CS 740, Spring 2018
Homework Assignment 1

Assigned: Thursday, January 18
Due: Tuesday, January 30, 9:00AM

The purpose of this assignment is to develop techniques for measuring code performance, to practice reasoning about low-level code optimization, and to develop your own performance analysis tool using binary instrumentation.

Policy

You will work in groups of two or three people in solving the problems for this assignment. Turn in a single writeup per group, indicating all group members as indicated below.

Logistics

Any clarifications and revisions to the assignment will be posted on the class "assignments" web page.

To get started, download `assignment1-handout.tar` from the course webpage to a directory accessible only to your team. We recommend using AFS space and the cluster machines `ghc01.ghc.andrew.cmu.edu` – `ghc50.ghc.andrew.cmu.edu` (you can log in with your Andrew credentials). In the following, *ASSTDIR* refers to the directory that is unpacked with `tar xvf assignment1-handout.tar` .

When you are ready to hand in your solution, upload it to Gradescope. Your submission should be a `.tar` file consisting of the following:

1. `writeup.txt or writeup.pdf`, your writeup in plain text or PDF.
2. `func_time.c` for Problem 3
3. `cachemiss.c` for the **Cache Misses** portion of Problem 5
4. `strcat-x64-annotated.dis` for Problem 6
5. a folder called `pintool`, your cache profiling tool for Problem 8

To make your submission, do a `make submit`.

Please hand in your assignment using Gradescope. You may submit as many times as you would like. This is a long assignment—start early, and note that the last question is worth 45 points. Finally, most questions involve both implementation and analysis. Do not disregard the analysis portions! They make up a considerable portion of the allotted points.

Using Interval Timers

Measuring performance is fundamental to the study of computer systems. When comparing machines, or when optimizing code, it is often useful to measure the amount of time that it

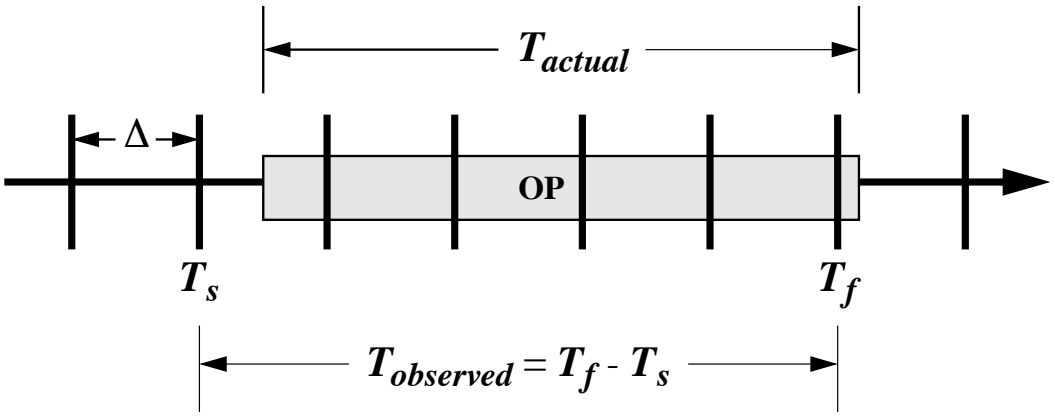

Figure 1: Time Measurement with an Interval Timer

takes (preferably at the resolution of processor clock cycles) to execute a particular operation or procedure. Some machines have special facilities to assist in measuring performance. Even without such facilities, almost all machines provide *interval timers*—a relatively crude method of computing elapsed times. In this assignment, you will investigate how to reason about and control the accuracy of timing information that can be gathered using interval timers. One of the goals is to develop a *function timer* which accurately measures the execution time of any function on any machine.

