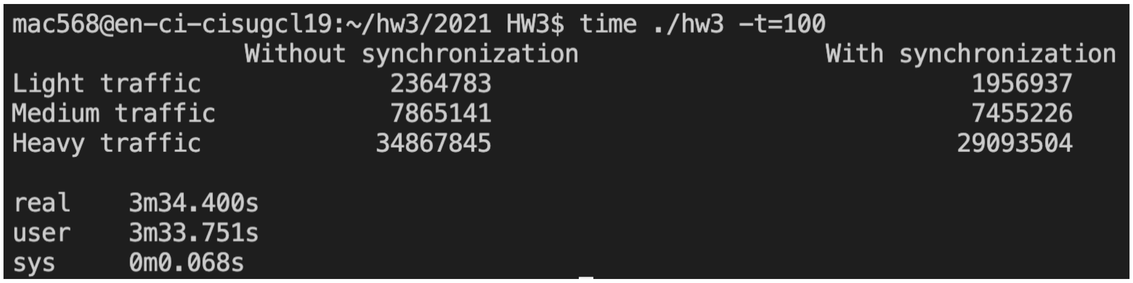
Performance Analysis of Traffic Project

Step 1: Performance Costs

a. First, compile with the flag `-g`

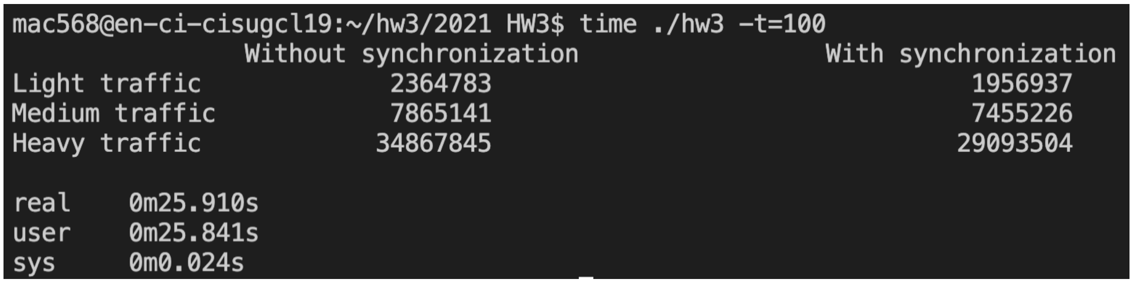
Time it takes running with debugger information:



```
mac568@en-ci-cisugcl19:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic             2364783                                 1956937
Medium traffic            7865141                                 7455226
Heavy traffic            34867845                                29093504

real    3m34.400s
user    3m33.751s
sys     0m0.068s
```

b. Next, compile with flags `-O3` and `-pg`

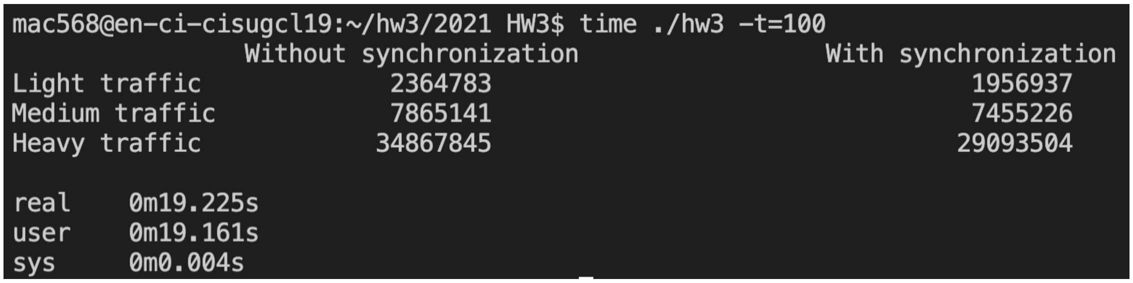
Time it takes running with optimization and profiling, but with no debugger information:



```
mac568@en-ci-cisugcl19:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic             2364783                                 1956937
Medium traffic            7865141                                 7455226
Heavy traffic            34867845                                29093504

real    0m25.910s
user    0m25.841s
sys     0m0.024s
```

