CS426 Unit Project

Chengsong Zhang (cz81) & Jiachen Yuan (jyuan19)

Algorithms

UnitLoopInfo

The loop identification pass starts with a post-order traversal on the DominatorTree to identify all the loop header nodes. A node (basic block) is a loop header if there are some back edges pointing to it, meaning the node dominates its own predecessor(s).

For each back edge that is associated with a node, we identify the natural loop (member blocks) by picking out nodes that are at the same time dominated by the loop header and reacheable to the source basic block of the back edge. The former sets of nodes are easily marked by DominatorTree's API `DT.getDescendants()`, and the latter is marked using a bottom-up BFS traversal starting from the source node of the back edge. This per-back-edge natural loop members information in LoopMeta struct that is attached to each loop headers in the CFG.

The loop identification pass also tease out the nested relationships between natural loops. For each loop member of an identified loop in the bottom-up order, the algorithm tries to find members that are also loop headers but whose loops have not been given a parent. The algorithm checks whether all the members of the sub loop are in the member set of the loop, and it yes, it recognize the sub-loop as an inner sub-loop of the identified loop and set the parent pointer as well. We set up the nested loop relationships immediately after the agnostic loop identification in the last paragraph because the post-order traversal order and the parent pointer check help in make sure the nested relationships are built from innermost to outermost efficiently.

Besides identifying the loop headers and nested relationships of the natural loops, there are other housekeeping data structures that are added after design of the LICM pass. For example, the algorithm also specifically marks all the top-level (outer-most) natural loops. It also identifies all the exit blocks of each loop.

UnitLICM

LICM tries to identify loop-invariant operations and move them before the loop body to reduce duplicate or unnecessary executions.

Our LICM starts with giving each natural loop a preheader node. If the loop header only has one possible entry point, then the preheader is just the entry block. If the loop header has more than one entry points, then a preheader basic block is inserted before the loop header. All the branch instructions in the original predecessors of the loop header are modified so that they branch to the preheader instead. Also, phi usage in the loop header need to be changed to refer to the preheader nodes.

Our LICM iterates through natural loops from innermost to outermost. And for every instruction in a loop, it check to see whether it can be hoisted. It has to have no alias (if ld/st instructions), dominates all exit blocks of the loop (or no but safe to speculative execute), and all uses in the instruction should be loop-invariant, either being constants, defined outside loop, or depending on something else in the loop that is already marked loop invariant.

After all the instructions in a loop has been visited, the hoistable instructions are inserted at the preheader and statistics are collected. Notice since our algorithm moves from innermost loop to outermost loop, it is possible for some instructions to be "bubbled up" along all the loop preheaders. We double counted the statistics number for these hoist because it would be messy to distinguish first-time hoist.

UnitSCCP

SCCP starts from entry block as the first flow source block and initialize all function arguments as bottom (cannot be constant).

SCCP visits each flow source blocks by visiting all instructions until SCCP cannot find any new flow source blocks. Then SCCP visits each SSA source instructions until SCCP cannot find any new SSA source instructions. SCCP store all constant evaluations for each instructions.

Then SCCP replace each constant instructions with the evaluated constant and count the number of unreachable blocks.

SCCP has a command method for visiting each instruction:

- When meeting branch instruction, SCCP evaluates the condition and add the corresponding branch target block as a flow source block. SCCP only add the branch target block when it is absolutely necessary to visit that block in the future:
  - either the condition is not constant or the condition evaluates to that branch
  - the branch target block is not visited yet
- When meeting unary instruction, SCCP evaluates the operand and apply opcode to the operand if the operand is constant.
- When meeting binary instruction, SCCP evaluates both operands and apply opcode to the operands if both operands are constant.
- When meeting compare instruction, SCCP evaluates both operands and apply comparison predicate to the operands if both operands are constant.
- When meeting phi instruction, SCCP evaluates all incoming values and apply meet, i.e. evaluate according to bottom <= constant <= top.
- When meeting select instruction, SCCP evaluates the condition and apply meet to the operands if the condition is not constant, otherwise evaluate the selected operand according to the condition.
- When meeting getelementptr instruction, SCCP evaluates the base pointer and evaluate the offset according to the indices if the base point might be constant.

