(b)]) & |\mathsf{succ}(b)| = 1 \end{cases} \\
\mathsf{PA\_IN}[b] &= \mathsf{dep\_clean}(\mathsf{canon}_e(\mathsf{PA\_OUT}[b] - \mathsf{TMP\_GEN}[b] \\
&\qquad -\mathsf{ANTIC\_IN}[b], \mathsf{ANTIC\_IN}[b])) \qquad (10)
\end{aligned}
$$

Other definitions:

$$
\begin{aligned}
\mathsf{dep\_phi\_trans}(S, S', b, b') &= \{\mu'(e, S, S', b, b') | e \in S\} && (11) \\
\mu'(t, S, S', b, b') &= \mu(t, S, b, b') && (12) \\
\mu'(v_1 \ \mathsf{op}\ v_2, S, S', b, b') &= \mathsf{lookup}(\mu'(e_1, S, S', b, b'))\ \mathsf{op}\ \mathsf{lookup}(\mu'(e_2, S, S', b, b')), && (13) \\
&\quad \text{where } e_1, e_2 \in S \cup S', \mathsf{lookup}(e_1) = v_1, \mathsf{lookup}(e_2) = v_2 && \\
\mathsf{dep\_clean}(S, S') &= \{e | e \in S, (\mathsf{live}(e, S) \wedge \mathsf{live}(e, S'))\} && (14)
\end{aligned}
$$

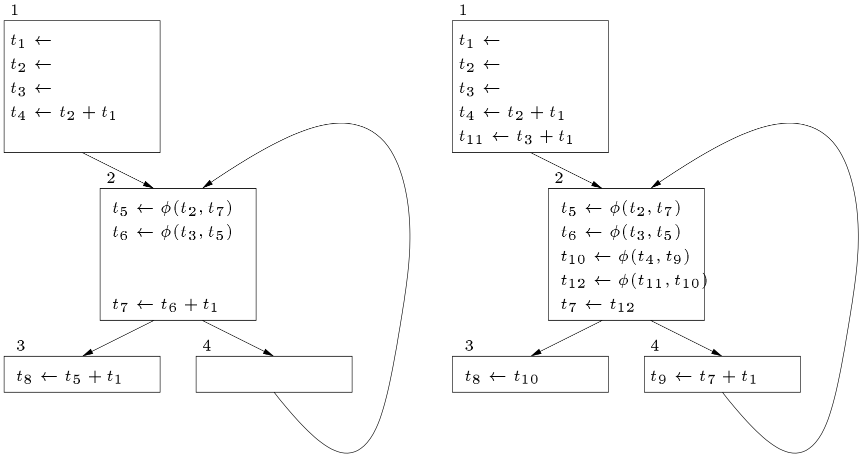

Figure 2: Why partial anticipation is tough

(15)

To implement partial anticipation, it would be very inefficient to union all successor full and partial anticipation sets, then choose canonical representatives, and then to delete those that are fully anticipated, as the flow equation suggests. Rather, iterate through each expression in ANTIC_IN and PA_IN for each sucessor, and if the expression's value is not already in PA_OUT or ANTIC_OUT, we add it to PA_OUT.

Notice that the flow equation for PA_OUT where there is more than one sucessor excludes passing partially anticipated expressions across a backedge (that is, when the sucessor dominates the block). This is necessary for termination. Consider the program in Figure 2(a). In the body of the loop, a value is incremented each iteration, but the result is rotated around t_5 and t_6, so that the value from a previous iteration is preserved. The computation $t_5 + t_1$ is redundant in all cases except when the loop body is executed exactly twice: if the loop is executed only once, its value is stored in t_4; if it is executed more than twice, its value was computed in the third-to-last loop execution. Assuming we name values with the same numbering as the temporaries which represent them, the expression $v_5 + v_1$ is fully anticipated into block 3. On a first pass, it is not anticipated into block 4, so it is partially anticipated in block 2. If we were to propagate it accross the back edge 4-2, it would become $v_7 + v_1$ in block 4, which would require a new value, say v_9. On the next pass, it would become partially anticipated in block 2. If we propagated partial anticipation through the backedge again on the next iteration, it would become $v_8 + v_1$ since v_7 is represented by $v_6 + v_1$ which maps to $v_5 + v_1$, representing v_8. This new expression would again need a new value; this process would recurr infinitely.

