Advanced Compiler Report

1st Hung Tso Shiao
dept. CSIE
National Taiwan University
Taipei, Taiwan
r13922163@ntu.edu.tw

***Abstract*—This report presents the design and implementation of an LLVM pass for optimization, including theoretical analysis, experimental results, and correctness proofs.**

***Index Terms*—LLVM, Optimization, Algorithms, Experimental Evaluation, Correctness.**

I. Algorithm Design

A. Optimization Strategy

Loop-Invariant Code Motion (LICM) is an optimization that identifies computations within a loop that yield the same result across all iterations, then hoists these computations to a block that dominates the loop (often called the preheader). By removing redundant instructions from the loop body, LICM can reduce the per-iteration cost of the loop. Our pass implements a naïve or basic LICM strategy using the new LLVM pass manager. In essence, it operates as follows:

1) **Identify a Suitable Preheader:** We begin by checking whether the loop has a single, well-defined preheader basic block. Without this block, we cannot safely place hoisted instructions.
2) **Iterative Invariant Discovery:** We maintain a set I of loop-invariant instructions. Initially, I is empty. We scan the instructions in all basic blocks of the loop (except the preheader), looking for instructions that:
   - Have no side effects (e.g., no writes to memory, no volatile operations).
   - Are safe to speculate, indicated by LLVM's internal checks.
   - Have all operands already in I or defined outside the loop.

   If an instruction meets these criteria, we insert it into I. This process repeats until no new loop-invariant instructions are found (i.e., it "converges").
3) **Hoisting:** After convergence, all instructions in I are moved into the preheader. This ensures each instruction is computed once, rather than once per iteration.

By default, our pass is conservative—it avoids hoisting any instructions that may read or write memory, unless it can prove safety (e.g., no aliasing). This simple design ensures correctness at the cost of missing potential hoisting opportunities.

B. Theoretical Analysis

Consider a loop of n iterations, where an instruction I is computed every iteration. If I depends only on values that are constant over all iterations, then I itself is constant with respect to the loop iteration variable.

a) Loop Body Cost:: If I remains in the loop, it is executed n times.

b) After LICM:: By hoisting I into the preheader, it executes only once, reducing the loop body's operation count by up to $n-1$ occurrences of I.

Hence, for each hoisted instruction, the per-iteration cost decreases, which can lead to a measurable overall speedup when n is large. While a single instruction's overhead may be small on modern hardware, multiple hoistable instructions or repeated loops can yield more substantial gains.

c) Ideal Scenario:: In the ideal case where many instructions inside the loop are invariant, we reduce the loop-body time from T to $T-k$, where k is the cumulative cost of hoisted operations. The resulting speedup can be approximated by:

$$\text{Speedup} \approx \frac{n \times T}{n \times (T-k)} = \frac{T}{T-k}.$$

For small k relative to T, the gain may be modest. However, hoisting more expensive operations or many instructions can produce significant performance improvements.

C. Implementation Considerations

a) Preheader Detection:: Hoisting requires a single entry point to the loop. If a loop does not have a unique preheader (e.g., multiple predecessors), the pass skips transformation to avoid introducing incorrect control flow.

b) Safety and Side Effects:: We rely on LLVM's built-in function `isSafeToSpeculativelyExecute` to ensure instructions will not fault or produce different results when moved. We also skip any instructions that *may* read or write memory, including loads, stores, or calls with potential side effects.

c) Iterative Discovery:: Instructions become "loop-invariant" only if all of their inputs are known loop-invariant or defined outside the loop. Discovering them in a single pass is insufficient because hoisting one instruction can enable another to become invariant. We solve this by iterating until no new invariants appear (a *fixed-point approach*).

d) No Alias Analysis:: Our prototype omits alias analysis, so we do not hoist *any* memory instructions. Real-world LICM uses advanced memory-dependence checks to safely move loads and certain calls.

e) Phi Nodes and Terminators:: We do not attempt to move `PHI` nodes or terminator instructions (like `br`, `ret`). These often have complex dependencies or control-flow semantics that must remain within the loop.

