Exploration of Memory Address Space Layouts and Cache Warmup Effects On The Runtime of Sorting Algorithms

A Good First Introductory Case Study to the Field of Performance Evaluation

Shukry Zablah

06 May 2019

Abstract

In this paper we explore and verify the existence and ongoing impact on runtimes of the pseudo-random effects of memory layout at the time of execution. These effects are still relevant today as we show that modern hardware does not address these conditions that make accurate performance evaluation harder. This paper is a perfect introduction to the field of performance evaluation at the undergraduate level because it explores these phenomena in the context of well known sorting algorithms.

Contents

1 Introduction

It is an error for the computer scientist to look at a computer through the eyes of a natural scientist. Computers are deterministic machines, and the same input will always cause the same output. The intricacies of a machine obscure its deterministic nature, which is why phenomena such as the effect on runtime of memory layout of processes at runtime is surprising. This phenomenon is widely known to introduce pseudo-random variation to program runtimes, such that evaluation of their performance has to take into account this challenge (Myktowicz et al).

In this paper we verify that the challenges for performance evaluation still remain in modern hardware, and we provide an evaluation of a selection of sorting algorithms that will be helpful for undergraduate students interested in the field of performance evaluation. We find that the displacement of the environment size, which in turn affects the memory layout of the process at runtime, can have a measurable but unpredictable effect on the runtime of programs. Additionally, our exploration of the different sorting algorithms leads us to explain how cache effects have an impact on the first iteration of algorithms that call `malloc()`, the memory allocator.

The paper is organized into three sections. First, in the **Experimental Design** section we discuss our implementation choices, procedures, and our methods to both time and evaluate our programs. Next, in the **Experimental Results** section we discuss in depth the outcomes of the experiment and how these results verify the existence of the effects we mentioned previously. The last section is the **Conclusion**.

2 Experimental Design

This evaluation was performed by a large collaborative effort from the COSC365 - Performance Evaluation students in Spring 2019. In choosing stable benchmarks to measure the effects we have described, we chose implementations in C (eddyerburgh) for the following sorting algorithms:

	Algorithm	Asymptotic Runtime
1	Bubble Sort	$O(n^2)$
2	Counting Sort	$O(n)$
3	Insertion Sort	$O(n^2)$
4	Merge Sort	$O(n * log(n))$
5	Quick Sort	$O(n * log(n))$
6	Radix Sort	$O(n)$
7	Selection Sort	$O(n^2)$

Table 1: These are the sorting algorithms that we chose to evaluate. Note that we have a variety of runtimes, both comparison based and not. We observe in Fig 1 their performance relative to each other.

We needed C implementations to make our performance evaluation as accurate as possible. Implementations in other languages would also time the functional wrappers and safeties built into those languages, which would give inaccurate results. Additionally, we chose sorting algorithms because introductory level courses make a good job at describing the theoretical runtimes of these algorithms. Benchmarking programs that are well-behaved in this manner is not limited to sorting algorithms, but they provide a load that is easily adjustable, and do work for a similar end (i.e. sorting).

Our independent variable is the size of the environment variables in the system. Since the environment variables live in the highest portion of the virtual address space of every process, then this displacement would also cause the displacement of the stack and its contents. Particularly, we measure the difference in behavior of programs under a 16B change in the size of the environment variables up to a total of 4KB (a page size). To create this change, we grow a specific environment variable before running each program, thus ensuring that the process runs under the desired conditions.

In order to time our programs we considered a variety of different tools such as Unix's `time` command, but settled for the high-precision `clock_gettime()` function in the `time.h` library. This function provides a friendly interface for high-precision timing mechanisms, of which we chose `CLOCK_PROCESS_CPUTIME_ID` in order to get the virtualized timing information for a particular process. If we were to choose `CLOCK_MONOTONIC` we would be able to time the actual time spent on both the process and the kernel. This function gives nano-second precision, which allows us to measure the small variations that we expect from the differences in memory layout of the processes.

The timings themselves were done on Amherst College's high-performance computer cluster. The task management system is Condor Version 8.6.5 running on x86-64_RedHat6. The cluster consists of 120 CPU cores running on 6 computers. The nature of our experimental setup allows us to run the timing experiments quickly and efficiently. We coordinated our efforts by developing a variety of shell scripts that would generate the description of the tasks to submit to Condor, collect the results, and enter them into a MySQL database for statistical analysis in R. (For more information on the architecture of the college's computer cluster please refer to the college's website.)

