EEL5764: Computer Architecture

Performance Analysis of x86 Multi-Core Architecture

Teammates: Harshit Raj (23414989)
Nitesh Bakhati (69913637)
Dev Hitesh Parmar (87730832)
Kanak Sharma (88089607)

Goal:

To analyze the performance of the x86 multi-core architecture in gem5 by scaling the number of cores, cache associativity, and cache size.

Workload:

We are utilizing the Parallel Merge Sort algorithm, implemented in C++ and utilizing OpenMP.

Methodology:

- Workload Development
- Simulation Scripts Development
- Data Extraction and Visualization
- Result Analysis

Workload Development:

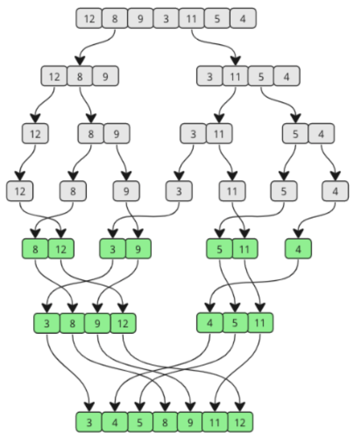
To evaluate the performance of an x86 multicore architecture, we needed a realistic, parallel, and compute-intensive workload that stresses both the CPU and the memory hierarchy. For this, we designed and implemented a **Parallel Merge Sort** algorithm using **C++ with OpenMP.** We use **OpenMP** (Open Multi-Processing), which is a standard API for parallel programming on shared-memory systems, to implement parallelism. It provides easy constructs that enable us to turn the recursive merge sort into a parallel merge sort algorithm.

<u>Merge Sort</u>

We chose **Merge Sort** because it is naturally heavy and scalable. It performs many recursive function calls. It uses large arrays during execution, which results in frequent memory access and high L1/L2 cache activity. It is a classic divide-and-conquer sorting algorithm that works by repeatedly splitting the array into smaller parts, sorting them, and then merging the results.

Dividing - The original array is repeatedly divided into two halves, and this process continues until each sub-array contains only one element.

Conquering - Once the array is split into individual elements, the algorithm begins sorting and merging them back. Adjacent elements are compared and arranged in sorted order.



Implementing Parallel Merge Sort

To implement the parallel merge sort, we have three major functions:

<u>Merge Function</u>

This function **merges** two sorted subarrays. It uses three pointers for the left and right halves, and a temporary array to traverse the data. It compares elements of two subarrays and places the smaller value into the buffer. The sorted merged array is copied back to the original array.

<u>serialMergeSort() Function</u>

This function runs entirely on a single thread with no OpenMP and performs standard single-threaded sorting. We use serial merge sort below serial threshold because creating OpenMP tasks costs CPU Cycles. For small data chunks, parallel execution is not efficient. If we tried to use OpenMP tasks all the way down to a single integer, the simulator would spend more time managing the tasks than sorting the numbers.

<u>parallerMergeSort() Function</u>

This function is the parallel, task-based version of the merge sort implementation using OpenMP. It enables the algorithm to run on multiple cores simultaneously. First, we check the **Size**. If the array chunk is small, we exit parallel mode and call the serial function. Second, we check the **Depth**. If we have already split the work enough times to utilize all our cores, there is no benefit to splitting further. We switch to serial execution to save the CPU from managing unnecessary tasks. In this function, we calculate the midpoint and spawn two parallel OpenMP tasks, one for the left half and one for the right. We wait for all the tasks to finish and then call the merge function to prevent race conditions.

<u>Main Function</u>

This function sets up the input data, allocates and initializes a random array of size N with seed 0 to ensure that the array contains the same sequence of random numbers for every simulation run. We enter the parallel region with #pragma omp parallel, which wakes up all the threads available to the processor. However, we immediately use #pragma omp single since we want one manager thread to start the process and then delegate sub-tasks to the other threads as the recursion deepens.

