COL380: Assignment-0
Code Profiling

Ishita Chaudhary
2019CS10360
January 16, 2023

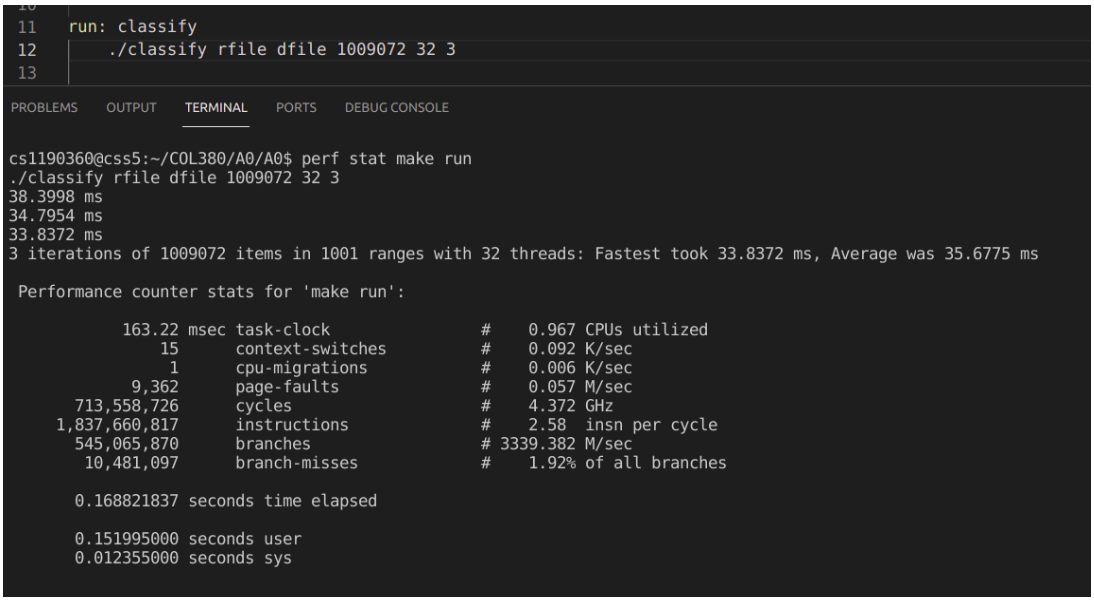
Introduction

The aim is to learn how to profile code, identify and analyze the hotspots, and suggest and implement further changes to the code to improve its performance. We will use the tool *perf* to do the same.

Running Perf

The following experiments were run on CSS cluster and analyzed via the perf profiling tool.

We vary the number of threads in the program and run perf stat for thread values 1,4,8, 12,....,32 and plotted the values of the total time elapsed and cycles as a function of the number of threads. As we can see below, the data is obtained for 32 threads.

