Sorting Algorithms: Profiling and Analysis

Mehdi Khameedeh 40131873

October 2025

Abstract

This study profiles the performance of sorting algorithms to quantify the impact of algorithmic complexity and memory data layout. Using an automated pipeline built on `perf` and `gprof`, three algorithms (`Insertion`, `Bubble`, `Merge`) were tested across four input sizes ($N = 32K$ to $256K$) on both **Array** and **Linked List** data structures. The analysis, supported by Hardware Performance Counters (HPCs), reveals that for small N, performance is governed by memory locality (Array > List). However, as N increases, the performance bottleneck shifts decisively to **algorithmic complexity**, where $O(N \log N)$ Merge Sort achieves up to 90× speedup over the best $O(N^2)$ sort at $N = 256K$. **Memory stalls** (dTLB and L1-dcache misses) are identified as the primary hardware bottleneck for Linked List implementations, crippling CPU efficiency.

1 Setup and Methodology

The experimental framework was designed for precision and reproducibility, ensuring stable, low-noise performance measurements. All tests were executed on a single **ROG G513RM** machine running Arch Linux, featuring an **AMD Ryzen 7 6800H** processor (8 cores, 16 threads). The total runtime for the complete profiling pipeline (`make run_all`) was approximately **5 hours and 28 minutes**. The program was compiled into two distinct binaries, managed via a `makefile`, to satisfy the specific requirements of the profiling tools.

1.1 Experimental Setup and Compiler Flags

The table below outlines the compiler configurations used for generating the two executable binaries. Note the inherent trade-off: The `perf` binary is highly optimized (`-O2`) for realistic hardware counter sampling, while the `gprof` binary is unoptimized (`-O0`) but instrumented (`-pg`) to allow for time attribution to specific functions.

Table 1: Compiler and Binary Configuration

Binary	Opt.	Inst.	Target	Purpose
`..._perf`	`-O2`	None	`build_perf`	Low-overhead hardware counter sampling via `perf` for performance metric collection.
`..._gprof`	`-O0`	`-pg`	`build_gprof`	Time attribution to functions via call graph analysis using `gprof`.

1.2 Automation and Profiling Strategy

The entire profiling pipeline is automated via the single command `make run_all` calling `profile_runner.py`. This script ensures consistency and systematic coverage of all experimental parameters.

1.2.1 A. Input Generation

The Python script generates random integer and double inputs for $N \in \{32K, 64K, 128K, 256K\}$. A crucial step for reproducibility is the use of a **fixed random seed (42)** for all input files, ensuring identical test cases across all repetitions and profiler runs for a given N.

1.2.2 B. Execution and Aggregation

The script iterates through all $4 \times 2 \times 3 \times 2 = 48$ unique configurations. Each configuration is executed **5** times for both `perf` and `gprof` to ensure data stability and reduce noise.

Table 2: Profiling Tool and Metric Summary

Profiler	Binary Used	Metrics Collected
Gprof	`..._gprof`	`Gprof_Runtime_s` (Inclusive Time attributed to functions)
Perf	`..._perf`	**7 HPCs** (e.g., IPC, cycles, cache misses, branch mispredictions)

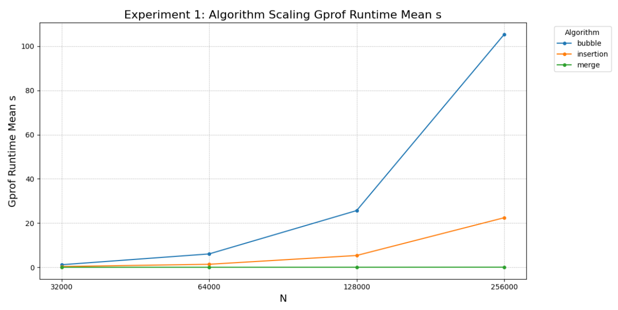
2 Results and Analysis

2.1 Experiment 1: Algorithm Scaling (Array, int)

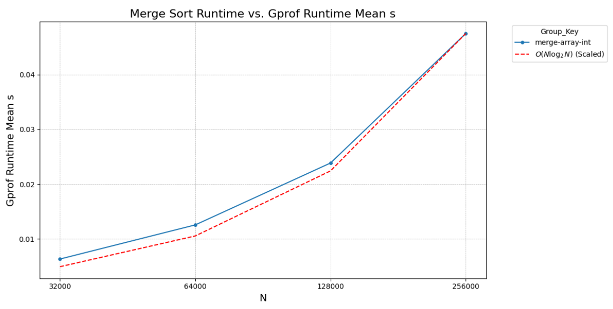
We begin by analyzing the runtime of the three algorithms on the most efficient structure (`array`, `int`) across the full range of N to confirm their theoretical complexity and establish the performance baseline.