<u>Compiling Workloads</u>

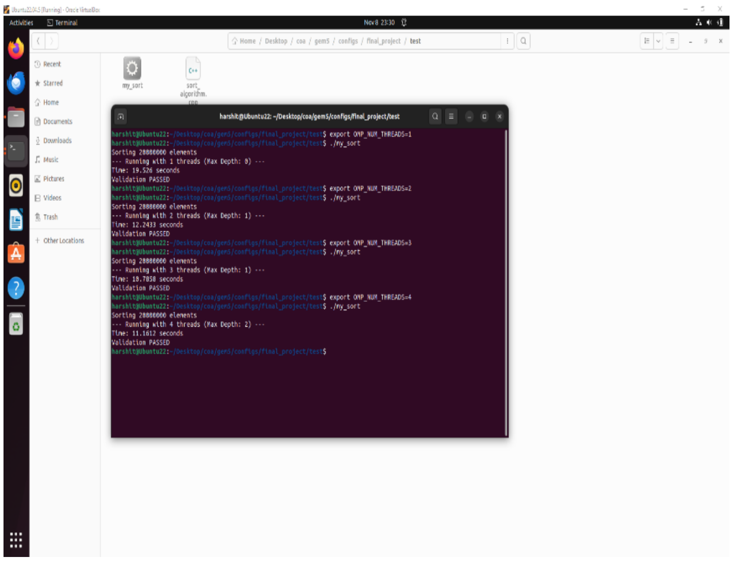
To run our code on the simulation, we compile it using the following command.

g++ -o sort_algorithm_binary -fopenmp -static sort_algorithm.cpp

Where, **g++** is the GNU C++ Compiler, **-o sort_algorithm_binary** is the name of the output executable file, **-fopenmp** enables the OpenMP runtime library, **-static** flag forces static linking of all the libraries, and **sort_algorithm.cpp** is the source code that implements merge sort in C++ with OpenMP

<u>Testing Workload</u>

Before running our simulations on Gem5, we check it on our native machine. We manually set the environmental variable by changing the number of threads to 1, 2, 3, and 4 to test the workload. The execution time decreases as we increase the number of threads, which confirms that our OpenMP directives were working. We also confirm that our parallel merge sort logic is working correctly by validating that the resulting array is sorted.



```
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ export OMP_NUM_THREADS=1
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ ./my_sort
Sorting 20000000 elements
--- Running with 1 threads (Max Depth: 0) ---
Time: 19.526 seconds
Validation PASSED
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ export OMP_NUM_THREADS=2
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ ./my_sort
Sorting 20000000 elements
--- Running with 2 threads (Max Depth: 1) ---
Time: 12.2433 seconds
Validation PASSED
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ export OMP_NUM_THREADS=3
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ ./my_sort
Sorting 20000000 elements
--- Running with 3 threads (Max Depth: 1) ---
Time: 10.7858 seconds
Validation PASSED
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ export OMP_NUM_THREADS=4
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$ ./my_sort
Sorting 20000000 elements
--- Running with 4 threads (Max Depth: 2) ---
Time: 11.1612 seconds
Validation PASSED
harshit@Ubuntu22:~/Desktop/coa/gem5/configs/final_project/test$
```

Simulation Scripts:

To evaluate the performance of the x86 multi-core architecture in gem5, we designed three simulation scripts that run the same workload under different hardware configurations.

<u>Core Scaling</u>

The Core scaling script evaluates system performance as CPU core count increases. To automate this, we wrote a bash script that functions as a loop. It iterates through our target core counts, which are 2, 4, 8, and 16. In each iteration, it constructs a new gem5 command and runs the same workload. It safeguards the data by moving all output files into a uniquely named folder for each core count. Since gem5 overwrites its output file every time.

Cache Associativity Scaling

Associativity defines how flexible the cache is. By increasing associativity to 2, 4, or 8, we allow the cache to store those fighting variables in different ways. For this script, we froze the Core Count at 8 and loop through associativity values of 1, 2, 4, and 8 for both the L1 and L2 cache.

Cache Size Scaling

Cache size scaling reveals how memory capacity impacts miss rate and execution latency. To capture the interaction between L1 and L2, we froze the CPU count and associativity values and used a Nested Loop. The outer loop selects the L1D size (16kB, 32kB, 64kB, and 128kB), while the inner loop selects the L2 size (128kB, 256kB, 512kB, and 1MB).

