Звіт до лабораторної роботи 6
з предмету “Системне програмування”
виконав студент 3 курсу факультету
комп’ютерних наук та кібернетики
групи МІ-31
Бідзіля Святослав Олексійович

Постановка задачі

Для обраної (скомпільованої) програми:

1) побудувати **FlameGraph** виконання:
1.1) **продемонструвати** процес **побудови** при захисті роботи (+3б.)
1.2) **пояснити** (інтерпретувати) отримані на графіку результати (+3б.)
(можна для самої програми або для системи в цілому)
(див.: https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html#Instructions)

2) **зібрати** та **пояснити** статистику її виконання, зокрема:
2.1) /usr/bin/time -- verbose <<prog>> (+2б.)
2.2) perf stat -d <<prog>> (+2б.)
2.3) perf report (after perf record) (+2б.)
(perf: створення і аналіз логів, траси виконання)
(див.: записи лекцій + https://www.brendangregg.com/perf.html#Examples)

3) заміряти **енерговитрати** (**Power consumption**):
3.1) системи при виконанні програми (+3б.)
3.2) виключно досліджуваної програми (+3б.)
(див.: https://luiscruz.github.io/2021/07/20/measuring-energy.html , але проявіть творчість!)

4) порівняти параметри виконання програми до та **після оптимізації** (або ключами -Ox, або внесенням змін у її вихідний код):
4.1) пояснити різницю в **асемблерному** коді до та після виконання **оптимізації** (+3б.)
4.1.1) пояснити на прикладі інструкцій **AVX**-* розширень (якщо застосовно), наприклад **SIMD** інструкції (+3б.)
(можна користуючись https://godbolt.org)
4.2) порівняти час та інші показники (див. п.п. 2.1, 2.2, тощо) виконання до і після оптимізації (+3б.)
4.3) продемонструвати зміни на FlameGraph (п. 1) після оптимізації (+3б.)
4.4) порівняти енерговитрати (п. 3) після оптимізації (+3б.)
(див.: записи лекцій, матеріали з попередніх пунктів, https://godbolt.org/)

Реалізація

Код реалізовано як репозиторій (https://github.com/anakib1/KNU_SP_lab6), який складається з таких частин:
bin - віхідні файли
codes - коді програм для дослідження
out - скомпільовані програми з codes
scripts - скрипти для реалізації задач ЛР.

Flamegraph

Для побудови flamegraph може бути використано скрипт

flame.sh program-name

Вміст скрипта:

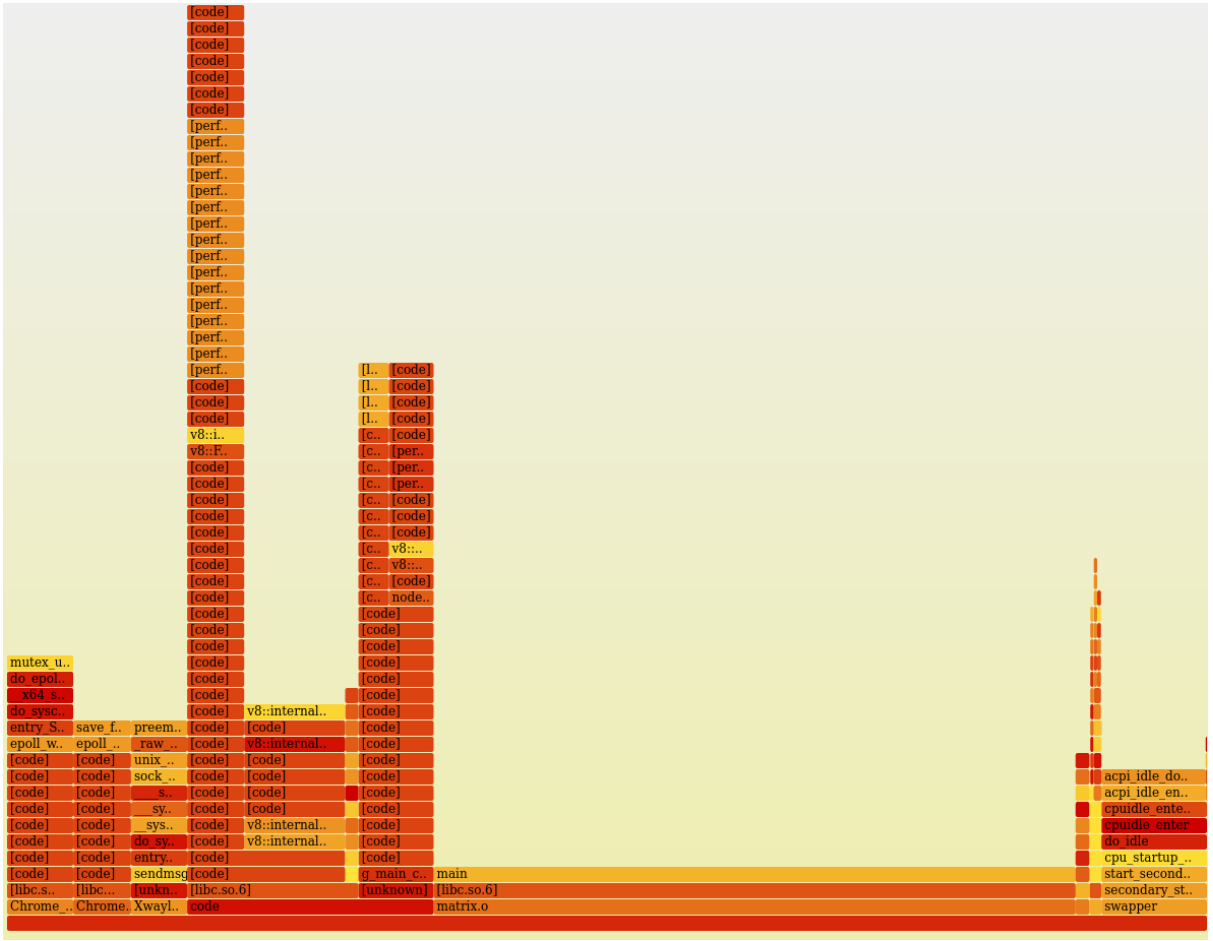
```
#!/bin/bash

name="$1"
perf record -F 99 -a -g "out/${name}.o"
perf script | FlameGraph/stackcollapse-perf.pl > bin/out.perf-folded
FlameGraph/flamegraph.pl bin/out.perf-folded > bin/perf.svg
```

Як ми бачимо, він просто рекордить виконання потрібного файлу, записує результат та викликає флеймграф від результату.

Приклад:

```
sviatoslav@debian:~/Documents/lab6$ sudo scripts/flame.sh matrix
-1065635116
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.415 MB perf.data (101 samples) ]
sviatoslav@debian:~/Documents/lab6$
```

Цей скрипт створює файл perf.svg в теці bin. Відкривши його в браузері, бачимо такий інтерфейс



Дослідження графіку:

Сконцентруємося на основних рисах графіку.
Кольори
 червоний - юзер левел викликли.
 зелений (не присутній) - джава
 оранжевий - кернел виклики
 жовтий - c++ виклики, прямі функції.
Важливо зазначити, що флеймграф не є часовим виконанням коду - х-вісь не є віссю часу, це є просто відсортованим за алфавітом елементами стеку. Все в у-вісі просто знаходится в порядку виклику стеку і відрізняється за кольорами.

