COL380- Assignment 0

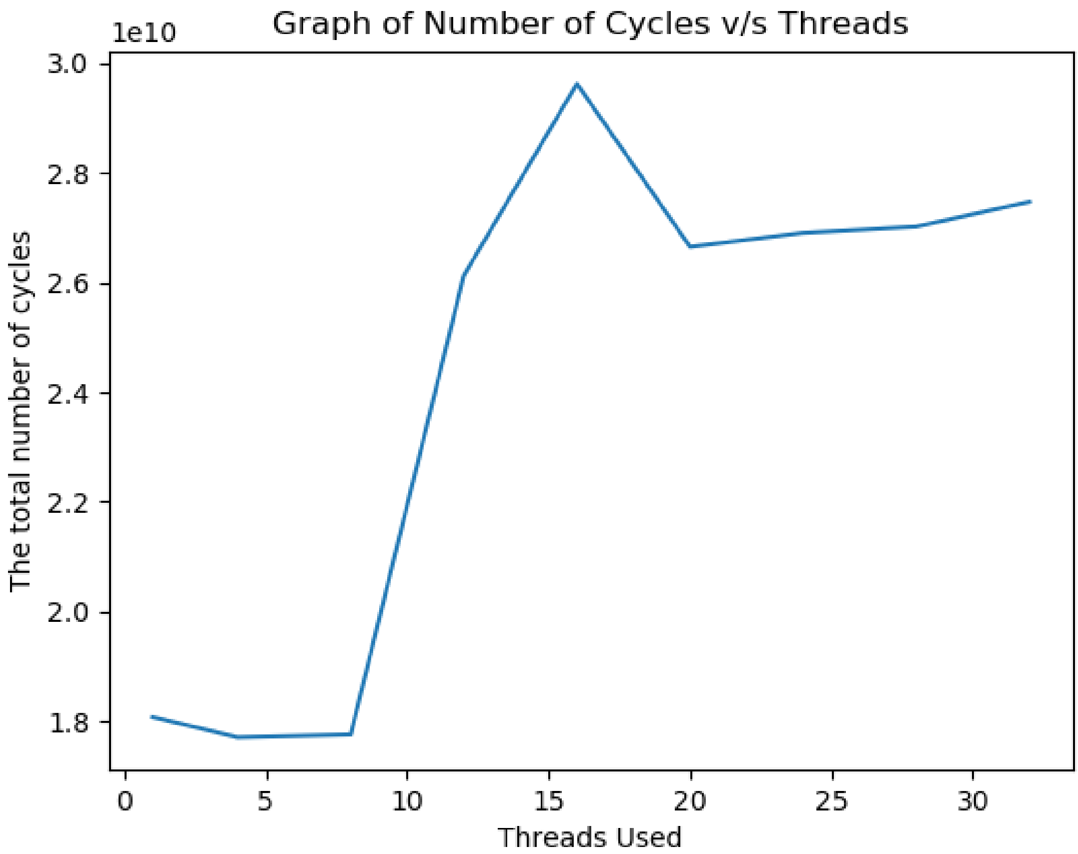
Name : Ujjwal Mehta | Entry No. : 2020CS10401

In this assignment we have learnt to use the profiling tool to profile and optimize our code and below is the corresponding description of each part.

