Joel Howse - 1507593

COMPX304 ASSIGNMENT 3 REPORT

SOFTWARE DESIGN AND APPROACH

I had a standard and expected approach. I created large arrays of increasing sizes, with the intention of increasing the array size beyond the size of the last level cache (LLC).

To measure the performance of an array, the program would access an element in the array and increment the value. A simple operation which should always have a constant execution time. For each array size we want to evaluate, the algorithm loops for a constant number of steps and records the execution duration.

The initial array sizes to search are hardcoded in, ranging from 1KiB to 64MiB. This should encompass the LLC sizes of most modern processors. The sizes are scaled exponentially to give reasonably spaced steps between 1KiB and 64MiB. Each step is scaled by twice the previous value.

Once a reasonable range has been found on the exponential scale, the algorithm performs a second search using a linear scale between the identified range. This allows the algorithm to get a closer estimation as the exponential step sizes were not comprehensive enough but work well at searching many possible sizes.

The most likely range the cache size is assumed to be in is the range of array sizes where the time difference was the largest. On a graph this is usually where the line is the steepest. The reasoning for this is that when the execution duration increases significantly from the previous, we might assume that the CPU has needed to use another cache level, or in our case start accessing the main memory. The same logic is applied to the second loop where we actually return our estimate, we return the bottom array size of the pair that had the largest difference in timings.

To get around the inconsistencies of CPU prefetching, the program uses a pseudo-random number generator to access random indices in the arrays while measuring the performance. This makes it impossible for the CPU to estimate where the next access will be. Hitting the same cache line twice with the above technique is unlikely, but to make sure it never happens I created a structure with a size of 64 bytes, the cache line size on the machine being tested. This means that for each array access you can guarantee that it will never be on the same cache line, assuming you are not accessing the same index twice.

TESTS CONDUCTED & RESULTS MEASURED

I built in some plotting functionality which plots the array sizes against the execution time. This helped a lot with the debugging and refining process, as you can visualize why the program makes the decisions it does.

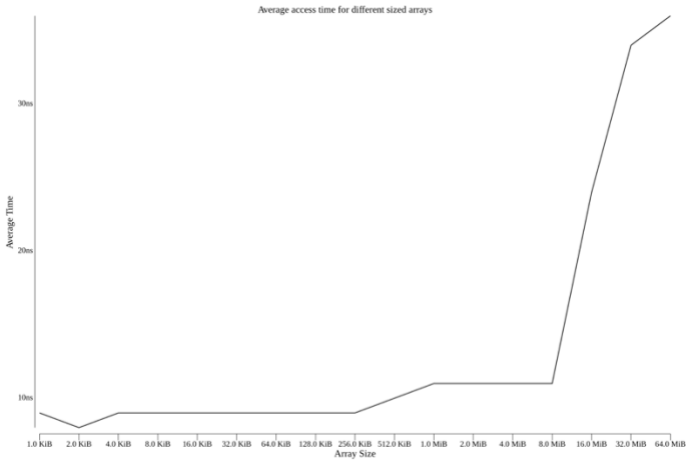
My first idea was to sequentially access a large array and look for any spikes (meaning that access had missed the cache) but this is obviously flawed, CPU prefetching can easily predict where you will access next and prefetch that data. From there I tried some of the techniques talked about in the linked blog post, using the plotting to help me understand what was happening, which lead me to my solution.

The specifications of the PC I ran are:

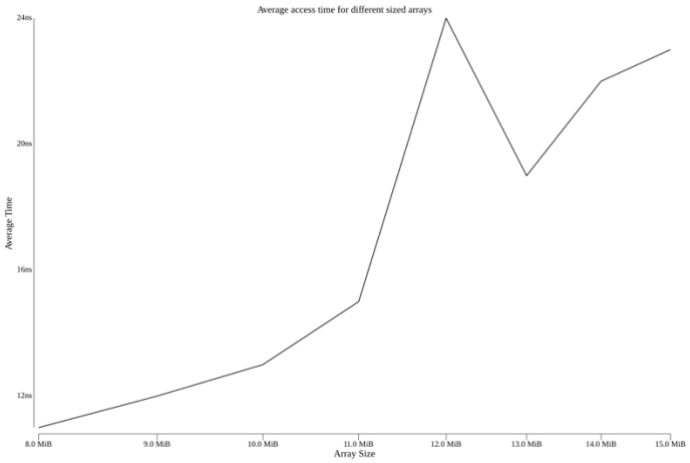
- LLC (L3): 12MiB
- Cache Line Size: 64 bytes

Generally, the algorithm correctly found the cache size. I found that if I didn't run enough loops then the results would be more sporadic. If it didn't get the exact cache size then it would always get close, only of by one or two points.

Example Plots For Algorithm



Step 1. Exponential search through array sizes



Step 2. Converge to result with linear search

Discussion

As discussed above the results were generally good and accurate. The main issue with the algorithm is the randomness. I found on my machine the algorithm would guess correctly ~75% of the time, and for the other 25% of time it would be off by one or two linear steps. Very occasionally the exponential step would fail, and the result would be completely wrong, this happened maybe once or twice in my testing.

One concern I have with this approach is the number of hard-coded values, I'm not sure how well the algorithm would perform on different machines. The approach I've used here works great on my linux amd64 machine, but on my MacBook Air with a m1 processor, the algorithm would always overestimate the cache size. A different approach would be required for the M1, the LLC was always found at the first increase in execution time. While the accesses were happening within the LLC the speed was constant (which is impressive!). M1 also has some other quirks like unified memory which integrates the RAM directly into the CPU. This is novel hardware architecture and it doesn't surprise me the results were different!