Mini-Assignment - 3

Vikas Patnala (CS20BTECH11037)

February 12, 2024

Q1

There are different compiler-flags/compiler-options are there a few of them are $-O1, -O2, -O3, -Os, -Oz$. These pass pipelines are provided by both *clang* and *opt* (we will only focus on passes which are added by opt now)

(a)

The generated codes for these optimizations are in the directory `Mini-Asgn-3_CS20BTECH11037/outs/`.

- $-O1$: This is a low level optimization it will just include the optimizations that don't interfere with debugging for example some simple and basic optims such as constant folding, basic inlining and dce.

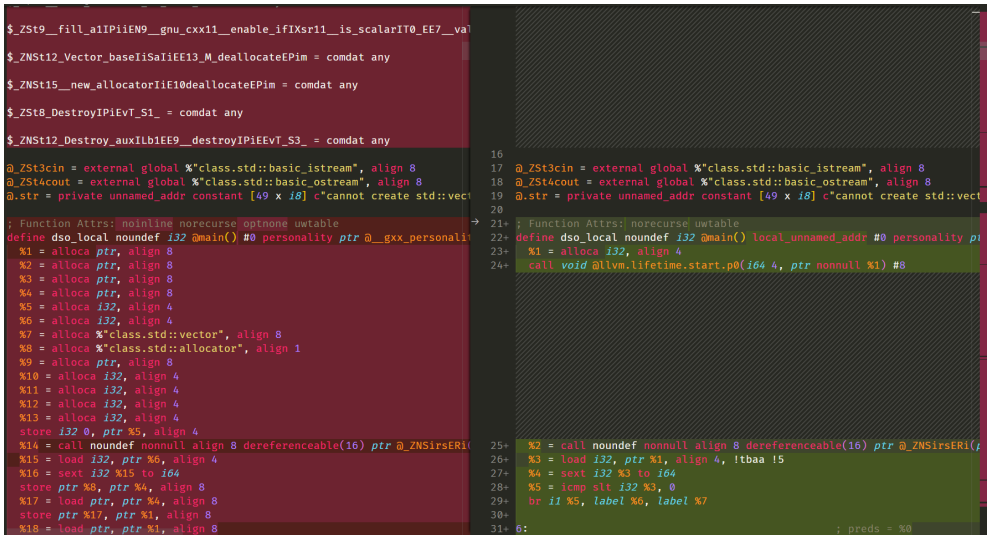

Figure 1: O0-O1 comparision

O0 produces a lot of LLVM IR for CPP functions (such as vector and string related functions) and *O1* remove non-neccessary functions and the code size dropped from 980 lines to 270 lines which is very considerable

- $-O2$: This include moderate level optimization such as advanced inlining, loop unrolling, alias analysis. This is expected as a good default optimization that is used by most programmers

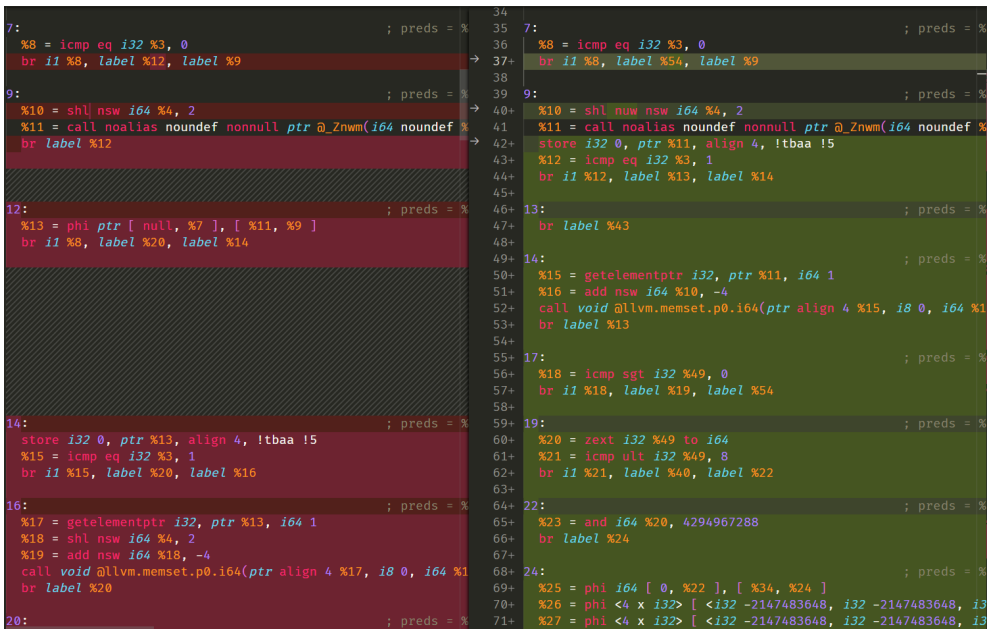

Figure 2: O1-O2 comparision

O2 adds some lines of code which are related to optimizations loop unrolling as the input cpp files contains some loops in it.