D. Code Example

The following example demonstrates the effect of Loop Invariant Code Motion (LICM) on a simple LLVM IR function. The original code contains a loop where the multiplication instruction is unnecessarily recomputed in every iteration. After applying LICM, the multiplication is hoisted to the preheader, optimizing the loop by reducing redundant computations.

a) Original Code:: The original LLVM IR:

```
define void @simple_loop(i32 %N) {
entry:
  br label %loop

loop:
  %i = phi i32 [0, %entry], [%inc, %loop]
  %res = mul nsw i32 5, 7
  %inc = add nsw i32 %i, 1
  %cond = icmp slt i32 %inc, %N
  br i1 %cond, label %loop, label %exit

exit:
  ret void
}
```

In the original code, the instruction `%res = mul nsw i32 5, 7` is computed inside the loop in every iteration, even though its operands are constants and do not depend on the loop variable.

b) After LICM:: After applying LICM, the optimized LLVM IR:

```
define void @simple_loop(i32 %N) {
entry:
  %res = mul nsw i32 5, 7
  br label %loop

loop:
  %i = phi i32 [0, %entry], [%inc, %loop]
  %inc = add nsw i32 %i, 1
  %cond = icmp slt i32 %inc, %N
  br i1 %cond, label %loop, label %exit

exit:
  ret void
}
```

The multiplication instruction `%res = mul nsw i32 5, 7` has been moved to the preheader (just before the `br label %loop`), ensuring that it is computed only once, outside the loop.

This example highlights how LICM can reduce the computational cost of loops by hoisting invariant operations outside the loop body.

II. Implementation Details

Our loop-invariant code motion pass is implemented as an out-of-tree plugin for LLVM's *new pass manager* (NPM). It detects and moves invariant instructions from the loop body into the preheader when possible. This section briefly highlights our approach and key considerations.

A. Pass Implementation Methodology

We register our pass in `llvmGetPassPluginInfo()`, enabling it to be invoked via:

```
opt -load-pass-plugin=./MySimpleLICMPass.so \
    -passes="loop(my-simple-licm-pass)"
```

Within the `run()` method of our pass, we:

1) Check whether a loop has a unique *preheader*.
2) Iterate through all instructions in the loop, identifying those with no side effects and loop-invariant operands.
3) Record such instructions in a set, then hoist them before the preheader's terminator once discovery converges.

If any modifications are made, we return `PreservedAnalyses::none()`, invalidating analyses that may depend on the changed IR.

B. Key Data Structures and Algorithms

We maintain a `DenseSet<Value *>` to track discovered loop-invariant instructions. In each iteration, any instruction that is *safe to speculate* (via `isSafeToSpeculativelyExecute`) and whose operands are already outside the loop or in our invariant set is deemed *hoistable*. Once no further instructions qualify, the process ends.

C. Integration with the LLVM Pipeline

Our pass is best demonstrated on IR that is either unoptimized or has only been partially optimized (*e.g.*, using mem2reg). A typical workflow is:

1) **IR Generation:**
   `clang -Xclang -disable-O0-optnone -O0 -emit-llvm -S test.c -o test.ll`
2) **Mem2Reg:**
   `opt -passes=mem2reg -S < test.ll > test_mem2reg.ll`
3) **Run Our Pass:**
   `opt -load-pass-plugin=./MySimpleLICMPass.so -passes="loop(my_simple_licm_pass)" -S < test_mem2reg.ll > test_licm.ll`
4) **Final Compilation:**
   `clang -O0 test_licm.ll -o test_licm`

Although LLVM's built-in LICM may perform more sophisticated analysis (*e.g.*, alias checks), our pass illustrates the fundamental mechanics of hoisting.

D. Handling Edge Cases

We skip loops without a single preheader and do not hoist instructions that *may read or write memory*, as we lack alias analysis. By excluding `PHI` nodes and terminators (unhoistable by default), we preserve correct data and control flow. Despite these constraints, our pass remains valid and effective for purely arithmetic or side-effect-free instructions.

III. Experimental Evaluation

This section details our experimental setup, describing the test cases used to assess the Loop-Invariant Code Motion (LICM) pass, reporting performance results, comparing with baseline (unoptimized) code, and discussing the statistical significance of observed improvements.

A. Description of Test Cases and Benchmarks

To evaluate our pass under varying complexity and real-world relevance, we selected the following C programs:

1) **Basic Invariant Loop (basic.c):** A straightforward loop running $\texttt{N} = 10^7$ iterations that repeatedly adds `(b*c)` and a small modulo result `i % 100`. The constants `b=5` and `c=7` are trivially loop-invariant, providing a clear demonstration of the hoisting transformation.
2) **Nested Loop (nested_loop.c):** Two nested loops operating on a 2D array `arr[M][N]`. Within the inner loop, some operations involve loop-invariant constants (e.g., multiplying `k=8` by `i`). This test verifies that our pass can identify invariants in an inner loop context and handle typical array-based workloads.
3) **Matrix Multiplication (matrix_multiplication.c):** A classic $\texttt{N} \times \texttt{N}$ matrix multiplication routine with `N=512`. Although matrix multiplication is memory-bound and dominated by load/store operations, there are still some opportunities for hoisting loop-invariant operations (like pointer arithmetic or extended index calculations). This benchmark highlights how our pass behaves on a more realistic numeric kernel.

