An Implementation of Lazy Code Motion for Machine SUIF

Laurent Rolaz

Swiss Federal Institute of Technology
Processor Architecture Laboratory
Lausanne
28th March 2003

1 Introduction

Optimizing compiler often attempts to eliminate redundant computations either by removing instructions or by moving instructions to less frequently executed locations. The *code removing* optimization attempts to determine when two instructions compute the same value, and then decides if one of the instructions can be eliminated. The *code motion* optimization rely on data-flow analysis to determine the set of locations where each computation will produce the same value and to select the ones that are expected to be least frequently executed.

Lazy Code Motion (LCM) is an algorithm developed by Knoop, Rüthing and Steffen [3, 4]. It is a descendant of partial redundancy elimination that avoids unnecessary code motion. Drechsler and Stadel present a variation of this technique that is more practical [2], thus this version will be implemented. LCM operates at the global scope and can eliminate redundancies in a single procedure. The base idea of this algorithm is to compute points in the flow graph where the insertion of the evaluation of a term t will make all present evaluation of t redundant. Then the algorithm eliminates all of the original evaluation. If the algorithm attempts to insert an evaluation into a block that already contains an evaluation, neither the insertion nor the deletion will be performed.

Section 2 will quickly present the partial redundancy elimination optimization. Then section 3 will be a detailed description based on the implementation of the LCM algorithm as a Machine SUIF optimization pass.

2 Partial Redundancy Elimination

Partial redundancy elimination (PRE) is an optimization introduced by Morel and Renvoise [5] that combines common subexpresssion elimination (CSE) with loop invariant code motion. Partially redundant expression are redundant along some execution paths, but not necessarily all. In general, PRE moves code upward in the procedure to the earliest point where the computation would produce that same value without lengthening any path through the program.

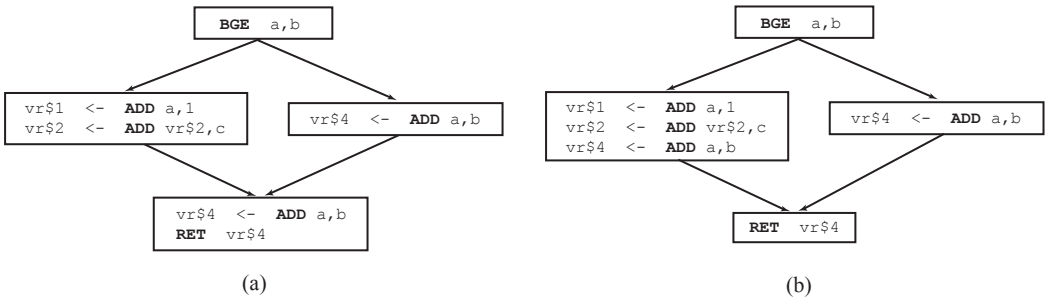

Figure 1: Program improved by partial redundancy elimination. (a) shows a partially redundant computation of the term **`ADD a,b`** in the last block. (b) moves it upward to a less frequently executed point.

3 Description of the Algorithm

Computing the point at which to insert evaluation of a term t is a two-step process. First the algorithm computes the earliest points in the flow graph where evaluations of t can be inserted without violating safety or profitability. In fact, the algorithm guarantees the minimal number of evaluations of t on all paths. Then the insertion points are pushed later on each path until the last points where the insertions can occur without increasing the number of executed evaluations. The second step avoids unnecessary code motion. This features is important when code motion interacts with register allocation and other optimization. Each replacement affects register allocation because it has the potential of shortening the live range of a term's operands and lengthening the live range of its target temporary.

3.1 Boolean Properties Associated with Terms

For each term, boolean properties are defined. Some of these properties are computed locally in a block, and other properties depend on interactions of different blocks and are called global.

3.1.1 Local Properties

The local properties are *transparency*, *availability* and *anticipability*.

