Advanced Compiler, Fall 2024, HW4: Lazy Code Motion: An LLVM pass implementation

B09401064 Yu-Tong Cheng

This is an implementation of lazy code motion from what we've learned in the advanced compiler class. What's different is that this implementaion deals with the LLVM IR, which is already in SSA form, so the scenario is simpler.

The original goal of this homework project was to complete the LCM pass. However, due to some struggles in implementation, this pass cannot be implemented correctly in current version, with unexpected insertions and deletions destroying the input code structure. This report represents my best effort to explain the ideas and implementation on the topic.

I. Algorithm Design

The problem formulation of this report is based on the one taught in class. I've searched it up and find other methods of partial redundancy elimination, such as [2]. Here we follow the Chapter 10.3 of textbook Engineering a Compiler, 3rd ed [3].

A. Explanation of the optimization strategy

Lazy code motion is a way of eliminating **partial redundancy**, since some of them may not be eliminated by common subexpression elimination or loop invariant code motion methods. It aims to reduce the lifetime of registers by postponing instructions as late as possible, being a good practice under the pressure of register number limitation. [4]

B. Theoretical analysis of potential improvements

Theoretically, LCM can handle both fully and partial redundancy, inherently solving loop invariant code motion, common subexpression elimination and more cases.

C. Implementation considerations and challenges

The entire problem can be divided into three parts:

- Collecting information of each node and edge independently (`ExprKill`, `DEExpr`, `UEExpr`, `Earliest`, `Insert`, `Delete`);
- Solving three data flow analysis problem:
  - Availibility (forward flow);
  - Anticipability (backward flow); and
  - `Later`/`LaterIn` (forward flow).
- Code motion: Insertion and deletion.

The first two parts are successfully handled in most cases by the current implementation besides the `Delete` information, where the bug has not be fully identified. The last part was where I encountered most challenges on the postprocessing after code motion (replacing uses, etc). We will leave the discussion of current implementation in the Implementation Details section.

Here, a work: Partial redundancy elimination with lazy code motion [2] is the main reference to this homework project for technical details, which will be referred to "PRE project". Other sources include ChatGPT (chat link attached in code document), llvm website documentation and other similar works.

D. Code examples illustrating the transformation

Here we provide an example (in `tests/simple/if.c`):

```c
#include <stdio.h>
int main()
{
  int r1 = 1, r2 = 2, r3;

  if (r1 < 5) {
    r3 = r1 + r2;
    printf("%d\n", r3);
  }
  else {
    printf("%d\n", r1);
  }

  return 0;
}
```

When converted into LLVM IR, it becomes:

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 1, ptr %2, align 4
  store i32 2, ptr %3, align 4
  %5 = load i32, ptr %2, align 4
  %6 = icmp slt i32 %5, 5
  br i1 %6, label %7, label %15

7:                        ; preds = %0
  %8 = load i32, ptr %2, align 4
  %9 = load i32, ptr %3, align 4
  %10 = add nsw i32 %13, %14
  store i32 %10, ptr %4, align 4
  %11 = load i32, ptr %4, align 4
  %12 = call i32 (ptr, ...)
   ↪ @printf(ptr noundef @.str, i32
   ↪ noundef %11)
  %13 = load i32, ptr %2, align 4
  %14 = load i32, ptr %3, align 4
  br label %19

15:                    ; preds = %0
  %16 = load i32, ptr %2, align 4
  %17 = call i32 (ptr, ...)
   ↪  @printf(ptr noundef @.str, i32
   ↪  noundef %18)
  %18 = load i32, ptr %2, align 4
  br label %19

19:                    ; preds = %15, %7
  ret i32 0
}
```

`r3 = r1 + r2` is a loop invariant with corresponding IR:

```
  %8 = load i32, ptr %2, align 4
  %9 = load i32, ptr %3, align 4
  %10 = add nsw i32 %13, %14
```

, and it should be moved out of the loop. That is, insertions should happen at the end of block `0` and deletion at block `7` to move this paragraph of codes, such that it becomes an IR semantically equivalent to:

```
#include <stdio.h>
int main()
{
  int r1 = 1, r2 = 2, r3;
  r3 = r1 + r2; // moved outof the
   ↪  loop

