CSE 231 Project 1

Michael Barrow
mbarrow@eng.ucsd.edu
University of California, San Diego

Jules Testard
jtestard@eng.ucsd.edu
University of California, San Diego

Konstantinos Zarifis
zarifis@eng.ucsd.edu
University of California, San Diego

Abstract

This report describes the implementation and experimental results of custom LLVM passes applied to supplied benchmark programs. The high level concept of each pass is discussed, followed by key implementation details. Finally we offer conjecture on experimental application of these passes.

1 Introduction

LLVM is a collection of modular compiler tools. The projects original goal was a flexible compilation strategy for arbitrary programming languages able to perform both static and dynamic compilation.
This language flexibility is mainly achieved with a common intermediate program representation during the compilation known as LLVM byte code. Source language and target machine independent optimisations can be made to the LLVM byte code, which is the 231 course project subject of interest.
Our prescribed source language is C++ and our target architecture is x86. The project goal is to profile a set of LLVM compiled C++ benchmarks programs using the **pass** feature of the **opt** LLVM module. Three pass functionalities are required:

- Collecting static instruction counts
- Collecting dynamic instruction counts
- Profiling branch bias

The required depth of understanding and proficiency in LLVM module code increases incrementally with each of these three functionalities. Therefore the report is split into three mini-reports to focus on the self contained lessons learned from implementing them. Mini-reports have the following sub sections:

- **Pass Algorithm Description:** A high level description of an *algorithm* used to implement functionality.
- **Physical Implementation summary:** Details on key LLVM concepts and api's used to implement the functionality *algorithm* as an *opt pass*.
- **Benchmark Analysis:** Conjecture based running the *opt pass*.

2 Static instruction count

Problem statement: *Write a pass that counts the number of static instructions in a program.*
Problem instance: *Output for each instruction the number of times it appears in the program.*

2.1 Pass Algorithm description

Because the functionality is a static analysis, it is sufficient to run a pass on the compiled

code of each benchmark. We store instruction op codes and their corresponding count in a C++ map structure. Each time an instruction is found in the source code, it is either added to an existing map entry or a new entry is created and initialized to 1.

A high level algorithm description is given on algorithm 1, where :

- I is an input program instruction list in LLVM byte code (.bc) format
- i is an individual instruction within I
- M is a C++ map of the form <`string,int`>

Input: M,I
$t \leftarrow 0$
forall the $i \in I$ **do**
  if $M.containsKey(i)$ **then**
    M.valueForKey(i)+=1
  else
    M.insertKeyValuePair(< i,1 >)
forall the $keyValuePair \in M$ **do**
  print("Found "keyValuePair.value()
  "counts of: "keyValuePair.key())
  t += keyValuePair.value()
print("total instructions: "t)
Algorithm 1: Static instruction count algorithm

2.2 Static pass implementation summary

While the above is the abstract description of what we are trying to achieve, we have to adapt our algorithms to the syntax and semantics of an LLVM opt pass framework. The general purpose of an opt pass is to instrument an input bitcode file and ouptut its instrumentation. This process is done using different C++ classes (and methods) depending on the scope of the instrumentation. We chose to use the `ModulePass` class (and `RunOnModule(Module &M)` method) which gives us simultaneous access to all the functions in the input file, make are counting task easier.

In order to represent the

The most challenging of the above was understanding how instructions are represented by LLVM and accessing them from opt modules.

Our opt pass is able to iterate through the source module by implementing the **runOnModule()** virtual function of a **ModulePass** class.

```
virtual bool runOnModule(Module &M)()
```

Accessing instructions was accomplished by iterating through the **input source Module** using an **inst_iterator**. The Module contains all instructions from the compiled benchmark and the inst_iterator points at individual instructions.

```
for (Module::iterator m = M.begin(), e
                    for (inst_iterator I =
```

2.3 Benchmark analysis

We analysed module performance by comparing the logged instruction count with manual and automatic counts of the intermediate .ll byte code of each compiled benchmark. The automated counts were performed with the grep command:
¿¿cat [benchmark-name.ll] — grep [instruction-name] -c
The counts were accurate, confirming the pass' correct functionality.

