CS 201 Section 1 Lab 4 – due 6pm, Wednesday, March 8, 2000

Experimental Determination of Cache Size

Write a program that, when run on a SPARC, determines and prints out the size of the processor's level one data cache (i.e., the cache closest to the processor), in bytes. Your program must do this by making relative timing measurements on different memory access patterns, and noting when access slows down markedly. Details of how to do this are described below.

You can assume the following characteristics of the level one data cache: it is distinct from the level one instruction cache; it is direct-mapped; its size is a multiple of 2; it is no larger than 2MB. Your program should be designed to work on *any* SPARC that has these characteristics. (Of course, you can only test it on a limited range of SPARCs, but you should be confident that it will work on others too.)

All the SPARCs listed in `http://www.cs.pdx.edu/~apt/cs201/sparc_names` do have these characteristics; in fact, they all have 16KB level one data caches. But note that the cache size varies among different SPARC models, and we may test your program on SPARCs not available to you, so you cannot just hard-code the answer! Also, there may be some system call that returns the cache size directly, but you may not use this either.

Your program should take less than a minute to run. It is ok for it to put out progress information as it runs, so long as its final line of output unambiguously shows the cache size it has determined.

Algorithm

The basic idea is to exploit an unfortunate characteristic of direct-mapped caches. Suppose we repeatedly alternate between reading at address a and address $a + s$. The first read at each address will probably cause a cache miss. If s is less than the cache size, then the first reads will store the contents of both addresses in the cache, so subsequent reads should be quite fast. But if s is exactly *equal* to the cache size (or is an integer multiple thereof), each read at $a + s$ will force out the cache entry for a, and vice-versa, so *every* read will cause a cache miss, and hence will be slow.

The idea, therefore, is to measure the time taken to do a large number of pairs of reads with $s = 0$, and then repeat while gradually increasing s (by powers of two) until the time suddenly increases. To make the times long enough to measure, we need to do each pair of reads *many* times (say, several million); this also eliminates any effects due to possible cache misses on the first pair of reads.

We can use the computer's own internal timer to do the timing measurements, and feed back the results to our program. Note that the computer's timer is imprecise, so that we are likely to get slightly different times even if we measure exactly the same sequence of reads twice. So when looking for a value of s that induces a larger time, we must make sure that the time is *substantially* larger, say by 20%.

Implementation

Your program may consist of assembly code, C code, or a mixture of the two. There is no fundamental need to use any assembly code, but if you decide to depend on C code, make sure you know exactly what assembly code the C compiler is generating, and beware of compiler "optimizations" that affect the memory read behavior you are trying to measure.

You will probably find the `timeit.s` file, available on the course web page, to be useful. This file defines a function `timeit`, which measures execution time of a specified routine. To measure the execution time of a procedure call

```
foo(arg1,arg2,...);
```

replace it by the call

```
timeit(foo,arg1,arg2,...);
```

`timeit` performs the underlying call to `foo`, discards the result, and returns the CPU time used by `foo`. This works for any `foo` that has at most 5 integer arguments. The time is returned as an integer number of units; each unit represents 10 milliseconds. Note that the CPU time may be much less than the elapsed real ("wall clock") time, if more than one person is using the machine.

Submission

To submit this lab, send one or more files containing all your C and/or assembly code. One of these files must contain a main program. The files should be made into a bundle using `shar`. If you use the `timeit.s` file, include it in the bundle. Mail the bundle as the body of an email message to `cs201-1acc`, e.g.,

```
shar foo.c bar.s | mail cs201-1acc
```

Your submission will be built by unpacking your files into a clean directory and using the command

```
cc -o sizetest *
```

and then tested running `sizetest`.