MANCHESTER 1824
The University of Manchester

Computer Systems Architecture (EEEN30040)

Laboratory 2 - Cache Memory Behaviour of ARM Processors

November 5, 2017

Lecturer: Dr. John Oakley

Student ID: 9563426

Contents

1. Introduction

This report will attempt to quantitatively describe and explain cache memory behaviour of ARM processors, in particular the ARM Cortex-A53. The laboratory is divided into two experiments.

In the first experiment, pre-written code is compiled and run. This code will continuously write to and read data from memory locations while gradually increasing the stride (0 - 16384) between successive data. While doing so the program will measure the Memory Access Time (MAT) of each iteration. From first principles, it is expected that as stride increases, MAT also increases until stride largely exceeds cache line size.

To be more specific, stride is defined as the increment or step size between locations in memory of successive data. A simplified abstraction of cache memory was drawn to clarify this concept and is shown in figure 1.1 below.

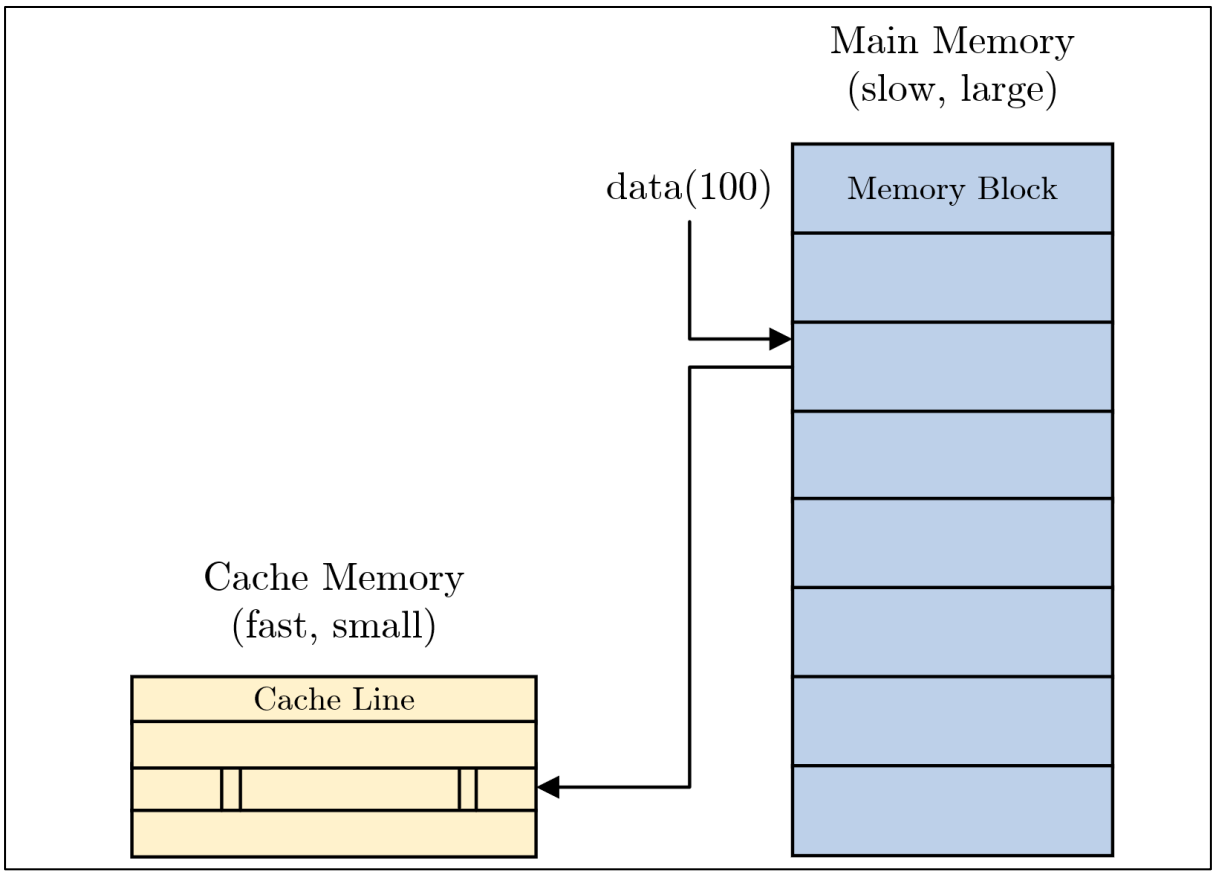


Figure 1.1 - Cache memory abstraction

The diagram in figure 1.1 shows that memory is clustered in blocks (containing multiple pieces of data) in the large but slow main memory. However, memory is organised in lines for the small but fast cache memory. When a piece of data must be read, if it is not present in the cache, it will be fetched from a block of main memory and the block will be copied into the cache line so that next time it is accessed in can be read faster. Main memory block size and cache line length must be equal. Thus, when the first piece of data of a program is read, a miss will always occur and is called *first reference miss*. The ratio of cache hits relative to cache misses can be expressed as cache hit rate, given by the following equation:

$$CHR = \frac{C_h}{C_m} \tag{1}$$

where: CHR *is hit rate,* C_h *is number of cache hits and* C_m *is cache misses*

Therefore, if the stride is low (i.e. small gap between memory locations) program data is statistically more likely to be stored within the same block of main memory. Consequently, as soon as one piece of data from that block is read, the entire block is copied to a cache line and many other pieces of data from the program are also likely to be included in that cache line. This means when the next piece of data must be read it will likely be already in the cache which will result in a cache hit. It follows that for low strides, cache hit rate is high which results in fast memory access times. This is commonly referred to as taking advantage of *locality of reference*.