Transparency A term is said to be *transparent* in a block B it its operands are not modified by the execution of all the instructions in B:

```
bool LazyCodeMotion::Transp(CfgNode *BB, Instr *T) {
  // check each instruction in BB and see if one of the operands of T
  // are modified by the current instruction
  for (InstrHandle h = start(BB); h != end(BB); ++h) {
    Instr *instr = *h;
    if (UsesInstr(T, instr))
      return false;  // T is modified by instr
  } // end for
  return true; // no modification detected
}
```

Local Availability A term t is said to be *locally available* in a block B if there is at least one evaluation of t in the block B, and if the instructions appearing in the block after the last evaluation of t do not modify its operands:

```
bool LazyCodeMotion::Comp(CfgNode *BB, Instr *T) {
  // check each instruction in BB from end to start
  for (InstrHandle h = last(BB); h != end(BB); --h) {
    Instr *instr = *h;
    // found a computation of T
    if (SameTerm(instr, T))
      return true;
    // T is modified before a computation of T was found
    if (UsesInstrSR(T, instr))
      return false;
  } // end for
  return false;   // no computation detected
}
```

Local availability of a term grants that the last evaluation of this term in the block will deliver the same result as would an evaluation of this term placed at the end of the block.

Local Anticipability A term t may be locally anticipated in a block B if there is at least one evaluation of t in B, and if the instructions appearing in B before the first evaluation of t do not modify its operands:

```
bool LazyCodeMotion::Antloc(CfgNode *BB, Instr *T) {
  // check each instruction in BB from start to end
  for (InstrHandle h = start(BB); h != end(BB); ++h) {
    Instr *instr = *h;
    // found a computation of T
    if (SameTerm(instr, T))
      return true;
    // T is modified before a computation of T was found
    if (UsesInstrSR(T, instr))
      return false;
  } // end for
  return false;   // no computation detected
}
```

Local anticipability of a term grants that the first evaluation of this term in the block will deliver the same result as would an evaluation of this term placed at the beginning of the block.

3.1.2 Global Properties

The meaning of availability and anticipability can be extended to a complete program. The availability of a term at a given point implies that an evaluation of this term placed at this point would deliver the same results as the last computation of this term made before this point. Similarly, the anticipability of a term at a given point implies that an evaluation of this term placed at this point would deliver the same results as the first evaluation of this term made after this point. In practice, we will concentrate on points which are block entries or exists.

Because of these two insertion points, we have to refine the properties of availability and anticipability. The global properties are availability on entry of a block (`AVIN`), anticipability on entry of a block (`ANTIN`), availability on exit of a block (`AVOUT`), and anticipability on exit

of a block (ANTOUT). The relations between global and local properties are easily expressed in the form of systems of boolean equations. There are many approaches to solve these systems. The algorithm implemented in our pass is based on an iterative approach that use work list. For more practical informations see [6].

Global Availability A term t is available on entry to a block B if it is available on exit from each predecessor of B. A term is available on exit from a block B if it is locally available or if it is available on entry of the block and transparent in this block:

$$\begin{aligned} \texttt{AVIN}_B(t) &= \begin{cases} \texttt{false} & \text{if } B \text{ is the entry block} \\ \prod_{P_i \in Pred(B)} \texttt{AVOUT}_{P_i}(t) & \text{otherwise} \end{cases} \\ \texttt{AVOUT}_B(t) &= \texttt{COMP}_B(t) + \texttt{TRANSP}_B(t) \cdot \texttt{AVIN}_B(t) \end{aligned}$$

```
bool LazyCodeMotion::Avin(CfgNode *B, Instr *T) {
  if (preds_size(B) == 0)
    return false;   // it's the entry block
  for (int i = 0; i < m; ++i) {
    if(! Avout(get_pred(B, i), T))
      return false;
  } // end for
  return true;
}
```

```
bool LazyCodeMotion::Avout(CfgNode *B, Instr *T) {
  // the Avout list of Basic Block
  static set<CfgNode *> AvoutSet;
  ....
  // find B in Avout
  return (AvoutSet.find(B) != AvoutSet.end());
}
```