To get a sense of the experimental setup we provide a view of the first six observations in our results.

	algorithm	length	range	iteration	runtime	envsize
1	bubble	150.00	1000.00	1.00	19354.00	0.00
2	bubble	150.00	1000.00	2.00	16652.00	0.00
3	bubble	150.00	1000.00	3.00	14870.00	0.00
4	bubble	150.00	1000.00	4.00	14043.00	0.00
5	bubble	150.00	1000.00	5.00	12768.00	0.00
6	bubble	150.00	1000.00	1.00	15015.00	1008.00

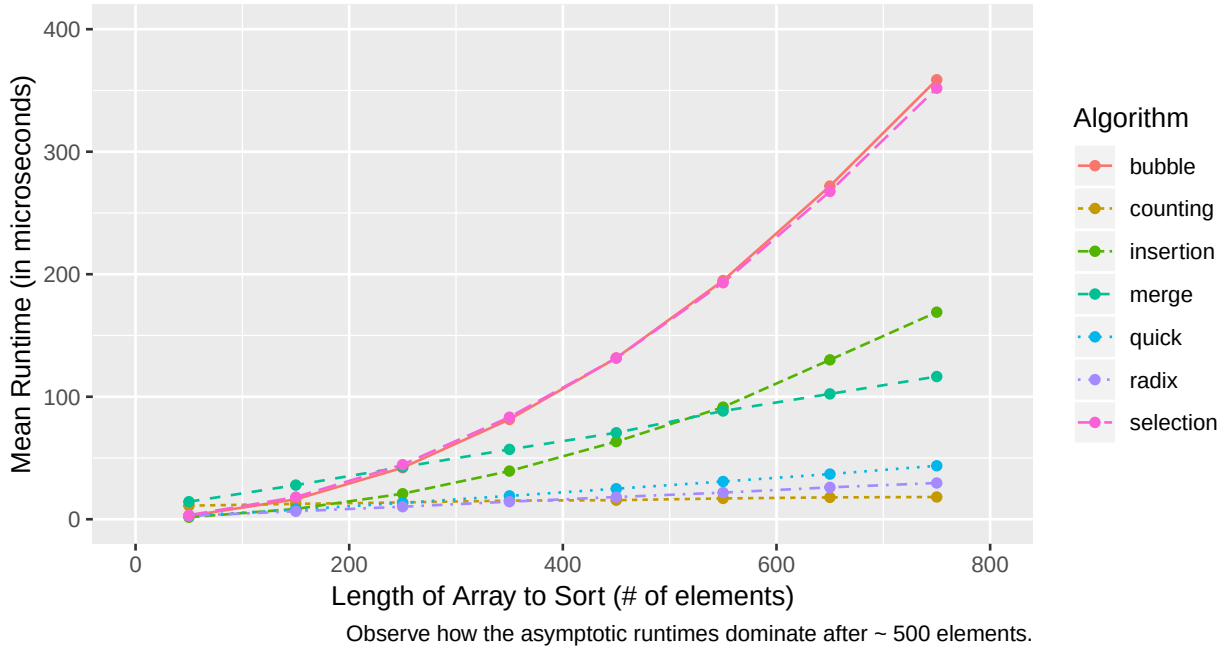
Table 2: First six observations of the timings without an initial call to malloc(). Notice the columns give you an idea of the independent variables in our experiment.

Overall, for each of the seven algorithms we ran and timed 5 iterations at every one of the input lengths to

sort (from 50 to 750), for array elements that were within 0 and 1000. In every one of these iterations we also tested the 257 different environment sizes.

The reason we chose these input lengths was because a preliminary set of runs showed that the asymptotic runtime of the sorting algorithms had a great effect quickly. Observe the following figure to see this effect:

Fig 1: Mean Runtime of Sorting Algorithms by Input Size



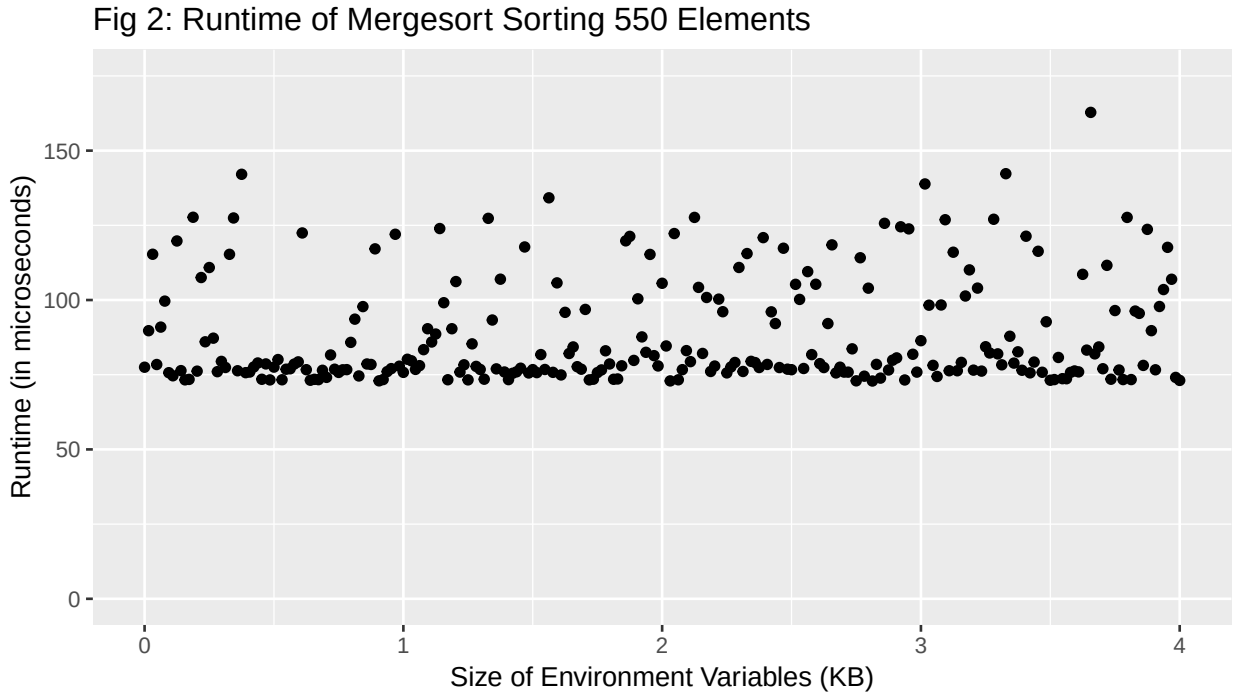
Observe how the asymptotic runtimes dominate after ~ 500 elements.

Finally, for reasons discussed in section 3.2, that were evident after the analysis of the runs without an initial call to `malloc()`, we reran all the experiment with the same parameters but also included an initial call to `malloc()` for every run. This initial call to malloc did nothing and was freed. Its purpose was to avoid timing the cache warmup that would result from the initial call to `malloc()`. These decisions allowed us to consider a variety of dimensions in our analysis, and have a clear picture of the effect that we were trying to measure.

3 Experimental results

3.1 Sensitivity to Memory Address Space Layout

To prove that the memory layout of a process introduces unpredictable variation, we single out all 257 iterations of mergesort with an array to sort of length 550. The only independent variable that is left is the environment variables size (recall we use this as a way to displace the memory addresses).



The impact of the environment size on the runtime is evidently not related to the environment size itself. This is significant because it shows that even on modern hardware, the effect of displacing the stack by growing the environment variables size still has a significant impact in the performance evaluation of programs. Specifically, timing experiments fail to accurately represent the time spent on a task on one run. The following table helps show the impact in the worst-case scenarios:

	Algorithm	Length	Max Difference (in microseconds)
1	selection	750.00	364.56
2	selection	650.00	269.46
3	bubble	750.00	249.59
4	bubble	650.00	216.85
5	insertion	750.00	195.81
6	selection	550.00	184.99