After evaluation, SCCP applies meet to the instruction and the evaluated lattice for the current instruction. We only need the SSA instruction as the SSA source instruction when

- the SSA instruction has not been evaluated to constant yet
- the SSA instruction has not been visited yet

Collaboration

Algirithm	Typer	Observer
UnitLoopInfo	Jiachen Yuan	Chengsong Zhang
UnitLICM	Jiachen Yuan	Chengsong Zhang
UnitSCCP	Chengsong Zhang	Jiachen Yuan

The above table shows how our work is divided in the high level, but we took a pair-programming approach and switch over relatively frequently for all passes. Both members helped with the debugging+testing process throughout the project.

Experiments

Use command `./scripts/run_all_tests.sh` to run all the benchmarks testcases. The output will be in `tests_out/`.

LICM

We show below the benchmark results for running both complicated sequences of passes and only UnitLICM pass. The "total" column shows the total number of hoists made by the UnitLICM pass, including all of store, load, computational, and a bunch of other kinds of instructions. Due to our algorithm (see UnitLICM section), we double-count a hoist along its way of being moved up in the hierarchy of nested loops.

It can be seen that UnitLICM can do more optimization when it is the only optimization pass being run, and computational hoists are usually the most common type of instructions being hoisted.

We mainly do tests on the given testcases. Other unit tests are written to `tests_private/`. Run `./scripts/run_test.sh [test_names] tests_private` to run tests. LICM relevant [test_names] are `test_single_loop_compute_hoist.c`, `test_double_loop_compute_hoist.c`, `test_load_hoist.c`, and `test_store_hoist.c`. The .ll files will be in `tests_private/`

Complicated Sequence of Passes (in spec)

filename	store	load	computational	total	store(licm only)	load(licm only)	computational(licm only)	total (licm only)
nsieve-bits.c.out	0	0	0	0	0	0	0	0
nesting.c.out	0	0	0	0	0	0	0	0
partialsums.c.out	0	0	1	1	0	9	10	21
almabench.c.out	0	2	5	9	0	2	5	75
PR491.c.out	0	0	0	0	0	0	0	0
spectral-norm.c.out	0	0	0	0	0	0	0	14
doloop.c.out	2	0	2	4	3	0	2	5
ffbench.c.out	0	4	26	34	0	4	19	25
n-body.c.out	0	0	0	30	0	0	0	48
matmul.c.out	0	0	3	7	0	0	5	18
random.c.out	0	0	0	0	0	0	0	0
one-iter.c.out	0	0	0	0	0	0	0	0
puzzle.c.out	0	0	0	0	0	0	0	0
recursive.c.out	0	0	0	0	0	0	0	0
fannkuch.c.out	0	1	1	4		0	1	1

Only UnitLICM

filename	store	load	computational	total
nsieve-bits.c.out	0	0	0	0
nesting.c.out	0	0	0	0
partialsums.c.out	0	9	10	21
almabench.c.out	0	2	5	75

filename	store	load	computational	total
PR491.c.out	0	0	0	0
spectral-norm.c.out	0	0	0	14
doloop.c.out	3	0	2	5
ffbench.c.out	0	4	19	25
n-body.c.out	0	0	0	48
matmul.c.out	0	0	5	18
random.c.out	0	0	0	0
one-iter.c.out	0	0	0	0
puzzle.c.out	0	0	0	0
recursive.c.out	0	0	0	0
fannkuch.c.out	0	1	1	8

I give a brief high-level example why certain store instruction can be hoisted. In `doloop.c` there is a `if (i > 0) *p = m;` statement. Notice that the value of `i` does not really change across different iterations because given initial statement `i=1`, the program always move value to `j` and later restore that value to `i`. Plus, the store does not depend on `m`, and thus the if statement, especially the store instruction inside it, is loop-invariant. Thus this is moved to the preheader basic block after our UnitLICM pass.

