IIT DELHI

COL380: Introduction to Parallel and Distributed Programming

Assignment-0 Report

Prof. Subodh Kumar

ARYAN SANTOSH SAHOO

2019CS10335

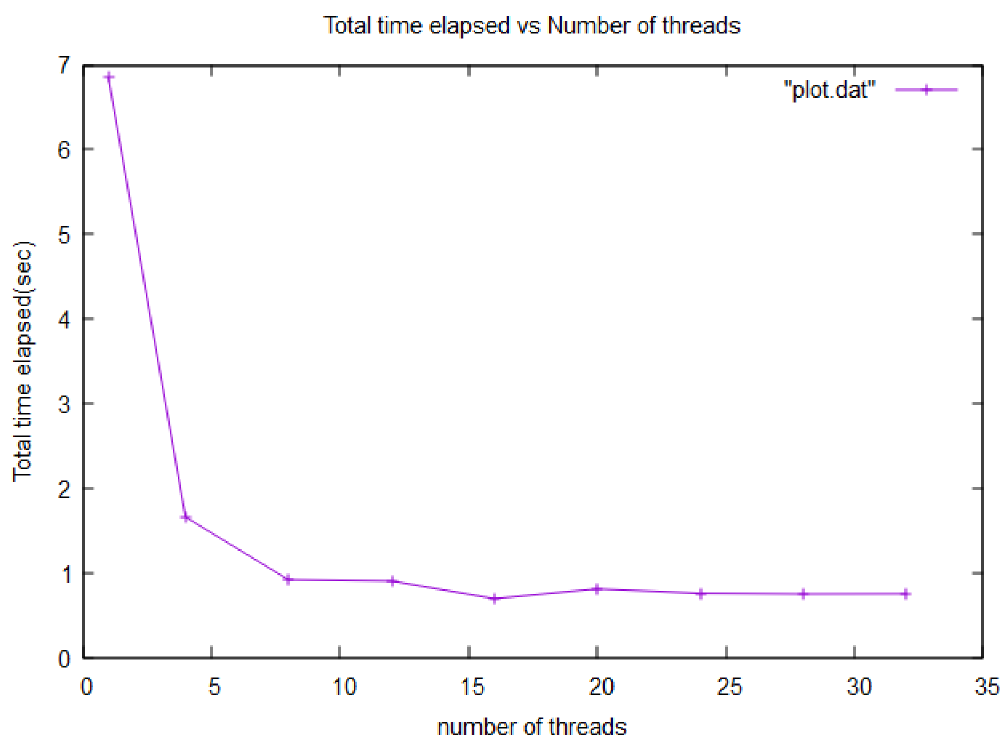
2.1 Perf Stat

Number of lines read: 1009072

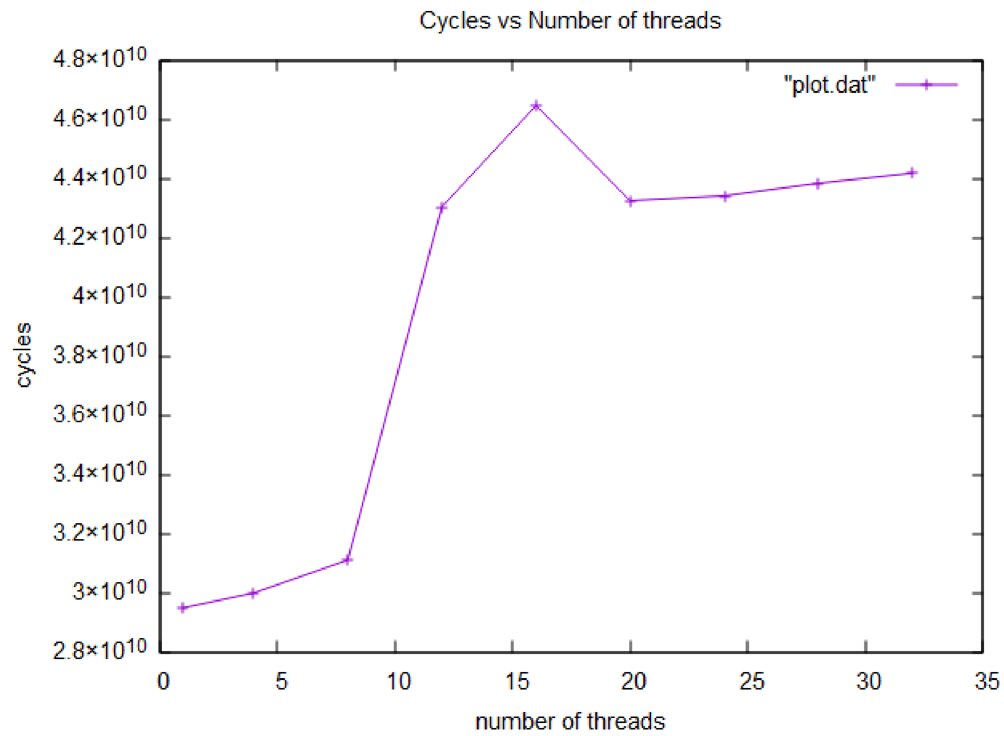
Number of runs: 6

Number of threads	Time elapsed (sec)	Cycles
1	6.853918571	29505554053
4	1.664863805	30006091171
8	0.923874249	31119619758
12	0.909170374	43036352278
16	0.701718816	46460936690
20	0.815588766	43250471275
24	0.764130756	43409226990
28	0.756588426	43839599972
32	0.758183896	44175903686

Time elapsed vs number of threads:



Cycles vs number of threads:



Perf stats for threads = 1

```
Performance counter stats for './classify rfile dfile 1009072 1':

        6,311.13 msec task-clock                #    0.921 CPUs utilized
             158      context-switches          #   25.035 /sec
               8      cpu-migrations            #    1.268 /sec
          15,347      page-faults               #    2.432 K/sec
  29,50,55,54,053      cycles                   #    4.675 GHz
  64,86,21,58,042      instructions             #    2.20  insn per cycle
  23,00,25,45,350      branches                 #    3.645 G/sec
     61,36,05,265      branch-misses            #    2.67% of all branches
1,47,52,46,14,665      slots                    #   23.375 G/sec
  40,87,82,89,045      topdown-retiring         #     26.4% retiring
  65,95,21,80,673      topdown-bad-spec         #     42.7% bad speculation
  43,84,91,98,757      topdown-fe-bound         #     28.4% frontend bound
   3,91,09,64,866      topdown-be-bound         #      2.5% backend bound

      6.853918571 seconds time elapsed

      6.304360000 seconds user
      0.007975000 seconds sys


cs11903350css7:~/Desktop/A0$
```

Analysis:

The above screenshot shows the perf stat command run on one thread to give the estimate of key CPU parameters like cycles, total time elapsed, branches and branch-misses.

As expected we see sharp decrease in the time elapsed as the number of threads increases but then it becomes almost constant as the number of threads increase above 8.

The reason for that being that each thread gets allotted different tasks and they together complete it in less time. But using too many threads can hurt performance as it adds overhead in scheduling. The code here in this case is not optimised and isn’t very CPU extensive hence we see a usual decrease at first in the time elapsed and then a stagnation.

Branches misses also follows the same trend as increase in the number of threads means less misses as each thread has to take care of a smaller task now.

The number of cycles increases with number of threads because the context switching increases.

2.2 Perf Record

Perf record was run and a *“perf.data”* file was generated.

Perf report was run to inspect the file and then it was annotated for further analysis.