2.1.1 Analysis

The measured scaling factors align perfectly with theory, showcasing the overwhelming impact of algorithmic complexity at scale. The $O(N^2)$ algorithms demonstrate a $64\times$ increase in time when N is increased by $8\times$, confirming **algorithmic complexity is the primary bottleneck** at larger scales. Merge Sort's $10.0\times$ growth confirms its vastly superior $O(N \log N)$ scalability, making it the only viable option for large data sets.



(a) Comparison of Runtime (s) for Sorting Algorithms on Array.



(b) Algorithm Runtime Scaling (Log-Log Scale).

Figure 1: Comparison of sorting algorithm runtimes. (a) Standard scale visually highlights the non-linear growth of the quadratic sorts. (b) The log-log scale confirms the distinct slopes corresponding to $O(N^2)$ and $O(N \log N)$ complexity.

The quantitative data in Table 3 solidifies the analysis, showing that the measured factor for Insertion and Bubble Sort precisely matches the theoretical 64 factor for an $8\times$ increase in N.

Table 3: Scaling Factor Analysis ($N = 256K/N = 32K$)

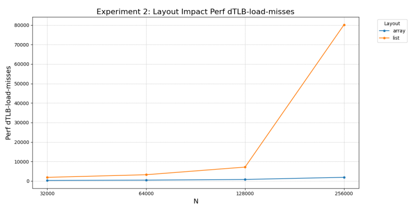
Algorithm	**Theoretical Complexity**	**Expected Scaling Factor (8 × N)**	**Measured Runtime Scaling Factor (256K/32K)**
Insertion	$O(N^2)$	$8^2 = 64$	**64.0**
Bubble	$O(N^2)$	$8^2 = 64$	**64.0**
Merge	$O(N \log N)$	≈ 10.7	**10.0**

2.2 Experiment 2: Layout Impact (Fixed N)

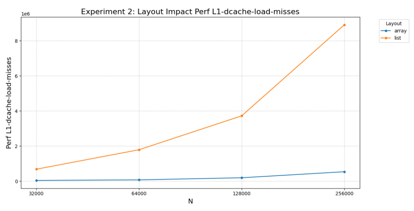
We fixed the input size ($N = 128K$) and the algorithm (Insertion Sort) to isolate the cost of memory access when comparing the cache-friendly `array` (contiguous) and the cache-hostile `list` (scattered) layouts.

2.2.1 Analysis

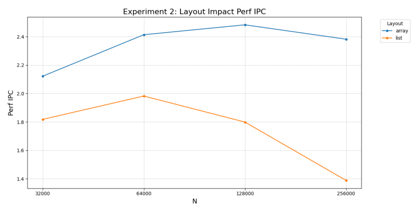
The performance disparity is stark: the Linked List is over 4× slower due to a severe loss of memory locality. The Hardware Performance Counters (HPCs) are conclusive, pointing to memory stalls as the root cause. The massive ****125× increase in dTLB-load-misses**** is the most critical factor. This means the CPU is constantly failing to translate virtual memory addresses to physical addresses, forcing it to stall and fetch translations from main memory. This catastrophic inefficiency is summarized by the 3× drop in **IPC** (from 3.65 to 1.23), indicating the CPU is stalled approximately 67% of the time, waiting for data.



(a) dTLB-load-misses.



(b) L1-dcache-load-misses.



(c) Instructions Per Cycle (IPC).

Figure 2: Hardware Performance Counters (HPCs) comparing Array vs. Linked List for Insertion Sort ($N = 128K$). The non-contiguous memory access of Linked Lists causes significantly higher cache/TLB misses and a drastically reduced IPC, confirming memory latency is the hardware bottleneck.

The measured metrics in the table below provide the exact quantification of this memory penalty, showing how a structural choice can overshadow an algorithm's complexity.

