Exploring the Performance of Loop Quasi-Invariant Code Motion

Jason T. Weeks, Ian Yelle

School of Computer and Cyber Sciences, Augusta University, Augusta, GA, United States

Summer Scholars Program — Center for Undergraduate Research and Scholarship — Innovate · Investigate · Discover

Background

Compilers are powerful tools that transform code written in programming languages into a format that computers can execute. During this transformation process, compilers can additionally perform optimizations to improve the performance of the code.

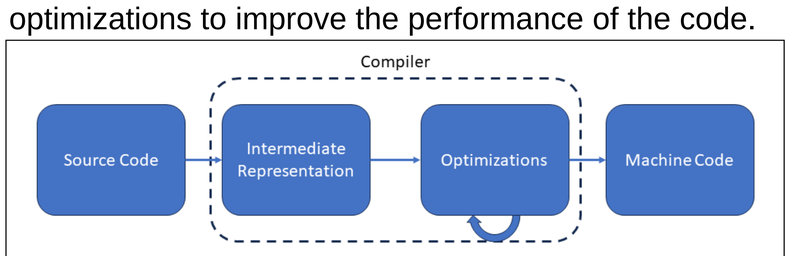


Fig. 1: A visual example of how a compiler transforms code from a programming language to machine code

Such improvements are all done without requiring any additional effort from the programmer, and new optimizations are continuously being developed to further enhance compilers. This work assesses the performance of a new optimization called Loop Quasi-Invariant Code Motion (LQICM).

Quasi-invariant code stops changing after a finite number of runs, and thus ends up running more than necessary. Therefore, if it is removed from the loop and only run the number of times it needs to, it would in theory decrease the runtime of the code.

The process of removing this code is called hoisting:

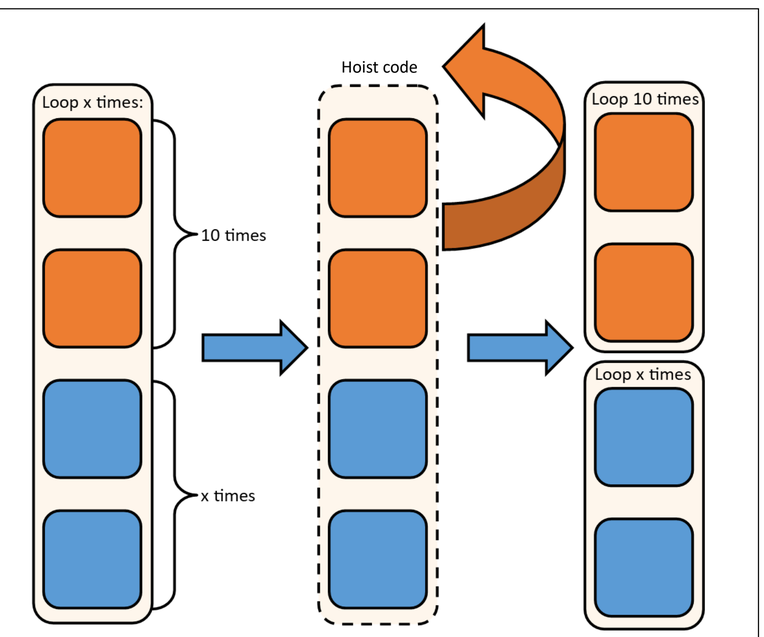
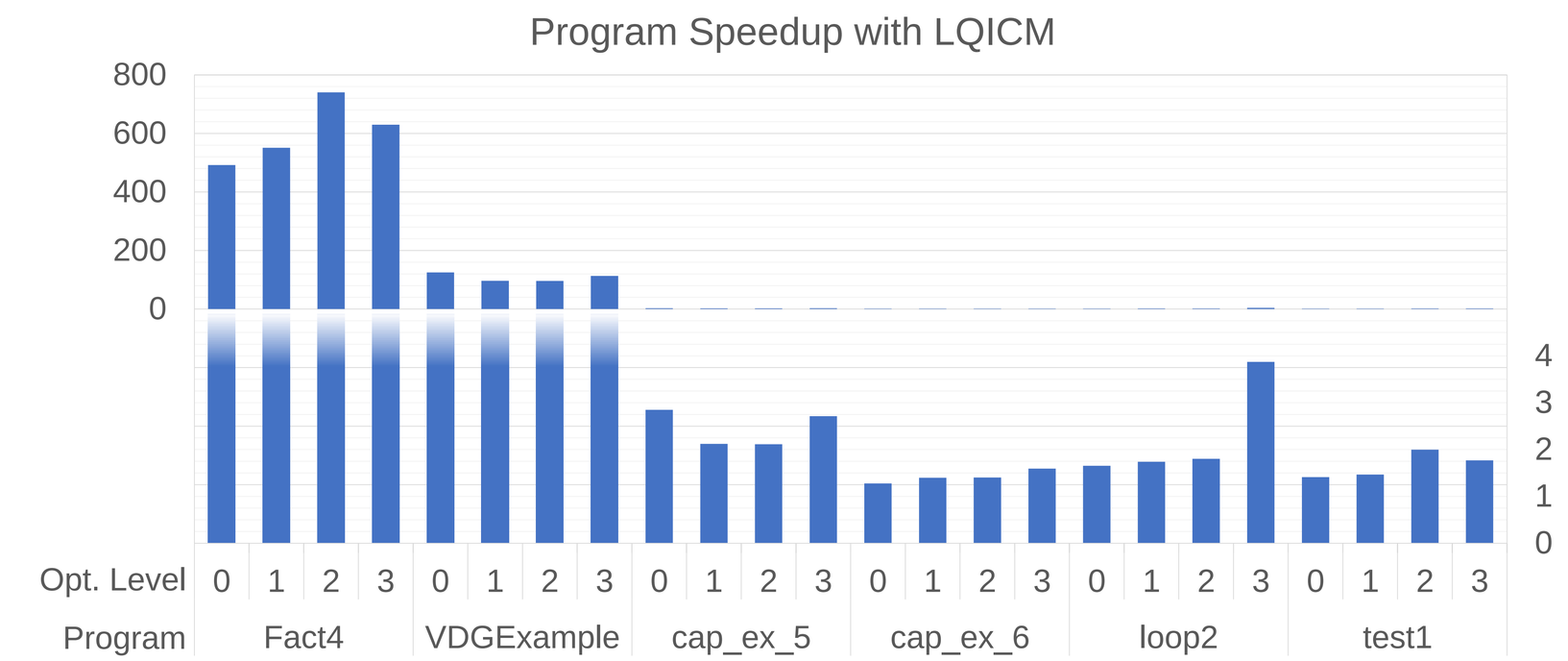


