INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 1

APAR AHUJA | ENTRY NO. 2019CS10465

Course - COL380 | Prof. Subodh Kumar

January 16, 2023

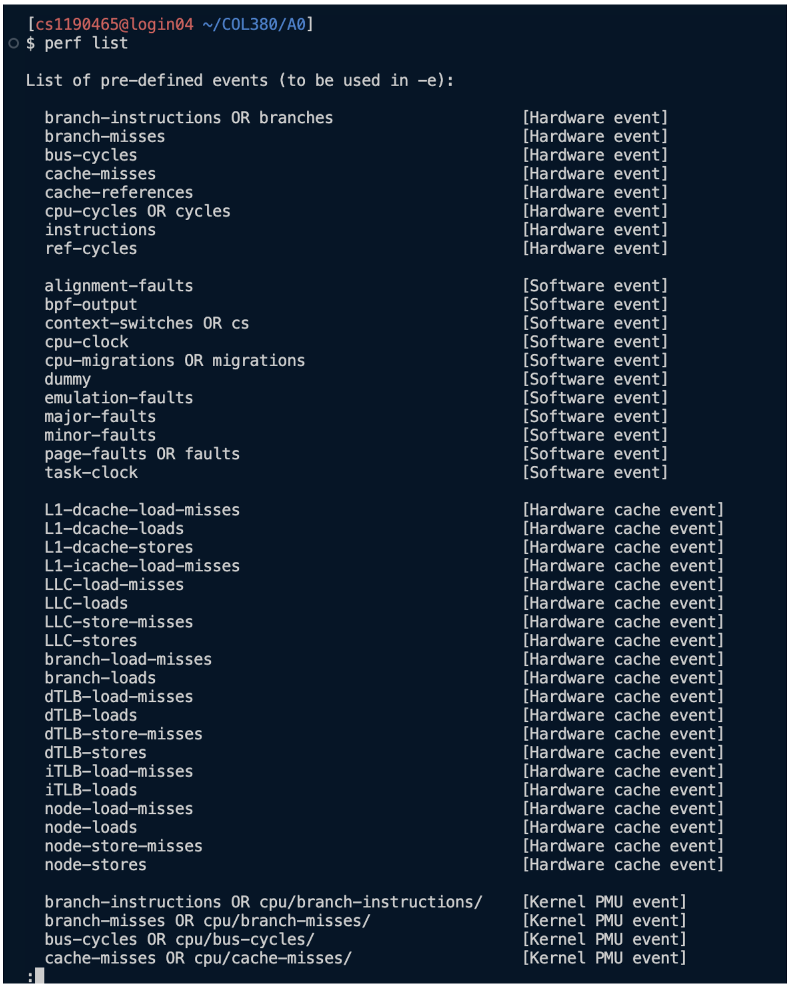
Contents

Chapter 1

Setting up and Running Perf

1.1 Perf List

Command: `perf list`



```
[cs1190465@login04 ~/COL380/A0]
$ perf list

List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  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]
  iTLB-loads                                         [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]

  branch-instructions OR cpu/branch-instructions/    [Kernel PMU event]
  branch-misses OR cpu/branch-misses/                [Kernel PMU event]
  bus-cycles OR cpu/bus-cycles/                      [Kernel PMU event]
  cache-misses OR cpu/cache-misses/                  [Kernel PMU event]
:
```

Figure 1.1: Perf List Sample Output

1.2 Perf Stat

Command: `perf stat ./classify rfile dfile 1009072 4 3`

```
[cs1190465@klogin01 ~/COL380/A0]
$ perf stat ./classify rfile dfile 1009072 4 3
456.781 ms
453.704 ms
448.787 ms
3 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 448.787 ms, Average was 453.09 ms

 Performance counter stats for './classify rfile dfile 1009072 4 3':

         5,456.71 msec task-clock:u              #    3.723 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
            5,699      page-faults:u             #    0.001 M/sec
   17,20,23,87,012      cycles:u                  #    3.153 GHz
   28,25,27,82,880      instructions:u            #    1.64  insn per cycle
   10,09,68,29,154      branches:u                # 1850.352 M/sec
      40,86,81,868      branch-misses:u           #    4.05% of all branches

       1.465623071 seconds time elapsed

       5.442322000 seconds user
       0.023926000 seconds sys
```

Figure 1.2: Perf Stat Sample Output

Number of Threads	Total Time Elapsed	Cycles	Average Time (3 reps)
1	5.3010	16,66,74,75,595	1,731.42
4	1.4549	17,16,50,86,335	451.63
8	0.8651	17,18,02,92,326	254.11
12	0.6283	17,40,36,25,045	174.24
16	0.5786	17,94,03,27,551	156.82
20	0.5413	18,75,56,72,345	144.53
24	0.4878	19,07,78,41,695	128.63
28	0.4845	20,69,83,12,054	123.97
32	0.4696	23,05,11,47,916	122.93
63	0.4570	25,34,55,14,440	116.78

Figure 1.3: Number of Threads vs Total Time, Cycles, Average Runtime (3 repetitions)

