1 CPU Profiling

In the root directory you can find a directory called test_1. It contains test40, which is an electromagnetic test from the Geant 4 framework. You can run it in the following way:

```
make
./test40 < ./test40.in50
```

Valgrind is a framework for dynamic binary instrumentation. It provides several different tools for memory and CPU profiling. In this step, we want to use callgrind in order to obtain the Top 10 CPU hotspots. The beauty of callgrind is that its output format can be nicely visualized with kcachegrind. You can execute it with:

```
kcachegrind callgrind.out.27425
```

What are the top 10 most commonly called functions?

	function	percentage
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

2 Heap Profiling

In this section we will use `gperftools` in order to profile the previous binary in terms of memory. `gperftools` is installed under `/root/gperftools`. You can find `libtcmalloc` under `/root/gperftools/.libs` and `pprof` under `/root/gperftools/src/pprof`. Details about `gperftools` and how to use its heapchecker and heapprofiler, can be found here: `http://goog-perftools.sourceforge.net/doc/`
What are the top 10 memory consumers?

	function	size
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

What is the total size of virtual memory and Rss?

Are there any memory leaks? If so which one is the largest?

3 Code Optimization

The folllowing code examples are taken from the Intel Software Optimization Cookbook.

3.1 Loops

The directory test_2 contains a file named loop.c. This loop iterates over all the elements and sums them up:

```
for(i = 0; i < LENGTH; i++)
{
     a += buffer[i];
}
```

As we learned in the lecture a CPU is capable of executing several instructions per clock cycle. However, this loop has data dependencies which results in limited instruction level parallelism. Compile the program with `gcc loop.c -o loop`. What is the execution time?

How can you make this loop run faster?

What is the execution time of the optimized loop?

What is the execution time when you compile the loop with `-O3`? And what could be the reason for the improvement? Hint: You can go to `https://gcc.godbolt.org/` and copy/paste the code there. This website allows to see which parts in the assembly code change with different optimization flags.

What happens when you remove the print statement and recompile with `-O3`?

The directory test_3 contains another example of a for-loop. This time there is the following branch in the loop:

```
for(i = 0; i < LENGTH; i++)
```

```
{
     if (a%2 == 0)
        a += buffer[i];
     else
        a -= buffer[i+1];
}
```

Why are branches in a loop critical for performance?

What is the execution time?

How can you optimize it? And what is the improvement?

3.2 Cache usage

Directory test_4 contains a search.c with some code snippets. The following data structure is given:

```
typedef struct PhoneBookEntry{
   char Name[16];
   char Surname[16];
   char email[16];
   char phone[10];
   char mobilephone[10];
   char address1[16];
   char address2[16];
   char city[16];
   char state[2];
   char zip[5];
   struct PhoneBookEntry *pNext;
} PhoneBook;
```

The following search function is used for finding names in the phonebook:

```
PhoneBook * FindName(char* name, PhoneBook * pHead)
{
  while (pHead != NULL)
  {
     if strcmp(name, pHead->Lastname) == 0)
       return pHead;
     pHead = pHead->pNext;
  }
    return NULL;
}
```

The function makes very bad usage of the cache. The size of a cache line in L1 is 64-byte, in L2 is 128-byte. The total size of the structure is 131 Bytes. In each

iteration it uses only 24 Bytes out of the 64-Byte L1 cache line. How can you modify the datastructure and the search in order to increase cache usage? Your task will be to write a function in order to create a phonebook with random data, to measure the current search function and to improve the search and the data structure.

4 Debugging

Directory test_5 contains a buggy program compiled with debugging symbols. It is crashing with a segmentation fault. Can you find the reason for this crash with gdb? You can debug the program with

```
gdb ./buggy
(gdb) run
```

With `bt` you can print the entire stack, with `b` you can set breakpoints, with `n` and `s` you can go to the next line or step into functions.