Function Inlining Heuristics

ECE 466/566 Spring 2023

Due: Tuesday, April 18, 11:59 pm

You are encouraged to comment directly on this document!

Objectives

- Implement a function inlining heuristic.
- ECE 566: Design your own heuristic and report the results.

Specification

In this project, you will implement a function inlining heuristic. A function that performs inlining is provided for you; **your code only decides on which call instructions to perform inlining**.

C	`#include “inline.h”`
	
	`LLVMValueRef Call = ....; // a call instruction`
	`InlineFunction(Call); // inline the called function at the Call`

C++	`#include "llvm/Transforms/Utils/Cloning.h"`

	`CallInst *CI = ....; // a call instruction`
	`InlineFunctionInfo IFI;`
	`InlineFunction(*CI, IFI);`

Basic Inlining Support

All 466 and 566 students must implement the basic inlining support. It's configured by three knobs:

Flag	Description
`-inline-function-size-limit=N`	Only consider call sites that call function, F, with `numInstructions(F) < N.` If not given on the command line, assume a very large value (e.g. no constraint).
`-inline-growth-factor=M`	Do not allow the program to grow more than M times larger than it begins. M is an integer. The growth factor is determined based on the number of instructions before inlining and the number after inlining. If not given on the command line, assume a large value (e.g. no bigger than 10x) that still allows the compilation to finish.
`-inline-require-const-arg`	Only inline functions that include a constant argument. If not given on the command line, do not require a constant.

These flags can be combined in any combination. Your code must obey all specified constraints. You should use a greedy approach to inlining. If you find a call site and it meets all constraints, then inline the function. Note, only functions defined within a module can be inlined. So you also need to verify that the function has basic blocks to be sure that it's not just a declaration.

You will want to use a worklist-based approach. First, add all suitable call instructions to the worklist, and then remove them and inline them as long as all constraints are met.

Experiment with these flags and study the impact on number of instructions and performance. Also, feel free to add other statistics to help you understand what is going on. **But, do not change the name of the stats in the starter code.**

Advanced Inlining (566 only)

Using what you learn from the basic inlining support, design a heuristic that provides an average overall performance improvement on wolfbench.

Flag	Description
`-inline-heuristic`	When this flag is specified, only apply your own heuristic, and ignore the other flags. Your heuristic may not use input from any other flags because the testing scripts will not know about them. Only -inline-heuristic will be given to trigger your new approach. This must work in the same submission with the other basic support described above. All of the features must be available in a single binary.

Infrastructure

Save pending work

You will need to update your repository to get the latest version of code. Then rebuild everything. Either commit or stash any pending work:

1. `git commit -a -m”some changes I made blah blah blah”`
   Or...
2. `git stash`

Case 1: Basic cloned repository

Pull the latest copy and then merge conflicts:

```
      git pull
```

Case 2: Remote tracking branch (Advanced option from Project 0)

If you have the advanced setup from Project 0 with a remote tracking branch, do this:

```
      git checkout work # replace work with your branch of choice
```

Replay your commits on top of the class repo:

```
      git pull --rebase ece566 main
```

Code Development and Testing Setup

You may implement your project in either C or C++. Use one of these files to implement and test your code:

	Stub implementation to edit	Binary to use for testing
C++	projects/p3/C++/p3.cpp	build/p3
C	projects/p3/C/inline.c	build/p3

This is only a starting point! A stub version of the entry point to the optimization (DoInlining~~CommonSubexpressionElimination~~) function has already been implemented for you in either C or C++ starter file.

You may not change the name of the `DoInlining` function. You may add or modify other content of the files except for: (1) how the statistics are dumped and (2) the specific optimizations enabled to run before and after inlining. These will be used by the grader script and must be retained as is. However, you may add more statistics if it's helpful.

CLion Setup

Import the project into CLion and configure it to use the LLVM toolchain you set up in Project 0. Repeat the steps in that document if you don't remember all of the steps. Then, build your code as usual for CLion.

Manual Setup for VCL or in the container

Make a build directory in the source folder, configure it using cmake, and then build and test your code. Note, if you are working inside the container, you will need to go to the host shared volume under /ece566. Please adjust the PREFIX path appropriately for your use case.

C	`cd PREFIX/ncstate_ece566_spring2023/projects/p3/C`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`

C++	`cd PREFIX/ncstate_ece566_spring2023/projects/p3/C++`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`

Testing

Test your code using wolfbench as you did in Project 0.

1. Make a directory for testing. In the `p3/C` or `p3/C++` is fine, but you can put it anywhere you want:
   a. `mkdir p3-test`
   b. `cd p3-test`

2. Assuming that `p3-test` is in your C or C++ directory:
```
path/to/wolfbench/configure --enable-customtool=/path/to/p3
```

4. Compile the benchmarks to test your code:
```
make all test
```
5. To verify that the test cases continue to function properly after optimization, use an extra make flag that will compare the outputs to the correct outputs:

```
make test compare
```

6. If you encounter a bug, you can run your tool in a debugger this way:

```
1. make clean
2. make DEBUG=1
```

   This will launch lldb using your tool on the first un-compiled input file, which is likely the one failing. You can set breakpoints directly in your source files. This option will only work on the VCL image for debugging with lldb installed.
   For more info on how to use lldb.