c. Finally, compile with `-O3` and nothing else

Time it takes running with optimization, but with no profiling and no debugger information:



```
mac568@en-ci-cisugcl19:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic             2364783                                 1956937
Medium traffic            7865141                                 7455226
Heavy traffic            34867845                                29093504

real    0m19.225s
user    0m19.161s
sys     0m0.004s
```

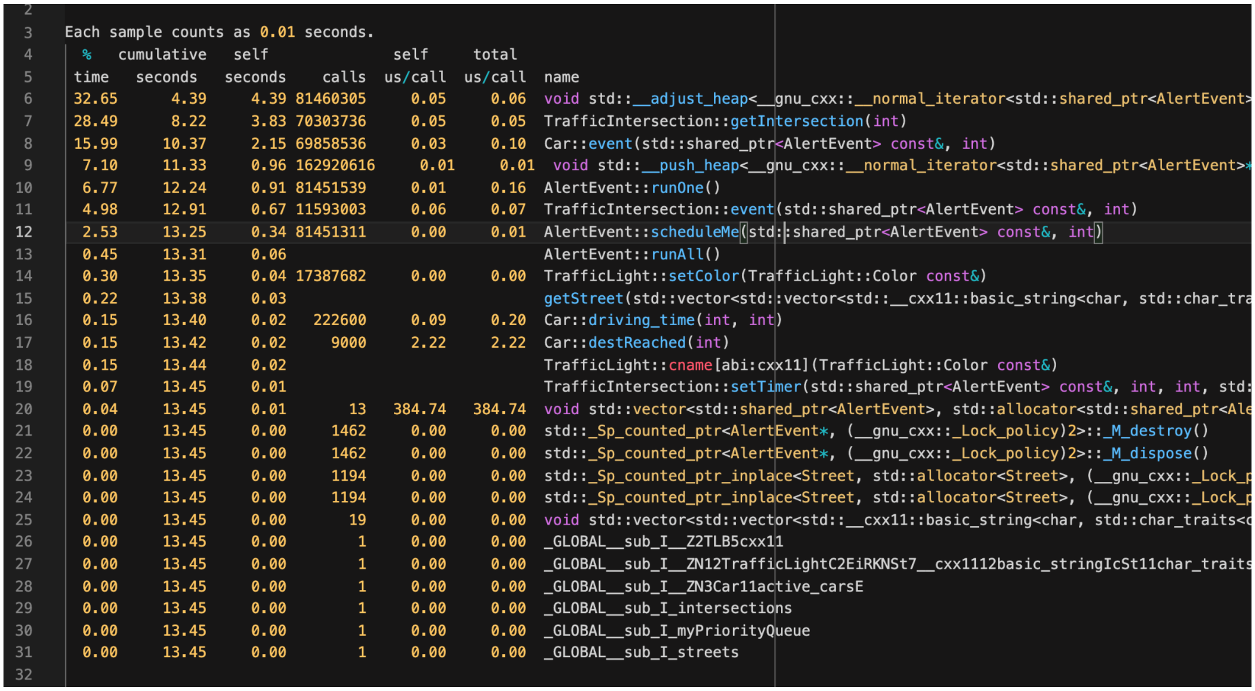
Discussion of Step 1:

We can see that compiling with just the `-g` flag takes the most time; it takes approximately three and a half minutes to finish running. The `-g` flag tells the program to produce debugging information for GDB when it is executed. Producing this extra information has a lot of overhead, so hence, this way of running takes a lot of time.