This *“perf.data”* was renamed to *“perf_1.data”.*

By seeing the annotated assembly code, we can point out the part of the code which is the most CPU time extensive.

```
 38.03 |        ┌──jg      93
  0.35 |        │  shl     $0x6,%rax
  0.06 |        │  add     %rbp,%rax
  0.06 |  4e:   │  mov     %r13d,0x4(%r12)
  0.38 |        │  mov     (%rax),%rdx
  0.19 |        │  cmp     %r9d,0x8(%rax)
       |        │↓ jbe     b9
       |        │  lea     (%rdx,%rdi,1),%rax
       |        │  add     %ebx,%ecx
  2.50 |        │  mov     (%rax),%edx
  0.04 |        │  add     $0x1,%edx
  0.08 |        │  mov     %edx,(%rax)
  0.02 |        │  mov     %ecx,%eax
       |        │  cmp     %ecx,(%r8)
       |        │↓ jbe     b0
  0.00 |  70:   │  cltq
  0.02 |        │  lea     (%r10,%rax,8),%r12
       |        │  mov     0x8(%rsi),%eax
  0.03 |        │  mov     (%r12),%edx
  0.00 |        │  test    %eax,%eax
       |        │↓ jle     a8
  0.01 |        │  mov     (%rsi),%r11
       |        │  lea     -0x1(%rax),%r14d
       |        │  xor     %eax,%eax
  0.00 |  8a:   │  mov     %eax,%r13d
  9.26 |        │  cmp     (%r11,%rax,8),%edx
 14.83 |        │↑ jge     40
 17.94 |  93:   └→ lea     0x1(%rax),%r13
  1.02 |           cmp     %rax,%r14
       |         ↓ je      a8
       |           mov     %r13,%rax
 14.86 |         ↑ jmp     8a
```

Assembly instruction:

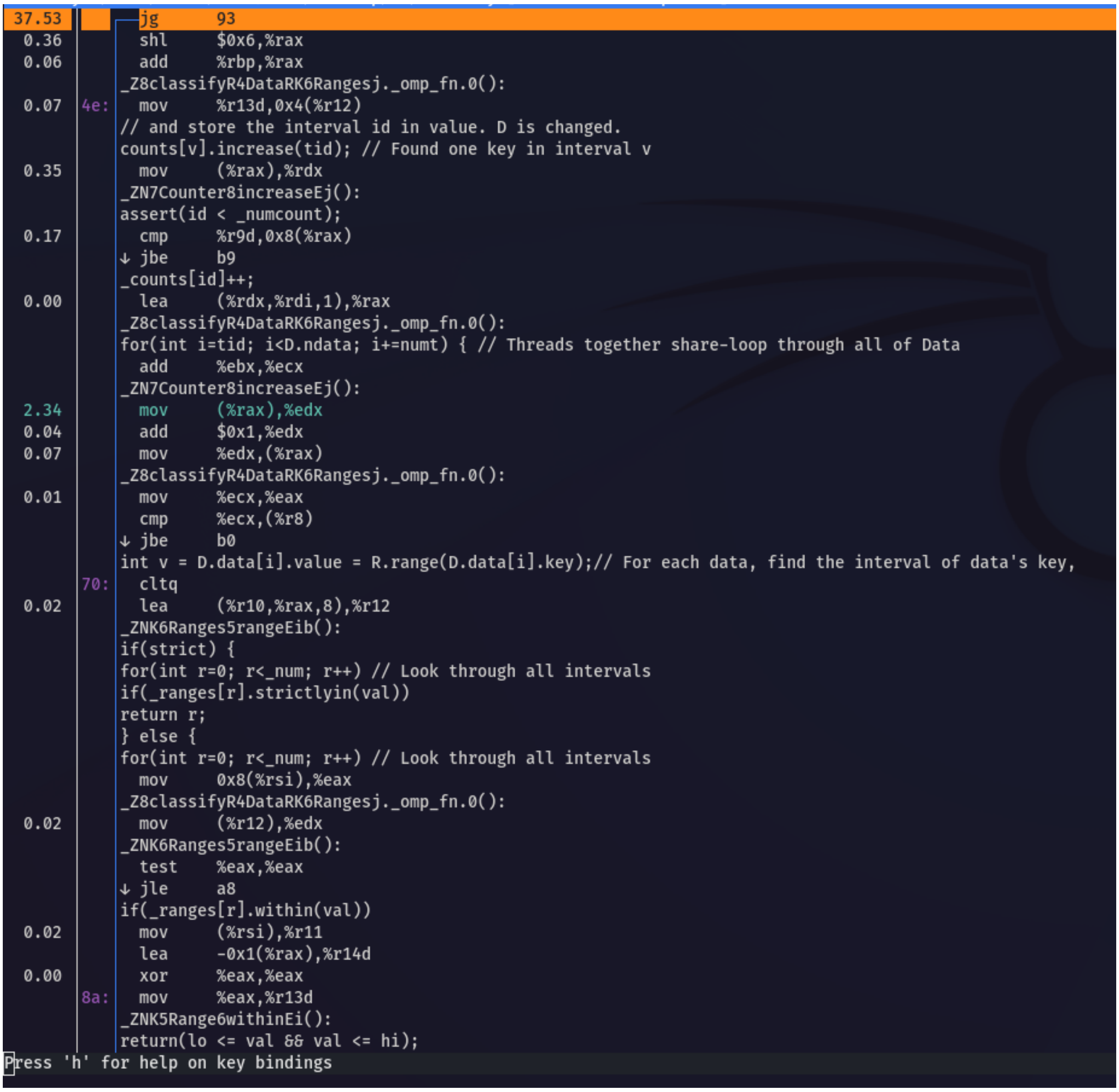
jg 93

(38.03% CPU time)

That line in the assembly code maps to the bool in *“classify.h”*

To get further clarity we change the *“makefile”* so that now it shows both the source code and assembly code, we do this by adding a “*-g*” flag.

This *“perf.data”* file was renamed to *“perf_2_1.data”*



```
37.53      jg      93
 0.36      shl     $0x6,%rax
 0.06      add     %rbp,%rax
         _Z8classifyR4DataRK6Rangesj._omp_fn.0():
 0.07 4e:  mov     %r13d,0x4(%r12)
         // and store the interval id in value. D is changed.
         counts[v].increase(tid); // Found one key in interval v
 0.35      mov     (%rax),%rdx
         _ZN7Counter8increaseEj():
         assert(id < _numcount);
 0.17      cmp     %r9d,0x8(%rax)
         ↓ jbe     b9
         _counts[id]++;
 0.00      lea     (%rdx,%rdi,1),%rax
         _Z8classifyR4DataRK6Rangesj._omp_fn.0():
         for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
           add     %ebx,%ecx
         _ZN7Counter8increaseEj():
 2.34      mov     (%rax),%edx
 0.04      add     $0x1,%edx
 0.07      mov     %edx,(%rax)
         _Z8classifyR4DataRK6Rangesj._omp_fn.0():
 0.01      mov     %ecx,%eax
           cmp     %ecx,(%r8)
         ↓ jbe     b0
         int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
      70:  cltq
 0.02      lea     (%r10,%rax,8),%r12
         _ZNK6Ranges5rangeEib():
         if(strict) {
         for(int r=0; r<_num; r++) // Look through all intervals
         if(_ranges[r].strictlyin(val))
         return r;
         } else {
         for(int r=0; r<_num; r++) // Look through all intervals
           mov     0x8(%rsi),%eax
         _Z8classifyR4DataRK6Rangesj._omp_fn.0():
 0.02      mov     (%r12),%edx
         _ZNK6Ranges5rangeEib():
           test    %eax,%eax
         ↓ jle     a8
         if(_ranges[r].within(val))
 0.02      mov     (%rsi),%r11
           lea     -0x1(%rax),%r14d
 0.00      xor     %eax,%eax
      8a:  mov     %eax,%r13d
         _ZNK5Range6withinEi():
         return(lo <= val && val <= hi);
Press 'h' for help on key bindings
```

3. Hotspot Analysis

As stated above we had renamed the generated *“perf.data”* file to *“perf_2_1.data”* after changing the makefile to show the source code along with the assembly code.

