Alexander Barry: amb
Ahmad Humayun: ahmad35

Group: amb_and_ahmad35
ECE/CS 5544 Spring 2023

Final Project - Lazy Code Motion

Section 1: Problem Statement

Modern compilers perform several optimization "passes" on a program for various objectives. Most of these passes serve to either reduce the amount of unnecessary code for security reasons or reduce the amount of unnecessary computation in the program in order to make it more efficient. One such optimization pass is a Lazy Code Motion (LCM) pass that eliminates the redundant and partially redundant computation of expressions by computing the expression once, caching the result inside a temporary variable and reusing it. However simply caching the result at the earliest possible opportunity can increase register lifetimes, hurting performance. LCM seeks to balance this tradeoff by creating temporary variables at the latest possible insertion points.

In this project, we implement an LCM pass using LLVM. We seek to evaluate the performance benefits of removing redundant computations on a set of microbenchmarks. Furthermore, we evaluate the benefit of the Postponable Expressions pass, which reduces register pressure by delaying the caching of the redundant computation where possible. We compare the two versions of LCM. Finally, we also evaluate the difference between LCM and Loop Invariant Code Motion (LICM) for optimizing loop invariants.

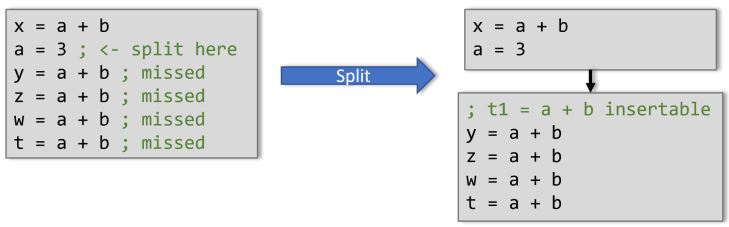
Section 2: Approach

LCM consists of four standard dataflow analysis (DFA) passes, that are each implemented as a separate LLVM pass: (1) Anticipated Expressions; (2) (Will be) Available Expressions; (3) Postponable Expressions and (4) Used Expressions. Apart from these we have four additional passes that serve different purposes. We have a preprocessing pass that runs before the DFA passes and sets up the CFG with additional basic blocks. We implement LCM as a separate pass, that depends on these five DFA passes, to transform the code and insert temporaries and replace redundant expressions. In order to perform an ablation study to evaluate the usefulness of the postponable expressions pass, we implement a seventh pass that uses the first two DFA passes and simply caches the redundant expression at the earliest possible program point. Finally, we include a post-processing pass that re-consolidates blocks that were split previously during preprocessing to avoid increased branching overhead.

2A. Preprocessing Pass: The preprocessing pass inserts basic blocks (BBs) into the CFG that will be useful when we start inserting temporaries. The goal of doing this is two fold: (1) Security, we don't want to change program semantics; and (2) Optimization opportunities, we want to increase the opportunities of optimization as much as possible to get the most out of LCM.

The first kind of splitting we do is along "critical edges", which are edges that go from a BB with multiple successors to a BB with multiple predecessors. Since LCM attempts to optimize redundant expressions by inserting temporary computations, it carries the risk of introducing an expression computation along a path that did not originally exist on that path. This can lead to program semantics being changed. This problem occurs on critical edges. However, if we add a new BB along this edge, we can insert any temporaries we want over there, without affecting the semantics of the original program. As a simplification, we split any edge between a block and a successor with multiple predecessors.

The second kind of splitting we do is for increasing avenues for eliminating redundant computations for LCM. The algorithm in the class slides and standard ALSU text make a simplifying assumption of one BB per instruction, which is not true practically [1]. Since DFA analysis computes the results of each pass at the program points before the entry and after the exit of a BB, these are the only points the analysis holds for and therefore, are the only points at which we can insert expressions. This means we cannot insert a temporary variable in the middle of the basic block which can lead to multiple uses of an expression being left



unoptimized. The figure on the left shows a case where this may occur. Since a=3 reassigns an operand of a+b we cannot insert t1=a+b at the start of the basic block to replace all the redundant a+b expressions. However, if we split the block after a=3 we open up an avenue for optimization. A similar opportunity arises when multiple uses of an expression occur within the same basic block, but this expression was not seen prior and may not be used after the block. The algorithm would otherwise not detect the potential temporary variable insertion at the start of the block, as the Used Expressions out set for the BB will not include this expression.