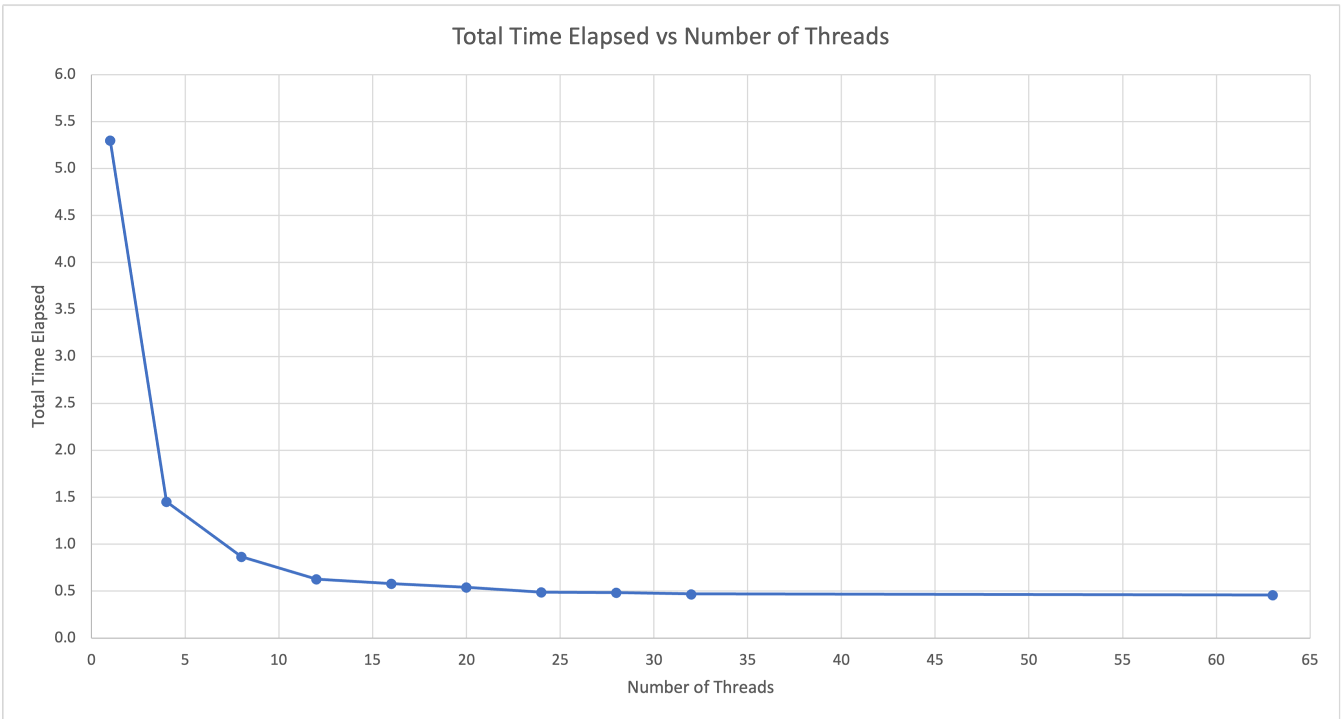


Figure 1.4: Number of Threads vs Total Time (3 repetitions)

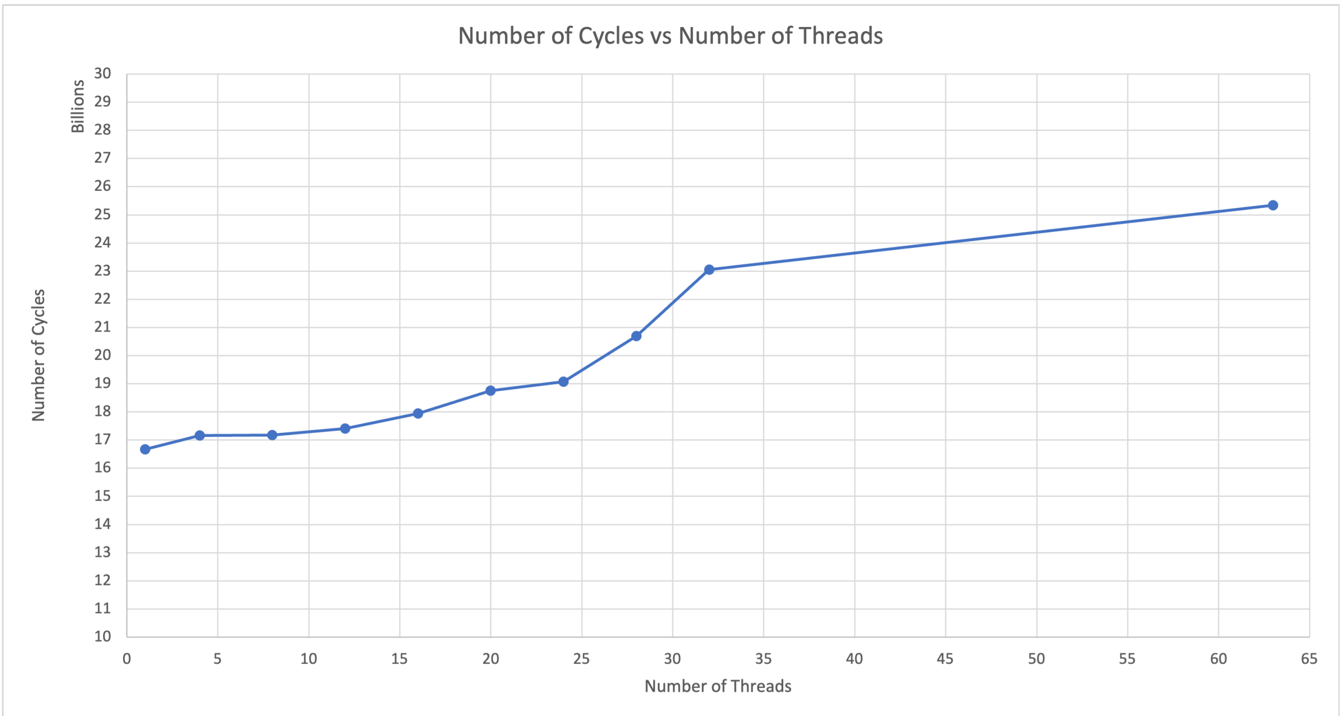


Figure 1.5: Number of Threads vs Cycles (3 repetitions)

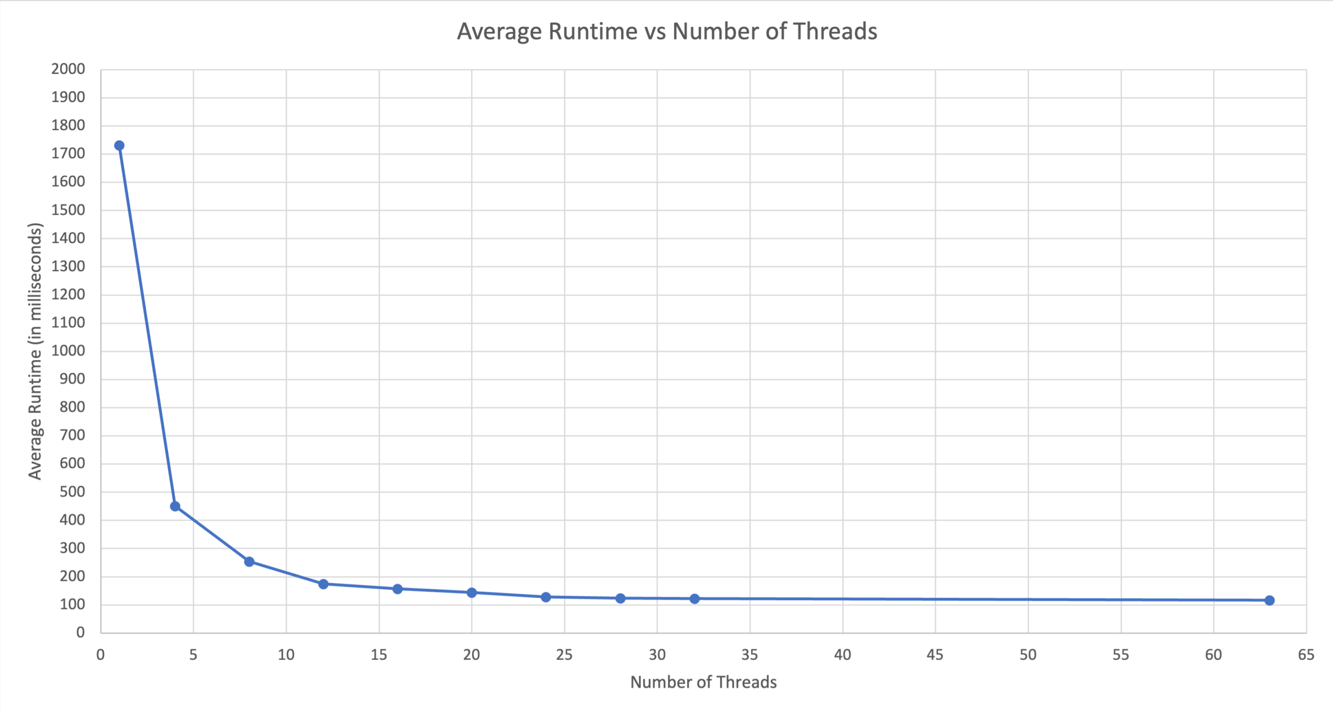


Figure 1.6: Number of Threads vs Average Runtime (3 repetitions)

Question. Do you observe any pattern in these plots?
Answer. The total time elapsed decrease with increase in number of threads. The decrease is not linear. Further, the reduction in time per extra thread is also reducing. The total number of cycles increase with increase in number of threads. Further, the increase in cycles per extra thread is increasing.

