Computer Architecture Assignment-2

Megha Patidar, Sr.No : 24758

meghapatidar@iisc.ac.in

1. Introduction

As outlined in the assignment, finding sufficient physical memory contiguity to allocate large or huge pages is often challenging. Therefore, it's crucial to carefully select the virtual address regions (here I have used 2MB regions) to minimize TLB misses for an unknown workload. This can be achieved by using the **perf mem tool** to capture the memory addresses where TLB misses occur.

Once we have these addresses, we can group them into 2MB regions and analyze the number of misses within each region with the help of python script file named **analyze.py**. The goal is to identify and allocate 'n' optimal 2MB regions in the main.c file to effectively reduce the total number of TLB misses.

2. Methodology

We will discuss the methodology and commands in stepwise manner as mentioned in the assignment.

- **Step 1: Collect a sample of the memory accesses**

As per given instructions for step one, we need to run following commands:

a. SRNO=24758 make run

Working: This command will look for a file name MakeFile, which contains some set of instructions to compile and run the given program. Typically, it is used to build and execute a program.

Output:

```
gcc -g -Wall -Wextra -Werror -O2 main.c -L. -l:libwork.so -o main
./main 24758
Base: 0xae40000000 End: 0xae80000000 Size: 40000000 bytes
Done work, time was 31.426232720
Work completed successfully
```

b. perf mem record -o access_data_before.perf -- ./main 24758

Working: This command analyzes a specific run of program (triggered by make run), along with memory related performance monitoring using perf and generated data is saved in a file. The program's behaviour is also customized by setting the environment variable SRNO=24758, which is my serial number.

- **Perf mem record :** This part of the command tells perf to record memory-related events (such as load/store accesses, virtual addresses or memory-related cache misses) while running a specific task. The record subcommand means that perf will capture performance data(sampled memory access trace) while the program runs, which can be analysed later.
- **-o access_data_before.perf :** -o specifies output file where the performance data will be saved, here it will be saved in file named as access_data_before.perf.

- **./main 24758:** It is representing that program main, which is in current directory, is being run by passing a required argument value 24758(my serial number).

Output:
Base: 0xae40000000 End: 0xae80000000 Size: 40000000 bytes
Done work, time was 31.135236360
Work completed successfully
[perf record: Woken up 36 times to write data]
[perf record: Captured and wrote 9.021 MB access_data_before.perf (130985 samples)]

c. perf mem report -i access_data_before.perf > address_1.txt
Working: This command reads the data collected from earlier command, read it from input file name access_data_before.perf and instead of printing the output on the terminal, it saves into a file name address_1.txt

- **Step 2: Write a Python script to find the most optimal locations for large pages**

After collecting sampled memory access trace, we need find virtual address regions(2MB), which would be most beneficial to be mapped using 2MB large pages. For implementing this task, we were asked to write a python script file name analyze.py.

Logic: I have divided the whole address space into 2MB regions, counted number of L2 miss count(TLB misses) in each region and then used n regions with most TLB misses for better performance of the workload. L2 misses are considered as TLB misses because if something is misses in L2 then by default it was a miss in TLB. (Additional- I have also thought to implement this by some other logic. I was thinking to first extract the addresses having L2 miss in given address range and store the addresses in sorted manner. After storing, we will analyse various continuous 2MB regions and find ones with most TLB misses, but I failed to conclude a concrete logic for implementing the last part of my logic.)

The explanation of script file is as follows:

a. I have started by defining the range of address space in integer, as outlined in Step 1 output and specified the region size, which is 2MB. I have also created a dictionary miss_count_region that stores L2 miss count for each region, initializes it to 0.
b. Function get_region is used to map addresses to 2MB regions. It takes memory address as parameter and checks if it falls under mentioned address range, if it does then calculate which 2MB region and return the region index.
c. After above initialization, code opens address_1.txt file which was generated earlier, reads and checks it with the regular expression which is written specifically to find L2 miss and it's corresponding address. The extracted hexadecimal address is converted in integer and passed to function get_region, to determine its belonging region. If address is within specified range, then increment the miss count by 1.
d. Sorted_regions is the sorted version of miss_count_region dictionary, sorted on the basis of L2 miss count.
e. Further, I have defined value of n (number of regions). N number of regions with highest miss counts are identified and stored in an array. The addresses of these regions are then stored in a file named 'largepages.txt' in decimal format, one per line.
f. At last, I have plotted a graph to visualize the TLB misses in different regions.

- **Step 3: Use large pages in the program**

Lastly, we were supposed to change marked section in main.c file to use large pages of size 2MB, each at the virtual address regions that I chose in above step. I have used the file largepages.txt (generated in above step) for reading addresses as unsigned long int (uintptr_t type is used to ensure that address values can hold pointer values accurately). Further, the code iterates through 'n' addresses (stored in array), allocate a large page at each address using mmap. Madvise system call function is used to suggest kernel to handle mapped regions as large pages.

