COMP25212.1 Measuring Cache Performance

Duration: 1 x two-hour lab session

Resources:Any computer with a C compiler. This script is intended for those using Linux, but if you wish to use Windows, (or Solaris, FreeBSD or RiscOS) you are welcome to do so. But beware – some of the library calls may be different, and you'll need to fix them yourself.

AIMS

To investigate, by running a series of tests, the performance of the cache memory system of the computer

LEARNING OUTCOMES

To understand how to use microbenchmarks to investigate particular aspects of computer performance

To understand how large differences in performance can result from relatively small changes in workloads.

INTRODUCTION

In later lab sessions, you will use a simulation program to investigate how cache memory may perform in a theoretical system. In this exercise, you will use a real program, running in real time, to measure how the real system in the lab performs under a variety of cache memory load scenarios.

PREPARATION

Copy the program "cachetesta.c" from:

/opt/info/courses/COMP25212/ex1/cachetesta.c

For convenience, you should work in a directory ~/COMP25212/ex1, since **`labprint`** and **`submit`** will require files in that directory.

Also for convenience, a listing of the program is attached to this script.

PART 1 – (1 mark)

Compile the file (e.g. **`make CFLAGS=-O2 cachetest`**)

Run it (e.g. **`./cachetest`**)

Record the final line of output here:____

PART 2 – (1 mark)

Edit the first declaration in the program so that **`nStructs`** takes the value one million.

Compile the program and run it again.

Record the final line of output here:____

PART 3 – (2 marks)

Record the information reported about the CPU (in /proc/cpuinfo on Linux):

Vendor_id:____

Cpu family: ____

Model: ____

Model name: ____

Cache size: ____

PART 4 – (2 marks)

Why are the times reported in PART 1 and PART 2 different?

____

PART 5 – (4 marks)

Investigate the effect of varying **`nStructs`** on the performance of the cachetest program, plotting your results with an appropriate scale. (Hint: use powers of 10 initially) (Hint: plot your results as you go – using the appropriate graph paper in this script) Pay particular attention to the regions in which performance changes quickly with **`nStructs`**. Plot your results with an appropriate scale.

PART 6 – (4 marks)

Explain how cache size reported in PART 3 relates to the values of **`nStructs`** at which performance changes quickly.

____

PART 7 – (3 x 2 marks)

Identify different regions of your graph with different performance characteristics and explain:

In region 1: ____

In region 2: ____

In region 3: ____

Optional Extras:

PART 8 – (5 marks)

The cachetest program as written simply tests memory reads over different cache working sets. Modify the program so that every iteration also writes into the structure and repeat the performance measurement.

How does the characteristic vary?

PART 9 – (4 marks)

The cachetest program as written uses reads to a structure of a certain number of bytes in width. What happens if we double the size of the structure? What happens if we halve its size?

PART 10 – (10 marks)

How could we enhance the cachetest program to determine whether a processor cache was direct mapped, set associative, or fully associative?

cachetesta.c

```
#include <stdio.h>

/*
  Cache Timing Test

  Access a (user-configurable) number of data structures
  distributed through memory

  and time report the run-time taken

*/


int nStructs = 15;              /* how many structures to keep accessing - defines working set */


/*********************************************************************************************

                                     THE TEST

*********************************************************************************************/


struct entry {                  /* this is the data structure we access */
  struct entry* next;           /* a pointer to the next structure */
  int padding[7];               /* and some extra variables */
};

int maxstructs = 16777216/sizeof(struct entry);

/* allocate at least 16 million bytes of RAM
   in case there's a really HUGE cache */

struct entry *base;             /* pointer to table of data structures */

static struct entry *testPtr;   /* this pointer is used to follow the linked list of structures *
/

static int remaining = 0;       /* countdown variable - count # of structures accessed */

/*
  INITIALIZATION
*/

#include <stdlib.h>

void
initialize() {
  int i;
  static int stride = 5;        /* PLEASE CHOOSE A PRIME STRIDE!!! */
  base = (struct entry *) calloc(maxstructs, sizeof(struct entry)); /* allocate memory area
                                                                       see "man 3 calloc" */
  for (i = 0; i < (maxstructs-1); i++) {
    int j = (i*stride) % maxstructs;
    int k = (j+stride) % maxstructs;
    base[j].next = &(base[k]);          /* link j'th struct to k'th struct */
    base[k].next = &(base[0]);          /* strictly only for very last entry:
                                           ensure that final entry links
                                           back to the first */
  }
}

/*
  THE TIMED TEST
*/

void
runtest() {                     /* access next entry in linked list -
                                   keep following the list until we've accessed nStructs
                                */
  if (remaining == 0) {         /* if we've accessed enough entries */
    testPtr = &(base[0]);       /* reinitialize the pointer - start at beginning */
    remaining = nStructs;       /* and reset the counter */
  }
  remaining = remaining - 1;    /* keep count */
  testPtr = testPtr->next;      /* and --ACCESS THE DATA STRUCTURE-- */
}


/*
  TIMING INFRASTRUCURE
*/

#include <sys/resource.h>

double
getutime() {                      /* returns real number, giving seconds of
                                     user time consumed by this process */
  struct rusage ru;
  getrusage(RUSAGE_SELF, &ru);
  return (1.0 * ru.ru_utime.tv_sec + (0.000001 * ru.ru_utime.tv_usec));
}

/* run the test "N" times, and return the time (in real seconds) this takes */

double
runit(int n) {
  double t1 = getutime();
  int i;
  for (i = 0; i < n; i++) {
    runtest();                    /* access next element */
  }
  return (getutime()-t1);
}

/* MAIN PROGRAM */

/*
  two-step process to calibrate run time:

  a) run test enough times to take at least 0.1 seconds
  b) then run the test for an estimate 1 second

  and report run time per iteration
*/

int initial = 10000;                     /* how many times to try running loop at first */

int
main(int argc, char **argv) {
  double runtime;
  int iterations;

  initialize();

  runtime = runit(initial);

  while(runtime < 0.1) {
    initial += initial;
    runtime = runit(initial);
  }

  printf("%d iterations take %7.5f seconds\n", initial, runtime);

  iterations = 1.0 * initial / runtime;

  printf("trying %d iterations\n", iterations);

  runtime = runit(iterations);

  printf("%d iterations take %7.5f seconds\n", iterations, runtime);

  runtime = runtime / (1.0 * iterations);

  printf("one iteration takes %12.10f seconds\n", runtime);

  printf("one iteration takes %12.3f  nsec\n", runtime * 1000000000.0);

  return(0);

}
```