# Assignment 2: Optimizing Memory Usage with Large Pages to Minimize TLB Misses

E0243 High Performance Computer Architecture

**Nehal Shah**
SR - 24623
nehalshah@iisc.ac.in

October 18, 2024

---

This report presents the analysis and optimization of memory performance by reducing **TLB** (Translation Lookaside Buffer) misses through the allocation of large pages (2MB). The objective was to identify virtual address regions with frequent TLB misses and map them using large pages to enhance overall performance. TLB misses occur when the mapping from virtual to physical memory is not found in the TLB, leading to performance penalties. Using the **perf** tool, we analyzed memory access patterns to locate these high-miss regions and modified the memory allocation strategy by employing the **mmap** system call to allocate large pages (2MB) in these hotspots. This approach effectively minimized TLB misses, as fewer TLB entries were needed to map larger memory regions, resulting in a significant improvement in the workload's execution speed.

## Command Used to Collect Memory Access Data :

**perf** is a widely-used Linux performance measurement tool that monitors various Performance Monitoring Unit (PMU) events. In this project, we used perf to track memory access events, particularly focusing on TLB misses. These events help in identifying the memory regions where large page allocation could be beneficial.

**Commands:**

- The following command collect memory access and generates a file (**perf.data)** that captures memory access information including TLB misses, which is critical for optimizing memory usage.

```
perf mem record -o perf.data make run SRNO=24623
```

  This **mem record** sub-command focuses on recording memory access events, allowing for detailed analysis of memory behavior during program execution.

  The **-o perf.data** specifies the output file where the recorded memory events will be stored. In this case, the data will be saved to perf.data, which can later be analyzed.

  The **make run** command invokes the make tool to build and run the project defined in the Makefile. The run target typically executes the program with the defined settings and dependencies.

  The **SRNO=24623** sets an environment variable SRNO to 24623 for the duration of the make run command.

- The following command is used to convert the collected perf data into a human-readable text report:

```
perf report -i perf.data --stdio > perf_script_output.txt
```

  This **perf report** command reads performance data from the **perf.data** file and generates a detailed analysis report in text format.

The **-i perf.data** option specifies the input file containing the recorded data.

The **--stdio** flag ensures that the report is output in a text-based format.

The **>** operator redirects the output to **perf_script_output.txt**, saving the generated report in a file for further analysis or reference.

## Analysis Methodology :

### 1. Data Processing and Analysis

The script **analyze.py** processes memory access data with the following steps:

- ***Collecting Memory Access Data***: The script first runs the command perf mem record -o perf.data -- make run SRNO=24623 to collect memory access traces. These traces, saved in a binary file perf.data, include information about TLB misses and are critical for identifying memory usage patterns.
- ***Converting Perf Data:*** After collecting the data, the script converts the binary perf.data file into a readable format using the command perf mem report --stdio. The output is stored in a text file named perf_script_output.txt, which is analyzed to extract memory access information.
- ***Parsing Memory Access Data:*** The script applies a regular expression to the memory report in perf_script_output.txt to extract addresses associated with L2 misses. A crucial constraint is applied during this step: only memory addresses that fall within the range between the **base** and **end** addresses (determined during the data collection process) are considered for analysis. This ensures that only the relevant memory regions are processed. Each memory address is grouped into its corresponding 2MB region, and a tally of L2 misses is maintained for each region.
- ***Identifying Optimal Regions:*** The script sorts the 2MB regions by the number of L2 misses and selects the top n regions with the highest miss counts. These regions are considered the most optimal for large-page allocation, as reducing the number of TLB misses in these areas would yield the highest performance improvement.
- ***Saving Optimal Regions:*** The top n regions, in decimal format, are saved to **largepages.txt**. Each line in this file represents the base address of a 2MB region that should be mapped to a large page.By including the constraint that only memory addresses within the collected base and end addresses are analyzed, the script ensures that only the most relevant parts of the program's address space are optimized for large-page allocation.

