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 three people in solving the problems for this assignment. (A group of two may be necessary depending on the class size—groups of one are definitely not allowed.) Turn in a single writeup per group, indicating all group members.

Logistics

Any clarifications and revisions to the assignment will be posted on the “assignments” web page on the class WWW directory.

In the following, HOMEDIR refers to the directory:

/afs/cs.cmu.edu/academic/class/15740-f09/public

and ASSTDIR refers to the subdirectory HOMEDIR/asst/asst1.

Please hand in your assignment as a hard copy of formatted text.

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 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 $\Delta$ time units, the counter is incremented by $\Delta$. 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 $\Delta$ 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 handy timer package called `ASSTDIR/etime.c`. You should use this package for all measurements in the assignment. See `ASSTDIR/example.c` for a simple example of how to use the package. 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:

```c
init_et ime();
Ts = get_et ime();
Op;
Tf = get_et ime();
T_observed = Tf - Ts;
```

Note: This code has been tested to work on the andrew linux machines. In particular, code must be compiled with gcc 3.4 to work correctly.

**Problem 1: Bounded Measurement Error**

Consider a processor with a 500 MHz clock rate where precisely one addition operation can be performed every clock cycle, and where the value of $\Delta$ 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 $\Delta$ for Your Timer**

Write a C procedure that uses measurements to estimate (as accurately as possible) the value of $\Delta$ 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 \(n\):

```c
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 \(\Delta\) 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**

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

```c
typedef 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 \(\Delta\) using the scheme from the previous problem; (2) calculate \(T_{threshold}\) as a function of \(\Delta\) 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**

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: Alternative Timer Algorithms**

Recall that in Problem 3, you repeatedly doubled the value of \(n\) until it was sufficiently large (i.e. until \(T_{aggregate}\geq T_{threshold}\)). Now consider the following three algorithms for increasing the value of \(n\):

Algorithm 1: Set $n = 1$ initially, and repeatedly multiply $n$ by a factor of 2 until $n$ is sufficiently large (i.e. the algorithm used in Problem 3).

Algorithm 2: Set $n = 1$ initially, and repeatedly multiply $n$ by a factor of 7 until $n$ is sufficiently large.

Algorithm 3: Set $n = 1$ initially, and repeatedly add (not multiply!) 50 to $n$ until $n$ is sufficiently large.

Your goal is to minimize the total amount of time that your timing routine takes to accurately time a function. In this problem, you will evaluate the three algorithms described above based on this criteria.

Part 1: Assuming that $T_{\text{threshold}} = 75$ milliseconds, and assuming that the timing loop surrounding $\text{Op}$ involves zero overhead, compute how long it would take for each of the three algorithms for increasing $n$ to accurately time $\text{Op}$ for each of the following three cases: 
(i) $T_{\text{actual}} = 15.0$ milliseconds, (ii) $T_{\text{actual}} = 250.0$ microseconds, and (iii) $T_{\text{actual}} = 200.0$ microseconds.

Part 2: Based on a quantitative analysis of their worst-case behaviors, evaluate which of the three algorithms for increasing $n$ is most desirable for measuring functions where $T_{\text{actual}}$ is an arbitrary value no greater than a microsecond and where $\Delta$ for the interval timer is at least 10 milliseconds.

---

Optimizing the strchr() Routine

The purpose of these next problems is to get hands-on experience with machine-level programming. Our interest is in being able to understand, measure, and optimize 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 strchr() routine, which is part of the C library.

The following list summarizes the interface and behavior of strchr().

```c
char * strchr(const char *s, int c);
```

- The strchr() function returns the first occurrence of character $c$ in string $s$. The ending ‘\0’ is considered to be the final part of the string.
- The function returns NULL, if the character $c$ does not occur in string $s$.
- If you pass an out of bounds or NULL pointer to strchr(), the function generates a segmentation violation.

The file ASSTDIR/strchr_naive.c contains a straightforward (but naive, from a performance perspective) implementation of strchr() in C called “my_strchr()”. The file ASSTDIR/strchr_naive.s contains the x86 assembly code generated using the command:
gcc -O -S strchr_naive.c
The file `ASSTDIR/strchr.asm` contains a disassembled version of the `strchr()` routine taken from the Unix library `/usr/lib/libc.a` on an x86 machine. (This was disassembled with `objdump`.)

**Problem 6: Understanding the `strchr()` Assembly Code**

Generate an “annotated” version of both `ASSTDIR/strchr_naive.s` and `ASSTDIR/strchr.asm` 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  # ecx = 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.