The overall operation of an interval timer is illustrated in Figure 1. The system maintains a (user-settable) counter value which is updated periodically. That is, once every Δ time units, the counter is incremented by Δ. Using the Unix library routine `getitimer`, the user can poll the value of this counter. Thus, to measure the elapsed time of some operation `Op`, the user can poll the counter to get a starting value T_s, perform the operation, and poll the counter to get a final value T_f. The elapsed time for the operation can be *approximated* as $T_{observed} = T_f - T_s$. As the figure illustrates, however, the actual elapsed time T_{actual} may differ from $T_{observed}$ significantly, due to the coarseness of the timer resolution. Since the value of Δ is around 10 *milliseconds* for most systems, this error can be very significant.

We have encapsulated the Unix interval timer routines for you in a as part of a package called *ASSTDIR*`/perf.c`. You should use this package for the measurements in the assignment. One notable feature is that it converts the measurements to units of seconds, expressed as a C `long double`. The procedure for timing operation `Op` is then:

```
init_etime();
Ts = get_etime();
Op;
Tf = get_etime()
T_observed = Tf - Ts;
```

See *ASSTDIR*`/example.c` for a simple example of how to use the interval timer. Note: This code has been tested to work on the general purpose Linux machines (`linux.andrew.cmu.edu` and `linux.gp.cs.cmu.edu`) using GCC.

Problem 1: Bounded Measurement Error (5 points)

Consider a processor with a 2 GHz clock rate where precisely one addition operation can be performed every clock cycle, and where the value of Δ for the interval timer is 10 milliseconds. You would like to time a section of code (`Op`) consisting purely of a sequence of back-to-back additions.

If your code sequence consists of 10^5 additions, what will the relative measurement error of $T_{observed}$ with respect to T_{actual} be? How about for 10^9 additions? As always, show all of your work.

Problem 2: Measuring Δ for Your Timer (10 points)

Write a C procedure that uses measurements to estimate (as accurately as possible) the value of Δ on any UNIX machine. Provide a listing of your code along with a brief description of your scheme.

We can improve the accuracy of the measurements by making sure that the activity we measure has sufficient duration to overcome the imprecision of interval timers. That is, we can accurately measure the time required by `Op` by executing it n times for a sufficiently large value of n:

```
init_etime();
Ts = get_etime();
for (i=0; i<n; i++) {
    Op;
}
Tf = get_etime()
T_aggregate = Tf - Ts;
T_average = T_aggregate/n;
```

How do we choose a large enough value of n? The idea is that n must be large enough such that $T_{aggregate}$ is larger than the minimum value ($T_{threshold}$) which guarantees a relative measurement error less than the desired upper bound of E. The value of $T_{threshold}$ can be computed based on Δ and E. However, since the elapsed time for `Op` is unknown, we cannot compute the minimum value of n ahead of time.

One approach is to start with $n = 1$, and continue doubling it until the observed $T_{aggregate}$ is large enough to guarantee sufficient accuracy (i.e. it is larger than $T_{threshold}$).

Problem 3: Implementing a Function Timer (5 points)

Implement a function timer in C that uses the doubling scheme outlined above to accurately measure the running time of any function on any system. Your function timer should have the following interface

```
typdef void (*test_funct)(void);
double func_time(test_funct P, double E);
```

where P is the function to be timed and E is the maximum relative measurement error. These prototypes are already defined for you in *ASSTDIR*/`func_time.h`. Implement your `func_time()` function in a separate file called `func_time.c`.

Your function timer should: (1) determine the timer period Δ using the scheme from the previous problem; (2) calculate $T_{threshold}$ as a function of Δ and E; and then (3) repeatedly double n until $T_{aggregate} \geq T_{threshold}$. It should work for any function on any system, regardless of the running time of the function or the timer period of the system.

Problem 4: Testing Your Function Timer (5 points)

Test your function timer using the program *ASSTDIR*/`freq.c`, which uses `func_time()` to estimate the clock frequency of your machine. This routine assumes that your machine executes an integer addition in one clock cycle. This is a safe assumption for most modern processors.

Turn in the output string from *freq.c* and the type of system you ran it on.

Problem 5: Using Hardware Counters (10 points)

Modern CPUs provide a variety of counters that enable us to get more data when measuring performance. Here we will experiment with timing and cache miss instrumentation.

