Your names: Oreofe Solarin
Email: ons8@case.edu
Course: CSDS 337 - Compiler Design
Instructor: Dr. Vipin Chaudhary

Problem Set - 5
ID: 123456789
Term: Spring 2024
Due Date: 15th April, 2024

Number of hours delay for this Problem Set:	Put hours here
Cumulative number of hours delay so far:	Put hours here
I discussed this homework with:	Put names here

Problem 1 - 10 points
The C code to compute Fibonacci numbers recursively is shown below. Suppose that the activation record for f includes the following elements in order: (return value, argument n, local s, local t); there will normally be other elements in the activation record as well. The questions below assume that the initial call is $f(5)$.

```
int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}
```

a Show the complete activation tree.

a What does the stack and its activation records look like the first time $f(1)$ is about to return?

Solution:

a

a

Problem 2 - 10 points
In a language that passes parameters by reference, there is a function $f(x; y)$ that does the following:
$x = x + 1; \quad y = y + 2; \quad return \quad x + y;$
If a is assigned the value 3, and then $f(a; a)$ is called, what is returned?

Solution:

Problem 3 - 10 points
The C function f is defined by:

```
int f(int x, *py, **ppz) {
    **ppz += 1; *py += 2; x += 3; return x+*py+**ppz;
}
```

Variable a is a pointer to b ; variable b is a pointer to c , and c is an integer currently with value 4. If we call $f(c; b; a)$, what is returned?

Solution:

Problem 4 - 70 points
Write a C or C++ program to showcase some of the features of Clang and LLVM. You want to write a C program that will best showcase the features of various optimizations. Use time function inside the code to measure program time for performance measurements. Use well-known algorithms (sorting, searching, numerical computation, etc.) and state where they are used if the code is more obscure.

1. For a given architecture, compare the time it takes for different types of compiler optimization. Also use compiler option to minimize code size. Compare the codes (target assembly code) for all cases to isolate the types of optimizations implemented.
2. Show at least three different optimizations in your code that are affected by compiler optimizations. Use Transform (most important), and Utility Passes (https://llvm.org/docs/Passes.html) that are applicable to your example with complete makefile (or command line script) for the passes and appropriate documentation of results. You may also want to refer to clang.llvm.org

Deliverables: A zip file containing

- **File with your code**
- **README text file with directions to run the various programs**
- **Report showing original code and optimized code snippets and the particular optimization used; results (table of performance). Similarily for the bonus portion.**
  - **Full names and Case IDs**
  - **(not required) any special notes about your implementation the grader should be aware of**

0.a Comparison of Compiler Optimizations

0.a.1 Size of Assembly Code

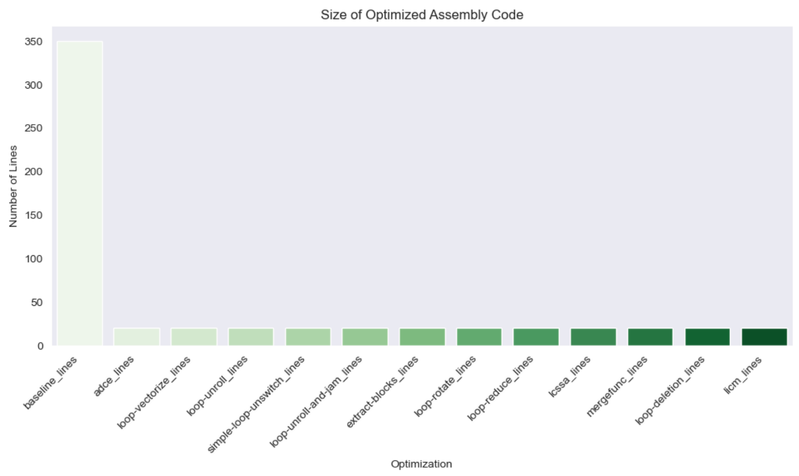


Figure 1: Size

The size of the generated assembly code varied significantly depending on the compiler optimization level and the specific optimizations applied. The mean size of the assembly

code across all cases was approximately 46 lines, with a standard deviation of 91.25 lines. The smallest assembly code size was observed for most cases, with 21 lines, while the largest was 350 lines for the baseline case.