With increasing stride, the data is less and less likely to be stored in a small number of blocks thus for every block loaded into cache there is a smaller chance of also loading other relevant program data. This means cache hits are less likely as *line replacement* occurs, thus cache hit rate will decrease. Memory access time will therefore be longer as data is less likely to be loaded from cache and more likely to instead be fetched from main memory which takes longer. This is referred to as exhibiting weak locality of reference and is a particularly suboptimal approach to programming. This is because increasing stride effectively defeats the

benefits of using a cache by increasing MAT to levels comparable to main memory reads. These are the fundamentals which the first program exploits to artificially defeat the cache mechanism by increasing the stride.

In the second experiment, the code of the first experiment is modified in-lab by adding an additional `do while` loop to change the working set size (WSS). The program will therefore not only increase the stride from 0 to 16384, as before, but will also increase working set size (0 - 33554432). It is worth noting that the working set size was set very high for the first experiment (33554432 or 2^{25}). The second experiment will produce data showing the impact of increasing stride for each separate value of WSS.

From first principles, it is expected that for very low values of WSS, memory access time will not be impacted by increasing stride whereas for higher values of WSS memory access time will get slower with increasing stride (as seen in the first experiment).

This is because by limiting working set size, all the data used in the program, even if spread out in many blocks, will be able to fit in the cache. Thus, for low WSS, the first few memory accesses will indeed be cache misses (*compulsory misses*) but, long term, all the data will be in the cache and the hit rate will be high (as there won't be *capacity misses* or line replacements). This means MAT remains very fast even at high strides.

However, there exists a threshold WSS after which the program no longer fits into the cache and cache replacement strategies begin. As cache replacement occurs, having very large strides does make a difference as cache replacement will have to occur far more frequently as data is not well clustered and the program fails to take advantage of locality of reference (as seen in the first experiment). In this case for high WSS, while the stride remains low memory access times remain fast, but as stride increases memory access time will drop off as capacity misses occur.

Details of these programs' operation is explained further in section 3.

2. Background

The first generation of Raspberry Pi computers was brought to market in February 2012 [1]. While various generations differ slightly in their hardware, all Raspberry Pi computers are single PCB computers using microSD to store usually Linux-based OSes (although Windows 10 IoT Core can be supported) [2]. In this laboratory we will be using the Raspberry Pi 3 Model B which is similar to its predecessors but still contains some slight differences.

The RPi 3 runs on an ARMv8-A 64-bit architecture. More specifically is uses the 4-core ARM Cortex-A53 processor running at a clock speed of 1.2GHz. This CPU does support overclocking provided reasonable cooling is available. The RPi 3 used in lab will however be run at standard clock speeds to extend reliability and lifetime. The instruction set architecture is load/store and is classified as RISC, as with all ARM processors [3].

This A53 CPU has an 8-stage pipeline, supports Virtual Memory and has 2-stage memory hierarchy with L1 & L2 caches. Although the ARM processors are Von Neumann by design, this memory has Harvard Implementation (shared access for program memory and data memory). The cache replacement strategy is pseudo-random. The write policy can be configured as either write-back or write-through. Data prefetching is supported [3].

The L1 cache size is 32KB and the L2 cache size is 512KB. More specifically, L1 memory is split as 16KB for Instruction and 16KB for Data in the Raspberry Pi 3, however this is configurable [4]. The L1 Instruction memory is 2-way set associative while the L1 Data memory is 4-way set associative. The L1 cache line length is fixed at 64 bytes. The L2 memory is 16-way set-associative. The L2 cache line length is also fixed at 64 bytes. This memory can be shared with the GPU [3].

The SDRAM main memory is 1GB which is shared dynamically between CPU & GPU. Further details are outside the scope of this report.

The GPU is a Broadcom VideoCore IV (BCM2837). This has a performance of 24 GFLOPS. The instruction set for this GPU is RISC. Further details are outside the scope of this report.

The RPi 3 was manufactured in two variants: the first by Sony (UK, hardware revision #a02082) and the other by Embest (China, hardware revision #a22082).

3. Results & Analysis

3.1. Information Gathering

A basic shell script was written to gather information about the Raspberry Pi 3 being used. This script can be found in Appendix 6.1.

The shell script collects the following information: processor information, model information, CPU/GPU/RAM usage statistics, Debian Linux version, kernel version. The collected information can also be found in Appendix 6.1.

The script correctly identifies the Raspberry Pi as 'Raspberry Pi 3 Model B' (UK version) with 1GiB of SDRAM, under an ARMv8-A architecture. It is worth noting the CPU is indeed ARMv8 but `proc/cpuinfo` reports ARMv7 since the Raspbian distribution still uses a 32-bit ARM kernel and the CPU boots in 32-bit compatibility mode.

From the gathered information, it is found (amongst other discoveries) that the processor is indeed a 64-bit 4-core ARM Cortex-A53 processor.

This information is to be used to compare results obtained from in lab experiments with literature values and results. Such a comparison should allow for a thorough evaluation and analysis.

3.2. Cache Behaviour with Increasing Stride

In this section we will observe the behaviour of cache memory when the stride between the locations of successive memory accesses is incremented.

The code provided (Appendix 6.2) is rather simple. It begins by declaring all the variables to be used in the program and pointing the program to an output .csv file for storing the results.

It then goes on to allocate blocks of memory for use in the program using the `malloc()` command. 'Malloc' is short for memory allocate. This is not to be confused with the working set size, which will be explored in section 3.3. Working set size (WSS) is the size of the memory actually being used during the program, either for writing or reading. On the other hand, memory allocated using `malloc()` is simply set aside regardless of whether or not it will be used. The amount of memory allocated is set by `MAXIDX` using a `#define` command (set at 33554432 or 2^{25}).

The program is then composed of one large `while` loop which executes the core code repeatedly incrementing stride exponentially until a maximum stride is reached. The maximal value of stride is set at 16384 or 2^{14}.