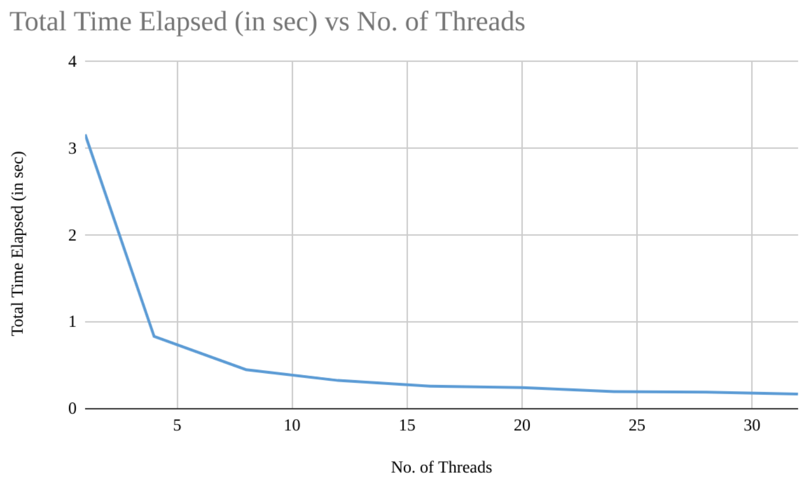


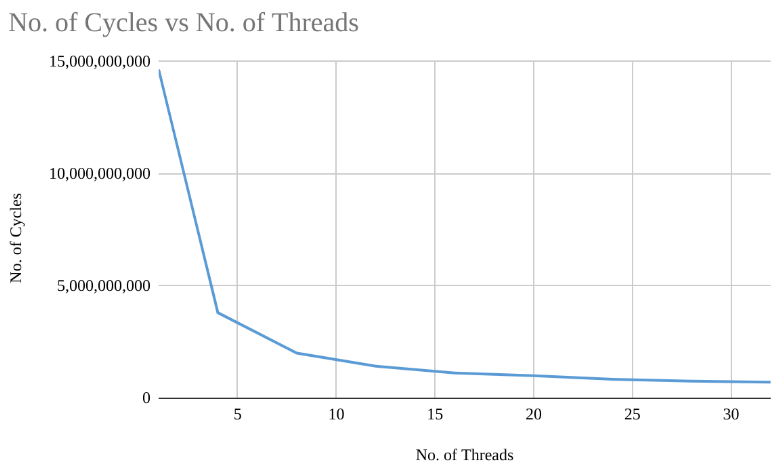
```
11  run: classify
12      ./classify rfile dfile 1009072 32 3
13

PROBLEMS   OUTPUT   TERMINAL   PORTS   DEBUG CONSOLE

cs1190360@css5:~/COL380/A0/A0$ perf stat make run
./classify rfile dfile 1009072 32 3
38.3998 ms
34.7954 ms
33.8372 ms
3 iterations of 1009072 items in 1001 ranges with 32 threads: Fastest took 33.8372 ms, Average was 35.6775 ms

 Performance counter stats for 'make run':

           163.22 msec task-clock                #    0.967 CPUs utilized
               15      context-switches          #    0.092 K/sec
                1      cpu-migrations            #    0.006 K/sec
            9,362      page-faults               #    0.057 M/sec
      713,558,726      cycles                    #    4.372 GHz
    1,837,660,817      instructions              #    2.58  insn per cycle
      545,065,870      branches                  # 3339.382 M/sec
       10,481,097      branch-misses             #    1.92% of all branches

      0.168821837 seconds time elapsed

      0.151995000 seconds user
      0.012355000 seconds sys
```

The following values were obtained while running the experiment; the graphs of the data are attached below.

No. of Threads	Total Time Elapsed (in sec)	No. of Cycles
1	3.162526922	14,644,647,666
4	0.834439504	3,810,059,666
8	0.44995137	2,011,276,679
12	0.325487913	1,430,046,821

16	0.259278178	1,122,379,915
20	0.244099116	1,005,897,561
24	0.197652154	846,661,987
28	0.190470714	765,351,738
32	0.168821837	713,558,726





As we increase the number of threads, the total time elapsed and the number of cycles decreases exponentially due to the SIMD structure of the core. Hence, concurrent execution of the same instruction in various threads can occur simultaneously.

For further experiments, we take four threads and ten repetitions.

```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14142682732
classify  /home/btech/cs1190360/COL380/A0/A0/classify [Percent: local period]
Percent│        lea      -0x4(%r15,%rax,4),%r9
       │        lea      0x8(%rdx),%rax
       │        lea      (%rax,%r14,1),%rdi
       │      → jmp      355c <classify(Data&, Ranges const&, unsigned int)+0x2dc>
       │        nop
       │        add      $0x8,%rax
       │        cmp      %ecx,0x4(%rdx)
  19.04│      → jne      3572 <classify(Data&, Ranges const&, unsigned int)+0x2f2>
       │        mov      (%rdx),%rdx
       │        mov      %esi,%r8d
       │        add      (%r9),%r8d
       │        add      $0x1,%esi
       │        mov      %rdx,0x0(%r13,%r8,8)
   0.01│        mov      %rax,%rdx
       │        cmp      %rax,%rdi
   2.75│      → jne      3558 <classify(Data&, Ranges const&, unsigned int)+0x2d8>
       │        add      %r12d,%ecx
       │        cmp      %ecx,%r10d
       │      → jg       3538 <classify(Data&, Ranges const&, unsigned int)+0x2b8>
       │        mov      -0x38(%rbp),%rbx
       │        xor      %fs:0x28,%rbx
       │        mov      -0x48(%rbp),%eax
       │      → jne      360e <classify(Data&, Ranges const&, unsigned int)+0x38e>
```

```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14142682732
classify  /home/btech/cs1190360/COL380/A0/A0/classify [Percent: local period]
Percent│        cltq
   0.05│        lea      (%r9,%rax,8),%rdi
       │        mov      0x8(%r14),%eax
   0.01│        mov      (%rdi),%edx
       │        test     %eax,%eax
       │      → jle      33f2 <classify(Data&, Ranges const&, unsigned int)+0x172>
   0.01│        mov      (%r14),%r11
       │        lea      -0x1(%rax),%r15d
       │        xor      %eax,%eax
       │        mov      %eax,%ecx
   8.35│        cmp      (%r11,%rax,8),%edx
  11.46│      → jge      3390 <classify(Data&, Ranges const&, unsigned int)+0x110>
  15.52│        lea      0x1(%rax),%rcx
   0.77│        cmp      %r15,%rax
       │      → je       33f2 <classify(Data&, Ranges const&, unsigned int)+0x172>
       │        mov      %rcx,%rax
  11.13│      → jmp      33dc <classify(Data&, Ranges const&, unsigned int)+0x15c>
       │        mov      -0x48(%rbp),%rax
       │        xor      %ecx,%ecx
       │      → jmp      339f <classify(Data&, Ranges const&, unsigned int)+0x11f>
       │        movabs   $0x1ffffffffffffffe,%rdx
       │        movslq   0x8(%r14),%rax
       │        cmp      %rdx,%rax
```

```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14142682732
classify  /home/btech/cs1190360/COL380/A0/A0/classify [Percent: local period]
Percent          mov        %eax,%r8d
                 mov        0x8(%rbx),%r9
                 mov        %eax,%esi
                 shl        $0x2,%r8
               → jmp        33c3 <classify(Data&, Ranges const&, unsigned int)+0x143>
                 nop
  0.26           cmp        0x4(%r11,%rax,8),%edx
 29.48         → jg         33e4 <classify(Data&, Ranges const&, unsigned int)+0x164>
  0.19           shl        $0x6,%rax
  0.12           add        -0x48(%rbp),%rax
                 mov        %ecx,0x4(%rdi)
  0.26           mov        (%rax),%rdx
  0.12           cmp        0x8(%rax),%r10d
               → jae        35ef <classify(Data&, Ranges const&, unsigned int)+0x36f>
  0.01           lea        (%rdx,%r8,1),%rax
                 add        %r12d,%esi
  0.22           mov        (%rax),%edx
  0.02           add        $0x1,%edx
  0.04           mov        %edx,(%rax)
                 mov        %esi,%eax
  0.01           cmp        %esi,(%rbx)
               → jbe        33fa <classify(Data&, Ranges const&, unsigned int)+0x17a>
                 cltq
```