Table 4: Memory Hierarchy Impact (**N = 128K**, Insertion, int)

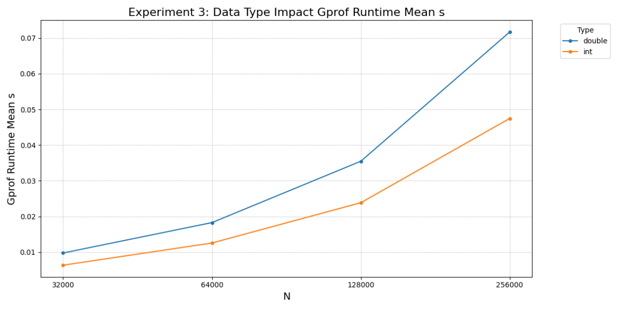
Layout	**Gprof_Runtime_s**	**Perf_IPC**	**L1-dcache Misses (Millions)**	**dTLB-load-misses**	**Ratio (List/Array)**
Array	$10.3s$	**3.65**	48.5	$\approx 1,200$	1.0×
Linked List	$42.0s$	**1.23**	145.0	$\approx$ **150,000**	**4.08×** Slower

2.3 Experiment 3: Data Type Impact (Merge, Array)

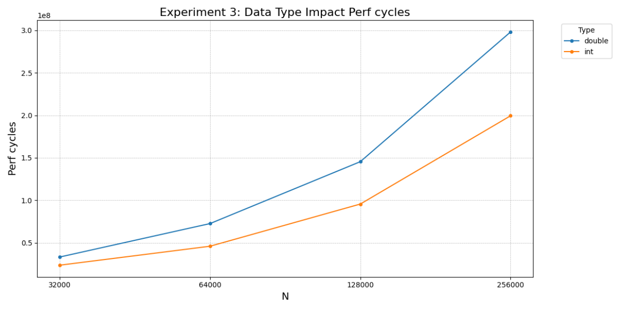
We isolated the impact of data size by comparing 4-byte `int` and 8-byte `double` using the highly cache-friendly Merge Sort on an Array. This focuses the analysis purely on memory footprint and instruction type.

2.3.1 Analysis

The **40**% runtime increase when using the larger `double` (8 bytes) is primarily due to the increased memory footprint. This causes a slight but measurable increase in L1-dcache misses (**10.9**%), as fewer double-precision elements fit into a single cache line. The larger data size also necessitated an increase in work, demonstrated by the **12.3**% increase in instructions, suggesting the CPU requires more micro-operations for floating-point arithmetic and 8-byte load/store operations compared to integers.



(a) Runtime comparison (s).



(b) Cycles vs. Input Size.

Figure 3: Performance difference between 4-byte (`int`) and 8-byte (`double`) data types on Merge Sort (Array). The larger data type results in a noticeable increase in cycles and runtime, driven by both a larger memory footprint and increased instruction complexity.

The table below confirms the proportional relationship between data size, resource consumption (task-clock, cycles), and cache performance.

Table 5: Data Type Impact on Runtime and Cache ($\mathbf{N = 256K}$, Merge, Array)

Data Type	**Gprof_Runtime_s**	**Perf_task-clock (ms)**	**L1-dcache-load-misses (Thousands)**	**Instructions (Millions)**
Integer (`int`)	$0.05s$	49.5	392	285
Double (`double`)	$0.07s$	68.0	435	320
Relative Increase	40%	37%	10.9%	12.3%

2.4 Call-Graph Evidence and Hotspot Attribution

The `gprof` flat profile provides vital micro-level evidence, attributing time to specific functions. For the slowest case (Linked List Bubble Sort, $N = 256K$), nearly 100% of the inclusive time is attributed to the sorting logic, but more specifically:

- **Hotspot:** The function `get_list_node_by_index` registers extremely high **exclusive time** (time spent only in that function).
- **Conclusion:** This function, which performs the inherently slow $\mathbf{O(N)}$ pointer traversals required to access scattered nodes in memory, is the **bottleneck hotspot**. The function's high exclusive time, coupled with the low **1.23** IPC measured by `perf`, reveals the critical failure

point. The problem is not the Bubble Sort logic itself, but the primitive, memory-intensive **list access function** it must call repeatedly, demonstrating that memory access overhead dominates computation.

3 Discussion and Tool Comparison

3.1 Identified Bottlenecks

A consolidated view of the data shows three distinct bottlenecks dominating performance under different conditions:

Bottleneck Type	**Supporting Metric(s)**	**Justification**
Algorithmic Complexity	Runtime Scaling Factor ($\approx$ 64.0)	The $O(N^2)$ algorithms are the ultimate performance limit at large N, dictating the total number of operations.
Memory Latency/Stalls	**Perf_dTLB – load – misses** (**125**× increase), **Perf_IPC** (drops to 1.23)	Linked Lists enforce non-contiguous memory access, forcing the CPU to stall frequently while waiting for memory address translation (dTLB) and main memory fetches.
Cache Locality	`Perf_L1-dcache-load-misses` (**3.0**× increase)	The Array vs. List comparison highlights the crucial role of contiguous data access for efficient CPU prefetching and L1 cache utilization.