Статистика виконання програми

Для прикладу, розглянемо програму, яка виконує множення матриць: matrix.cpp

```cpp
#include <iostream>
#include <random>


const int n = 500, m = 500, l = 500;
int MX = 1000000;
int m1[n][m];
int m2[m][l];

int res[n][l];

int main() {

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            m1[i][j] = random() % MX;
        }
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < l; j++) {
            m2[i][j] = random() % MX;
        }
    }

    for (int i = 0; i < n; i++){
        for (int j = 0; j < m; j++) {
            for (int k = 0; k < l; k++) {
                res[i][k] += m1[i][j] * m2[j][k];
            }
        }
    }

    int sum = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < l; j++) {
            sum += res[i][j];
        }
```

```
    }

    std::cout << sum << '\n';
}
```

Скомпілювавши програму в matrix.o за допомогою скрипта compile.sh ми можемо виміряти час виконання по-перше за допомогою утиліти time

```
bash: compile.sh: command not found
sviatoslav@debian:~/Documents/lab6$ scripts/compile.sh matrix
sviatoslav@debian:~/Documents/lab6$ time out/matrix.o
-1065635116

real    0m0.475s
user    0m0.467s
sys     0m0.008s
sviatoslav@debian:~/Documents/lab6$
```

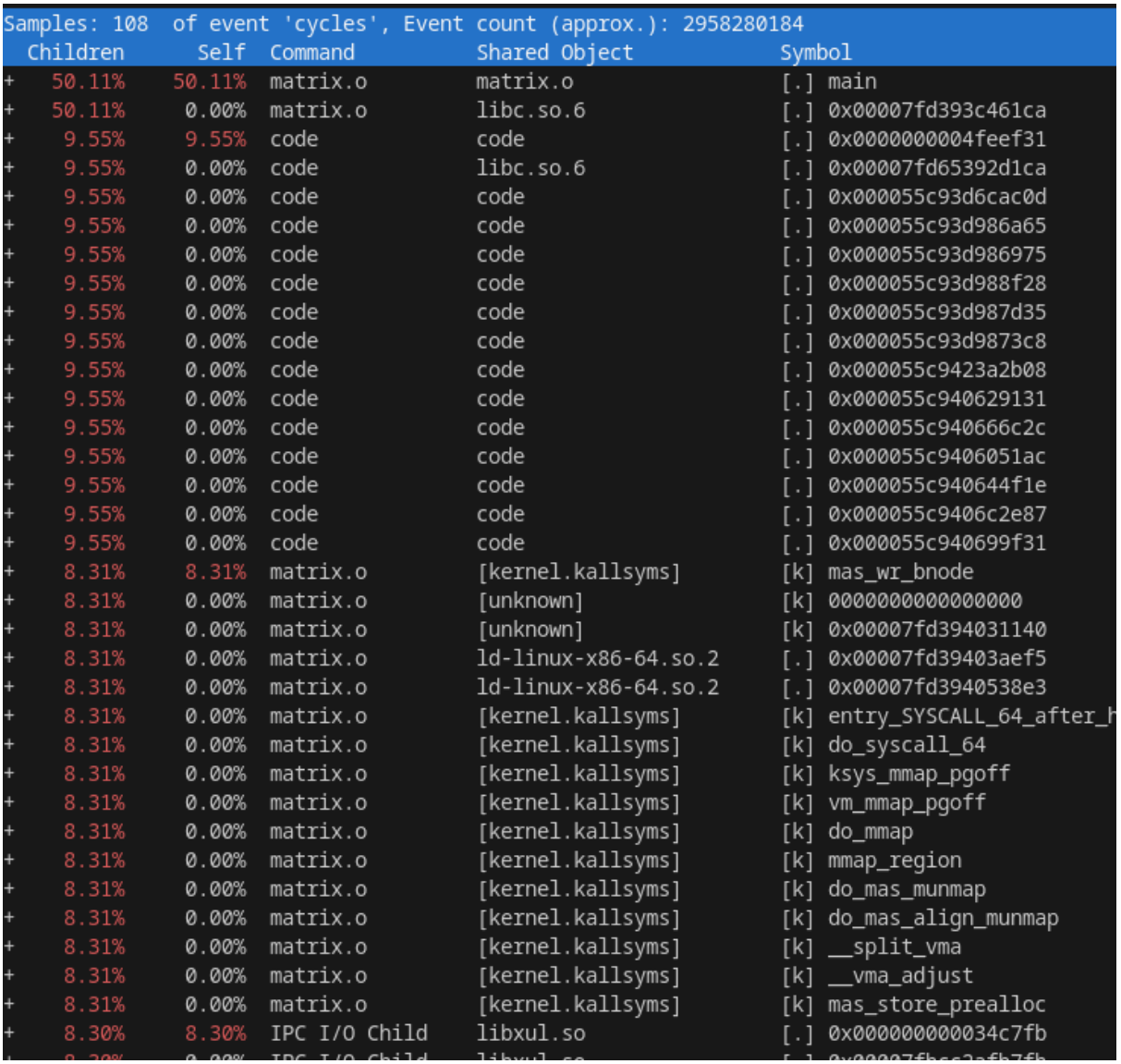
Також можемо виміряти більш точні і важливі статистики запустивши sudo perf stat -d out/matrix.o:

```
sviatoslav@debian:~/Documents/lab6$ sudo perf stat -d out/matrix.o