The core code simply records the average memory access time which it displays on the terminal screen and saves in the .csv file. It does this by first executing a simple incrementing base-loop (10000000 times), then executing a similar loop but with an additional memory write and memory read command via an array of data (whose size is set by `MAXIDX`). It extracts memory access time by subtracting the execution time of the base-loop to the execution time of the loop containing the additional memory access. This is how the program is able to provide the .csv file with data containing a memory access time for each value of stride.

Before inspecting the results of the experiment, it is possible to hypothesise an expected curve of memory access time against stride for a simple cache system. This curve, plotted in MATLAB using the principles explained in

section 1, models the memory access time using a sigmoid function fit. This is shown in figure 3.2.1.

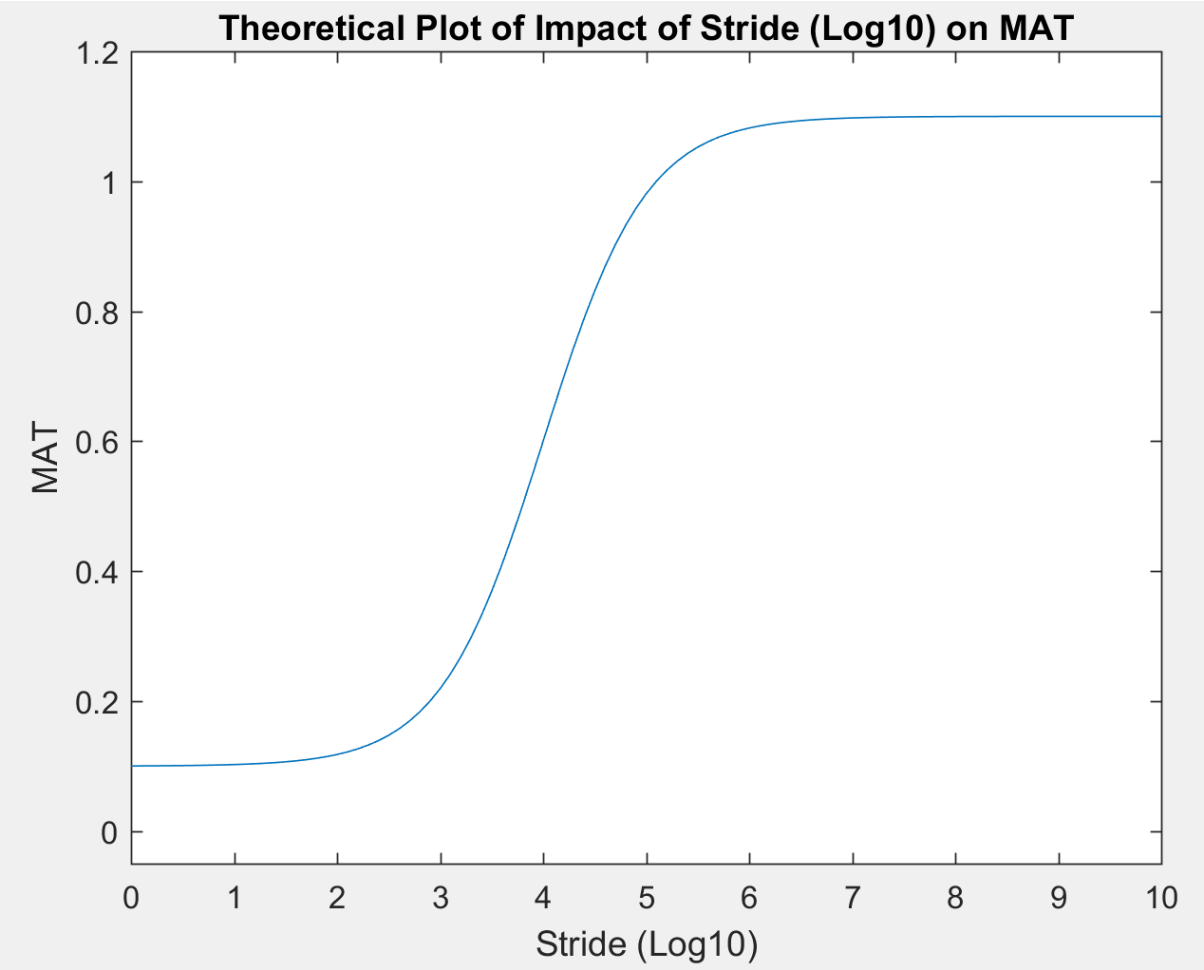


Figure 3.2.1 - Expected impact of stride (Log10) on MAT

The values used here for the memory access time (x axis) and for stride (y axis) are arbitrary as we are mostly concerned by the shape. For small values of stride, we expect locality of reference to be exploited as data will be concentrated in a small number of blocks thus CHR will be high (as explained in section 1). Therefore, the MAT will remain low for the start. As the stride is incremented up to the point where it approaching cache line size, MAT will start to increase dramatically as cache misses become increasingly statistically likely (again explained in section 1). It is worth noting that cache lines can wrap around to store data. The MAT increase does however trail off into a plateau as stride largely exceeds line size. At this stage, since the stride exceeds the line size, there will be a worst-case of one piece of data per block. Any further increase in stride will not

impact performance as the code's special exponential incrementation function circumnavigates wrap around, thus MAT reaches a plateau here.

Now that we have a theoretical prediction, we can verify this against the actual data. The data is presented as a logarithmic graph of memory access time against stride, as seen in figure 3.2.2 shown below.