Time

Another way to improve the accuracy of our measurements is to use a more precise timer. In addition to the interval timer (`get_etime()`), *ASSTDIR*/`perf.c` provides a similar hardware-based timer: `get_etime_hw()`.

Modify your `func_time.c` to include a function

```
double func_time_hw(test_funct P, double E);
```

that uses `get_etime_hw()`. Since the hardware counter resolution is smaller, you may find it helpful to measure $k\Delta$ and divide by k for $k \gg 1$ to obtain a useful Δ.

Cache Misses

We saw in class that cache interactions can influence performance. *ASSTDIR*/`perf.c` contains functions `start_cachemiss_count` and `get_cachemiss_count` that can be used as follows:

```
start_cachemiss_count();
Ops;
misses = get_cachemiss_count();
```

Write two blocks of code that perform the same number of loads and stores, but that produce different numbers of cache misses. Explain why you think they will behave differently, and measure cache misses for each with `start_cachemiss_count` and `get_cachemiss_count`, reporting the mean for each over 10 runs.

Optimizing the `strcat()` Routine

Our interest is in being able to understand and measure the machine code generated by a compiler. This is a far more useful skill than being able to churn out pages of assembly code by hand. Parts of this assignment involve compiling, disassembling, and running x86 code. In the next several problems, we will be focusing on the performance of the `strcat()` routine, which is

part of the C library. The following paraphrased excerpts from the `strcat()` man page describe its interface and behavior:

```
char *strcat(char *dest, const char *src);
```

- The `strcat()` function appends the `src` string to the `dest` string, overwriting the '\0' character at the end of `dest`. A pointer to the resulting string, `dest`, is returned.
- The `src` and `dest` strings must not overlap, and the `dest` string must have enough space for the result.
- If you pass an out of bounds or NULL pointer to strcat, the function generates a segmentation violation.
- There are no return values reserved to indicate an error.

The file *ASSTDIR*/`strcat_naive.c` contains a straightforward (but naive, from a performance perspective) implementation of `strcat()` in C called "`my_strcat()`". The file *ASSTDIR*/`strcat_naive.s` contains the x86 assembly code generated using the command: `gcc -O -S strcat_naive.c`

The file *ASSTDIR*/`strcat-x64.dis` contains a disassembled version of the `strcat()` routine taken from the Unix library `/lib64/libc.so.6` on an x64 machine. (This was disassembled with `objdump`.)

Problem 6: Understanding the `strcat()` Assembly Code (10 points)

Generate an "annotated" version of both *ASSTDIR*/`strcat_naive.s` and *ASSTDIR*/`strcat-x64.dis` using the following conventions:

- Put comments at the top of a code segment describing register usage and initial conditions.
- Put comments along the right hand side describing what each instruction does.

NOTE: Comments of the form:

```
# The following 2 instructions use registers eax, ecx, edx.
add    %ecx,   %edx   #  edx = edx + ecx
mov    (%eax), %ecx   #  ecx = Mem[eax]
```

are useless and will receive little (if any) credit. Instead, we would like to see comments like the following:

```
# Throughout the loop: eax holds i, ecx holds n
# At the beginning of the loop: edx = &v[0]
add    $1,  %eax              # i = i + 1
mov    (%edx, %eax, 4), %ecx  # n = v[i]
```

In other words, your comments should convey semantic information from the source code, and not simply reiterate what would be obvious to anyone who can read x86 assembly code.

Note: for full credit, you must identify the goal of the operations involving `0xfefefefefefefeff`. For extra credit (5 points), you may explain how it works, including the significance of each operation.

Problem 7: Measuring the Performance of the `strcat()` Routines (10 points)

Use the performance code you have written above to instrument both the `strcat_naive()` routine in *ASSTDIR*`/strcat_naive.c` and C library implementation of `strcat()` on the various `strcat()` calls contained in *ASSTDIR*`/strcat_test.c`. For each call, produce the following:

1. Time as measured by your interval timer code
2. Time as measured by the hardware timer code
3. Cache misses

Note that you should produce separate timing numbers for each of these individual calls to `strcat()`, and be sure to call the initialization routine in this file before you start timing things to ensure that the cache is warm.

