PROFILING THE IMPACT OF CACHING, MEMORY ACCESSES, AND CHOICE OF DATA STRUCTURES

Tatiana Gabel

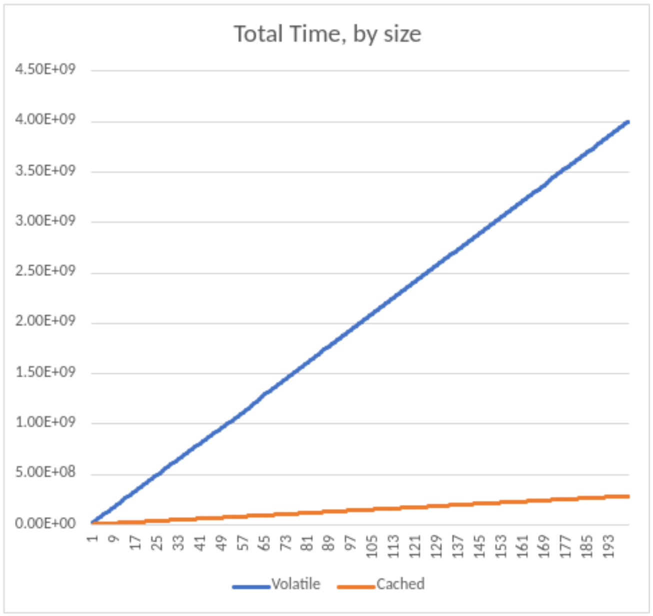
Oct 6th, 2023

Task 1 Comparing Caching and not caching.

For this task we compared the performance of a loop storing simple arithmetic operations when we used a variable with the Volatile keyword and one without. From the description of volatile in the assignment, a volatile variable is a variable that is not cached and instead is always immediately stored in main memory. An article on the volatile keyword explained that often variables will be stored in a cache/buffer, as they will likely be used soon afterwards, so the current value is not always written in main memory. In contrast, a volatile variable makes the processor flush the variable's buffer when it is updated, so main memory always contains the updated value (BaelDung 2023). So, for this task I assumed that the volatile variable experiment would have a longer average time, as the value must be read and written to main memory after every update, unlike the regular variable which makes use of the much faster caches.

Using size as a dependent variable (Experiments=200, seed=42)

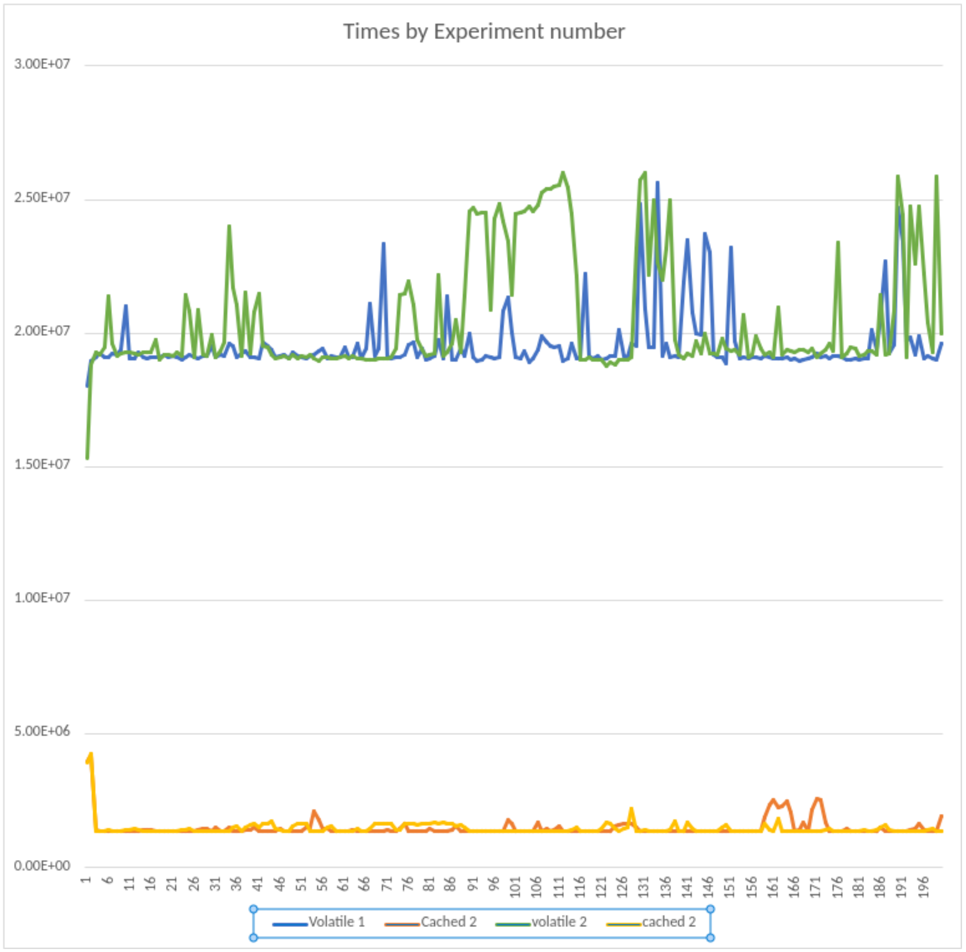
Size	Volatile Time (s)	Regular Time (s)
2.5 *10^5	.0013	.0003
2.5*10^6	.013	.003
2.5*10^7	.13	.03



