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

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 Description:** 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

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: $M$,$I$
1  $t \leftarrow 0$
2  forall the $i \in I$ do
3      if $M.containsKey(i)$ then
4          $M$.valueForKey($i$)+=1
5      else
6          $M$.insertKeyValuePair(< $i$,1 >)
7  forall the $keyValuePair \in M$ do
8      print("Found "keyValuePair.value() "counts of: "keyValuePair.key())
9      $t$ += keyValuePair.value()
10 print("total instructions: "$t$)
```

Algorithm 1: Static instruction count algorithm

2.2 Physical Implementation Description

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 **forall the** $i \in I$ iteration, we use the iterator abilities of the Module and Function classes as follows :

```
for (Module::iterator m = M.begin(),
   e = M.end() ; e != m ; ++m) {
   for (inst_iterator I =
   inst_begin(m), E = inst_end(m) ;
   I != E ; ++I) {...}
```

The body of the algorithm (lines 3 to 6) could then be implemented using standard C++. Finally, to output the results of our analysis we implemented the `void print(raw_ostream &OS, const Module) const` function which allows use to print out our analysis result using the `-analyze` flag of the `opt` module.

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.

3 Dynamic instruction count

3.1 Pass Algorithm Description

We want to write a pass which counts the number of times each instruction is used when a program is executed. We cannot do this by a simple static analysis of the program, we need to instrument its bitcode by adding instructions that will count each instruction in the initial program whenever it is used during the execution, while making sure the instructions we added are not counted themselves.

```
1  count(Instructions $I_B$) begin
2      foreach $i \in I_B$ do
3          if $M.containsKey(i)$ then
4              $M$.valueForKey($i$)+=1
5          else
6              $M$.insertKeyValuePair(< $i$,1 >)
7  print() begin
8      $t = 0$
9      forall the $keyValuePair \in M$ do
10         print("Found "keyValuePair.value()
           "counts of: "keyValuePair.key())
11         $t$ += keyValuePair.value()
12     print("total instructions: "$t$)
```

Algorithm 2: Functions inserted in the input program; I_B is the set of instruction for a given basic block

To achieve this purpose, we insert a counting function and a printing function (both of which can be seen on algorithm 2) in the input programs, as well as calls to that function at the end of each basic block (see figure 1). Given that all instructions in a basic block are executed atomically, a single call to the counting function is sufficient.

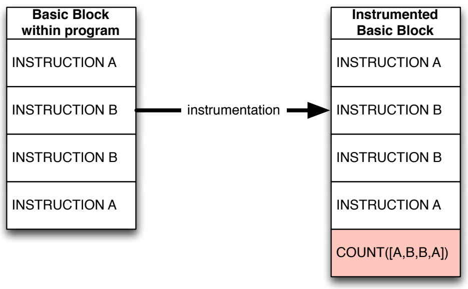

Figure 1: bitcode instrumentation

3.2 Physical Implementation Description

Functions shown on algorithm 2 are written in C++, compiled separately and then linked to the input program later (as described in the instructions), but calls to these functions have to be inserted using the `opt` pass.

Again, we used the `ModulePass` class and `RunOnModule(Module &M)` method for the implementation. First, we iterate through each basic block (using the `Module` and `Function` iterators) and collect the instructions for that block using the `CI.getOpcodeName()` method provided by the LLVM library for each instruction `CI`. We then concatenate all of these instructions into some comma separated string called `result`. Finally, we use the following code snippet to call the linked function :

```
// BE points to the end of the BB
builder.SetInsertPoint(BB,BE);
Value* myStr =
builder.CreateGlobalStringPtr(result);
builder.CreateCall(hookCount,myStr);
```

As can be seen from this snippet, we used an `IRBuilder` class to help constructing the call insertion. We did not have to concatenate instructions (we could have used an array of strings), but strings are easier to pass to call instructions using the LLVM library than arrays of strings. Notice that the C++ implementation of the `count` function accepts a `char*` as input, accordingly.

3.3 Benchmark Analysis

In the execution of the gcd example, the main function calls the `gcd()` function on input `(72,32)`. We can easily predict that the `gcd` will be called twice recursively before returning to main, which in turn will return at the end of execution. No other function calls are made. As such we should expect the `ret` instruction to be called 4 times, which is precisely the number of times it is called in the benchmark we generated.

4 Profiling branch bias

4.1 Pass Algorithm Description

We want to write a pass which helps us predict, for each function of the input program, what is the bias that any branch in the function is taken during the execution. Similarly to the previous section, we need bitcode instrumentation to accomplish this task. As done previously, we insert independent functions to the input program described in algorithm 3.

These functions expect a function name as the input and increment the found or taken count for that function. This count is eventually displayed when the program terminates. We insert a call instruction to the `branchFound` function right above a branch instruction and the `branchTaken` function right above the first instruction of the basic block reached when the branch is taken.

```
1  branchFound($fName$) begin
2      if $Found.containsKey(fName)$ then
3          $Found.valueForKey(fName) += 1$
4      else
5          $Found.insertKeyValuePair(< fName, 1 >)$
6  branchTaken($fName$) begin
7      if $Taken.containsKey(fName)$ then
8          $Taken.valueForKey(fName) += 1$
9      else
10         $Taken.insertKeyValuePair(< fName, 1 >)$
11 print() begin
12     foreach $fName \in Found.keys()$ do
13         $v_f \leftarrow Found.valueForKey(fName)$
14         $v_t \leftarrow Taken.valueForKey(fName)$
15         $print(fName + "" + v_f + "" + v_t + "" + \frac{v_t}{v_f})$
```

Algorithm 3: where *Found* and *Taken* are C++ maps of the form `<string,int>` and *fName* is a string expected to be an functionName from the input

4.2 Physical Implementation Description

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 = BB->end(); BI != BE; ++BI){
   if(isa<BranchInst>(&(*BI)) ) {
    BranchInst *CI =
    dyn_cast<BranchInst>(BI);
    if (CI->isUnconditional())
      continue;
    //Else conditional branch found...
```

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 takenInsertPt
= block->getFirstInsertionPt();
```

4.3 Benchmark Analysis

Let's have a closer look at one of the benchmark applications and specifically at the gcd.cpp. Function gcd() gets called from the main method with two arguments (72, 32), it checks if the second argument is equal to zero and since it's not, the branch is not taken. Instead it recursively calls itself with arguments (32,72%32=8). Once again though the second argument is not equal to zero and the branch is not taken but the function gets called again with arguments (8, 32%8=0). This time the second argument is equal to zero and the branch is taken, thus it returns to the main function and the program terminates successfully. To sum up, there was only one function with a branch, and this branch was encountered three times but it was taken only once, so the bias is going to be 1/3 as we can see at the log file of the output.