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

1 Introduction

In this work, we aim to verify the newly conceptualized compiler optimization, *Loop Quasi-Invariant Code Motion* (LQICM) [2], which addresses quasi-invariant (QI) code. Specifically, we first identify key patterns in programs that exhibit QI, then find real-world and constructed examples with these patterns. We then apply our optimization and observe changes to the program's runtime. The results from this offered runtime speedups up to 700 times faster with LQICM applied. The remainder of the abstract proceeds as follows: in Section 2, we define the target of our optimization and the optimization in detail, in Section 3 our methodology, and in Section 4 the results collected through benchmarking our example programs.

2 Background

Quasi-invariant code makes observable environmental changes for some number of loop runs before reaching a fixed state, referred to as the *degree of invariance*, becoming redundant in the remaining runs. Code with a degree of invariance of 1 (redundant after one or fewer loop runs) is referred to as *invariant code*, which is taken care of by the existing compiler pass, *Loop Invariant Code Motion* (LICM). However, LICM fails to remove any code of degree greater than 1, or *quasi-invariant* code. This niche is addressed by LQICM. Theoretically, removing the QI code from the loop at this point should decrease the loop's overall runtime. This code removal process is called *hoisting*. Figure 1 depicts the hoisting process, wherein the QI code block and its necessary dependencies are removed from the loop and placed in a separate loop before the remnants of the original, running only until the fixed point is reached. With the code block now hoisted, the quasi-invariance is no longer present.

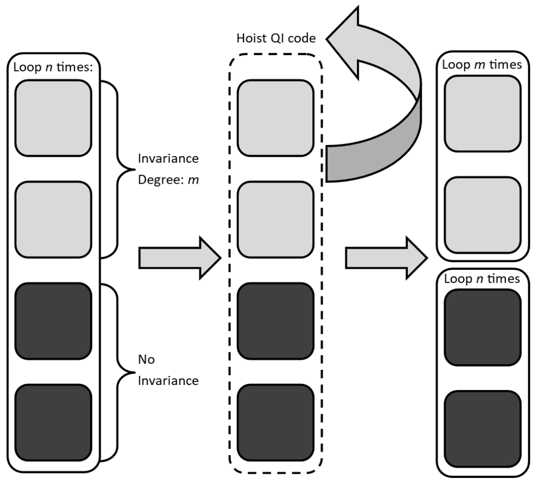

Figure 1: Hoisting in Action

To ensure the maintenance of problem semantics, one must accurately identify code to be hoisted with a dependency graph or the like. This technique was used to great effect with the code examples in [2].

3 Methods

To test the performance of LQICM, we first identify examples containing quasi-invariant code blocks, then apply the transformation manually to obtain an optimized version of the code. We measure the runtime of both the optimized and unoptimized versions in combination with various compiler optimizations to measure the benefits of LQICM with and without these optimizations. Using these results, we calculate the speedup ratio by dividing the runtime of the unoptimized code by the runtime of the optimized code: $S = T_{\text{Pre}}/T_{\text{Post}}$.

3.1 Gathering Examples

We test our optimization with six examples, four from pre-existing code repositories [3] and two new examples of our own design. Upon manually applying the optimization, creating pre- and post-optimization forms of each program, we are left with 12 files. These files are compiled with LLVM, each generating an intermediary representation. We compare the optimized intermediaries to their unoptimized counterparts to ensure semantic equivalence, adjusting the optimized files as necessary.

3.2 Building a Benchmark

In order to measure the runtime of the optimized code, we developed a C benchmark to record data at various optimization levels. While initially code examples needed to be pasted into the benchmarking program, we have since added a function to read all files from a designated folder. The benchmark then iterates through the files at various optimization levels, running a set number of times and recording the mean and average runtimes. Once every file is measured, the collected results are exported to `.csv` and multiple `.txt` formats [1].