---

**Problem 7: Measuring the Performance of the `strchr()` Routines**

Use your interval timer code to measure the performance of both the `my_strchr()` routine in `ASSTDIR/strchr_naive.c` and C library implementation of `strchr()` on the various `strchr()` calls contained in `ASSTDIR/strchr_test.c`. Note that you should produce separate timing numbers for each of these individual calls to `strchr()`, 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: Implementing a Better Version of `strchr()` in C**

Write your own version of `strchr()` in C. Your code must behave correctly, but at the same time it should be as efficient as possible. You should create a version of your code which only uses C constructs (i.e. no explicit assembly code). In addition, you may optionally create a second version of your code which uses the GCC assembly code directives (i.e. “ASM”) if it further enhances performance. For further information on how to use assembly code directives in gcc, see the “info” pages on gcc (under “C extensions”). These info pages are reproduced on the
Assignment and exam information class web page under Assignment 1. Use a minimal number of ASM statements—do not simply reproduce large amounts of hand-coded assembly in your C code. Be sure to compile your code using the “-O” optimization flag.

Measure the performance of your C-only code and your assembly-augmented code (if applicable). If your assembly-augmented code achieves better performance than your C-only code, discuss why you are not able to achieve comparable performance using only normal C constructs. Also, compare your code with both the naive and UNIX library versions of strchr(). If your performance falls short of the UNIX library version, explain why.

Problem 9: Writing Your Own Performance Analysis Tool using Pin

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 is a publicly available tool (developed by Intel), which you can access at the following web site: http://rogue.colorado.edu/pin. There is a nice tutorial on how to use Pin on the web site, and there are a number of example tools in the Pin distribution.

Your goal is to develop (and use) your own tool to analyze cache performance. This tool will not only report aggregate numbers such as total cache misses, total execution cycles, etc., but it will also produce a rank-ordered profile of which specific instructions in the binary (identified by their instruction address, which is also called the program counter or “PC” value when they execute) were most costly in terms of cache miss stall times.

As you complete all aspects of this problem (described in more detail below), you will be creating a cache analysis tool that can be configured to run in one of three different modes:

1. without prefetching;
2. with prefetching and infinite memory bandwidth;
3. with prefetching and finite memory bandwidth.

While the Pin distribution already includes a cache analysis tool, that tool is overkill in many respects and also lacking key functionality in other respects (e.g., it doesn’t implement the style of prefetching and profiling that we would like you to implement). 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.)

Next-\(k\)-line Prefetching: The specific style of prefetching that we would like you to implement is hardware-based, next-\(k\)-line prefetching. This is an extremely simple prefetching scheme that attempts to exploit spatial locality beyond cache block boundaries. This approach works as follows: when the processor suffers a cache miss while accessing a cache block at address \(A\), the hardware will not only fetch block \(A\) from memory to satisfy the cache miss, it will also launch prefetches of the next \(k\) contiguous cache blocks in the virtual address space. For example, assuming that \(k = 4\) and the cache block size is 32 bytes, if the processor suffered a miss while performing a read of address \(0x80012352\), the hardware would not only fetch the block starting

6
at address 0x80012340 to satisfy that cache miss, it would also prefetch the blocks starting at addresses 0x80012360, 0x80012380, 0x800123a0, and 0x800123c0. (The exact timing of these prefetches will be discussed later in further detail.)

**Cache Configuration Assumptions:** To model the basic functionality of your caches, please implement and assume the following:

- You should model only a single level of “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, and associativity should be parameters to your simulator. Assume the same configuration for both the instruction and data caches. It is safe to assume that cache size and line size are powers of two. For associativity, you only need to model direct-mapped and 2-way set-associative caches.

- If a 2-way set-associative cache is chosen, you should implement a least-recently-used (LRU) replacement policy within each set.

**Timing Model Assumptions:** In addition to modeling cache misses, your tool must keep track of overall execution time. Timing is important not only to understand overall performance, but also to model the timeliness of prefetches (especially in the finite memory bandwidth case). To help simplify your task, we have chosen an extremely simple timing model. (Timing is far more complex than this in realistic cache simulators.) Please implement and assume the following regarding your timing model:

- The execution time of each dynamic instruction is the sum of three components: (i) instruction processing time, (ii) instruction cache stall time, and (iii) data cache stall time.

  **Instruction processing time:** assume that this is one cycle for each x86 instruction. (Note that this is not realistic, but it will simplify your implementation.)

  **Instruction cache stall times:** any additional stalls beyond the normal hit time when fetching the instruction from the instruction cache. (Note that all instructions reference the instruction cache.)

