INTRODUCTION TO ADVANCED STUDIES-II (CS-402-01)

HOMEWORK-1

GitHub Repository link: https://github.com/kartheekbasava/CS402-HW

1. (a) Using trace files, i.e. files that contain addresses issued by some CPU to execute some application(s), draw the histogram of address distribution for each of them (2x20 points). On the Ox axis of the plot, you will have the address as a decimal number. On the Oy axis, you will have the number of occurrences for each particular address.
NOTE: the range of addresses is vast, attempting to plot everything will result in a histogram with very little detail. Instead, select a range of addresses (a few hundred of them) where you have non-zero values on Oy.
Comment based on the histograms (5). The files to use are:
cc1.din
spice.din

The first file contains the trace obtained by the CC compiler compiling itself, and the second one comes from the running SPICE ("Simulation Program with Integrated Circuit Emphasis"), the general-purpose, open-source analog electronic circuit simulator.

Each line in the file has two fields: the first field indicates what kind of operation the CPU performs (read, write, etc.), and the second field is the address. Here is what the number in the first field means:

0: read data
1: write data
2: instruction fetch

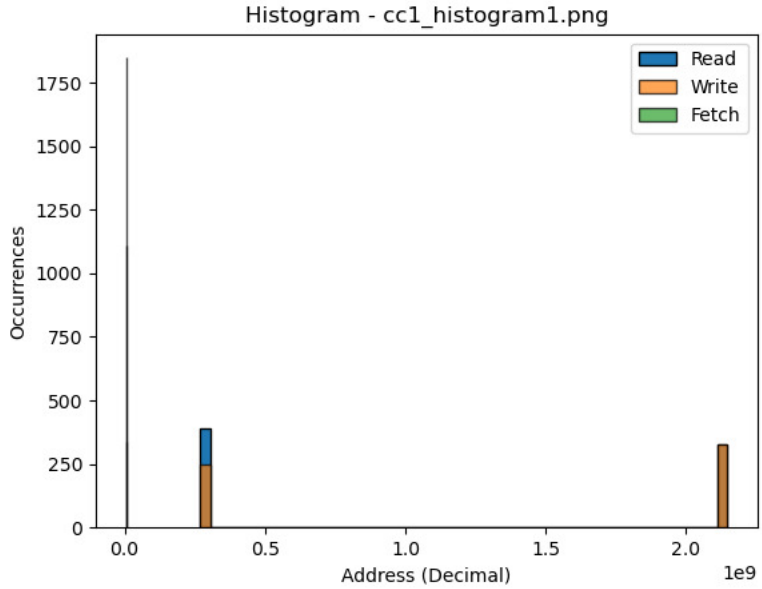
The second field is the address being referenced: the address is a hexadecimal byte address between 0 and ULONG_MAX inclusively.

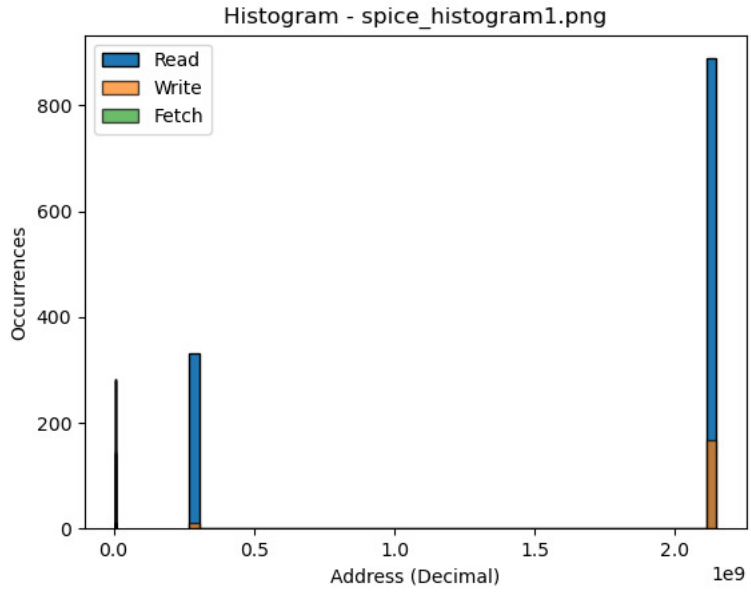
Ans) The following Histograms are plotted using the Python code on a Virtual Linux Machine.

