Loop Invariant Code Motion: Optimization and Evaluation

Chen, Hao-Wen
Department of Computer Science and Information Engineering
National Taiwan University
Email: b11902156@ntu.edu.tw

***Abstract*—Loop Invariant Code Motion (LICM) is a fundamental compiler optimization that moves computations invariant to a loop outside of the loop body. This report details the algorithm design, implementation, and correctness proof of LICM within the LLVM framework, enriched with practical test cases demonstrating its application and impact.**

I. Algorithm Design

A. Optimization Strategy

Loop Invariant Code Motion identifies statements within a loop that compute the same result regardless of the number of iterations. These statements are hoisted outside the loop to reduce redundant computations, improving runtime efficiency.

B. Theoretical Analysis

The theoretical improvement is proportional to the number of loop iterations. If a loop executes N times, hoisting an invariant computation saves $N - 1$ redundant executions. For nested loops, the savings are multiplicative based on the nesting depth.

C. Implementation Considerations

The LICM algorithm requires:

- Detecting loop-invariant instructions.
- Verifying safety of movement to ensure semantics are preserved.
- Handling side effects, such as memory writes or function calls.

D. Code Example

Consider the following code snippet:

```
for (int i = 0; i < n; i++) {
    int x = a + b;
    arr[i] = x * i;
}
```

After applying LICM:

```
int x = a + b;
for (int i = 0; i < n; i++) {
    arr[i] = x * i;
}
```

II. Implementation Details

A. LLVM Pass Methodology

The LICM pass in LLVM analyzes loops using the dominator tree and loop structure analysis. Instructions are hoisted to the preheader block of the loop when proven invariant.

B. Key Data Structures

- `LoopInfo`: Identifies natural loops in the control flow graph.
- `DominatorTree`: Ensures that hoisted instructions dominate all loop exits.
- `AliasAnalysis`: Verifies memory safety.

C. Integration with LLVM Pipeline

LICM integrates with LLVM's optimization pipeline, typically running after loop analysis passes like `-loop-simplify`. It updates analyses such as `DominatorTree` and `LoopInfo` post-transformation.

D. Edge Case Handling

Special cases include:

- Loop-dependent branches.
- Non-deterministic functions or memory writes.
- Multi-threaded memory models.

III. Experimental Evaluation

A. Test Cases and Benchmarks

To evaluate the effectiveness and correctness of LICM, we present the following test cases, implemented in C. These test cases demonstrate different scenarios where LICM can be applied or where special handling is required.

B. Test Case: Basic LICM

```
void basicTest(int n, int a, int b) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        int x = a + b;
        sum += x * i;
    }
}
```

This test involves a single loop with an invariant computation `a + b`, which can be hoisted outside the loop to avoid redundant calculations.

C. Test Case: Nested Loops

```
void nestedTest(int n, int m, int a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        int constant = a + i;
        for (int j = 0; j < m; j++) {
            product *= constant * j;
        }
    }
}
```

This test evaluates LICM in the presence of nested loops. Only invariants in the outer loop can be hoisted.

D. Test Case: Conditional Branches

```
void conditionalTest(int n, int c) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (c > 0) {
            int temp = c * 2;
            count += temp;
        }
    }
}
```

This test includes conditional branches. LICM can hoist invariants under specific conditions.

E. Test Case: Non-Invariant Function Calls

```
void nonInvariantTest(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        int x = rand();
        sum += x;
    }
}
```

Here, `rand()` prevents LICM as it generates a new value in every iteration.

F. Test Case: Multi-threading Edge Case

```
void multiThreadTest(int n, int *arr) {
    int fixedValue = arr[0];
    for (int i = 0; i < n; i++) {
        arr[i] = fixedValue + i;
    }
}
```

This test highlights potential challenges in multi-threaded environments, where shared variables might change unexpectedly.

IV. Correctness Proof

A. Invariant Preservation

LICM preserves correctness by hoisting only those computations that are:

- Loop-invariant, determined using def-use chains.
- Dominating all loop exits to ensure visibility.
- Free of side effects, verified using AliasAnalysis.

B. Edge Case Analysis

Edge cases such as exceptions or floating-point precision are addressed by ensuring the transformation respects program semantics.

C. Testing Methodology

Tests include:

- Unit tests for synthetic cases.
- Integration tests on real-world benchmarks.
- Validation using LLVM's `opt` tool.

V. Conclusion

LICM demonstrates significant performance improvements with minimal overhead. The provided test cases showcase its practical applicability and effectiveness. Its integration into LLVM's optimization pipeline ensures applicability across diverse workloads.