It is clear that as the size of the numbers used increased the average amount of time it took increased proportionally. However, it is interesting to note that while the regular time is significantly shorter than the volatile time, they also remain closely proportional, which likely means that the average time gets longer due to the type of operation, aka it had O(n) time complexity. That being said the slope of the volatile variable is steeper, meaning with larger and large data sets, the gap between the time of the cached variable and volatile one becomes greater.

Besides the fact that it takes longer to access main memory than cache, volatile variables also limit the amount of rewriting the compiler can do, because it cannot reorder operations above or below a volatile variable, like it typically would speed up execution (BaelDung 2023).

Something that I found interesting about the record times between the volatile variable and regular variable was the standard deviation between values. The data from the volatile variable had a larger variance in the values. From this I understand that the time it takes to access the cache is a lot more consistent than the time it takes to access main memory.



Standard Deviation Volatile	Standard Deviation Cached
1.13 Milliseconds	0.35 Milliseconds

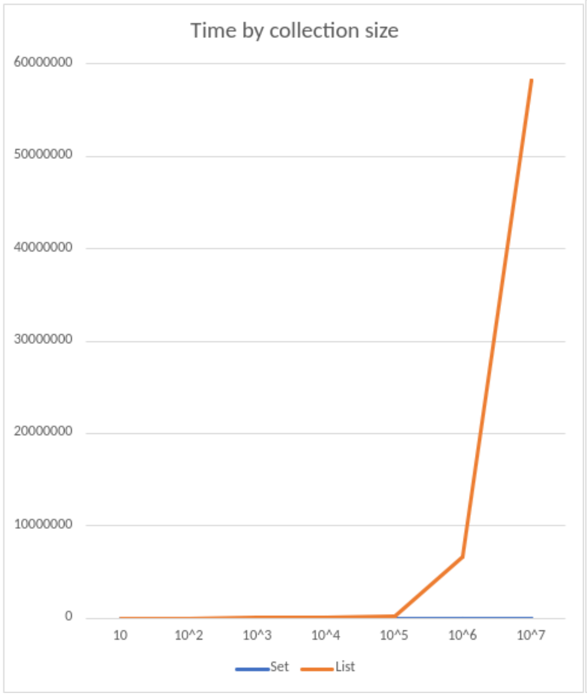
Additionally, I wonder if the spikes are meaningful. For example, the first few experiments for the cached variable recorded a longer time. Could this be because the totalTime variable was not being stored in a higher-ranking cache until it had been used a few times. The other spikes in time might just be random or due to the cache preparing to focus on different instructions, adding a few nanoseconds to the total.