To run the script analyse.py the below is the command we will use.Here n is number of top regions we want in lagepages.txt file in my case it is 8

```
python analyse.py 8
```

### 2. Memory Allocation Enhancement

After identifying regions with the highest TLB miss rates, the original main.c file is modified to allocate 8 large 2MB pages at these virtual addresses using mmap. mmap creates a direct mapping between a memory region and a process's address space, allowing the program to access memory like regular memory through pointers. This approach bypasses the need for standard file I/O functions like read() or write(), offering more efficient memory access. The modification dynamically adjusts memory allocation for up to 8 large pages, optimizing access in the regions with the highest miss rates and reducing TLB misses.These are the modifications made in main.c for the allocation of memory:

This below code declares a pointer ptr that will hold the starting address of the memory region allocated by mmap.

```
void *ptr = mmap((void *)address, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE |
                MAP_ANONYMOUS | MAP_FIXED | MAP_HUGETLB, -1, 0);
```

***mmap((void *)address, PAGE_SIZE, ...):*** mmap is a function used to map a region of memory into the process's address space.In which (void *)address specifies the starting address for the memory mapping. Using a cast to (void *) ensures that the address is treated as a pointer.

***PAGE_SIZE:***This defines the size of the memory region to be allocated (e.g., 2 MB for huge pages).

***PROT_READ | PROT_WRITE:***Specifies the protection level of the memory.That memory can be read and written too.

***MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED | MAP_HUGETLB:***Various flags that modify the behavior of the mapping.

***-1, 0:***-1 Indicates that no file descriptor is used since MAP_ANONYMOUS is specified.0 Offset for the mapping, which is ignored in this case because there's no file.

### 3. Performance Testing and Validation

After implementing the large-page allocation in the main.c file, the program is executed to verify the performance improvements. The number of TLB misses and overall program speed are compared before and after the large-page deployment.

Testing is done to ensure that the program works for any given value of n (the number of large pages). The results are analyzed to validate the effectiveness of the large page allocation in reducing TLB misses and improving performance. Performance gains are quantified by measuring the speedup and reduction in TLB misses.

## Results and Graphs :

In this section, we analyze the number of TLB misses for different 2MB virtual address regions in the program's address space, both before and after the allocation of large pages. We focus on the top n regions with the highest TLB misses, as determined by the analysis script (**analyze.py**), which saves the optimal regions in the **largepages.txt** file.

| 2MB Virtual Address Region | TLB Misses |
|---|---|
| 0x187b5000000 | 4257 |
| 0x187b1a00000 | 2297 |
| 0x18782400000 | 2287 |
| 0x187b0000000 | 2239 |
| 0x1879d800000 | 2207 |
| 0x18783400000 | 2192 |
| 0x187b9200000 | 2184 |
| 0x187b7800000 | 2164 |

**Table.1 Largepages.txt file before memory allocation**

| 2MB Virtual Address Region | TLB Misses |
|---|---|
| 0x187b1400000 | 1583 |
| 0x187aec00000 | 1550 |
| 0x187a1e00000 | 1504 |
| 0x1878e400000 | 1210 |
| 0x18782c00000 | 1149 |
| 0x187bfe00000 | 1000 |
| 0x18787e00000 | 944 |
| 0x18793000000 | 311 |

**Table.2 Largepages.txt file after memory allocation**

Above are the top 8 virtual address regions from the Largepage.txt file, which exhibit the highest miss counts. A comparison graph of approximately 512 2MB regions has been generated based on the trace data.

During data collection using perf commands, the base address was recorded as **0x18780000000**, and the end address was **0x187c0000000**. This corresponds to a memory size of approximately **40000000 bytes** (about 1 GB).

Two graphs are presented to compare the TLB miss distribution before and after the allocation of large pages The x-axis represents different 2MB virtual address regions, while the y-axis represents the number of TLB misses for each region.

