Introduction

This project involves writing a program to identify the following three characteristics of cache: the number of levels and the size of each level, the line size, and the associativity of each level.

Strategy

Cache Levels and Sizes:

The number of levels and the size of each level are found by measuring the time needed to update a constant number of elements in arrays ranging from 8kb to 32MB. The run time for a given array size is compared to the time for the previous array size. If the current time at least is 14% greater (determined experimentally) than the previous time, then a level of cache has been identified since the array has outgrown a lower level of cache and can only fit in a slower, higher level of cache.

Line Size

The line size is determined using a similar strategy. The difference is that size of the array is held constant while the stride is varied with each run, ranging from a stride of 2 elements (8 bytes) to 128 elements (512 bytes). For each stride size, the program compares the ratio of run time to the run time for the previous stride size. The stride producing the maximum percent change multiplied by sizeof(int) represents the line size since this is the point at which the maximum number of loads will be performed. In other words, at this point, each element accessed will require a new line to be loaded.

Associativity

This portion of the program runs for each cache size found previously. For each cache size found, the stride is set to the cache size. The array size varies from four times the stride to 20 times the stride. Every stride element is updated in the array, and if the index of the array element to be updated is greater than the array size, then the index is reset to 0. The time to update the array is measured, and the program determines the array size which produces the maximum change in run time. This maximal value represents the point at which the number of lines accessed is greater than the number of sets. As a result, there are cache conflicts, and the run time increases.

Questions and Anomalies

It is interesting to note that my program completely fails to identify any levels of cache when every element in the array is updated. In order to see differences in run time, it is necessary to increase the stride to approximately the level of the line size. It is my understanding that the act of updating the elements in the array should counteract prefetching, but this does not appear to be the case with my i7. To avoid overtuning the program to my specific machine, I chose a stride of 256 bytes, a relatively large multiple of 2, which is much larger than my line size of 64 bytes.

The program could be improved by refining the timing aspect of my program. In researching this project I read of various strategies to improve the accuracy of timing. One such strategy involves running a certain number of of loop iterations before starting

the timing in order to minimize the effect of the loop overhead. It would be interesting to see if such a strategy would improve the functioning of my program by increasing the signal to noise ratio.

Other questions to explore include: Does initializing the array affect the run time? Does initializing the array using random instead of sequential numbers affect the run time? Does reinitializing the array before each task alter the results? (I do not see why it would.) Would disabling hardware prefetching affect the results?

Analysis and Results

Based on trials, my program is >90% accurate in determining the number of levels cache, the sizes of L1 and L2, and the associativity of L1. My program is about 60% effective in finding the size of L3, the line size, and the associativity of L2. (The program typically slight underestimates the size of L3 and the line size.) My program fails to correctly find the associativity of L3. I am not sure if this failure is due to a flaw in my strategy or a flaw in my implementation or both.

There are probably countless flaws with my approach, and there are many ways that the program could be improved. For example, in finding the size of the levels of cache, the program simplistically takes the first three increases in run time of over 14%. A more robust implementation could search for the largest changes in run time. (This shortcoming frequently leads the program to underestimate the size of L3.)

Also, it is worth noting that my code is horribly inefficient. A revision of this program would be much, much shorter. I erred in writing each task independently, so there is significant redundancy, particularly the code to iterative through the array and the code for finding changes in run time, which is repeated for each of the tasks. A better version of this program could use a single procedure, which could vary either the array size or the stride, to measure the number and size of the levels, the line size, and the associativity.

Conclusion

In summary, a program may be written to “break” cache varying the array size and the stride, and the results may be used to measure the number of levels of cache, the size of each level, the line size, and the associativity of each level. To measure the number and size of levels of cache, one increases the size of the array to be updated while maintaining the stride. To measure line size, one varies the stride while maintaining a constant array size, which should be at least twice as large as the largest level of cache. Finally, to measure the associativity, one increases the array size while maintaining a constant stride, which is equal to the size of level of cache being analyzed.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define MAX ((1024*1024*32)/sizeof(int))//32 MB array
#define MIN 2048 //min array size =8kb
#define ITERATIONS 1024*1024*10
int array2_size = 1024*1024*30;
static int array2[1024*1024*30];
static int array[MAX];
int current_size;
struct timeval tvafter,tvpre;
int results_counter = 0;
int elapsed_time = 0;
int levels_found = 0;//number of cache levels found
float output = 0.0;//current time / prev time
int time_results[28];//time to pass through at each array size
int size_results[28];//array size corresp to time
int test_index = 0;
int sizes[4];

void initialize_array() {
        for (int i = 0; i < MAX; i++) array[i] = rand() % MAX;
}