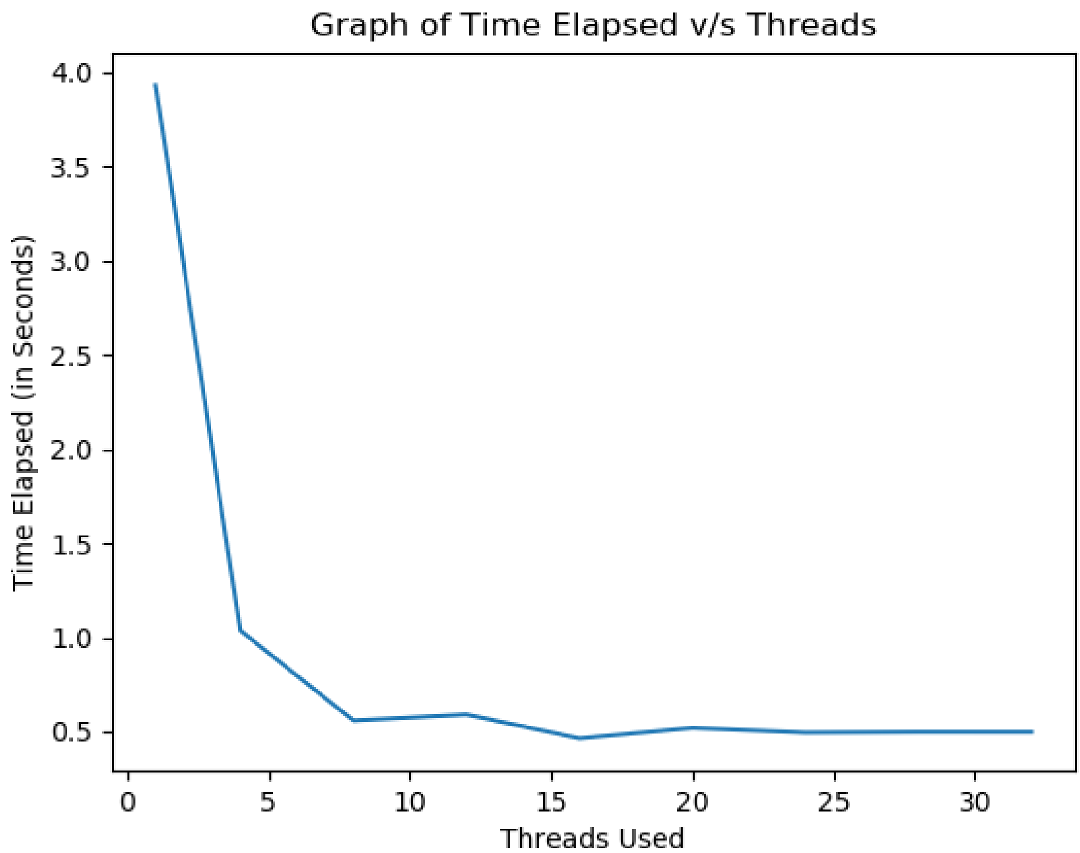
2.1 Perf Stat

On running the given program by varying the number of threads used, we get the following plots.

1. The graph of the number of cycles with varying threads is as follows:



2. The graph of time elapsed with varying threads is as follows:



Now here above we observe that the peek for the number of cycles and the bottom for the time elapsed occurs when the number of threads are 16 and this happens because the number of CPUs inside the css cluster computers (css2 is the one on which I ran my program) are 16 so all the CPU gets utilized parallely(hence maximum cycles and minimum time) while running the program and after that there is no significant change in graph since threads scheduled keep on switching on the CPUs as result of which we get the above graph.

2.2 Perf Record

We use the perf record command in order to generate the perf.data file which contains the cycles event that can be read using the perf report command using which we can obtain the information regarding the number of cycles, assembly code and the percentage of cpu which is used when we run the given classify file.