Which assembly instruction takes the most CPU time?
The "Jump if Greater" instruction (abbreviated as jg) takes the most CPU time.

Can you map the instruction to the part of the source code it corresponds to?
This corresponds to the jump within the for loop which maps each data point to its respective range.

```
#pragma omp parallel num_threads(numt)
{
   int tid = omp_get_thread_num(); // I am thread number tid
   for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
      int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
                                                      // and store the interval id in value. D is changed.
      counts[v].increase(tid); // Found one key in interval v
   }
}
```

We add the "-g" flag in the makefile along with CFLAGS to allow the perf report to show the source code and assembly instructions.

Hotspot Analysis
Report the top hotspot in your write-up (attach a screenshot of the perf report showing the code and the percentage of time taken).
The top hotspot is the jump within the for loop which identifies the range of each data point, as reported above. The below screenshot confirms the above-mentioned claim, the address and offsets correspond to the for loop in the source code.

```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14386687119
classify  /home/btech/cs1190360/COL380/A0/classify [Percent: local period]
Percent│      hi = b;
       │      }
       │
       │      bool within(int val) const { // Return if val is within this range
       │      return(lo <= val && val <= hi);
  0.27 │        cmp     0x4(%r11,%rax,8),%edx
 28.14 │      → jg      33e4 <classify(Data&, Ranges const&, unsigned int)+0x164>
  0.24 │        shl     $0x6,%rax
  0.09 │        add     -0x48(%rbp),%rax
       │      _Z8classifyR4DataRK6Rangesj():
       │        mov     %ecx,0x4(%rdi)
       │      // and store the interval id in value. D is changed.
       │      counts[v].increase(tid); // Found one key in interval v
  0.31 │        mov     (%rax),%rdx
       │      _ZN7Counter8increaseEj():
       │      assert(id < _numcount);
  0.10 │        cmp     0x8(%rax),%r10d
       │      → jae     35ef <classify(Data&, Ranges const&, unsigned int)+0x36f>
       │      _counts[id]++;
       │        lea     (%rdx,%r8,1),%rax
       │      _Z8classifyR4DataRK6Rangesj():
       │      for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
```

```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14386687119
classify  /home/btech/cs1190360/COL380/A0/classify [Percent: local period]
Percent│        mov     0x8(%rbx),%rdx
       │      if(D.data[d].value == r) // If the data item is in this interval
       │      D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
       │        movslq  %ecx,%rax
       │      int rcount = 0;
       │        xor     %esi,%esi
       │      D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
       │        lea     -0x4(%r15,%rax,4),%r9
       │        lea     0x8(%rdx),%rax
       │        lea     (%rax,%r14,1),%rdi
       │      → jmp     355c <classify(Data&, Ranges const&, unsigned int)+0x2dc>
       │        nop
       │        add     $0x8,%rax
       │      if(D.data[d].value == r) // If the data item is in this interval
       │        cmp     %ecx,0x4(%rdx)
 19.16 │      → jne     3572 <classify(Data&, Ranges const&, unsigned int)+0x2f2>
       │      D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
       │        mov     (%rdx),%rdx
       │        mov     %esi,%r8d
       │        add     (%r9),%r8d
       │        add     $0x1,%esi
       │        mov     %rdx,0x0(%r13,%r8,8)
       │      for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
```

```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14386687119
classify  /home/btech/cs1190360/COL380/A0/classify [Percent: local period]
Percent│        mov     (%r14),%r11
       │        lea     -0x1(%rax),%r15d
       │        xor     %eax,%eax
       │        mov     %eax,%ecx
       │      _ZNK5Range6withinEi():
       │      return(lo <= val && val <= hi);
  8.54 │        cmp     (%r11,%rax,8),%edx
 11.83 │      → jge     3390 <classify(Data&, Ranges const&, unsigned int)+0x110>
       │      _ZNK6Ranges5rangeEib():
       │      for(int r=0; r<_num; r++) // Look through all intervals
 15.00 │        lea     0x1(%rax),%rcx
  0.92 │        cmp     %r15,%rax
       │      → je      33f2 <classify(Data&, Ranges const&, unsigned int)+0x172>
       │        mov     %rcx,%rax
 12.19 │      → jmp     33dc <classify(Data&, Ranges const&, unsigned int)+0x15c>
       │        mov     -0x48(%rbp),%rax
       │      return r;
       │      }
       │      return BADRANGE; // Did not find any range
       │        xor     %ecx,%ecx
       │      → jmp     339f <classify(Data&, Ranges const&, unsigned int)+0x11f>
       │      _Z8classifyR4DataRK6Rangesj():
       │      }
```

