CS2106 Optional Challenge 1

by Lee Jia Wei

Introduction

I approached this challenge of writing a recursive program to compute the 42nd VCN in 3 ways:

1. Using `%rbp` to store/read data on the stack
2. Using `%rsp` to store/read data on the stack
3. Using `malloc` for memoisation and help of registers to store/read data on the stack

The result of the 42nd VCN from all 3 variants is **299853833105**. The code is written for `gcc` compiler. For debugging, I utilised `gdb` which is also available on the `xcne` clusters.

Files

There should be 1 driver program, 3 assembly programs and 3 ready-to-run binaries which represent the 3 variants named according.

Driver

`main.c` links the assembly program to be run with an entry function signature `long long ent(void)`. The function is set to run for `N_ITERATIONS`, which defaults to `5`. The whole process will be timed and the average time taken for an iteration will be shown.

Assembly Programs

`vcn.s` contains the source code for variant 1 where `%rbp` is used to store/read data on the stack through the recursive calls.

`vcn_wo_bp.s` contains the source code for variant 2 where `%rbp` and its variants are **not** used. Instead, `%rsp` is used to store/read data on the stack through the recursive calls.

`vcn_memoised.s` contains the source code for variant 3 where memoisation is used.

Testing

Building

To build and compile:

```
gcc main.c [vcn_variant].s -o [vcn_variant]
```

where `[vcn_variant]` refers to the variant of choice.

Running

Compiled binaries are named accordingly and are also available to run. A sample run of the programs are as follows:

```
jiawei@xcne2:~/cs2106/oc1$ ./vcn
The value is 299853833105
Total time taken for 5 iterations: 2544.713786s
Average time for 1 iteration: 508.942757s
```

```
jiawei@xcne3:~/cs2106/oc1$ ./vcn_wo_bp
The value is 299853833105
Total time taken for 5 iterations: 2428.650751s
Average time for 1 iteration: 485.730150s
```

```
jiawei@xcne2:~/cs2106/oc1$ ./vcn_memoised
The value is 299853833105
Total time taken for 5 iterations: 0.000139s
Average time for 1 iteration: 0.000028s
```

Findings

For practical reasons, each of the variants is run for 5 iterations only as each execution of the non-memoised variant takes about 8-9 minutes.

Putting the memoised version aside for now (since it is not part of the challenge but more of my own experiment), we can see from the sample output above that the variant that does not use `%rbp` runs about 20 seconds faster than the variant that uses `%rbp`.

More Instructions

One explanation for this finding is because of the the exponential number of instructions required if we were to use `%rbp`. For instance, let us take `vcn(4)` as calculating the 4th VCN. Then, we can observe that:

`vcn(4)` creates [4] stack frames, for `vcn(3)` [1], `vcn(2)` [1], `vcn(1)` [1] and `vcn(0)` [1].

`vcn(5)` creates [7] stack frames, for `vcn(4)` [4], `vcn(3)` [1], `vcn(2)` [1] and `vcn(1)` [1].

`vcn(6)` creates [13] stack frames, for `vcn(5)` [7], `vcn(4)` [4], `vcn(3)` [1] and `vcn(2)` [1].

Square brackets [ ] are used to denote the number of stack frames.

We can then formulate a formula to calculate the number of stack frames created for `vcn(n)`:

$$4 + 3 \times (2^{n-4} - 1),\ n \geq 4$$

Therefore, we can see that for `vcn(41)`, there will be:

$$4 + 3 \times (2^{41-4} - 1) = 412,316,860,417$$

stack frames being created.

As a result, additional number of instructions in multiples of 412316860417 would be required to be executed, which is not a small number and results in a longer time taken for the execution to complete. Such additional instructions include `pushq %rbp`, `popq %rbp` and `movq %rsp, %rbp`.

More Stack Memory

Moreover, for every function call in the recursive execution, 8 additional bytes (as written for 64-bit architecture) would be required to store the old base pointer onto the stack so that we can restore it after. Since we are in deep recursion as shown in the section above, this can have a significant impact on the stack memory.

Conclusion

Not using the base pointer and solely relying on the stack pointer indeed **boosts the performance** of deep recursive program like this one. This is mainly because of the number of stack frames being created during the execution.

Just some closing words (and also for myself to look back in the future). This has been an interesting challenge as it was my first time writing some `x86-64` code. Met with a lot of frustrations due to segfaults and obviously wrong results along the way. Also learnt about the importance of using a debugger, especially so when bugs are not so obvious in "less readable" code like assembly.