0.a.2 Runtime of Optimization

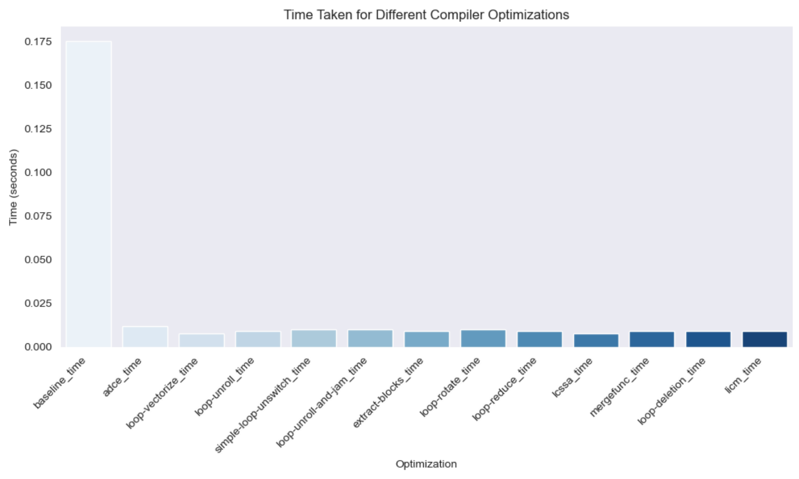


Figure 2: Runtime

The runtime of optimization also showed variation across different optimization levels and specific optimizations. On average, the optimization runtime was approximately 0.022 seconds, with a standard deviation of 0.046 seconds. The fastest optimization runtime was observed for the loop-vectorize optimization, with a minimum runtime of 0.008 seconds. On the other hand, the baseline case had the longest optimization runtime, with a maximum of 0.175 seconds.

0.b Effect of Compiler Optimizations on Code Performance

0.b.1 Transform Passes

1. Aggressive Dead Code Elimination (ADCE): This optimization pass focuses on removing dead code that is unlikely to be executed. It helps in reducing code size and improving runtime performance by eliminating unnecessary computations.

2. Loop Vectorization: Loop vectorization is a crucial optimization technique that transforms scalar loops into SIMD (Single Instruction, Multiple Data) instructions, allowing for parallel execution of loop iterations. This optimization can significantly enhance performance, especially on architectures with SIMD support.

3. Loop Unrolling: Loop unrolling aims to reduce loop overhead by replicating loop bodies multiple times, thereby reducing the number of iterations and improving instruction-level parallelism. This optimization can lead to improved runtime performance, especially for loops with a small number of iterations.