What is the prospective problem which makes this code snippet the top hotspot? Can the code be optimized to improve the performance of this hotspot? If it can be optimized, suggest the optimizations in the write-up.

Yes, we can optimize the code to improve the performance of this hotspot. Instead of giving consequent data reads and writes to different threads, we should allocate work on contiguous blocks of data to each thread so that the cache miss rate while reading reduces significantly. This ensures that time is not wasted in unnecessarily loading and storing cache lines by different threads, thus saving time in each iteration.

After running the perf record with the desired flags, we get the following stats on the original code:

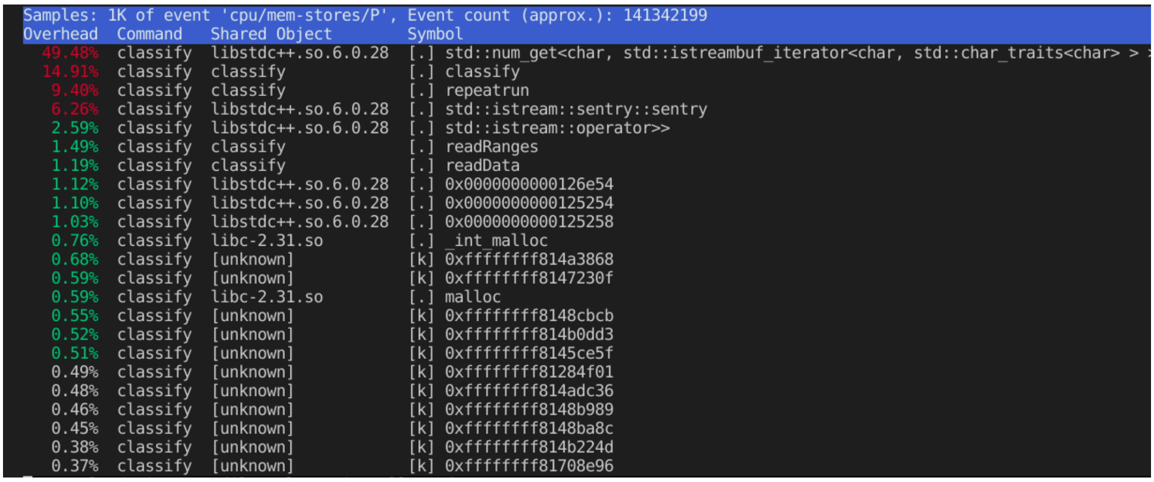
```
Available samples
10K branches
8K branch-misses
1K cache-misses
22 page-faults
10K cpu-cycles
```