//update elements in array
void get_size() {
        current_size = MIN;
        double it = ITERATIONS;
        double total = current_size * it;
        int a = 0;
        while (current_size <= MAX) {
                gettimeofday (&tvpre , NULL);
                for (int k = 0; k < it; k++) {
                        for (int j = 0; j < current_size; j=j+64) array[j] = array[j]+1;
                }
                gettimeofday (&tvafter , NULL);
                elapsed_time = ((tvafter.tv_sec-tvpre.tv_sec)*1000+(tvafter.tv_usec-
tvpre.tv_usec)/1000);
                size_results[results_counter] = current_size;
                time_results[results_counter] = elapsed_time;
                results_counter++;
                if (current_size <= 262144) {
                        current_size = current_size*2;
                        it = it*0.5;
                } else if (current_size > 262144 && current_size < 2621440) {
                        current_size = current_size+262144; //increase size by 1mb
                        it = total / current_size; }
                else {
```

```
                current_size = current_size+524288; //increase size by 2 mb
                    it = total / current_size;
                }
                total = current_size * it;
        }//end while
}

//find levels and size of each levels
void find_levels() {
        int size_counter = 0;
        float change[27];
        float max_change = 0;
        for (int i = 1; i < 28; i++) {
                output = ((float)time_results[i]/(float)time_results[i-1]);//output = current/previous
                change[i] = output;//store output in %change array
                if (output > max_change && size_results[i-1] > 16384) {
                        max_change = output;
                        test_index = i-1;//store previous prev index
                }
                if (output > 1.14 && levels_found < 3) {
                        if (size_results[i] < 262144) {
                                sizes[size_counter] = (size_results[i-1]);
                                printf("Level %d: %d kb\n", levels_found+1,(sizes[size_counter] *
4)/1024);
                                size_counter++;
                        } else {
                                sizes[size_counter] = (size_results[i]);
                                printf("Level %d: %d mb\n", levels_found+1, (sizes[size_counter]
*4) / (1024*1024));
                                size_counter++;
                        }
                        levels_found ++;
                }
        }
        if (levels_found == 0) printf("This machine has 0 levels of cache.\n");
                else printf("This machine has %d levels of cache.\n", levels_found);
}

//find the line size by varying the stride
void get_line_size() {
        elapsed_time = 0;
        int prevTime = -1;
        int prevSize = 0;
        float max = 0;
        int stride_counter = 0;
        int line_size = 0;
        for (int stride = 1; stride <= 128; stride=stride*2) {
                long long accesses = 0;
                output = 0;
```

```
                gettimeofday (&tvpre , NULL); //start time
                for (int r = 0; r < 1024*10; r++) {
                        for (int j = 0; j < MAX/4 && accesses < 1024*1024*1024; j = j + stride) {
                                array[j] = array[j]+1;
                                accesses++;
                        }
                }
                gettimeofday (&tvafter , NULL);
                elapsed_time = ((tvafter.tv_sec-tvpre.tv_sec)*1000+(tvafter.tv_usec-
tvpre.tv_usec)/1000);
                output = ((float)elapsed_time/(float)prevTime);//output = current/previous
                if (prevTime != -1) {
                        if (output > max) {
                                max = output;
                                line_size = stride;
                        }
                }
                prevTime = elapsed_time;
                prevSize = stride;
        }
        printf("The line size is %d bytes.\n", line_size*4);
}

void initialize_array2() {
        for (int i = 0; i < array2_size; i++) array[i] = rand() % array2_size;
}

/*measure associativity for each level identified by setting stride = cache size
and increasing array size*/
void find_associativity() {
        int accesses = 0;
        initialize_array2();
        int stride_a = 0;
        int number = 0;
        for (int j = 0; j < levels_found; j++) {
                accesses = 0;
                float max = 0.0;
                int assoc_size = 0;
                float output = 0;
                int prev_size = 0;
                int prev_time = -1;
                stride_a = sizes[j];
                int limit = stride_a * 20;
                for (number = stride_a * 4; number <= limit;) {
                        gettimeofday (&tvpre , NULL); //start time
                        int element = 0;
                        for (int i = 0; i < 1024*1024*1024; i++) {
                                array2[element] = array2[element] + 1;
                                element = element + stride_a;
```

```
                        accesses++;
                        if (element > number) element = 0;
                }
                gettimeofday (&tvafter , NULL);
                elapsed_time  =  ((tvafter.tv_sec-tvpre.tv_sec)*1000+(tvafter.tv_usec-
tvpre.tv_usec)/1000);
                if (prev_time != -1) output = (float)elapsed_time / (float)prev_time;
                if (output > max) {
                        max = output;
                        assoc_size = prev_size;
                }
                prev_size = number;
                number = number+stride_a;
                prev_time = elapsed_time;
        }
        printf("Level %d associativity: %d\n", j+1, (assoc_size/stride_a));
    }
}

int main () {
    initialize_array();
  get_size();
    find_levels();
    if (levels_found != 0) {
        get_line_size();
        find_associativity();
    }
    return 0;
}
```