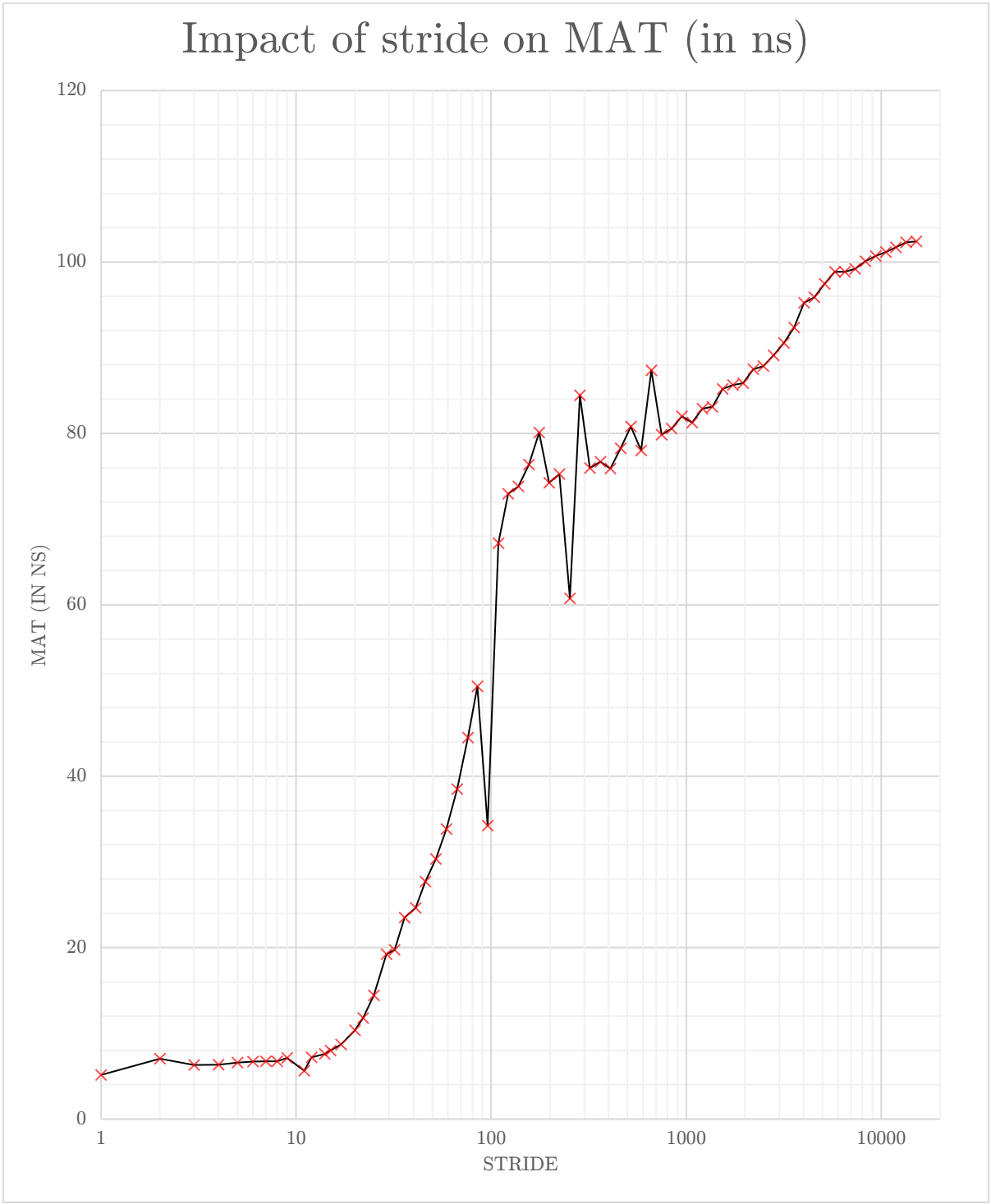


Figure 3.2.2 - Impact of stride on MAC

Figure 3.2.2 has a similar overall shape (sigmoid-like) to the predicted theoretical plot even if it contains two cache levels rather than one. By examining this figure 3.2.2 we can discover a lot of information regarding the processor's architecture, especially its cache line size. There are also additional features in figure 3.2.2 such as large drops that were not considered in the theoretical plot. These will be explained here.

For the starting values of stride (1 to 16 bytes), MAT remains very low around 3 to 7 ns. This is because the cache can exploit locality of reference (as seen in section 1) by storing data densely in cache lines (high hit rate).

The MAT begins to increase drastically after a stride of approximately 16. This is likely to be due to an increase in the cache line replacement which must now occur to deal with data being increasingly spread over a large number of blocks. This suggests between strides of 10-100 bytes, the value of stride is approaching the value of cache line size. From the literature value, we expect the graph to behave in this way, hence a cache line size of 64 bytes can be confirmed. At this stride, the MAT of 50ns reveals a CHR of ~50%, since at stride of 10000 MAT is ~100ns. This MAT increase continues until a stride of approximately 91 where a large drop is observed.

This large drop is likely to be due to the processor's L1 cache prefetching capabilities. Prefetching is a technique used by processors for speeding up access time during run-time by fetching data expected to be accessed in subsequent commands. Hence here the processor had prefetched the data in advance which meant at the stride of 90 it achieved a particularly fast memory access time of roughly 35ns. From this section at a stride of 90 to a stride of approximately 10000, it is estimated that the data is so spread out in memory blocks that both L1 and L2 caches have to undertake line replacement extremely frequently. This will cause increasingly slow MAT which becomes comparable to main memory read speeds.

This increase is interrupted by a drop at a stride of approximately 250. This second drop is likely due to the processor's L2 cache prefetching

capabilities. This prefetching was carried out in advance to prepare for slow access times. Thus, at a stride of around 250 there was a significantly faster memory access time.

After strides of 8000 onward, the start of a plateau can be seen. This suggests the increase in stride no longer worsens MAT since the worst-case scenario has been reached. Read speeds here are similar to main memory read speeds and caching has henceforth been defeated.

Other than the large drops which were not predicted by the theoretical plot, some small noisy spikes occur beyond strides of 200. This is because, unlike in our theoretical model, cache hits and cache misses in L1 and L2 are probabilistic. Hence, we notice a variance associated with the MAT. This is because there may be slightly 'lucky' or 'unlucky' iterations which will either access data mostly present in L1 & L2 caches or mostly present in main memory respectively. In addition, the plateau does not occur at a stride of 64 since the cache is set-associative rather than fully associative.