Memory Profiling

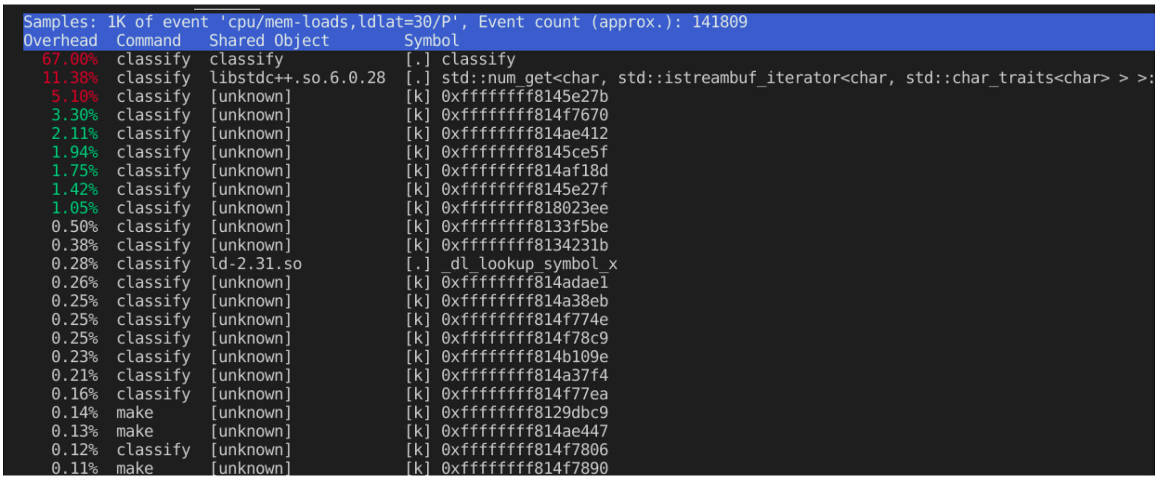
Run perf mem record on the given code, and analyze the report generated.

We get 437 CPU memory loads and ~1000 CPU memory stores. The report for each of them is attached below.

```
Available samples
437 cpu/mem-loads,ldlat=30/P
1K cpu/mem-stores/P
```



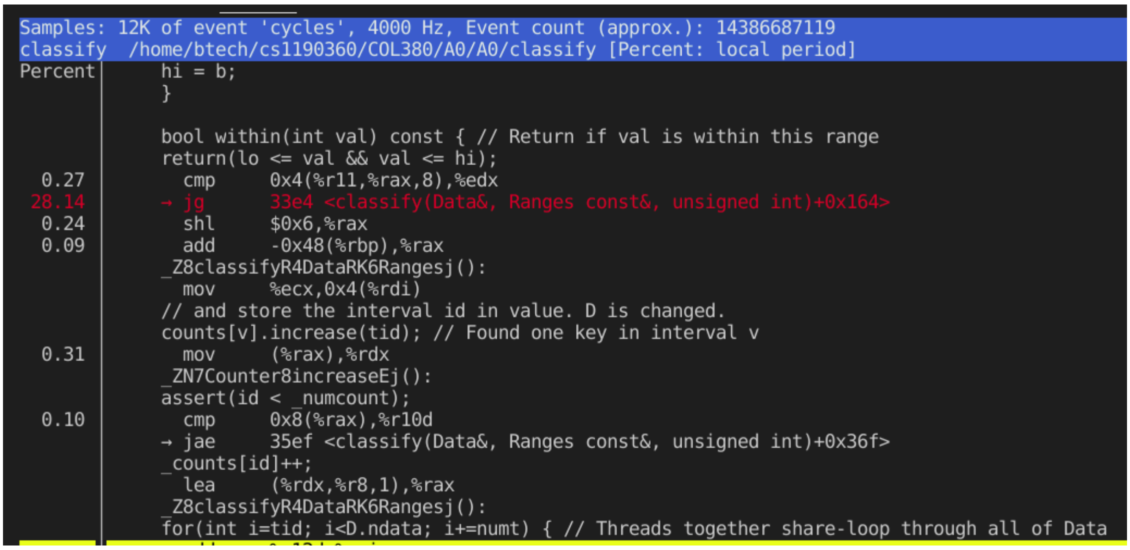
```
Samples: 1K of event 'cpu/mem-stores/P', Event count (approx.): 141342199
Overhead  Command   Shared Object        Symbol
  49.48%  classify  libstdc++.so.6.0.28  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > :
  14.91%  classify  classify             [.] classify
   9.40%  classify  classify             [.] repeatrun
   6.26%  classify  libstdc++.so.6.0.28  [.] std::istream::sentry::sentry
   2.59%  classify  libstdc++.so.6.0.28  [.] std::istream::operator>>
   1.49%  classify  classify             [.] readRanges
   1.19%  classify  classify             [.] readData
   1.12%  classify  libstdc++.so.6.0.28  [.] 0x0000000000126e54
   1.10%  classify  libstdc++.so.6.0.28  [.] 0x0000000000125254
   1.03%  classify  libstdc++.so.6.0.28  [.] 0x0000000000125258
   0.76%  classify  libc-2.31.so         [.] _int_malloc
   0.68%  classify  [unknown]            [k] 0xffffffff814a3868
   0.59%  classify  [unknown]            [k] 0xffffffff8147230f
   0.59%  classify  libc-2.31.so         [.] malloc
   0.55%  classify  [unknown]            [k] 0xffffffff8148cbcb
   0.52%  classify  [unknown]            [k] 0xffffffff814b0dd3
   0.51%  classify  [unknown]            [k] 0xffffffff8145ce5f
   0.49%  classify  [unknown]            [k] 0xffffffff81284f01
   0.48%  classify  [unknown]            [k] 0xffffffff814adc36
   0.46%  classify  [unknown]            [k] 0xffffffff8148b989
   0.45%  classify  [unknown]            [k] 0xffffffff8148ba8c
   0.38%  classify  [unknown]            [k] 0xffffffff814b224d
   0.37%  classify  [unknown]            [k] 0xffffffff81708e96
```



