Homework 6 - Implementing LLVM Optimization Passes

CSIE 5054 - Advanced Compiler Design
National Taiwan University

Total Points: 8
Due Date: 2025/12/19 23:59
TA Email: llvm@csie.ntu.edu.tw

Contents

1 Announcement

1. The homework contains only 1 problem.
2. In case you encounter technical issues with Homework 6, kindly consult online resources initially, as most problems are likely related to your local environment. If challenges persist, reach out to our TAs following these guidelines:
   - Title your email [AC-HW6][Summary-Of-Your-Issue]. Please note that we will **NOT** receive emails in other formats as we have applied filters to our email system.
   - Provide detailed information about your computer, including the operating system.
   - Outline the methods you attempted previously, the resources you consulted, the steps you followed, and the results of your efforts.
   - Note that ambiguous requests, such as attaching screenshots without proper descriptions, will not be answered. **Such emailing will lower your priority.**
   - For guidance on how to formulate effective technical inquiries, please refer to How To Ask Questions The Smart Way. This resource can help you structure your questions to get quicker, more precise responses from the TAs.

2 Problem 1

In this homework, you will implement an optimization pass in LLVM to improve program performance while maintaining correctness. You will need to carefully design, implement, test, and validate your optimization.

2.1 Requirements

2.1.1 Core Implementation (8 points)

You must implement one optimization pass thoroughly and effectively. Requirements for the pass:

- Design and document your optimization algorithm in detail
- Implement the pass using the LLVM framework
- Create comprehensive test cases demonstrating performance improvements
- Prove the correctness of your optimization rigorously

2.1.2 Available Optimization Passes

The following areas represent optimization opportunities in LLVM. These are guidelines for exploration - you are expected to research, design, and implement your own specific optimization strategies within these domains or propose new ones.

1. **Lazy Code Motion**
    - Implement partial redundancy elimination for arithmetic expressions
    - Compute availability and anticipability through data flow analysis
    - Place computations at optimal points to minimize redundancy
    - Support both local and global code motion
2. **Loop Optimizations**
    - Choose one of the loop-based optimizations
    - Specify optimization goals and constraints
    - Implement dependency analysis for chosen optimization
    - Build cost model for transformation decisions
    - Handle loop bounds and step size analysis
3. **Dead Store Elimination**
    - Identify and remove redundant stores
    - Implement basic alias analysis
    - Handle store-load dependencies
    - Consider memory consistency requirements
4. **Function Inlining**
    - Implement size-based inlining decisions
    - Handle parameter and return value adjustments
    - Consider simple recursive functions
5. **Global Code Placement**
    - Build basic block frequency profile
    - Reorder blocks to improve instruction cache
    - Handle branch alignment
    - Preserve correct control flow
6. **Custom Optimization**
    - Other optimization techniques
    - Theoretical foundations
    - Performance improvement focus

Note on Implementation Before selecting an area, thoroughly understand:

- The theoretical foundations and algorithms
- Implementation challenges and tradeoffs
- Optimization opportunities and limitations
- Potential performance impacts

2.2 Deliverables

2.2.1 Technical Report

Submit a technical report in **IEEE two-column** format containing:

1. **Algorithm Design:**
    - Detailed explanation of the optimization strategy
    - Theoretical analysis of potential improvements
    - Implementation considerations and challenges
    - Code examples illustrating the transformation
2. **Implementation Details:**
    - LLVM pass implementation methodology
    - Key data structures and algorithms used
    - Integration with the LLVM optimization pipeline
    - Handling of edge cases and special conditions
3. **Experimental Evaluation:**
    - Description of test cases and benchmarks
    - Performance measurements and analysis
    - Comparison with unoptimized code
    - Statistical significance of improvements
4. **Correctness Proof:**
    - Formal or informal proof of correctness
    - Invariant preservation
    - Edge case analysis
    - Testing methodology and results

2.3 Testing Requirements

2.3.1 Test Cases and Benchmarks

For each optimization pass:

- Provide a representative set of test cases that demonstrate your optimization's effectiveness
- Include test cases that cover:
    - Simple cases to verify basic functionality
    - Complex scenarios that demonstrate real-world applicability
    - Edge cases that verify robustness