- $-O3$: Performs most aggressive optims that may increase the compilation time and code size as well and still may improve performance these includes passes such as loop unswitching, loop interchange, loop distribution and more.

- $-Os$: Performs optims which are similar to the $O2$ but focusses on reducing the code size and includes passes such as dead argument elimination, global dead code elimination. The main goal of this optim is to produce the smaller binaries

- $-Oz$: More aggresive optimization that Os focusing on code size to produce smallest possible binaries and this pipeline includes passes such as merge functions and much more regres optims which involved reduction in code size.

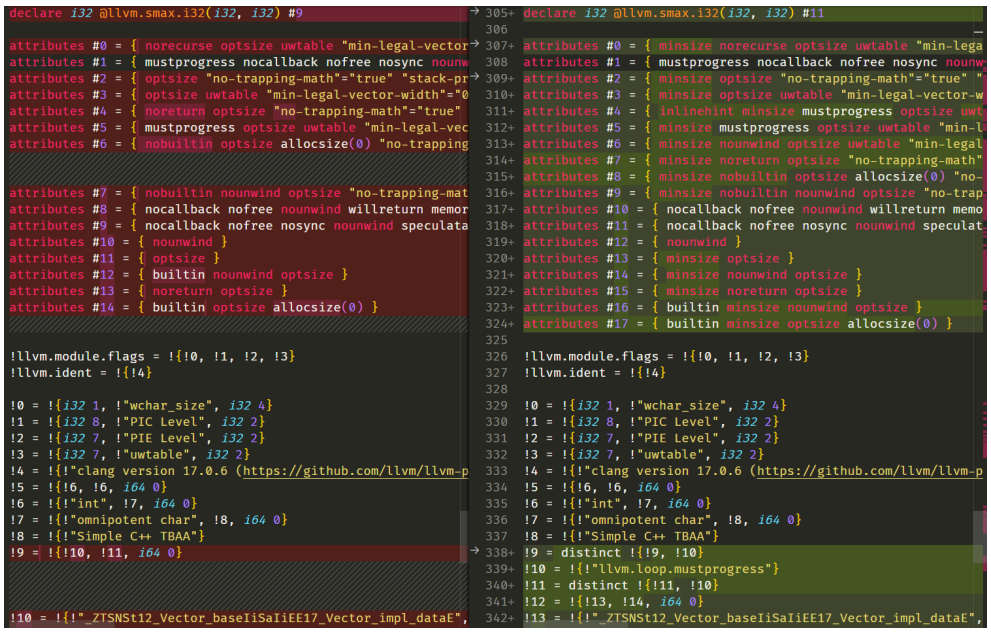

Figure 3: Os-Oz comparision

Os adds the function attribute *optsize* to all the function when we use it where as *Oz* also adds the function attribute *minsize* and using *Oz* incresed the lines of code which I think added for size optimizations.

The exact passes that a particular pass pipeline uses can be found by the command

```
$BUILD_DIR -O3 -mllvm -debug-pass=Structure
```

(b)

There are some specific passes which are added or removed when we are moving from one pass to another and some of them are

- -O0 to -O1 :
  - passes such as constant folding, constant propagation, loop invariant code motion, dce are added here.
  - These added passes improve the code quality, making the code efficient, suitable for debugging and still executes faster.

- -O1 to -O2 :
  - passes such as loop optimizations, function inlining and tail call elimination are added here.
  - These passes improve the runtime performance by optimizing cirital sections such as loops and function calls.
- -O2 to -O3 :
  - Most aggressive passes such as loop vectorization, interprocedural optimizations and much more are added.
  - These optimizations are particularly effective for computationally intensive codes and these can result in significant speed but adding them could also increase the cost of compilation time and code size.
- -O3/-O2 to -Os/-Oz :
  - This transition can remove passes such as loop unrolling and function inlining can be remove because using them can increase the code size and these mainly focus on reduction of code size and the passes such as function merging can be added to reduce the code size.
  - This transition is useful for the resource constrained environments where if we don't have sufficient resources such as disk space to accommodate large codes this optimization can help in the large code base by reducing code size significantly resulting in faster load times and improving cache efficiency.