The second longest running time (but much faster than before) comes with optimizing *and* profiling, but *not* producing debugging information; when we compile with the `-pg` flag and the `-O3` flag, we get about 26 seconds of running time. However, when we just optimize and do nothing else (using just the `-O3` flag), we get about 19 seconds of running time. This makes sense because the `-pg` flag writes extra code along with the program to write profile information that the coder can analyze. However, when you do not request the program to do any profiling, you avoid the overhead that would go into generating the extra profiling code.

Running with *only* the `-O3` flag clearly runs the fastest. This makes sense because the program makes use of all optimizations. Also with this flag alone, the program doesn't waste overhead generating any debugging information or generating any profile-writing code.

Step 2: Profiling and Performance Analysis



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 32.65      4.39     4.39 81460305     0.05     0.06  void std::__adjust_heap<__gnu_cxx::__normal_iterator<std::shared_ptr<AlertEvent>
 28.49      8.22     3.83 70303736     0.05     0.05  TrafficIntersection::getIntersection(int)
 15.99     10.37     2.15 69858536     0.03     0.10  Car::event(std::shared_ptr<AlertEvent> const&, int)
  7.10     11.33     0.96 162920616     0.01     0.01  void std::__push_heap<__gnu_cxx::__normal_iterator<std::shared_ptr<AlertEvent>*
  6.77     12.24     0.91 81451539     0.01     0.16  AlertEvent::runOne()
  4.98     12.91     0.67 11593003     0.06     0.07  TrafficIntersection::event(std::shared_ptr<AlertEvent> const&, int)
  2.53     13.25     0.34 81451311     0.00     0.01  AlertEvent::scheduleMe(std::shared_ptr<AlertEvent> const&, int)
  0.45     13.31     0.06                             AlertEvent::runAll()
  0.30     13.35     0.04 17387682     0.00     0.00  TrafficLight::setColor(TrafficLight::Color const&)
  0.22     13.38     0.03                             getStreet(std::vector<std::vector<std::__cxx11::basic_string<char, std::char_tra
  0.15     13.40     0.02   222600     0.09     0.20  Car::driving_time(int, int)
