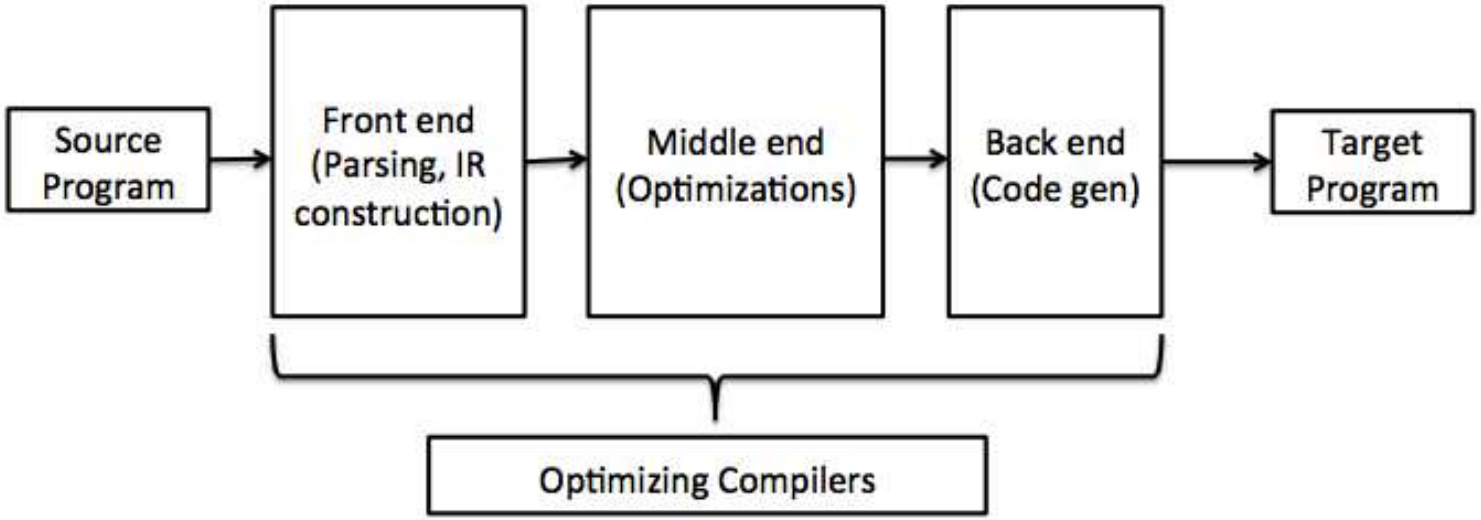
GLOC: A Generic and Automatic Source to Source Compiler for ILOC Programs

Prasanth Chatarasi

Advanced Compiler Construction (COMP 512),
Department of Computer Science
Rice University