While the cache line size of 64 bytes can be correctly identified from this graph, one odd point can be seen which is the very first MAT at a stride of 1. This starting value seems faster than all the other MAT whereas one would expect this value to be, if anything, ever so slightly slower than other L1 cache memory access times due to the first reference miss. This speed increase could be explained by a software optimisation mechanism known as software prefetching. This occurs during the compiling of programs to speed up the first few memory accesses by predicting the data that is to be read during the program execution.

3.3. Cache Behaviour with Increasing Working Set Size

In this section we will use an in-lab modified program (based on the first program) to vary both the stride and the working set size in order to measure their impacts on memory access time. The developed code can be found in Appendix 6.3.

The main modification that was made to the code was to place the main section of the program in another `do while` loop. This loop allowed the program to, as well as incrementing stride, increment the working set size of the program. Therefore, the program operates as described in part 3.2 but iterates once for each working set size (incremented in exponential doubling from 1 to 2^{25}). This new program also includes an extra `fprintf` parameter near the end to ensure the additional data is written to a new column in the .csv file. This file is then used to plot 2D and 3D graphs for data analysis. The 2D log graph is shown in figure 3.3.1 below.

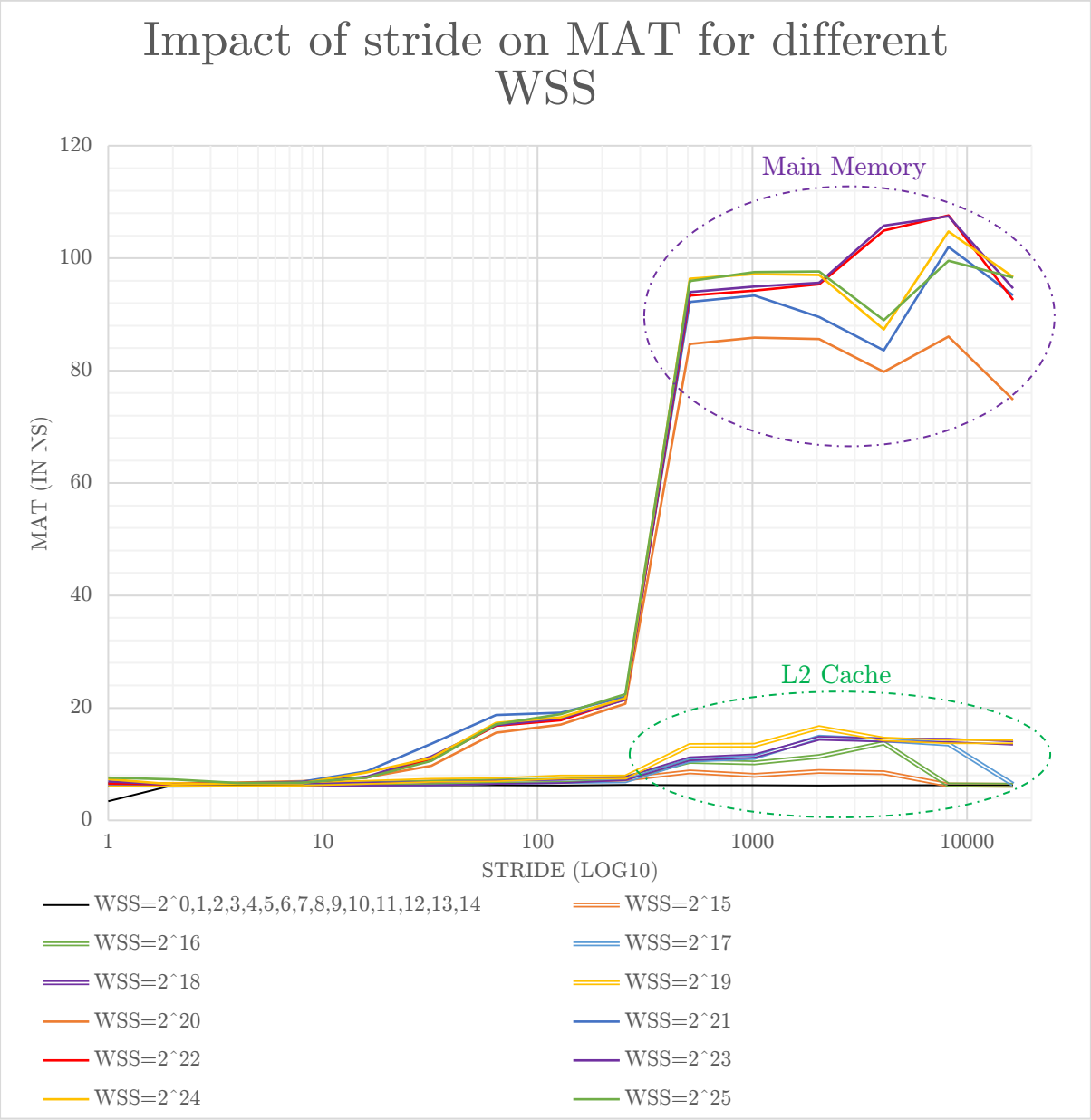


Figure 3.3.1 - Impact of stride on MAT for different WSS

Since 26 different working set sizes were investigated, overlapping lines were merged together (shown in a single black line in figure 3.4.1). This graph contains the same odd point identified in part 3.2 which is the very first memory access time when stride is 1 and working set size is 1. Once again this was unexpected but could be explained by software prefetching.

While the WSS remains below 2^{15}, the impact of stride on memory access time is negligible. This is because the program's WSS is so small that all its memory blocks will fit in the L1 cache irrespective of stride. Therefore, even if the data stored is in separate memory blocks and does not take advantage of locality of reference, the cache hit rate will remain high as all the data has been loaded into the L1 cache. This will result in MAT consistently around 6ns as shown by the black series in figure 3.3.1.