Now we analyse the top hotspot in the code and see if it can be optimised to reduce runtime.

```
              _ZNK5Range6withinEi():
              lo = a;
              hi = b;
              }

              bool within(int val) const { // Return if val is within this range
              return(lo <= val && val <= hi);
 0.35   40:     cmp     0x4(%r11,%rax,8),%edx
37.53           jg      93
 0.36           shl     $0x6,%rax
 0.06           add     %rbp,%rax
              _Z8classifyR4DataRK6Rangesj._omp_fn.0():
 0.07   4e:     mov     %r13d,0x4(%r12)
              // and store the interval id in value. D is changed.
              counts[v].increase(tid); // Found one key in interval v
 0.35           mov     (%rax),%rdx
              _ZN7Counter8increaseEj():
              assert(id < _numcount);
 0.17           cmp     %r9d,0x8(%rax)
```

```
 0.02           mov     (%rsi),%r11
                lea     -0x1(%rax),%r14d
 0.00           xor     %eax,%eax
        8a:     mov     %eax,%r13d
              _ZNK5Range6withinEi():
              return(lo <= val && val <= hi);
 9.50           cmp     (%r11,%rax,8),%edx
14.76         ↑ jge     40
              _ZNK6Ranges5rangeEib():
              for(int r=0; r<_num; r++) // Look through all intervals
18.19   93:     lea     0x1(%rax),%r13
 0.99           cmp     %rax,%r14
              ↓ je      a8
 0.00           mov     %r13,%rax
15.11         ↑ jmp     8a
                nop
        a8:     mov     %rbp,%rax
              return r;
              }
```

We can see that the highlighted part of the assembly code takes the most CPU time.

Upon closer analysis we find out that it occurs because of a call from *“classify.cpp”* to *“classify.h”* where a certain variable is searched throughout the entire range using Boolean conditions, it is very inefficient and the performance can be improved by hashing or sorting through the range and searching by index.

For the final part to get an estimate of the branches, branch misses, cache misses, page faults and CPU cycles the following perf record command was run in the terminal:

perf record -e branch-instructions,branch-misses,cache-misses,page-faults,cpu-cycles make run

The generated *“perf.data”* file was then renamed to *“perf_2_2.data”*

4. Memory Profiling

Perf mem record was run and the data was stored into *“perf_3.data”*

This file contains data on the memory resources used by the code.

Two hotspots were identified:

```
       _ZN7Counter8increaseEj():
       assert(id < _numcount);
 1.20    cmp    %r9d,0x8(%rax)
       ↓ jbe    b9
       _counts[id]++;
         lea    (%rdx,%rdi,1),%rax
       _Z8classifyR4DataRK6Rangesj._omp_fn.0():
       for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
         add    %ebx,%ecx
       _ZN7Counter8increaseEj():
91.71    mov    (%rax),%edx
         add    $0x1,%edx
         mov    %edx,(%rax)
       _Z8classifyR4DataRK6Rangesj._omp_fn.0():
         mov    %ecx,%eax
 0.03    cmp    %ecx,(%r8)
       ↓ jbe    b0
       int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
  70:    cltq
         lea    (%r10,%rax,8),%r12
```

```
         xor    %esi,%esi
       D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
         lea    -0x4(%r14,%rdx,4),%r11
         lea    0x8(%rcx),%rdx
         lea    (%rdx,%r12,1),%r8
       ↓ jmp    64
         nop
 0.09  60:  add    $0x8,%rdx
       if(D.data[d].value == r) // If the data item is in this interval
 0.07  64:  cmp    %eax,0x4(%rcx)
96.85       jne    81
       D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
 0.59       mov    0x18(%rbx),%r9
            mov    (%rcx),%rcx
            mov    %esi,%r10d
            add    $0x1,%esi
 0.49       add    (%r11),%r10d
            mov    0x8(%r9),%r9
            mov    %rcx,(%r9,%r10,8)
       for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
 1.60  81:  mov    %rdx,%rcx
            cmp    %rdx,%r8
          ↑ jne    60
```

