Behind the Merge Sort Algorithm

Brandon Lee

Introduction

When choosing what algorithm to use as a basis for my mini-project, my main considerations were my understanding of the algorithm, and the difficulty extracting data from it. Having studied CMPT225, Data Structures and Algorithms alongside CMPT295, my first instinct was to choose one of the sorting algorithms we worked with in our assignments. After doing some testing between Insertion, Merge, and Quick sort, I decided upon Merge Sort.

While I had initially planned to cover all three algorithms, I began running into problems as I tested the sorting algorithms on various inputs. Insertion sort, which worked reliably, began to hit insane runtimes at array sizes not even close to what we were going to use, so I decided against Insertion sort, as the variance in times would make it difficult to extract data. Quick sort, an O(n log n) sorting algorithm, began to show its weaknesses as it was called to operate on the already sorted and reverse sorted datasets, as well as when called upon to work on larger arrays. On reverse sorted datasets, the pivot value chosen would always be the last value in the array, resulting in *n* number of recursive calls with extremely slow sorting. The excessive recursive calls would then lead to stack overflows due to the call stack having *n* number of calls from the function pushed to it, resulting in the program running into segmentation faults. Because Merge Sort has an extremely consistent number of operations as well as running times, I settled upon this algorithm for the project.

Testing

To begin, I first used the internet to find an implementation of Merge Sort to use in this assignment. I opted not to use my own implementation I lacked certain optimizations and instead opted to use the code from Geeks for Geeks for my program. The first data I gathered was the runtimes of the sorting algorithm on different input types. I started by first retrieving the cache sizes on the CSIL machines so that I could see how memory locality would come into play as I tested every array size. Using the lscpu command, I was able to retrieve cache sizes of 192KiB, 1.5MiB, and 12MiB for the L1, L2, and L3 caches on CSIL. Because I chose to do my testing on arrays of C++ integers (4 bytes), the resulting array sizes were 49152, 196608, and 3588096, respectively. In addition to the three array sizes, I also chose an array size of 100000000 to represent an array which very obviously does not fit into the caches. Once array sizes were in order, I tested the runtimes of Merge Sort under four different conditions in each array size. For each cache there was a call to Merge Sort using an unsorted random array, a sorted array, a reverse sorted array, and a random but many duplicate values array. There were two goals associated with the tests in each cache. First, we wanted to see if the CPU would be able to accurately branch predict to cut down on operating time, and second, we wanted to see if memory locality came into play.

When testing, I used the C++ clock() function to time every sorting call, and then perf to record all branch predictions and misses. The results from the timing tests was that for all array sizes, the algorithm had the same relative running times between the random, reversed, sorted, and many duplicates arrays. The consistent behaviour was that the sorting of the reversed and sorted arrays was done in nearly the exact same amount of time, the fully random array being sorted in almost 2.5x the time it took for the reversed and sorted, and then the many duplicates array taking more time to sort than the reversed and sorted, but less time than the fully random.

This behaviour was consistent across the array sizes fit for the L1, L2, L3, and greater than L3 caches, with an outlier evident in the running time of the many duplicates sort that took place in the L1 cache. Rather than the many duplicates sort performing as expected, which is, that it would be faster than the sorted array but slower than the random, it instead performed at a faster rate than the sorted array. We will get into a possible explanation into this result later.

