CMPUT 379
Assignment 3
Performance Study
Group 31

Stefan Vidakovic
Will Nichols

April 06, 2018

Introduction

The following report details a performance study with respect to memory access for three common sorting algorithms, merge sort, quick sort and heap sort. The specific implementations of these sample programs can be found in the assignment folder, briefly speaking, the sorting in all three is performed on an array of randomly generated numbers (using rand()) where the array size is specified as input on execution. Subsequently, the working set (approximated by number of unique page references) of these programs was calculated with the use of valgrind's lackey tool and our valws program. Output from valws was plotted on a time series and frequency histogram using gnuplot. Various input scenarios were tested, including different array sizes to sort, different window sizes, as well as the option to ignore instruction fetches. For the sake of a concise report not all plots were included, however they can be found in the assignment folder.

Results

As can be seen in all of the time series figures, the sorting programs behaved in essentially the same manner with regards to magnitude of memory accesses, with an initial, approximately linear growth followed by a long plateau with negligible oscillation for the rest of the program's runtime. The differences between the three sorting algorithms can be seen primarily in the plateau or one might say steady state working set size. As seen in figures 1-3 (found in Appendix) where the inputs are the same across all three, it is immediately evident that quick sort is the most efficient in terms of memory references, followed by merge sort and finally heap sort. The efficiency of quick sort comes as over the others comes as no surprise as it sorts in place, and does so purely on the stack, through recursion. Where merge sort requires two arrays to sort (not in place), hence in general resulting in more memory references, quick sort eliminates this redundancy. This is shown by the difference in steady state working set size of qsort vs merge sort, (figures 1 and 3), with merge sort having roughly 115,000 more unique pages. This behaviour is consistent for larger input array sizes and window sizes, as can be found in the assignment folder. The difference between qsort and heap sort is slightly more substantial, roughly 470,000 more unique pages used in heap sort. This can be explained by heap sort's usage of a heap data structure located separately from the stack. Naturally accessing and editing a heap in memory on every recursion results in a higher working set.

As seen in figures 4 and 5, where the performance of heapsort is measured with an array size of 100000, first with instruction pages ignored (figure 4), and subsequently included (figure 5), the working set increases substantially with the inclusion of instruction pages in the calculation. Such a result is naturally expected.

Appendix

Figure 1. Quick sort working set, array size 5000, page size 1024 bytes, window size 1 million pages, instructions ignored.

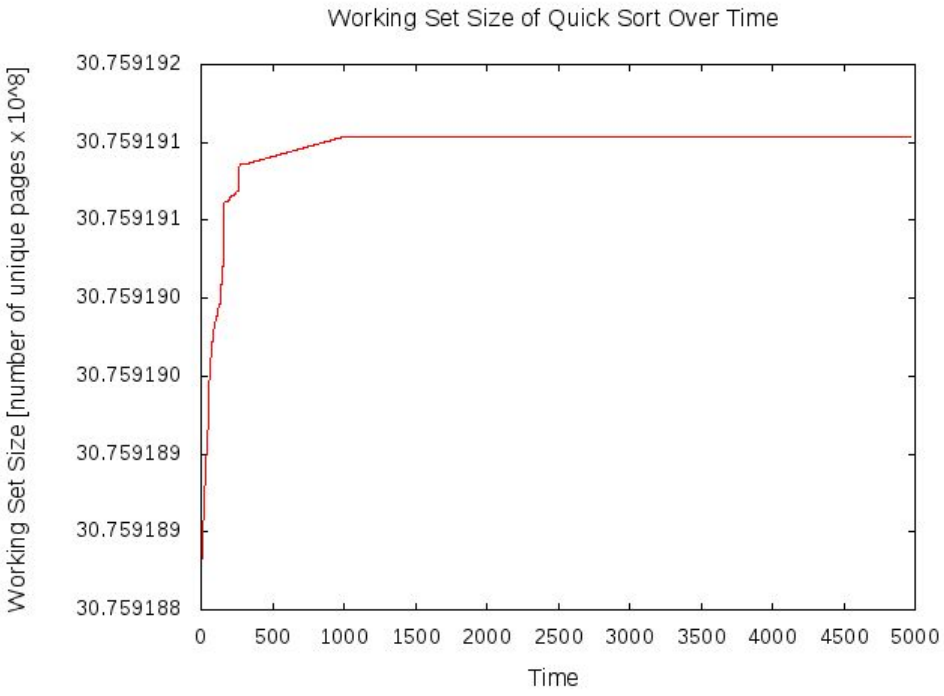

Figure 2. Heap sort working set, array size 5000, page size 1024 bytes, window size 1 million pages, instruction ignored.