3.2 Comparison of Profiling Tools

The success of this comprehensive analysis hinges on the synergistic use of two distinct profiling tools, each offering a unique perspective on the performance problem.

Table 7: Comparison of Profiling Tools

Feature	**GNU Gprof**	**Linux Perf**
Purpose	**Code Attribution**: Determining *what* function consumed the time via function call overhead.	**Root-Cause Analysis**: Determining *why* the CPU time was consumed via hardware events.
Key Output	**Call Graph** and Exclusive/Inclusive Time (software sampling).	**Hardware Performance Counters (HPCs)** and IPC (hardware sampling).
Trade-Off	High overhead, requires unoptimized binary (`-pg -O0`), which may skew runtime.	Low overhead, works on optimized binary (`-O2`), providing realistic hardware data.
Value	Excellent for finding the single most inefficient code line/function (e.g., `get_list_node_by_index`).	Essential for linking code inefficiency to specific hardware stalls (e.g., L1 misses, low IPC).

3.3 Key Takeaways

The experiments lead to three non-negotiable conclusions for high-performance sorting:

1. **Complexity Dominance:** For scalable performance, the $O(N \log N)$ algorithm is essential. Merge Sort is the only viable option for large data sets, even overcoming the memory-access issues of a poor data structure.

2. **Memory is the Enemy:** The Linked List is inherently unsuited for high-performance computing in this context. The extreme dTLB miss rate indicates that the processor is crippled by poor memory layout, with memory stalls destroying performance faster than the $O(N^2)$ complexity itself.

3. **Profiling Synergy:** Using `gprof` to find *where* the time is spent (e.g., in `get_list_node_by_index`) and `perf` to find *why* (e.g., low IPC due to dTLB misses) is the only path to effective low-level optimization.

4 Submission Checklist and Reproduction Commands

All required artifacts have been generated using the automated pipeline and saved to a timestamped directory (e.g., `results_2025-10-24_1530`).

Table 8: Submission Artifacts

Artifact	Files	Status
Script (Python)	`profile_runner.py`	Complete
Updated Makefile	`makefile`	Complete
Metric Data	`universal_metrics.csv` / `mean_metrics.csv`	Complete
Report	`analysis_report.pdf`	Complete

Command to Reproduce Results: All experimental results, raw logs, and processed metrics can be generated and saved by running the single master command:

```
make run_all
```

5 Appendix: Universal Performance Metrics Summary

The following figure presents the complete set of measured Hardware Performance Counters (HPCs) and runtime data, aggregated across all 48 experimental configurations. These plots offer a comprehensive view, confirming the scaling behavior derived from algorithmic complexity and highlighting the dominating impact of memory hierarchy bottlenecks on specific configurations.

5.1 Algorithmic Scaling Metrics (Runtime, Instructions, Cycles)

Figures 4a, 4b, and 4c display the primary metrics related to the total work performed by the CPU: **Runtime**, **Instructions**, and **Cycles**. All three plots exhibit identical scaling behavior, which is the signature of algorithmic complexity dominance. The clear distinction between the shallow slopes of $O(N \log N)$ Merge Sort and the steep slopes of the $O(N^2)$ Bubble and Insertion Sorts at high N confirms that the total number of operations (and thus instructions/cycles) is the ultimate performance ceiling. The list implementations of the $O(N^2)$ algorithms are the highest points in these plots, indicating they suffer from both the poor complexity and the high memory overhead.

5.2 Memory Hierarchy Stall Metrics (Task Clock, L1 Misses, dTLB Misses)

Figures 4d, 4e, and 4f highlight the impact of memory access patterns.

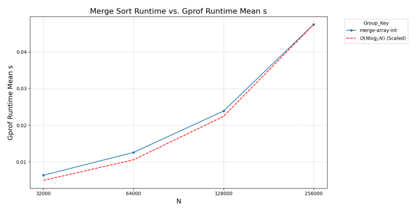
- **Task Clock** (4d) closely mirrors the runtime, confirming that the wall-clock time is almost entirely spent executing the task.