2B. Anticipated Expressions Pass: This pass computes expressions that are "anticipated" at the entry and exit of each BB. "Anticipation" of an expression at a program point means that the expression will eventually be computed along every path downstream from that program point. This is a standard backwards dataflow pass that works on a set of expressions and uses intersection as its meet operator.

2C. (Will be) Available Expressions Pass: This pass computes expressions that will be available at the entry and exit of each BB. These are expressions that are anticipated at a program point but not subsequently killed. This is a standard forwards dataflow pass that works on a set of expressions and uses intersection as its meet operator. This can be used to find the earliest placement location for an expression, since the earliest point we can place the expression is at a point where it has been anticipated but is not yet available.

2D. Postponable Expressions Pass: However, placing an expression at the earliest point can lead to long register lifetimes. It is smarter to postpone this computation to right before a use occurs. The Postponable pass finds the expressions for which insertion of a temporary variable can be postponed. An expression can be postponed to a program point if that expression has already seen the earliest placement at that point but has not already been used. This is a standard forwards dataflow pass that works on a set of expressions and uses intersection as its meet operator. This can be used to compute the latest program points where the temporary variable can be inserted.

2E. Used Expressions Pass: Finally the Used expressions pass computes if an expression is used after a program point, which is similar to liveness but for expressions. This is a standard backwards dataflow pass that works on a set of expressions and uses union as its meet operator.

2F. LCM: This is where the bulk of the optimization logic happens. In this pass, we use information from the Used and Postponable expressions passes to figure out the latest blocks where a temporary variable can be inserted, and which blocks require instructions to be replaced with the temporaries. The challenge here when working in SSA form, is that some insertions may be at a program point where multiple predecessor paths compute the same expression and have been replaced with temporary variables, making the temporary variable at the given program point a PHINode of two other temporaries. To make matters worse, this may be a recursive problem. Thankfully, LLVM has a helper class called SSAUpdater that can handle these code transformations for us. In the case of multiple equivalent temporary expressions computed along different paths, we provide the SSAUpdater with each symbolic value and the block at which it becomes available. The SSAUpdater is then able to replace specific uses of instructions with the correct temporary variable, inserting additional PHINodes as necessary.

There is an additional challenge, in that replacing and/or erasing an LLVM Instruction successfully updates the CFG, but does not update the universal sets of expressions we maintained throughout our dataflow analysis, and internally would corrupt our previous bit-vectors. We solve this problem by computing and maintaining maps of Instructions to the temporaries that will replace them, or in the case of multiple predecessors, the corresponding SSAUpdater. Only after we've computed all expression work do we replace the Instruction instances.

2G. LCM: Earliest Placement: This pass is almost exactly the same as the original LCM pass. The only difference is that the temporary variables are placed at the *earliest* possible program point rather than the latest. This pass depends only on the first three passes, i.e. preprocessing, anticipated and will-be available.

2H. Postprocessing Pass: The postprocessing pass iterates the BBs in the CFG and detects any that terminate in an unconditional branch to a single successor, of which it is the only predecessor. This arises from our preprocessing

simplifications. To avoid introducing unnecessary branch instructions, this pass merges the two blocks back together and eliminates the branch.

Section 3: Experimental Results

3A. Evaluation Questions: Our evaluation seeks to answer the following questions:

(1) Does LCM successfully reduce the amount of computation done by the program at runtime?
(2) How big of an impact does reducing register lifetimes have in terms of improving performance?
(3) Is LCM better than LICM for optimizing loop invariants?