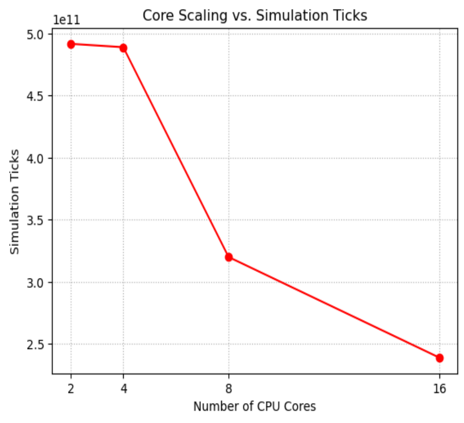
Data Extraction and Visualization:

To systematically extract performance metrics from gem5 simulation logs and generate visual plots for analysis, we created a Python script that reads gem5 stats files from all the output directories and extracts selected metrics using regex. We then organize the results and generate plots and heatmaps with the help of the matplotlib library. The script generates the plot and saves the output plot to the **output_plots** directory.

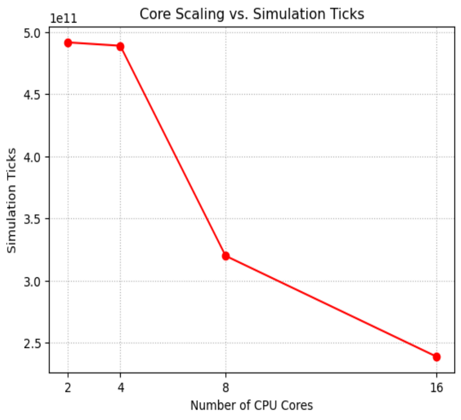
Result Analysis:

Core Scaling - Ticks Measurement

The graph shows how simulation ticks decrease as the number of CPU cores increases from 2 to 16. Ticks remain consistently high around 5.0. This indicates that moving from 2 to 4 cores provides a **negligible** benefit because the overhead of managing more threads nearly cancels out the parallel speedup. There is a **sharp drop** from 4 to 8 cores. This is the region where the algorithm's parallelism effectively utilizes the hardware. Ticks **drop** further from 8 to 16 cores, but comparatively less than 4 to 8 cores, indicating diminishing returns caused by memory contention and limited parallel regions. For this workload, an **8-core** CPU is the optimal choice.



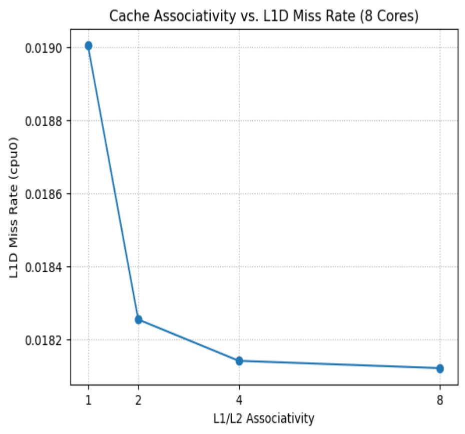
Core Scaling - Execution Time Analysis



The graph shows how the execution time reduces as the number of CPU cores increases. We see the exact same trend when we look at the wall execution time in seconds. The sort algorithm took about 0.49 seconds on 2 and 4 cores. It dropped to 0.32 seconds on 8 cores. Then, it reached 0.24 seconds on 16 cores. While 16 cores are technically the fastest, 8 cores are the most efficient configuration. Going from 8 to 16 cores doubled our hardware cost and power consumption but only gave us a roughly ~25% speedup.