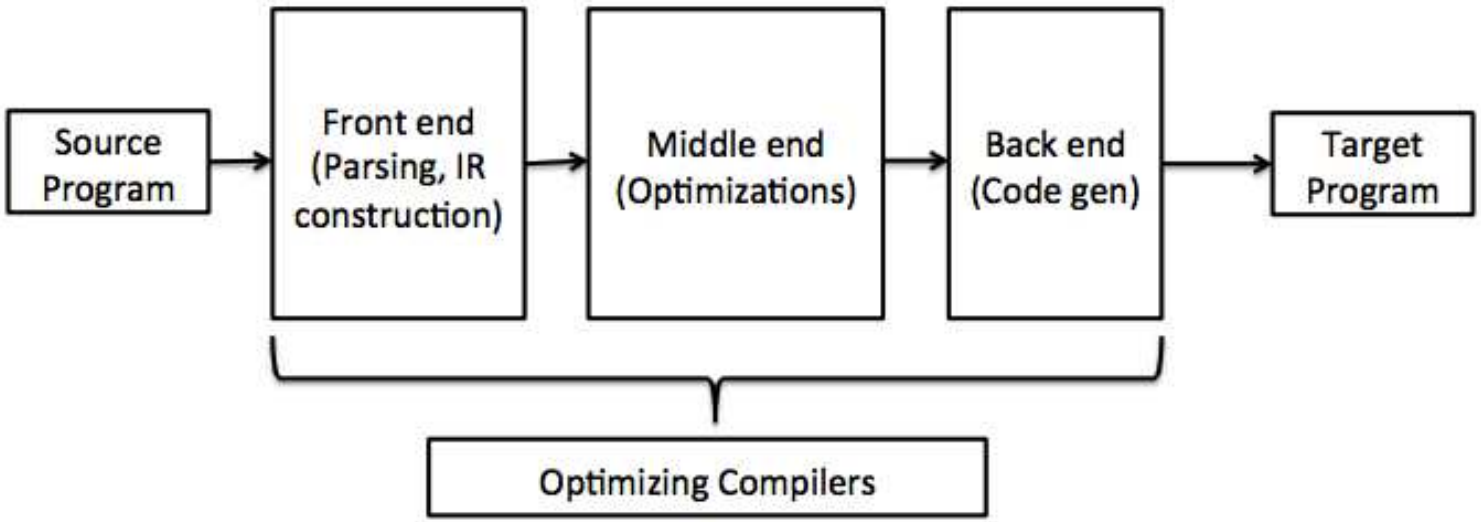
May 7, 2015

Optimizing Compilers



- **Goal:** Build an ILOC optimizer to improve running time

Optimizing Compilers



- Implemented an optimizer and code generator
- Re-used lexical analyzer and parser from ILOC Simulator

Optimizations = Analysis + Transformations

- Analysis
  - Super value numbering
  - Def-Use chains
  - Dominators
- Transformations
  - Common sub-expression elimination
  - Algebraic identities
  - Dead code elimination
  - Conditional propagation
  - Strength reduction
  - Loop Invariant Code motion (LICM)
  - CLEAN optimization

Grouping of Analysis with Transformations

- Super value numbering
  - Common Sub-expression Elimination (CSE)
  - Algebraic identities
- Def-Use chains
  - Dead code elimination
  - Conditional propagation
- Dominators and loop body construction
  - Loop Invariant Code motion (LICM)

- Strength reduction
- CLEAN optimization

Why did we choose ?

- Super value numbering - Easy to start
  - Common Sub-expression Elimination (CSE)
  - Algebraic identities
- Simple Strength reduction - Next Easy step
- Dominators - Expected Huge benefit !!
  - Loop Invariant Code motion (LICM)
- CLEAN optimization - Help LICM
- Def-Use chains - Hard, but beneficial
  - Dead code elimination
  - Conditional propagation

Why did we choose ?

- Super value numbering - Easy to start
  - Common Sub-expression Elimination (CSE)
  - Algebraic identities
- Simple Strength reduction - Next Easy step
- Dominators - Expected Huge benefit !!
  - Loop Invariant Code motion (LICM)
- CLEAN optimization - Help LICM
- Def-Use chains - Hard, but beneficial
  - Dead code elimination
  - Conditional propagation

Why did we choose ?

- Super value numbering - Easy to start
  - Common Sub-expression Elimination (CSE)
  - Algebraic identities
- Simple Strength reduction - Next Easy step
- Dominators - Expected Huge benefit !!
  - Loop Invariant Code motion (LICM)
- CLEAN optimization - Help LICM
- Def-Use chains - Hard, but beneficial
  - Dead code elimination
  - Conditional propagation

Why did we choose ?

- Super value numbering - Easy to start
  - Common Sub-expression Elimination (CSE)
  - Algebraic identities
- Simple Strength reduction - Next Easy step
- Dominators - Expected Huge benefit !!
  - Loop Invariant Code motion (LICM)
- CLEAN optimization - Help LICM
- Def-Use chains - Hard, but beneficial
  - Dead code elimination
  - Conditional propagation

Why did we choose ?

- Super value numbering - Easy to start
  - Common Sub-expression Elimination (CSE)
  - Algebraic identities
- Simple Strength reduction - Next Easy step
- Dominators - Expected Huge benefit !!
  - Loop Invariant Code motion (LICM)
- CLEAN optimization - Help LICM
- Def-Use chains - Hard, but beneficial
  - Dead code elimination
  - Conditional propagation

Code generator

- Generated ILOC code back from control flow graph
- Used Breadth first traversal for code generations