The Python code is attached in the zip file.

The Python code uses matplotlib. pyplot library to provide MATLAB-like interface and numpy for numerical operations. We provided the file path of the given files “cc1.din”&”spice.din” , before providing the files as mentioned in the question I considered only a few hundred of addresses from both files. The code generates Histograms for cc1.din and also for spice.din , displaying Read, Write, and Fetch in the graph.

As told y axis is for occurrences and x axis is for Addresses in decimal.





Comments on the Histograms of cc1.din and spice.din

- The main differences were observed in the range of addresses where cc1.din had a wider range and spice. din had a narrower range.
- Distribution of addresses peaked by 50 for cc1.din, suggesting a focus on specific memory locations, and for spice. din it was more uniform, which indicates memory access address more evenly.
- Cc compiler exhibits a more localized memory access pattern during its execution. While the Spice accesses memory addresses more uniformly.

- This is due to the different nature of the tasks each program performs. The compiler likely needs to repeatedly access specific memory locations to store and retrieve intermediate results during compilation, while the simulator may access memory more evenly as it performs calculations across the entire circuit.
- The tails of the distribution for cc1.din fall off more rapidly than those for spice.din. This might indicate that the compiler accesses a smaller set of addresses outside its main focus area around address 50 and similarly the longer tails in spice.din suggests it potentially accesses a wider range of addresses compared to the compiler, possibly due to its need to interact with various components of the simulated circuit.
- The cc1.din looks for any smaller spikes within the main peak of the cc1.din histogram. These could indicate repeated access to specific sub-routines or data structures during compilation whereas the spice.din: Identify any notable spikes in the spice.din histogram beyond the general uniform distribution. These might hint at specific memory locations involved in critical calculations or data exchanges within the simulation.

b) What is the frequency of writes? What is the frequency of reads?

- From the Python code used above, we are able to generate the frequencies of write and read.

Ans) For cc1.din

Frequency of Writes (cc1): 575
Frequency of Reads (cc1): 719

These results indicate that in the trace file generated by the CC compiler (cc1.din.txt), there are 575 write operations and 719 read operations. The balance between writes and reads might suggest a balanced workload, where the CPU performs a significant number of both write and read operations.

For Spice.din

Frequency of Writes (spice): 177
Frequency of Reads (spice): 1221

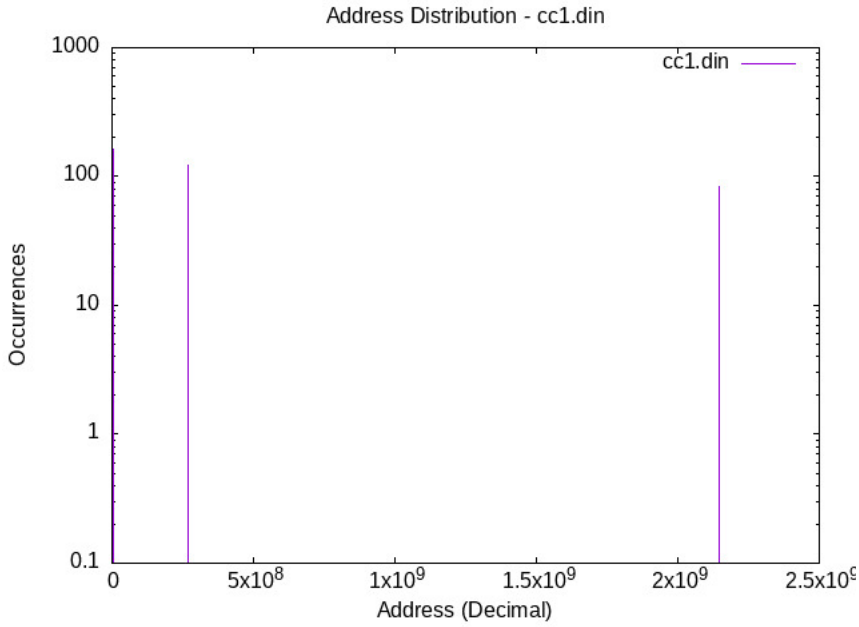
In the trace file from running the SPICE simulator (spice.din.txt), there are 177 write operations and 1221 read operations. The higher frequency of reads compared to writes may indicate that the SPICE simulation involves more read-heavy operations. This could