[sudo] password for sviatoslav:
-1065635116

 Performance counter stats for 'out/matrix.o':

            504.44 msec task-clock                       #    0.998 CPUs utilized
                 2      context-switches                 #    3.965 /sec
                 0      cpu-migrations                   #    0.000 /sec
             1,101      page-faults                      #    2.183 K/sec
     1,670,211,646      cycles                           #    3.311 GHz                      (61.95%)
         1,617,486      stalled-cycles-frontend          #    0.10% frontend cycles idle     (61.94%)
     1,146,119,857      stalled-cycles-backend           #   68.62% backend cycles idle      (61.94%)
     5,171,788,813      instructions                     #    3.10  insn per cycle
                                                  #    0.22  stalled cycles per insn  (62.34%)
       136,649,976      branches                         #  270.897 M/sec                    (63.13%)
           290,700      branch-misses                    #    0.21% of all branches          (63.42%)
     1,782,423,824      L1-dcache-loads                  #    3.534 G/sec                    (63.03%)
         8,184,358      L1-dcache-load-misses            #    0.46% of all L1-dcache accesses  (62.24%)
    <not supported>      LLC-loads
    <not supported>      LLC-load-misses

       0.505247943 seconds time elapsed

       0.501335000 seconds user
       0.004010000 seconds sys
```