Order of optimizations

- Order of optimizations is crucial to achieve good performance
- Heuristics
  - Algebraic Identities => Constant propagation
    - multI r0, 0 => r1
  - Constant propagation => Dead code elimination (DEAD)
    - Constant propagation enables redundant stores
  - Constant propagation => Strength reduction
    - loadI => r4; multI r8, r4 => r12; Can be replaced by lshiftI
  - Loop invariant code elimination (LICM) => CLEAN
    - Landing pads from LICM can be merged during CLEAN

Phase of optimizations

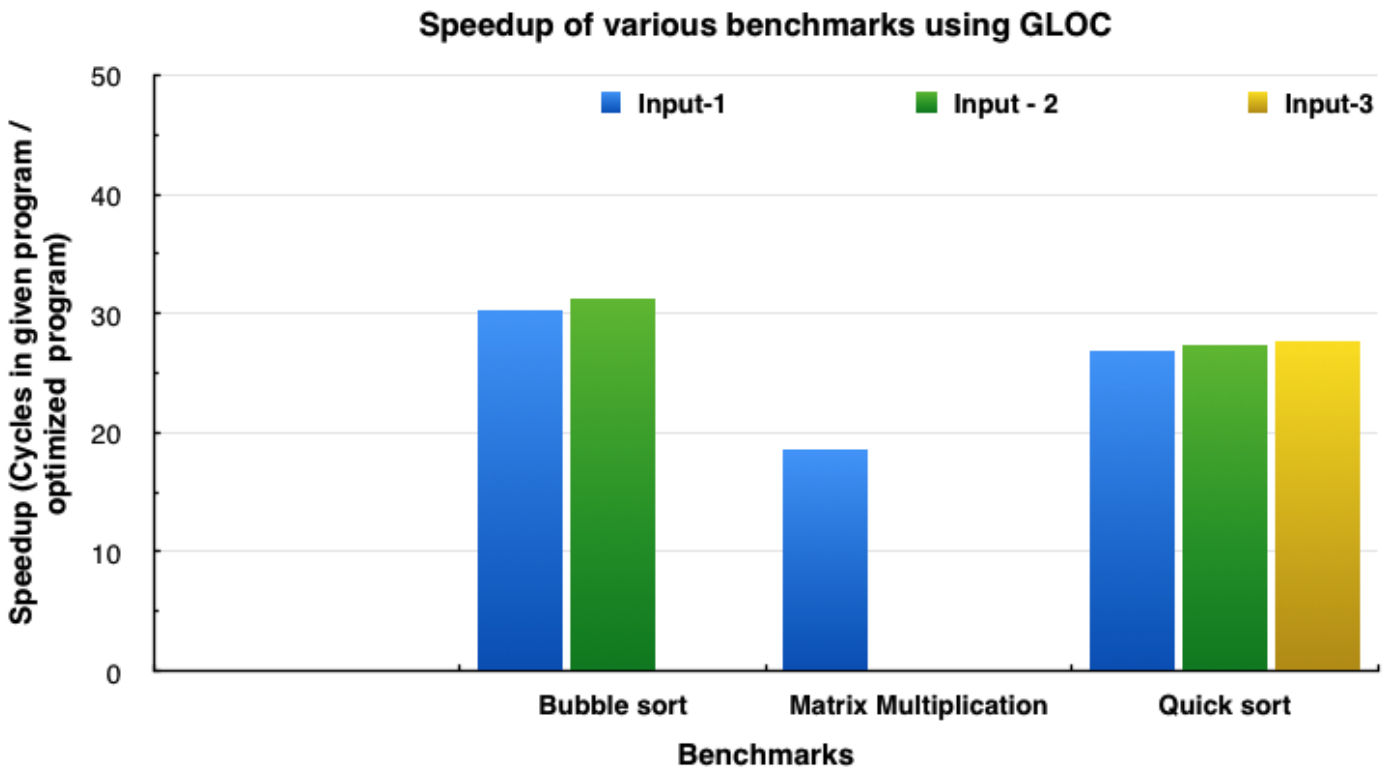
- Phases during optimizer
  - Phase-1
    - Common subexpression elimination
    - Algebraic Identities
    - Constant Propagation
    - Dead code elimination
    - Strength reduction
  - Phase-2
    - Loop Invariant Code Motion (LICM)
    - CLEAN

