DOCUMENTATION FOR THE LLVM PROJECT

INTRODUCTION AND PROBLEM STATEMENT

LLVM is the name of a compiler that began as a research project in UIUC and is now maintained at Apple. Its main website is located here. LLVM is open-source software, and has been installed in Glue (Linux) for use in this class project.

The goal of this project is to get familiar with LLVM by doing a type of function cloning which can reduce the execution time of the program. Function cloning refers to the process when a copy (clone) of a function is made, and then the clone is specialized in some way to make it faster in some cases. In the cases the compiler thinks the clone is faster, it is called, otherwise the original function is called.

One of the instructions contributing to the cost of a call is the run-time cost of the return instruction at the end of the callee function. This return instruction pops (loads) the return address from the stack, and then jumps to that address. Since this return jumps to a run-time-specified quantity, it is called an indirect call. (In contrast, direct calls are when a call jumps to a constant, like the constant address of a known function.) Such indirect calls can be slow on architectures that do not have suitable branch prediction hardware for returns.

In this project, assume that the target processor does not have any branch prediction hardware. As a result, the execution of indirect control-transfer instructions is slow, in comparison to direct unconditional control transfer instructions. Our goal in this project is to avoid using return instructions if possible. To this end, we clone the function, and then in the body of cloned function, we replace the return instruction with a direct branch to the next instruction after the call site. For example, consider the following simple C code:

```
main () {
        int a = 10;
        int p;
        p = pow2(a);
        printf("pow2 of A = %d\n", p);
}
int pow2(int x) {
    return x * x;
}
```

The above C code will be conceptually transformed into:

```
int g = 0;

main () {
        int a = 10;
        int p;
        pow2_clone(a);
label:
    p = g;
        printf("pow2 of A = %d\n", p);
}


int pow2(int x) {
    return x * x;
}
```

```
void pow2_clone(int x) {
    g = x * x;
    __asm__(pop);
    goto label;
    return;
}
```

The example above shows how you can create a clone of function pow2() in a compiler. The cloned function (pow2_clone()) should not have any return value. Instead, you need to define a new global variable (g in the above code). Then before each return in the cloned function, assign the return value to g, insert an __asm__ function to perform a pop instruction to remove the (now redundant) return address from the stack, and then insert a goto to a label just after a particular call site of the cloned function. The cloned function is only called from one call site, hence specializing the cloned function to that call site is possible. Finally after the call site that calls the cloned function, an instruction must be added which assigns g to wherever the original function's return value was written to (p in the above example).

In the cloned function above, the combination of the pop instruction followed by a direct branch provides the same functionality as a return instruction. Once the goto is inserted, the return instruction at the end of the cloned function becomes unreachable, and will never be executed. However it needs to be there, because if it is deleted we will get a compilation error from LLVM saying the function is not terminated by a return.

All transformations in LLVM are performed on the intermediate representation (IR) of the code. The C code will be translated by LLVM automatically to internally use single static assignment (SSA) form, which will be discussed later in class. For now, suffice it to say that SSA translates code such that every variable is defined (assigned) only once; thus the name.

Although the code above is the conceptual output, the actual output is a little different. Do not actually generate the __asm__(pop) and goto label instructions; instead actually replace them both by a single call to a function named pop_direct_branch(). The reason is that the transformation must be done in IR level where there is no access to assembly instructions, hence the __asm__(pop) cannot be generated. Instead call pop_direct_branch() which has been provided to you in a separate c file that is included with each program you compile. This function would provide the exact same functionality of pop instruction followed by the goto statement. It finds the label it needs to jump to by locating which cloned function it is being called from, and locating the single call site to it. You do not need to write this pop_direct_branch() function, so its implementation is outside the scope of this function. Instead just use it as described here.

Your goal is to write a *function cloning* pass in LLVM that automatically performs the above transformation on any program provided to LLVM. It should clone every function whose name begins with the letter p (like the function pow2()), and it should clone such functions once for each of its call sites. For example if such a function is called from three call sites, three clones of the original function are produced. You need not delete the original function. (However if you run LLVM with optimizations, it will delete any unused functions, such as the original function after cloning.)

