CS426 MP5 Report
December 16, 2023
Jeffrey Hui (jhui8)
Flora Zheng (floraz2)

Loop Analysis: The pass identifies all of the natural loops in each function and basic blocks in each loop.
LICM: Loop-Invariant Code Motion (LICM) deletes code from the body of the loop. If there are no side effects, the LICM optimization pass moves code up to the preheader block or down to the exit basic blocks. The LLVM implementation of LICM moves aliased memory locations in the loop to live in registers. The conditions for the analysis (v = a op b) follow: Both a and b are constants. while (b) { v = 2 + 3; /* … */ }. Both a and b are defined before the loop (SSA ensures that there is a single dominating definition for each). X = …; while (b) { v = x + 1; /* … */ } Both a and b are referring to the variables that are in the loop but already determined to be loop invariant. X = …; while (b) { v = x + 1; t = v * 2; /* … */ } Option 1: Because the code is in SSA form, move to the block before the header. If there is no single block that dominates the loop header, create a loop preheader. One concern is if the loop-invariant computation is costly. Option 2: Perform option 1 along with the loop's condition.

```
if (cond) {
    t = a op b;
    while (cond) { /* … */ }
}
Example:
x = 1; y = 0;
while (y < 10) {
  t = min(x, 2)
  y = y + x
}
Becomes:
x = 1; y = 0;
t = min(x, 2)
while (y < 10) {
  y = y + x
}
```

```
Example:
i = 0;
while (i < 10) {
        v = a op b
        v is used here
        i = i + 1
}
Becomes:
v = a op b
i = 0;
while (i < 10) {
        v is used here
        i = i + 1
}
```

SCCP: Sparse Conditional Constant Propagation
The optimization substitutes SSA variables with constants and prunes unreachable conditional branches because of discovered constants. The transformation is safe due to propagating constant expressions and most languages permit the deletion of instructions. The result is fewer computations in most cases. The opportunity is symbolic constants and conditionally compiled code. The algorithm assumes statements are assignments or branch statements. Every non-Φ statement is in a separate basic block. The CFG is in SSA form where the flow edges are edges in the CFG graph and the SSA edges are edges from the definition to uses of the variable. Use the constant propagation lattice L $(S, \leq, \sqcap, \top, \bot)$. At the beginning, every definition has value $\top$ because the definition could be constant. At the beginning, the edges from the start node are reachable and the other CFG edges are unreachable. Use the FlowWL as the flow edges worklist and SSAWL as the SSA edges worklist. The algorithm visits a statement S if some incoming edge is executable. Disregard the Φ-argument if the incoming CFG edge is not executable. If the variable changes value, add SSA out-edges to SSAWL. If the CFG edge is executable, add it to FlowWL.
Example:

```
j = 1;
if (j > 0) {
        i = 1;
} else {
        i = 2;
}
k = i + 3;
```

Description of Code Status

- Loop Analysis Pass

This pass is fully functional and can identify all loops, loop preheader, loop start block, and loop end block.

- Loop Invariant Code Motion Pass

The pass is partially functional. It performs all the functions of a correct LICM except that it cannot move the innermost loop's invariant to the outermost loop's preheader in the case of nested loops. The code motion is only moving the current loop invariant into its own preheader (one level up). For the given tests, almabench and ffbench give a segmentation fault when running and partialsums becomes an infinite loop, the reason is unknown.

A possible solution to the code motion problem is to calculate the correct loop preheader in the Loop Analysis Pass such that all inner loop's preheader is the same with the outermost loop's preheader, thus when moving the instructions we can directly move them to the correct place. Another possible solution is to have an outer loop wrap around our algorithm that keeps doing LICM until all inner loops' invariants are moved to the outermost layer.

- Sparse Conditional Constant Propagation Pass

This pass is partially functional. It performs all the functions of a correct SCCP except that it cannot handle the bitcast instruction.

For bitcast operation, we also used the mutateType method but somehow causes invalid IR. The compiler is able to generate IR but when compiling to binary, some compare instructions are invalid such that icmp ne i32 %1, %2 becomes icmp ne i32 %1, i32 3 if %2 is proven to be constant 3 and were casted to i32.

Contributions

For the loop analysis pass and SCCP, Jeffrey did the typing and Flora was the observer. Flora initially implemented a partially functioning LICM that failed to compile, and later switched to Jeffrey's implementation that only has basic functionality.

Statistics

Our passes rely on mem2reg and the loop-rotate pass, and we run LICM and SCCP independent of each other. Therefore the statistics for LICM is obtained by running mem2reg, loop-rotate, unit-licm. Similarly the stats for SCCP is the result of running mem2reg, loop-rotate, unit-sccp. We rely on mem2reg because it eliminates most load and store instructions for SCCP, which we will not be handling. For loop-rotate pass, it ensures each loop has a preheader, a single backedge (which implies that there is a single latch), and all exit blocks are dominated by the loop header. In addition, it also converts for-loop to do-while loops and inserts a loop guard, which makes LICM easier to implement. Since loop-rotate ensures dedicated exit of the loop, we only need to check that an instruction dominates that one exit.