The assembly instructions which take the most CPU time are given in below screenshots which are indicated by the **red colour** .

```
              push    %rbx
              mov     0x10(%rdi),%rbp
              mov     0x18(%rdi),%ebx
            → callq   omp_get_thread_num@plt
              mov     (%r12),%r8
              cmp     (%r8),%eax
            ↓ jae     b0
              mov     %eax,%edi
              mov     0x8(%r8),%r10
              mov     0x8(%r12),%rsi
              mov     %eax,%r9d
              shl     $0x2,%rdi
              mov     %eax,%ecx
            ↓ jmp     70
              xchg    %ax,%ax
  0.36 │ 40:  cmp     0x4(%r11,%rax,8),%edx
 37.77 │    ↓ jg      93
  0.39 │      shl     $0x6,%rax
  0.01 │      add     %rbp,%rax
  0.02 │ 4e:  mov     %r13d,0x4(%r12)
  0.36 │      mov     (%rax),%rdx
  0.22 │      cmp     %r9d,0x8(%rax)
       │    ↓ jbe     b9
       │      lea     (%rdx,%rdi,1),%rax
       │      add     %ebx,%ecx
  2.43 │      mov     (%rax),%edx
```

```
  0.03 │      mov     (%r12),%edx
       │      test    %eax,%eax
       │    ↓ jle     a8
  0.02 │      mov     (%rsi),%r11
       │      lea     -0x1(%rax),%r14d
       │      xor     %eax,%eax
  0.02 │ 8a:  mov     %eax,%r13d
  9.71 │      cmp     (%r11,%rax,8),%edx
 13.88 │    ↑ jge     40
 18.66 │ 93:  lea     0x1(%rax),%r13
  1.29 │      cmp     %rax,%r14
       │    ↓ je      a8
       │      mov     %r13,%rax
 14.67 │    ↑ jmp     8a
       │      nop
       │ a8:  mov     %rbp,%rax
       │      xor     %r13d,%r13d
```

```
       │        nop
  0.07 │60:     add      $0x8,%rdx
       │64:     cmp      %eax,0x4(%rcx)
 51.65 │      ↓ jne      81
       │        mov      0x18(%rbx),%r9
  0.06 │        mov      (%rcx),%rcx
       │        mov      %esi,%r10d
       │        add      $0x1,%esi
       │        add      (%r11),%r10d
  0.03 │        mov      0x8(%r9),%r9
       │        mov      %rcx,(%r9,%r10,8)
  0.23 │81:     mov      %rdx,%rcx
       │        cmp      %rdx,%r8
 47.96 │      ↑ jne      60
```

Yes, we can map the assembly instructions to the corresponding part of source code by just compiling our files with an additional **-g flag**.

3 Hotspot Analysis

After adding the -g flag we can clearly see the source code and the assembly code and upon looking at the annotate part, we can see that the hotspot lines of code are as follows:

```
       │      bool within(int val) const { // Return if val is within this range
       │      return(lo <= val && val <= hi);
  0.42 │40:     cmp      0x4(%r11,%rax,8),%edx
 38.26 │      ↓ jg       93
  0.35 │        shl      $0x6,%rax
  0.02 │        add      %rbp,%rax
       │      _Z8classifyR4DataRK6Rangesj._omp_fn.0():
       │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
  0.43 │        mov      (%rax),%rdx
       │      _ZN7Counter8increaseEj():
       │      assert(id < _numcount);
  0.28 │        cmp      %r9d,0x8(%rax)
       │      ↓ jbe      b9
       │      _counts[id]++;
  0.02 │        lea      (%rdx,%rdi,1),%rax
       │      _Z8classifyR4DataRK6Rangesj._omp_fn.0():
       │      for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop throu
       │        add      %ebx,%ecx
```

```
            ↓ jle      a8
             if(_ranges[r].within(val))
  0.02         mov      (%rsi),%r11
               lea      -0x1(%rax),%r14d
               xor      %eax,%eax
  0.03   8a:   mov      %eax,%r13d
             _ZNK5Range6withinEi():
             return(lo <= val && val <= hi);
  9.59         cmp      (%r11,%rax,8),%edx
 14.58       ↑ jge      40
             _ZNK6Ranges5rangeEib():
             for(int r=0; r<_num; r++) // Look through all intervals
 18.25   93:   lea      0x1(%rax),%r13
  1.26         cmp      %rax,%r14
             ↓ je       a8
               mov      %r13,%rax
 14.05       ↑ jmp      8a
               nop
```

