Analysis: Memory Performance in Java

A Comparative Study of Volatile Variables, Array Access, and Data Structure Search Performance

Prepared for: CS250 Computer Systems Foundations

Amber Ferrell

Colorado State University
Department of Computer Science

amber.ferrell@colostate.edu

Analysis: Memory Performance in Java

This report will review the results of several experiments conducted in Java. Aspects of this experiment will include memory access patterns, caching effects, and data structure efficiency in Java. For each experiment, there are two controlled parameters. The size of the structure being evaluated is consistently 25,000,000 and the seed value, used for randomization or initialization in the experiments, is forty-two across all experiments. The number of times the experiments are conducted will change for each experiment to find averages and sums. Time is calculated in nanoseconds.

Volatile vs Non-Volatile Performance Report

The first experiment involves observing how the use of the volatile keyword in Java impacts loop performance. An experimental run computes running totals with regular and volatile operations to assess the execution time difference between the two types of operations.

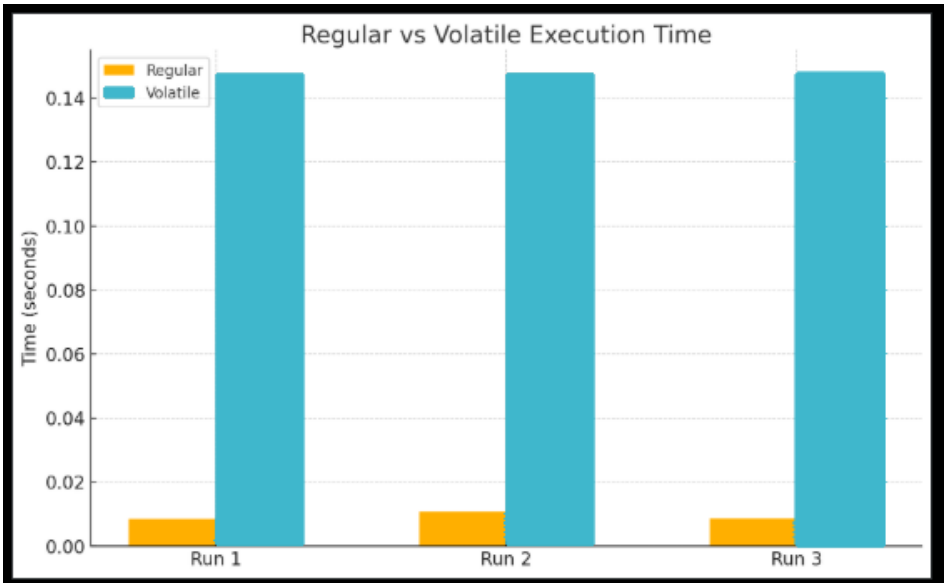
Size	Experiments	seed	Regular Execution time	Volatile Execution Time	Avg Regular Sum	Avg Volatile Sum
25000000	20	42	0.00847	0.14748	-12,500,000.00	-12,500,000.00
25000000	1	42	0.01083	0.14761	-12,500,000.00	-12,500,000.00
25000000	200	42	0.00852	0.14749	-12,500,000.00	-12,500,000.00

Performance Comparison of Regular vs. Volatile Variable Access

This table presents execution time differences between regular and volatile variable access in Java. Regular operations benefit from CPU caching and execute significantly faster, 0.00847–0.01083 seconds, while volatile operations enforce direct memory access, incurring a substantial overhead 0.14748–0.14761 seconds. Despite this difference, both methods yield consistent computational results.

The results clearly show a substantial difference in execution times between the regular and volatile implementations. The execution time for the regular variable ranged from 0.0085 to 0.0108 seconds across different runs. The volatile version of the loop consistently took around 0.1475 seconds, making it approximately seventeen times slower than the non-volatile version. Despite the performance difference, the computed sums remained consistent at -12,500,000, verifying correctness.

The performance penalty is mainly due to the extra memory access cost associated with volatile. Each update operation on a volatile variable completely bypasses the CPU caches and goes directly into main memory, and because of that, all benefits of local caching are lost, and execution slows down. The volatile keyword is indispensable for providing memory visibility in concurrent programming, at the expense, however, of significant performance penalties for single-threaded activities. If high-speed operations are needed, it is best to avoid volatile for heavily updated variables and use other tools for concurrency control for thread-safe operations with better efficiency.



Regular vs Volatile Execution Time
The bar chart compares the execution times of regular and volatile variable access across three runs. The y-axis is time in seconds, while the x-axis labels the runs. The orange bars are the time taken by the volatile method of the experiment while the blue represent non-volatile times. The graph illustrates the significant performance difference between regular execution, which is faster due to potential caching, and volatile execution, which is slower as it bypasses cache to ensure memory visibility.