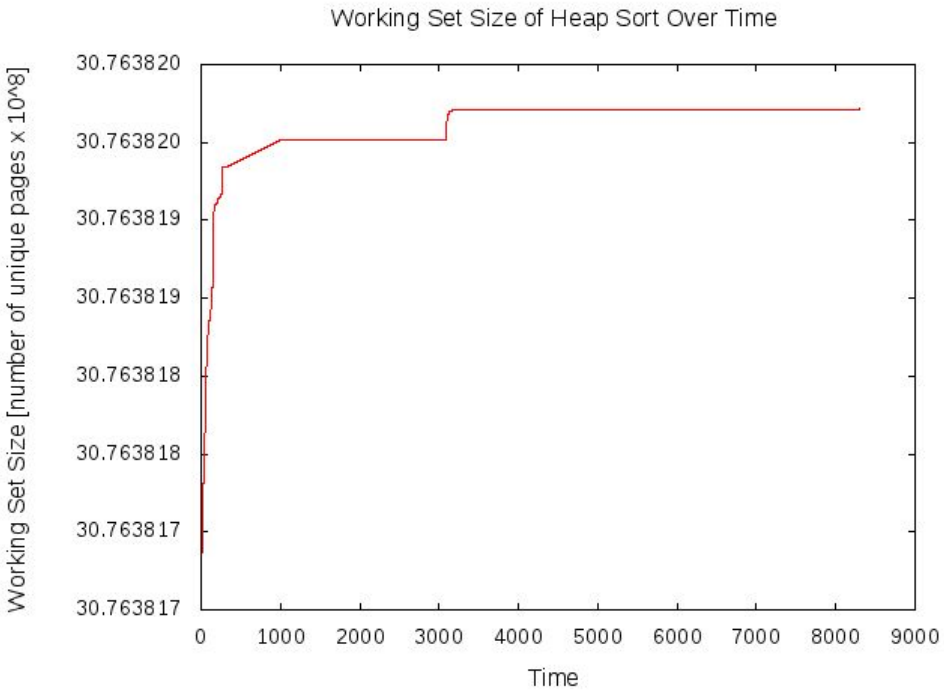

Figure 3. Merge sort working set, array size 5000, page size 1024 bytes, window size 1 million pages, instructions ignored.

Figure 4. Heap sort working set, array size 100000, page size 1024 bytes, window size 1 million pages, instruction ignored.

Figure 5. Heap sort working set, array size 100000, page size 1024 bytes, window size 1 million pages, instruction included.

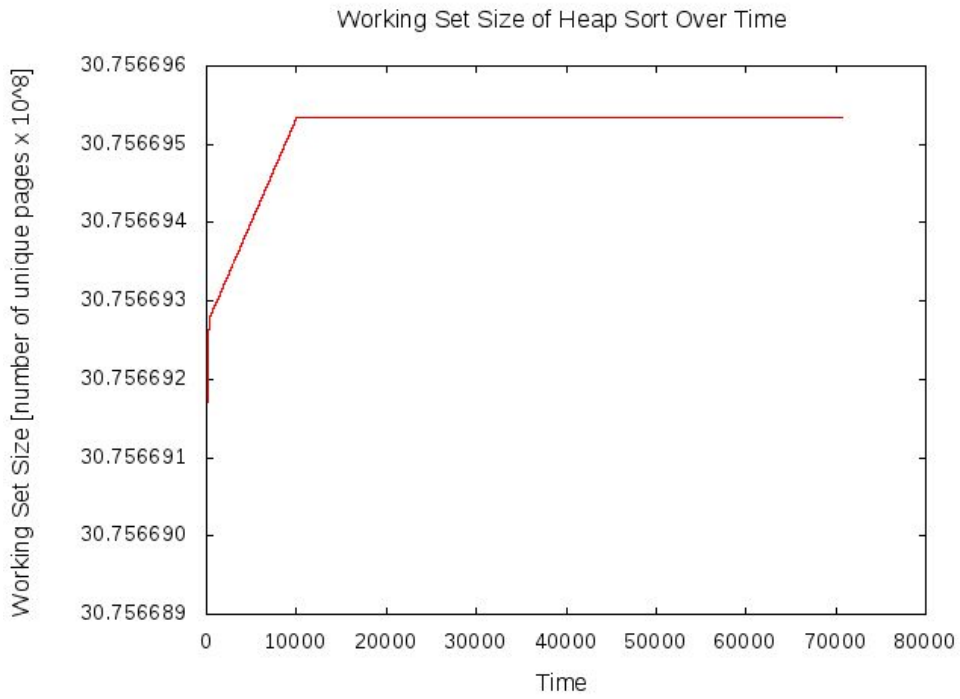

Figure 6. Heap sort frequency histogram, array size 1000, page size 1024, windows size 1 million pages, instructions included.