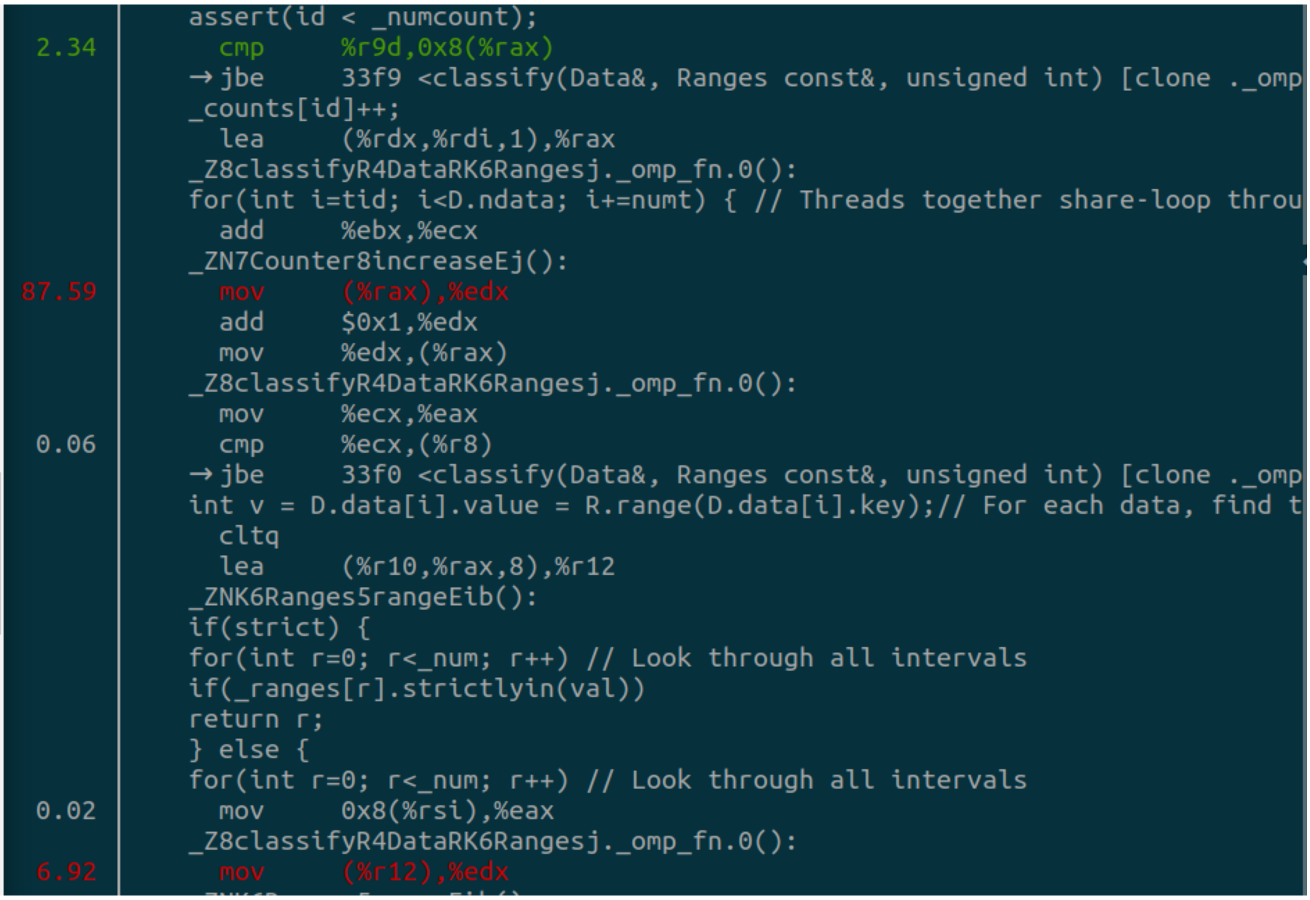
The prospective problem which makes this code snippet the top hotspot is that whenever we are finding the corresponding range in which we should assign a given data point then we call the **within method** the total number of ranges times due to which we have to load the address of this method a lot of times to execute it.

Yes, the code can be optimized in order to improve the performance of this hotspot and in order to optimize it quite significantly, we can change our algorithm a little bit to find the range corresponding to a given data point with a better time complexity like O(logN) rather then linear scanning using Tree like data structures to store ranges.

