EE382M – 15: Assignment 3

Professor: Lizy K. John
TA: Jee Ho Ryoo
Department of Electrical and Computer Engineering
University of Texas, Austin

Due: 11:59PM October 21, 2014

1. Introduction

The goal of this assignment is to familiarize yourself with a popular profiling tool. Profiling is one of the fundamental steps used to understand the characteristics of a program. This assignment will give you some exposure to write C/C++ based program analysis tool that interfaces with other public domain tools used by computer architecture research community. In this assignment you will develop a program profiling tool using a binary instrumentation system called PIN. PIN provides rich Application Program Interface (API) that allows program analysis routines to instrumentation calls at arbitrary locations in the executable.

1.1 Setup and Requirements

- You will need access to a Linux machine with the Pin tool installed. You should be able to install any kit for gcc version 3.4 or above.

- Pin tool requires approximately 100MB, so make sure you clean up your directory choose to use the LRC machines. If you are running out of space, consider using the `/scratch` directory.

1.2 Deliverables

The following deliverables must be completed for this assignment

- Pin analyzer code
- Assignment report

2. Pin Tool

2.1 Overview

Pin is a dynamic binary instrumentation tool for x86 ISA that profiles the code on the fly. Since it profiles the runtime binary, there is no need to recompile the source code. This tool provides a rich API that allows context information to be injected to the user's profiling code as parameters. Please download the Pin tool at the following website:
https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

This tool comes with a few example codes, so if you are not familiar with this tool, please take a look at these codes before you start this assignment. You will need to access the Pin API as well, so please carefully read the following document as well:
https://software.intel.com/sites/landingpage/pintool/docs/67254/Pin/html/

2.2 Profiling Lawrence Livermore Loop

Along with this Lab, you are given one of the Lawrence Livermore Loop (LLL) programs (lll3.c). Compile the code with –O2 optimization and –S flag to see the assembly.

% gcc –O2 –S lll3.c

Please look at the raw assembly code (of the LLL inner loop) and tell how many assembly instructions are used. In your report, guess what the dynamic number of instructions would be based on the assembly code you see (you can look in the source to see the loop bound). Now, run your Pin tool to report the actual dynamic number of instructions. Show the following numbers in the report

Instruction count (expected) based on Assembly code = your educated guess
Instruction count reported by PIN = PIN results

Do the numbers match?

If they don't match, repeat the experiment with loop bound of 200,000. Use the difference from the first experiment to the second experiment to compute the instruction count for 100,000 iterations. Now, show the following numbers based on this new experiment.

Instruction count (expected) based on Assembly code = your educated guess
Instruction count reported by PIN = PIN results

Do the numbers match better? Explain your results.

2.3 Profiling Lawrence Livermore Loop with multiple optimization levels

Compile the given LLL code (lll11.c) with –O0 and –O2 optimization. Report the total dynamic number of instructions using PIN tool for both compilation levels. Assuming IPC of 1, what is the speedup of compiler optimization level –O2 over –O0?

2.4 Evaluate Your Assignment 1 Cache Simulator

Determine how many dynamic instructions does it take for your cache simulator to simulate a cache when you feed in the sample config and trace files provided with this assignment. Essentially, you will use PIN to profile your own cache simulator that you designed in assignment 1. Profile your own (i) single core single-level, and (ii) single-core two-level cache simulators Profile the cache simulator binaries obtained with –O0 and –O3 compiler optimizations. Report the instruction counts at the 2 optimization levels. What is the number of instructions needed to simulate one trace file entry in each case?

2.5 Writing Your Profiling Code and Running your analyzer

In this part, you will run your PIN analyzer with (i) LLL3 (ii) LLL11 (iii) SAXPY (iv) DAXPY (v) gcc and (vi) bzip2 benchmarks. You will write your own PIN-based

profiling tool that does the data collections described below in one run. Name your code as `program_analyzer.cpp`. For each benchmark, please include the following in the report.

1. Instruction mix must be collected with the total number of dynamic instructions, integer, floating point, load, store, branch, and other instructions.

2. Collect details on the percentage of total branches that are taken and the percentage of total forward branches that are taken.

3. Calculate the average basic block size. Pin uses a different definition of the basic block than the classical textbook definitions. For this assignment, you can assume basic block size to be the number of instructions between two consecutive branches/calls in the dynamic instruction stream.

4. Also collect Read-After-Write (RAW), Write-After-Write (WAW) and Write-After-Read (WAR) distribution in the dynamic instruction stream. RAW dependency distance is defined as the total number of instructions in the dynamic instruction stream between the production (write) and consumption (Read) of a register instance. For example, if instruction X writes to Register R, and subsequently register R is read by instruction Y, the dependency distance is Y-X instructions. If two consecutive instructions are dependent, the distance is 1. An instruction may have multiple operands and each operand may have a different dependency distance to its producer. Find dependency distance for each operand. Similarly, you can calculate the WAW and RAW register dependency distances. You will present your results as the total number of RAW, WAW, and WAR register dependencies that have a distance up to 2, 8, 32 and greater than 32 instructions.