Question. If yes, what is the prospective reason for it?
Answer. The total time elapsed reduces with more threads due to parallelization, completing the task faster. However, this decrease in time is not linear as there are other factors. One reason for the decrease in time is the overhead of creating threads. Each thread requires resources to be created. Another reason is that multiple threads are working on the same data, and they may compete for cache and memory.
For the increase in cycles, creating multiple threads can introduce additional overhead, which can consume more cycles. Multiple threads access shared resources, thus, locks may be required, which may consume additional cycles. Further, some threads may finish their work before others, leading to idle time and increased cycles. The increase in cycles per extra thread is increasing, could be due to the fact that the amount of work that can be done in parallel decreases.

1.3 Perf Record

1.3.1 A1.

Command: `perf record ./classify rfile dfile 1009072 4 10`

```
[cs1190465@klogin01 ~/COL380/A0]
$ perf record ./classify rfile dfile 1009072 4 10
496.426 ms
484.311 ms
481.211 ms
479.952 ms
487.497 ms
482.902 ms
483.296 ms
484.292 ms
472.152 ms
481.142 ms
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 472.152 ms, Average was 483.318 ms
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 2.850 MB perf.data (74644 samples) ]
```

Figure 1.7: Perf Record Sample Output

1.3.2 A2.

Command: `perf report`

```
Samples: 74K of event 'cycles:uppp', Event count (approx.): 57376024080
Overhead  Command   Shared Object        Symbol
  71.92%  classify  classify             [.] classify
  26.30%  classify  classify             [.] classify
   0.39%  classify  [unknown]            [k] 0xffffffff9c38a4ef
   0.26%  classify  libstdc++.so.6.0.19  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
   0.24%  classify  libgomp.so.1.0.0     [.] 0x0000000000018b1f
   0.19%  classify  libgomp.so.1.0.0     [.] 0x0000000000018c97
   0.13%  classify  libgomp.so.1.0.0     [.] 0x0000000000018b21
   0.09%  classify  libgomp.so.1.0.0     [.] 0x0000000000018c99
   0.09%  classify  classify             [.] repeatrun
   0.07%  classify  libgomp.so.1.0.0     [.] 0x0000000000018b10
   0.05%  classify  libstdc++.so.6.0.19  [.] std::istream::sentry::sentry
   0.05%  classify  libgomp.so.1.0.0     [.] 0x0000000000018c88
   0.03%  classify  [unknown]            [k] 0xffffffff9c394098
   0.02%  classify  libstdc++.so.6.0.19  [.] std::istream::operator>>
   0.02%  classify  classify             [.] readRanges
   0.02%  classify  libgomp.so.1.0.0     [.] 0x0000000000018b13
   0.02%  classify  libgomp.so.1.0.0     [.] 0x0000000000018c8b
   0.02%  classify  libstdc++.so.6.0.19  [.] 0x000000000008c365
   0.01%  classify  libstdc++.so.6.0.19  [.] 0x000000000008c357
   0.01%  classify  libstdc++.so.6.0.19  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::do_get
   0.01%  classify  classify             [.] readData
   0.01%  classify  classify             [.] classify
   0.01%  classify  libstdc++.so.6.0.19  [.] std::locale::id::_M_id
   0.00%  classify  libstdc++.so.6.0.19  [.] 0x000000000008c3a1
   0.00%  classify  libstdc++.so.6.0.19  [.] 0x000000000008c3ed
   0.00%  classify  libstdc++.so.6.0.19  [.] std::string::_Rep::_M_dispose@plt
   0.00%  classify  libstdc++.so.6.0.19  [.] 0x000000000008c3bf
   0.00%  classify  libstdc++.so.6.0.19  [.] 0x000000000008c3e0
   0.00%  classify  libstdc++.so.6.0.19  [.] 0x000000000008bb0a
   0.00%  classify  libc-2.17.so         [.] malloc
   0.00%  classify  ld-2.17.so           [.] _dl_relocate_object
   0.00%  classify  classify             [.] std::istream::operator>>@plt
```

Figure 1.8: Perf Report Sample Output

Command: `perf annotate`

```
Samples: 74K of event 'cycles:uppp', 4000 Hz, Event count (approx.): 57602888726
classify  /home/cse/btech/cs1190465/COL380/A0/classify [Percent: local period]
Percent│       nop
  0.04 │       mov    0x8(%rbx),%esi
       │       mov    %eax,%r9d
       │       cltq
       │       lea    (%r10,%rax,8),%r8
  0.04 │       mov    (%rbx),%rcx
       │       test   %esi,%esi
  0.08 │       mov    (%r8),%edx
       │     → jle    402068 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xa8>
  0.05 │       xor    %eax,%eax
       │     → jmp    402020 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x60>
       │       nop
 15.02 │       add    $0x1,%rax
  7.66 │       cmp    %eax,%esi
       │     → jle    402068 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xa8>
 21.16 │       cmp    (%rcx,%rax,8),%edx
 23.69 │     → jl     402018 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x58>
  0.16 │       cmp    0x4(%rcx,%rax,8),%edx
 26.17 │     → jg     402018 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x58>
  0.28 │       mov    %eax,0x4(%r8)
       │       cltq
  0.00 │       shl    $0x6,%rax
       │       add    %r13,%rax
  0.35 │       cmp    %edi,0x8(%rax)
  0.08 │       mov    (%rax),%rdx
  0.00 │     → jbe    40206c <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xac>
  0.01 │       lea    (%rdx,%r11,1),%rax
  5.15 │       mov    (%rax),%edx
  0.02 │       add    $0x1,%edx
  0.03 │       mov    %edx,(%rax)
       │       mov    0x18(%rbp),%eax
  0.00 │       add    %r9d,%eax
```

Figure 1.9: Perf Annotate Sample Output

1.3.3 A3.

The command "`jg 402018 <classify(Data, Ranges const, unsigned int) [clone ._omp_fn.0]+0x58>`" takes **26.17%** of the time, the most time among all instructions.