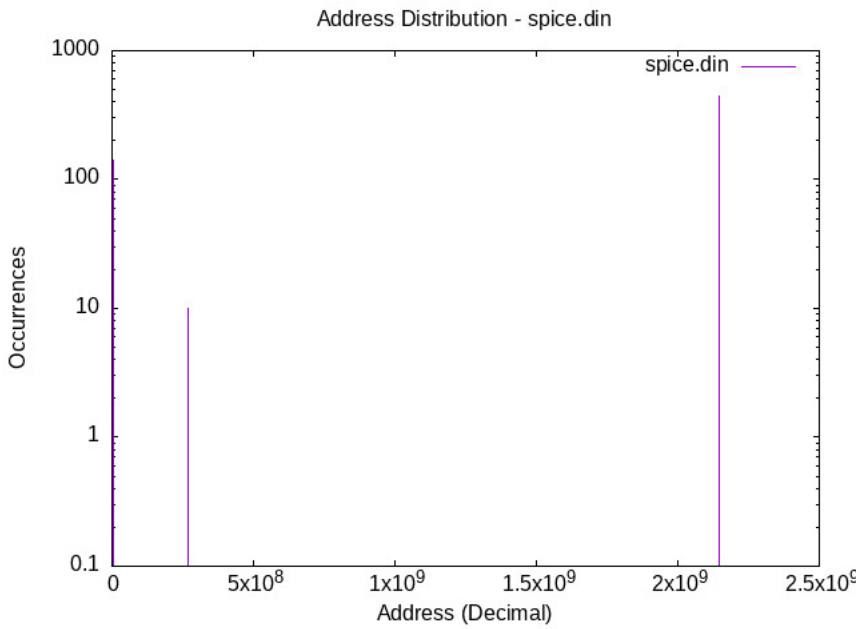
be expected in a simulation scenario where the simulator is fetching and processing a large amount of input data.

- A higher frequency of reads may suggest that the workload is more data-intensive, requiring the CPU to fetch and process information frequently.
- A balance between writes and reads might indicate a workload that involves both data processing and data storage.

Note: I am able to plot the histograms by using the gnuplot and awk, sort, and uniq commands, and also for the same I am providing the histograms below. As gnuplot is having high precision it could not have a clear histogram.

```
┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ awk '$1 == 0 || $1 == 1 || $1 == 2 {print $2}' cc1.din.txt | sort | uniq -c | awk '$1 > 1 {print $2, $1}' | sort -k2,2nr | head -n 200 > cc1_histogram_data.txt

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ awk '$1 == 0 || $1 == 1 || $1 == 2 {print $2}' spice.din.txt | sort | uniq -c | awk '$1 > 1 {print $2, $1}' | sort -k2,2nr | head -n 200 > spice_histogram_data.txt

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ nano plot_histograms.gp

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ nano plot_histograms_spice.gp

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ gnuplot plot_histograms.gp

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ gnuplot plot_histograms_spice.gp
```





The frequencies of read, and write for gnuplot histograms are as same as the results we got by using Python I,e.
Frequency of Writes (cc1): 575
Frequency of Reads (cc1): 719
Frequency of Writes (spice): 177
Frequency of Reads (spice): 1221

```
┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ # Frequency of Writes (cc1)
awk '$1 == 1 {count++} END {print "Frequency of Writes (cc1):", count}' cc1.din.txt
Frequency of Writes (cc1): 575

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ # Frequency of Reads (cc1)
awk '$1 == 0 {count++} END {print "Frequency of Reads (cc1):", count}' cc1.din.txt
Frequency of Reads (cc1): 719

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ # Frequency of Writes (spice)
awk '$1 == 1 {count++} END {print "Frequency of Writes (spice):", count}' spice.din.txt
Frequency of Writes (spice): 177

┌──(kali㉿kali)-[~/Downloads/Scripts]
└─$ # Frequency of Reads (spice)
awk '$1 == 0 {count++} END {print "Frequency of Reads (spice):", count}' spice.din.txt
Frequency of Reads (spice): 1221
```

