Student Name: Ryan Protheroe Student Number: 20069587

# CISC-221 Computer Architecture Lab The Memory Hierarchy Lab 7: Memory Performance

Instructor: Dave Dove ( [dove@cs.queensu.ca](mailto:dove@cs.queensu.ca) ).

1 Introduction

In this lab you will collect data relating to the performance of various layers of the Raspberry Pi memory hierarchy and investigate how performance can be altered significantly by the memory access patterns of a user program.

You will analyze the data that you collect and draw conclusions relating to the information collected in this lab.

# 2 Logistics

You must do this lab individually and submit your solution to the associated drop-box in OnQ under your own name, before the scheduled due date and time.

# 3 Instructions

There are two parts to this lab. In the first part, you will run performance measurement programs, adjusting input and measurement parameters. Your results should reveal relationships between array sizes, access patterns and performance as well as comparative access times of multiple layers of the memory hierarchy.

In the second part, you will draw conclusions about the results that you obtained.

## 4 The tools you will be using.

**chase**: This is a simple pointer chasing program that creates an array of linked list entries that alternately access the lowest half, then the upper half of the array. Each array element consists of 8 (4 byte) pointers that will fill one 32 byte cache line of the raspberry pi. By defining the number of array elements when executing the program, you will be adjusting the array size. Although not necessary to understand, further information can be found at the end of this document about this program.

**latency**: This program uses the same pointer chasing program but instead of measuring total program execution time (User time), it measures the number of cycles taken for each memory access. You will set the number of array elements in the same way as with the chase program but the number of iterations is always 1000 (by default). This program creates a file *samples.dat* which reports the number of cycles required for each memory access. You will then use the **memhist** script to turn the analyze the results and print out a histogram of memory access times.

**memhist**: a bash script (copy from /cas/course/ydrive/cisc221) to analyze and print out a histogram of the samples.dat file generated by the latency program.

# 5 Helpful Details

References:

OnQ: Archived Lecture Notes: Storage Management, Raspberry Pi Memory

Course text: Chapters 6 and 9.

**6 What you need to do:**

## Part 1

1. Log into the raspberry pi assigned to your group and create a new directory for this lab.
2. Complete Table 1 below by running the ***chase*** program for each row of the table and recording the resulting run time reported in the file: ***chase.txt***. Run times should range from around 2 to 26 seconds. For each row of the table run the command: chase –i –s ***elements*** –n ***iterations*** where *elements* is the value listed in the Array Elements column and *iterations* is the value listed in the Iterations column.

Adjusting the number of array elements (each array element will fill one cache line) will increase the size of the array. Decreasing the number of iterations will decrease the total number of instructions executed. By increasing the size of the array and decreasing the number of iterations at the same time, the total number of instructions executed will remain constant and the User Time results should vary according to just the data access time.  
After completing the table, group the User Time readings into 3 tiers where Tier 1 covers the shortest times and Tier 3 covers the longest times. Each tier should differ significantly from other tiers in User Time.

1. Now you will vary array size and see the effect on the data access time. Accessing data from the cache will be faster than accessing data from main memory. Accessing data from main memory will take even longer if there isn’t an entry in the data Micro TLB or the Main TLB as an additional memory access is required to get the entry from the page table in main memory.  
     
   To measure we will use a program called ***latency***. We will run the program over the same list of array element sizes as before (we will be using the same list of Array Elements and resulting Array Sizes). The program tests the latency of data accesses (number of cycles) over 1000 samples. The measurements are output to a file called ***samples.dat***. After the latency program finishes, run the script ***memhist*** to print a histogram of access cycle times.  
     
   Complete Table 2 below by running the latency program with the following options: ***latency –s <elements>*** where <*elements*> is the number in the Array elements column of the table. After the latency program completes print a histogram of the results by executing the command: ***memhist***.  
     
   The output of the ***memhist*** program might look something like this:

62 563 ##################################################

61 311 ##################################

9 61 #######

70 22 ###

63 22 ###

123 3 #

69 2 #

64 2 #

17 2 #

138 2 #

Look at the first line of the histogram. The first number in the line refers to the number of cycles. The second number refers to the number of samples that had a latency of that number of cycles. The hash marks are an old-school text-based representation of a bar graph representing the relative number of samples (compared to the total number of samples) that had that access time.  
From the histogram above we see that most samples (563+311) had a latency of 61 or 62 cycles while 61 samples had an access time of 9 cycles, and 44 samples (22+22) had an access time of 63 or 70 cycles. The remaining single # marks result from a variety of other things that are going on and can be ignored as not statistically relevant.

Again group the readings in the histogram into the three tiers you created after completing Table 1 ranging from fastest to lowest. There will be some reading that are transient and don’t fit well into any of the three categories. They result from other activities that are happening during the execution of the program such as timer interrupts and various other minor events. Ignore these readings. You should be able to confidently categorize at least 800 of the 1000 total samples.

Notes:  
The measurement process adds 6 cycles to each memory latency measurement (3 cycles to start the cycle counter and 3 more to stop it). This means that the actual memory latency is 6 cycles less than the number reported in the first column of the histogram table. For example, a reported latency of 62 cycles actually has a latency of 62 – 6 = 56 cycles.

The clock frequency of the pi is 700MHZ. That means that 1 memory cycle takes 1/f = 1/700 Mhz = 1.43 nanoseconds

## Part 2

1. Answer the following questions. The powerpoint slides in the Raspberry Pi Memory Hierarchy pdf have the numerical information required to answer these questions. You will find it useful to review the course text and archived lecture notes to make sure you understand how the Translation Lookaside Buffer works.  
   1. What is the maximum size of array that will fit into the cache?