  0.15     13.42     0.02     9000     2.22     2.22  Car::destReached(int)
  0.15     13.44     0.02                             TrafficLight::cname[abi:cxx11](TrafficLight::Color const&)
  0.07     13.45     0.01                             TrafficIntersection::setTimer(std::shared_ptr<AlertEvent> const&, int, int, std:
  0.04     13.45     0.01       13   384.74   384.74  void std::vector<std::shared_ptr<AlertEvent>, std::allocator<std::shared_ptr<Ale
  0.00     13.45     0.00     1462     0.00     0.00  std::_Sp_counted_ptr<AlertEvent*, (__gnu_cxx::_Lock_policy)2>::_M_destroy()
  0.00     13.45     0.00     1462     0.00     0.00  std::_Sp_counted_ptr<AlertEvent*, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
  0.00     13.45     0.00     1194     0.00     0.00  std::_Sp_counted_ptr_inplace<Street, std::allocator<Street>, (__gnu_cxx::_Lock_p
  0.00     13.45     0.00     1194     0.00     0.00  std::_Sp_counted_ptr_inplace<Street, std::allocator<Street>, (__gnu_cxx::_Lock_p
  0.00     13.45     0.00       19     0.00     0.00  void std::vector<std::vector<std::__cxx11::basic_string<char, std::char_traits<c
  0.00     13.45     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z2TLB5cxx11
  0.00     13.45     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN12TrafficLightC2EiRKNSt7__cxx1112basic_stringIcSt11char_traits
  0.00     13.45     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN3Car11active_carsE
  0.00     13.45     0.00        1     0.00     0.00  _GLOBAL__sub_I_intersections
  0.00     13.45     0.00        1     0.00     0.00  _GLOBAL__sub_I_myPriorityQueue
  0.00     13.45     0.00        1     0.00     0.00  _GLOBAL__sub_I_streets
```

Which methods are costing the most CPU time?

Above is the `gprof` output page. We can see that there are a few methods that take up a large percentage of the program's run time (more than 15 percent), there are a handful of methods that take a considerable amount of time (between 1 and 10 percent) but much less than the first few, and then there are many methods that do not take much time at all (less than 0 percent).