Для того, щоб проаналізувати стек виконання ще краще, можемо викликати sudo stat_report.sh matrix. Цей файл викликає вбудований в перф report:

```
#!/bin/bash

name="$1"
perf record -F 99 -a -g --output="bin/perf.data" "out/${name}.o"
perf report -i bin/perf.data
```

Приклад виклику:



```
Samples: 108  of event 'cycles', Event count (approx.): 2958280184
  Children      Self  Command          Shared Object            Symbol
+   50.11%    50.11%  matrix.o         matrix.o                 [.] main
+   50.11%     0.00%  matrix.o         libc.so.6                [.] 0x00007fd393c461ca
+    9.55%     9.55%  code             code                     [.] 0x0000000004feef31
+    9.55%     0.00%  code             libc.so.6                [.] 0x00007fd65392d1ca
+    9.55%     0.00%  code             code                     [.] 0x000055c93d6cac0d
+    9.55%     0.00%  code             code                     [.] 0x000055c93d986a65
+    9.55%     0.00%  code             code                     [.] 0x000055c93d986975
+    9.55%     0.00%  code             code                     [.] 0x000055c93d988f28
+    9.55%     0.00%  code             code                     [.] 0x000055c93d987d35
+    9.55%     0.00%  code             code                     [.] 0x000055c93d9873c8
+    9.55%     0.00%  code             code                     [.] 0x000055c9423a2b08
+    9.55%     0.00%  code             code                     [.] 0x000055c940629131
+    9.55%     0.00%  code             code                     [.] 0x000055c940666c2c
+    9.55%     0.00%  code             code                     [.] 0x000055c9406051ac
+    9.55%     0.00%  code             code                     [.] 0x000055c940644f1e
+    9.55%     0.00%  code             code                     [.] 0x000055c9406c2e87
+    9.55%     0.00%  code             code                     [.] 0x000055c940699f31
+    8.31%     8.31%  matrix.o         [kernel.kallsyms]        [k] mas_wr_bnode
+    8.31%     0.00%  matrix.o         [unknown]                [k] 0000000000000000
+    8.31%     0.00%  matrix.o         [unknown]                [k] 0x00007fd394031140
+    8.31%     0.00%  matrix.o         ld-linux-x86-64.so.2     [.] 0x00007fd39403aef5
+    8.31%     0.00%  matrix.o         ld-linux-x86-64.so.2     [.] 0x00007fd3940538e3
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] entry_SYSCALL_64_after_h
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] do_syscall_64
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] ksys_mmap_pgoff
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] vm_mmap_pgoff
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] do_mmap
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] mmap_region
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] do_mas_munmap
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] do_mas_align_munmap
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] __split_vma
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] __vma_adjust
+    8.31%     0.00%  matrix.o         [kernel.kallsyms]        [k] mas_store_prealloc
+    8.30%     8.30%  IPC I/O Child    libxul.so                [.] 0x000000000034c7fb
```

Power consumption

Для розробки аналізу використання потужності програмою було використано функцію powertools, яка присутня на MacOS з APM-процесорами. Було написано простий парсер виводу, який агрегує сумарне використання потужності.

Всі дані знаходяться тут

https://github.com/anakib1/MangoPower

Приклад з запуском програми на множення рандомних матриць 1000x1000 100 разів

```
sviatoslavbidzilia@Sviatoslavs-Laptop MangoPower % sudo ./run.sh
16299 mW

15173 mW

14521 mW

16176 mW

18123 mW

15617 mW

16277 mW

16094 mW

13490 mW

14612 mW

Totally consumed 156382 mW
```

Приклад виводу чистої системи

```
sviatoslavbidzilia@Sviatoslavs-Laptop MangoPower % sudo ./run.sh
263 mW

103 mW

248 mW

4553 mW

524 mW

156 mW

3476 mW

172 mW

49 mW

127 mW

Totally consumed 9671 mW
```

Ми явно бачимо різницю, яка і є різницею в споживанні.

Виконання оптимізацій

Для аналізу виконання оптимізацій нам знадобиться більш проста програма - demo.cpp

```cpp
#include <iostream>

int main() {

    int n = 10;
    int m = 20;
    int k = 100000000;
    int s = 0;
    for (int i = 0; i < k; i ++ ) {
        s += (n * n * n + m * m + 188);
    }

    std::cout << s << '\n';
}
```

Як ми бачимо, ця програма має виконати послідовність арифметичних обчислень. Без жодної оптимізації, код на асемблері виглядатиме так:

```
1   main:
2           push    rbp
3           mov     rbp, rsp
4           sub     rsp, 32
5           mov     DWORD PTR [rbp-12], 10
6           mov     DWORD PTR [rbp-16], 20
7           mov     DWORD PTR [rbp-20], 100000000
8           mov     DWORD PTR [rbp-4], 0
9           mov     DWORD PTR [rbp-8], 0
10          jmp     .L2
11  .L3:
12          mov     eax, DWORD PTR [rbp-12]
13          imul    eax, eax
14          imul    eax, DWORD PTR [rbp-12]
15          mov     edx, eax
16          mov     eax, DWORD PTR [rbp-16]
17          imul    eax, eax
18          add     eax, edx
19          add     eax, 188
20          add     DWORD PTR [rbp-4], eax
21          add     DWORD PTR [rbp-8], 1
22  .L2:
23          mov     eax, DWORD PTR [rbp-8]
24          cmp     eax, DWORD PTR [rbp-20]
25          jl      .L3
26          mov     eax, DWORD PTR [rbp-4]
27          mov     esi, eax
28          mov     edi, OFFSET FLAT:_ZSt4cout
29          call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
30          mov     esi, 10
31          mov     rdi, rax
32          call    std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_t
33          mov     eax, 0
34          leave
35          ret
```