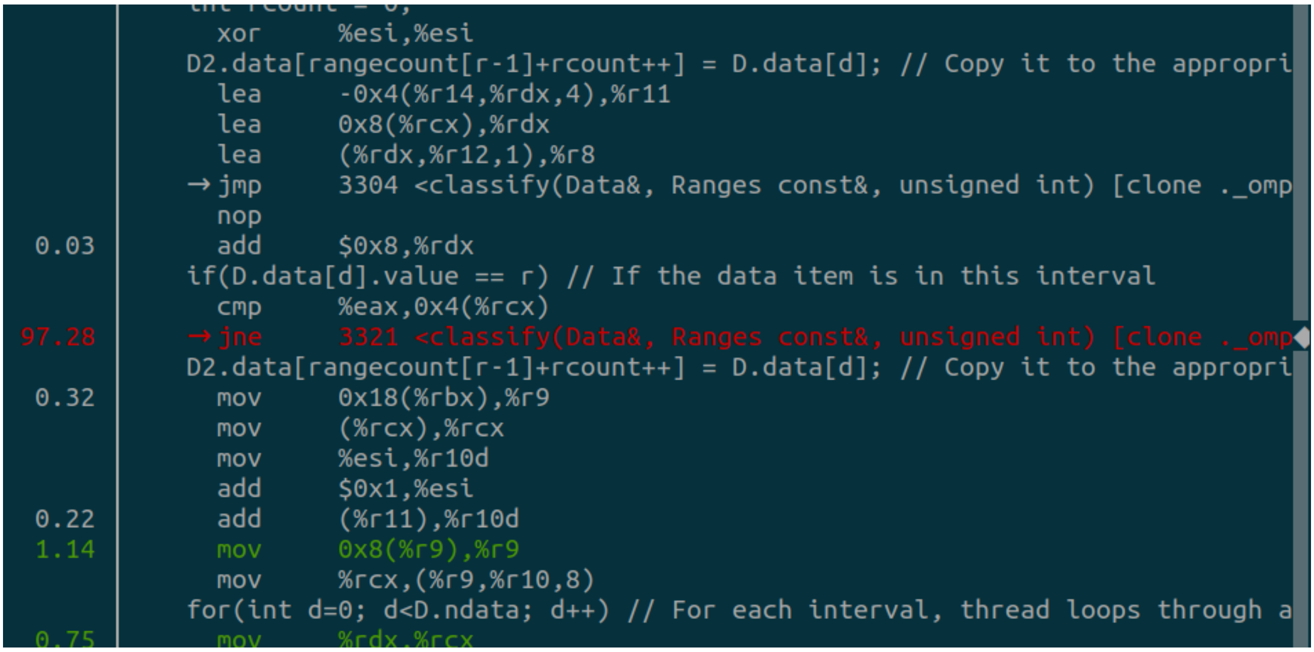
4 Memory Profiling

In this part of assignment, we will optimize our program further in order to make it more cache friendly by removing the instances of false sharing.

The top 2 hotspots obtained after running perf mem report on the original code are:



```
        assert(id < _numcount);
  2.34    cmp     %r9d,0x8(%rax)
        → jbe     33f9 <classify(Data&, Ranges const&, unsigned int) [clone ._omp
        _counts[id]++;
          lea     (%rdx,%rdi,1),%rax
        _Z8classifyR4DataRK6Rangesj._omp_fn.0():
        for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop throu
          add     %ebx,%ecx
        _ZN7Counter8increaseEj():
 87.59    mov     (%rax),%edx
          add     $0x1,%edx
          mov     %edx,(%rax)
        _Z8classifyR4DataRK6Rangesj._omp_fn.0():
          mov     %ecx,%eax
  0.06    cmp     %ecx,(%r8)
        → jbe     33f0 <classify(Data&, Ranges const&, unsigned int) [clone ._omp
        int v = D.data[i].value = R.range(D.data[i].key);// For each data, find t
          cltq
          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
  0.02    mov     0x8(%rsi),%eax
        _Z8classifyR4DataRK6Rangesj._omp_fn.0():
  6.92    mov     (%r12),%edx
```



```
          xor     %esi,%esi
        D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropri
          lea     -0x4(%r14,%rdx,4),%r11
          lea     0x8(%rcx),%rdx
          lea     (%rdx,%r12,1),%r8
        → jmp     3304 <classify(Data&, Ranges const&, unsigned int) [clone ._omp
          nop
  0.03    add     $0x8,%rdx
        if(D.data[d].value == r) // If the data item is in this interval
          cmp     %eax,0x4(%rcx)
 97.28  → jne     3321 <classify(Data&, Ranges const&, unsigned int) [clone ._omp
        D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropri
  0.32    mov     0x18(%rbx),%r9
          mov     (%rcx),%rcx
          mov     %esi,%r10d
          add     $0x1,%esi
  0.22    add     (%r11),%r10d
  1.14    mov     0x8(%r9),%r9
          mov     %rcx,(%r9,%r10,8)
        for(int d=0; d<D.ndata; d++) // For each interval, thread loops through a
  0.75    mov     %rdx,%rcx
```