Disallowing partial anticipation propagation through a back edge, however, does not prevent us from identifying and eliminating the redundancy in this case. On the second pass, block 4 anticipated $v_5 + v_1$, making that expression now fully anticipated in block 2. Since it is partially available (t_4 from block 1), we insert $t_9 \leftarrow t_7 + t_1$ in block 4. This insertion makes the value for $v_6 + v_1$ partially available, and since it is fully anticipated at the beginning of block 1, we insert $t_{11} \leftarrow t_3 + t_1$ in block 1. The phis $t_{10} \leftarrow \phi(t_4, t_9)$ and $t_{12} \leftarrow \phi(t_{11}, t_{10})$ allow us to eliminate the computations for t_7 and t_8, shortening the single-loop interation scenario. See Figure 2(b).

For Insert, we add an extra step for each join point where we iterate over partially anticipated expressions.

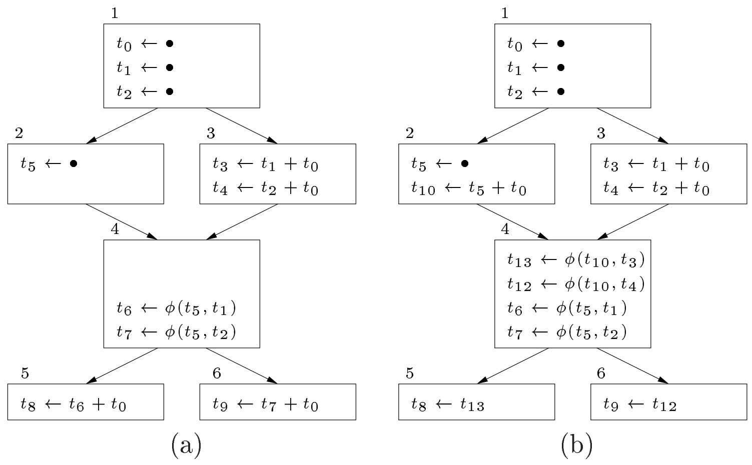

Figure 3: Code placement

We do not want to make insertions if something is only partially anticipated, so we create a phi only if the equivalent value exists in all predecessors We need an extra variable *by_all* which we set to false if we inspect a predecessor where find_leader returns null. We insert only if *by_all* is true.

2 Improving precision: Knoop et al's frontier

2.1 Problem

Our goal is to eliminate redundant computations without introducing spurious computations. Therefore as our algorithm makes changes in the program, we make sure never to lengthen any program path by a computation, though we may lengthen it by moves and phis. So far, our rule for merging values (or leaders of the same value) in a phi is that such a merge is appropriate at a join point either if the corresponding expression is fully anticipated and at least partially available or if it is at least partially anticipated and fully available. In the former case, we are assured that any insertion we may need to make will allow an elimination later in all paths from that point. In the later case, we are assured that no insertions of computations are necessary.

Although this approach is safe, it is not complete. Consider the program in Figure 3(a). At the join point in block 3, the expression $t_6 + t_0$ is partially anticipated. It is available from block 2: translated through the phis, it becomes $t_1 + t_0$, the leader of whose value it t_3. It is not available in block 1. The expression $t_7 + t_0$ is also partially anticipated, and it also is available from block 2: translated through the phis, it becomes $t_2 + t_0$ in block 2, the leader of whose value is t_4. It is not available in block 1. Partially anticipated and partially available, these do not qualify for optimization in the algorithm defined so far. However, they can be optimized according to our goals. Both of the expressions translate through the phis to $t_5 + t_0$. If we insert that computation in block 1 and store it in t_{10}, we can use t_{10} as the operand to two phis, one to merge it with t_3 and knock out the computation $t_6 + t_0$, the other to merge it with t_4 and knock out $t_7 + t_0$. See Figure 3(b).