Як ми бачимо, він просто чесно виконує операції програми, послідовно рахуючи відповідь.
додамо флаг -O1 до компілятора. Тепер ми маємо:

```
main:
        sub     rsp, 8
        mov     eax, 100000000
.L2:
        sub     eax, 1
        jne     .L2
        mov     esi, -113789952
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
        mov     rdi, rax
        mov     esi, 10
        call    std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_t
        mov     eax, 0
        add     rsp, 8
        ret
```

Як ми бачимо, тепер, весь код виконується в одну дію, оскільки програма детермінована і компілятор предрахував усі математичні вирази в ній.
Замірявши час виконання програми до та після маємо:
До

sviatoslav@debian:~/Documents/lab6$ time out/demo.o
-113789952

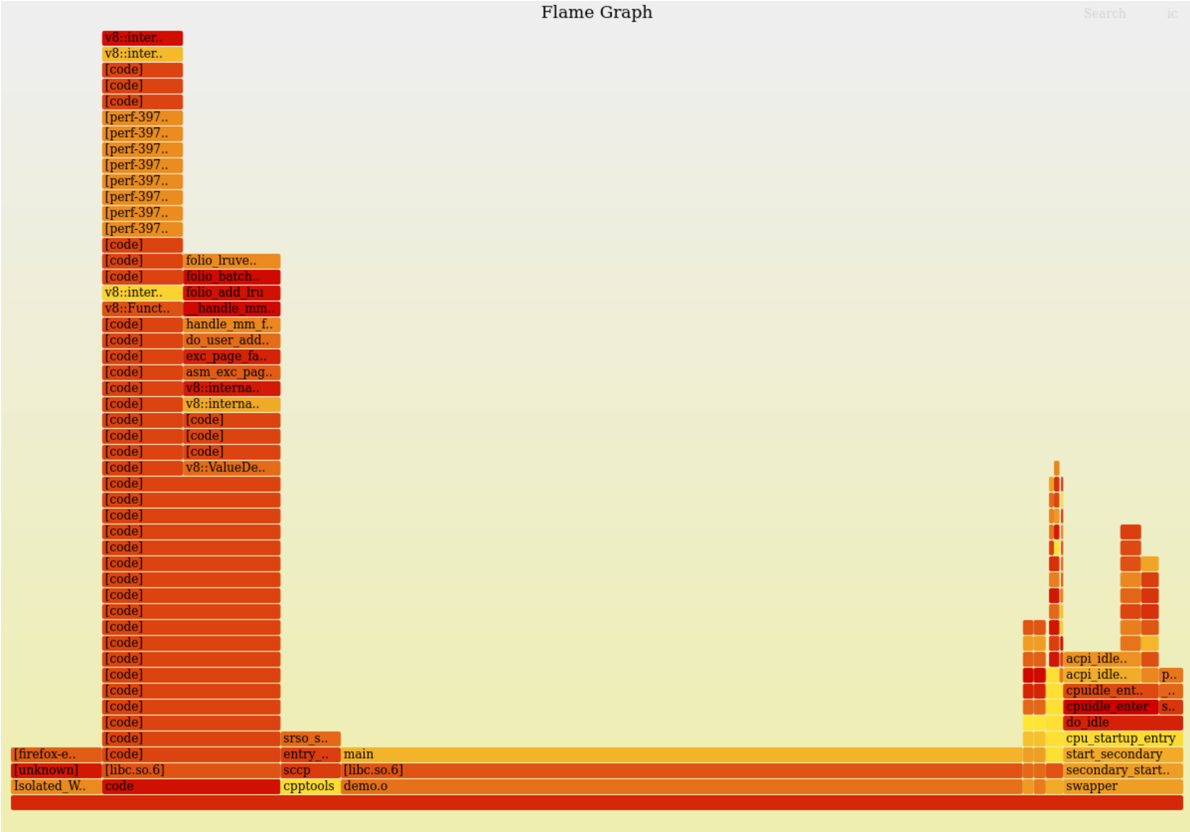
real 0m0.252s
user 0m0.247s
sys 0m0.005s