Cache Associativity - L1D Cache Miss Rate

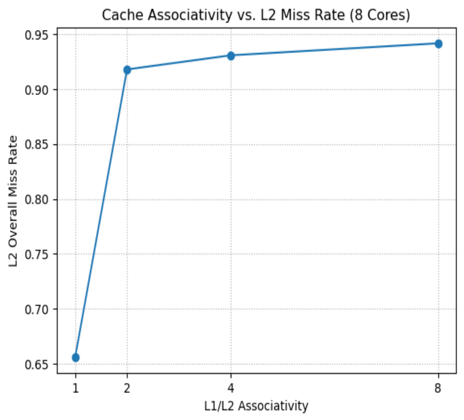
The graph shows how the L1 data cache miss rate **decreases** as the associativity increases.



The graph starts with the **highest** miss rate at 1-way (Direct Mapped), indicating that the cache suffers significantly from conflict. There is a steep drop in the miss rate when moving from 1-way to 2-way associativity. This suggests that simply providing a second storage option for each index in cache resolves most of the mapping conflicts. However, from 2-way to 4-way and 8-way, the line effectively flatlines. This indicates that the remaining misses are compulsory, meaning we must fetch the data from RAM at least once, or capacity misses, where the cache is simply full. For this workload, a **2-way associativity** cache is the optimal choice.

Cache Associativity - L2 Cache Miss Rate

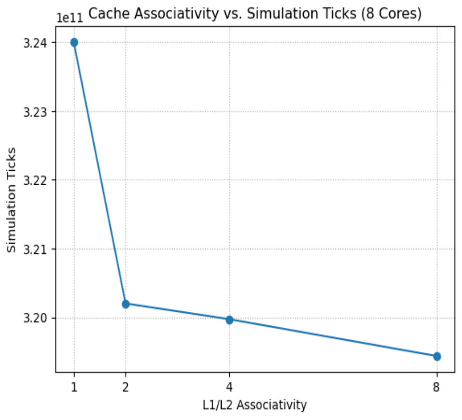
The graph shows that the overall L2 global miss rate **increases** as associativity increases.



Unlike the L1 cache, the L2 miss rate drastically increases as associativity improves, rising from 1-way to 2-way and drifting higher at 4-way and 8-way. This phenomenon is known as the Filtering Effect. This occurs because the L1 cache becomes more efficient at higher associativity and captures all the easy, predictable memory accesses, leaving mostly difficult requests that are more likely to miss. When we upgraded the L1 cache to 2-way, it started catching all those easy hits. Suddenly, the L2 cache stopped seeing the easy traffic. The only requests reaching it were the random, unpredictable ones that the L1 cache couldn't handle. Because L2 Cache is now only seeing the hardest problems, its miss rate goes up.

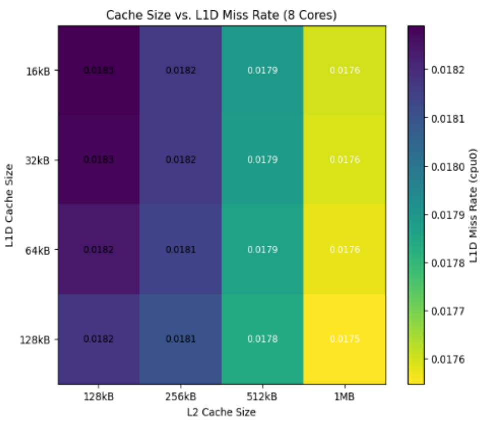
Cache Associativity - Ticks Measurement

The graph shows how the total simulation ticks reduce as the associativity increases. The total simulation ticks **drop** significantly from 1-way to 2-way. This confirms that resolving the conflict misses at L1 directly translates to faster execution. Beyond 2-way associativity, the curve **almost flattens**. The improvement from 2-way to 4-way and 8-way is marginal. This confirms the Filtering Effect. Even though the L2 cache looked like it was struggling, the overall system became faster because the L1 cache was handling more of the load. Most of the **performance** benefit is achieved simply by moving away from a direct-mapped cache. Higher associativity adds extra **hardware complexity** with **minimal** impact on actual runtime.