Description of the 2 Systems:

Details	Mac OS	Linux
Manufacturer	Apple Inc.	AuthenticAMD
CPU - type	Apple M1	AMD Ryzen 7
Amount of Memory	8 GB 2133 MHz LPDDR3	59 gb
Operating Sysytem	macOS	Kali GNU/Linux
Compiler	Xcode, Apple's official integrated development environment (IDE)	gcc (Debian 13.2.0-5) 13.2.0
Processor	1.6 GHz Dual-Core Intel Core i5	1

S.NO	Time lapse for integers on MAC (in nano-seconds)	Time lapse for integers on LINUX (in nano-seconds)
1	3792777764	19106883474
2	3748328880	16913326132
3	3542594814	18249645614
4	3316731088	16795097844
5	4011789698	16832561492
Average	3682444448.8	17579502911.2

S.NO	Time lapse for Double on MAC (in nano-seconds)	Time lapse for Double on LINUX (in nano-seconds)
1	4098666477	20505201472
2	4255600711	22359384710
3	5397760946	20252238332
4	5119712308	20640234612
5	4519723529	21235963487
Average	4678292794.2	20998604522.6

S.NO	Time-lapse for interchange of matrix, integers on MAC (in nano-seconds)	Time-lapse for interchange of matrix, integers on LINUX (in nano-seconds)
1	3886139267	17939341229
2	3883740305	17455125411
3	4787648877	17916637516
4	4423993021	18171154831
5	4246578477	17862296099
Average	4245619989.4	17868911017.2

S.NO	Time-lapse for interchange of matrix, Double on MAC (in nano-seconds)	Time-lapse for interchange of matrix, Double on LINUX(in nano-seconds)
1	5134555352	21669772631
2	4894412529	20569177008
3	5090689593	20661432819
4	4988128785	20316643353
5	5060080355	27024852254
Average	5033573322.8	22048375613

Calculating the Performance ratio:

- **for timelapse of integers**: System 1(MAC)/System2(Linux)

(3682444448.8/17579502911.2)

= 0.20947375289

=if we multiply by 100 it will become 20.9473

It implies macOS is 20.94% faster than the Linux

for timelapse of double: System 1(MAC)/System2(Linux)

(4678292794.2/20998604522.6)

=0.22279065207

= if we multiply by 100 it will become 22.279

It implies macOS is 22.279% faster than the Linux

Calculating the Performance ratio:

- **for timelapse of interchange of matrix in integers**: System 1(MAC)/System2(Linux)

(4245619989.4/17868911017.2)

= 0.2375981382

=if we multiply by 100 it will become 23.75

It implies macOS is 23.75% faster than Linux

for timelapse of interchange of matrix double: System 1(MAC)/System2(Linux)

(5033573322.8/22048375613)

= 0.22829678753

= if we multiply by 100 it will become 22.82

It implies macOS is 22.82% faster than the Linux

Calculating the clock rate ratio:

Mac/Linux

= 1.5 /1

= 1.5

=1.5*100

=150

It implies macOS is having 150% more clock rate than LINUX

Q) Based on the retail price of the two systems, which one is more cost effective.

Ans) Linux is generally more cost-effective than macOS. Linux operating systems are open-source and often available for free, reducing upfront costs. Additionally, Linux can run on a wide range of hardware, including older machines, extending the lifespan of existing equipment. In contrast, macOS is exclusive to Apple hardware, which tends to be more expensive. While macOS offers a polished user experience, Linux provides a cost-effective solution for users seeking robust functionality without the premium price tag associated with Apple products.