CS738 - Compiler Optimizations

# Assignment 1 - LLVM Architecture

Shruti Sharma

Sept 2020

## 1 Infrastructure

LLVM is a clean and modular compiler infrastructure with a set of reusable libraries and well-defined interfaces. It is implemented in C++. It also supports a variety of front-ends be it C/C++/Fortran. Its target architectures include X86, ARM, SPARC, MIPS etc. Moreover, it is an open source framework that provides lots of tools to compile and optimize the code. These tools are made up of various libraries that are linked together and are provided by LLVM.
LLVM compiler system includes JIT (Just-In-Time) code generation system. It has various APIs and debugging information to simplify code development. It offers support for garbage collection.
**LLVM IR**
LLVM IR is in SSA (Static Single Assignment) form. That means each variable has only one definition in the code. It is strongly typed and has RISC-like instruction set with opcodes.

## 2 Optimizations

LLVM includes an aggressive optimizer, including scalar, profile-driven, and some simple loop optimizations. There are different set of passes at each optimization level. Some of the optimizations provided by it are:

1. **Dead Code Elimination (DCE)** - In this pass of LLVM, global unused variables/inline functions are pruned. This can be done by computing liveness of those symbols by monitoring the dependency graph. Removing basic blocks with no predecessors. Merging basic blocks with their predecessor that have only one successor. Removing basic blocks that contain only unconditional branch.

2. **Virtual Register Allocation** - To map memory segments/addresses to variables. LLVM register allocation is greedy in nature but fast. The number of physical registers depend on target while virtual registers can be infinite. Optimizations like spill code placement and live range splitting are done. Allow code changes. Two types of mapping are available - direct and indirect.

3. **Common Subexpression Elimination (CSE)** - CSE is removal of redundant code written by coders or which pops up during partial optimization of code. To avoid re-computation it is preferable to use precomputed values. Such repetitive computations are eliminated by an early CSE. Transformation of code happens.

# 3 Experiment

Here is a sample C program.

Listing 1: First.c

```
#include<stdio.h>
int main(void)
{
    printf("To the first LLVM IR");
}
```

The corresponding LLVM IR for the above code is

```
; ModuleID = 'first.c'
source_filename = "first.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"
```

```
@.str = private unnamed_addr constant [21 x i8]
 c"To the first LLVM IR\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
   ([21 x i8], [21 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...) #1
```

---

The above code was generated with the command -
***clang first.c -S -emit-llvm -o first.ll***