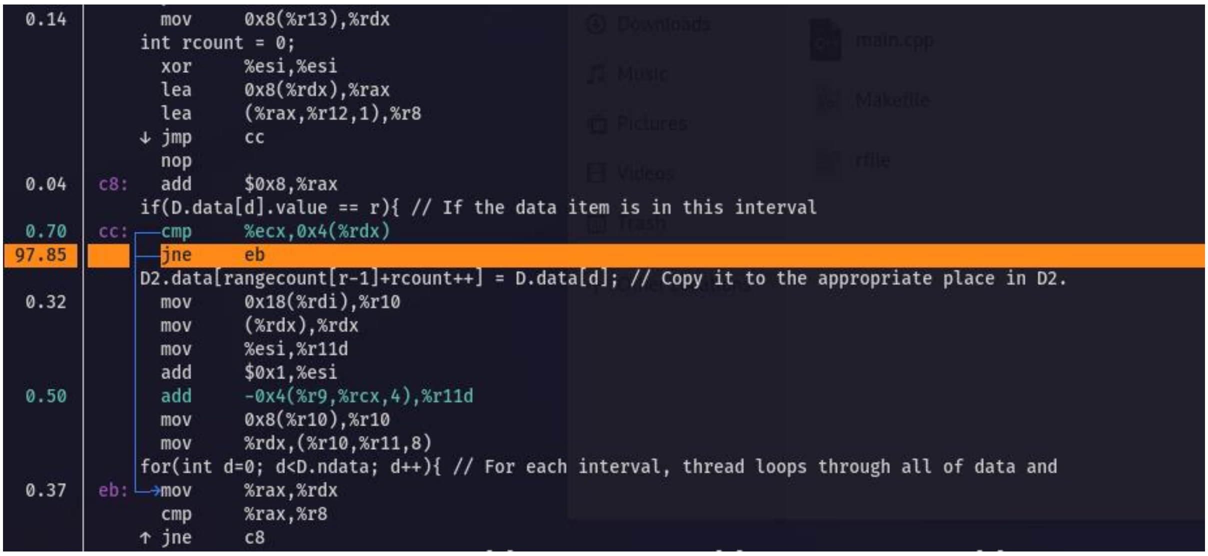
Perf record -e cache-misses command was used to add the cache misses to the report as well this was stored in as *“perf_5_1.data”*

The code was modified to make it more cache friendly keeping the algorithm largely same.

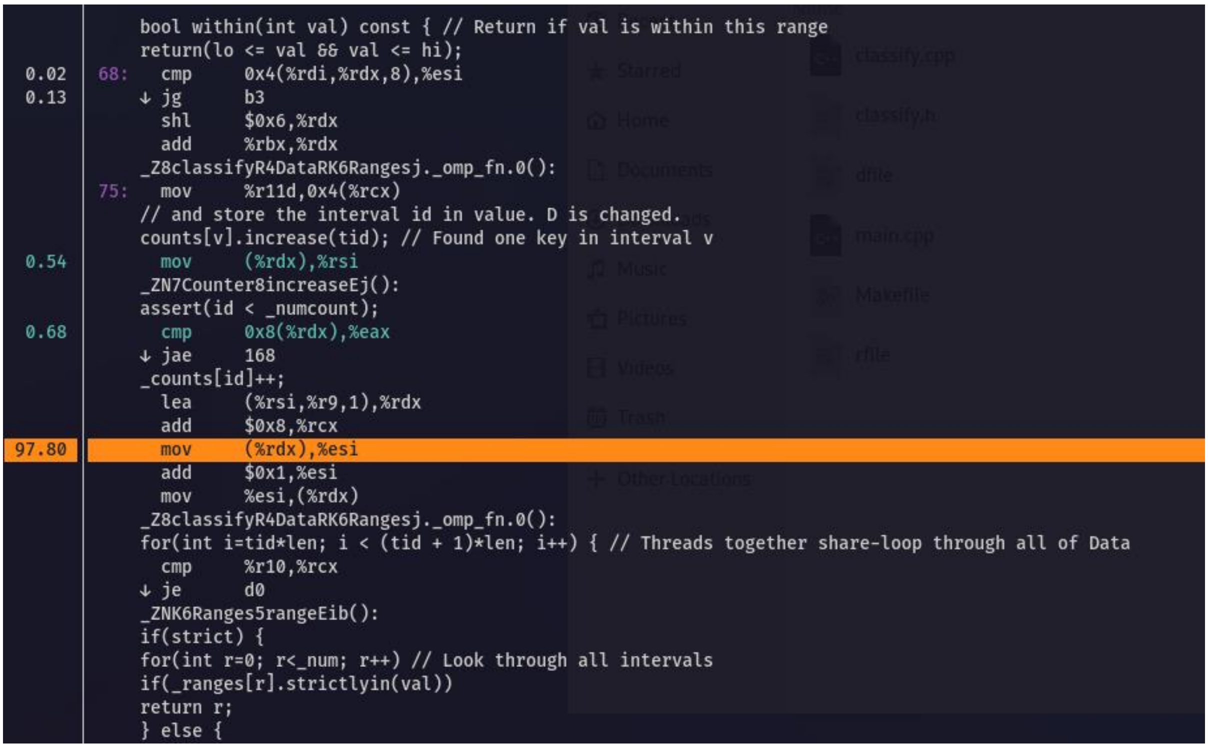
Reducing the number of loops and making iterations a bit easier the improve in time complexity was approximately 10ms but the code has better cache handling.