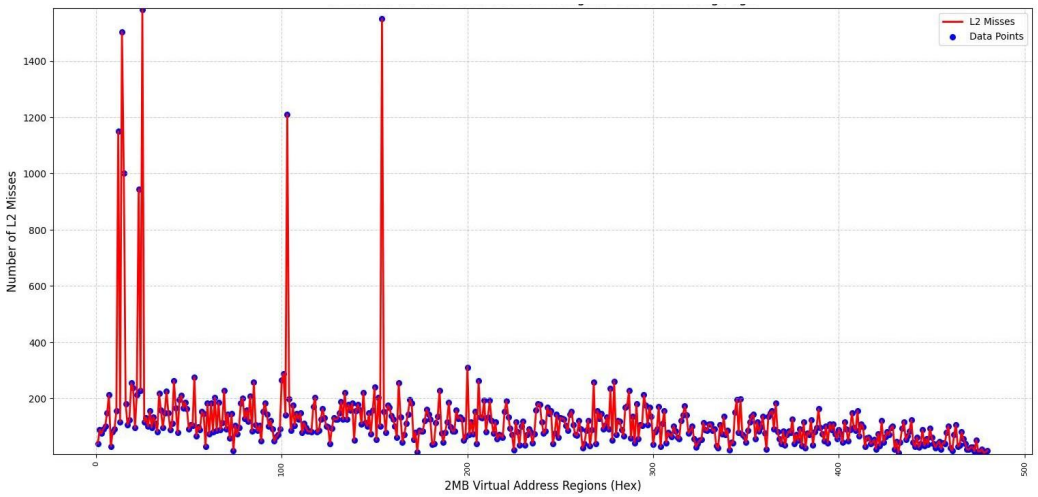


**Fig.1 Shows the distribution of TLB misses across different 2MB regions Before Allocating Large pages.**

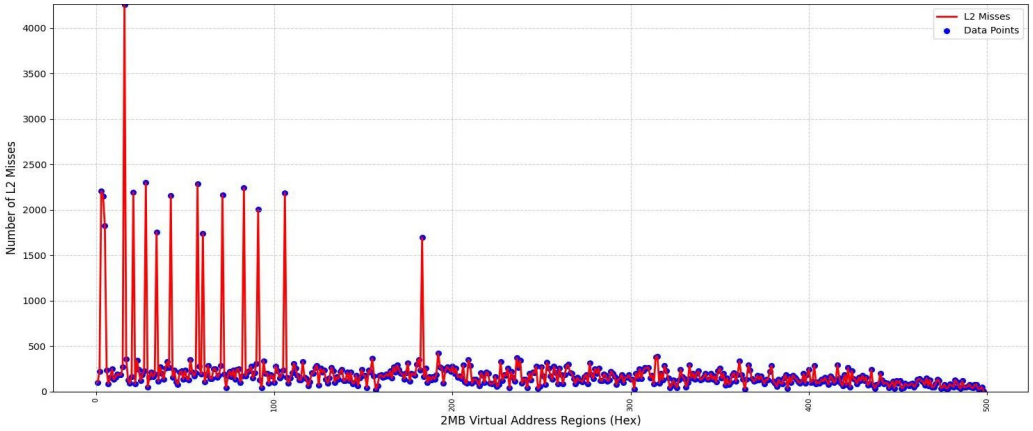


**Fig.2 Shows the distribution of TLB misses across different 2MB regions After Allocating Large Pages.**

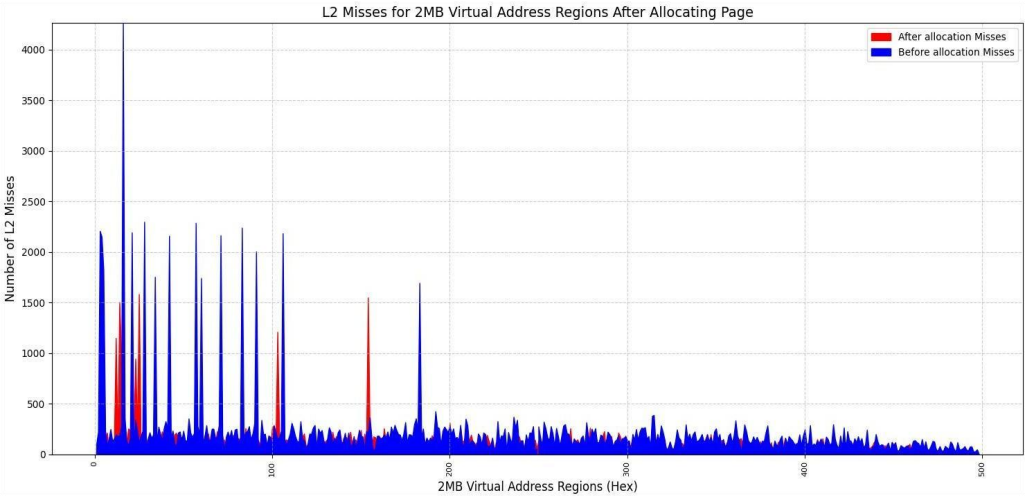


**Fig.3 combined graph for comparision.**

The Below graph shows the performance metrics on y-axis and number of counts on X-axis to compare the performance improvement.