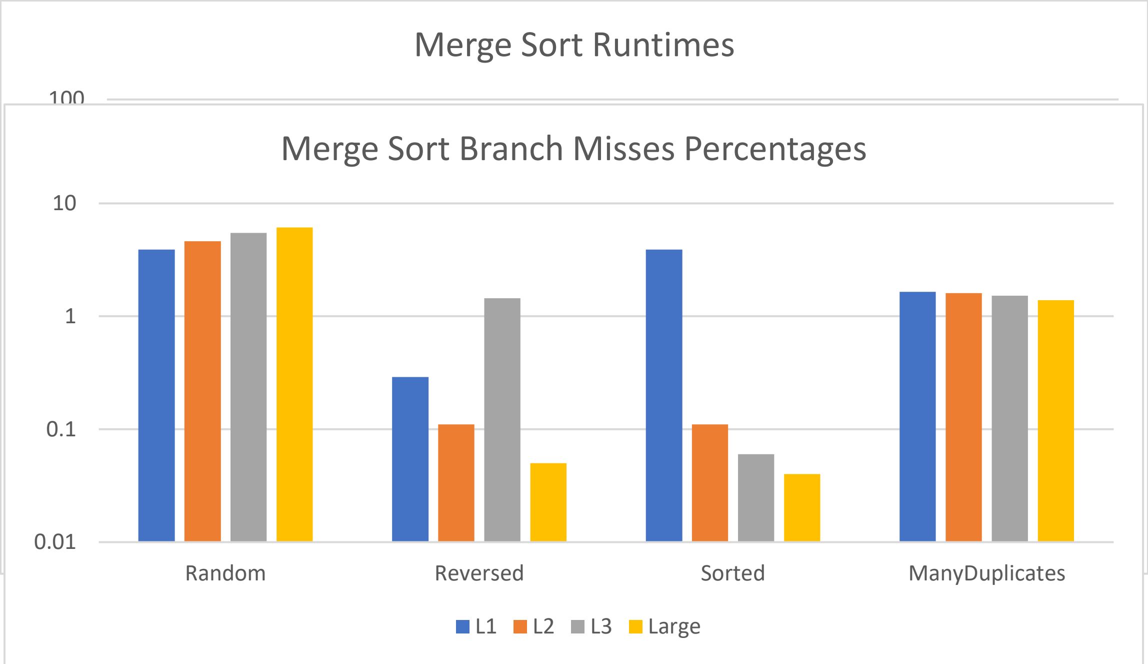

Figure 2 Merge sort Branch Misses on a logarithmic scale to display the relative amount of branch misses between cache sizes.

For examining the branch predictions, I chose to single out the reversed array sort as my testing mode. The reason for this is that with the way the merge sort algorithm works, with every recursive call to merge, the values split into the "right" side array will have to be placed back into the array first, and then the values originally split into the "left" side array will have to be placed back into the array second. Usually, the merging process involves comparing the left and right arrays element by element and then choosing which one goes back into the original array, but because the array was reverse sorted the comparison will always evaluate to whichever value is in the right array will go into the original array first. This is by nature a predictable outcome, and as such my prediction was that there'd be fewer cache misses in the reverse sorted array than in the randomly sorted array. What was observed was that the CPU was able to accurately predict *for the most part* when it came to sorting the reversed and already sorted arrays. With an obvious outlier in the sorted L1-fit array that reported a much higher percentage of branch misses. This can be explained by the fact that with a smaller array, any amount of branch misses that take place will contribute a larger portion to the program as a whole's branch mispredictions. Otherwise, the results were as expected, the random sorts' branch misses steadily increased from 3.89% up to 6.12% for the large dataset, the sorted and unsorted arrays had branch misses as low as 0.05% and ignoring the outlier, misses as high as 1.45%. The many duplicates sort had branch misses between 1.39% and 1.65%.

Brandon Lee

Analysis

There were a few results from testing that were cause for further analysis into the "why" of how it happened. Most notably, the runtime of the many duplicates sort within the L1 cache seemed to perform faster than sorting the already sorted array. While normally I would attribute this to just cache locality, if that were the case then the fully random sort would have a similar runtime. The difference between the many duplicates and the fully random sorts is the range and size of the values. When propagating values into the many duplicates array, I did so by using rand() %10, effectively limiting the range of values to be between 0 and 10. What this likely leads to is long strings of the same value, where an entire array before the merging process can be entirely one number. When these long strings of a single number are present, it's likely that the CPU is able to predict with higher accuracy which number is next to be slotted back into the array during the merging process. This is evidenced by the lower percentage of branch misses than the fully random sorting.

Another thing to note is that despite the large dataset being too large to fit within our caches, it had no notable impacts on runtimes. The large dataset was roughly 30x the size of the L3 cache, and surprisingly had 30x the running time as the L3 sort, clocking in at 15 seconds to sort the large dataset versus the 0.5 seconds to sort the L3 sized data set. This has the implication that memory locality has no impact on a sorting algorithm such as merge sort, which uses recursive function calls to do the merging.

Conclusion

Overall, Merge Sort behaved in ways that were unexpected to me. I had anticipated that the recursive function calls might mess with the data locality and thus disrupt the cache locality, whether this is the exact case with my test, I am unsure. While I had assumed that the branch predictor would make quick work of the reversed and sorted arrays, I did not anticipate its interaction with the many duplicates array with the merging algorithm. Out of all the tests done I found the results from the many duplicates array to be the most surprising, but mostly due to the way the merge sort algorithm naturally groups duplicate values as the recursion unravels, leading to branch predictable subarrays. If I were to examine another algorithm, I'd choose one that does not use recursive function calls so that I might be able to determine whether the recursive function calls disrupted the memory locality.