Discuss the relative performance differences between the two versions of the routine, and whether they make sense given your analysis of the assembly code.

Problem 8: Writing Your Own Performance Analysis Tool using `Pin` (45 points)

Dynamic binary instrumentation (DBI) is a powerful technique for writing program analysis tools. DBI works by rewriting an executable on-the-fly to insert instrumentation code. DBI infrastructures also provide an interface for specifying user code (i.e. a tool) to be invoked as the program executes, as well exactly where and when this code should be invoked.

In this assignment, you will be using `Pin` (a DBI infrastructure for x86) to write your own tool for analyzing cache performance. `Pin` (`http://www.pintool.org`) is a publicly available (but not open source) tool developed by Intel. There is a nice tutorial on how to use `Pin` [1]; we highly recommend you read at least the first half. There are also a number of example tools in the `Pin` distribution.

Your goal is to develop (and use) your own tool to analyze cache performance. The goal of this tool is not simply to report overall cache misses, but to help identify *which* memory references (and which dynamic instances of those instructions) are responsible for causing the most cache misses.

While the `Pin` distribution already includes a cache analysis tool, that tool is overkill for our purposes in terms of the sophistication of its cache model, and it is also lacking some key functionality that we would like for you to implement for the sake of understanding *when* cache misses occur. Hence we would like you to write your own tool from scratch. (You are free to look at the existing tool, but you are better off starting with a clean slate, given how little of that code you will want to reuse.)

Regarding your cache model, we would like you to implement the following:

[1] `https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html/`

- Single-level "split" (i.e. separate) instruction and data caches, such that all instruction references go to the instruction cache, and all data references go to the data cache. (Note that a realistic cache hierarchy would have multiple levels of cache, but we are only asking you to model a single level in this assignment.)
- The *cache size*, *line size*, *cache miss penalty* and *associativity* should be parameters to your simulator. Assume that the configuration of both the instruction and data caches is the same. These are the only cache parameters that you need to support. It is safe to assume that cache size and line size are powers of two.
- For the associativity parameter, you only need to support direct-mapped and 2-way set associative. If a cache is 2-way set associative configuration, you should implement a least-recently-used (LRU) replacement policy within each set.
- Assume the following regarding the time that it takes the processor to execute each instruction. Ignoring cache misses, the normal execution of each instruction takes 1 cycle. In addition, if a given instruction suffers either an instruction cache miss or a data cache miss (it is possible for one instruction to suffer either or both of these types of cache misses), then the processor suffers an additional M cycles per cache miss. Hence an instruction that suffers no cache misses executes in 1 cycle, an instruction that suffers an instruction cache miss but not a data cache miss (or vice versa) executes in $1 + M$ cycles, and an instruction that suffers *both* instruction and data cache misses executes in $1 + 2M$ cycles. Assume that the processor executes only one instruction at a time, and that none of these times are overlapped with the execution times of other instructions.
- An instruction may have multiple memory reads and/or a memory write. Write your simulator to service the first read, then the second read, and then the write (of course, omitting the operations of this sequence that do not occur).

You will be recording not only the total cache misses for the instruction and data caches, but also a profile of the cache behavior for individual instructions and data references. Regarding the output of your tool, you should present summary statistics for each cache as well as a rank ordering of the most significant data references and instruction references according to their contribution to absolute misses for that particular cache. At minimum, your tools should present the information illustrated in Figure 2 for each entry in this rank-ordered table, including the program counter (PC) value of the given instruction. Given the rank-ordered cache miss profile illustrated in Figure 2, you could look up the PC values in disassembled code to match these behaviors back to the application source code.

