Final Report
Partial Redundancy Elimination Using Lazy Code Motion

Shambhavi Kuthe
Virginia Tech

Rohit Mehta
Virginia Tech

1 Problem Statement and Motivation

This report explores Partial Redundancy Elimination (PRE) using Lazy Code Motion (LCM), a technique similar to Loop Invariant Code Motion (LICM) [4], but it is more conservative in the sense that it only moves code that is guaranteed to be executed outside the loop. In PRE, the compiler first identifies expressions that are evaluated within the loop, and then determines if these expressions are executed on every iteration of the loop or only on some iterations. If an expression is executed only on some iterations, the compiler creates a new basic block that computes the expression and moves this block outside the loop. The new basic block is then executed only on the iterations where the expression is evaluated, reducing the total number of instructions executed within the loop. The idea is to eliminate the computations that may be executed multiple times with same input values. In this project, we implemented a PRE optimization pass in LLVM and measure it's results on variety of applications. Consider the following code fragment:

```
int a = b + c;
int d = e + f;
int g = b + c;
int h = i + j;
int k = b + c;
```

In this code, the expressions b + c are computed three times. These computations are partially redundant, meaning that they are not exactly the same, but they compute the same value. Partial redundancy can occur when two or more expressions are computed that are not exactly the same, but they compute the same value for some inputs. In this case, the expressions b + c are not identical, but they produce the same value every time they are computed. To eliminate this partial redundancy, we can use a technique called partial redundancy elimination (PRE). PRE works by identifying partially redundant expressions and factoring them out into a separate computation that is executed only once, with the result stored in a temporary variable. This temporary variable is then used in place of the redundant expressions. After applying PRE, the code fragment would be transformed as follows:

```
int temp = b + c;
int a = temp;
int d = e + f;
int g = temp;
int h = i + j;
int k = temp;
```

By factoring out the partially redundant expression b + c into a temporary variable, we have eliminated two of the three redundant computations. This reduces the total number of computations executed by the program and can improve its performance. In summary, partial redundancy occurs when two or more expressions are computed that are not identical, but they produce the same value for some inputs. Partial redundancy elimination (PRE) is a technique that can be used to eliminate partially redundant expressions by factoring them out into a separate computation that is executed only once, with the result stored in a temporary variable. This is the basic idea and motivation behind our project.

2 Related Work

2.1 Partial Redundancy Elimination based on SSA Form:

[1] The paper describes a new algorithm for partial redundancy elimination (PRE) that is based on Static Single Assignment (SSA) form. PRE is a compiler optimization technique that aims to eliminate expressions that are computed repeatedly in a program by computing them only once and then reusing the result. The SSA form is a popular intermediate representation used by compilers that makes it easier to perform many kinds of optimizations, including PRE. The authors present their algorithm, called the SSA-based PRE algorithm, which works by first identifying expressions that may be partially redundant, and then using a set of rules to

transform the program into a form where the redundancies can be eliminated. The algorithm is designed to work efficiently on large programs and to handle complex control flow structures.

2.2 Lazy Code Motion by J. Knoop, O. Ruthing, and B. Steffen:

[3] The paper describes a new compiler optimization technique called "lazy code motion" that aims to eliminate redundancies in a program by moving computations out of loops and into surrounding code. The technique works by first identifying expressions that can be moved out of a loop, and then deferring the actual movement of the expression until it is actually needed. This avoids the need to perform expensive computations that may not be used, and can result in significant performance improvements. The authors present their algorithm for lazy code motion, along with a formal analysis of its correctness. They also describe a prototype implementation of the algorithm and provide experimental results showing its effectiveness on a set of benchmarks. Overall, the paper provides a valuable contribution to the field of compiler optimization, and the lazy code motion technique has become a standard optimization used in many modern compilers.

2.3 A Variation of Knoop, Ruthing, and Steffen's Lazy Code Motion

[2] The paper presents a variation of the "lazy code motion" technique introduced by Knoop, Ruthing, and Steffen in 1992. The variation is designed to address some of the limitations of the original technique, such as its inability to handle certain loop structures. The authors present their variation of the algorithm, which works by first identifying expressions that can be moved out of a loop, and then performing a more sophisticated analysis to determine whether the expression should be moved. The variation also includes additional optimizations to reduce the number of computations performed outside of the loop. The paper includes a detailed description of the algorithm, along with examples to illustrate how it works. The authors also compare their variation to the original technique and show that it can handle a wider range of loop structures and often produces better results. Overall, the paper provides a valuable contribution to the field of compiler optimization, and the variation of lazy code motion introduced by Drechsler and Stadel has become a standard technique used in many modern compilers.

3 Implementation

To implement the PRE Technique we have implemented these 4 Passes:

- Anticipated Expressions
- Will be Available Expressions
- Postponable Expressions
- Used Expressions

3.1 Anticipated Expressions

Anticipated expressions are expressions that are evaluated outside of a loop, but whose results are used inside the loop. In other words, they are expressions that are computed before the loop begins and whose values are "anticipated" by the loop as it executes. We can say that an expression is anticipated if its value computed at a program point p is used along all subsequent paths.

For our implementation, we use our dataflow analysis framework and set the analysis parameters accordingly to get the analysis result. We also define our transfer functions to calculate the used and killed expressions to give the resultant transfer function.

	Anticipated Expressions
Domain	Sets of Expressions
Direction	Backward
Transfer Function $f_b(x)$	$f_b(x) = EUse_b \cup (x - EKill_b)$
Meet	$\cap$
Boundary	in[exit] = $\varnothing$
Initialization	in[b] = {all expressions}

3.2 Will be Available Expressions

An expression is termed as will be available if it is anticipated at a point but subsequently killed on all paths reaching program point p. Based on the results of Anticipated and will be available passes, we compute the earliest position where an instruction can be placed.