The methods taking up the most CPU time are the `adjust_heap()` method for the priority queue, the `get_intersection()` function for the `TrafficIntersection` objects, and the `event()` function for the `Car` objects. The `adjust_heap()` method takes up 32.65% of the program's run time, which is 4.39 seconds. The `get_intersection()` method takes up 28.49% of the program's run time, which is 3.83 seconds. The `Car::event()` function takes up 15.99% of the program's run time, which is 2.15 seconds.

The next most expensive functions are `push_heap`, `runOne`, `event` (for TrafficIntersection objects), and `scheduleMe`. These take up 7.10%, 6.77%, 4.98%, and 2.53% of the program's running time, respectively. All other methods each take up less than 1% of the program's running time.

The priority queue is quite expensive!

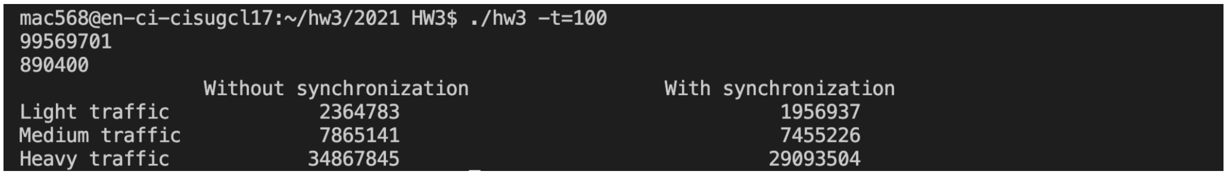
We can confirm that the priority queue is quite expensive. Two out of the seven most expensive methods in the program are associated with the priority queue. They are the `adjust_heap` and `push_heap` methods. The `adjust_heap` method makes sense to be so expensive because this is essentially the "bubble-up" method for what is essentially a "min-heap" in the event queue that we have; each time we push an event to our queue, the min-heap bubbles the event up to the front of the queue based on its time field; the events with lowest time get bubbled to the top, ahead of events that are already in the heap. This functionality of the queue has a lot of overhead. The `push_heap` method may not be as expensive of a method, but it is the most called method in the entire program; it is called 162 million times! For context, the next most frequently used method is called 81 million times. Overall, the priority queue is clearly very expensive.

Notice that `stoi` and `stod` are not listed

The `stoi` and `stod` methods are not shown in the `gprof` output page because they are templated functions. They get instantiated during compilation instead of in the executable, so they are not found when profiling.

Counting how many times `stoi` and `stod` were called

I tracked the number of `stoi` and `stod` occurrences and printed the results before the output.

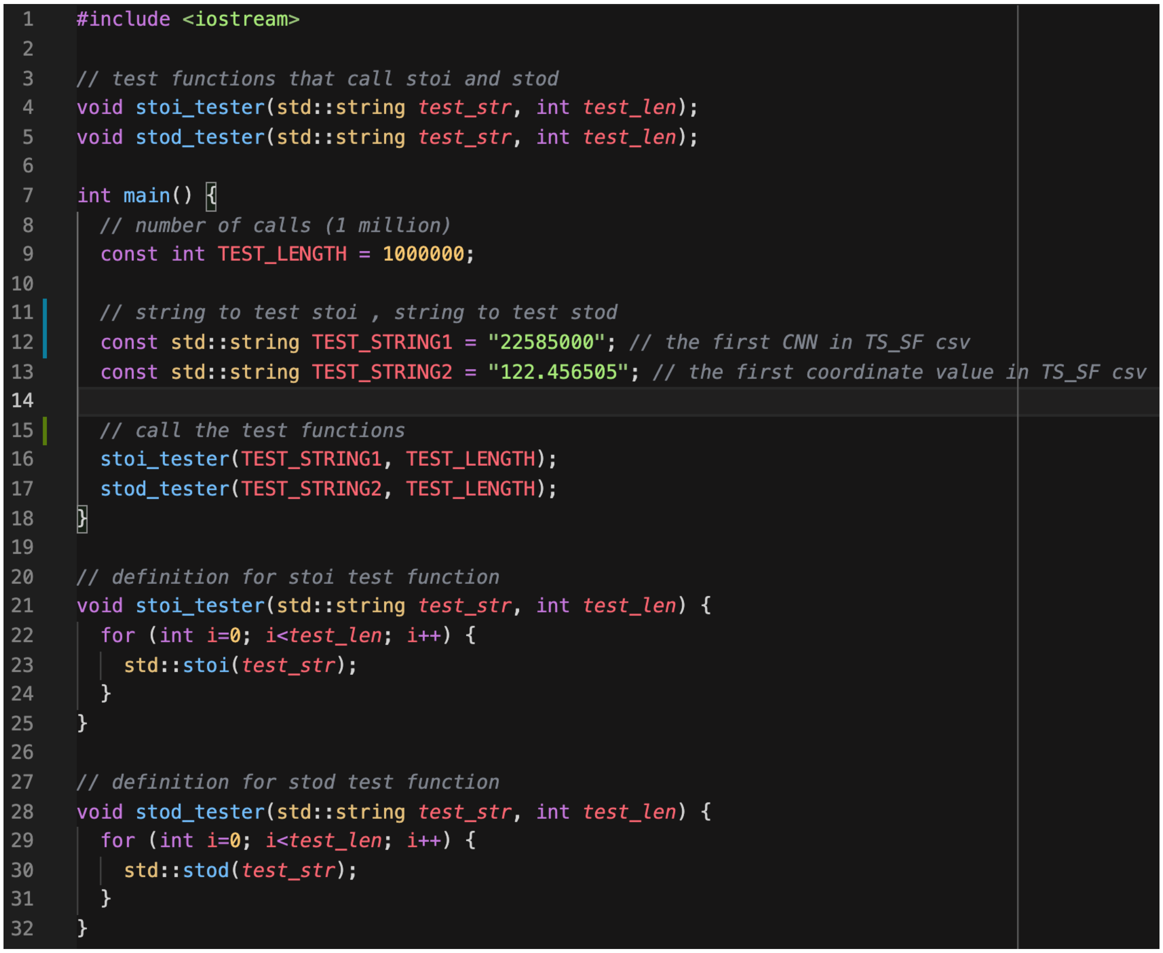