Your pass itself will work on LLVM IR provided as input, and produce modified LLVM IR as output, allowing easy integration into the existing LLVM compiler. To do so, you must become familiar with LLVM. Please follow the steps in the rest of this document to achieve this.

COLLABORATION POLICY

This project should be done by students in groups of two each. That arrangement is ideal since one member of a team might get stuck by themselves, and collaboration can help solve this problem, and encourages learning. When doing the project in groups, both students are expected to be knowledgeable

with all aspects of the project.

THINGS TO SUBMIT

The project should be uploaded electronically on the class website (the elms/canvas site). The deadline for submission is 11:59pm on Thursday, May 4, 2017. Later, the TA will email the class mailing list with the exact instructions on how and what to submit. However the submission will include at least the following:

(i) The source code for your pass.

(ii) The output IR after running your pass on each of the examples in the llvm/test_codes directory.

(iii) Any modified Makefile you use to compile your pass.

MACHINE TO USE LLVM

Although most computers on the university glue system use the Solaris OS, a few use the Linux OS. Since LLVM is compatible with Linux but not Solaris, *you must use a Linux glue machine for this project, as otherwise the software will not work*. The following glue Linux machine has been provided to you for this project:

compute.ece.umd.edu

Please log into the above machine using ssh or some other secure remote login method. You can use your existing UMD glue account name and directory ID to login. If you do not have a glue account, please obtain one immediately. If you are having problems accessing your glue account, or you don't have a glue account, the UMD IT help desk recommended that all students call them to get help. Please call them at:

Phone Support: 301.405.1500
Hours: Mon - Fri 8:00 a.m. - 6:00 p.m.

BASICS TO USE LLVM

A brief tutorial to use llvm is described here. LLVM is already installed on the system, so don't try reinstalling it. Some of the important steps and additional steps are listed below.

First remember to add the llvm and llvm-gcc install directories to your environment variable PATH using

```
setenv PATH /afs/glue/class/old/enee/759c/llvm/llvm-3.4-install/opt/bin/:$PATH
```

Next, copy over the example files in /afs/glue/class/old/enee/759c/llvm/test_codes/ to a local directory in your home directory. Then the output bytecode file (e.g., example1.bc) can be obtained from source code written in C (e.g., example1.c) using:

```
clang -c -emit-llvm example1.c -o example1.bc
```

Bytecode is the Intermediate Representation of the LLVM compiler. Please test your programs on the test codes placed in the test_codes folder.

This bytecode can be converted to assembly code (e.g., example1.s) using:

```
llc < input bytecode > -o < output assembly file >
```

This < output assembly file > can be assembled to an executable (e.g., example1) using gcc:

```
gcc  < assembly file > -o < output executable >
```

We can use llvm-dis to disassemble bytecode to a human readable IR file (e.g., example1.ll)

```
llvm-dis < input bytecode > -o < output human readable IR file >
```

This output file is human readable. The instructions in it will be explained in the next section.

This human readable IR can be assembled to bytecode using llvm-as as follows

```
llvm-as < input human readable IR file > -o < output bytecode >
```

INTRODUCTION TO LLVM_IR

The human readable IR that was obtained in the last section can be opened using any editor of your choice.

The first thing that will be helpful in understanding the human readable LLVM would be to understand few of the instructions present in the intermediate representation of LLVM. The language reference manual is present here. A few instructions that would be good to know are:

- Terminator Instructions
- Binary Operations
- Memory Access and Addressing Operations
- Integer Compare Instruction The types present in llvm are described here.

IMPLEMENTATION IN LLVM

You can follow the following procedure to implement the function cloning pass:

0. Define a new global variable and initialize it with an arbitrary value.
1. At every call site in the program, check if the cloning needs to be done.
2. If so, create a new function by cloning the original function definition. If the original function returns a value, change the type of cloned function to void.
3. Then you need to add two IR instructions before the return instruction. First instruction should store the return value, if any, to the global variable defined in step 0. Second instruction is a call to the function *pop_direct_branch*. Finally, you should modify the return instruction to return void.
4. Then, if the original function has a return value, you need to add one IR instruction after the call site which should load the return value from the global variable to the intended variable.
5. Replace the function at the call site with the newly cloned function. Pay attention to the fact that the cloned function is void and does not have any return value.