	Number of hoisted store instructions	Number of hoisted load instructions	Number of computational instructions that are hoisted
almabench.c	0	1	1
doloop.c	0	0	1
fannkuch.c	0	0	0
ffbench.c	0	0	12
matmul.c	0	0	5
n-body.c	0	0	0

nesting.c	0	0	0
nsieve-bits.c	0	0	0
one-iter.c	**0**	0	0
partialsums.c	0	9	12
PR491.c Has no loops	0	0	0
puzzle.c	0	0	0
random.c	0	0	0
recursive.c Has no loops	0	0	0
spectral-norm.c	0	0	7

	Number of instructions removed	Number of basic blocks that are made unreachable	Number of instructions replaced with (simpler instructions)
almabench.c	0	0	0
doloop.c	6	2	10
fannkuch.c main fannkuch	 0 0	 0 0	 0 0
ffbench.c main fourn	 1 1	 0 0	 1 1
matmul.c	0	0	0
n-body.c advance, energy, offset_momentum, main	0	0	0
nesting.c foo	6	1	11
nsieve-bits.c	1	0	1
one-iter.c foo	7	2	11

partialsums.c	0	0	0
PR491.c	0	0	0
puzzle.c	0	0	0
random.c	6	1	9
recursive.c	0	0	0
spectral-norm.c	0	0	0

Discussion of the experimental results

SCCP: The statistic of the number of basic blocks that are made unreachable measures the number of removed conditional branches. One example of unreachable is that in a branch instruction like br i1 %1, <true branch>, <false branch>, if the algorithm has proven that %1 is a constant, then we can discard either the true branch or the false branch depending on the value of %1. For the number of instructions removed, it simply means the number of constant definitions. Since they are constant, we can replace all their uses with that constant and delete the definition itself. Similarly, for the number of instructions replaced, it's the number of uses of a constant which we can replace. For example, we have instructions that assign %1 = 5 and immediately we have %2 = %1 + 3, then we can remove %1 = 5 and %2 can be replaced with 8. For the one-iter testcase, one example of an instruction removed is j = i + 1. One example of an instruction replaced with a simpler instruction is return i; is replaced with return 2;. One example of a basic block that is made unreachable is the condition (j > 0) is deleted. SCCP proves the condition if (j > 0) becomes if (2 > 0), deletes the condition if (j > 0), and propagates the constant so i = j becomes i = 2. For the one-iter testcase, one example of a missed opportunity for optimization with SCCP is not replacing br label %7 with br label %10 and deleting basic blocks 7, 8, 9. Deleting the basic blocks can be handled in other LLVM passes such as the simplify-cfg pass.

LICM: For the testcase doloop.c, a hoisted store instruction is *q = m; which corresponds to store i32 %0, ptr %2, align 4 and a second hoisted store instruction is i = m / -3 - (1 + j); which corresponds to store i32 %0, ptr %1, align 4 because n = %0, m = n = %0, and m is stored. A hoisted computational instruction is i = m / -3 - (1 + j); which corresponds to %4 = sdiv i32 %0, -3 and %5 = add nsw i32 %4, -2. An example of a hoisted load instruction is %50 = load i32, ptr %15, align 4 from matmul.c.

Missed Opportunity:

LICM: Right now our algorithm cannot move the innermost invariants to the outermost loop in case of nested loops. For tests/nestingloops.c, int c = a + b; is moved out of the for (int j = 0; j < 10; j++) loop and inside the for (int i = 0; i < 10; i++) loop. However, int c = a + b; can be moved outside both for loops.

SCCP: Our algorithm cannot handle cast instructions such as ZEXT or SEXT, which occurred frequently as a pattern in the provided test cases. If a potential constant instruction is dominated by those cast instructions, then we wouldn't be able to propagate constants further. For example, matmul.ll has %17 = zext i32 %16 to i64.

Conclusion

Our report details the outcomes of code analysis and optimization passes, specifically Loop Analysis, Loop-Invariant Code Motion (LICM), and Sparse Conditional Constant Propagation (SCCP). The Loop

Analysis Pass successfully identifies loops, preheaders, and blocks. However, the LICM pass is only partially functional, encountering issues with nested loops. Proposed solutions involve refining loop preheader calculations or implementing a continuous LICM process. The SCCP pass is also partially functional, substituting SSA variables with constants but facing runtime errors in specific cases, especially with cast instructions. Our experiment suggests exploring alternative representations for constant values to address these challenges. The statistics rely on mem2reg and loop-rotate passes for LICM and SCCP, showcasing partial success in handling the specified optimizations. The partial success in statistics highlights the need for further refinement and exploration in optimizing code transformations. This experiment emphasizes the intricate nature of compiler optimizations and the ongoing development of more robust optimization techniques and tools for enhanced code efficiency and performance in various programming contexts.

References

Jing, Y. (2019, December 30). Loop invariant code motion and loop reduction for Bril. Loop Invariant Code Motion and Loop Reduction for Bril. *https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/loop-reduction/*

LLVM loop terminology (and canonical forms). LLVM Loop Terminology (and Canonical Forms) - LLVM 18.0.0git documentation. (n.d.). *https://llvm.org/docs/LoopTerminology.html#loop-terminology-loop-rotate*

LLVM's analysis and transform passes. LLVM's Analysis and Transform Passes - LLVM 18.0.0git documentation. (n.d.). *https://llvm.org/docs/Passes.html#loop-rotate-rotate-loops*

Anastos, M. (2019, October 23). Sparse conditional constant propagation. Sparse Conditional Constant Propagation. *https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/sccp/*

Wegman, M. N., & Zadeck, F. K. (1991). Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems, 13(2), 181–210. *https://doi.org/10.1145/103135.103136*