4 Results

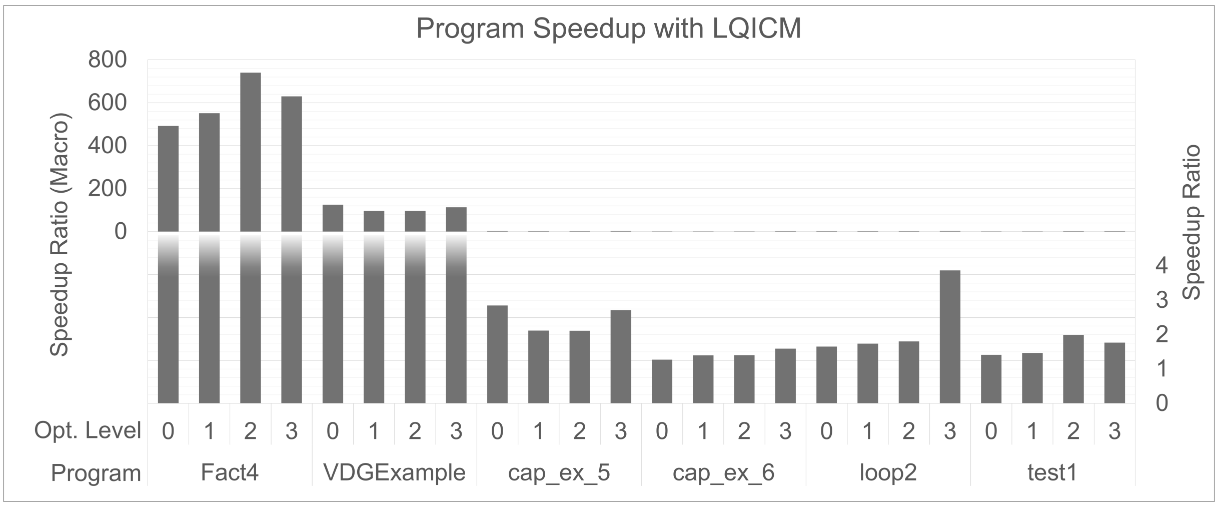

Figure 2: Benchmark Results (Top graph is macro scale, bottom graph is normal scale)

As seen in Figure 2, every program experienced some measurable speedup due to LQICM, with one example achieving over $80\times$ speed-up and another over $700\times$ speed-up. This performance demonstrates LQICM as a viable optimization technique, regardless of additional optimizations.

5 Difficulties and Challenges

Our work on verifying the performance of LQICM was beset by two main issues: finding and identifying QI code and working with LLVM. Finding code examples with QI proved difficult, as determining the presence of QI is non-trivial, requiring the use of a dependency graph for each file. As for LLVM, since we are using code snippets for our test programs, it tends to over-optimize these programs, often reducing the code down to only return statements and removing all semantics. This was remedied by applying the *volatile* keyword to all variables in our files. Additionally, our attempts at automating the hoisting process were hindered by poor LLVM documentation.

6 Future Work

Future work includes the implementation of the compiler pass to automate the hoisting process and finding more samples that demonstrate quasi-invariant code. Additionally, we intend to continue to improve the benchmark to reduce variability and ensure more accurate results. Once these tasks are complete, we will run the improved benchmark with an increased sample size to develop a clear understanding of any performance boost offered by our optimization.

References

[1] Justice Howley, Jason T. Weeks, and Ian Yelle. LQICM Benchmark - GitHub, 2023. URL: `https://github.com/jweeks2023/LQICM-Benchmark`.

[2] Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk detection. automated technology for verification and analysis. *Automated Technology for Verification and Analysis*, pages 91–108, 2017. `doi:https://doi.org/10.1007/978-3-319-68167-2_7`.

[3] Thomas Rubiano, Neea Rusch, and Assya Sellak. Loop Quasi-Invariant Chunk Motion - GitHub, 2021. URL: `https://github.com/statycc/LQICM_On_C_Toy_Parser`.