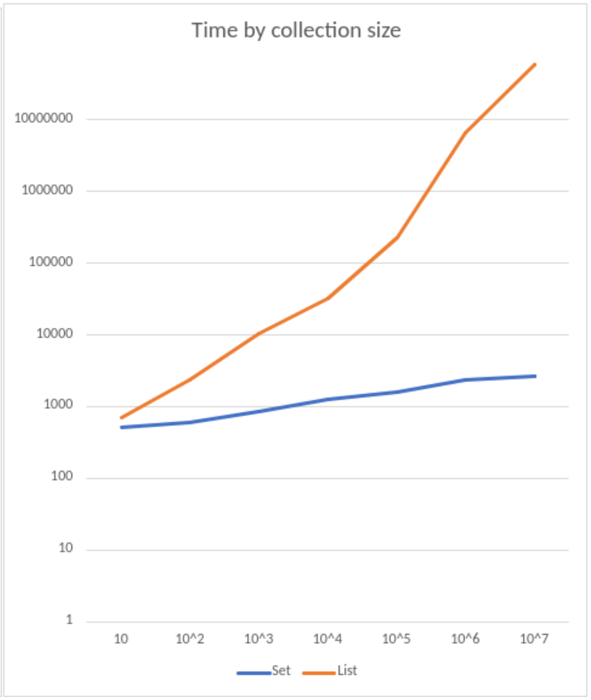
If we apply this same logic, it makes sense that main memory would be less consistent. As more processes may require the bus, delaying the processor's ability to write the updated value to main memory. To double check this I ran the experiment again and added its values to the graph. Between volatile 1 and 2 there is a lot of variability, and there is a lot of variability between cached 1 and 2, but it is smaller than that of volatile.

Task 3 Comparing Tree Set vs Linked List

For task 3 I compared the speed of accessing a value within a tree set or a linked List. These are both data structures but have different properties and implementations. A set is unordered while a list is ordered. This does not have as much of an influence as the actual implementation of each does on the speed. For example, a tree is a data structure that allows for quickly finding a value, because the way it stores values on average should make the path to the value, we are looking for shorter than it would be in a list. Conversely, a linked list must traverse all prior values to find the value it is looking for. Due to what I know about time complexity from 165 I believe that the tree set will have faster access times.