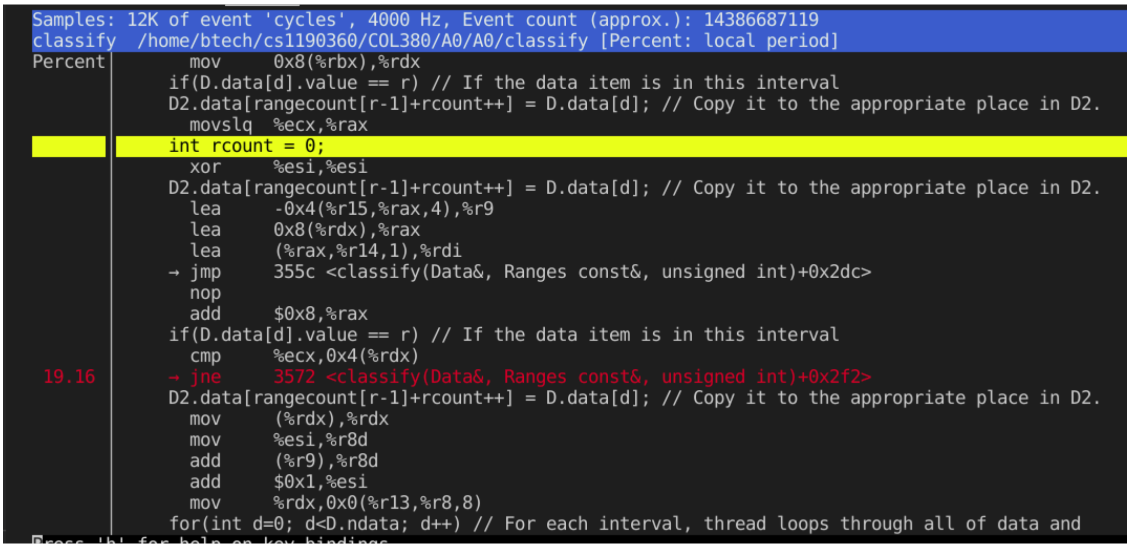
```
Samples: 1K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 141809
Overhead  Command   Shared Object        Symbol
  67.00%  classify  classify             [.] classify
  11.38%  classify  libstdc++.so.6.0.28  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >:
   5.10%  classify  [unknown]            [k] 0xffffffff8145e27b
   3.30%  classify  [unknown]            [k] 0xffffffff814f7670
   2.11%  classify  [unknown]            [k] 0xffffffff814ae412
   1.94%  classify  [unknown]            [k] 0xffffffff8145ce5f
   1.75%  classify  [unknown]            [k] 0xffffffff814af18d
   1.42%  classify  [unknown]            [k] 0xffffffff8145e27f
   1.05%  classify  [unknown]            [k] 0xffffffff818023ee
   0.50%  classify  [unknown]            [k] 0xffffffff8133f5be
   0.38%  classify  [unknown]            [k] 0xffffffff8134231b
   0.28%  classify  ld-2.31.so           [.] _dl_lookup_symbol_x
   0.26%  classify  [unknown]            [k] 0xffffffff814adae1
   0.25%  classify  [unknown]            [k] 0xffffffff814a38eb
   0.25%  classify  [unknown]            [k] 0xffffffff814f774e
   0.25%  classify  [unknown]            [k] 0xffffffff814f78c9
   0.23%  classify  [unknown]            [k] 0xffffffff814b109e
   0.21%  classify  [unknown]            [k] 0xffffffff814a37f4
   0.16%  classify  [unknown]            [k] 0xffffffff814f77ea
   0.14%  make      [unknown]            [k] 0xffffffff8129dbc9
   0.13%  make      [unknown]            [k] 0xffffffff814ae447
   0.12%  classify  [unknown]            [k] 0xffffffff814f7806
   0.11%  make      [unknown]            [k] 0xffffffff814f7890
```

We can observe that 67% of the memory loads are done in the function classify.