Now based on the above hotspot obtained, we can clearly identify 2 issues in the code which are leading to instances of **false sharing** and these are the lines

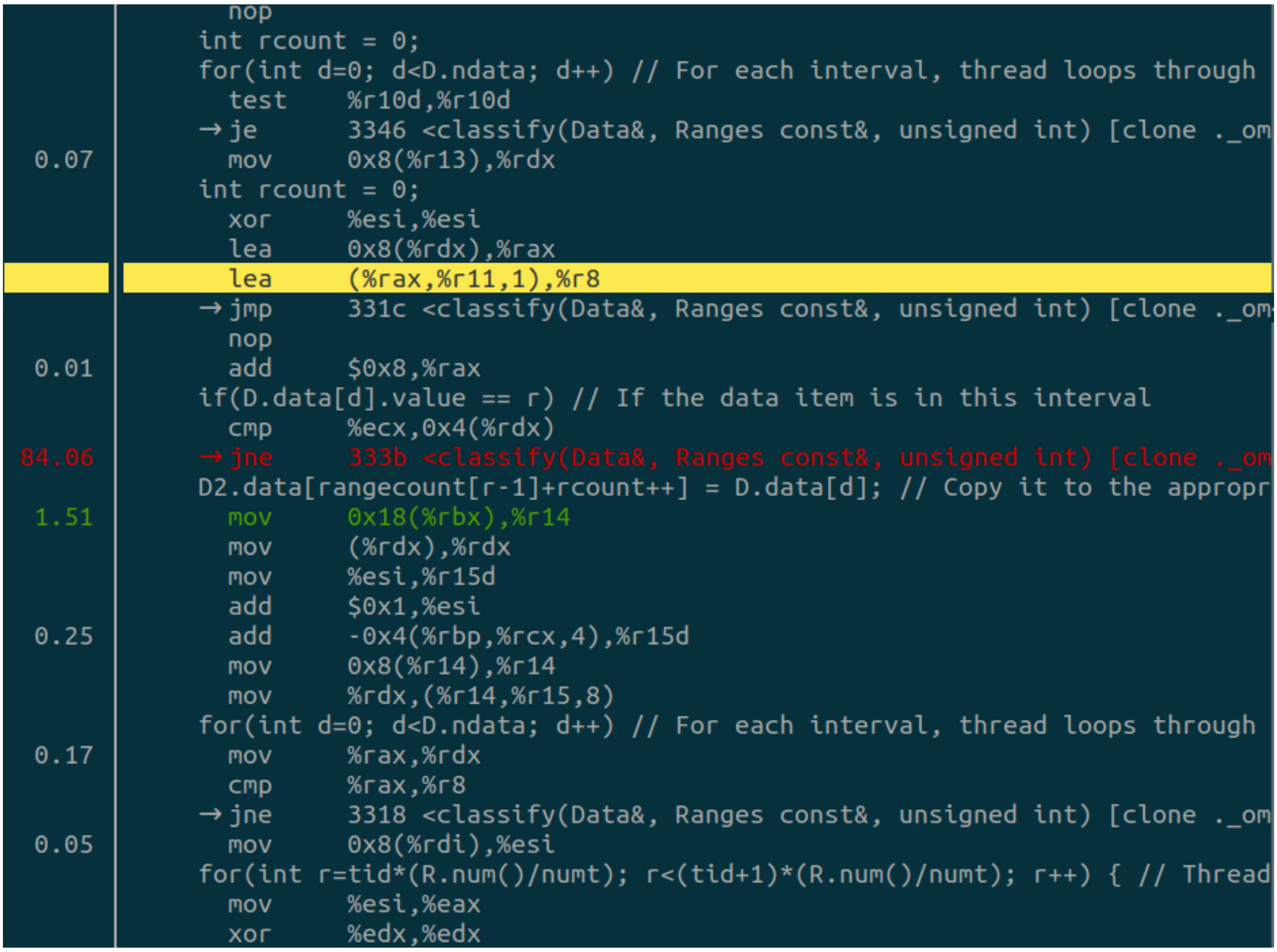
```
counts[v].increase(tid); // Found one key in interval v
```

```
 D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the
appropriate place in D2.
```