3B. Microbenchmarks: Our microbenchmark suite contains seven programs, each testing a different scenario:

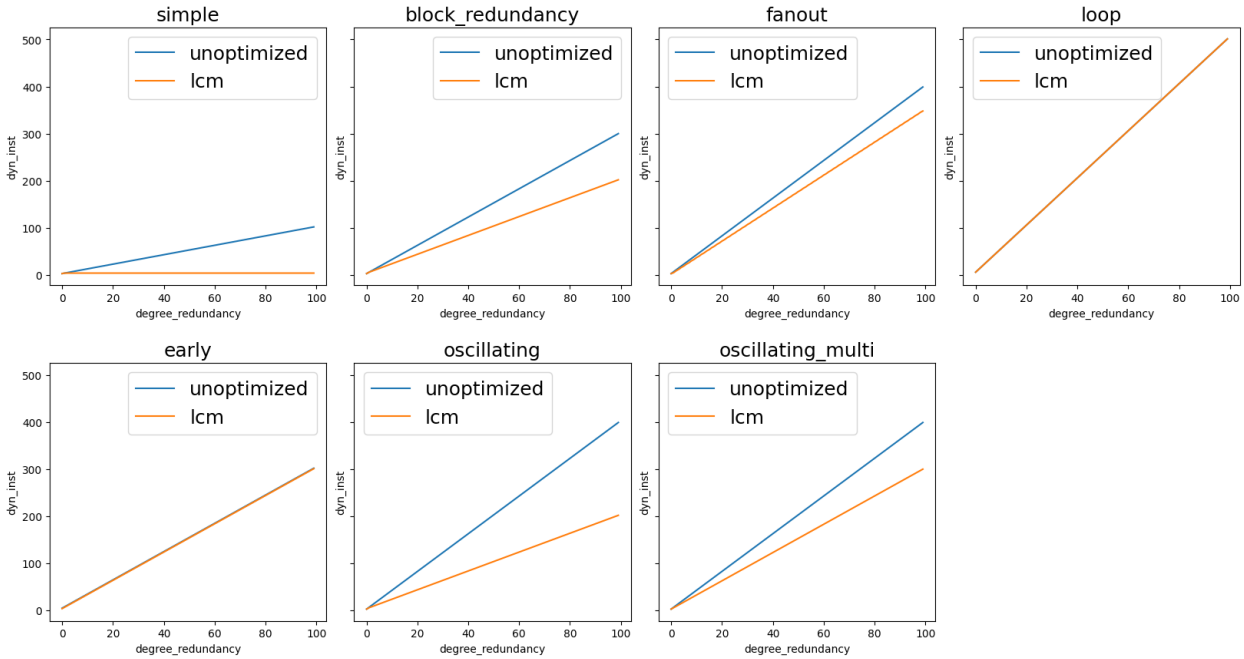
- `Simple` - This is the most basic case for redundancy, a single basic block with multiple redundant expressions.
- `Block Redundancy` - Recursively inserts the same redundant expression into only the left / true path of a conditional branching statement at increasing numbers of levels of the CFG, creating redundancy across multiple blocks. Each left path re-inserts the expression into the left / true path of its successors, widening the graph.
- `Oscillating` - Computes the same redundant expression on the true path of a conditional branch, before joining back to the same block at each level and re-evaluating the expression. This ensures partial redundancy.
- `Oscillating Multi` - Similar to oscillating, but instead of one redundant expression, computes a different redundant expression for each level of branch blocks and joining successors.
- `Fanout` - Uses a switch instruction to introduce ever-increasing amounts of successor blocks in a CFG at each level, fanning out before joining back to a common block. Each level introduces a new expression, and approximately half of the successors contain a redundant computation as their predecessor, while the other half re-computes a common redundant expression. This evaluates increasing levels of partial redundancy and its impact on the PHINode merging.
- `Loop` - Creates a simple loop, where the body re-computes an invariant expression redundantly.
- `Early` - Generates increasing numbers of BBs with unique expressions along a conditionally branching true path. Each level rejoins to a common successor block. The test finally computes a redundant expression twice at the end of the function. The test enables us to measure the effectiveness of latest vs earliest placement of temporaries.