Memory Access Performance Report

This task analyzes the time needed to access different portions of a large array. It provides a comparison of access times between two different portions of an array filled with randomly selected integers. The first 10% of the array will be accessed then one randomly selected integer

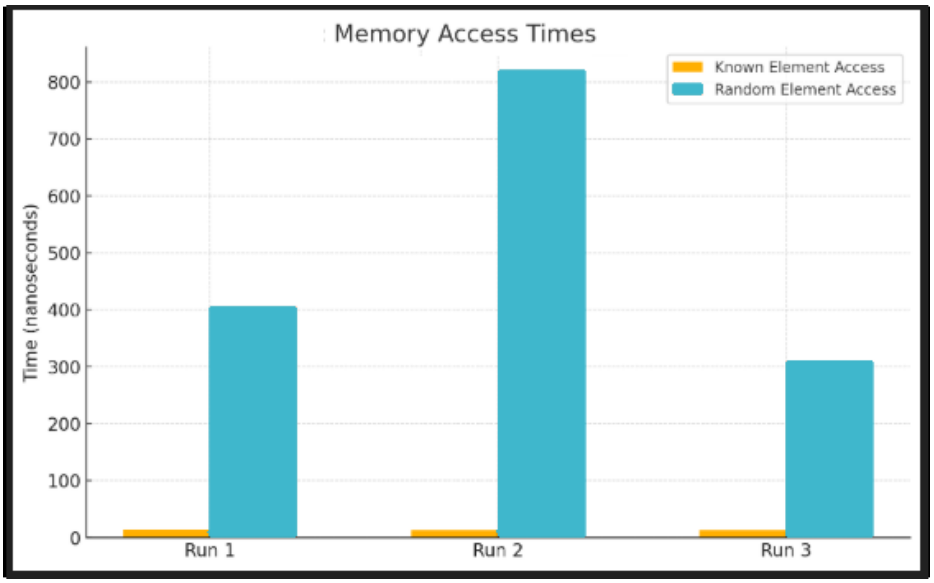
from the last 10 % will be accessed. This experiment helps illustrate the effects of memory locality and caching on performance.

Size	Experiments	seed	Avg Time to Access Known Element (ns)	Avg Time to Access Random Element (ns)	Sum of Accessed Elements
25000000	20	42	13.78	405.60	-622,940,708,490.00
25000000	1	42	13.51	820.00	-623,797,425,765.00
25000000	200	42	13.62	310.02	-623,296,762,828.86

Memory Access Performance
The table presents the result. The Avg Time to Access Known Element (ns) column shows the average time to access elements in the first 10% of the array, with times ranging from 13.51 to 13.78 nanoseconds. The Avg Time to Access Random Element (ns) column displays the average time to access a single random element in the last 10% of the array, with times ranging from 310.02 to 820.00 nanoseconds. The "Sum of Accessed Elements" column shows the cumulative sum of accessed elements, consistently around -623 trillion. The significant performance difference between accessing known and random elements in the array.

The experiment yielded that on average, retrieving elements from the first 10% of the array took between 13 to 14 nanoseconds. Accessing a single randomly selected element from the last 10% provided a range from 310 to 820, varying widely. The sum of accessed elements remained consistent across multiple experimental runs, confirming that access patterns do not affect the correctness of calculations. Memory locality, based on these results, is important in performance optimization.

The first 10% of the array is more likely to be in the CPU cache due to sequential access patterns, making retrieval nearly instantaneous. In contrast, accessing a random element from the last 10% of the array incurs a significant delay, as it requires fetching data from slower memory levels. The experiment confirms that memory locality significantly affects performance. When designing data structures and algorithms, ensuring sequential or predictable access patterns can drastically improve efficiency. Minimizing random memory access patterns, especially in performance-critical applications, should help in avoiding cache misses and unnecessary memory fetch operations.



Memory Access Times

This bar chart presents the memory access times for known and random elements across three experimental runs. The x-axis is each experimental run, while the y-axis shows the time taken in nanoseconds. The orange bars are the time needed to access a known element from the first 10% of the array, which remains consistently low across all runs. The blue bars are the time needed to access a random element from the last 10% of the array, which shows significantly higher and more variable times.