Before running this code, we need to set the huge pages configuration through kernel by using command:

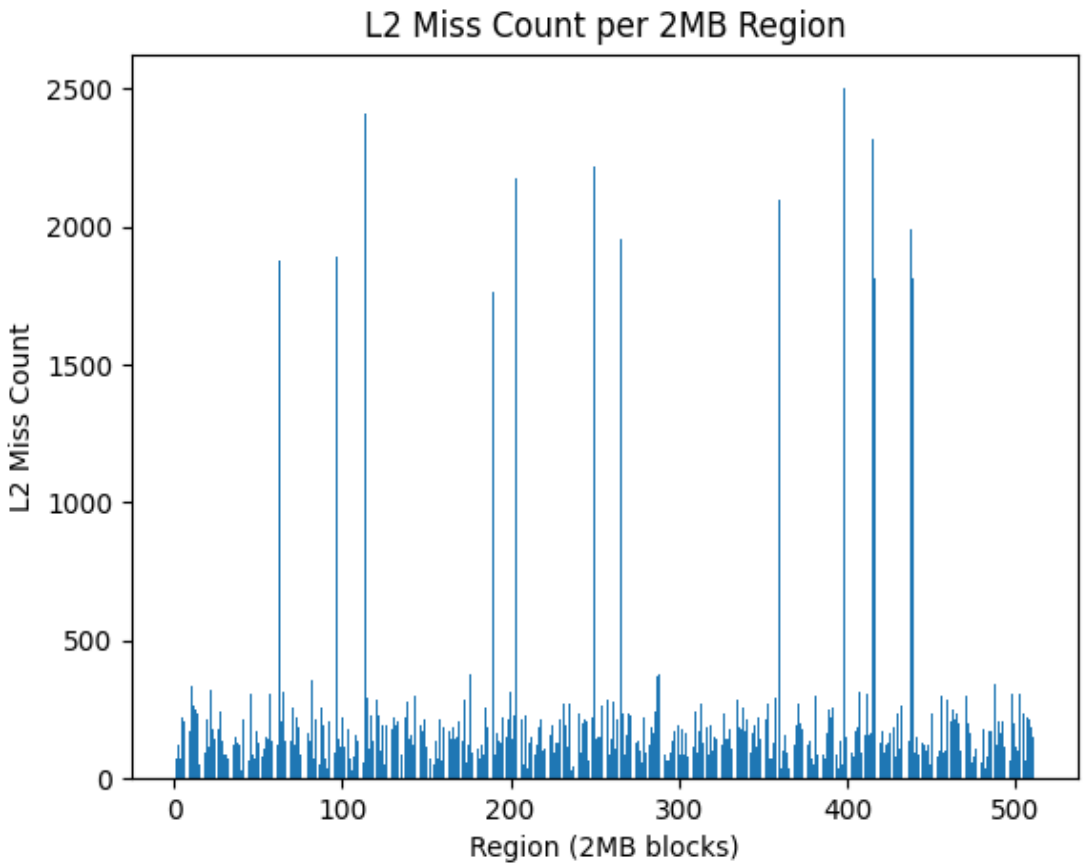
➔ sudo sysctl -w vm.nr_hugepages=n

And can verify the allocation using :