We create 100 versions of each benchmark, varying the number of redundant expressions in the code . We automate the generation process of the benchmarks using a python script using the `llvmlite` library [2]. This library provides a python wrapper around LLVM libraries. We use its wrapper around the IRBuilder [3] class to generate the IR. The code for generating these microbenchmarks can be found in `gen-ir.py`.

Each version or iteration of the benchmark attempts to increase the level or effect of redundancy in some way. For `Simple`, this means inserting an additional evaluation of the redundant expression, allowing for one additional redundant elimination per test. For `Block Redundancy`, this means increasing the number of levels containing the redundant expression, meaning we insert an additional if/then successor block pair with the redundant expression use for each benchmark iteration. For `Oscillating` and `Oscillating Multi`, we similarly insert another level with one additional redundant expression use per benchmark version. `Fanout` behaves similarly, though in addition, at each new level of switch statement and successors, we increase the number of switch cases / successor blocks. However, as only one path through will be taken at runtime, it effectively increases the redundancy by one computation per level. For `Loop`, increasing redundancy means increasing the number of iterations of the loop (or executions of the loop body), one additional time per benchmark version. With `Early`, as we are evaluating the effect of earliest vs latest placement, each version of the benchmark inserts one additional level of the branching successors and unique computations, extending the graph in between the earliest and latest placement points for the final redundant expression, and increasing the lifetime of early temporaries placed at the start of the CFG..

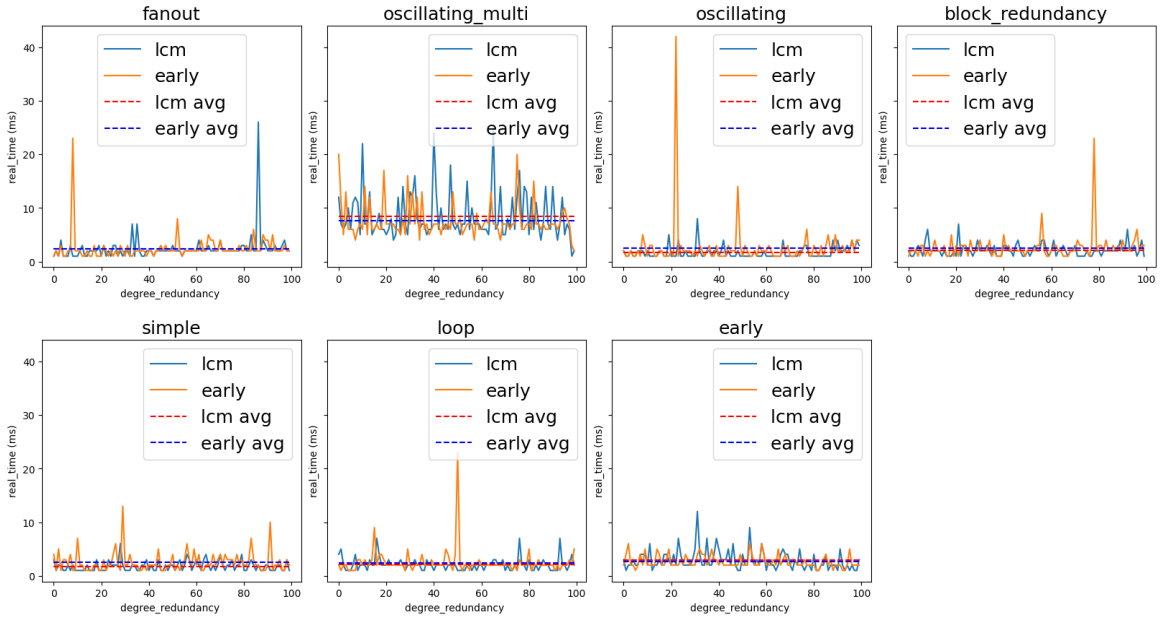
3C. Reduction in redundant computation (graphs on next page): To answer this question, we optimize all our microbenchmarks using our LCM implementation and compare the number of dynamic instructions executed using `lli`. The x-axis shows the degree of redundancy for the benchmark and the y-axis shows the number of dynamic instructions executed by the program. The graphs show more or less what we expected to see, LCM reduces the number of dynamic instructions executed. This is expected because when LCM removes redundant expressions, say a+b in x = a+b, it will not assign a