Figure 1: Example correct pass output

3 Dynamic instruction count

Problem statement: Write a pass that instruments an input benchmark to count the number of times each instruction executes **Problem instance:**

- count program instructions at runtime
- output-per instruction count

3.1 Pass description

The pass differs from the prior pass in that it instruments the target byte code to perform an analysis on itself. After instrumentation, the benchmark program executes and dynamically counts instructions, outputting the total at termination. This dynamic analysis is therefore two phase with phase 1 being the opt pass and phase 2 being program execution.
First we describe the intended instrumented binary diagrammatically in Figure2

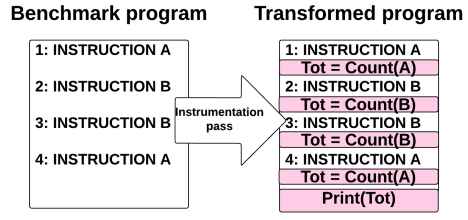

Figure 2: Required bytecode transformation

With a target binary defined we describe a code transformation that the opt pass must perform: where:
I is an input program instruction list in LLVM byte code (.bc) format
i is an individual instruction iterator I

The *count()* function can be recycled from section the *static count* pass. Similarly the *Print()* function can be recycled.

Input: I
1 $x \leftarrow 0$
2 $i \leftarrow$ ref(I.at(0))
3 **while** $i + 1 \neq 0$ **do**
4 $\quad i' \leftarrow$ **Count**(valAt(i))
5 $\quad i$.insertBefore(i')
6 $\quad i$ += 1
7 $i \leftarrow$ ref(I.last())
8 i.insertBefore(**print**())
Algorithm 2: Dynamic instruction count instrumentation pass

3.2 Instrumentation pass implementation summary

Besides the skills used for the static pass, implementation of the dynamic pass presented challenges of inserting instrumentation functions to the test bench code and further augmenting that code to call inserted functions.
Ensuring instrumentation functions were in scope during the opt pass was documented in the one of the project breif "hints" sections. Counting sorted lists of instructions was a probem solved during the static instruction count problem. Therefore the major challenges were:

- Finding a function signature in LLVM byte code
- Inserting a call to a function signature above a byte code instruction
- Passing parameters to an inserted function call

Finding a function signature **COSTAS IMPROVED THIS, I WILL SAY NO MORE**

Inserting a function call

4 Profiling branch bias

4.1 Pass description

Our pass tests the exit point of all basic blocks for conditional branches.
Should one be found, an instrument is added to increment a global counter of total branches. We then locate the 'taken' target of the branch in .ll byte code and instrument that to increment a second global counter of taken branch. where:

Input: I
1 **forall the** $i \in I$ **do**
2 **if** *i.isEqualTo(toInstruction(ConditionalBranch))* **then**
3 i.insertInstructionAbove($*Call%_incBranches*)
4 $t \leftarrow i$.getTakenBlockEntryInstruction()
5 t.insertInstructionAbove($*Call%_incTakenBranches*)
Algorithm 3: Branch bias algorithm

I is an input program instruction list in LLVM byte code (.bc) format
i is an individual instruction within I

The helper functions for house-keeping, counter incrementing and output formatting are not described, being recycled code slightly modified from functions used in the previous passes.

4.2 Instrumentation pass implementation summary

This opt pass builds on the dynamic instrumentation methods applied to the previous pass.
The additional requirement is run-time analysis of program logic rather than the run-time observation of it performed previously.
In addition to all skills of previous passes, analysing logic requires:

- The ability to instrument a particular instruction
- An ability to dynamically consume target code control flow data

Our opt pass locates conditional branches using the basic block iterator to evaluate instructions and a simple logic test of a dynamic cast and class member test to locate conditional branches:

```
for(BasicBlock::iterator BI = BB->begin(), BE
        if(isa<BranchInst>(&(*BI)) ) {
        BranchInst *CI = dyn_cast<BranchInst>(
        if (CI->isUnconditional())
                continue;
                //Else conditional branch foun
```

We dynamically consume control flow data by instrumenting the 'taken' conditional branch of interest. In .ll code a taken branch is the 0'th successor instruction of a conditional branch. We consume the *CI* pointer created previously to instrument the 'taken' target basic block.

```
BasicBlock *block = CI->getSuccessor(0);
                    BasicBlock::iterator taken
```

4.3 Benchmark Analysis

4.4 Conclusions

The project was completed succesfully . Learning outcomes were a working understanding of LLVM optimiser passes. We are capable of transforming and instrumenting code. We have developed an efficient method to develop, test and debug the LLVM source tree with a debug api and symbol indexed code base via the eclipse CDT. Understanding how to instrument code has provided a powerful tool for analysis of any optimisation heuristics applied by other opt passes. Instrumentation will allow us to both formulate and test heuristics based on the performance of compiled code.
To conclude, this project has provided us with

a solid foundation for the second upcoming project for CSE 231

References

Notes

[1] Remember to use endnotes, not footnotes!