Fig. 2: A visual example of how hoisting quasi-invariant code works.

Hoisting, as shown in the diagram above, involves removing the quasi-invariant code from within the loop and placing it in a separate loop that only runs the specific number of times that specific code needs. Once quasi-invariant code is identified, it is copied out of the original loop and placed in a new loop before the original, and making the new loop only run the number of times the code needs to be run before it stops changing, referred to as the degree of invariance.

Results



Methods

To test the performance of LQICM, we first had to find examples that had quasi-invariant code blocks within them. From there, we performed the transformation manually, so that we would have an unoptimized and optimized version of the code example in question. We did this for each example we obtained (4 were found in existing code repositories, and 2 were created, resulting in 6 total files). We then measured the runtime of each program, using varying levels of compiler optimizations as well to measure the benefits of our optimization with and without compiler optimizations. Once we gather these results, we then calculate the speedup ratio by dividing the runtime of the unoptimized code with the runtime of the optimized code. See chart above for specific results.

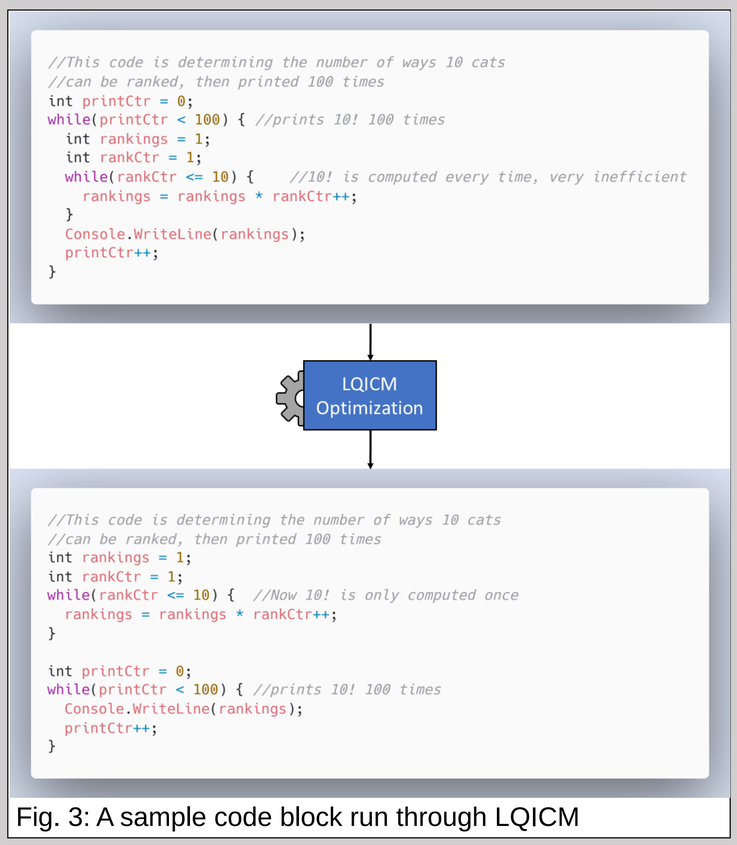


Fig. 3: A sample code block run through LQICM

Conclusion

Every program experienced some measurable speedup due to LQICM, with two of our examples being over 80 times faster thanks to our optimization. This means that, performance-wise, LQICM is a viable way to optimize code, both with and without additional optimizations added.

Future Work

Future work on this project will involve implementing the actual compiler pass so that the hoisting process can be automated, along with finding more samples that demonstrate quasi-invariant code. In addition, the benchmarking software can continue to be improved upon to reduce variability and ensure more accurate results. Finally, with the increased sample size we can run the improved benchmark to get a clear understanding of the performance boost our optimization offers.

Benchmark/References

Acknowledgements

- Clément Aubert and Neea Rusch for their advisement
- Justice Howley for his work up to SSP
- Mark Holcomb for his invaluable advice throughout the project
- CURS for funding and allowing us to continue our research