1.3.4 A4.

The command corresponds to `val <= hi` in the code. It's optimised by the compiler to return false when `val > hi` (hence the jump if greater instruction).

```
  bool within(int val) const { // Return if val is within this range
      return(lo <= val && val <= hi); // original
  }
```

Figure 1.10: Source Code

1.3.5 A5.

Tweak: `CFLAGS=-std=c++11 -O2 -fopenmp -g`

Chapter 2

Hotspot Analysis

2.1 A1.

Tweak: `CFLAGS=-std=c++11 -O2 -fopenmp -g`
Command: `make clean; make; perf record ./classify rfile dfile 1009072 4 10`

```
[cs1190465@klogin01 ~/COL380/A0]
$ perf record ./classify rfile dfile 1009072 4 10
460.507 ms
454.193 ms
448.633 ms
459.133 ms
451.419 ms
457.869 ms
465.869 ms
461.163 ms
455.05 ms
456.868 ms
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 448.633 ms, Average was 457.071 ms
[ perf record: Woken up 9 times to write data ]
[ perf record: Captured and wrote 2.760 MB perf.data (72282 samples) ]
```

Figure 2.1: Perf Record with Symbols Sample Output

2.2 A2.

The top hotspot is the function call `Range::within` which takes $20.96+23.64+0.13+26.32 = $ **71.05%** of the time. You can see the jump instructions below the `return(lo <= val && val <= hi)` statement in the figure 2.2, attached below.

```
       │    → jle    402068 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xa8>
       │    _ZNK5Range6withinEi():
       │              return(lo <= val && val <= hi); // original
 20.96 │      cmp    (%rcx,%rax,8),%edx
 23.64 │    → jl     402018 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x58>
  0.13 │      cmp    0x4(%rcx,%rax,8),%edx
 26.32 │    → jg     402018 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x58>
       │    _Z8classifyR4DataRK6Rangesj._omp_fn.0():
  0.31 │      mov    %eax,0x4(%r8)
```

Figure 2.2: Perf Report with Symbols Sample Output

2.3 A3.

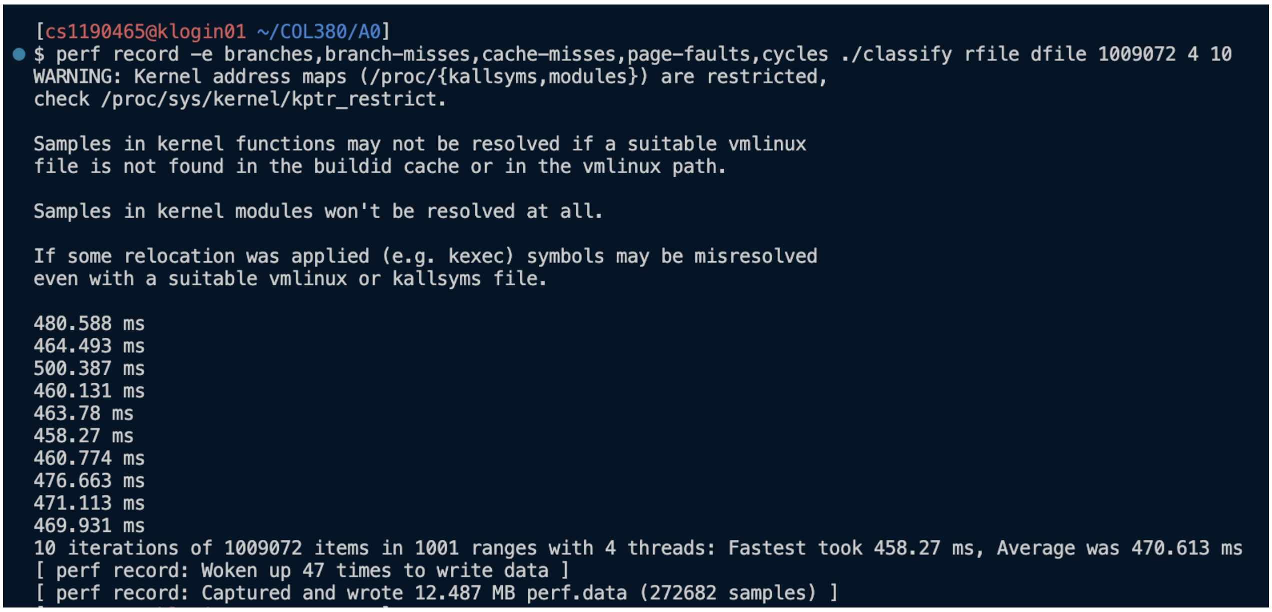
The function itself is small and can't be optimised further. Thus, it became a hotspot because it is called too many times. It is called for each data value in the code to linearly find over all ranges. The number of ranges is 1000. The number of data values are 1009072. Hence, this function is called a total of around 10^9 times.

2.4 A4.

The function itself can't be optimised further. However, the number of calls to function can be reduced by tweaking the search algorithm. Sorting the ranges with respect to start point can improve the number of calls from `O(n)` to `O(logn)`. Further, we can use a hash table and bucket sort to count.

2.5 A5.

Command: `perf record -e branches,branch-misses,cache-misses,page-faults,cycles ./classify rfile dfile 1009072 4 10`



```
[cs1190465@klogin01 ~/COL380/A0]
$ perf record -e branches,branch-misses,cache-misses,page-faults,cycles ./classify rfile dfile 1009072 4 10
WARNING: Kernel address maps (/proc/{kallsyms,modules}) are restricted,
check /proc/sys/kernel/kptr_restrict.

Samples in kernel functions may not be resolved if a suitable vmlinux
file is not found in the buildid cache or in the vmlinux path.

Samples in kernel modules won't be resolved at all.

If some relocation was applied (e.g. kexec) symbols may be misresolved
even with a suitable vmlinux or kallsyms file.

480.588 ms
464.493 ms
500.387 ms
460.131 ms
463.78 ms
458.27 ms
460.774 ms
476.663 ms
471.113 ms
469.931 ms
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 458.27 ms, Average was 470.613 ms
[ perf record: Woken up 47 times to write data ]
[ perf record: Captured and wrote 12.487 MB perf.data (272682 samples) ]
```

