Abstract

In this work, we use LLVM passes to profile programs in its Intermediate Representation (IR). To this extent, we count the number of instructions statically in the IR, as well as the instruction executions at runtime. Additionally, we analyze the branch bias per function. Through this process, we familiarized ourselves with the LLVM framework for program analysis, extension, and modification.

1 Collecting Static Instruction Counts

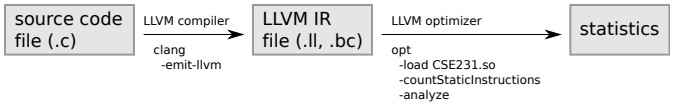

Figure 1: Workflow for collecting instruction counts statically.

Figure 1 shows the compilation and linkage workflow for this part of the project. We provide an LLVM pass that does not modify the input program but merely outputs program statistics to `stdout`.

1.1 Basic Idea

In this section, we design an LLVM pass that prints out the number of instructions statically at compile time. This pass consists of two stages. The first stage traverses all instructions. We keep track of the number of occurrences for each instruction. We maintain a C++ map to associate instructions with their counter. In the second stage, just before the pass terminates, we print out the contents of the map. With this high-level information, we next describe the implementation details.

1.2 Implementation

We define our pass `CountStaticInstructions` as a subclass of `ModulePass`. Within our pass, we construct a map to store the instruction counts. As our pass class might be instantiated multiple times, we want to ensure only one map instance is created by making it static.

The first stage goes through every instruction. To this end, we override the `runOnModule` method provided by `ModulePass`. Within this method, we iterate through every function, every basic block and every instruction using their respective iterators. Along with the method `Instruction::getOpcodeName`, we record the number of occurrences for each instruction using the map.

The second stage prints out the statistics we recorded by overriding the `print` method provided by `ModulePass`. This method is called whenever the pass is complete and the `-analyze` flag is passed to the LLVM optimizer. Note that the optimizer does not modify the program here.

2 Collecting Dynamic Instruction Counts

2.1 Basic Idea

At a high level, this pass obtains dynamic instruction counts by inserting an instrumentation module into the program at compile time. At the end of every basic block, the LLVM pass inserts calls to our instrumentation module, which is subsequently linked to the original executable. These calls record the executed instructions within the current basic block. In this way, our statistics-collection code in the module is executed on the fly at runtime. When the program is run, it prints out the dynamic instruction counts just before the main function terminates.

2.2 Implementation

The structure of the pass is similar to the static one. We go through every module, every function and every basic block. For each basic block, we statically create a C++ map, which keeps track of the instructions in the basic block as well as their counts. Next, we find the terminator of this block; an example would be a `ret` or `jump` instruction.

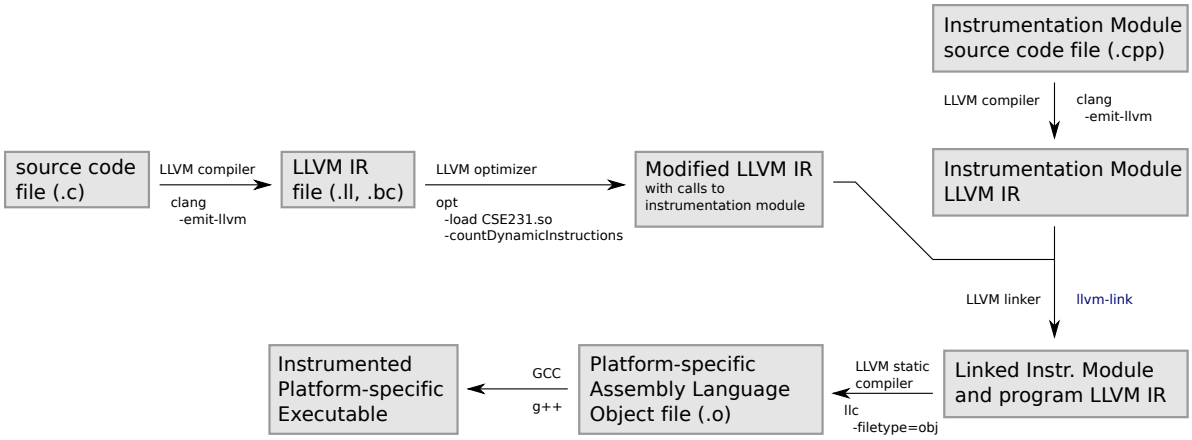

Figure 2: Compilation and linkage workflow for Section 2 and Section 3.

Just before the terminator instruction, the pass makes a call to the statistics-collection code, supplying the opcode (string) - count (integer) pair as arguments[1]. We could have invoked the external statistics-collection code upon every instruction, but this would have degraded the performance for large basic blocks that are called frequently. Given that entering a basic block implies that every instruction within it must be executed, we can significantly reduce the number of calls to the external statistics-collection code. This is done by aggregating the instruction counts per basic block at compile time, and recording them once before exiting a basic block[2].