There are a number of machines available with Pin installed for building and running this portion of the assignment: log in to any of `ghc01.ghc.andrew.cmu.edu` – `ghc50.ghc.andrew.cmu.edu` with your Andrew credentials. To build your Pintool, run

```
make obj-intel64/pin_cache.so
```

in the `pintool` directory. To run your Pintool, run

```
PIN_ROOT=/afs/cs.cmu.edu/academic/class/15740-s18/public/pin-3.0
$PIN_ROOT/pin -t obj-intel64/pin_cache.so [args...] -- binary
```

where *binary* is the program you wish to instrument. You may want to export `PIN_ROOT` from your `.bash_profile`.

```
Overall Performance Breakdown:
==============================
  Instruction Execution:     2724M cycles (  4.2%)
  Data Cache Stalls:        40700M cycles ( 63.5%)
  Instruction Cache Stalls: 20700M cycles ( 32.3%)
  ------------------------------------------------
  Total Execution Time:     64124M cycles (100.0%)

Data Cache:
===========
 Configuration: size = 64KB, line size = 32B, associativity = 2-way,
      miss latency = 100 cycles

 Overall Performance: 1324M References, 407M Misses, Miss Rate = 30.7%,
    Data Cache Stalls = 40700M cycles

 Rank ordering of data references by absolute miss cycles:
                                                 Total  Contribution
                                          Miss   Miss   to Total Data
        PC       Type References Misses   Rate Cycles Miss Cycles
     ----------  ---- ---------- ------  ----- ------ ------------
  1. 0x47601208  Load   201.7M    53.1M  26.3%  5310M    13.0%
  2. 0x4769148c  Store  349.2M    46.5M  13.3%  4650M    11.4%
  3. 0x476327c0  Load    71.0M    39.2M  55.2%  3920M     9.6%
  4. 0x47842074  Load   101.2M    32.8M  32.4%  3280M     8.1%
        ...       ...     ...      ...    ...    ...      ...
 20. 0x47832148  Store   68.2M     5.3M   7.8%   530M     1.3%

Instruction Cache:
==================
 Configuration: size = 64KB, line size = 32B, associativity = 2-way
      miss latency = 100 cycles

 Overall Performance: 2724M References, 207M Misses, Miss Rate = 7.6%,
    Inst Cache Stalls = 20700M cycles

 Rank ordering of instruction references by absolute miss cycles:
                                      Total  Contribution
                                Miss  Miss   to Total Inst
        PC      References Misses  Rate Cycles Miss Cycles
     ---------- ---------- ------ ----- ------ ------------
  1. 0x41621378   171.7M   88.1M  51.3%  8810M   42.6%
  2. 0x41486910    43.2M   31.7M  73.4%  3170M   15.3%
        ...          ...     ...   ...     ...    ...
```

Figure 2: Example of output from the initial cache miss profiling tool.

Your mission is the following:

Part 1: Build a Pin-based cache profiling tool (from scratch) that can generate output as illustrated in Figure 2 (your output does not have be perfectly aligned, but it must be readable, e.g., columns clearly delineated). Using micro-benchmarks (i.e. small pathological programs that you write yourself), `start_cachemiss_count`/`get_cachemiss_count`,

and possibly the output from other cache simulators, verify that it is working correctly. Describe the process that you went through to do this, and show your micro-benchmarks along with your analysis of their behavior on your cache profiling tool.

Part 2: Run your Pin-based cache profiling tool on the test programs in the directory *ASSTDIR*/`pintool/cache_test` using the four configurations shown in Table 1. Show the results of your tool for each of these four configurations. Discuss how the differences between successive configurations affect performance, and whether there are any surprises regarding how the profile of important cache misses changes, etc.

Table 1: Configurations to use when profiling the test programs.

Parameter	Configuration 1	Configuration 2	Configuration 3	Configuration 4
Cache size	8 KB	8 KB	8 KB	32 KB
Line size	64 B	64 B	128 B	128 B
Cache Miss penalty	100 cycles	100 cycles	100 cycles	100 cycles
Associativity	Direct-Mapped	2-way Set Assoc.	2-way Set Assoc.	2-way Set Assoc.