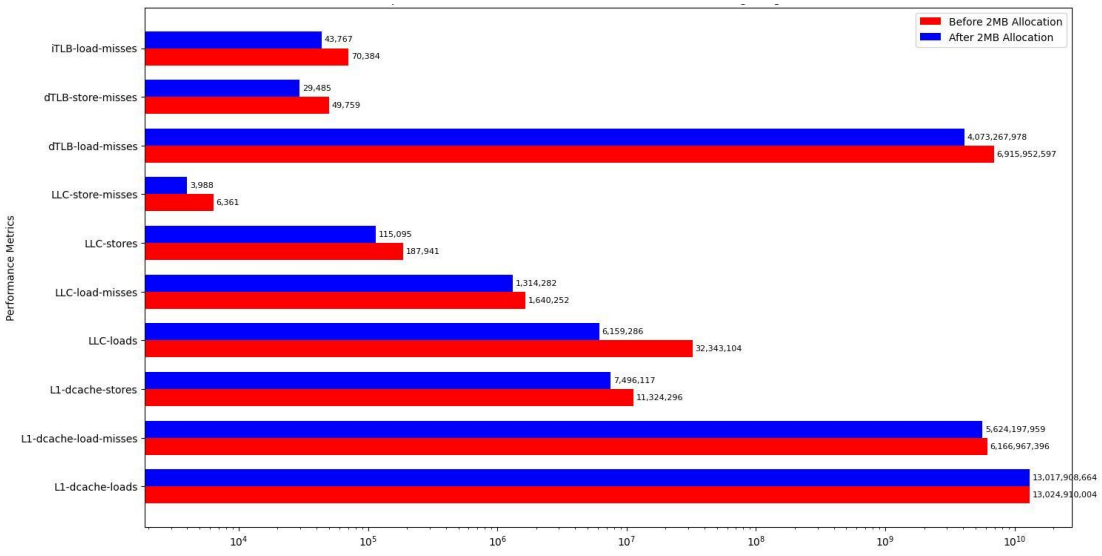


**Fig.3 comparison of performance metrics Before and after 2MB Huge page allocation**

## Performance Analysis :

### Speedup Calculation

The speedup was calculated using the execution times before and after using large pages. The formula used is:

$$\text{Speedup} = \frac{\text{Execution Time Before deploying large pages}}{\text{Execution Time After deploying large pages}}$$

Given:

- Execution time before allocation: **30.608161194** seconds
- Execution time after allocation: **18.274826241** seconds

Now, apply the formula:

$$\text{Speedup} = \frac{30.608161194}{18.274826241} \approx 1.675$$

This means the speedup is approximately **1.675** times, indicating a performance improvement by reducing TLB (Translation Lookaside Buffer) misses through the use of large pages.

## Conclusion:

In conclusion, the allocation of large pages significantly reduced the number of TLB misses, leading to a noticeable performance improvement. The optimization strategy effectively targeted regions with high TLB misses,demonstrating the benefit of judicious large page allocation. Challenges included selecting the optimal addressregions and ensuring correct integration with the modified program.

**CPU Specifications:**

| Arch | x86_64 |
|---|---|
| Model | Intel(R) core(TM) i5-11500 |
| CPU Mhz | 4600 |
| L1d Cache | 288KB |
| L1i cache | 192KB |
| LLC | 12MB |
| Address size | 39 bits physical, 48 bits virtual |

**Table.3 CPU specifications**

## References:


[1] https://man7.org/linux/man-pages/man1/perf.1.html

[2]https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/monitoring_and_managing_system_status_and_performance/getting started-with perf_monitoring-and-managing-system-status-and-performance#introduction-to perf_getting-started-with-perf

[3]https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/monitoring_and_managing_system_status_and_performance/recording-and-analyzing performance-profiles-with-perf_monitoring-and-managing-system-status-and performance

[4] https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/monitoring_and_managing_system_status_and_performance/monitoring-application performance-with-perf_monitoring-and-managing-system-status-and performance#monitoring-application-performance-with-perf_monitoring-and managing-system-status-and-performance

[5] 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

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