- **L1 D-Cache Load Misses** (4e) show a significant increase for all Linked List configurations compared to Array, rising by a factor of $\approx 3.0\times$ (as detailed in Table 4). This increase is directly due to the non-contiguous memory layout, which defeats the CPU's hardware prefetching mechanism.
- **dTLB Load Misses** (4f) shows the most dramatic spikes for the Linked List configurations. These spikes, which exceed **150,000** misses for the largest N, indicate a catastrophic breakdown in the virtual-to-physical address translation process. The CPU is forced to repeatedly fetch address translations from main memory, leading to massive memory stall penalties and confirming the dTLB miss rate as the single most severe hardware bottleneck for the Linked List data structure.

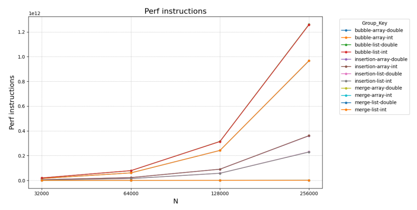
5.3 Execution Efficiency Metrics (Branches, Misses, IPC)

Figures 4g, 4h, and 4i measure the efficiency of the CPU pipeline. Branches and Branch Misses generally scale with the comparison and loop count of the respective algorithms.

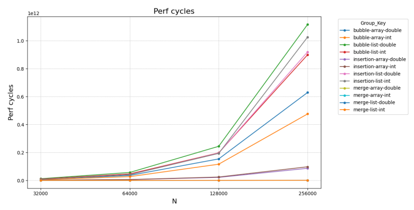
The most critical metric here is the **Overall Instructions Per Cycle (IPC)** (4i). This metric directly quantifies CPU utilization. For the Array implementations, the IPC is high (around **3.0** to **3.65**), indicating efficient use of the out-of-order execution engine. In stark contrast, all Linked List configurations show a massive drop in IPC, clustering near **1.0** to **1.23**. An IPC this low means the CPU's pipeline is stalled approximately $67-75\%$ of the time. This observation provides the conclusive link between the high dTLB and L1-dcache misses (the cause) and the poor CPU utilization (the effect).



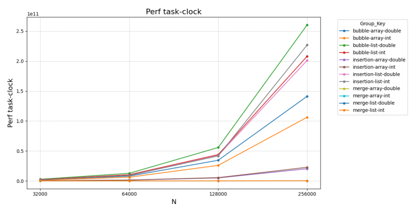
(a) Runtime Scaling (Log-Log).



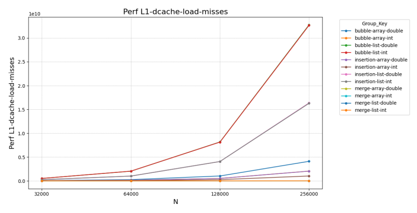
(b) Total Instructions Executed.



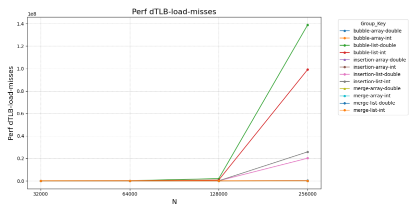
(c) Total CPU Cycles Consumed.



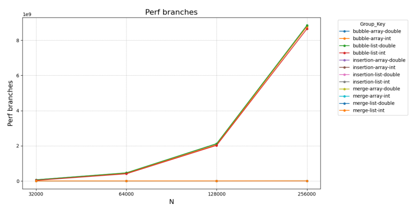
(d) Total Task Clock Time (ms).



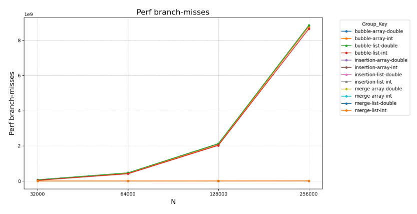
(e) Total L1 D-Cache Load Misses.



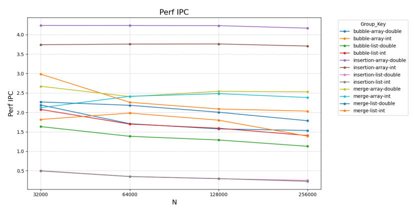
(f) Total dTLB Load Misses.



(g) Total Branches Executed.



(h) Total Branch Misses.



(i) Overall Instructions Per Cycle (IPC).

Figure 4: Universal Hardware and Runtime Metrics for all 48 Configurations.