Hints on how to write the above passes are in the “Steps involved” section below.

INFRASTRUCTURE TO WRITE YOUR OWN PASS

A sample project has been created for you to write your own pass. A good initial step before writing your own pass is to compile the pass in the sample folder and check if it works. This will ensure that you do not have any environment bugs when you write your own pass. The example pass present in the lib folder

called Hello prints out the name of every function present in the bytecode. Follow the steps below to get it running on your machine.

(1) First copy the sample directory present at /afs/glue/class/old/enee/759c/llvm/sample to your local directory, say ~/xyz/sample. You can use "cp -R" to copy a directory and its contents.

(2) Create two other directories at the same level (i) obj directory, say ~/xyz/obj and (ii) an install directory called opt at ~/xyz/opt

(3) Configure the build system in the obj directory as follows:

~/xyz/obj$ ../sample/configure --with-llvmsrc=/afs/glue/class/old/enee/759c/llvm/llvm-3.4.src --with-llvmobj=/afs/glue/class/old/enee/759c/llvm/llvm-3.4-install/obj --prefix=/homes/your-user-name/xyz/opt

(4) You can build it and test it out as follows:

~/xyz/obj$ make install

```
llvm[0]: Installing include files
make[1]: Entering directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib'
make[2]: Entering directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib/Hello'
llvm[2]: Compiling Hello.cpp for Release+Asserts build (PIC)
llvm[2]: Linking Release+Asserts Shared Library libHello.so
llvm[2]: Building Release+Asserts Archive Library libHello.a
llvm[2]: Installing Release+Asserts Shared Library /homes/neroam/xyz/opt/lib/libHello.so
llvm[2]: Installing Release+Asserts Archive Library /homes/neroam/xyz/opt/lib/libHello.a
make[2]: Leaving directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib/Hello'
make[1]: Leaving directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib'
```

[There would be many more lines ... only few reproduced here]

(5) Set the PATH variable to also have the llvm installation as follows:

~/xyz/obj: setenv PATH /afs/glue/class/old/enee/759c/llvm/llvm-3.4-install/opt/bin/:$PATH

Check that the installation is correct as follows

```
~/xyz/obj: opt --version
LLVM (http://llvm.org/):
  LLVM version 3.4
  Optimized build.
  Built Mar 17 2014 (12:45:22).
  Default target: x86_64-unknown-linux-gnu
  Host CPU: penryn
```

(6) You can run your pass as follows:

~/xyz/obj: opt -load ../opt/lib/libHello.so -hello /afs/glue/class/old/enee/759c/llvm/test_codes/example1.bc -o example.1hello.bc

The above command will optimize the example1.bc bytecode with the “-hello” optimization and produce as output optimized bytecode in example1.hello.bc. This optimization is present in the libHello.so library.

[The output should look like]

Hello: main

It prints out the names of all functions present in this module.

Later when you write your own optimization, write it in the sample/lib directory. The compile and install it using steps (3) and (4) above. This should produce a new library with a .so extension. Then use the above command with the library name that you generated in place of libHello.so, and the name of your optimization in place of "-hello". This will apply your transformation to the bytecode file specified.

WRITING YOUR OWN PASS

In LLVM the transformations and optimizations are written as **Passes**. The documentation about an LLVM Pass and method to write it is here . The main sections that will be of interest to this project are

- Introduction
- Quick Start
- ModulePass
- FunctionPass
- AnalysisRequired

The example code present in the sample folder has an example code and Makefiles. The documentation in this section and the example code should help you in your project. You can add a new pass to the sample code by creating a new directory in the lib folder and adding the name of this new folder to the Makefile in the lib folder.