  **Data cache stall times:** any additional stalls beyond the normal hit time when fetching data from the data cache. (Note that only instructions that load or store memory need to access the data cache.)

Note that instructions can potentially suffer cache misses in both the instruction and data caches. Hence an instruction that suffers a 100 cycle stall due to an instruction cache miss plus a 100 cycle stall due to a data cache miss would take a total of 201 cycles to execute (including the additional 1 cycle for normal instruction processing).

- While the currently executing instruction suffers a cache miss, the processor pipeline halts until the miss completes. In other words, it does not attempt to “run ahead” in any way (which is something that modern pipelines do). In particular, if the processor suffers an instruction cache miss, it waits until this completes before starting an access to the data cache. The processor does not move on to the next instruction until the instruction access (which can take zero or more cycles), the data access (also zero or more cycles), and the normal instruction processing time (always 1 cycle) have all completed.
• Ignore writebacks of dirty data. (Assume that they happen instantaneously.)

• One parameter to your simulator is its prefetching mode, which can be set to either (i) no prefetching, (ii) prefetching with infinite bandwidth, or (iii) prefetching with finite bandwidth. If one of the prefetching modes is specified, the value of \( k \) for the next-\( k \)-line prefetching should also be specified. (By default, set \( k \) to 3.)

• Two additional parameters will be used to specify the timing of cache misses and prefetches:

  **Cache Miss Latency** (\( L \)): the number of cycles that a cache miss will normally take to complete. Use this same parameter for both instruction and data cache misses. (By default, set this parameter to 100 cycles.) In the absence of queueing delays due to limited memory bandwidth, this will be equivalent to the cache miss stall time. Hence for the no prefetching and prefetching with infinite bandwidth models, each cache miss will take exactly \( L \) cycles.

  **Memory Repeat Rate** (\( R \)): this parameter will be used in the prefetching with finite bandwidth model to account for limited throughput. Use this same parameter for both instruction and data cache misses. (By default, set this parameter to 20 cycles.)

• **Timing under the No Prefetching Model:** Because there is no concurrency of any form in this very simple model (i.e. the pipeline stalls upon misses, and there are never any concurrent accesses within the memory system), it is possible to compute the total cache stall time by multiplying the number of cache misses by the cache miss latency (\( L \)) at the end of the simulation. (Don’t forget to include the instruction processing time in overall execution time.)

• **Timing under the Prefetching with Infinite Bandwidth Model:** Under this model, when a cache miss occurs due to the processor accessing either an instruction or data (we call these “demand misses”), the hardware launches prefetched for the next \( k \) cache lines in parallel with the demand miss. Due to our (unrealistic) assumption of infinite memory bandwidth, all of these memory accesses complete simultaneously. Because the processor was stalled during this time, any prefetches will have completed before the processor can accessed prefetched lines. Hence the timing remains very simple to model.

• **Timing under the Prefetching with Finite Bandwidth Model:** The timing in this case is more interesting, and it does require that you carefully track time as the simulation proceeds. When a demand miss occurs, the hardware launches prefetches for the next \( k \) cache lines. However, because of the memory module’s limited throughput, the prefetches are only partially overlapped with the demand miss and with each other. If the demand miss begins at time \( T \), the demand miss will complete at time \( T + L \), the first of \( k \) prefetches will complete at time \( T + L + R \), the second prefetched line will complete at time \( T + L + 2R \), the third will complete at time \( T + L + 3R \), etc. This creates some interesting complications in terms of timing when the processor resumes execution and attempts to access memory again while prefetches are still in flight:

  **Scenario 1:** processor accesses location \( X \) which hits in the cache: Since \( X \) is already in the cache (it does not match any outstanding prefetches), this hit can proceed as normal, despite the fact that the memory system is still busy processing prefetches for other locations.

  **Scenario 2:** processor accesses location \( X \) which misses in the cache, but matches an outstanding prefetch: In this case the processor should stall until
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, prefetch repeat rate = 20 cycles

Overall Performance: 1324M References, 407M Misses, Miss Rate = 30.7%, Data Cache Stalls = 40700M cycles, Average Miss Latency = 100 cycles

Rank ordering of data references by absolute miss cycles:

<table>
<thead>
<tr>
<th>PC</th>
<th>Type</th>
<th>References</th>
<th>Misses</th>
<th>Rate</th>
<th>Miss Cycles</th>
<th>Miss Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 0x47601208 Load</td>
<td>201.7M</td>
<td>53.1M</td>
<td>26.3%</td>
<td>5310M</td>
<td>100.0</td>
<td>13.0%</td>
</tr>
<tr>
<td>2. 0x4769148c Store</td>
<td>349.2M</td>
<td>46.5M</td>
<td>13.3%</td>
<td>4650M</td>
<td>100.0</td>
<td>11.4%</td>
</tr>
<tr>
<td>3. 0x476327c0 Load</td>
<td>71.0M</td>
<td>39.2M</td>
<td>55.2%</td>
<td>3920M</td>
<td>100.0</td>
<td>9.6%</td>
</tr>
<tr>
<td>4. 0x47842074 Load</td>
<td>101.2M</td>
<td>32.8M</td>
<td>32.4%</td>
<td>3280M</td>
<td>100.0</td>
<td>8.1%</td>
</tr>
</tbody>
</table>
| ... ... ... ... ... ... ... ... ... ... ... ... ... ...
| 20. 0x47832148 Store | 68.2M | 5.3M | 7.8% | 530M | 100.0 | 1.3% |

Instruction Cache:
==================
Configuration: size = 64KB, line size = 32B, associativity = 2-way, miss latency = 100 cycles, prefetch repeat rate = 20 cycles

Overall Performance: 2724M References, 207M Misses, Miss Rate = 7.6%, Inst Cache Stalls = 20700M cycles, Average Miss Latency = 100 cycles

Rank ordering of instruction references by absolute miss cycles:

<table>
<thead>
<tr>
<th>PC</th>
<th>References</th>
<th>Misses</th>
<th>Rate</th>
<th>Miss Cycles</th>
<th>Miss Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 0x41621378</td>
<td>171.7M</td>
<td>88.1M</td>
<td>51.3%</td>
<td>8810M</td>
<td>100.0</td>
</tr>
<tr>
<td>2. 0x41486910</td>
<td>43.2M</td>
<td>31.7M</td>
<td>73.4%</td>
<td>3170M</td>
<td>100.0</td>
</tr>
</tbody>
</table>
| ... ... ... ... ... ... ... ... ... ... ... ... ... ...

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

the corresponding prefetch completes, at which point it can resume execution (even if other prefetches are still in flight).

**Scenario 3:** processor accesses location X which misses in the cache, there are outstanding prefetches, but none of them match X: In this case the prefetches are not helpful to the given demand miss, but we cannot preempt the prefetches, so we need to wait until the memory resource becomes available. If this occurs at time
$T$, then the demand miss will complete at the maximum of either (i) $T + L$ cycles or
(ii) $R$ plus the completion time of the last outstanding prefetch.

- Note that demand misses do not preempt outstanding prefetches; the memory system
  handles requests in a first-come, first-served fashion.

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.

Your mission is the following:

**Part 1:** Build a Pin-based cache analysis tool (from scratch) that can generate output as il-
lustrated in Figure 2. Using micro-benchmarks (i.e. small pathological programs that
you write yourself) and possibly the output from other cache simulators, verify that it is
working correctly.

**Part 2:** Extend your cache analysis tool from Part 1 to implement the next-$k$-line prefetching
with infinite memory bandwidth.

**Part 3:** Extend your cache analysis tool from Part 1 to implement the next-$k$-line prefetching
with finite memory bandwidth.

**Part 4:** We have distributed four test programs for your cache profiling tool, in the folder
ASSTDIR\cache_test.

Compare the performance of no-hardware prefetching (tool from Part 1) with infinite-
bandwidth hardware prefetching (tool from Part 2) and finite-bandwidth hardware
prefetching (tool from Part 3) on these 4 programs, as well as at least 2 others you have
written. (Do not just trivially modify the test programs we gave you.)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Typical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache size</td>
<td>8-32 KB</td>
</tr>
<tr>
<td>Line size</td>
<td>64-128B</td>
</tr>
<tr>
<td>Cache miss penalty</td>
<td>20, 50, 100, 200 cycles</td>
</tr>
<tr>
<td>$K$ (prefetcher parameter)</td>
<td>1,5,10</td>
</tr>
<tr>
<td>Associativity</td>
<td>1 or 2 way</td>
</tr>
</tbody>
</table>

A table of typical values for the parameters in your experiments is above. Be sure to run
experiments with at least two different values of $k$ chosen, as well as at least 3 different
cache miss penalties, and varying set associativity, chosen from the table of typical values.

Analyze the effects on performance. In your analysis, be sure to answer the following
questions:

1. Is performance getting better or worse? Quantify these effects (e.g., measure queue
   lengths, average miss penalty, etc.).
2. Can any of these effects be explained by the source code of the test programs?
3. How does varying \{\text{associativity, cache miss latency, k}\} affect your prefetcher?
4. Based on your experiments, what would you suggest as a more effective hardware prefetcher design?