(c)

The impact of different pass sequences in terms of performance and code size is

Performance Impact

- Loop Optimizations : passes such as loop unrolling, loop vectorization, loop fusion uses advantage of instruction level parallelism and aims to optimize the loops for better performance at runtime and these can have a large impact when there is a large code base typically using more loops.

- Function Inlining : This pass can reduce the overhead caused by calling the small function as it can inline the function when it's size is less and it can significantly improve the perfomance and it also opens the scope to enable other optimizations such as constant propagation and dce.

Code Size Impact

- Dead Code Elimination : This pass eliminates the unreachable(which includes variables, functions, branches) or redundant code and hence reducing the code size.
- Basic Block and Function Merging : Merging identical basic blocks or function merging can reduce code size significantly this optimization is more effective when there is identical code patterns appear in multiple places.

Q2

(a)

The directory structure where the transformation passes are present is

```
llvm/
└── lib/
    └── Transforms/
        ├── AggresiveInstCombine
        ├── Hello
        ├── InstCombine
        ├── Instrumentation
        ├── IPO
        ├── Scalar
        ├── Utils
        └── Vectorize
```

If we want to write our custom passes we will create a new directory for our pass.

(b)

The passes are registered using the *RegisterPass* template class which takes the pass struct as argument and the pass name and description as parameters.

```
static RegisterPass<NewPass> X("new_pass", "My first pass",
                               false /* Only looks at CFG */,
                               false /* Analysis Pass */);
```

here the *new_pass* is our new pass name. we can also register the pass by using *RegisterStandardPasses*.

(c)

The pass scheduling is a process of determining the order, dependencies of applying optimizations passes to the input code. The *O3* optimization pipeline is the most aggressive level of optimization in LLVM which enables many of the passes aiming the performance and code size improvment as we seen in the previous question.

The Passes are ordered in the sequence that maximized the optimization effect while minimizing the overhead and the specific ordering of passes is determined experimentally. Some passes have dependencies between them i.e. requiring a certain pass to be executed before others and this should be ensured by the *PassManager*.

These pipeline optimizations can happen iteratively(can involves multiple iterations of passes to optimize the code further). This pipeline scheduling also involves the architecture and adaptive optimizations which is the optimizations can be turned on dynamically.

The passes are added to the *O1, O2, O3* pipelines in the file `llvm/lib/Passes/PassBuilderPipelines.cpp` and the level of optimization is there is `llvm/lib/Passes/OptimizationLevel.cpp` file

(d)

Implementation part of the Pass Dead Code Elimination(dce) This pass will do both dead instruction elimination and dead code elimination. Dead Inst Elimination performs a single pass over the function removing instructions

that are obviously dead. Dead Code Elimination is similar, but it rechecks instructions that were used by removed instructions to see it they are newly dead.

The **Scope** of the pass is *Function level* only i.e. as the pass is inherited from the *FunctionPass* it only sees the Function level and below (basic block and instruction) and cann't be able to do anything Module level

- The DCE.cpp files has implementation of two passes first one is *RedundantDbgInstElimination* it eliminates the reduntant debug instructions in the code and the other one is *DCELegacyPass* which has the implementation of Dead code elimination.
- The DCE uses the *TargetLibraryInfo* pass to know whether the instruction can be removed or not.
- The *runOnFunction* function contains two lines of code one is *skipFunction* and other is calling the *eliminateDeadCode* function
- The first line *skipFunction* which checks that whether the optimization should have to done or are there any optnone flags on the function.
- The function eliminateDeadCode takes Function and TragetLibraryInfo output as a input now eliminateDeadCode uses the *SmallSetVector* of size 16 named worklist (this worklist is used to revisit the dead instruction which are returned by the *DECInstruction* function). Now the *eliminateDeadCode* iterate over each instruction and calls the *DCEInstruction* function(which adds the instruction to the worklist if the instuction is dead and removes it from the IR using the TargetLibraryInfo analysis) and now after returning back to *eliminateDeadCode* the algorithm iterates over the worklist and checks if there is any changes(any dead code is there) then the function will return true else false.