  if (r1 < 5) {
    printf("%d\n", r3);
  }
  else {
    printf("%d\n", r1);
  }

  return 0;
}
```

II. Implementation Details

A. LLVM pass implementation methodology

This work is created based on the **llvm-pass-backbone** [1] project. It contains both backbones of static registration and dynamic plugin form of the pass, and provides basic environmental construction to build a LLVM pass.

B. Key data structures and algorithms used

1) Data structure: Here we use the STL from both std and llvm namespace.

- std: map, queue
- llvm: SmallVector (replace std::vector), BitVector
- Control flow graph:
  Informations of each node (`BasicBlock`) and edge (`std::pair<BasicBlock*, BasicBlock*>`) are build upon the CFG generated by LLVM and stored in structures `BasicBlockInfo` and `EdgeInfo` respectively.
- Expression:
  `llvm:Instruction` with binary operators or store/load operations are transformed into structure `Expression`. For `StoreInst`, the ”destination” is operand 1; for other types of instructions, it follows the original convention of the LLVM IR.
  The conversion is donne in `Expression InstrToExpr(Instruction*)` method.
- Set operation:
  This implementation uses `llvm::BitVector` to represent sets of expression. The size of bitvectors is the number of all globally collected expressions. By representing sets as bits, set operations can be easily and rapidly done using logical NOT(`flip`), AND(&) and OR(|).

2) Algorithm:

- Downward/Upward exposed expression:
  The two problems are solved by a simple pass over the function. The pseudocode is as below:

```
defined = {}
B: basic block

for I in order(B.instructions):
    expr = instrToExpr(I)
    for op in I.operands:
        if op in defined:
            B.DEExpr.add(expr)
    defined.add(I.dest)
```

with `order(B.instructions)` being as displayed in **Table I**.

TABLE I
Instruction scanning order in DEExpr/UEExpr.

Exposed	`order`
Downward	End to start (`B.rbegin()` to `B.rend()`)
Upward	Start to end (`B.begin()` to `B.end()`)

- Worklist algorithm
  The worklist algorithm is implemented with some degree reference to the PRE project (but still different). Below is the pseudocode of the whole process:

```
Q: queue // std::queue<BasicBlock*>
frontiers: currently pending in queue
startNodes: a set of nodes

Q.push(startNodes)
frontiers.insert(startNodes)

while(!Q.empty())
  P = Q.pop()
  frontiers.remove(P)

  P_changed = false
```

```
update(P, P.attr1) //(1)
P_changed |= (P.attr1 changed)

for R in getForwardNodes(P):
  update(P.attr1, R.attr2) //(2)
  if ((P_changed || R.attr2 changed)
 ↪  && R not in frontiers)
    Q.push(R)
    frontiers.insert(R)
```

The `startNodes` and `getForwardNodes` depend on the direction of data flow, as listed in **Table II**.

TABLE II
DATA FLOW IN WORKLIST ALGORITHM

Data flow	`startNodes`	`getForwardNodes(P)`
Forward	entry block	successors(P)
Backward	exit blocks	predecessors(P)

In this way, it is guaranteed that

- Each node P will be visited before each R in ”getForwardNodes(P)”.
- Each edge will be visited in the data flow direction.
- Each update of a node will be pushed down to all ”forward nodes” of P, and each block reachable from ”startNodes”.

For (1), (2) listed in the pseudocode:

- Availability: Forward problem.
  * Initialization:

$$AvailIn(n) = \begin{cases} \{\}, n = n_0 \\ \{all\}, n \neq n0 \end{cases}$$

$$AvailOut(n) = \{all\}$$

  * (1):

$$AvailOut(n) = DEExpr(n)$$

$$\cup (AvailIn(n) - ExprKill(P))$$

  * (2):

$$AvailIn(n) = \bigcap_{m \in pred(n)} AvailOut(m)$$

R ∈ successors(P) $\Longrightarrow$ ”R.AvailIn &= P.AvailOut”

- Anticipability: Backward problem
  * Initialization:

$$AntOut(n) = \begin{cases} \{\}, n \in n_f \\ \{all\}, n \notin n_f \end{cases}$$

$$AntIn(n) = \{all\}$$

  * (1):

$$AntIn(n) = UEExpr(n)$$

$$\cup (AntOut(n) - ExprKill(n))$$

  * (2):

$$AntOut(n) = \bigcap_{m \in succ(n)} AntIn(m)$$

R ∈ predecessors(P) $\Longrightarrow$ ”R.AntOut &= P.AntIn”

- Later: Forward problem.
  * Initialization:

$$LaterIn(n) = \begin{cases} \{\}, n = n_0 \\ \{all\}, n \neq n_0 \end{cases}$$

$$Later(n) = \{all\}$$

  * (1):

$$Later(i, j) = Earliest(i, j) \cup (LaterIn(i) - UEExpr(i))$$

  * (2):

$$LaterIn(j) = \bigcap_{i \in pred(j)} (Later(i, j)), j \neq n_0$$

R ∈ successors(P) $\Longrightarrow$ ”R.LaterIn &= (P, R).Later”

3) Difference with the referenced PRE project: The PRE project follows another problem formulation in terms of partial redundancy elimination. It first splits all edges where the two endpoint basic blocks have multiple successors and predecessors by inserting new blocks, and at the same time splits all basic blocks into single-instruction blocks. For code motion (transforming CFG), it first inserts the needed instructions at the beginning of each block, updates phi instructions, and then replaces the partial redundancy with new variables.

C. Integration with the LLVM optimization pipeline

The `LCMPass` is integrated with other optimization pipeline by adding a compiling flag as:

```
clang -fpass-plugin=<path/to/.so/file> \
something.c
```

III. EXPERIMENTAL EVALUATION

A. Description of test cases and benchmarks

1) Simple testcase: Basic testcases are generated and displayed in `tests/simple`. They are two simple cases of loop invariant code motion.

2) Complex scenarios: Here we use benchmarks from `llvm-test-suite/SingleSource/Benchmarks` [5]. The selected benchmark, with reference to the PRE project [2], are:

BenchmarkGame, CoyoteBench, Dhrystone, Linpack,
McGill, Misc, PolyBench, Shootout, Stanford.

`lit` from LLVM project is used for testing these benchmarks, following the instructions from [5].

B. Performance measurements and analysis

1) Simple cases: Since the LCM pass cannot be executed currently due to the segmentation fault that yet to be debugged, the performance on simple cases cannot be tested.

2) Complex scenarios: Even though the LCM pass is wrong, somehow the test suite workflow still works. Here the numeric data are displayed in **Table III**, but correctness is not guaranteed. The conditions are with reference to [2] to examine the difference on performance between:

- Baseline: `mem2reg` only, which is transforming IR into SSA form;
- Basic: `mem2reg, gvn, simplifycfg`, which are all LLVM native optimizing passes;
- LCM: `mem2reg, LCMPass` only, to test the efficacy of lazy code motion;
- Basic+LCM: `mem2reg, gvn, simplifycfg, LCMPass`, to examine the additional efficacy of LCMPass and if its effect is just overlapped with the existing passes, or is there more it can do;
- O3: use `-O3` flag as the maximal optimized performance for comparison.

TABLE III
Results on the LLVM SingleSource benchmark.

Passes used	Baseline	Basic	LCM	Basic+LCM	O3
mean	12.9263	13.0220	12.67394	12.85376	12.69304
std.	35.7272	35.8817	33.32701	35.42112	34.4389
min.	0.0000	0.0010	0.0005	0.0000	0.0000
25%	0.1657	0.1615	0.1422	0.1442	0.1466
50%	3.3856	3.4816	2.8977	3.0694	3.1421
75%	9.6741	9.6663	10.1342	9.3212	9.652
max	318.6317	317.0101	285.2476	317.3949	306.3979
Passed cases	105/105 (100.00%)	105/105 (100.00%)	105/105 (100.00%)	105/105 (100.00%)	105/105 (100.00%)

IV. Correctness Proof

A. Formal or informal proof of correctness

Lazy code motion is first proposed by J. Knoop et. al at 1992 [6]. The formal verification can be seen in [7].

The proof of this implementation itself can be inducted in an informal way following the definitions and the class slides.

B. Testing methodology and results

1) Simple examples: Using `tests/simple.sh`, we can find out that the LCM pass does not pass any of the simple cases due to segmentation fault of LLVM front end, yielding not executable files.

2) Benchmarks: The correctness is also checked on the benchmark in **Table III**. Over all 105 tests, all experiments pass all of them. It is sure that the LCM pass is executed in the experiments that should contain it (LCM, Basic+LCM); why they still pass the benchmark is yet to be investigated.

V. Discussion

A. Why does the LCM pass fail?

The concrete reason is yet to be investigated; neither is the result semantically correct in the level of information collection (unexpected deletions and insertions happen), nor is the code motion implemented correctly - I think there should be some use replacements that is not handled correctly, or that can be the direct result of incorrect insertion and deletion, leading to multiple ”<badref>” in the resulting LLVM IR (generated ”.ll” files using `-emit-llvm -S` when compiled with clang). I hope that the issue can be solved in the near future, since all the testing scripts have been written, only to wait for the completion of the pass. The entire framework of lazy code motion is quite clear, so even though I've been struggling for a few weeks, I believe that the debugging can be done in no time with some key advice.

For code writing history of this homework, please follow the link: https://github.com/christmaskid/AC_HW4.

References

[1] https://github.com/sampsyo/llvm-pass-skeleton.
[2] https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/pre/. Implementation can be found at: https://github.com/Neroysq/llvm-pre
[3] Keith D. Cooper and Linda Torczon. Engineering a Compiler, Edition 3. 02 Oct 2022.
[4] Ryan Doenges. Lazy Code Motion. The CS 6120 Course Blog, October 26, 2019. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/lazy-code-motion/.
[5] LLVM test suite. https://llvm.org/docs/TestSuiteGuide.html. SingleSource benchmarks can be found at: (https://github.com/llvm/llvm-test-suite/tree/main/SingleSource/Benchmarks).
[6] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992. Lazy code motion. SIGPLAN Not. 27, 7 (July 1992), 224–234. https://doi.org/10.1145/143103.143136.
[7] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified validation of lazy code motion. SIGPLAN Not. 44, 6 (June 2009), 316–326.https://doi.org/10.1145/1543135.1542512.