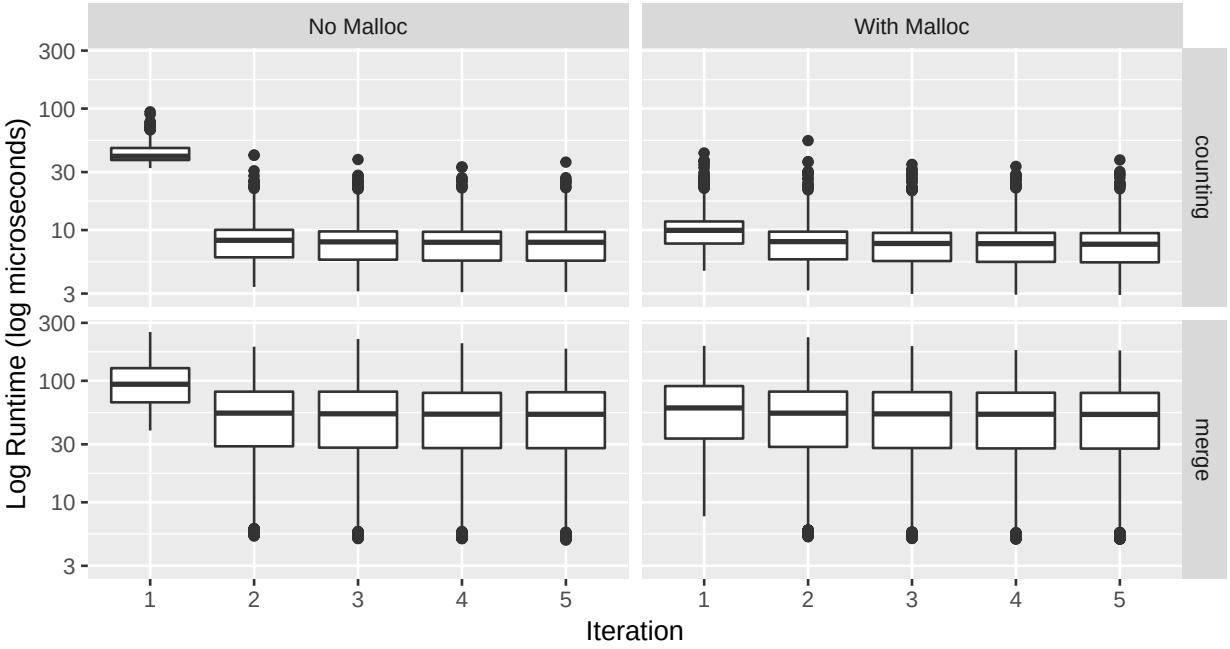
Table 3: The top maximum differences of any combination of algorithm and length of input. The largest difference means that there were 364.56 microseconds between the slowest and fastest runs under the same load.

We observe that the maximum difference for any combination of algorithm, and input length is of about 365 microseconds. This means that there was 365 microseconds between the slowest and fastest runs for the same load! This is proof of the complexity that sensitivities in the memory address layouts add to performance evaluation.

3.2 Cache Effects

Under closer examination of our data, the variation in times was not wholly caused by the growth of the environment variables. We also realized that some of the first iteration of the sorting algorithms were slower than the other four iterations. This is evidence of the existence of an effect in timing due to the caches. Caches are dedicated memory units that are usually operating at much higher speeds than main memory (RAM) but have extremely limited size, not more than a few of kilobytes. Caches operate under the assumption that memory pages that are recently used will be used again soon. To this end, caches store data that a program can access faster than when it fetches the same data from main memory. However, for the data to be in the caches it has to be used already, so the first time the program uses that portion of the data, it will have to fetch the page from main memory. This is the explanation we attribute to the slowdown seen in the first iterations.

Fig 3: Comparison of First and Subsequent Runtimes by Sorting Algorithm



Counting sort and merge sort are the only two algorithms in which the implementation requires the usage of `malloc()`, the memory allocator. Since `malloc()` has to be called in order to sort the array per each call to our sorting function, we are in effect timing some of the cache warmup of the inner data structures of the allocator. To fix this we tried running a dummy call to the allocator before calling the sorting algorithm. This decreased the difference between the first run and subsequent runs.

Two observations can be made. The first one, counting sort had a greater difference when comparing its first run to its subsequent runs. Further exploration uncovered that the implementation of counting sort had a memory leak, i.e. it did not free the memory it allocated. This could relate to the greater difference as compared to merge sort. The second obsevation is that the slowdown is decreased with the initial call to malloc, but is still present. This could be the effect of instruction caches warming up, but we have not tested the root cause of this effect, which is still only present in these two algorithms.

4 Conclusions

Even when efforts are made to control an environment for performance evaluation, there can still be undesired variation. Modern hardware is still susceptible to small variations in the memory layout of a program, and has a strong effect on their runtime. This effect is unpredictable, and we demonstrated it by artifically shifting the data by pushing the environment size memory space into lower addresses in each of our trials. We found that the first iterations of the sorting algorithms were not slower than the rest, only for counting and merge sort. This difference was less pronounced when there were initial call to mallocs, but still present.

4.1 References

- Producing Wrong Data Without Doing Anything Obviously Wrong!, by Myktowicz, et al.
- Stabilizer: Enforcing Predictable and Analyzable Performance, by Curtsinger and Berger
- Amherst College General Computing Cluster Specifications
- https://github.com/eddyerburgh/c-sorting-algorithms