GSB/RF
September 6, 2001

Project 1 — Identifying Memory Hierarchies

This is the first project in the course *External Memory Algorithms and Data Structures*. The project should be done in groups of 2–3 persons. At the end of the project each group is expected to hand in a report documenting the work done, the behavior observed, and the possible explanations of this behavior. The report should be handed in by **September 27, 2001**. Tuesday, September 18, we will spend some time on discussing any problems that might occur while addressing these tasks. It is expected that by then *everybody have obtained some preliminary results*. In other words, start programming right away, even though the final report is only to be handed in later.

The project is centered around identifying the different memory layers on real machines, and estimating the parameters of each of the memory layers. The goal is *not* to do major implementation work, but rather to

- gain experience in measuring the performance of actual programs,
- make observations about the influence of the memory hierarchy on the running time of a very simple program—specifically to see whether individual components of the memory hierarchy show themselves when the input size grows.

Tasks

1. Write a simple C or C++ program that takes arguments n, d, and r and allocates an array of size n and cyclic runs trough the array and accesses entries $0, d, 2d, \ldots, d\lfloor\frac{n-1}{d}\rfloor$, in total visiting r entries. The program should not generate any output.

   One possible approach it to use the allocated array to store the next positions to visit, i.e. $A[id] = (i+1)d$ and $A[d\lfloor\frac{n-1}{d}\rfloor] = 0$.

2. Make a similar program that accesses the array entries in *random* order.

3. Run your programs for different values of n and d and measure the time used by the program. Choose r such that running time becomes acceptable.

   Identify the properties of the measured results and try to give an explanation of the measured times, especially how the memory hierarchy influences the growth in the running time when the problem size n increases. Repeat your experiments on two or more different machine architectures (e.g. Sun, HP, SGI, PC).

   From your results try to identify the following parameters of the machines:

   - RAM size
   - Cache size (L1 and L2 if possible)
   - Cache line size (L1 and L2 if possible)
   - Virtual memory page size
   - Access time for the different levels of the memory hierarchy

Hints

- There are quite a number of UNIX commands to time the execution of a program, among these `time` and `timex`. Read the man page for usage of these. In some shells (e.g. csh), there is also a built-in version of `time`—read the man page for the shell for information on this. There are also corresponding C library routines called `times`, `getrusage`, and gettimeofday (which also have man pages). The availability may differ from machine to machine, though. It is part of the project to consider how to best time the execution of your programs.
- If there are too many program instructions per memory access in your program, the cost of going from cache to RAM may be hard to see in the running times. Your programs should therefore be as optimized as possible, even at the cost of normal programming conventions. In-line functions, use global variables, minimize the number of `if/else`'s etc. Also, set the compilers options for maximal optimization.
- Computing random numbers is expensive.
- Make graphs of the results, e.g. using `gnuplot`. A sensible value to plot could be running time divided by asymptotic complexity (e.g. n or $n \log n$), with n on a logarithmic x-axis.
- Run the program a suitable number of time for each input size and use the average running time. This should level out fluctuations due to other processes.
- *Do not run experiments on file servers!* Rather, try to find a machine with no other user processes executing.
- The file `/users/verksted/Maskiner/udstyrs_reg` contains information about the hardware configuration of the machines at DAIMI. Under Linux the files `/proc/cpuinfo` and `/proc/meminfo` contain information about cache and memory size.
- Sometimes it might be useful to look at the assembler code generated by the compiler to check if the code generated by the compiler is as expected (compilers can optimize away unexpected parts of the code).
- When collecting results from many different executions, it is useful to annotate the output with sufficient information such that it later is clear what execution generated what output. Possible information are: machine id, program id (e.g. CVS version id), and run-time arguments.
- To perform several different executions, it is useful to have a script or meta-program that performs all the executions. This also allows executions to be performed on different machines quite easily.
- Your program should not produce any output (as output is extremely slow compared to access to e.g. RAM). However, during development you should make sure that your program is actually doing what it is supposed to do. One way of achieving this is to incorporate a *checker*—i.e. a routine *verifying* the desired property, e.g. that the array is actually in sorted order after performing a sorting algorithm. An $O(n)$ time routine for checking sortedness is easy to make (how?).