CSE 231 Project Report: LLVM Project

Zhenchao Gan, Tao Li, Antonella Wilby

November 3, 2015

Overview

In this project, we implemented LLVM passes for collecting static instruction counts, dynamic instruction counts, and profiling branch biases. Each LLVM pass takes in a program in its Intermediate Representation (IR) and uses that information to profile the program. In section one, we describe a pass that counts the number of instructions in the IR, which is the static instruction count (IC). In section two, we describe a pass that counts the number of instructions executed at runtime, or the dynamic IC. In section three, we describe a pass that analyzes the branch bias for each function in a program, i.e. the taken/not-taken ratio of the branch instructions in the program. We used each pass to profile four benchmarks: `welcome.cpp`, `compression.c`, `gcd.cpp`, and `hadamard_test.c`.

1 Collecting Static Instruction Counts

We implemented an LLVM pass that takes in the Intermediate Representation of a program, and counts the number of occurrences of each separate instruction, then prints the results after the program has been analyzed.

1.1 Algorithm and Implementation

We defined our pass `csi` as a struct inheriting from `ModulePass`. We first create a C++ map to store the instructions encountered while profiling the program and the associated instruction counts. The algorithm first iterates through a given Module (e.g., `hadamard.bc`). Within the Module, the algorithm iterates through every function, and within the function the algorithm iterates through each basic block. As it iterates through the basic block, the algorithm gets instructions using the function `Instruction::getOpcodeName()` and increments the associated counter within the map for each instruction in the basic block. After the entire IR has been analyzed, the algorithm prints the contents of the map to `stdout`.

1.2 Benchmark

We discuss the results of our `csi` pass on the `compression` benchmark. Figure 1 shows static instruction counts for a select set of common instructions, with some lesser-used instructions

add	526
br	3143
call	473
getelementptr	2806
load	6989
mul	37
store	3044
sub	272
TOTAL	22167

Figure 1: Selected static instruction counts from the `compression` benchmark

omitted for brevity. Please refer to the logs for complete statistics.

The total static instruction count for the program is 22167 instructions. As would be expected, the most common instructions include `br` instructions, `load` instructions, and `store` instructions.

2 Collecting Dynamic Instruction Counts

We implemented an LLVM pass called `cdi` for computing the dynamic instruction count of a program, or the number of executed instructions. Our pass instruments the program by inserting a call during execution to a function in our instrumentation module that counts the number of times each instruction executes.

2.1 Algorithm and Implementation

The instrumentation module contains a map for collecting instruction counts, a function called `collectRuntimeInfo()`, and a function called `dumpInfo()`. The `collectRuntimeInfo()` function takes in an opcode name and a counter for that opcode, and uses the counter to update the counter for that opcode stored inside the map. The `dumpInfo()` function is called at the very end of the pass execution, and prints the contents of the instruction counter map to `stdout`.

As in our first pass, this pass iterates through each module, function, and basic block in the program. For each basic block, the program creates a C++ map to store the instructions executed during execution of that block. If the program encounters any of the opcodes `jump`, `ret`, or `br`, it means the end of the basic block has been reached, so we insert a call to our instrumentation module. We use the function `Module::getOrInsertFunction()` and a call to `IRBuilder::CreateCall()` to insert a call to the instrumentation module's `collectRuntimeInfo()` function in order to add the instruction counts gathered during this basic block to the overall instruction counter map. In the pass, we call this instrumentation module function with the mangled function name `_Z18collectRuntimeInfoPci`.

add	14606
br	28848
call	140
getelementptr	33676
load	86528
mul	29
store	29634
sub	1780
TOTAL	241847

Figure 2: Selected dynamic instruction counts from the `compression` benchmark

After the entire program has executed, we use these same functions to insert a call to the `dumpInfo()` function in the instrumentation module. To determine if it is the main function that is executing, we to see if the instruction is a `return` function and if its parent function name is `main`. If so, we use the `Module::getOrInsertFunction()` and `IRBuilder::CreateCall()` functions to insert a call to our `dumpInfo()` function using the mangled function name `_Z8dumpInfov`.

