Lazy Code Motion

Max Hailperin

2009-05-04

Step 1: Preliminaries

This is a simplified variant of the dragon book's version of Lazy Code Motion (LCM), Algorithm 9.36, which in turn was based on Knoop, Rüthing, and Steffen's original algorithm as they presented it in a 1992 conference paper [1]. The revised version in their 1994 journal article [2] is perhaps simpler, but sticking close to our course textbook seems desirable.

Compared with the textbook, the additional simplification is that we will focus on occurrences of a single expression; for example, we might focus just on reducing the number of times `a*b` is computed. Thus, rather than needing to ask what set of expressions is anticipated in each block (and similar questions), we can just ask what blocks the expression is anticipated in.

We also will make explicit an assumption that is sometimes omitted in the textbook: the entry and exit blocks play no other role. That is, both are empty, the entry block has no predecessors, and the exit block has no successors.

We will also retain two other simplifying assumptions from the textbook. First, each basic block contains at most one statement prior to the code-motion transformation. Second, whenever a block has multiple predecessors, each of those predecessors must be an empty block with only a single successor. (Step 1 of Algorithm 9.36 ensures this second property by inserting empty blocks.)

Assume that the expression being optimized is `a*b` and that the name `t` is not in use. Our goal, then, is to replace some occurrences of `a*b` with `t` and insert the statement `t = a*b` at the start of some blocks. Along the way to LCM, we will encounter two closely related code-motion transformations, Busy Code Motion (BCM) and Almost Lazy Code Motion (ALCM), which make different choices as to where these replacements and insertions should take place.

If a block contains an occurrence of `a*b` prior to the optimization, we say it *uses* the expression. If the block contains an assignment to either `a` or `b`, we say it *kills* the expression. Note that any block that both uses the expression and kills it will always use the expression first, before killing it, because of our assumption that each block contains only one statement.

Step 2: Computing "anticipated in"

To compute the list of blocks where the expression is "anticipated in" (that is, anticipated at the top of the block), perform the following steps:

1. Write down all the block numbers.
2. Cross off the exit block.
3. Cross off any block that kills the expression and doesn't also use it.
4. Repeat as long as any changes occur: for any crossed-off block, cross off any predecessor that doesn't use the expression.

Step 3: Computing "available in"

Next we compute the list of blocks where the expression is "available in" (that is, available at the top of the block). However, note that the dragon book is using a special definition of "available" for LCM that is different from Section 9.2's definition. To compute the blocks where the expression is "available in" using this special definition:

1. Write down all the block numbers.
2. Cross off the entry block.
3. Cross off the successors of blocks that kill the expression.
4. Repeat as long as any changes occur: for any crossed-off block that is also crossed-off in the "anticipated in" list, cross off each successor.

Step 4: Computing "earliest"

To compute the blocks that are "earliest," just list those that were crossed off from the "available in" list but left uncrossed on the "anticipated in" list.

At this point, we have all the information to do BCM. In BCM, all of the original uses of the expression are replaced by the new temporary, `t`. The new computations `t = a*b` are inserted at the top of those blocks that are earliest. This eliminates redundant computations, but may do some computations earlier than there is any reason to.

Step 5: Computing "postponable in"

To compute the list of blocks where the expression is "postponable in" (that is, postponable at the top of the block), perform the following steps:

1. Write down all the block numbers.
2. Cross off the entry block.
3. Cross off the successors of blocks that use the expression.
4. Repeat as long as any changes occur: for any crossed-off block that is not listed as "earliest," cross off each successor.

Step 6: Computing "latest"

First, as a preliminary, make a list of "candidate" blocks by listing all those blocks that show up on either the "earliest" list or the "postponable in" list (not crossed off).

Second, list the blocks that are "latest" as follows: write down those block numbers that appear on the "candidate" list and either use the expression or have a successor that is not a candidate.

At this point, we have all the information to do ALCM. As in BCM, all of the original uses of the expression are replaced by the new temporary, `t`. The new computations `t = a*b` are inserted at the top of those blocks that are latest. This eliminates just as much redundant computations as BCM's placement, without doing any computations earlier than there is any reason to. The only remaining defect is that in some cases the expression may be computed into `t` when the only use of that value is in the very next instruction, within the same block. In this case, the computation isn't unnecessarily early, but the temporary is unnecessarily employed.

Step 7: Computing “used out”

To compute the list of blocks where the expression is “used out” (that is, would prove useful at the bottom of the block), perform the following steps:

1. For each block that uses the expression and isn’t latest, add its predecessors to the “used out” list (if they aren’t yet on it).

2. Repeat as long as any changes occur: for any listed block that isn’t latest, add its predecessors to the list (if they aren’t yet on it).

Step 8: The LCM transformation

The insertions and replacements are essentially the same as those listed earlier for ALCM. However, for any block in which ALCM would perform both insertion and replacement, those changes should be suppressed if the block is not listed as “used out.”

References

[1] 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.

[2] 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.