➔ grep Huge /proc/meminfo

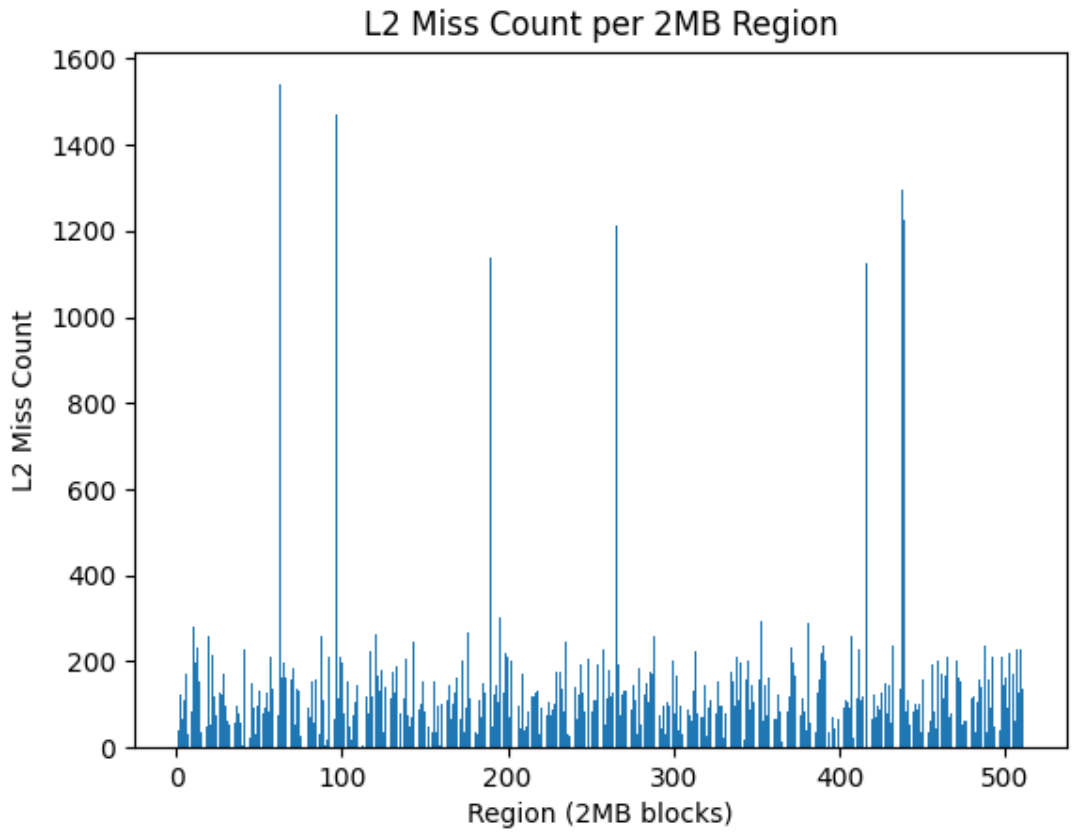
3. Graphs

a. Before allocating huge pages:



b. After allocating huge pages:

Following is the graph for n=8:



4. Results and conclusion

We can clearly see from the graphs mentioned above that the TLB misses (L2 misses) have significantly decreased, which shows that my approach is correct. I'm not claiming about its optimality because I know that there are more ways to solve this, and one of the approaches I have mentioned above may decrease the misses even more. The decrease in number of TLB misses can also been seen by 'perf stat' command.

- **Before allocating large/huge pages:**

Command – perf stat -e dTLB-load-misses,dTLB-loads ./main 24758
Output-
Base: 0xae40000000 End: 0xae80000000 Size: 40000000 bytes
Done work, time was 30.752708707
Work completed successfully

Performance counter stats for './main 24758':

7,033,843,582 dTLB-load-misses # 54.05% of all dTLB cache accesses
13,013,800,072 dTLB-loads

30.753901739 seconds time elapsed

30.752993000 seconds user
0.0 sys

- **After allocating large/huge pages:**

Commands–

1. SRNO=24758 make run
   Output-
   gcc -g -Wall -Wextra -Werror -O2 main.c -L. -l:libwork.so -o main
   ./main 24758
   Base: 0xae40000000 End: 0xae80000000 Size: 40000000 bytes
   Done work, time was 19.511888168
   Work completed successfully
2. perf mem record -o access_data_before.perf -- ./main 24758
   Output-
   Base: 0xae40000000 End: 0xae80000000 Size: 40000000 bytes
   Done work, time was 31.135236360
   Work completed successfully
   [perf record: Woken up 36 times to write data]
   [perf record: Captured and wrote 9.021 MB access_data_before.perf (130985 samples)]
3. perf stat -e dTLB-load-misses,dTLB-loads ./main 24758
   Output-
   Base: 0xae40000000 End: 0xae80000000 Size: 40000000 bytes
   Done work, time was 19.551050319
   Work completed successfully

   Performance counter stats for './main 24758':

   4,386,623,265 dTLB-load-misses # 33.72% of all dTLB cache accesses
   13,009,646,881 dTLB-loads

   19.552224887 seconds time elapsed

   19.547481000 seconds user
   0.003999000 seconds sys

In conclusion, we can see that my approach is decreasing the TLB misses along with the execution time of the workload. So, if one wants to decrease their execution time and TLB misses, finding addresses for maximum TLB misses and then allocating it for execution is an effective approach.

5. References

[1]https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/monitoring_and_managing_system_status_and_performance/profiling-memory-accesses-with-perf-mem_monitoring-and-managing-system-status-and-performance#profiling-memory-accesses-with-perf-mem_monitoring-and-managing-system-status-and-performance

[2]https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/monitoring_and_managing_system_status_and_performance/profiling-memory-accesses-with-perf-mem_monitoring-and-managing-system-status-and-performance#profiling-memory-accesses-with-perf-mem_monitoring-and-managing-system-status-and-performance

[3] https://www.man7.org/linux/man-pages/man2/madvise.2.html

[4] https://www. man7.org/linux/man-pages/man2/mmap.2.html

[5] https://www.google.com/