Global Anticipability A term t may be anticipated on exit of a block B if it can be anticipated on entry of each successor of the block:

$$\begin{aligned} \texttt{ANTOUT}_B(t) &= \begin{cases} \texttt{false} & \text{if } B \text{ is an exit block} \\ \prod_{S_i \in Succ(B)} \texttt{ANTIN}_{S_i}(t) & \text{otherwise} \end{cases} \\ \texttt{ANTIN}_B(t) &= \texttt{ANTLOC}_B(t) + \texttt{TRANSP}_B(t) \cdot \texttt{ANTOUT}_B(t) \end{aligned}$$

```
bool LazyCodeMotion::Antout(CfgNode *B, Instr *T) {
  if (succs_size(B) == 0)
    return false;    // it's one exit block
  for (int i = 0; i < m; ++i) {
    if(! Antin(get_succ(B, i), T))
      return false;
  } // end for
  return true;
```

```
bool LazyCodeMotion::Antin(CfgNode *B, Instr *T) {
  // the Antin list of Basic Block
  static set<CfgNode *> AntinSet;
```

```
  ....
  // find B in Antin
  return (AntinSet.find(B) != AntinSet.end());
}
```

3.2 Computing the Earliest Points of Insertion

Lazy Code Motion need first to determine the earliest points in the procedure at which evaluations of a term t can be inserted to make all current evaluations of t redundant. These point must satisfy the conditions of safety and profitability.

According to the algorithm's version of Drechsler and Stadel, insertions are considered on the edges of the procedure flow graph. This simply means that an evaluation of t is made if that edge is traversed. Edges are noted (P, B), where P is one of the predecessor of B.

Consider an arbitrary edge (P, B) and a term t. Under what conditions would it be the earliest point at which to insert an evaluation of t ?

- t should be anticipated at the beginning of S.
- t should not be available at the end of P. If it is available at the end of P, then there is no profitable point in inserting a new evaluation, since it would only create two consecutive evaluations of t.
- There should be one of two reasons that the evaluation cannot be placed earlier:
  - Either t is not transparent in the preceding block P or t is not anticipated at the end of P. An earliest insertion would not be safe.
  - There is a path out of P that does not contain an evaluation of t. An earliest insertion would not be profitable.

We can directly translate these conditions into equations:

$$\texttt{EARLIEST}_{(P,B)}(t) = \begin{cases} \texttt{ANTIN}_B(t) \cdot \overline{\texttt{AVOUT}_P(t)} & \text{if } P \text{ is the entry} \\ \texttt{ANTIN}_B(t) \cdot \overline{\texttt{AVOUT}_P(t)} \cdot (\overline{\texttt{TRANSP}_P(t)} + \overline{\texttt{ANTOUT}_P(t)}) & \text{otherwise} \end{cases}$$

```
inline bool LazyCodeMotion::Earliest(CfgNode *P, CfgNode *S, Instr
*T) {
  if ((! Avout(P, T)) && (Antin(S, T)))
    return (get_number(P) == 0) || (! Transp(P, T)) || (! Antout(P, T));
  else
    return false;
```

3.3 Computing the Latest Point of Insertion

Inserting evaluations of each term t on the edges described by `EARLIEST` makes all evaluations of t at the beginning of blocks redundant. The intuition behind this transformation is to move up evaluations as far as possible while maintaining safety. In fact `EARLIEST` gives an optimal solution in term of number of evaluations performed. However life-times of temporaries are not taken into account and the transformation is far from optimal when taking into consideration the length of time that values might stay in registers.

The idea to solve this problem is to delay as far as possible the insertion of an evaluation of t while maintaining computational optimality. In other words, we need to find the latest efficient points.