2.2 Solution

An enhancement to our algorithm covering these cases is complicated but still elegant. Consider what is happening at the join point. The computation needed at the join point is $t_5 + t_0$. The question is, should we insert that computation? Since it is not equivalent to any fully anticipated expression, we know it will not be used in block 4. Therefore the answer is, it should be inserted if it is used in (all paths from) block 5 and (all paths from) block 6. How do we know if these will indeed be used? It will be used on block 5 if we make a phi for $t_6 + t_0$; it will be used on block 6 if we make a phi for $t_7 + t_0$. How do we know if we should indeed make these phis? We should make a phi for a given expression if it is partially available (which is true in our cases) and is fully hypothetically available (in other words, if it is already available or will be after safe insertions). Thus an expression at a join point is "fully hypothetical" if on every predecessor either it is already available or we will insert a computation for it.

In the words of a children's song, there is a hole in the bucket. Whether or not we insert a computation ultimately depends on whether or not we will insert it. To state the situation formally, suppose π stands for an expression that is partially anticipated and partially available (a "partial-partial"). Let u range over expressions which would be the right-hand side of hypothetical instructions, possibly to be inserted. Suppose that for each partial-partial π, we have a map μ_π which associates predecessors to expressions for hypothetical instructions. Suppose further that for each expression for a hypothetical instruction u, we have a map μ_u which associates successors with sets of partial-partials. We then have the following system of predicates:

$$
\begin{aligned}
\mathsf{will\_insert}(u) &= \forall\ s \in \mathsf{succ}, \mathsf{used}(u, s) && (16)\\
\mathsf{used}(u, s) &= \exists\ \pi \in \mu_u(s) | \mathsf{make\_phi}(\pi) && (17)\\
\mathsf{make\_phi}(\pi) &= \mathsf{part\_avail}(\pi) \wedge \mathsf{full\_hyp}(\pi) && (18)\\
\mathsf{part\_avail}(\pi) &= \exists\ p \in \mathsf{pred} | \mathsf{avail}(p, \pi) && (19)\\
\mathsf{avail}(\pi, p) &= \mathsf{find\_leader}(\mathsf{ANTIC\_IN}[p], \mathsf{lookup}(\mathsf{equivalent\_expression}(\mathsf{succ}(p), p, \pi))) \\
&\quad \neq \mathsf{null} && (20)\\
\mathsf{full\_hyp}(\pi) &= \forall p \in \mathsf{pred}, \mathsf{avail\_or\_hyp}(\pi, p) && (21)\\
\mathsf{avail\_or\_hyp}(\pi, p) &= \mathsf{avail}(p, \pi) \vee \mathsf{will\_insert}(\mu_\pi(p)) && (22)\\
& && (23)
\end{aligned}
$$

The system is recursive, and if will_insert is true, the recursion will be infinite. We believe the converse is true. Thus we search for a contradiction to this system, and consider it to be true if no contradiction is found. In terms of implementation, will_insert should return true if it is reentered for the same argument.

2.3 Algorithm

To implement this solution, we need a small change to BuildSets: we need to build and maintain a map, *part_succ_map*, which associates an expression with the set of blocks on which the partially anticipated expression is fully anticipated but which are successors to blocks on which it is partially anticipated. Note that this might not be simply a single block; if a block has three successors and an expression is anticipated on two of the three, we need to know about both. Thus *part_succ_map* is not a simple map associating a key with a single entry but a multi-map associating a key to a set of entries. Thus we assume another operation on this map, set_add, which takes a map, a key, and an entry; instead of overwriting any older entry for the given key, it adds that entry to the set of entries for that key.

More needs to be done to Insert. We need a map (*hyp_map*) which will associate hypothetical computations with maps (μ_u) from successors to sets of partial-partials, and a map (*part_part_map*) which will associate partial-partials with maps (μ_π) from predecessors to hypothetical computations. While doing insert for partially anticipated expressions in a given block, if an expression is also partially available (if it is a partial-partial), take note of it by making a μ_π for it; for each predecessor, μ_π should have a simple expression already available or a hypothetical computation which could be inserted. Each hypothetical computation should have its μ_u, which we populate by the successors of the current block found among the blocks at which the current partial-partial is fully anticipated. After iterating through the partially anticipated expressions, we consider what hypothetical computations should have insertions and which partial-partials consequently should phis be made for.

References

[1] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Code motion and code placement: Just synonyms? *Lecture Notes in Computer Science*, 1381, 1998.

[2] Thomas VanDrunen and Antony L Hosking. Value-based partial redundancy elimination. Submitted, 2004.