And the above lines there is false sharing because the threads are accessing the elements of the array in an **interleaved manner** which is leading to writing in same line by 2 different threads resulting in a lot of cache misses.

These instances of false sharing can be handled by making each thread process a contiguous section of the array hence making them independent from each other, hence we create our own matrix for the counts rather then using the counter object since where each thread will operate on a row increasing the cache hit rate and similarly removing interleaving inside the lower for loop as well.

After performing the above optimizations, we get the following report screenshots:



```
               nop
             int rcount = 0;
             for(int d=0; d<D.ndata; d++) // For each interval, thread loops through
               test    %r10d,%r10d
             → je      3346 <classify(Data&, Ranges const&, unsigned int) [clone ._om
   0.07        mov     0x8(%r13),%rdx
             int rcount = 0;
               xor     %esi,%esi
               lea     0x8(%rdx),%rax
               lea     (%rax,%r11,1),%r8
             → jmp     331c <classify(Data&, Ranges const&, unsigned int) [clone ._om
               nop
   0.01        add     $0x8,%rax
             if(D.data[d].value == r) // If the data item is in this interval
               cmp     %ecx,0x4(%rdx)
  84.06      → jne     333b <classify(Data&, Ranges const&, unsigned int) [clone ._om
             D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropr
   1.51        mov     0x18(%rbx),%r14
               mov     (%rdx),%rdx
               mov     %esi,%r15d
               add     $0x1,%esi
   0.25        add     -0x4(%rbp,%rcx,4),%r15d
               mov     0x8(%r14),%r14
               mov     %rdx,(%r14,%r15,8)
             for(int d=0; d<D.ndata; d++) // For each interval, thread loops through
   0.17        mov     %rax,%rdx
               cmp     %rax,%r8
             → jne     3318 <classify(Data&, Ranges const&, unsigned int) [clone ._om
   0.05        mov     0x8(%rdi),%esi
             for(int r=tid*(R.num()/numt); r<(tid+1)*(R.num()/numt); r++) { // Thread
               mov     %esi,%eax
               xor     %edx,%edx
```

```
 0.02 │     mov     0x8(%r13),%rdx
      │   int rcount = 0;
      │     xor     %esi,%esi
      │     lea     0x8(%rdx),%rax
      │     lea     (%rax,%r11,1),%r8
      │   → jmp     339c <classify(Data&, Ranges const&, unsigned int) [clone ._om
      │     nop
      │     add     $0x8,%rax
      │   if(D.data[d].value == r) // If the data item is in this interval
 0.02 │     cmp     %ecx,0x4(%rdx)
13.56 │   → jne     33bb <classify(Data&, Ranges const&, unsigned int) [clone ._om
      │   D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropr
 0.04 │     mov     0x18(%rbx),%r9
      │     mov     (%rdx),%rdx
      │     mov     %esi,%r10d
      │     add     $0x1,%esi
 0.09 │     add     -0x4(%rbp,%rcx,4),%r10d
      │     mov     0x8(%r9),%r9
      │     mov     %rdx,(%r9,%r10,8)
      │   for(int d=0; d<D.ndata; d++) // For each interval, thread loops through
```

Hence we see a significant decrease in terms of usage percentage.

Finally after running perf record with cache-misses flag on original as well as optimized code, we see that their is an increase in our cache hit rate.