INF-2200 Computer architecture and organization

Assignment №1

Report

1. Introduction.

In this assignment, my task was to implement a micro-benchmark, identify hotspots, using profiler and reimplement the hotspots part into x86 assembly.

2. Technical background.

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1) Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).

2) Initially, let m equal 2, the smallest prime number.

3) Enumerate the multiples of m by counting to n from 2m in increments of m, and mark them in the list (these will be 2m, 3m, 4m, ... ; the m itself should not be marked).

4) Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

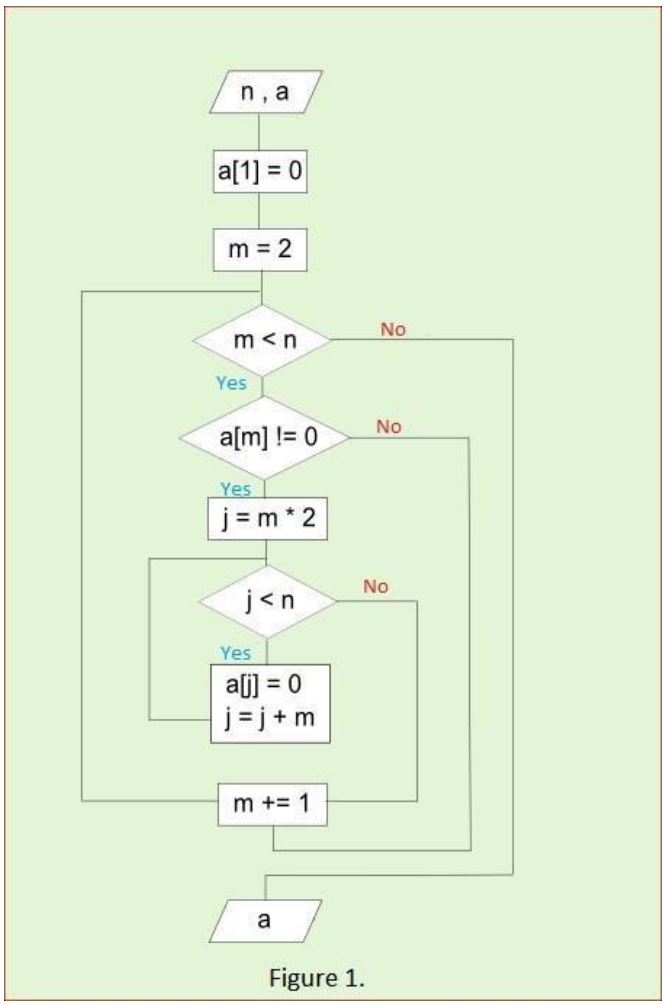

Figure 1.

3. Design.

In the beginning, I create two arrays with 100.000.000 elements in each. Using C language, I implement Eratosthenes sieve algorithm. With the help of gprof profiler, we can notice that there are two functions: main (here I create two arrays) and c_function (Eratosthenes sieve implementation). C_function uses around 90% of the total running time of the program, while main function only 10% (Figure 1.).

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 91.88      3.28     3.28        1     3.28     3.28  c_function
  8.12      3.57     0.29                             main
```

.

Figure 1.

I repeat profiling several times, but the result is almost the same. After that, I change the size of my arrays and I can see, that self time of the main function stay with the same value, while c_function's self time begins change. Therefore, I understand that c_function is a hotspot.

So, the next step is to reimplement c_function into asm_function. The result is natural (Figure 2).

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 57.38      3.42     3.42        1     3.42     3.42  c_function
 34.90      5.50     2.08                             for_loop
  4.87      5.79     0.29                             main
  2.10      5.92     0.12                             start_loop
  0.50      5.95     0.03                             continue
  0.25      5.96     0.01                             asm_function
```

Figure 2.

We can see, that the whole asm_function (which includes for_loop, start_loop, continue) works much faster, than c_function.

4. Implementation.

As a profiler I use gprof: in Makefile we add a -pg flag after -m32 and -masm=att cflags. After compilation, we get gmon.out file. Using gprof command we transfer this file into some text file. Open it and see the results of time running.

5. Discussion.

The code does not have any mistakes or bugs. It is readable and good-structured, so the user could understand what I want to write in the code. To optimize the program, I implement the code with different optimization levels (Figure 3).

Flat profile:

Optimization O0

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
56.72	3.29	3.29	1	3.29	3.29	c_function
35.86	5.37	2.08				for_loop
5.00	5.66	0.29				main
1.81	5.76	0.10				start_loop
0.52	5.79	0.03				continue
0.09	5.80	0.01				asm_function

Flat profile:

Optimization O1

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
49.13	2.27	2.27	1	2.27	2.27	c_function
44.59	4.33	2.06				for_loop
3.03	4.47	0.14				main
1.84	4.55	0.09				start_loop
0.97	4.60	0.04				continue
0.43	4.62	0.02				asm_function

Flat profile:

Optimization O2

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
92.27	2.03	2.03				for_loop
4.55	2.13	0.10				start_loop
3.18	2.20	0.07				continue

As we can see, the higher optimization level gives faster execution speed. Starting from the -O2 level and higher, a program can become unstable and lead to crash. However, our program does not require so much optimization. And it also important to notice, that optimization does not influence on the assembly part.

6. Conclusion.

While I was writing two functions for C and Assembly, I decided that C code is more logical, while assembler is closer to the processor, as it gives us a chance to work with memory manually. It's necessary to know a memory structure and working of processor to get a succesfully work program in Assembly.

The disadvantage of high-level languages is a huge size of programs compared to programs in low-level. So basically, high-level languages are used for the development of computer software and devices that have a large amount of memory. Low-level languages are usually used for writing small software programs, device drivers, joint modules with non-standard equipment, programming of specialized microprocessors in which are the most important requirements of compactness, performance and direct access to hardware resources.