  - Cases where the optimization should and should not be applied
- Select benchmarks that are:
  - Representative of typical use cases
  - Meaningful for your chosen optimization
  - Able to demonstrate significant performance impact

2.3.2 Performance Evaluation

- Focus on relevant metrics for your optimization:
  - Execution time for runtime optimizations
  - Memory usage for space optimizations
  - Code size for size optimizations
  - Compilation time
- Provide clear before/after comparisons
- Include analysis showing the significance of improvements

2.4 Reproducibility Requirements

2.4.1 Experimental Environment

All implementations and experiments must be reproducible on the **ws1** server:

- Document any specific environment variables or configurations needed
- Ensure all dependencies are available on **ws1** or properly documented for installation
- Test your complete submission on **ws1** before final submission

2.4.2 Important Notes

- Your grade will partially depend on our ability to reproduce your results on **ws1**
- All performance measurements must be conducted on **ws1** for fair comparison
- If your implementation requires specific hardware features, verify their availability on **ws1**

2.5 Submission Guidelines

2.5.1 Code Submission

- Submit source code for all optimization passes
- Include build instructions and dependencies
- Provide scripts for running tests
- Document any external libraries or tools used

2.5.2 Report Format

- Use **IEEE two-column** format in LaTeX
- Include clear sections for each optimization pass
- Provide citations for referenced work

2.6 Submission Instructions

1. Prepare your submission files:
   - Technical report in PDF format named `report.pdf`
   - Source code directory containing all implementation files
   - Test cases and evaluation scripts
2. Create a zip file named `[your_student_id].zip` (e.g., `r12345678.zip`)
3. Submit the zip file to NTU COOL before the deadline
4. The zip file structure should be like:

```
r12345678.zip
├── report.pdf
├── src/
│   ├── [your optimization pass files]
│   └── [other source files]
└── tests/
    ├── [test cases]
    └── [evaluation scripts]
```

Note: Submissions with missing components will **NOT** be graded.

2.7 Grading Criteria

2.7.1 Core Implementation (8 points)

- Correctness and robustness (3 points)
  - Maintains program semantics
  - Handles edge cases properly
- Performance improvement (3 points)
  - Significant speedup demonstrated
  - Consistent improvements across test cases
  - Reasonable compilation overhead
- Documentation and testing (2 points)
  - Clear technical documentation
  - Comprehensive test suite
  - Thorough performance analysis

2.7.2 DO

- Implement each optimization pass from scratch
- Write your own analysis and transformation logic
- Use LLVM's core APIs and data structures

2.7.3 DON'T

- Copy or directly use existing LLVM pass implementations
- Call existing optimization passes from your pass
- Submit code from other sources without proper attribution

2.8 Additional Notes

- **Start early** to allow time for testing and refinement
- Consult LLVM documentation for implementation details
- Consider both compile-time and run-time performance
- Document any assumptions or limitations
- Good luck with your homework and final exam :)

3 Plagiarism and Academic Integrity

It is important to adhere to the university's academic integrity policy while completing this assignment. Please keep the following in mind:

1. **Do Not Share Your Code**: All submissions must be your own work. Sharing your code with others or obtaining code from others, including online resources, is considered plagiarism and will be treated as academic misconduct.
2. **Use of External Resources**: You are encouraged to textbooks, lecture notes, and official documentation to assist your understanding. However, directly copying code or solutions from external sources (e.g., GitHub, StackOverflow, or similar) without attribution is not allowed.
3. **Collaboration Guidelines**: You may discuss general concepts and strategies with classmates, but all coding and detailed design decisions must be completed **independently**.
4. **Consequences of Plagiarism**: Plagiarism, cheating, or any form of academic dishonesty will result in a zero on the assignment, and may lead to further disciplinary action as per university regulations.

4 Additional Resources

- LLVM Pass Development Guide: `https://llvm.org/docs/WritingAnLLVMPass.html`
- LLVM Developer Policy: `https://llvm.org/docs/DeveloperPolicy.html`
- Writing an LLVM Pass: 101: `https://llvm.org/devmtg/2019-10/slides/Warzynski-WritingAnLLVMPass.pdf`
- IEEE Two-column Format: `https://www.date-conference.com/format.pdf`