However, when working set size exceeds this threshold of 2^{14}, stride starts to have a small but not too dramatic impact on memory access time. For working set sizes 2^{15} to 2^{19}, stride starts to have an impact on memory access time beyond stride values of around 250. This is because when the working set size is between 2^{15} and 2^{19} some of the data will have to be placed in the L2 cache, which results in slightly slower memory access times. This is shown by double-lined series, circled in green in figure 3.3.1. This suggests the L1D cache can contain up to 2^{14} bytes (see code in appendix 6.3). As the `malloc` command is in bytes, the predicted L1D cache size is given by:

$$\hat{C}_{size} = WSS_{threshold} \qquad (2)$$

$$\therefore \hat{C}_{size(L1)} = 2^{14} = 16\text{KB}$$

This confirms the size of L1D data cache is 16KB, as in literature values.

For working set sizes of 2^{20} onward, stride impacts MAT even more than before which suggests the L2 cache is full since Main Memory accesses start to occur (single-lined series, circled in purple). Thus, the threshold WSS for L2 cache is 2^{19}. Again, from equation (2), the predicted L2 cache size is given by:

$$\hat{C}_{size(L2)} = 2^{19} = 512\text{KB} \qquad \text{from (2)}$$

This confirms the size of L2 cache which is 512KB, as in literature values.

While this 2D graph reveals some interesting features, a 3D plot may reveal some additional features. This is shown in figure 3.3.2 below.

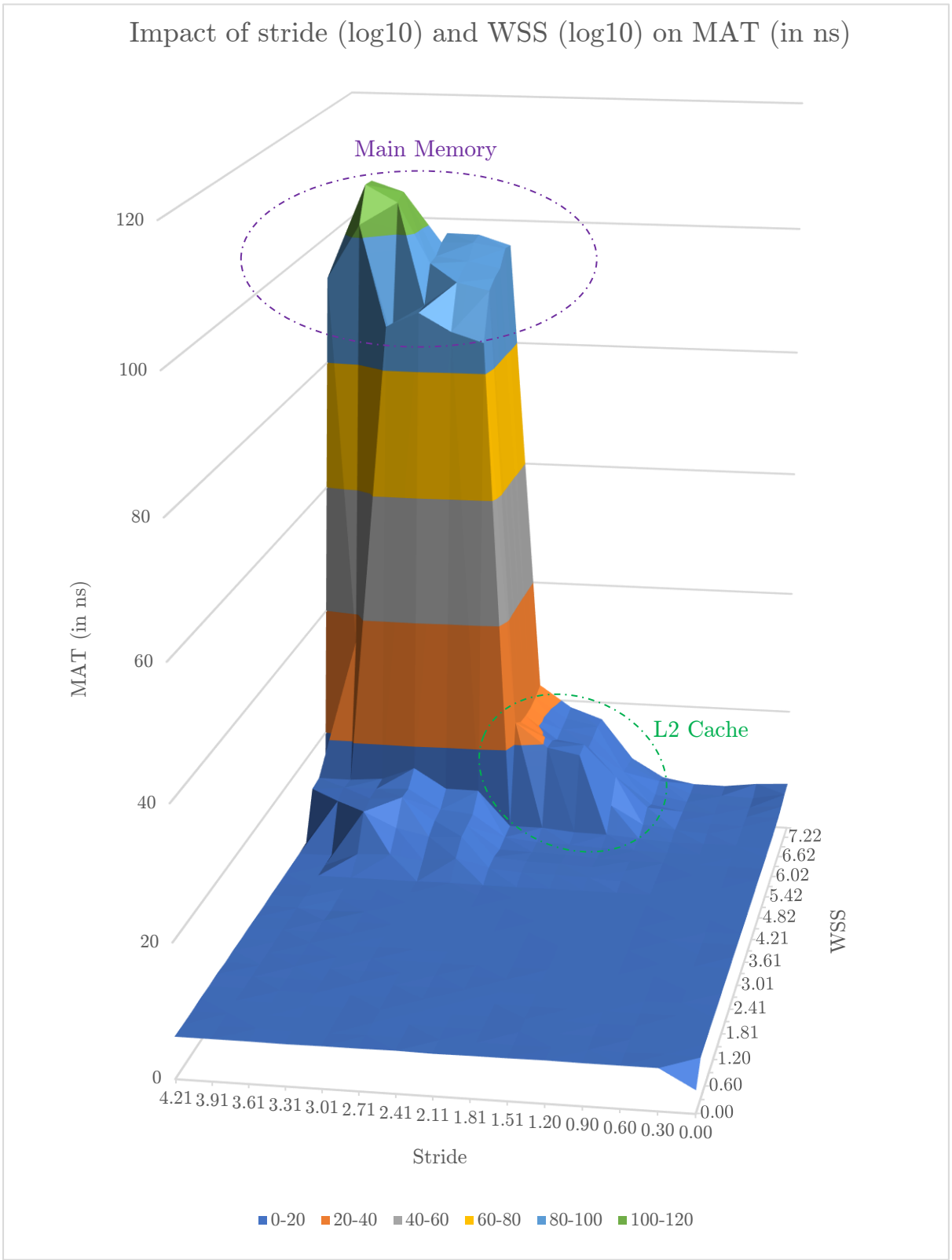


Figure 3.3.2 - Impact of stride (log10) and WSS (log10) on MAC

This 3D plot shows, once again, the odd point occurring when stride = 1 and WSS = 1.

Figure 3.3.2 also shows how different levels of memory access time are reached and through what path. This is shown by the multi coloured legend at the bottom of the figure.

The L1 dominated section is shown by the flat blue plateau at the bottom of the image.

The L2 dominated section is circled in green and labelled L2 cache, as in figure 3.3.1.

The main memory dominated section is circled in purple and labelled 'Main Memory', as in figure 3.3.1.

Both figures 3.3.1 and 3.3.2 clearly show that the ARM Cortex-A53 processor has two distinct levels of cache memory, L1 and L2. These two figures were also used to correctly calculate the sizes of both of these caches which were 16KB and 512KB respectively, in line with literature values.