0.b.2 Utility Passes

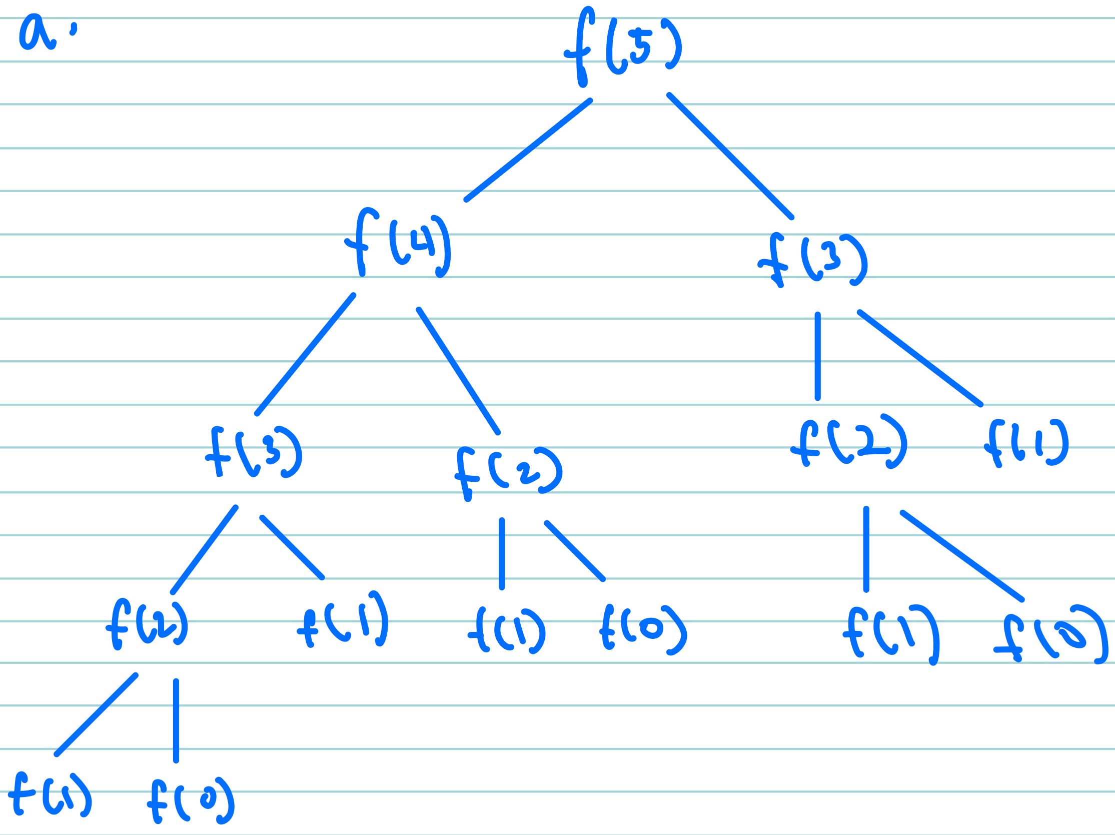
1. LCSSA (Loop-Closed SSA Form): LCSSA is a utility pass that transforms the code to ensure that all values used within a loop are in SSA (Static Single Assignment) form. This transformation simplifies subsequent analyses and optimizations, potentially leading to better code quality and performance.

2. MergeFunc: MergeFunc is a utility pass that merges identical functions across different translation units, reducing code size and improving cache locality. This optimization can have a significant impact on overall code size and runtime performance, especially for large codebases with repetitive function definitions.

3. Loop Deletion: Loop deletion is a utility pass that identifies and removes unnecessary loops from the code. By eliminating redundant loops, this optimization reduces execution time and code size, leading to improved performance and resource utilization.

Overall, the choice of compiler optimizations can have a substantial impact on code performance, with certain optimizations focusing on reducing code size while others aim to improve runtime performance. By carefully selecting and tuning optimization passes, developers can achieve the desired balance between code size and performance for their specific application requirements.

a.



b.

$$\begin{tikzcd}
f(5) \arrow[d, no head] \\
f(4) \arrow[d, no head] \\
f(3) \arrow[d, no head] \\
f(2) \arrow[d, no head] \\
f(1)
\end{tikzcd}$$

f(5), s = f(4), t = f(3)
f(4), s = f(3), t = f(2)
f(3), s = f(2), t = f(1)
f(2), s = f(1), t = f(0)
f(1), 1

$x = x+1 \longrightarrow a = a + 1 \longrightarrow a = 4$

$y = y+2 \longrightarrow a = a + 2 \longrightarrow a = 6$

$x + y \longrightarrow a + a \longrightarrow 6 + 6 \longrightarrow a = 12$

$\boxed{f(a, a) = 12}$

Sentence	x in f()	x out of f()	*py	**ppz
**ppz += 1;	4	5	5	5
*py += 2;	4	7	7	7
x += 3;	7	7	7	7

f(c, b, a) = 21