Size of the collection vs time it takes



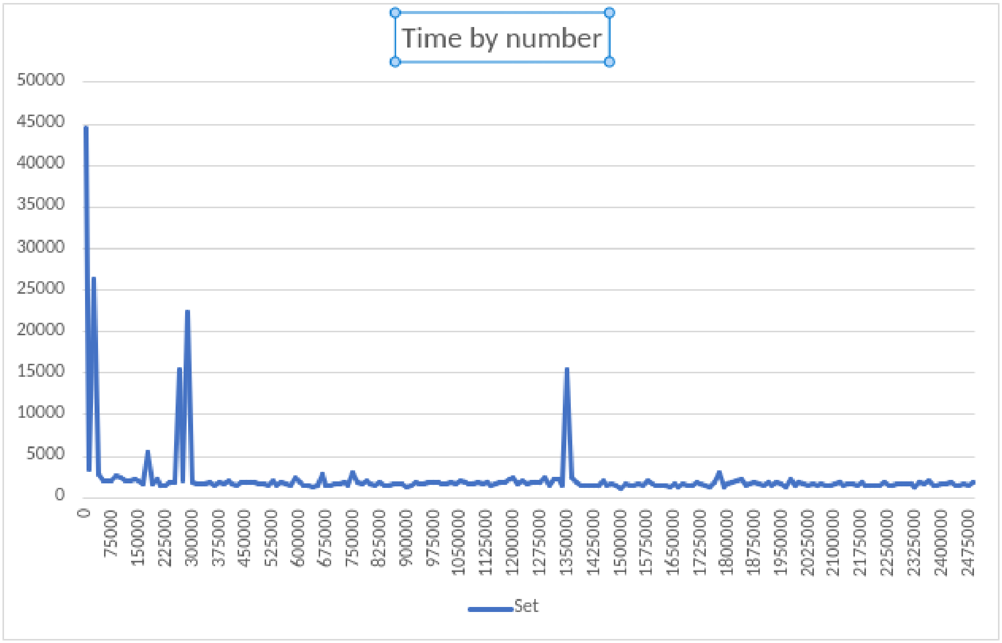


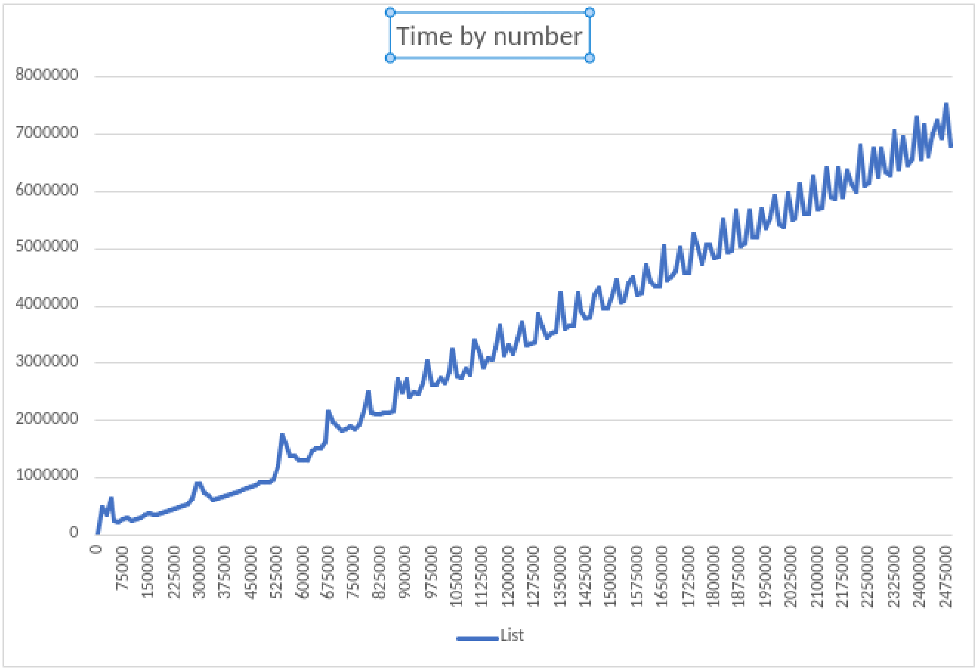
Based on a y-axis Linear scale

Based on a y-axis Logarithmic Scale

For my first statistics I recorded the average time (in nanoseconds) that it took the set or the list data type to a random value. The other parameters were 200 experiments, and seed 42. My independent variable in this case is the size of the collection (either a tree set or a linked list) as listed at the bottom. Starting from a length of 10 elements and increasing by a factor of 10 to 10^7. It is clear that the list took longer on average to locate the value than the set, however I included 2 graphs so that data is not misleading. In the first graph it appears that for the linked list there is an exponential relationship. However, looking at a graph with a logarithimic scale on both axes, it shows there is a slight curve in a linear line for linked list. Either way the Linked list increases as the length of the list increases, taking longer and longer on average to find a random value. Whereas the tree set increases at a slower rate, an increase in its length does not make the time it takes increase like it does with a linked list.

Location of value we are looking for vs the time





For my second statistic I decided I wanted to track the time it took to locate a value based on what number it was. The results also follow closely to what I expected. As the List is a linear graph, which means that it takes longer to find values farther along the list than it does values earlier within the list. I'm not sure why there are regular spikes, but it may have to do with how lists are searched. Similarly, the graph of sets shows that it takes around the same amount of time to find any value regardless of its value. The outliers are the lower 2 values where 0 was the hardest value to find within the tree set. Overall, that gives us some insight into why the set was faster than a LinkedList.

Sources

Various Authors (2023, July 25). *Guide to the volatile keyword in Java*. Baeldung. https://www.baeldung.com/java-volatile