INF-2200: Computer architecture and organization
Assignment 1

Adrian Moen, Amund Strøm
UiT id: abc123@uit.no & ast283@uit.no
GitHub users: AdrianMoen & ast283
GitHub Classroom: assignment-1-amund

September 22, 2022

1 Introduction

When a program is executed, a compiler will take the source code or high-level code such as C and compile it to a machine language such as Assembly. Assembly is essentially a textual representation of machine language, instructions which the computer translate directly to binary code. In this assignment we will take a sorting algorithm written in C, and re-write it in Assembly. We will then run a micro benchmark on both implementations to find out if our Assembly implementation executes faster than the Assembly code which the compiler creates. We will be using two different sorting algorithms, and run them against the highest optimization the compiler offers.

2 Methods

2.1 Design

In our case we choose to take two different sorting algorithms and re-write each individually to Assembly. These algorithms are called Cocktail sort and Quick sort, which are not the most efficient algorithms but are easy to understand and therefore is a great introduction to Assembly code.

Cocktail sort works by comparing the first position of the array with the second, and swap positions if the greatest number is the first element. Repeat this process to the last position of the array, and we will end up with the greatest number at the end. The next step is to do this the other way around from the last to the first position of the array, which will give us the lowest number at the start. Repeat this pattern and we will end up with a sorted list. To optimise this we will stop before the lowest and greatest numbers for the next iterations.

Quick sort works by first choosing what we call a pivot. From this pivot, we find the first item from the left part of the array that is larger than pivot, and the item from the end that is less. Swapping these, we continue until we have a new swap, and the index of item from start, is larger than item from end. Pivot is moved to the item from left index. By now, all items left from pivot is less or equal, and all elements to the right are above or equal. Calling this recursively with the new pivot position as a partition point, will eventually sort the whole array. There are different ways of optimizing this algorithm, such as dynamically picking pivots and so on.

Now that the algorithms are understood, it is much easier to re-write them into Assembly. The approach was to divide the C program into different subsections, such as loops and other statements, than go line by line in the C program and translate each individual line into Assembly. This approach made it easy to test and debug with regular intervals.

When implementing in Assembly there are two important concepts to keep in mind; The first is that you only have 8 registers that can hold values, which means that you have to prioritize if there is necessary to assign new variables. The second concept is that if you for some reason don't have enough registers and want to keep some of the values that are on them, you can save them on the stack. Using the stack is preferably avoided, since putting something on and off the stack is very time consuming. If this is done often, it will slow down programs drastically.

2.2 Implementation

As mentioned in the design section, there was important to look for solutions where you could avoid using the stack when implementing the Assembly code. The reason for this is that accessing the stack is time consuming and will slow down the efficiency of a program.

When re-writing the Cocktail sort, avoiding using the stack was not a problem. There was only 5 out of 8 registers that was needed to make the algorithm work. Therefor there was no point in using the stack, and matched quite well with the desired design.

The quicksort assembly implementation had a few more issues and deviates slightly from the theoretical implementation of such a sorting algorithm. Where our implementation differs, is based on how the C implementation that we borrowed from geeksforgeeks[4] completes this task. This implementation does not search from left and and right. Instaed, an index is incremented for each step, and another increments every time we find two elements to swap. Both indices start from the left, and exits once surpassing the pivot. Another thing to keep in mind is that the hand written assembly version does not only handle the partition part of the algorithm, which would be the hotspot, but the whole thing. As a result, the pushing and popping done in order to keep this recursive function in check, is probably less than optimal.

2.3 Experiments

To benchmark these algorithms we simply took the time before and after the algorithms were executed, both C and Assembly versions, and calculated how much time the program spent inside them. By doing so we know exactly how many seconds the algorithm as a whole uses to sort a list. This method does not locate the hotspots, which tells us where in the algorithm it spends most its time, it can be something like a function or loop. We did not find it necessary to locate the hotspots, since we re-wrote the whole algorithm and not specific parts of it.

The length of the arrays differ from the two algorithm. While quicksort handles 10 million numbers in under a second, cocktailsort can not complete this within a feasible timescale. Thus, quicksort will be sorting with 20 000 000 numbers and cocktail with 100 000 numbers. To eliminate as many variables as possible during the test, we seed the rand function with an arbitrary number, such that we get the same array every time, and the arrays are freed between the quick and cocktailsort

3 Results

Test	run 1	run 2	run 3	average	std deviation
Cocktail C	9.898	9.897	9.932	9.909	0.0199
Cocktail Asm	9.440	9.420	9.469	9.443	0.0246
Quicksort C	1.758	1.747	1.756	1.754	0.0059
Quicksort Asm	1.903	1.900	1.906	1.903	0.0030

Table 1: Array length; cocktail: 100 000, quicksort: 20 000 000, with O3 flag, numbers in seconds.

4 Discussion

4.1 Results

Looking at the results we can see a few interesting points. First of all we see that the two assembly sorting algorithms don't behave the same considering their C counterpart. The cocktail sort written in assembly consistently performs better than the C code, despite O3 optimization. The same can't be said about the quicksort however. If we turn the optimization down to O0, the assembly consistently performs twice as fast than the C counterpart, for both algorithms. If we drop the quicksort length down to 10 000 000, the assembly quicksort can even beat the C code in about half of the runs. An important detail to note, is that the assembly quicksort might be less than optimal based on the fact that the whole thing was written recursively in assembly. Had we only implemented the hotspot, it could have been a little different

4.2 Conclusion

We have, in this task, proved that we can indeed beat a compiler, given the right circumstances. While the compiler with O3 optimization beats the quicksort every time with 20 000 000 numbers, our cocktail sort beats the compiler.

5 Sources

References

[1] Tanenbaum, Andrew S. Modern Operating Systems, 4th edition. Pearson, 2015. Chapters 3.3-3.4.

[2] OS dev contributors. (31 August 2018). Paging. In OSDev wiki. Retrieved 22:24, April 27th, 2019, from `https://wiki.osdev.org/Paging`

[3] Wikipedia contributors. (2019, March 7). Page replacement algorithm. In Wikipedia, The Free Encyclopedia. Retrieved 18:45, April 29, 2019, from `https://en.wikipedia.org/w/index.php?title=Page_replacement_algorithm&oldid=886600514`

[4] https://www.geeksforgeeks.org/quick-sort/