Analyze Your Results

As part of this project, you will need to submit a report that describes how well your pass works. To accomplish this, you will need to:

1. Compile with multiple settings. For example, compile once with your pass, and once without it. You may also choose to apply optimizations or not.
2. Collect how many instructions are added by your pass in various conditions.
3. Measure the execution time of benchmarks with and without your pass.

The p3 binary already provides several helpful flags:

Flags	Meaning
`-no-inline`	Don't perform inlining at all
`-no-preopt`	Don't apply optimizations before inlining (mem2reg, CSE, SCCP, ADCE)
`-no-postopt`	Don't apply optimizations after inlining (mem2reg, CSE, SCCP, ADCE)

To help you, I'm providing a helper makefile that runs many configurations and then calls the scripts.

Makefile.opts

```
# SET THESE PATHS TO MATCH YOUR INSTALLATION
FULLSTATS=../../../../../wolfbench/fullstats.py
TIMING=../../../../../wolfbench/timing.py

all:
	make clean
	make EXTRA_SUFFIX=.None CUSTOMFLAGS="-no-inline -no-preopt -no-postopt"  test
	make EXTRA_SUFFIX=.O CUSTOMFLAGS="-no-inline" test
	make EXTRA_SUFFIX=.I CUSTOMFLAGS="-no-preopt -no-postopt"  test
	make EXTRA_SUFFIX=.IO test
	make EXTRA_SUFFIX=.IOA CUSTOMFLAGS="-inline-require-const-arg" test
	make EXTRA_SUFFIX=.IO10 CUSTOMFLAGS="-inline-function-size-limit=10" test
	make EXTRA_SUFFIX=.IO50 CUSTOMFLAGS="-inline-function-size-limit=50" test
	make EXTRA_SUFFIX=.IO100 CUSTOMFLAGS="-inline-function-size-limit=100" test
	make EXTRA_SUFFIX=.IOG2 CUSTOMFLAGS="-inline-growth-factor=2" test
	make EXTRA_SUFFIX=.IOG4 CUSTOMFLAGS="-inline-growth-factor=4" test
```

```
stats:
	python3 $(FULLSTATS) Instructions
	python3 $(FULLSTATS) nInstrPreInline
	python3 $(FULLSTATS) nInstrAfterInline
	python3 $(FULLSTATS) Inlined
	python3 $(FULLSTATS) ConstArg
	python3 $(FULLSTATS) SizeReq
	python3 $(TIMING)
```

Within p3-tests, you can invoke the makefile like so:

```
make -f path/to/Makefile.opts all
make -f path/to/Makefile.opts stats
```

Make sure your path to timing.py and fullstats.py are set correctly within the Makefile. You can hardcode them to an absolute path on your system. The default paths assume that you have p3-test under `p3/C++/build/p3-test`.

Getting Help

History shows that my specs are sometimes incomplete, incorrect, or confusing. Therefore, please start early. If you run into problems, please post a question to the Google forum, but remember do not post your code unless it's a question about specific starter code that I gave you.

Grading

ECE 566 students must work individually, but ECE 466 students are allowed and *encouraged* to form groups of two. Only one student needs to submit in ECE 466, but both names must appear in the submission document and in the headers of all submitted files.

Questions

You must submit a brief report along with your code with the following details:

1. Describe your implementation for basic inlining support in pseudocode. In particular, I want to understand your logic that selects calls to inline and applies the rules.

2. Collect data per benchmark that compares the number of instructions before and after inlining and the execution time. Show the flag combinations in Makefile.opts and include other interesting combinations. Include this data in either a graph or table in your report. Anayze and explain the trends you observe in this data and what you learned from it.

3. [566 only] Describe your heuristic. Explain how you came up with your heuristic and present data for how it performs. You may want to compare it to the analysis done for Q2 or collect additional data to justify your choices.

Uploading to GradeScope: Upload your p3.cpp or inline.c file. If you added other files or made changes to other files, please submit those, too. If you modified the CMakeLists.txt, upload that as well. Include the report as a pdf along with your code. It will be ignored by the autograder script but made available to the instructor.

We will test your code using the test cases provided in wolfbench and with some secret cases we did not provide.

The assignment is out of 100 points total. If you make no attempt and submit the provided code without meaningful changes (i.e. white space and comments do not count), you earn a maximum of 10 points total. Otherwise, assigned according to the rubric.

Rubric

ECE 566

10 points: Compiles properly with no warnings or errors
5 points: Code is well commented and written in a professional coding style
40 points: Transformations are applied according to the flags and run to completion with no testing or comparison failures.
20 points: Heuristic flag is applied and runs to completion with no testing or comparison failures.
25 points: Report with answers to the questions.

ECE 466

10 points: Compiles properly with no warnings or errors
5 points: Code is well commented and written in a professional coding style
60 points: Transformations are applied according to the flags and run to completion with no testing or comparison failures.
25 points: Report with answers to the questions