Consider a block P, a term t and all paths from the entry block E to P (noted $E \rightsquigarrow P$). Under what conditions would be an insertion of an evaluation of t delayed ? If each of the paths $E \rightsquigarrow P$ contains an edge where `EARLIEST` is true and there are no following instructions that evaluate t or destroy the transparency of t (kill t), then the insertion can be delayed until after P. The delayed insertion can occur just before a block B that either contains an evaluation of t or has an entering edge coming from an original evaluation of t. In those cases, delay the insertion just before block B. I hope that this intuition is clear because I always have trouble meeting myself there, anyway see [2, 4] for more details. Here are the equation to summarize these ideas:

$$\begin{aligned} \texttt{LATERIN}_B(t) &= \begin{cases} \texttt{false} & \text{if } P \text{ is the entry} \\ \prod_{P_i \in Pred(B)} \texttt{LATER}_{(P,B)}(t) & \text{otherwise} \end{cases} \\ \texttt{LATER}_{(P,B)}(t) &= (\texttt{LATERIN}_P(t) \cdot \overline{\texttt{ANTLOC}_P(t)}) + \texttt{EARLIEST}_{(P,B)}(t) \end{aligned}$$

A solution to this equations system can be found with an iterative algorithm that use work list. For more practical informations see [6]:

```
inline bool LazyCodeMotion::Laterin(CfgNode *BB, Instr *T) {
  // the LATERIN list of Basic Block
  static set<CfgNode *> LaterinSet;
  ....
  // find BB in Laterin
  return (LaterinSet.find(BB) != LaterinSet.end());
}
```

An intuition behind `LATERIN` is that an evaluation of a term t can be be moved through a block B without losing any benefit if $\texttt{LATERIN}_B(t)$ is true.

```
inline bool LazyCodeMotion::Later(CfgNode *P, CfgNode *B, Instr
*T) {
  if (Earliest(P, B, T))
    return true;
  else
    return (Laterin(P, T)) && (! Antloc(P, T));
}
```

3.4 Optimal Insertions and Deletions

Now, we need to compute two last things for a term t, first the final points of insertion and then the points where evaluations are deleted. We delay the insertion of evaluations of t to the the last possible point. That would be a point where an edge (P, B) satisfies the conditions for delay ($\texttt{LATER}_{(P,B)}(t)$) but the block at the head of the edge P does not ($\texttt{LATERIN}_B(t)$) because one of the other edges does not have a clear backward path to occurrences of `EARLIEST`.

$$\begin{aligned} \texttt{INSERT}_{(P,B)}(t) &= \texttt{LATER}_{(P,B)}(t) \cdot \overline{\texttt{LATERIN}_B(t)} \\ \texttt{DELETE}_{(B)}(t) &= \begin{cases} \texttt{false} & \text{if } P \text{ is the entry} \\ \texttt{ANTLOC}_B(t) \cdot \overline{\texttt{LATERIN}_B(t)} & \text{otherwise} \end{cases} \end{aligned}$$

The implementation is straightforward:

```
// the Insert predicate
inline bool LazyCodeMotion::Insert(CfgNode *P, CfgNode *B, Instr
*T) {
  return Later(P, B, T) && (! Laterin(B, T));
}
```

```
// the Delete predicate
inline bool LazyCodeMotion::Delete(CfgNode *B, Instr *T) {
  return (! B->get_number() == 0) && Antloc(B, T) && (! Laterin(B, T));
}
```