2.2 Benchmark

We discuss the results of our `cdi` pass on the `compression` benchmark. Figure 2 shows dynamic instruction counts for the same set of common instructions as in Section 1, with lesser-used instructions omitted. The total dynamic instruction count for the program is 241847 executed instructions. As expected, `branch`, `load`, and `store` instructions continue to make up the majority of executed instructions, but the `add` and `getelementptr` instructions are also executed very frequently. The number of `call` and `mul` instructions decreased, suggesting that there are branches in the program which are never taken.

3 Profiling Branch Bias

We created an LLVM pass called `pbb` and an instrumentation module which counts the number of taken and not-taken branches and computes their ratio during runtime.

3.1 Algorithm and Implementation

As with the dynamic instruction count pass, we created an instrumentation module with a function for collecting branch taken/not-taken statistics, a function for printing the statistics to the screen after finishing execution, and two maps, one for storing the number of times a branch was taken (`Truemap`), the other for storing the total number of times a branch comparison was evaluated (`Totalmap`). The statistics function, called `collectBranchInfo()`, takes in a branch function name and a boolean value indicating whether or not the branch

was taken (`true` for taken, `false` otherwise). It increments the counter for that branch in `Totalmap`, and if the boolean evaluates to `true` also increments the counter in `Truemap`. The statistics printing function, called `dumpInfo()`, is called just before the pass exits and computes the taken/not-taken ratio based on the values of the counters in the two maps, then prints this information to `stdout` in table form.

As in the first two passes, the algorithm iterates through the modules, functions, and basic blocks. In each basic block it checks each instruction to see if the instruction is a `branch` and if it is we further check whether it is a conditional branch. A conditional branch will have three operands. If so, it first gets the name of the branch using the `IRBuilder::CreateGlobalStringPtr()` function and the name of the instruction's parent's parent function. Then, it creates a vector for storing the operands to the `collectBranchInfo()` function, with the first operand the branch name string pointer just created, and the second operand the boolean result of the branch comparison (accessed by getting the instruction's first operand using the `getOperand()` function). Then, the pass inserts a call to our instrumentation module using the mangled function name `_Z17collectBranchInfoPcb` in the same manner as in the dynamic instruction count pass, using the `Module::getOrInsertFunction()` and IRBuilder::CreateCall() functions.

This pass inserts a call to `dumpInfo` in the same way as in the dynamic instruction count pass, using the mangled name `_Z8dumpInfov`.

3.2 Benchmark

Figure 3 shows the results of our `pbb` pass on the `compression` benchmark. Of the 25 functions, 16 of them are taken more times than they are not taken (i.e., a taken/total ratio of over 0.5). Additionally, of the eight branches that are taken over 100 times, all but one are taken more than they are not-taken, and many of these are taken over 75% of the time. This suggests that interesting optimizations can be made in the area of branch prediction using this information.

Function	Bias	Taken	Total
main	0.222222	2	9
mz_adler32	0.955882	65	68
mz_compress2	0.000000	0	3
mz_deflate	0.500000	5	10
mz_deflateBound	1.000000	1	1
mz_deflateEnd	1.000000	2	2
mz_deflateInit2	0.222222	2	9
mz_inflate	0.538462	7	13
mz_inflateEnd	1.000000	2	2
mz_inflateInit2	0.400000	2	5
mz_uncompress	0.000000	0	3
tdefl_calculate_minimum_redundancy	0.650246	264	406
tdefl_compress	0.541667	13	24
tdefl_compress_block	0.000000	0	1
tdefl_compress_lz_codes	0.704082	207	294
tdefl_compress_normal	0.611734	2127	3477
tdefl_create_comp_flags_from_zip_params	0.285714	2	7
tdefl_flush_block	0.666667	38	57
tdefl_flush_output_buffer	0.750000	3	4
tdefl_huffman_enforce_max_code_size	0.888889	96	108
tdefl_init	0.000000	0	1
tdefl_optimize_huffman_table	0.756056	1342	1775
tdefl_radix_sort_syms	0.979381	855	873
tdefl_start_dynamic_block	0.409742	715	1745
tinfl_decompress	0.863445	4521	5236

Figure 3: Branch biases from the `compression` benchmark