Algorithm in Optimizer

1. Input: ILOC and Optimization phase order
2. Construct CFG
3. Iterate till fixed point is achieved
    4. Construct Dominators, Def-Use chains,
    5. Optimize based on phase order
6. Generate optimized ILOC code

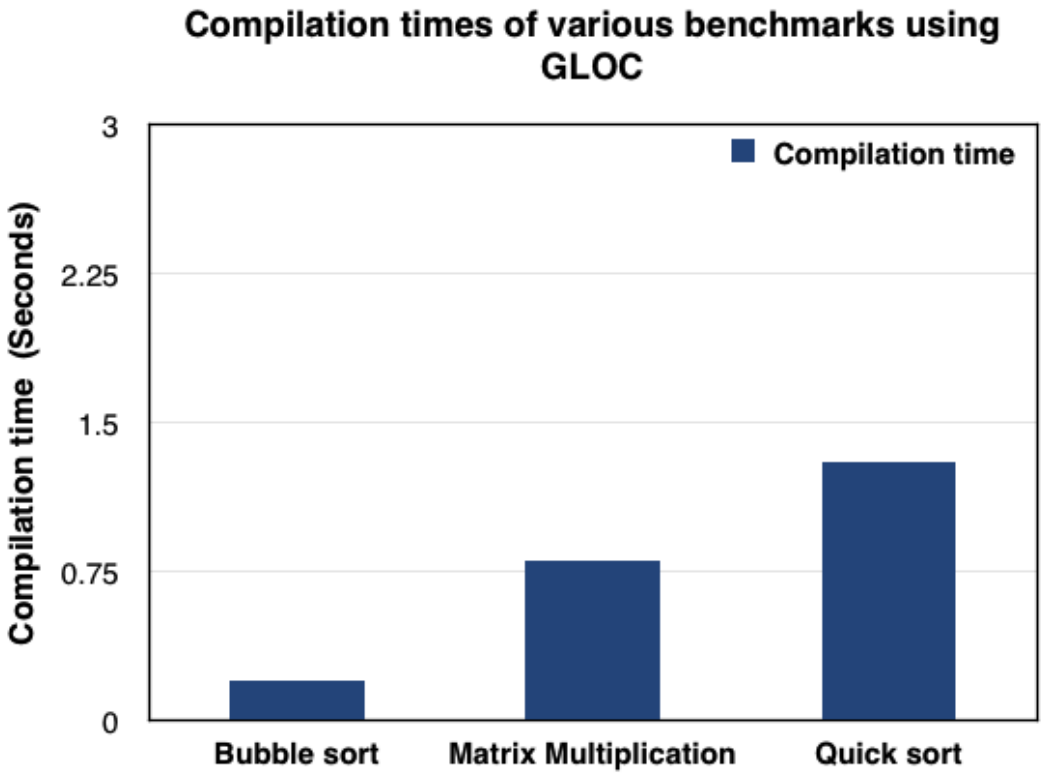
Experiments - Final SpeedUps

- Executed the optimizer on CLEAR machine
- Measured the cycles using simulator on the same CLEAR machine