Name Mangling The LLVM pass is calling a function in a linked compilation unit whose name was mangled at compile time. We have to call the C functions `void handleOpcode(char* opcode, int count)` and `void printOpcodeStatistics()`, and, therefore, in the instrumented program, we call functions with the name `_Z12handleInstructionPci` and `_Z21printOpcodeStatisticsv`, respectively[3].

Calling the Instrumentation Module Invoking the linked statistics-collection function from the LLVM pass is straightforward. We simply call it by its mangled name, using the `Module`'s `getOrInsertFunction` method, followed by `IRBuilder`'s `CreateCall` method. The statistics-collection function takes as arguments the instruction as a C-string and the number of its occurrences within a basic block as an integer. This function stores this information as a key-value pair into a static map in the instrumentation module file. As such, we can keep track of the dynamic instruction counts in this linked statistics-collection function. While our design choice of directly calling the function by its mangled name does couple the module pass to our external C++ file (instrumentation module), we still consider this a modular design since the two are conceptually coupled anyway, such that neither one is useful without the other one.

Printing Statistics Finally, we need to print out the statistics from the map in the external C++ file. It turns out to be another challenge. In the static pass, we simply override the pass' `print` function to display the statistics. In contrast, this is not feasible in the dynamic case, since the static pass will run and terminate at compile time and the dynamic instruction count can vary depending on the algorithm and its input (e.g., GCD with differing input values). We have to print the statistics just before the main function exits.

To determine the main functions termination point, we do the following for every instruction we see at compile time: we check if the instruction's parent function has the name `main`. If so, we further check if the instruction is a `return` instruction. If both of these conditions are satisfied, our LLVM pass will insert a call to our print-statistics function in the instruction module. If the `main` function has multiple `return` instructions, we insert the call at every one of them. In this way, the print-statistics function is guaranteed to be invoked just before the executable exits,

[1]This cannot be done after the terminator instruction, as LLVM would complain: every basic block should end with a terminator instruction. Besides, code after a terminator instruction could be unreachable.

[2]Note that the instruction counts inside a basic block is known at compile time. However, the number of executions of a basic block is not.

[3]We discovered the function names by examining the LLVM IR code generated from the instruction module.

assuming a standard termination (as opposed to exceptions such as seg-faults). A concern worth mentioning, albeit trivial to implement correctly, is that we must be sure to insert the call to record the dynamic count (including the `ret` opcode) prior to inserting the call to print the acquired statistics.

3 Profiling Branch Bias

3.1 Basic Idea

Profiling branch bias follows a similar structure as in the previous part. Again, we have an LLVM module, along with an instrumentation module to be linked to. The instrumentation module contains a function that gathers statistics of branches, as well as a function that prints out these statistics. At a high level, the LLVM module will insert calls to both of these functions, so that they can profile branch bias in run-time.

3.2 Implementation

The problem is inserting calls in the original executable. Our goal is to find all conditional branches; for each of them, we keep track of how often a branch condition is evaluated to either true or false. To this end, we iterate through every module, every function, every basic block and every instruction. For each instruction, we determine if it is a conditional branch. For each conditional branch, we retrieve the condition (could be an instruction) via `IRBuilder`'s `getCondition` method. We then create a new boolean value via a not-equals comparison between the condition value and a zero value (equivalent to false in C++). We then insert a call to our branch-profile function in the instruction module, passing in the boolean value. Additionally, we set the branch condition to this evaluated value using `IRBuilder`'s `setCondition` method, in order to avoid any behavioral changes based on stateful expression evaluations. For instance, if the branch condition is `x++ % 2`, we only want to evaluate it once.

```
branch <condition> <then-label> <else-label>
```

(a) Original IR.

```
Value v = <condition>
bool b = (v != 0)
profile_branch(<function-name>, b)
branch b <then-label> <else-label>
```

(b) Modified IR.

Figure 3: Pseudocode representation of IR modification to profile branch bias.

Our branch-profile function takes two arguments: the function's mangled name, and the boolean value representing the evaluated branch condition. The function maintains two static maps for the total number of executed branch instructions and how many of their conditions evalutated to true (per function), so that we can print out the statistics about the branches at the end of the program.

Similar to the previous part, we insert calls to the print-statistics function at every `return` instruction in the main function. This guarantees that the statistics are displayed just before the main function exits.

4 Example

In this section, we present the statistics gathered for the `gcd` benchmark. We found this program particularly interesting because it is conceptually simple while including the complexity of recursion.

In the static case, the `ret` instruction appears twice because `gcd` has two functions (`main`, `gcd`). In the dynamic case for input `gcd(72,32)`, the `ret` instruction is executed four times: thrice from `gcd` and once from `main`. This program has a single branch instruction which resides in the function `gcd`, implying it has reached the base case if true and otherwise returning the result of a recursive call. Since we have three dynamic `ret` executions from the function `gcd`, the final one must be the base case and the other two originate from the recursive calls. Therefore, the branch condition was true one out of three times, which is consistent with our observations.