Після

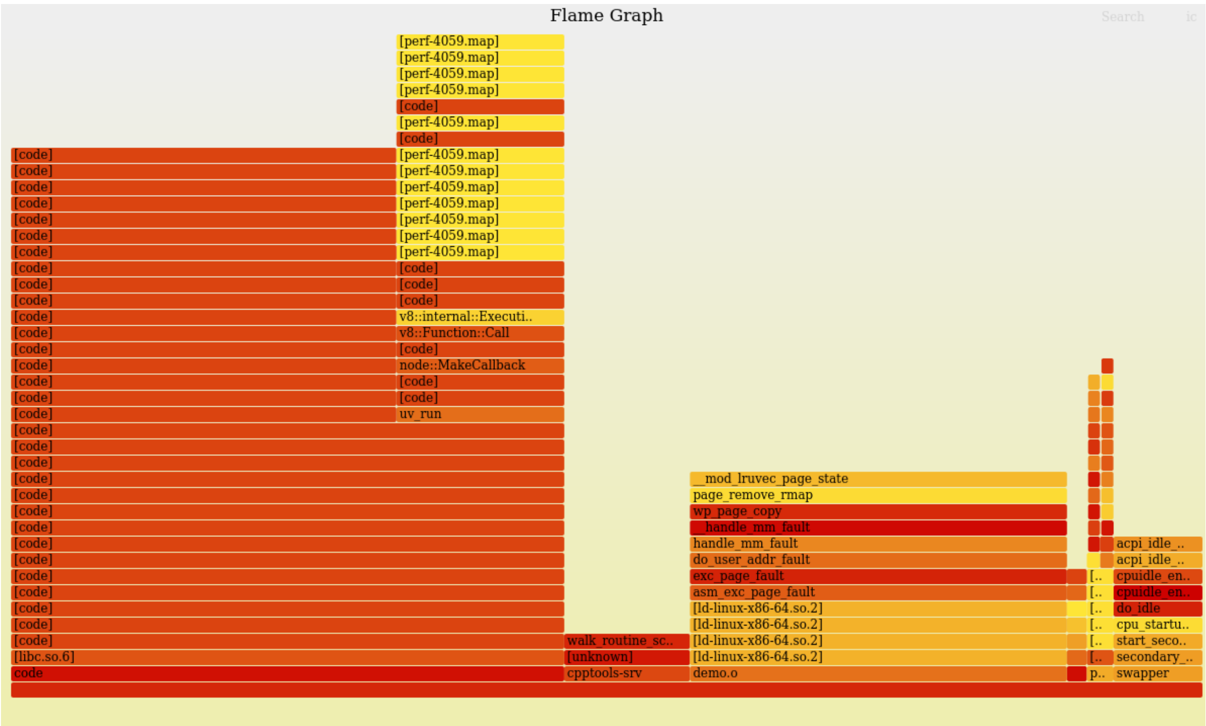
sviatoslav@debian:~/Documents/lab6$ time out/demo.o
-113789952

real 0m0.043s
user 0m0.042s
sys 0m0.002s

Як ми бачимо, виконання пришвидшилося на декілька порядків. Подивимося на флеймграф:
До:



Після:



Як ми бачимо, з флеймграфу повністю зникло виконання нашої програми, оскільки тепер, вона виконується в одну дію і не займає час.