The programmer's manual here gives an introduction to some important and useful API's in LLVM. A few sections of the programmer's manual that will be interest to this project are:

- Common Operations
- Core LLVM classes

These sections show methods to add and delete instructions in the input bytecode and also present methods for basic inspection and traversal routines.

The doxygen for the LLVM project is present at here.

STEPS INVOLVED

Below are helpful suggestions on how to implement some of the tasks in the project. You can choose to use them, but full credit will be given for other implementations, provided you write them yourself without assistance or copying code from elsewhere. Of course, you are allowed to look at code already in the LLVM compiler as examples.

0. Create a new global variable

To know the steps needed to define a global variable, create a c file which has a global variable. Then compile it to byte code and after that use the following tool to get the cpp file that would generate the

input byte code. Use the section of generated cpp code named "Global Variable Declarations" to find out the code that you need to include in your pass.

llc -march=cpp < input bytecode > -o < output cpp file >

Note: For the sake of simplicity, you can assume that every function starting with 'p' is either void or returns an int(i32) value. As a result, you only need to define one integer global variable, with the type int(i32).

1. Iterate over all instructions

Iterate over all instructions in the original code and for each instruction cast it to a CallInst, if this succeeds then you have an instruction that is of interest. The *CallInst* class is defined in http://llvm.org/doxygen/Instructions_8h_source.html. Some of functions of interest in this class are *getCalledFunction()* -- This will tell you which function is being called.

2. Check whether cloning needs to be done

Check the name of the function that you found in the previous step. If the name of the function starts with “p”, then it needs to be cloned.

3. Clone a function

It is present in the file
http://llvm.org/docs/doxygen/html/CloneFunction_8cpp_source.html#l00223
Make sure to change the cloned function to not to have any return value (void).

4. Storing the return value in the global variable

If the function returns any value, you need to create a new store instruction by calling the appropriate constructor from StoreInst class. The store instruction destination is the global variable. The source is the value which is returned by the ReturnInst instruction.

5. Adding the call to pop_direct_branch()

You need to create a new CallInst using appropriate constructor to call the function pop_direct_branch. This function is already part of this module. You can use “llc -march” command to see how a call instruction could be created and added to existing instructions. ‘

Note: The function pop_direct_branch() is already included as part of the program.
The implementation can be found in here: "/afs/glue/class/old/enee/759c/llvm/test_codes/pop_direct_branch.c"
Please ignore implementation details. You only need to call this function by the CallInst instruction that you add in this step.

6. Modify the return instruction

Since the cloned function is a void function, the ReturnInst at the end should be void and not return any value. This is a dead return instruction which would never be executed.

7. Loading the return value from global variable

If the function returns any value, a load instruction should be inserted after the callsite. This instruction should copy the return value from the global variable to the variable which was previously on the left-hand side of the CallInst.

8. Replace a call to a function to a call to the clone function.

There are two ways to accomplish this.
(a) You need to create a new CallInst calling your cloned function and new arguments. Use one of the constructors of the CallInst which will also let you specify a point to insert it. If you do this you also need to remove the original instruction from its parent. This is present in the *Instruction* class.

http://llvm.org/doxygen/Instruction_8h_source.html

(b) There is a method called *setCalledFunction* in the CallInst class. Use this to replace the old function with the new one.

Make sure to modify this CallInst not to have any return value.

Some useful links in general

(a) A list of all classes present in llvm -- http://llvm.org/doxygen/classes.html

(b) A list of all files present in llvm -- http://llvm.org/doxygen/files.html

(c) Link to llvm's programmer manual -- http://llvm.org/releases/3.4/docs/ProgrammersManual.html

Some subsections of interest here may be *cast templates, iterating over basic blocks in a function, iterating over instructions in a Basicblock and so on. Iterating over def-use chains, Creating and inserting new instructions, deleting instructions, replacing an instruction with another value, Creating types, the Value class, the Constant class, the Function class and so on.*

MORE USEFUL LINKS

- Writing an LLVM pass – Refer to the page titled iterating through Functions, Blocks, Instructions.