We can now implement a function to perform insertion end deletion of evaluation of a term t. The pass does not need to look at all blocks to see if `INSERT` is true, since t is anticipated at a block P at the head of an edge (P, B) where the insertion will occur:

```
// computing insertion and deletion points
bool LazyCodeMotion::Lazy_Update(Instr *T) {
  bool updated = false;
  // create a register to store the evaluation of T
  Opnd evalT = get_dst(new_instr_eval(T));
  // iterate over each BB
  for (int i = 0, nb_BB = size(unit_cfg); i < nb_BB; ++i) {
    CfgNode *B = get_node(unit_cfg, i);
    // T can be anticipated into B
    if (Antin(B, T)) {
      // eval of T needn't to be delayed
      if (! Laterin(B, T)) {
        // iterate overs each predecessors
        for (int j = 0, nb_preds = preds_size(B); j < nb_preds; ++j) {
          CfgNode *P = get_pred(B, j);
          // the later point of insertion
          if (Later(P, B, T)) {
            updated = true;
            // perform insertion
            if (succs_size(P) == 1) {
              append(P, new_instr_eval(T));
            } else {
              CfgNode *new_node = insert_empty_node(unit_cfg, P, B);
              append(new_node, new_instr_eval(T));
            } // end if
          } // end if
        } // end for
        // delete redundant evaluation of T
        if (Antloc(B, T)) {
          updated = true;
          // get each instruction of block B, find T, and replace evaluation
          for (InstrHandle hI = start(B); hI != end(B); ++hI) {
            Instr *instr = *hI;
            if (SameTerm(instr, T))
              replace(B, hI, new_instr_alm(get_dst(instr) ,suifvm::MOV, clone(evalT)));
          } // end for
        } // end if
      } // end if
    } // end if

    // seach evaluation of T and replace the dst with evalT
    ....
  } // end for
  return updated;
}
```

At last we implement the main function that perform the Lazy Code Motion transformation on a procedure in CFG form. Note the call to the function `local_CSE` that perform a

simple local Common Sub-Expression elimination inside a block. It is done to simplify the work of LCM. The implementation of this local CSE won't be explain but you can see [1] for many details. Here is the full implementation of the main function that perform LCM:

```
bool LazyCodeMotion::Lazy_Code_Motion() {
  bool updated = false;
  // iterate over each instruction
  for (CfgNodeHandle hBB = start(unit_cfg); hBB != end(unit_cfg); ++hBB) {
    CfgNode *BB = *hBB;
    // perform local CSE in each BB
    local_CSE(BB);
    // begin the lazy code motion process (LCM)
    // we have to take each Term in the program and apply LCM
    for (InstrHandle hI = start(BB); hI != end(BB); ++hI) {
      Instr *instr = *hI;
      // optimize only arithmetical Instructions and Loads
      if (! is_binary_exp(instr) && ! is_ldc(instr) &&
      ! reads_memory(instr)) continue;
      if (IsUpdatedTerm(instr)) continue;
      // call Lazy code motion algorithm
      updated |= Lazy_Update(instr);
    } // end for
  } // end for
  return updated;
}
```

One of the problem of the code transformed by LCM is that the function `Lazy_Update` may insert many useless copy operations. It would be a good idea to perform a simple local copy propagation after a full transformation of LCM (after a call to `Lazy_Code_Motion()`):

```
void LazyCodeMotion::do_opt_unit(OptUnit *unit) {
  ....
  bool updated = false;
  do {
    updated = Lazy_Code_Motion();
    int iter = 0;
    // perform a local copy propagation
    for (CfgNodeHandle hBB = start(unit_cfg); hBB != end(unit_cfg); ++hBB) {
      CfgNode *BB = *hBB;
      iter = 0;
      while (local_copy_prop(BB) && iter < MAX_COPY_PROP_ITER)
        iter++;
    } while (updated);
    ....
```

The local copy propagation algorithm won't be explain but you can see [1] for more details. Like the constant propagation, the copy propagation optimization let dead code in the code. Thus it is a good idea to perform a dead code elimination after LCM.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of knoop, ru¨thing, and steffen's lazy code motion. *ACM SIGPLAN Notices*, 28(5):29–38, 1993.

[3] J. Knoop, O. Ruething, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 224–234, San Francisco, CA, June 1992.

[4] 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.

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

[6] Robert Morgan. *Building an optimizing compiler*. Digital Press, 1998.