The max size of array that will fit into the cache is 160KB as a result from both L1 caches (16KB each) and the L2 cache(128KB)

* 1. Try to explain the relationship between the array sizes of Tier 1, Tier 2, Tier 3 and which level of the memory hierarchy supplies the data referenced.
* Tier 1 arrays are supplied with data from the L1 cache, this is because of the size of the arrays 8KB - 16KB, which is the size of the L1 caches (16KB).
* Tier 2 arrays are supplied with data from main memory, the array size is too large to fit in L1 cache, they fit into the 288KB memory coverage of main memory and it would be a “hit”.
* Tier 3 arrays are supplied with data from main memory, these are the larger arrays that don't fit into L1 cache and are a miss in the TLB  
  1. The data will always come from either the L1 cache or from main memory so why is there three tiers instead of two? (hint: it has to do with the TLB)

This is due to the fact that the tier 3 is a miss in the TLB and tier 2 is a hit in the TLB, hence why tier 2 is faster.

* 1. Explain how array size affects memory latency in terms of the capacity and actions of the Translation Lookaside Buffer (MicroTLB and MainTLB).

As the array size increases the chance of a miss increases since the arrays cannot be stored in cache, latency increases as a result. It must then reference the TLB’s to get the associated page table entry and you would get regular main memory latency. However if it isn't in the TLB’s or cache, then we have even more latency as we need to get the associated page table entry from main memory and this increases latency to at least one more main memory cycle.

## Table 1. Array Size vs Execution Time

## run: chase –i –s <Array Elements> -n <Iterations>; cat chase.txt

|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size** | **Array Elements** | **Iterations** | **User Time (seconds)** |
| 4MB | 131072 | 1024 | 24.650 |
| 2MB | 65536 | 2048 | 23.370 |
| 1MB | 32768 | 4096 | 25.830 |
| 512KB | 16384 | 8192 | 24.630 |
| 256KB | 8192 | 16384 | 25.520 |
| 128KB | 4096 | 32768 | 16.370 |
| 64KB | 2048 | 65535 | 11.100 |
| 32KB | 1024 | 131072 | 9.840 |
| 16KB | 512 | 262144 | 2.530 |
| 8KB | 256 | 524288 | 2.360 |

Now group your time readings into 3 tiers or groups with the least time variation within each tier and enter the tier information from Table 1 below.  
Use Tier 1 for the shortest times and Tier 3 for the longest times.

Tier 1 Time range: 2.36s - 2.53s Array size range: 8KB - 16KB

Tier 2 Time range: 9.84s - 16.37s Array size range: 32KB - 128KB

Tier 3 Time range: 23.37s - 25.83s Array size range: 256KB - 4MB

## Table 2. Array Size vs Memory Access Time (latency)

## run: latency –s <Array Elements> ; memhist

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Array Size*** | ***Array Elements*** | **Latency** | | | | | |
| **Tier 1 (fastest)** | | **Tier 2** | | **Tier 3 (slowest)** | |
| **cycles** | **#samples** | **cycles** | **#samples** | **cycles** | **#samples** |
| 4MB | 131072 |  |  |  |  | 116-121 | 831 |
| 2MB | 65536 |  |  |  |  | 116-121 | 867 |
| 1MB | 32768 |  |  |  |  | 116-121 | 844 |
| 512KB | 16384 |  |  |  |  | 144-147 | 942 |
| 256KB | 8192 |  |  |  |  | 116-121 | 844 |
| 128KB | 4096 |  |  | 61/62 | 429 | 123/124 | 438 |
| 64KB | 2048 | 9 | 56 | 61/62 | 887 |  |  |
| 32KB | 1024 | 9 | 101 | 61/62 | 844 |  |  |
| 16KB | 512 | 9 | 1000 |  |  |  |  |
| 8KB | 256 | 9 | 999 |  |  |  |  |

Tier 1 latency = (cycles -6) \* clock cycle time = 4.29ns

Tier 2 latency = (cycles -6) \* clock cycle time = 80.08ns (used 62 as cycles)

Tier 3 latency = (cycles -6) \* clock cycle time = 167.31ns (used an average of 122 as cycles)

## Additional information on how the chase program works.

The fundamental data structure in the pointer chasing program (called chase) is the linked list item:

typedef struct \_CacheLine  
{

struct \_CacheLine\* ptrCacheLine[8] ;

} CacheLine;

//the ptrCacheLine array consists of 8 4-byte (32-bit) pointers and will therefore fill one cache line of the pi.

The larger array of the chase program (8KB – 4MB in size) is initialized by the following code:

steps = array\_elements/2;

bottom = 0;

top = steps;

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

array[bottom].ptrCacheLine[0] = &array[top];

array[top].ptrCacheline[0] = &array[bottom+1];

top++;

bottom++;

}

array[(array\_elements-1)].ptrCacheLine[0] = NULL; //used to terminate linked list

The actual point chasing loop is quite simple:

void(test\_loop\_prototype(CacheLine\* aray)

{

CacheLine\* p;

for(p = array; p != NULL; p->ptrCacheline[0]);

}

//the program calls test\_loop\_prototype many times in order to increase the overall run time which improves repeatability and accuracy of the results.

## Pointer chasing access pattern over array[ ]