One small technicality that is worth mentioning is the decrease in MAT that occurs for strides beyond 10000 bytes. This is expected to perhaps be a consequence of changing the incrementation strategy to ×2 exponential increase, causing wrap around data placement.

4. Conclusions

This laboratory has successfully run two experiments which used C code to defeat the Cortex-A53's caching mechanism. In doing so, various cache architecture concepts were explored and the experiments yielded results supported by both theory and literature values.

The theoretical fundamental concepts were first outlined to provide context for the experiments. These made predictions on expected cache behaviour and helped foreshadow the results to come.

The Cortex-A53 technical reference manual as well as other pieces of documentation were then used to build up a background baseline of knowledge so that collected data could be compared to the actual values.

During the first experiment, line size was successfully quantitatively ascertained as 64 bytes. The results contained one source of systematic error (odd point) and some sources of random error which were due to the statistical nature of caching algorithms. These respectively contributed to the slight decreased accuracy of the first MAT reading and the decrease in precision of the other MAT readings.

During the second experiment the size of the L1D cache was determined correctly as 16KB while the size of the L2 cache was also found to be in line with ARM's specifications at 512KB. Sources of random and systematic error, as before contributed to decreasing accuracy and precision slightly but the origin of such errors was identified and carefully considered.

Both the cache size and line values uncovered are quite typical of standard industry caches. The pseudo-random cache replacement algorithm, while not being theoretically most efficient, proves to still be used successfully in industry in general purpose computers. Some light has also been shed on concepts outside the scope of this report such as software prefetching during compile time and this report aims to help open the door to further investigations in computer systems architecture.

5. Bibliography

[1] Raspberry Pi Foundation. (2011). *Usage - Raspberry Pi Documentation* [Online]. Available: https://www.raspberrypi.org/about/

[2] Raspberry Pi Foundation. (2015). *Windows 10 for IoT* [Online]. Available: https://www.raspberrypi.org/blog/windows-10-for-iot/

[3] ARM. (2013). *Cortex-A53 Technical Reference Manual* [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500e/BABCFDAH.html

[4] I. Thornton (VP Investor Relations at ARM), private communication. (13 Nov. 2017).

6. Appendices

6.1. Information Gathering (Shell Script)

Script:

```
#!/bin/sh
#Author: 9563426
#Subject: Computer Systems Architecture

cat /proc/cpuinfo #Processor information

cat /proc/device-tree/model #Model information (only available on recent raspi distro)

sudo apt-get install htop #Install special procmon package

htop #CPU/GPU/RAM usage

free -h -s 1 #RAM usage (updates every second)

cat /etc/debian_version #Debian version information

uname -a #Kernal version information
```

Output:

```
             total       used       free     shared    buffers     cached
Mem:          925M       480M       445M        23M        30M       309M
-/+ buffers/cache:       140M       785M
Swap:          99M         0B        99M

             total       used       free     shared    buffers     cached
Mem:          925M       480M       445M        23M        30M       309M
-/+ buffers/cache:       140M       785M
Swap:          99M         0B        99M

             total       used       free     shared    buffers     cached
Mem:          925M       480M       445M        23M        30M       309M
-/+ buffers/cache:       140M       785M
Swap:          99M         0B        99M

             total       used       free     shared    buffers     cached
Mem:          925M       480M       445M        23M        30M       309M
-/+ buffers/cache:       140M       785M
Swap:          99M         0B        99M
```

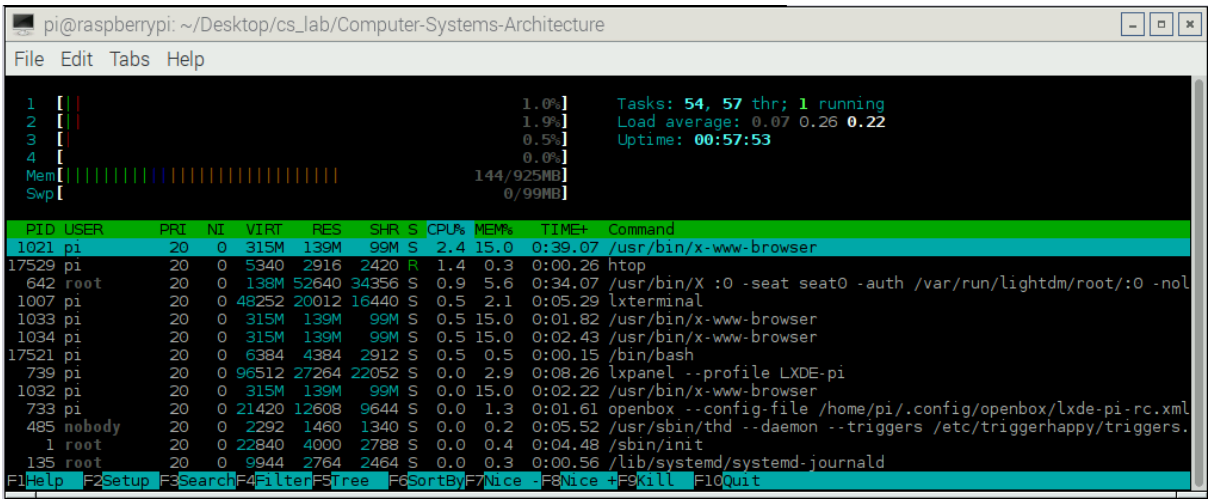
```
processor       : 0
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstr
m crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

processor       : 1
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstr
m crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

processor       : 2
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstr
m crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

processor       : 3
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 38.40
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstr
m crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

Hardware        : BCM2709
Revision        : a02082
Serial          : 0000000015987467
Raspberry Pi 3 Model B Rev 1.2 Reading package lists...
Building dependency tree...
Reading state information...
htop is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```



6.2. Increasing Stride (C Program)

```c
//0. Code information
    //Author: John Oakley
    //Subject: CSA
    //Year: 2017
    //Version: 2.0

//1. Var info
//wssize = working set size is the amount of memory needed to compute the answer to a problem.
//mat = Memory Access Time
//stride = the stride of an array (also referred to as increment, pitch or step size) is the number of locations in memory between beginnings of successive array elements, measured in bytes or in units of the size of the array's elements.