For compute instruction, I use the same program. There are LLVM IR instructions `%5 = sdiv i32 %0, -3` and `%6 = add nsw i32 %4, -2` after compiling to LLVM IR. They corresponds to `i = m / -3 - (1 + j);` in the source code. Since all uses in the instructions is loop-invariant or depend on loop-invariant instructions, they are lifted out of the natural loop.

For load instruction, I use the program `partialsums.c` and apply `mem2reg` and `unit-licm` passes as an example, there is a loop `for (kv=init; *(double *)(&kv)<=n; kv+=two) { poly += one /(kv*(kv+one)); Harmonic+= one / kv; zeta += one /(kv*kv); alt += av / kv; Gregory += av /(two*kv - one); }` In the LLVM IR of the loop, lots of uses are repetitive and not changed in the loop, UnitLICM put all loads in the preheader to make it cheaper to execute the loop.

In `nesting.c`, it is an example where LICM does not optimize. A close look at the program will find that the program's loop never changes the value of `i`. Therefore similar to the above example for store instruction, some of the store instructions can be theoretically moved out of the loop. However, the nested loop and conditionals complicates things, and it is better solved by applying SCCP instead.

SCCP

"/": left is the number running the complex command from the handout, right is running mem2reg + sccp only. In most cases, sccp can optimize more when we give less previous optimization passes.

I give a brief explanation why `test_conditional_branches.c.out` works but `test_constant_propagation.c.out` does not.

In `test_conditional_branches.c.out`, when we give handout instructions, the code will be optimized to be all printf statements before entering SCCP, so SCCP can optimize nothing. However, when we remove all but mem2reg, SCCP find `%1 = icmp eq i32 5, 5`, evaluate to `true` and replace all `$1` with `true`; SCCP also find `%.0 = phi i32 [ 10, %1 ], [ 20, %2 ]`, evaluate to `10` and replace all `%.0` with `10`.

In `test_constant_propagation.c.out`, whether I remove other optimization passes or not, SCCP only see printf statements so that it can do nothing.

Public testcases results

filename	removed	unreachable	replaced
ffbench.c.out	0/14	0/96	0/23
n-body.c.out	0/0	0/0	0/0
almabench.c.out	0/3	0/0	0/2
doloop.c.out	4/5	0/0	5/5
spectral-norm.c.out	0/0	0/0	0/0
fannkuch.c.out	0/0	0/0	0/0
one-iter.c.out	6/6	0/0	6/5
nesting.c.out	4/4	2/4	5/5
nsieve-bits.c.out	4/7	0/0	4/6
PR491.c.out	2/0	2/0	2/0
partialsums.c.out	0/7	0/7	0/18
random.c.out	4/4	0/0	5/6
recursive.c.out	0/0	0/0	0/0
matmul.c.out	0/0	0/0	0/0
puzzle.c.out	1/1	0/0	1/1

Private testcases results

filename	removed	unreachable	replaced
test_constant_folding.c.out	0/0	0/0	0/0
test_dead_code_elimination.c.out	0/0	0/0	0/0
test_redundant_computation.c.out	0/2	0/0	0/2
test_conditional_branches.c.out	0/2	0/1	0/2
test_constant_propagation.c.out	0/0	0/0	0/0

Conclusion

In this project, we have implemented loop identification, loop-invariant code motion and sparse conditional constant propagation. These optimizations do not work in all cases. Sometimes previous optimization passes might be so strong that make the current optimization pass having nothing to do. Nevertheless, we can still see imporvements made in some of the benchmark tests.

Reference

1. Prof. Gennady Pekhimenko. 2018. Compiler Optimization LICM: Loop Invariant Code Motion

2. Sasa Misailovic. 2023. CS426: Compiler Construction Lecture 20: Dataflow Analysis Finished, Review of Optimizations.
3. Sasa Misailovic. 2023. CS426: Compiler Construction Lecture 25: Back to Function-Level Program Optimization