Perf mem record was run again and data stored into *"perf_4.data"*

The hotspots were identified and they are now different from those before optimisation.



```
 0.14         mov     0x8(%r13),%rdx
            int rcount = 0;
              xor     %esi,%esi
              lea     0x8(%rdx),%rax
              lea     (%rax,%r12,1),%r8
            ↓ jmp     cc
              nop
 0.04   c8:   add     $0x8,%rax
            if(D.data[d].value == r){ // If the data item is in this interval
 0.70   cc:   cmp     %ecx,0x4(%rdx)
97.85         jne     eb
            D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
 0.32         mov     0x18(%rdi),%r10
              mov     (%rdx),%rdx
              mov     %esi,%r11d
              add     $0x1,%esi
 0.50         add     -0x4(%r9,%rcx,4),%r11d
              mov     0x8(%r10),%r10
              mov     %rdx,(%r10,%r11,8)
            for(int d=0; d<D.ndata; d++){ // For each interval, thread loops through all of data and
 0.37   eb:   mov     %rax,%rdx
              cmp     %rax,%r8
            ↑ jne     c8
```



```
            bool within(int val) const { // Return if val is within this range
            return(lo <= val && val <= hi);
 0.02   68:   cmp     0x4(%rdi,%rdx,8),%esi
 0.13       ↓ jg      b3
              shl     $0x6,%rdx
              add     %rbx,%rdx
            _Z8classifyR4DataRK6Rangesj._omp_fn.0():
        75:   mov     %r11d,0x4(%rcx)
            // and store the interval id in value. D is changed.
            counts[v].increase(tid); // Found one key in interval v
 0.54         mov     (%rdx),%rsi
            _ZN7Counter8increaseEj():
            assert(id < _numcount);
 0.68         cmp     0x8(%rdx),%eax
            ↓ jae     168
            _counts[id]++;
              lea     (%rsi,%r9,1),%rdx
              add     $0x8,%rcx
97.80         mov     (%rdx),%esi
              add     $0x1,%esi
              mov     %esi,(%rdx)
            _Z8classifyR4DataRK6Rangesj._omp_fn.0():
            for(int i=tid*len; i < (tid + 1)*len; i++) { // Threads together share-loop through all of Data
              cmp     %r10,%rcx
            ↓ je      d0
            _ZNK6Ranges5rangeEib():
            if(strict) {
            for(int r=0; r<_num; r++) // Look through all intervals
            if(_ranges[r].strictlyin(val))
            return r;
            } else {
```

Perf record -e cache-misses command was used again and the data stored in *“perf_5_2.data”*