Report the top 2 hotspots (attach a screenshot of the perf report showing the code and percentage of time taken).



```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14386687119
classify  /home/btech/cs1190360/COL380/A0/A0/classify [Percent: local period]
Percent │     hi = b;
        │     }
        │
        │     bool within(int val) const { // Return if val is within this range
        │     return(lo <= val && val <= hi);
   0.27 │       cmp    0x4(%r11,%rax,8),%edx
  28.14 │     → jg     33e4 <classify(Data&, Ranges const&, unsigned int)+0x164>
   0.24 │       shl    $0x6,%rax
   0.09 │       add    -0x48(%rbp),%rax
        │     _Z8classifyR4DataRK6Rangesj():
        │       mov    %ecx,0x4(%rdi)
        │     // and store the interval id in value. D is changed.
        │     counts[v].increase(tid); // Found one key in interval v
   0.31 │       mov    (%rax),%rdx
        │     _ZN7Counter8increaseEj():
        │     assert(id < _numcount);
   0.10 │       cmp    0x8(%rax),%r10d
        │     → jae    35ef <classify(Data&, Ranges const&, unsigned int)+0x36f>
        │     _counts[id]++;
        │       lea    (%rdx,%r8,1),%rax
        │     _Z8classifyR4DataRK6Rangesj():
        │     for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
```



```
Samples: 12K of event 'cycles', 4000 Hz, Event count (approx.): 14386687119
classify  /home/btech/cs1190360/COL380/A0/A0/classify [Percent: local period]
Percent│      mov     0x8(%rbx),%rdx
       │    if(D.data[d].value == r) // If the data item is in this interval
       │    D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
       │      movslq  %ecx,%rax
       │    int rcount = 0;
       │      xor     %esi,%esi
       │    D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
       │      lea     -0x4(%r15,%rax,4),%r9
       │      lea     0x8(%rdx),%rax
       │      lea     (%rax,%r14,1),%rdi
       │    → jmp     355c <classify(Data&, Ranges const&, unsigned int)+0x2dc>
       │      nop
       │      add     $0x8,%rax
       │    if(D.data[d].value == r) // If the data item is in this interval
       │      cmp     %ecx,0x4(%rdx)
 19.16 │    → jne     3572 <classify(Data&, Ranges const&, unsigned int)+0x2f2>
       │    D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
       │      mov     (%rdx),%rdx
       │      mov     %esi,%r8d
       │      add     (%r9),%r8d
       │      add     $0x1,%esi
       │      mov     %rdx,0x0(%r13,%r8,8)
       │    for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
```

These are the two hotspots, and both of them are a part of the original classify function. The first one (which consumes 28.14% of execution time) is a for loop which finds the accurate data range for each data point and the second one is also a part of the for loop where we are finally writing the data which we return.

Based on the hotspots you obtained, identify at least two issues in the code that makes it cache unfriendly and suggest improvements.

- For each data, we find the interval of data's key,and store the interval id in value and then increase the count of the interval by one. While performing this, the threads together share-loop through all of the data in such a way that consequent reads occur from different cache lines resulting in a lot of cache-misses and subsequently un-necessary loads and stores. Instead we can allocate work on contiguous blocks of data to each thread so that the cache miss rate reduces while performing this task.
- On analysing the code, it was observed that while computing D_2, using 2 for loops is redundant and can be computed using a single loop improving time complexity from O(R*D) to O(D)

Cache Line Width is 64 bytes, int size is 4bytes and Item (struct) size is 8bytes (key (4bytes), value(4bytes)), that is, it is desirable to access 8 contiguous (64/8) Item values and 16 contiguous (64/4) int values. So, each thread iterates over 8 contiguous elements (item struct) to compute the range for each element (first for loop), and iterates over 16 contiguous elements (int) to compute the elements in a particular range.
For this, two variable Items Accessed By One Thread (=8) and, Ints Accessed By One Thread (=16) are used.

So, a thread will access all elements in L1 data cache and will then move to another row.