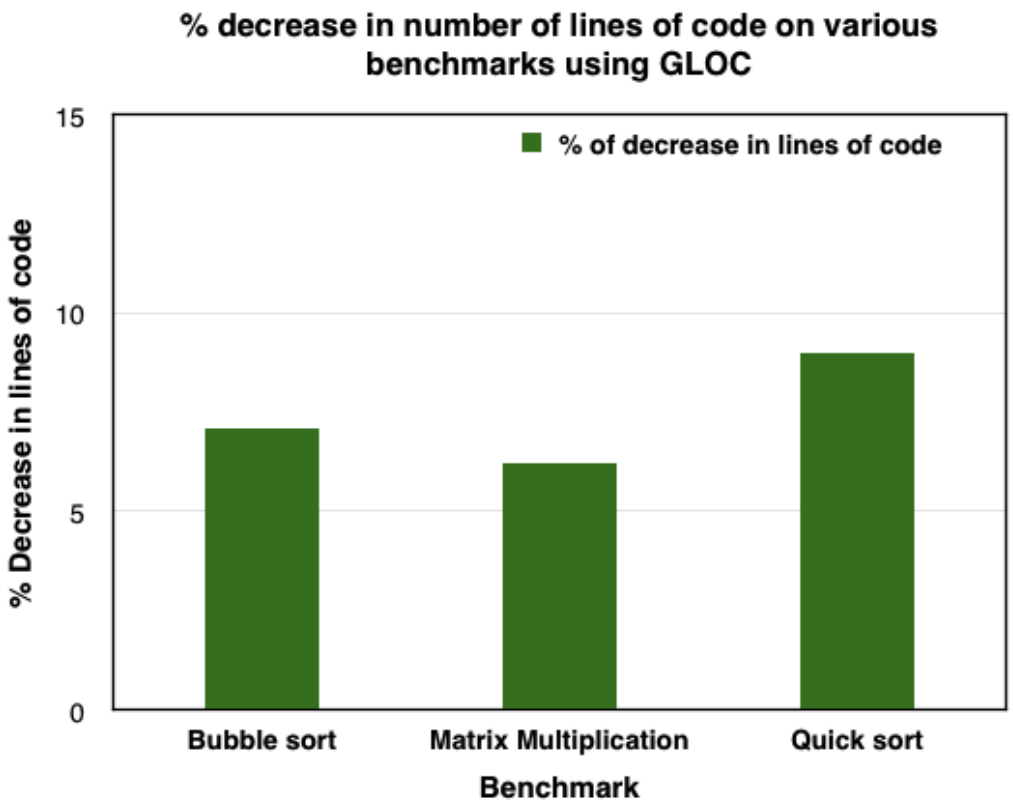
Experiments - Compile times

- Automatic source to source translation time
- Most of the time spent in construction of DEF-USE chains



Experiments - Reducing size of code

- Improvement in reducing # lines of code in program
- DEAD code optimization helps a lot !!



Loop invariance not helping much in the benchmarks !!

```
1 for-loop  (M time) {
2     for-loop  (N times) {
3         addI r10, 4 => r5 // Assume computation is invariant
4     }
5 }
```

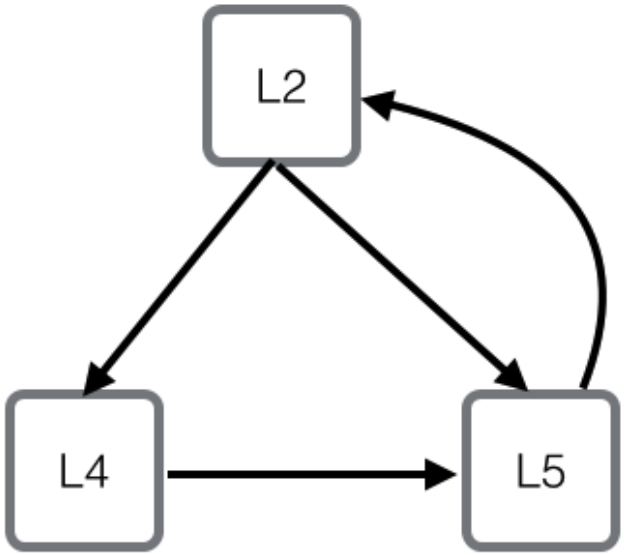
- Assume addI (or) i2i takes 1 cycle
- Before LICM, total cycles = M * N

```
1 for-loop  (M time) {
2     addI r10, 4  => r100
3     for-loop  (N times) {
4         i2i r100 => r5 // Assume computation is invariant
5     }
6 }
```

- After LICM, total cycles = M * N + N
- Most of the invariant instructions takes 1 cycle each
- No benefit in copying to landing pad

Dominators not helping to find all loops

- Assumption: Each back edge is associated to one loop
- Part of bubble sort CFG



- With dominators approach, Loop associated with back edge (L5 -> L2) is L2, L4, L5
- But, there is also loop with body as L2, L5

Conclusions

- GLOC: optimize ILOC programs
- 6K+ lines of code implemented using C++ STL
- An average speedup of 25 % on the given benchmarks
- Constant propagation, Dead code elimination and Strength reduction helped a lot !!
- DEF-USE chains are expensive both in computation and memory

- SSA could have uncovered some more opportunities for optimization

Acknowledgements

- A beautiful experience in understanding and implementing scalar optimizations
- Acknowledgments to Dr. Keith Cooper and the book on "Engineering a Compiler"
- Appreciate the help from Karthik Murthy