B. Performance Measurements and Analysis

All experiments were compiled into LLVM IR at `-O0` with `-Xclang -disable-O0-optnone` to allow transformations. We then applied either:

- **Baseline**: No custom LICM pass (just baseline IR).
- **LICM**: Our pass (with optional mem2reg if needed).

Execution time was measured using `clock()` in each program. Table I summarizes the observed runtimes (in seconds).

TABLE I
Runtime Comparisons for Baseline vs. LICM.

Test	Baseline (s)	LICM (s)	Speedup
basic.c	0.089	0.068	1.31×
nested_loop.c	0.007	0.005	1.40×
matrix_multiplication.c	1.312	1.261	1.04×

From these measurements, we observe:

- **basic.c**: Simple arithmetic showed a clear 1.31× speedup, indicating that hoisting trivial constants (`b * c`) effectively reduces the loop body cost.
- **nested_loop.c**: Even with a memory-bound inner loop, removing repeated calculations (multiplying `k * i`) boosted performance by about 1.40×.
- **matrix_multiplication.c**: We see a modest 1.04× improvement, likely because the kernel is heavily dominated by loads and stores, limiting the benefits of hoisting a few pointer arithmetic instructions.

C. Comparison with Unoptimized Code

Table I highlights that, for smaller or more arithmetic-centric loops (`basic.c` and `nested_loop.c`), our LICM pass can deliver significant speedups under a baseline `-O0` scenario. However, for a more complex, memory-heavy workload like matrix multiplication, the impact is less pronounced. Despite this, any reduction in loop overhead remains valuable for large-scale runs.

D. Statistical Significance of Improvements

Overall, these results confirm that hoisting loop-invariant arithmetic can yield tangible speedups in simpler loops, while more memory-intensive workloads benefit to a lesser degree.

IV. Correctness Proof

This section provides a brief justification of our LICM pass's correctness. We argue that hoisting loop-invariant instructions preserves program semantics, discuss how invariants remain valid after transformation, review potential edge cases, and summarize relevant testing outcomes.

A. Formal/Informal Proof of Correctness

Let I be an instruction in the loop body L with iteration variable $i \in [0, N)$. If I satisfies:

1) All operands of I are defined outside L or are themselves loop-invariant, and
2) I has no side effects (e.g., is *safe to speculate*),

then I's result is identical across all iterations of L. Therefore, evaluating I once *before* entering L is equivalent to evaluating I in each iteration, preserving the overall outcome of the program. Formally, if the set of values $\{\text{op}_j\}$ that L depends on does not change over iterations, the function $I(\{\text{op}_j\})$ is constant. Thus, moving I outside the loop does not alter the sequence of visible side effects or final results.

B. Invariant Preservation

Our pass relies on the notion of *loop invariants*: values that do not change as i progresses from 0 to $N - 1$. We track loop-invariant instructions in a fixed-point iteration, ensuring that each newly discovered invariant relies only on previously proven invariants or values outside L. Once this process converges, hoisted instructions remain valid in the preheader block without affecting data or control flow.

C. Edge Case Analysis

- **No Preheader:** If `L.getLoopPreheader()` is `nullptr`, we skip hoisting to avoid creating invalid or ambiguous control flow paths.
- **Memory Instructions:** We do not hoist instructions that read or write memory (unless proven safe by alias analysis), preventing incorrect reordering of memory accesses.
- **Terminator/PHI Nodes:** These are never hoisted, as they depend on control flow or represent multi-edge data merges, preserving correct SSA semantics.
- **Nested/Irregular Loops:** Our pass iterates once per loop and inspects each loop individually. If a loop's structure is unrecognized (multiple headers, no back edge, etc.), we do not perform LICM.

D. Testing Methodology and Results

We validated correctness using **Functional Equivalence Tests:** For each test program (Section III), we confirmed the output matched the baseline code's output. All tests produced identical results with or without LICM, confirming that our pass preserves functional correctness while achieving performance improvements in select cases.