5. Measure and report inter-reference temporal density function, which is an intrinsic measure of temporal locality. You will report the density function for the granularity of byte, cacheline and page granularity. Use 32B cacheline size and 4KB page size. See details at the end of this document.

3. Output Format

You analyzer in Section 2.5 must be able to print the stats in the following format:

```
<DynInsCount>
<IntCount>
<FPCount>
<LdCount>
<StCount>
<BrCount>
<PBrTaken>
<BrForwardTaken>
```

```
<AvgBBSize>
<RAWdep 0-2>
<RAWdep 3-8>
<RAWdep 9-32>
<RawDep greater than 32>
<wAWdep 0-2>
<wAWdep 3-8>
<wAWdep 9-32>
<wAWdep greater than 32>
<wArdep 0-2>
<wArdep 3-8>
<wArdep 9-32>
<wArdep greater than 32>
<temporal density_byte 0-2>
<temporal density_byte 3-8>
<temporal density_byte 9-16>
<temporal density_byte 17-32>
<temporal density_byte greater than 32>
<temporal density_cacheline 0-2>
<temporal density_cacheline 3-8>
<temporal density_cacheline 9-16>
<temporal density_cacheline 17-32>
<temporal density_cacheline greater than 32>
<temporal density_page 0-2>
<temporal density_page 3-8>
<temporal density_page 9-16>
<temporal density_page 17-32>
<temporal density_page greater than 32>
```

The above should be the only thing printed. **Make sure there are exactly 36 rows in this output**. **If any additional/fewer characters or debug information are printed, the grading script will consider them as incorrect.**

4. Submission Instructions

4.1 Submission instructions

Your assignment must be compressed in tar.gz format. Please use the following command to tar your submission:

% tar –zcvf <your last name>-a3.tar.gz <your working directory>

Make sure that the name of your working directory is named <your last name>-a3.

An automated script will be used to grade this assignment. Thus, if the script fails to grade your code due to not following instructions, you will lose substantial part of your grade.

There will be a submission link available on the course website, so please submit your tarball by deadline. Check the following before you tar your directory.

1. Please remove all your temporary files.
2. You are required to submit your code and report.
3. The report must be in the pdf format.
4. All graphs and figures must be included in the report. Figures not included in the report will not be looked, and thus, will not be grade.
5. Your code must only print information in the exactly the same format as what is presented in Section 3.

5. Helpful Tips

Lectures gave you information on the locality metrics. The following papers give more details:

[1] Conte, T.M.; Hwu, W.-M.W., "Benchmark characterization for experimental system evaluation," *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on* , vol.i, no., pp.6,18 vol.1, 2-5 Jan 1990
[2] John, L.K.; Vasudevan, P.; Sabarinathan, J., "Workload characterization: motivation, goals and methodology," *Workload Characterization: Methodology and Case Studies, 1999* , vol., no., pp.3,14, 1999
[3] Mattson, R.L.; Gecsei, J.; Slutz, D.R.; Traiger, IL., "Evaluation techniques for storage hierarchies," *IBM Systems Journal* , vol.9, no.2, pp.78,117, 1970
[4] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (PLDI '03).

6. Important Paragraphs from the References

Temporal locality is often defined as if an item is referenced, it will tend to be referenced again soon. The inter-reference temporal density function, *fT(x)*, is defined as the probability of there being *x* unique references between successive references to the same item. Consider the following sequence of instruction addresses:

3000, 3002, 3004, 3000, 3004, 3000, 3002

The number of unique address references *x* between the first reference to address 3000 and the next reference to it is 2. The number of unique address references between the first reference to address 3002 and the next reference to it is also 2 (there are 4 references but only 2 unique references). Similarly, there is only one unique reference between the two consecutive references to address 3004 and between the second and third reference to address 3000. Thus if we consider the instruction reference stream as a whole, there are two instances where *x* has value 1 and two instances where *x* has value 2. We normalize these counts over all possible values of *x* to get the inter-reference temporal density function *fT(x)* as *fT(0)* = 0, *fT(1)* = 0.5 *fT(2)* = 0.5. The formal definition of inter-reference temporal density function is defined in Definition 2.3 below. [1][2]