For our implementation, we use our dataflow analysis framework and set the analysis parameters accordingly to get the analysis result. We also define our transfer functions to calculate the killed expressions and use the result from the anticipated expressions pass to give the resultant transfer function.

	Available Expressions
Domain	Sets of Expressions
Direction	Forward
Transfer Function $f_b(x)$	$f_b(x) = (Anticipated[b].in \cup x) - EKill_b)$
Meet	$\cap$
Boundary	out[entry] = ø
Initialization	out[b] = {all expressions}

$$earliest[B] = anticipated[B].in - available[B].in$$

3.3 Postponable Expressions

An expression is Postponable at a program point if all paths leading to program point p have seen earliest placement of e but not a subsequent use. Using the result of Postponable expressions pass and the earliest calculated for each basic block, we calculate latest frontier as:

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap \left(e\_use_B \cup \neg(\bigcap_{S,succ(B)} (earliest[S] \cup postponable[S].in))\right)$$

	Postponable Expressions
Domain	Sets of Expressions
Direction	Forward
Transfer Function $f_b(x)$	$f_b(x) = (earliest[b] \cup x) - EUse_b)$
Meet	$\cap$
Boundary	out[entry] = ø
Initialization	out[b] = {all expressions}

For our implementation, we use our dataflow analysis framework and set the analysis parameters accordingly to get the analysis result. We also define our transfer functions to calculate the used expressions and use the result from the will be available pass to get the earliest expressions to give the resultant transfer function.

3.4 Used Expressions

Used expressions are expressions whose results are used within a basic block or loop. They can be identified using dataflow analysis techniques and can be used in conjunction with various optimization techniques to reduce the total number of computations executed by the program and improve its performance. The goal of identifying used expressions is to eliminate unnecessary computations that do not affect the outcome of the program. If an expression is used within a basic block or loop, it must be computed at least once. However, if the result of the expression is not used after the first computation, subsequent computations can be eliminated.

For our implementation, we use our dataflow analysis framework and set the analysis parameters accordingly to get the analysis result. We also define our transfer functions to calculate the used expressions and use the result from the postponable pass to get the latest expressions to give the resultant transfer function.

	Used Expressions
Domain	Sets of Expressions
Direction	Backward
Transfer Function $f_b(x)$	$f_b(x) = (Euse[b] \cup x) - latest[b])$
Meet	$\cup$
Boundary	in[exit] = ø
Initialization	in[b] = ø

3.5 Preprocessing

In the Preprocessing pass, we first split all the basic blocks such that every instruction is placed in a separate basic block for ease of implementation. Splitting of basic blocks is then followed by placing empty basic blocks on the critical edges i.e. blocks are placed on all the edges whose destination block has multiple predecessors.

3.6 Code Motion

To implement the lazy code motion algorithm, we first need to identify the blocks for which we need to introduce the temporary variables then followed by inserting the new temporary variables at the beginning of the basic blocks. Then we need to identify the blocks in which we need to replace the instructions and replace the instructions with the corresponding temporary variable defined.

4 Benchmarks

The code is evaluated against the following benchmarks -

- Misc/Flops - The 'c' program called Flops.c aims to calculate your system's floating-point 'MFLOPS' rating for FADD, FSUB, FMUL, and FDIV operations by using particular 'instruction mixes.' It can estimate the PEAK MFLOPS performance by utilizing register variables and minimizing interaction with main memory. Additionally, the program utilizes small execution loops to fit in any cache.
- BenchmarkGame/nsieve-bits
- Shootout/ary3

- Shootout/nestedloop

Below is the comparison of our LCM Implementation and LICM Implementation from Assignment 3.

Benchmark Name	PRE using Lazy Code	LICM
Benchmarks/Misc/Flops	5.361	5.423
Benchmarks/Misc/nsieve-bits	0.634	0.679
Benchmarks/Shootout/ary3	3.184	3.063
Benchmarks/Shootout/nestedloop	17.564	18.121

5 Output

Below is the final output after the code motion pass.

```
LCM Pass
Function name: func
Domain: {"i32 %0" + "i32 %1", "i32 undef" + "i32 %1", "i32 %0" - "i32 %1"}

BB Name: .split2
  Instr           %t = add i32 undef, %1
  Store           store i32 %t, ptr %tempPtr, align 4
BB Name: .split3.split
  Instr           %t1 = add i32 undef, %1
  Store           store i32 %t1, ptr %tempPtr, align 4
BB Name: .split2
  Load            %loadValue = load i32, ptr %tempPtr, align 4
Replaced.............
llvm-dis ./tests/test4_out.bc
```

The output shows the LCM pass run on a test file given in the project deliverables. The store and Instr print statements show the created temporary variables and Load print statement shows the replaced instructions.

6 Future Work

There is a lot of scope for future work in the current implementation of our lcm pass. Some of the work would include evaluating all the possible cases for other non-binary instructions.

7 Conclusion

We implemented the Partial Redundancy Elimination using Lazy code motion in LLVM and carried out benchmarks to compare our implementation with the Loop invariant code motion on multiple benchmark c files. We carried out our benchmarks on our local machine and measured the runtimes and discussed the evaluation.

References

[1] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form, 06 1997.

[2] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of knoop, rüthing, and steffen's lazy code motion. 28(5), 1993.

[3] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. New York, NY, USA, 1992. Association for Computing Machinery.

[4] David Monniaux and Cyril Six. Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion. New York, NY, USA, 2021. Association for Computing Machinery.