TreeSet vs. LinkedList Search Performance Report

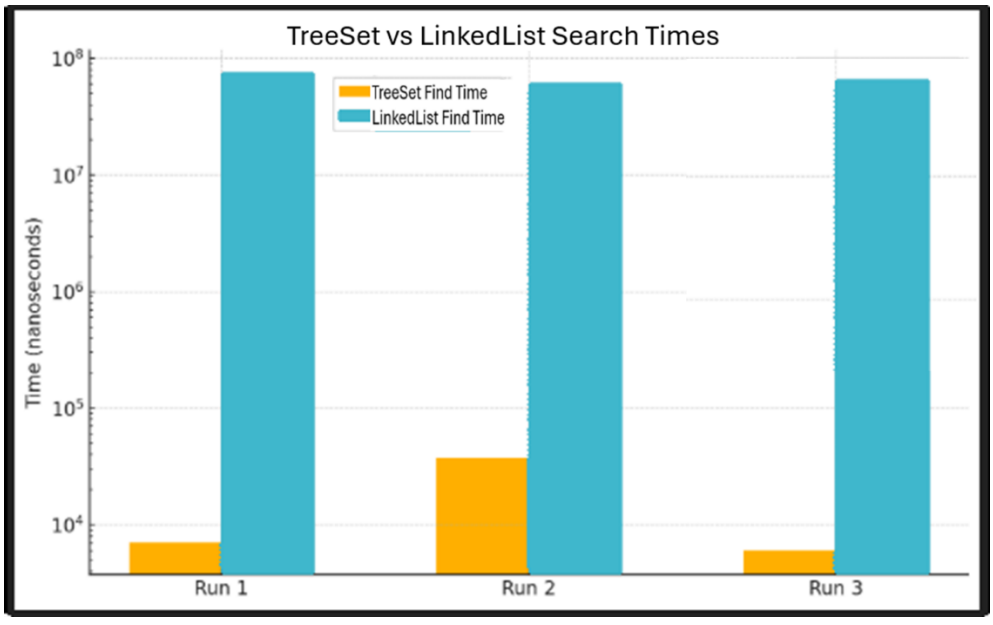
This experiment examines the efficiency of searching for elements in two different data structures: TreeSet and LinkedList. The goal is to measure the average time for each of the structures to find if the element exists. As seen in the table below, the average search time in the TreeSet ranged between 6,000 to 37,000 nanoseconds. In contrast, the average search time in LinkedList was significantly higher, between sixty million to seventy-three million nanoseconds.

These results align with theoretical expectations. A TreeSet uses a balanced tree structure, allowing lookups in O(log n) time, making searches significantly faster for large datasets. LinkedList searches require a full traversal in the worst case, resulting in O(n) time complexity, which becomes prohibitively slow as the dataset grows.

Size	Experiments	seed	Avg Time to Find in TreeSet (ns)	Avg Time to Find in LinkedList (ns)
25000000	20	42	7,101.85	73,451,552.20
25000000	1	42	37,353.00	60,121,208.00
25000000	200	42	6,018.91	60,907,103.56

Search Time Comparison – TreeSet vs. LinkedList
This table compares the average time needed to find the randomly generated number in a TreeSet and a LinkedList for different numbers of experiments. The TreeSet shows significantly lower search times, ranging from 6,018.91 ns to 37,353.00 ns. In contrast, the LinkedList has much higher search times, exceeding sixty million nanoseconds as it requires a full traversal for each lookup. These results confirm that TreeSet is far more efficient than LinkedList for search operations, especially in large datasets.

The findings reaffirm that TreeSet is vastly superior to LinkedList for search operations in large datasets. LinkedList is beneficial for frequent insertions and deletions due to its dynamic structure, but it is unsuitable for fast searching. Alternative data structures like TreeSet, HashSet, or ArrayList (with binary search) do not require a full traversal of all the elements they hold. These structures can be used to improve performance in Java applications when efficient search operations are needed.



TreeSet vs LinkedList Search Times
This bar graph compares the search time performance of a TreeSet and a LinkedList across three experimental runs, measured in nanoseconds. It shows a consistent performance gap, with the TreeSet (orange bars) showing significantly faster search times compared to the LinkedList (blue bars) in all runs.

Conclusion

This report explored how various memory access patterns, caching effects, and data structuring efficiencies impact performance in Java through multiple experimental procedures. The first experiment showed the significant performance penalties induced by the volatile keyword, as it forces direct memory access while bypassing CPU caches. Despite the slower execution, correctness remained intact, so there is a trade-off between memory visibility and speed in concurrent programming.

The second experiment analyzed memory access patterns and reinforced the importance of memory locality and caching. Sequential access to the first 10 percent of an array was significantly faster than randomly accessing just one random element from the last 10 percent. This experiment showed that performance-critical applications should minimize random memory access to avoid cache misses and unnecessary delays.

The third experiment, comparing TreeSet and LinkedList in search operations, confirmed that logarithmic time complexity data structures are more efficient than linear ones. TreeSet outperformed LinkedList tremendously. This experiment reinforced the importance of selecting the right data structure for the proper use case when efficient searching is a priority.

These experiments provide insight into how crucial the selection of memory access patterns, caching, and data structures are in improving Java performance. By understanding and using these principles, developers can build more efficient systems.