Figure 2.3: Perf Record with extra events Sample Output

Chapter 3

Memory Profiling

3.1 A1.

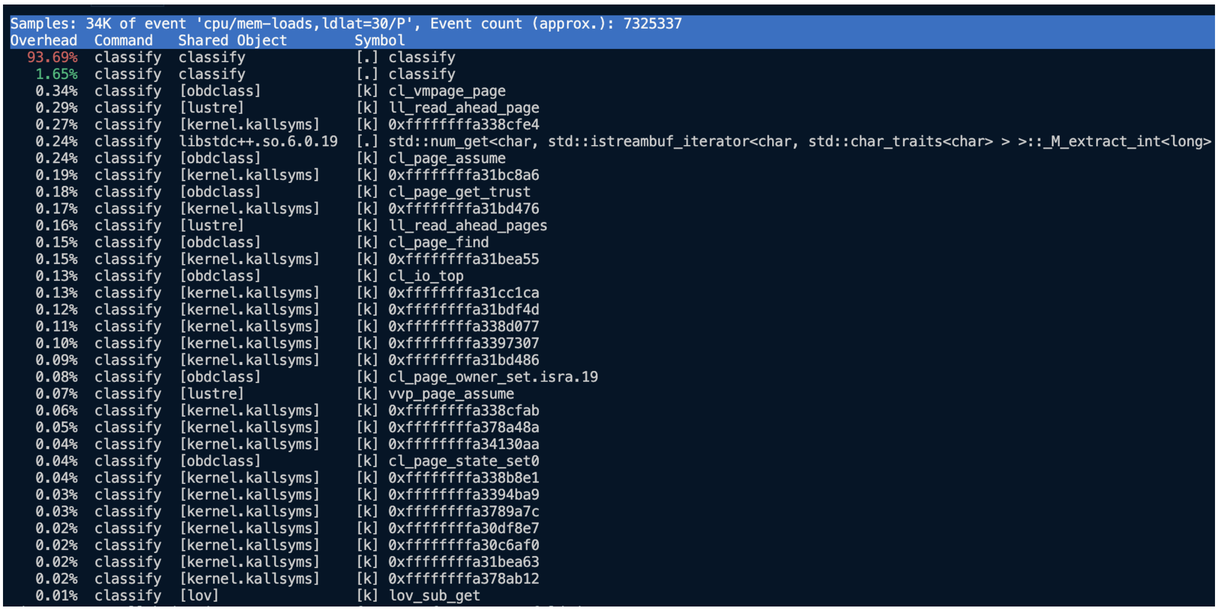
Command: `perf mem record ./classify rfile dfile 1009072 4 10`

```
[cs1190465@klogin01 ~/COL380/A0]
$ perf mem record ./classify rfile dfile 1009072 4 10
WARNING: Kernel address maps (/proc/{kallsyms,modules}) are restricted,
check /proc/sys/kernel/kptr_restrict.

Samples in kernel functions may not be resolved if a suitable vmlinux
file is not found in the buildid cache or in the vmlinux path.

Samples in kernel modules won't be resolved at all.

If some relocation was applied (e.g. kexec) symbols may be misresolved
even with a suitable vmlinux or kallsyms file.

517.676 ms
487.738 ms
494.071 ms
481.327 ms
485.553 ms
478.129 ms
468.218 ms
461.217 ms
485.801 ms
496.842 ms
10 iterations of 1009072 items in 1001 ranges with 4 threads: Fastest took 461.217 ms, Average was 485.657 ms
[ perf record: Woken up 19 times to write data ]
[ perf record: Captured and wrote 4.745 MB perf.data (68903 samples) ]
```

Figure 3.1: perf mem record Sample Output

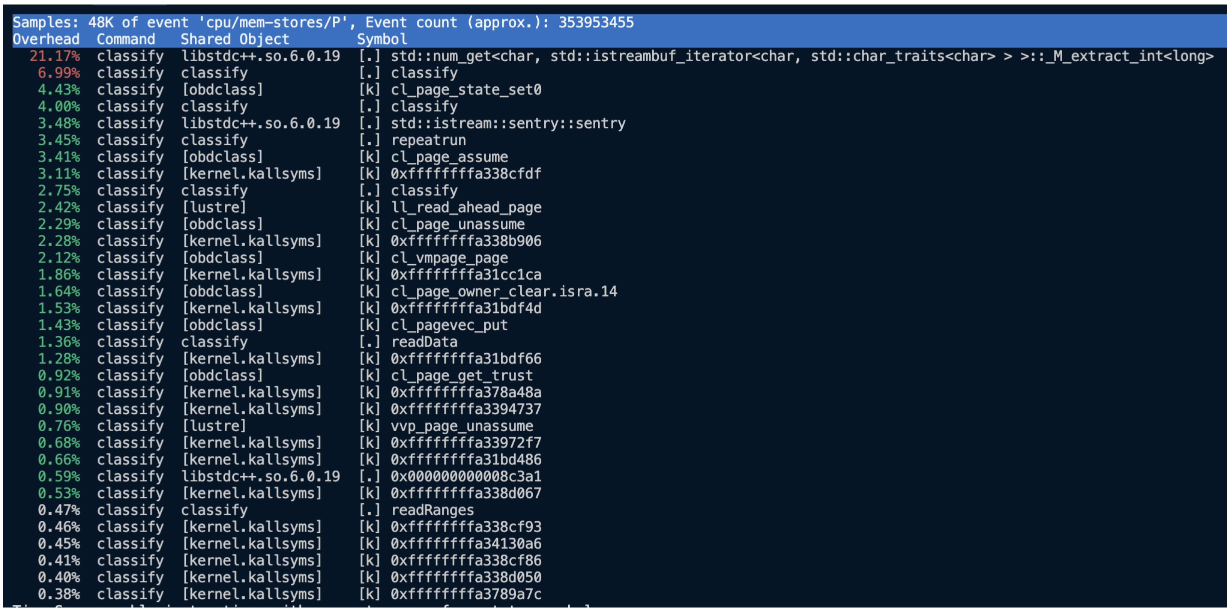
```
OUTPUT   TERMINAL   PORTS   DEBUG CONSOLE   PROBLEMS                perf
Available samples
34K cpu/mem-loads,ldlat=30/P
48K cpu/mem-stores/P
```

Figure 3.2: perf mem report Sample Output



```
Samples: 34K of event 'cpu/mem-loads,ldlat=30/P', Event count (approx.): 7325337
Overhead  Command   Shared Object         Symbol
  93.69%  classify  classify              [.] classify
   1.65%  classify  classify              [.] classify
   0.34%  classify  [obdclass]            [k] cl_vmpage_page
   0.29%  classify  [lustre]              [k] ll_read_ahead_page
   0.27%  classify  [kernel.kallsyms]     [k] 0xffffffffa338cfe4
   0.24%  classify  libstdc++.so.6.0.19   [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
   0.24%  classify  [obdclass]            [k] cl_page_assume
   0.19%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bc8a6
   0.18%  classify  [obdclass]            [k] cl_page_get_trust
   0.17%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bd476
   0.16%  classify  [lustre]              [k] ll_read_ahead_pages
   0.15%  classify  [obdclass]            [k] cl_page_find
   0.15%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bea55
   0.13%  classify  [obdclass]            [k] cl_io_top
   0.13%  classify  [kernel.kallsyms]     [k] 0xffffffffa31cc1ca
   0.12%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bdf4d
   0.11%  classify  [kernel.kallsyms]     [k] 0xffffffffa338d077
   0.10%  classify  [kernel.kallsyms]     [k] 0xffffffffa3397307
   0.09%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bd486
   0.08%  classify  [obdclass]            [k] cl_page_owner_set.isra.19
   0.07%  classify  [lustre]              [k] vvp_page_assume
   0.06%  classify  [kernel.kallsyms]     [k] 0xffffffffa338cfab
   0.05%  classify  [kernel.kallsyms]     [k] 0xffffffffa378a48a
   0.04%  classify  [kernel.kallsyms]     [k] 0xffffffffa34130aa
   0.04%  classify  [obdclass]            [k] cl_page_state_set0
   0.04%  classify  [kernel.kallsyms]     [k] 0xffffffffa338b8e1
   0.03%  classify  [kernel.kallsyms]     [k] 0xffffffffa3394ba9
   0.03%  classify  [kernel.kallsyms]     [k] 0xffffffffa3789a7c
   0.02%  classify  [kernel.kallsyms]     [k] 0xffffffffa30df8e7
   0.02%  classify  [kernel.kallsyms]     [k] 0xffffffffa30c6af0
   0.02%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bea63
   0.02%  classify  [kernel.kallsyms]     [k] 0xffffffffa378ab12
   0.01%  classify  [lov]                 [k] lov_sub_get
```

