CPE 631 Advanced Computer Systems Architecture: Homework #5

Problem #1 (20 points)

Compile and execute the sample PIN tool *inscount0.cpp* provided by Intel (in *path-to-pin/source/tools/ManualExamples*). What is the output of this program? Run at least two benchmarks (621.wrf_s and 657.xz_s) for the train input set from the SPEC2017 suite and explain the results. Please refer to http://lacasa.uah.edu/portal/Upload/tutorials/SPEC/SPEC_CPU2017.pdf to see how to run SPEC CPU 2017 benchmarks.

Problem #2 (30 points)

Modify the *inscount0.cpp* to count the number of basic blocks, number of memory reads, number of memory writes and the total number of executed instructions. Your tool should profile single-threaded applications and generate the output file as shown below. Demonstrate your program on simple benchmarks (e.g., matrix multiplication, accumulating the elements of the array) and profile at least two benchmarks from SPEC CPU 2017 suite (621.wrf_s and 657.xz_s) for the test input set.

```
// Benchmark Characteristics
// Time: Mon Feb 5 08:54:20 2018
// Benchmark mm_serial

// Program Statistics
Basic Blocks: 1000
Memory Reads: 3400
Memory Writes: 2600
Total Instructions: 32000
```

Note: Simple benchmarks can be found at */apps/arch/arch.tut/simpleBenchmarks*

Bonus: (up to 20 points)
Count and print the statistics for each thread in a separate column for multithreaded programs as shown below.

```
// Benchmark Characteristics
// Time: Mon Feb 5 08:54:20 2018
// Benchmark mm_parallel

// Program Statistics
                        Thread 0     Thread 1     Thread 2     Thread 3
Basic Blocks:           1000         100          100          100
Memory Reads:           3400         1200         1200         1200
Memory Writes:          2600         890          890          890
Total Instructions:     32000        10000        10000        10000
```

Problem #3 (100 points)
Design and implement a PIN instrumentation tool for profiling dynamic basic blocks (or streams) in a program. A dynamic basic block is defined as a sequential run of instructions that starts with an instruction that is a target of a taken branch and ends with the first taken branch in sequence. Thus, one dynamic basic block may contain one or more static basic blocks that are defined as follows: a sequential run of instructions that starts with an instruction that is a target of a taken branch and ends with the first branch in sequence. Each instruction stream is uniquely identified by a pair (SA, SL), where SA represents the stream starting address (the address of the first instruction in a stream) and SL represents the stream length (the number of instructions in the stream).

Your tool should profile a single-threaded application and generate an output file with the following information.

```
// Stream Profiler Output
// Time: Mon Feb 5 08:54:20 2018
// Benchmark 22665


// Program Statistics
numStreamS: 4135
numStreamD: 188789
numMemRef: 526405
numIrefs: 1522721
maxStreamLen: 313
avgStreamLen: 9.0
```

Table 1 explains the meaning of the statistics displayed above.

Table 1. Program statistics.

Variable	*Description*
numStreamS	The number of unique program streams (static).
numStreamD	The number of program streams executed (dynamic).
numMemRef	The number of memory referencing instructions executed (dynamic).
numIrefs	The number of instructions executed (dynamic).
maxStreamLen	The maximum stream length (the maximum number of instructions executed in a sequence without a branch being taken).
avgStreamLen	The average stream length (calculated as numIref/numStreamD).

Your analyzer should also print a stream table as follows.

```
// Format: stream id: (stream starting address, # instructions, # memory ref
instr.), (# number of occurrences);
// {(the next stream id, freq.)}
0: (0x0, 0, 0), (0); {  }
1: (0xf7e00af0, 2, 1), (1); { (2, 1) }
2: (0x32f7e01120, 31, 12), (1); { (3, 1) }
3: (0x32f7e011bc, 2, 0), (2); { (4, 1)(10, 1) }
4: (0x32f7e011a8, 7, 2), (11); { (5, 1)(4, 9)(10, 1) }
5: (0x32f7e011a8, 11, 2), (1); { (6, 1) }
6: (0x32f7e014d0, 4, 0), (3); { (7, 1)(13, 2) }
7: (0x32f7e014f0, 4, 0), (1); { (8, 1) }
8: (0x32f7e01510, 8, 1), (1); { (9, 1) }
9: (0x32f7e011b0, 6, 1), (1); { (4, 1) }
10: (0x32f7e011a8, 18, 4), (2); { (11, 2) }
11: (0x32f7e011bc, 13, 2), (3); { (3, 1)(11, 1)(12, 1) }
```

Format of an entry in the stream table is explained in the header above. Each line starts with an entry number (e.g., 1), the stream starting address (0xf7e00af0), the number of instructions in the stream (2), the number of memory referencing instructions in the stream (1), the number of occurrences of the stream (1). The entry is followed by a list of subsequent streams (2, 1), indicating that stream 1 is followed by stream 2 (once). Please note that a stream may have multiple streams following it (e.g., stream 4 in the example above). That simply means that the stream 4 ends by an indirect branch with multiple targets.

Demonstrate your program on a simple benchmark (e.g., matrix multiplication) program. Profile at least two SPEC CPU 2017 applications (621.wrf_s and 657.xz_s).

Bonus: (up to 30 points)

Numerous extensions are possible to this tool. One option is to provide visualization of program execution (e.g., each stream can be a dot, and its frequency can correspond to dot color). Another option is to support multithreaded programs.

Supplemental material:

(1) You should get familiar with PIN by reading pin manual and analyzing provided examples. The examples are in */apps/arch/pin-3.2-81205-gcc-linux/source/tools/ManualExamples* directory on our machine. You can copy this directory to your working directory. Alternatively, you can copy entire pin directory into your working directory.
(2) To compile your tool along with sample tools you can modify the existing makefile to include the name of your tool. To compile your tool along follow the instructions given in the pin user manual.
(3) One entry of the stream table could be defined as follows:

```
typedef struct {
    ADDRINT       sa;                      /* stream starting address */
    UINT32        sl;                      /* stream length */
    UINT32        scount;                  /* stream count -- how many times it has been 
executed */
    UINT32        lscount;                 /* number of memory-referencing instr. */
    UINT32        nstream;                 /* number of unique next streams */
    UINT32        nstreamid[MAX_LINK];     /* next stream id (index in the stream table) */
    UINT32        ncount[MAX_LINK];        /* counter how many times the next stream is 
encountered */
} stream_table_entry;
```

(4) An instruction stream ends with a taken branch. The last instruction stream ends by the last instruction in the program.
(5) To expedite search in the stream table, you should use a hash table (let us say with 4 times as many entries as the stream table to reduce probability of collisions), and have a hash function to quickly locate an entry in the stream table). The hash function can be based on the stream starting address and stream length (e.g., XOR function of certain subfields).
(6) Please try to verify that your analyzer is working correctly by using a relatively small and well-understood benchmark.