

Score: \_\_\_\_\_ Section: \_\_\_\_\_ Date: \_\_\_\_\_  
Name: \_\_\_\_\_

## **ECE 3055 Laboratory Assignment 4**

Due Date: Tuesday, April 5

In this lab, you will use Java or C/C++ to write a program that will evaluate and compare the cache hit rates for several cache architectures and then for a TLB. Given a particular memory organization (size of address space, and size of cache memory), the program will read a file containing a memory trace (a sequence of memory addresses) and determine which of the memory references will cause cache hits. The program should keep track of the total number of hits and misses generated by each cache over the entire trace. The hit rate should be plotted for different cache memory sizes and architectures (as detailed below). The trace file is available in zip format from the Lab Web page:

[http://www.ece.gatech.edu/~hamblen/3055/gcc\\_trace\\_txt.zip](http://www.ece.gatech.edu/~hamblen/3055/gcc_trace_txt.zip)

There are 1,000,000 memory references in the trace file stored one per line. These were collected from the gcc C compiler while it was running on a MIPS processor. The address is the second field on the line. The first field gives the type of the memory reference (2 for instruction fetch, 1 for a store's data memory write, and 0 for a load's data memory read). The third field gives the instruction value for a fetch and is always 0 for loads and stores. Keep track of miss rate for instructions and data. To help you get started on the lab, a simple program that reads the trace file, extracts the address, and performs some simple bit-wise operations similar to what is needed to extract bit fields from the address to determine the cache address is available at the URL below in both C and Java formats.

[http://www.ece.gatech.edu/~hamblen/3055/read\\_trace.c](http://www.ece.gatech.edu/~hamblen/3055/read_trace.c)

[http://www.ece.gatech.edu/~hamblen/3055/read\\_trace.j](http://www.ece.gatech.edu/~hamblen/3055/read_trace.j)

If you need help on the Java or C tools available on the ECE PC lab machines, see:

<http://www.ece.gatech.edu/~hamblen/3055/VisCtut.doc>

or

<http://www.ece.gatech.edu/~hamblen/3055/VisJtut.doc>

**(40%) Part 1:** On a single page, plot the cache miss rates versus cache memory size for a processor with two direct-mapped caches. The I cache is used for Instruction fetches only and the D cache is used only for data (lw, sw memory data operands). The separate Instruction and Data cache architecture used in most modern processors. The size of each cache is the same. Data points should be generated for cache memory sizes starting at 128 to 32K in powers of two. Include 64K and 128K, if possible. 128 means 128 cached data locations in each cache. Do not count writes in the hit rate calculation. Assume

byte addressing and the block size is one 32-bit word. Generate a plot of miss rate vs. cache size, one curve for instructions and one curve for data.

**(20%) Part 2-** Repeat part 1 for a block size of 1, 2, 4 and 8 words in the cache with the same number of cache data locations. Generate plots where the total data locations are constant while the block size varies – like figure 7.8.

**(40%) Part 3-** Assume a virtual memory system is used with 4K byte pages and an 8 entry direct mapped TLB (holds 8 page table entries). Assume 64K bytes of Physical memory is available for paging. Compute the TLB hit rate while running the instruction sequence. Use an LRU page replacement policy. Repeat with a 2, 4, and 16 entry TLB. Generate a plot of TLB miss rate vs. TLB size. Keep track of the total number of page faults. Do not forget to count compulsory page faults, i.e. automatic page faults that occur the first time each virtual page is referenced in the trace. When you throw away a page, assume it's TLB entry is marked as invalid (if it has one).

**Caution:** For Part 3, you will need to develop a page table data structure that is used by to track the virtual pages that it keeps in physical memory as memory locations from the trace are accessed. If you use a page table indexed by virtual page number, it is likely that your program will experience memory allocation problems and/or excessive running time due to large page table size. Consider instead using an inverted page table, which is indexed by physical page number (see Silberschatz and Galvin, pages 344-345, for a description of inverted page tables). You do not need to implement hashing in your inverted page table because the time to search this relatively small page table is not a significant concern in your program. You also need to carefully consider what information to store in a page table entry to enable the LRU policy to be implemented.

You **do not** have to write code to plot the data, only to calculate it. The plotting can be done by any method you wish, e.g. by hand or graph paper or with an available plotting program such as MS Excel.

```

#include <stdio.h>

void main()
{
    FILE *fp;
    int i, type;
    unsigned int address, instr; // "int" is 32 bits on Intel 32A
    unsigned int masked_addr;

    // open trace file for reading

    fp = fopen("gcc_trace.txt", "r");

    // read first 100 lines of trace file;
    // address holds virtual address in unsigned int format;
    // N.B. - your program must read all 1,000,000 lines but
    //       you might want to test it on the first 100 lines
    //       or so before running it on the entire trace file!

    for (i=0; i < 100; ++i) {

        // read 1 line of trace file

        fscanf(fp, "%d %x %x", &type, &address, &instr);

        // example of bit-wise operation in C;
        // mask off all but the 2nd least significant hex digit,
        // shift right by 4 bits, and print address in hex and
        // masked shifted address in decimal

        masked_addr = (address & 0x000000f0) >> 4;
        printf("%08x      %2d\n", address, masked_addr);
    }
}
/* ===== EXAMPLE DATA
=====

2 400170 8fa40000
0 7ffebc64 0
2 400174 3c1c1002
2 400178 279ce710
2 40017c 27a50004
2 400180 af85abb4
1 100192c4 0
2 400184 24a60004
2 400188 41080
2 40018c af80abb8
1 100192c8 0
2 400190 c23021

```

2 400194 27bdffe8  
2 400198 af86a470  
1 10018b80 0  
2 40019c afa00014  
\*/