#include <stdio.h>
#include <stdlib.h>
#include <sys/timeb.h>
#include <math.h>

#define MAXindex 33554432
#define MAXSTRIDE 16384 // 4^7 = 2^14
#define NMAX 10000000

int main()
{
    struct timeb t1, t2;
    char *data;
    int index;
    int stride = 1;
    double rstride = 1.0;
    int n;
    char v;
    int secs, msecs;
    double elapsed_time, mat, mat_baseline;

    FILE *fp;
    fp = fopen("cache_arm.csv", "w");
    data = (char*) malloc(MAXindex);

    // create a large array of type byte
    do
    {
        // PART 1. measure the loop timing without the additional memory accesses...
        index = 0;
        ftime(&t1); // get the current time into t1

        for (n = 1; n < NMAX; n++) //increments the variable index by two 10M times
        {
            index = (index + stride) % MAXindex; // increment index, wrap if necessary
            index = (index + stride) % MAXindex; // increment index, wrap if necessary
        }
```

```
49
50          ftime(&t2); // get the current time into t2
51          secs = t2.time - t1.time;
52          msecs = t2.millitm - t1.millitm;
53          elapsed_time = secs + msecs / 1000.0; //elapsed time in
            seconds
54          mat_baseline = 1e9 * elapsed_time / (2 * NMAX); // writes
            memory access time (baseline) to increase index by 1 to
            variable (in nanoseconds)
55
56
57          // PART 2. measure the loop timing with the additional
            memory accesses...
58          index = 0;
59          ftime(&t1);
60
61          for (n = 1; n < NMAX; n++)
62          {
63              v = data[index];
64              index = (index + stride) % MAXindex;
65              data[index] = v;
66              index = (index + stride) % MAXindex;
67          }
68
69          ftime(&t2);
70          secs = t2.time - t1.time;
71          msecs = t2.millitm - t1.millitm;
72          elapsed_time = secs + msecs / 1000.0; //elapsed time in
            seconds
73          mat = 1e9 * elapsed_time / (2 * NMAX); // writes memory
            access time to acess a memloc & increment index by stride
            to variable (in nanoseconds)
74
75          fprintf(fp, "%d, %g\n", stride, mat - mat_baseline);
76
77          printf("%d, %g\n", stride, mat - mat_baseline);
78
79          do // ensure exponential increment of scride (with no
            duplicate integers - ?) until max stride
80          {
81              rstride = rstride * pow(2, 2.1/12.1);
82          } while ((int)rstride <= stride);
83
84          stride = (int)rstride;
85
86      } while(stride <= MAXSTRIDE);
87
88      fclose(fp);
89  }
```

6.3. Increasing Working Set Size & Stride (C Program)

```c
//0. Code information
    //Author: John Oakley
    //Subject: CSA
    //Year: 2017
    //Version: 2.0

//1. Var info
//wssize = working set size is the amount of memory needed to compute the answer to a problem.
//mat = Memory Access Time
//stride = the stride of an array (also referred to as increment, pitch or step size) is the number of locations in memory between beginnings of successive array elements, measured in bytes or in units of the size of the array's elements.

#include <stdio.h>
#include <stdlib.h>
#include <sys/timeb.h>
#include <math.h>

#define MAXIDX 33554432
#define MAXSTRIDE 16384 // 4^7 = 2^14
#define NMAX 10000000


int main()
{
    int wssize = 1;
    FILE *fp;
    fp = fopen("cache_arm.csv", "w");

    do{
        struct timeb t1, t2;
        char *data;
        int idx;
        int stride = 1;
        double rstride = 1.0;

        int n;
        char v;
        int secs, msecs;

        double et, mat, mat_baseline;


        data = (char *)malloc(wssize); //create a large array of type byte

        do {
            //measure the loop timing without the additional memory accsses...
            idx = 0;
            ftime(&t1); //get the current time into t1

            for (n = 1; n < NMAX; n++)
            {
                idx = (idx + stride) % wssize; // increment idx, wrap if necessary
                idx = (idx + stride) % wssize; // increment idx, wrap if necessary
            }
```

```
54
55              ftime(&t2); //get the current time into t2
56              secs = t2.time - t1.time;
57              msecs = t2.millitm - t1.millitm;
58              et = secs + msecs / 1000.0;     //elapsed time in seconds
59              mat_baseline = 1e9 * et / (2 * NMAX); //print in
                nanoseconds
60
61              //measure the loop timing with the additional memory
                accesses ...
62              idx = 0;
63              ftime(&t1);
64
65              for (n = 1; n < NMAX; n++)
66              {
67                  v = data[idx];  //additional memory access
68                  idx = (idx + stride) % wssize;
69                  data[idx] = v;  //additional memroy access
70                  idx = (idx + stride) % wssize;
71              }
72
73              ftime(&t2);
74              secs = t2.time - t1.time;
75              msecs = t2.millitm - t1.millitm;
76              et = secs + msecs / 1000.0;     //elapsed time in seconds
77              mat = 1e9 * et / (2 * NMAX); //print in nanoseconds
78
79              fprintf(fp, "%d, %d, %g\n",wssize, stride, mat -
                mat_baseline);
80              printf("%d, %d, %g\n",wssize, stride, mat - mat_baseline
                );
81
82              stride = 2*stride;
83
84          } while (stride <= MAXSTRIDE);
85
86          wssize = 2*wssize;
87
88      }while(wssize <= MAXIDX);
89
90      fclose(fp);
91  }
```