Figure 3.3: perf report for Memory Loads Profile Sample Output



```
Samples: 48K of event 'cpu/mem-stores/P', Event count (approx.): 353953455
Overhead  Command   Shared Object         Symbol
  21.17%  classify  libstdc++.so.6.0.19   [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
   6.99%  classify  classify              [.] classify
   4.43%  classify  [obdclass]            [k] cl_page_state_set0
   4.00%  classify  classify              [.] classify
   3.48%  classify  libstdc++.so.6.0.19   [.] std::istream::sentry::sentry
   3.45%  classify  classify              [.] repeatrun
   3.41%  classify  [obdclass]            [k] cl_page_assume
   3.11%  classify  [kernel.kallsyms]     [k] 0xffffffffa338cfdf
   2.75%  classify  classify              [.] classify
   2.42%  classify  [lustre]              [k] ll_read_ahead_page
   2.29%  classify  [obdclass]            [k] cl_page_unassume
   2.28%  classify  [kernel.kallsyms]     [k] 0xffffffffa338b906
   2.12%  classify  [obdclass]            [k] cl_vmpage_page
   1.86%  classify  [kernel.kallsyms]     [k] 0xffffffffa31cc1ca
   1.64%  classify  [obdclass]            [k] cl_page_owner_clear.isra.14
   1.53%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bdf4d
   1.43%  classify  [obdclass]            [k] cl_pagevec_put
   1.36%  classify  classify              [.] readData
   1.28%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bdf66
   0.92%  classify  [obdclass]            [k] cl_page_get_trust
   0.91%  classify  [kernel.kallsyms]     [k] 0xffffffffa378a48a
   0.90%  classify  [kernel.kallsyms]     [k] 0xffffffffa3394737
   0.76%  classify  [lustre]              [k] vvp_page_unassume
   0.68%  classify  [kernel.kallsyms]     [k] 0xffffffffa33972f7
   0.66%  classify  [kernel.kallsyms]     [k] 0xffffffffa31bd486
   0.59%  classify  libstdc++.so.6.0.19   [.] 0x000000000008c3a1
   0.53%  classify  [kernel.kallsyms]     [k] 0xffffffffa338d067
   0.47%  classify  classify              [.] readRanges
   0.46%  classify  [kernel.kallsyms]     [k] 0xffffffffa338cf93
   0.45%  classify  [kernel.kallsyms]     [k] 0xffffffffa34130a6
   0.41%  classify  [kernel.kallsyms]     [k] 0xffffffffa338cf86
   0.40%  classify  [kernel.kallsyms]     [k] 0xffffffffa338d050
   0.38%  classify  [kernel.kallsyms]     [k] 0xffffffffa3789a7c
```

Figure 3.4: perf report for Memory Stores Profile Sample Output

3.2 A2.

Hotspot 1:

```
int tid = omp_get_thread_num(); // Original
for(int r=tid; r<R.num(); r+=numt) { // Thread together share-loop
.   int rcount = 0;
.   for(int d=0; d<D.ndata; d++) // For each interval, thread loops
.      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 D2.
}
```

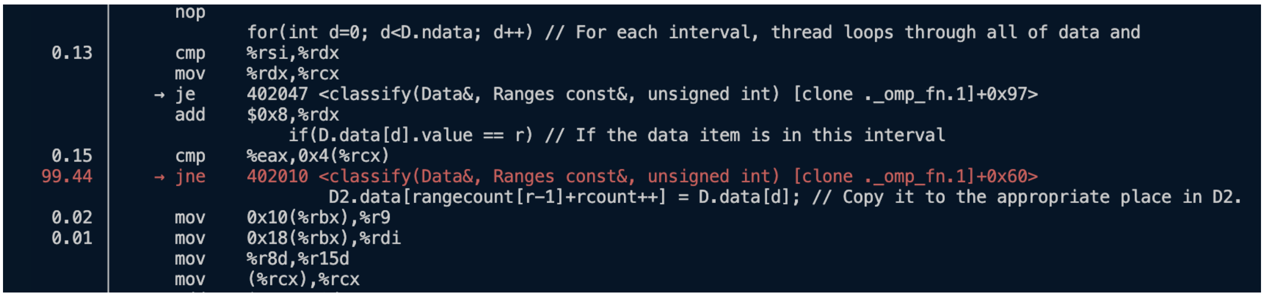


Figure 3.5: Hotspot 1 mem-loads (the second loop where we set D2)

Hotspot 2:

```
int tid = omp_get_thread_num(); // Original
for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through 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.
.   counts[v].increase(tid); // Found one key in interval v
}
```

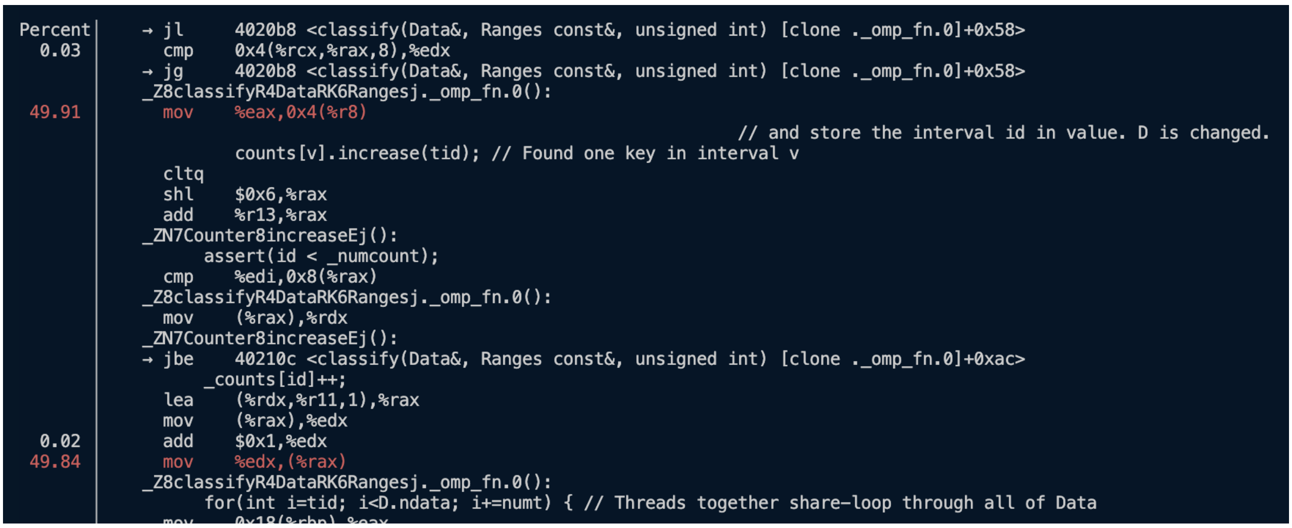


Figure 3.6: Hotspot 2 mem-stores (the first loop where we use counter)