DEFINITION2. 3: Define $f^T(z)$, the inter-reference temporal density function, $f^T(z)$,,to be the probability of there being z unique references between successive references to the same item,

$$f^T(x) = \sum_t P\left[u(w(t)) = x \right].$$

The interreference temporal density function is a measure of temporal locality of a reference stream. The performance of buffers managed under stacking replacement policies (e.g., LRU) depends directly on this measure of temporal locality. The hit ratio for a fully associative buffer of size N is $h(N) = \Sigma_{ty \le N} f^T(y)$.

```
Calc_loc_measures(r_i):
 begin
     if  not first time r_i encountered then
     begin
         d ← depth(r_i)
          remove r_i from the stack
          for  all r_j with depth(r_j) < d
          begin
              dist ← |α(r_j) − α(r_i)|
              f̂^S(dist) ← f̂^S(dist) + 1
          end
         f̂^T(d) ← f̂^T(d) + 1
     end
     push(r_i)
 end
```

Figure 1: The algorithm for calculating the locality distributions [3]

Figure 1 shows the algorithm to find the locality distribution while Figure 2 shows the histogram of the number of unique references. Figure 3 and Figure 4 are provided an example and to help you to write more efficient algorithm.

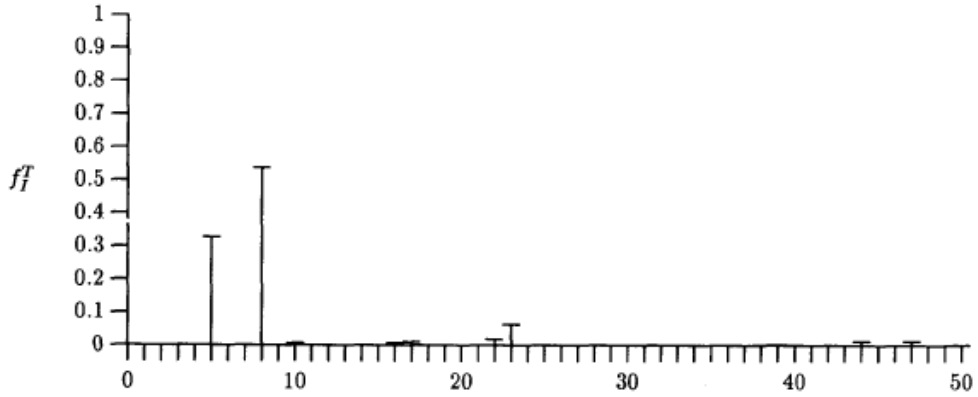

Figure 2: Number of unique references [4]

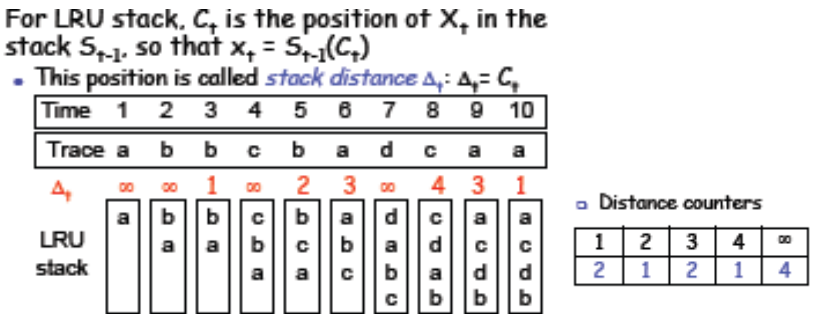

Figure 3: Stack Distance calculation

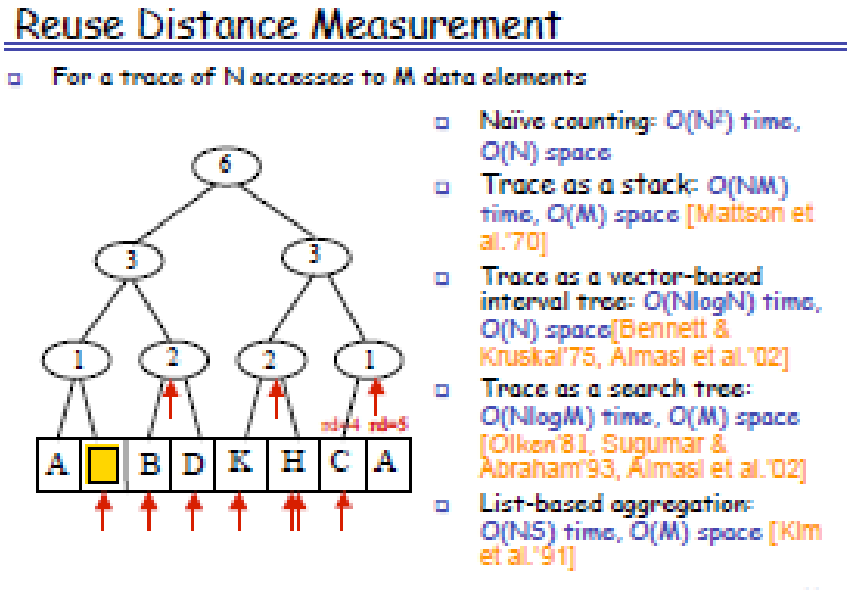

Figure 3: Complexity analysis