University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 1997 D.A. Patterson
Handout #2

Assignment 1
Exercises Due Thursday 9/11/97 at 4PM in box in 283 Soda
CS152 Computer Architecture and Engineering

PLEASE put the TIME OR TA NAME of the DISCUSSION section that you attend as well as your name, as homeworks will be handed out in discussion. No section, no homework.

Homework Policy: Homework assignments are due at 4PM. No late homeworks will be accepted, as we want to be able to discuss the assignments in section. Study groups are encouraged, but what you turn in must be your own work. As we agreed on the first day of class, the penalty for cheating is no credit for the full assignment. (If things get tough, its better to skip pieces of one assignment rather than take a chance on getting no credit.)

Please do the following problems from P&H: 1.51-53, 2.3, 2.4, 2.10, 2.18, 2.20, 2.42.

Chapter 2 of P&H discusses various aspects of performance in abstract terms. This problem will give you a chance to get some hands-on experience with measuring performance on real machines.

One benchmark for scientific computing is Linpack. It measures the time required to solve a system of linear equations. We will use this benchmark to do our own performance measurements.

The results of this benchmark for a wide range of machines are updated periodically. You can find the report by following the Linpack link on the CS152 home page:
`http://www-inst.eecs.berkeley.edu/~cs152/`
or by using this URL to find the report at the Linpack folks site:
`http://performance.netlib.org/performance/html/PDSreports.html`.

As you would expect, the Linpack benchmark is an executable and its source files are located in `~cs152/fa97/hw1`. The internals of the benchmark consist of 2 distinct parts: the daxpy loop (`daxpy.c`) and the driver (`timer.c`).

The daxpy loop is the heart of the benchmark. This loop is what is being measured for performance. Basically, we are testing how fast a certain machine can execute this simple loop. Below is the source code for the daxpy loop.

```
void daxpy(double y[], double a, double x[], int n) {
    int i;

    for (i = 0; i < n; i++)
        y[i] = a*x[i] + y[i];
}
```

It simply multiplies a vector by a scalar and adds it to another vector. Since the vectors and scalars are of type double, this test primarily measures the machines ability to compute floating point numbers. Therefore, performance for this benchmark is reported in MFLOP/s.

Before we get into performance, we need to think a bit about performance measurement. There are a couple of issues here.

First, in an operating system such as Unix, it is possible for many processes to be active at the same time and we as users really don't have any control over that. For example, the operating system could be handling interrupts, I/O activity, background processes, and requests by other users on the same machine. All of these external events can take system resources away from our benchmark program and skew the measurement. Unfortunately, these demands on the system can change unexpectedly (or asynchronously). As a result of this limitation, it is not good enough to measure only a single execution of the daxpy function. It is better to run it several times and obtain the average execution time for a single daxpy function. In addition, we should monitor whether each run of the daxpy function is somewhat consistent with all the other runs. If it isn't very consistent, then we'll know that skewing due to external events is playing a major role. Fortunately for us, the driver already handles this for us.

The second problem has to do with the precision of our measuring device. Even for a sizable vector, the time required for a single execution of a daxpy function may be less than the resolution(precision) of a typical Unix timer. To overcome this problem, we must make sure that the amount of time we use to run the daxpy function is much greater than the resolution of the timer. We can achieve this by simply running the daxpy function more than once between subsequent calls to the Unix timing routine, `times()`. Once we get a time for the batch run, we can then get the time for a single execution of the daxpy function by dividing the batch time with the number of runs in the batch. Again, fortunately for us, the driver already takes care of this issue. It can execute the daxpy function multiple times between each timing and will report a warning if the resolution of the timer introduces an error of more than 5%.

Now on with the assignment...

Problem 1: daxpy internals

Copy the contents of `~cs152/fa97/hw1` to an appropriate directory on your account. When you run `gmake` in that directory, it will compile the `daxpy` with and without optimization for the machine you are on and produce the assembly language listings and place them all in a subdirectory `bin-ARCH`, where ARCH is the machine name.

The `daxpy` executables take 3 command line arguments:

po.eecs% daxpy *vectorSize iterations tests*

1. *vectorSize* - size of the vectors to be used in the daxpy function
2. *iterations* - the number of times the daxpy function should be executed between each timing (or sampling).
3. *tests* - the number of timing samples we should obtain before determining results

This question will help you to get familiar with the internals of daxpy. **You should examine the two source files `daxpy.c` and `timer.c` in answering the following questions!**

Assume that a machine delivers 10 MFLOP/s on DAXPY and its timer has a resolution of 1/60 seconds. Let V be the vector-size, N be the number of iterations, and T be the number of tests.

(a) How many floating point arithmetic operations are performed during a single iteration in the daxpy loop?

(b) How many floating point arithmetic operations are performed in a single execution of the daxpy function?

(c) Write a formula for the amount of time spent in executing N iterations of the daxpy function.

(d) If we want the error introduced by the timer to be less than 5 percent, how many iterations (N) of the daxpy function must be executed for a vector of 1000 elements? For a vector of 100,000 elements?

Problem 2: Measuring daxpy

Now let's get some actual measurements.

Machine name				
Machine brand				
Clock rate				
Memory size				
Compiler Cmd	**DAXPY**	**DAXPY Opt**	**DAXPY**	**DAXPY Opt**
Vector Size	1,000	1,000	100,000	100,000
Num Iterations				
Num Tests				
Time per element				
MFLOPS				
MIPS				
CPI				

(a) You will now run optimized and unoptimized daxpy on two computers with different instructions sets: one of the DEC 5000/25 in 199 Cory and one of the HP 9000/715 in 347 Soda. To see what computers are available, type the command "clients" at the Unix prompt. Fill out the form above for both machines. Choose suitable values for Num-Iterations and Num-tests.

(b) Give formulas you used to determine MFLOPS, MIPS, and CPI. In obtaining these formulas, you may have to look at the assembly files for each daxpy executable and count the number of instructions executed. In doing so, only concentrate on the portion of the assembly file which constitutes the DAXPY loop and nothing else. (Hint: time per element (in micro-seconds) is the time required to execute a single iteration of the DAXPY loop.)

Extra Credit: Feel free to measure other machines in Cory or Soda labs that might have different performance from those two above. Use the "clients" command to see which might be good candidates. If you want to try this on some other machines, you may have to make a few changes to the code. Specifically, if the routine `times()` is not available on your system, you will need to find a routine that provides the user time. Check the comments in the code for more details.

Generic Problem 3: Accounting

To give us feedback on the workload, approximately how many hours did you spend on this assignment? How long before the deadline did you finish this assignment?