3.3 A3.

1. False Sharing - 1: The code uses a counter for each interval, with one counter per thread. This means that if multiple threads are working on different intervals that are close to each other in memory, they may be accessing the same cache line, causing false sharing. This can decrease the performance. Adding padding to counters and change `alignas(32)` to the cache line width can improve the performance.
2. False Sharing - 2: The current implementation's data access by each thread is not continuous, which results in cache misses and poor cache utilization. We can divide the data into numt blocks and assign each block to a thread so that each thread accesses continuous data, resulting in a cache-friendly memory access.
3. Nested For Loop: The code uses a nested loop to sort the data by interval. Within the inner loop, the code checks each item in the data set to see if it belongs in the current interval. We note that the number of ranges is small around 1000 but the number of data values is 1000000, which is much larger. Parallel processing over the data points rather than the number of ranges should perform better and reduce cache misses. Further, the double loop is not necessary and can even be optimized to just a single loop at the cost of a vector `rcounts[R.num]`.

3.4 A4.

Improvements:
1. Counter Alignas: Prof. Subodh suggested not to go ahead with padding and changing alignas. This optimisation was not implemented in the final code.
2. Continuous Thread Blocks: The blocks of data accessed by threads is made continuous blocks by using `for(int i=tid*block_size; i<(tid+1)*block_size && i<D.ndata; i++)`
The average time improved from **566ms** to **530ms**.

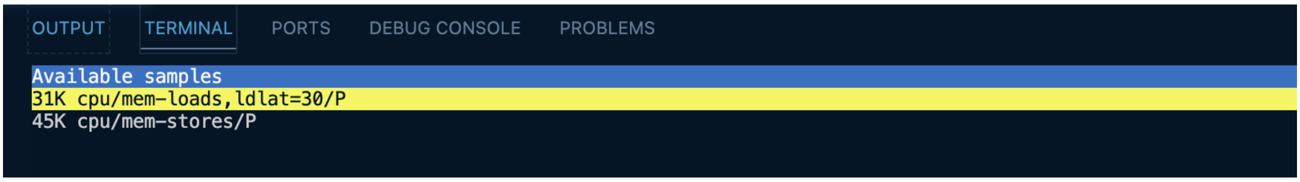


Figure 3.7: After Improvement 2 only

```
Samples: 31K of event 'cpu/mem-loads,ldlat=30/P', 4000 Hz, Event count (approx.): 6704386
classify  /home/cse/btech/cs1190465/COL380/A0/classify [Percent: local period]
Percent │          //             D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
        │      movslq %eax,%rdx
        │      xor    %r8d,%r8d
        │      lea    -0x4(,%rdx,4),%r10
        │      lea    0x8(%rcx),%rdx
        │      lea    (%rdx,%r14,1),%rsi
        │    → jmp    401f7c <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x6c>
        │      nop
        │          //      for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
        │      cmp    %rsi,%rdx
        │      mov    %rdx,%rcx
        │    → je     401fa7 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x97>
        │      add    $0x8,%rdx
        │          //          if(D.data[d].value == r) // If the data item is in this interval
   0.11 │      cmp    %eax,0x4(%rcx)
  99.84 │    → jne    401f70 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.1]+0x60>
        │          //             D2.data[rangecount[r-1]+rcount++] = D.data[d]; // Copy it to the appropriate place in D2.
   0.02 │      mov    0x10(%rbx),%r9
   0.00 │      mov    0x18(%rbx),%rdi
        │      mov    %r8d,%r15d
        │      mov    (%rcx),%rcx
        │      add    $0x1,%r8d
   0.01 │      add    (%r9,%r10,1),%r15d
   0.01 │      mov    0x8(%rdi),%rdi
        │          //      for(int d=0; d<D.ndata; d++) // For each interval, thread loops through all of data and
        │      cmp    %rsi,%rdx
```

Figure 3.8: Improvement 2 only - Hotspot 1

```
Samples: 45K of event 'cpu/mem-stores/P', 4000 Hz, Event count (approx.): 202164736
classify  /home/cse/btech/cs1190465/COL380/A0/classify [Percent: local period]
   0.03 │      cmp    0x4(%rsi,%rdx,8),%ecx
   0.01 │    → jg     402030 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0x70>
        │    _Z8classifyR4DataRK6Rangesj._omp_fn.0():
        │             int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
  51.61 │      mov    %edx,0x4(%r9)
        │                                                                  // and store the interval id in value. D is changed.
        │             counts[v].increase(tid); // Found one key in interval v
        │      movslq %edx,%rdx
        │      shl    $0x6,%rdx
        │      add    %r12,%rdx
        │    _ZN7Counter8increaseEj():
        │          assert(id < _numcount);
   0.08 │      cmp    %r10d,0x8(%rdx)
        │    _Z8classifyR4DataRK6Rangesj._omp_fn.0():
   0.07 │      mov    (%rdx),%rcx
        │    _ZN7Counter8increaseEj():
   0.00 │    → jbe    40208c <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xcc>
        │          _counts[id]++;
        │      lea    (%rcx,%rax,1),%rdx
        │    _Z8classifyR4DataRK6Rangesj._omp_fn.0():
        │          for(int i=tid*block; i<(tid+1)*block && i<D.ndata; i++) { // Threads together share-loop through all of Data
        │      add    $0x1,%r8d
        │    _ZN7Counter8increaseEj():
   0.82 │      mov    (%rdx),%ecx
   0.09 │      add    $0x1,%ecx
        │    _Z8classifyR4DataRK6Rangesj._omp_fn.0():
        │      cmp    %r11d,%r8d
        │    _ZN7Counter8increaseEj():
  47.17 │      mov    %ecx,(%rdx)
        │    _Z8classifyR4DataRK6Rangesj._omp_fn.0():
        │    → je     402078 <classify(Data&, Ranges const&, unsigned int) [clone ._omp_fn.0]+0xb8>
```

Figure 3.9: Improvement 2 only - Hotspot 2

3. Optimized Nested Loop: By interchanging the order of the loops, the outer loop iterates through the data items and the inner loop iterates through the intervals. Because the outer loop is now iterating over the large dataset (D.ndata) which is much larger than the number of intervals (R.num), so parallelizing the outer loop leads to a more effective use of threading.
We can also change the loop to a single loop as shown below -

```
int tid = omp_get_thread_num();
int block = D.ndata/numt;
for(int d=tid*block; d<(1+tid)*block && d<D.ndata; d++){
.   int r = D.data[d].value; // If the data item is in this interval
.   D2.data[rangecount[r-1]+rcount[r]++] = D.data[d]; // Copy it to D2.
}
```

The average time improved from **566ms** to **555ms**.

