COMP206 **Class Project** Assigned: November 24, 2003
Prof. Montek Singh Due: December 10, 2003

Instructions:

1. You may work on this project either individually, or in teams of two. If you work in a group, only one written report should be submitted, with names of both members of the team.

2. Please keep your report short — five pages should suffice (*e.g.,* 2–3 pages of plots, and a couple of pages of written material). Extra long reports may even invite penalties.

3. This project consists of two parts. Part I deals with branch prediction. Part II deals with caching.

4. Each part has several questions, some of which are marked *"Extra Credit."*

   - If you are working alone on this project, these extra credit questions are completely optional for you, but you will receive extra credit if you decide to attempt them.[1]
   - However, *if you are working in a team, you must attempt all of the extra credit questions; i.e.,* all questions are compulsory and will count towards your grade on the project.

[1]Extra credit earned on the project will be kept separate from your grade on this project, but will help you partly compensate for a not-so-good performance on homework assigments or exams.

1. **(PART I)** [60 points] This part deals with branch prediction. Consider the C program provided on the class website, reproduced here for your convenience:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define hrtime_t long long
hrtime_t _time_start, _time_end;
#define tick()  (_time_start = gethrvtime())
#define tock()  (_time_end   = gethrvtime())
#define elapsed_nanoseconds()  (_time_end - _time_start)

#define SAMPLES 10000
#define RANGE 400
#define THRESH 80

int array[SAMPLES];

int main()
{
  int i, j, k, sum;

  for (k = 0; k < RANGE; k++) {
    for (i = 0; i < SAMPLES; i++) array[i] = k*drand48(); /* Init loop */

    tick();

    for (sum = i = 0; i < SAMPLES; i++) { /* Loop-back branch */
      j = array[i];
      if (j >= THRESH) sum &= THRESH; /* Conditional branch */
      sum += j;
    }

    tock();

    fprintf(stderr, "%d", sum);
    printf("%5d \t%lld\n", k, elapsed_nanoseconds());
  }
  return 0;
}
```

Explanation:

- The loop marked `/* Init loop */` initializes the array with uniformly distributed random numbers in the range $[0, k)$.

- The system call `gethrvtime()` returns the time elapsed since the start of the process. The "hr" in the function name stands for "high resolution" time, reported in nanoseconds. The "v" stands for "virtual" time, as opposed to real time, *i.e.,* only counting the CPU cycles actually expended in executing this particular process.

(a) Briefly explain what the program does, and how.

(b) Quantify the predictability of the branch marked `/* Conditional branch */` as a function of k and THRESH. That is, derive an expression that relates predictability to the values of k and THRESH. Bear in mind that predictability means either the branch is easily predicted to be not taken, or is easily predicted to be taken.

(c) Note that there are two branches in the code—the loop-back branch and the conditional branch—and that they alternate in time. Will the use of a correlating predictor help? If yes, which values of m and n would you use for an (m, n) correlating predictor? Explain your answer.

(d) Compile the code on the following Sun SPARC machine: swift.cs.unc.edu.[2] Use *gcc* as your compiler, and generate three different executable by selecting the following optimization switches in *gcc*:

i. gcc -O0 proj.c
ii. gcc -O1 proj.c
iii. gcc -O3 proj.c

Which optimizations are selected by each of these command-line options? You may refer to any online documentation for the gcc compiler (*e.g.,* `http://www.dis.com/gnu/gcc/Optimize-Options.html`). Identify only 2–3 optimizations which you can recognize from our discussions in class. Please be brief.

(e) Why is the value of `sum` printed on `stderr` when it is actually not needed to plot the runtimes? Hint: Think about what a smart compiler may do. Or, try to remove the `fprintf` and see what happens.

(f) Once you have generated the four different executables, run them using the *ptime* utility:

```
$ /usr/proc/bin/ptime a.out > out
```

Plot the program output, *e.g.,* by importing the data into a spread-sheet. Try to use a "scatter plot," *i.e.,* do not connect the points by lines; simply plot the points so that the plots look clean. *Hint:* This is easily done using Excel's "import data" function; once you set up the import function, you can simply refresh your data using a couple of keystrokes.

(g) Analyze the three plots generated above, and give brief explanations. Identify *all trends* that you can see in the plots and try to account for them by giving several plausible explanations/conjectures. *Important:* it is not important that your explanations of the trends be actually correct; they must simply be plausible explanations consistent with observed data.

[2] If you don't have access to the machine, please see the instructor.

(h) **(Extra Credit)** Modify the code to add at least one extra conditional branch. Be sure that the modified code will indeed have the extra branch, in addition to the branches already there, even after aggressive code optimization by gcc. That is, it should be non-trivial for the compiler to "optimize away" your new branch. Try to be creative in how you select the new branch. In particular, the characteristics of the new branch should be fairly different from those of the branches already present in the code, and you must be able to indentify these characteristics from the plots of the runtimes generated by the program. Hint: Refer to your answer to part (e).

2. **(PART II)** [40 points] This part deals with reducing cache misses using *blocking*.

   (a) Refer to pages 433–434 of Hennessy/Patterson 3rd ed. The textbook gives an example of how the matrix multiplication operation can be made to run faster by using the technique of blocking, which exploits temporal locality. Design an experiment with the same, or similar, program fragments and quantify the benefit of this optimization. *Important:* be sure to use matrix sizes large enough to demonstrate the benefits of blocking, *e.g.,* 1000x1000 or larger.

   (b) Generate plots of runtimes for different blocking factors.

   (c) Analyze the plots generated and briefly discuss what you observe.

   (d) **(Extra Credit)** What can you conclude about the size of the cache from your experiments? For this question, assume that there is a single level of cache (not actually true), and that the cache is fully associative (also probably not true). Simply give a guess-estimate of the total amount of effective cache capacity by merely looking at the plot of runtimes. Once again, your answer need not match reality, but simply be reasonably supported by the measurements.