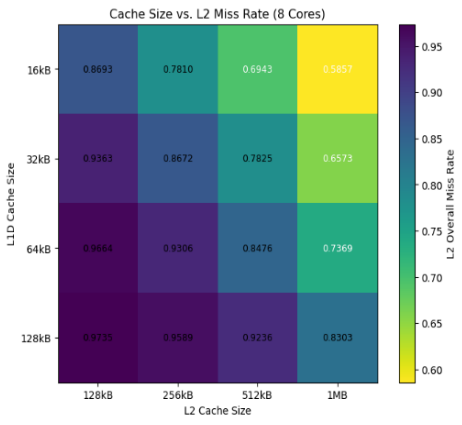
Cache Size – L1D Cache Miss Rate

The heatmap shows how the L1D miss rate changes across different L1 and L2 cache size combinations. The miss rate decreases slightly from **~0.0183** to **~0.0175** as both cache sizes increase. Increasing L1D size reduces miss rate as larger L2 sizes reduce backpressure on L1D. Larger caches reduce some capacity misses, but the misses occurring here are primarily Compulsory (first-access).



Cache Size – L2 Cache Miss Rate

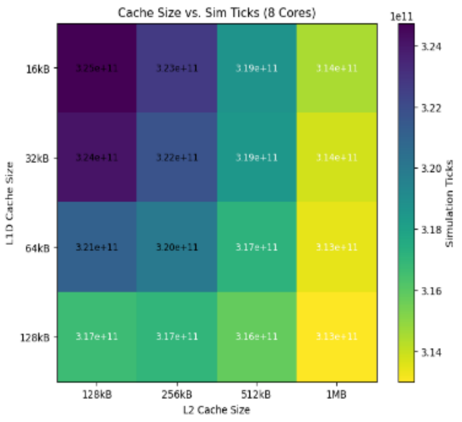
The heatmap shows how the global L2 miss rate varies with different L1 and L2 cache size combinations. Increasing L2 size from 128kB to 1MB significantly **reduces** the miss rate, as it dropped from ~0.87 down to ~0.58 in the top row. At 128kB (the left column), the cache is simply too small, causing constant evictions. By the time we get to 1MB (the right column), we have solved those capacity misses, dropping the miss rate to around ~58%. However, when we look vertically. As we increase the L1 size (moving down), the L2 miss



rate increases. This is that Filtering Effect again. A larger L1 captures more hits, leaving only the most difficult, random patterns for the L2 to handle, which increases the miss percentage.

<u>**Cache Size – Ticks Measurement**</u>

The heatmap shows how the total simulation ticks change with different L1 and L2 cache size configurations. The heatmap displays a clear diagonal gradient from dark purple (top-left) to bright yellow (bottom-right). This indicates that increasing both L1 and L2 cache sizes contributes to performance. While both axes affect the color, the shift is stronger when moving horizontally (increasing L2) than vertically (increasing L1). The total speedup between the worst case and the best case is approximately 3-4%.



Conclusion:

- **Parallelism is the primary performance driver**. Scaling from **2 to 8 Cores** yielded the most significant speedup, showing that 8 cores use parallelism effectively. Beyond 8 cores, memory bandwidth saturation leads to diminishing returns limited by sequential work, overhead, and memory contention.
- Moving from Direct Mapped to **2-way Associativity** eliminated the majority of L1 conflict misses. Higher associativity (4-way/8-way) offered negligible additional benefits, making 2-way the most efficient design choice.
  - *L2 miss rate increases with associativity because fewer, harder accesses reach L2, even though overall memory traffic and performance improve.*
- Performance is highly sensitive to **L2 Cache Size**, which buffers main memory access. L1 size had minimal impact, proving the workload's hot set fits within 16kB.
  - *Growing L1D from 16 kB to 128 kB slightly lowers miss rates and ticks, indicating only modest Locality benefits from a larger L1.*
  - *Increasing L2 from 128 kB to 1 MB significantly reduces L2 miss rate and modestly reduces ticks.*
  - *Larger L1D and L2 together improve performance by only ~3–4%, far less than the gains from adding cores.*

Final Verdict - The **parallel merge sort workload** is primarily compute-bound and parallelism-sensitive, with core scaling providing the biggest performance gains (~50%), while cache associativity and cache sizes act as secondary fine-tuning optimizations with very minimal performance gain (~4%).