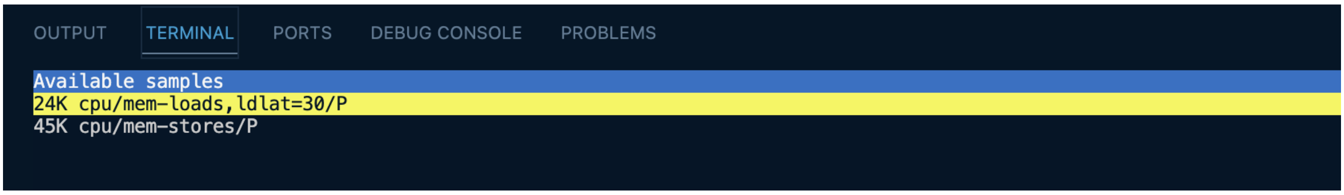


Figure 3.10: After Improvement 3 only

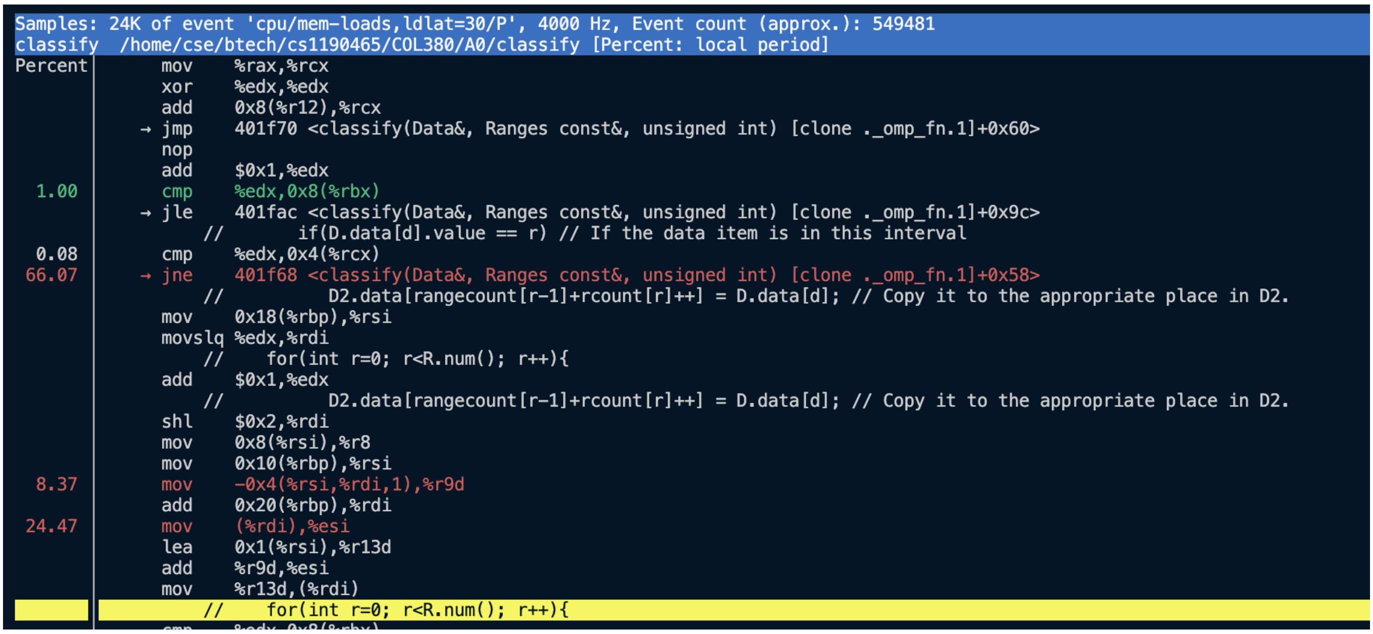


Figure 3.11: Improvement 3 only - Hotspot 1

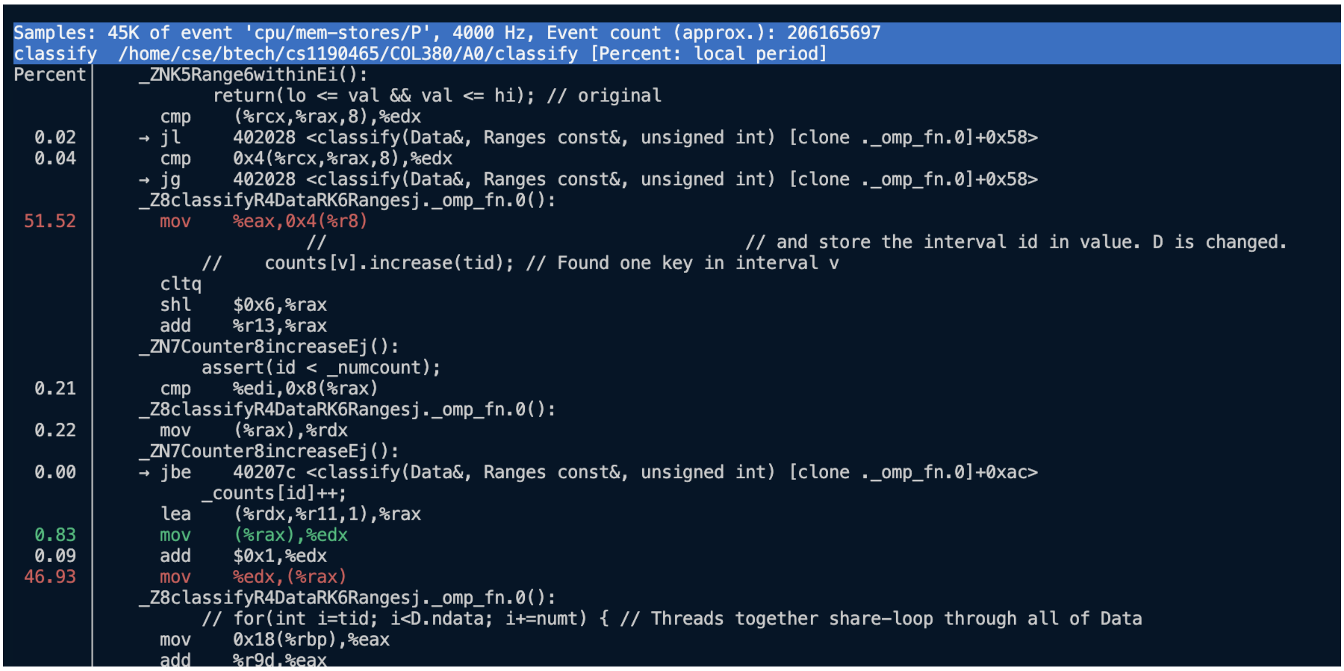


Figure 3.12: Improvement 3 only - Hotspot 2

Putting both improvements together reduced the time from **566ms** to **476ms**.

Figure 3.13: After Both Improvements Together

3.5 A5.

There is a small improvement in the cache hit rate, from 56K events to 54K events.
To improve it we can try the following suggestions –
1. New Improvements: Either add vectorization, Use hash-tables in algorithm, or Increase / Decrease alignas (cache-line-width) by a few times.
2. Implemented: Threads work on continuous chunks of memory to provide cache locality. Reorder the nested loops in the parallel section, so that the inner loop iterates over a smaller range of memory. By interchanging the order of the loops, the outer loop iterates over all data and the inner loop iterates over intervals, which results in better cache locality.