temporary variable t to x like x = t. Instead it will replace all uses of x with t. This is possible due to SSA since we know that this is the only definition of x in the code. For "loop" we see no improvement in the number of dynamic instructions executed because the Lazy Code Motion passes are unable to detect the eligible redundancy in the loop invariant. The expression in the loop body block is not available coming into the body, nor anticipated along all downstream paths from the loop header. There is no candidate location to insert a temporary.



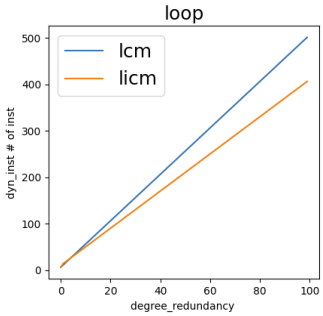
We additionally measured the number of unique postponable expressions detected by LCM, as well as the number of temporaries generated, *t*. *t* was constant for experiments where the same redundant expression was used (*t* = 1 for `Simple, Block,Early`; *t* = 2 for `Oscillating`), or else scaled with the degree of redundancy (1-to-1 for `Fanout`, 2-to-1 for `Oscillating Multi`). For `Loop`, *t* = 0. The number of postponables *p* was comparable: *p* = 0 for `Simple` and `Block`, *p* = 1 for `Early` and `Oscillating`, and *p* scaled 1-to-1 with the degree of redundancy for `Fanout` and `Oscillating Multi`.

3D. Register Lifetimes (graphs on next page): In order to measure the effect of register lifetimes on the execution of the benchmark, we measure the total execution time of the program. The idea is that with earliest placement, there will be more pressure on the register allocator and will result in longer register lifetimes. Longer register lifetimes means more data will need to be spilled to memory, and this will manifest itself in the form of longer running times. The graphs below show the time taken to execute the benchmark on our local hardware on the y axis along with the degree of redundancy on the x-axis. The solid lines show how execution time varies with degree of redundancy for each optimization and the dashed lines show the average execution time across all 100 variants of the program. The results are what we expect to see on average, albeit the data is noisy due to the small size of our benchmarks. For all of these benchmarks, there is no significant difference between the performance of the two optimizations as seen by the dashed lines. This is because the size of our microbenchmarks is too small for the differences to manifest. Furthermore the total execution time is so short (<10ms for most) that noise is significant, making the differences in the mean execution time unreliable.



We also compiled all benchmarks and ran the valgrind tool *cachegrind* on the compiled programs in an attempt to evaluate cache performance. However, almost all output was identical, indicating this was not a useful measure for our experiment.

3E: LCM vs LICM for optimizing loop invariants



To answer this question, we optimize our loop benchmark with both LICM and LCM. In this case, LICM manages to optimize the loop invariants and reduce instructions, whereas LCM optimized code gives the same performance as unoptimized code. As explained in 3C, this is because LCM was unable to detect the redundancy at all for the loop invariant. LICM meanwhile was able to hoist the invariant to a loop landing pad block and avoid the redundant instructions in each execution of the loop body.

Section 4: Conclusions

In this report, we presented our implementation of Lazy Code Motion, a hardware agnostic compiler optimization implemented as a series of LLVM passes. We showed that our LCM pass successfully reduces the number of instructions executed at runtime by the optimized code. We further showed that our LCM pass is not able to recognize all forms of redundancy, as demonstrated by LICM outperforming it on our loop benchmarks.

Section 5: References

1. Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston, MA: Pearson Addison Wesley, 2007. 644-54.
2. https://github.com/numba/llvmlite
3. https://llvm.org/doxygen/classllvm_1_1IRBuilder.html