Run perf mem record after the improvements and submit the screenshots.
There were 437 CPU memory loads which were reduced to 412 CPU memory loads after optimizations. That is, the optimized code is better in terms of the number of times memory loads were done by the CPU.

```
Available samples
412 cpu/mem-loads,ldlat=30/P
1K cpu/mem-stores/P
```

After Optimization

```
Available samples
437 cpu/mem-loads,ldlat=30/P
1K cpu/mem-stores/P
```

Before Optimization

Run perf with the cache misses flag on the original code and the final code you obtain. Do you see an improvement? If not, suggest ways to further improve cache hit rate.

```
Available samples
1K cache-misses
```

```
Samples: 1K of event 'cache-misses', Event count (approx.): 3638446
Overhead  Command   Shared Object        Symbol
  31.06%  classify  classify             [.] classify
  17.24%  classify  classify             [.] repeatrun
   4.62%  classify  [unknown]            [k] 0xffffffff81df90c0
   3.92%  classify  [unknown]            [k] 0xffffffff814fa233
   3.61%  classify  [unknown]            [k] 0xffffffff814f7880
   3.13%  classify  [unknown]            [k] 0xffffffff818023ee
   2.40%  classify  [unknown]            [k] 0xffffffff814fa219
   2.14%  classify  [unknown]            [k] 0xffffffff814fa285
   1.57%  classify  [unknown]            [k] 0xffffffff814f4894
   1.35%  classify  [unknown]            [k] 0xffffffff8148babd
   1.24%  classify  [unknown]            [k] 0xffffffff814f489f
   1.22%  classify  [unknown]            [k] 0xffffffff814fa2cd
   1.12%  classify  [unknown]            [k] 0xffffffff8148285d
   1.11%  classify  [unknown]            [k] 0xffffffff814fcc56
   1.11%  classify  [unknown]            [k] 0xffffffff814fa210
   1.08%  classify  [unknown]            [k] 0xffffffff814fcb80
Cannot load tips.txt file, please install perf!
```

Original Code

```
Samples: 1K of event 'cache-misses', Event count (approx.): 2827022
Overhead  Command   Shared Object        Symbol
  21.11%  classify  classify             [.] classify
   9.22%  classify  [unknown]            [k] 0xffffffff81df90c0
   9.08%  classify  classify             [.] repeatrun
   6.20%  classify  [unknown]            [k] 0xffffffff814fa219
   4.32%  classify  [unknown]            [k] 0xffffffff814fa285
   3.26%  classify  [unknown]            [k] 0xffffffff812c6c00
   2.87%  classify  [unknown]            [k] 0xffffffff814f4880
   2.83%  classify  [unknown]            [k] 0xffffffff8148bb17
   2.77%  classify  [unknown]            [k] 0xffffffff812c6c18
   2.73%  classify  [unknown]            [k] 0xffffffff818023ee
   2.60%  classify  [unknown]            [k] 0xffffffff814f7924
   2.22%  classify  [unknown]            [k] 0xffffffff814fcab9
   2.19%  classify  [unknown]            [k] 0xffffffff814f780e
   1.89%  classify  [unknown]            [k] 0xffffffff814fa210
   1.80%  classify  [unknown]            [k] 0xffffffff814fa233
   1.70%  classify  [unknown]            [k] 0xffffffff814f7910
   1.51%  classify  [unknown]            [k] 0xffffffff814f48bd
   1.45%  classify  [unknown]            [k] 0xffffffff814be8d6
```

Optimized Code

Perf was used to obtain the cache misses in the original code as well as in the modified code. The initial code was cache-unfriendly due to instances of false-sharing. Instead of giving consequent data reads and writes to different threads, in the modified version, we allocate work on contiguous blocks of data to each thread so that the cache miss rate while reading reduces significantly.

The above screenshots verify that the percentage of cache misses in the classify function reduces in the modified code.

To **further improve** cache-hit rate, we must use perf with more fine-grained flags such as those in the screenshot attached below to analyze the cache performance at different levels(L1, L2, L3) and optimize the code accordingly, also keeping in mind the kind of architecture or hardware being used.

```
L1-dcache-load-misses                              [Hardware cache event]
L1-dcache-loads                                    [Hardware cache event]
L1-dcache-stores                                   [Hardware cache event]
L1-icache-load-misses                              [Hardware cache event]
LLC-load-misses                                    [Hardware cache event]
LLC-loads                                          [Hardware cache event]
LLC-store-misses                                   [Hardware cache event]
LLC-stores                                         [Hardware cache event]
branch-load-misses                                 [Hardware cache event]
branch-loads                                       [Hardware cache event]
dTLB-load-misses                                   [Hardware cache event]
dTLB-loads                                         [Hardware cache event]
dTLB-store-misses                                  [Hardware cache event]
dTLB-stores                                        [Hardware cache event]
iTLB-load-misses                                   [Hardware cache event]
node-load-misses                                   [Hardware cache event]
node-loads                                         [Hardware cache event]
node-store-misses                                  [Hardware cache event]
node-stores                                        [Hardware cache event]
```