```
mac568@en-ci-cisugcl17:~/hw3/2021 HW3$ ./hw3 -t=100
99569701
890400
                Without synchronization                    With synchronization
Light traffic             2364783                                  1956937
Medium traffic            7865141                                  7455226
Heavy traffic            34867845                                 29093504
```

→ We can see that `stoi` was called 99,569,701 times and `stod` was called 890,400 times.

Small Tester Program

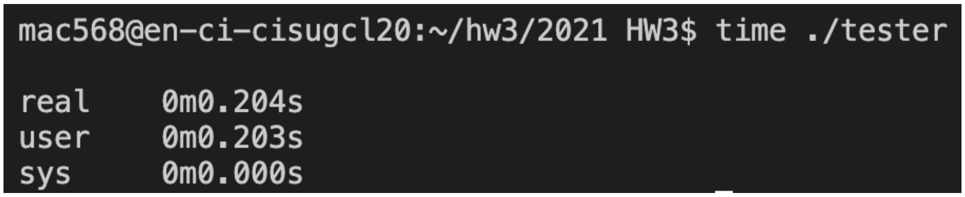
How costly are `stoi` and `stod`? I created the following program to see how much time is actually spent running these functions.



```cpp
#include <iostream>

// test functions that call stoi and stod
void stoi_tester(std::string test_str, int test_len);
void stod_tester(std::string test_str, int test_len);

int main() {
  // number of calls (1 million)
  const int TEST_LENGTH = 1000000;

  // string to test stoi , string to test stod
  const std::string TEST_STRING1 = "22585000"; // the first CNN in TS_SF csv
  const std::string TEST_STRING2 = "122.456505"; // the first coordinate value in TS_SF csv

  // call the test functions
  stoi_tester(TEST_STRING1, TEST_LENGTH);
  stod_tester(TEST_STRING2, TEST_LENGTH);
}

// definition for stoi test function
void stoi_tester(std::string test_str, int test_len) {
  for (int i=0; i<test_len; i++) {
    std::stoi(test_str);
  }
}

// definition for stod test function
void stod_tester(std::string test_str, int test_len) {
  for (int i=0; i<test_len; i++) {
    std::stod(test_str);
  }
}
```

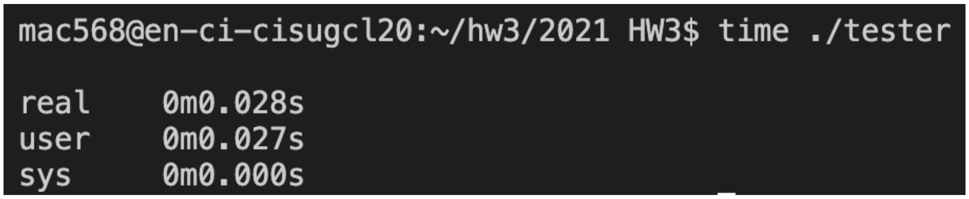
This program calls `stoi` and `stod` each 1 million times on arguments very similar to the ones used in the actual program (the CNN value of the first row and the latitude of the first row in Traffic_Signals_SF.csv).

We get the following time when we call both test functions (using `-O3`):



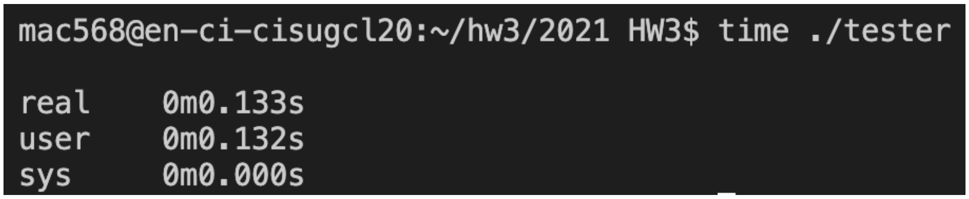
```
mac568@en-ci-cisugcl20:~/hw3/2021 HW3$ time ./tester

real    0m0.204s
user    0m0.203s
sys     0m0.000s
```

We get the following time when we call just the stoi test function (using `-O3`):



```
mac568@en-ci-cisugcl20:~/hw3/2021 HW3$ time ./tester

real    0m0.028s
user    0m0.027s
sys     0m0.000s
```

We get the following time when we call just the stod test function (using `-O3`):



```
mac568@en-ci-cisugcl20:~/hw3/2021 HW3$ time ./tester

real    0m0.133s
user    0m0.132s
sys     0m0.000s
```

Analysis and what I learned from using `gprof`

To put this into perspective, we know that `stoi` gets called about 99.5 million times and `stod` gets called just under 1 million times. While ignoring extraneous factors, we can do some arithmetic to figure out how much time these functions take in the actual program.

$$99569701\ stoi\ calls \div 1000000\ test\ calls\ =\ 99.569701$$
$$99.569701\ *\ 0.027\ seconds\ per\ million\ calls\ of\ stoi\ =\ 2.688381927\ seconds\ spent\ calling\ stoi$$

$$890400\ stod\ calls \div 1000000\ test\ calls\ =\ 0.8904$$
$$0.8904\ *\ 0.132\ seconds\ per\ million\ calls\ of\ stod\ =\ 0.1175328\ seconds\ spent\ calling\ stod$$

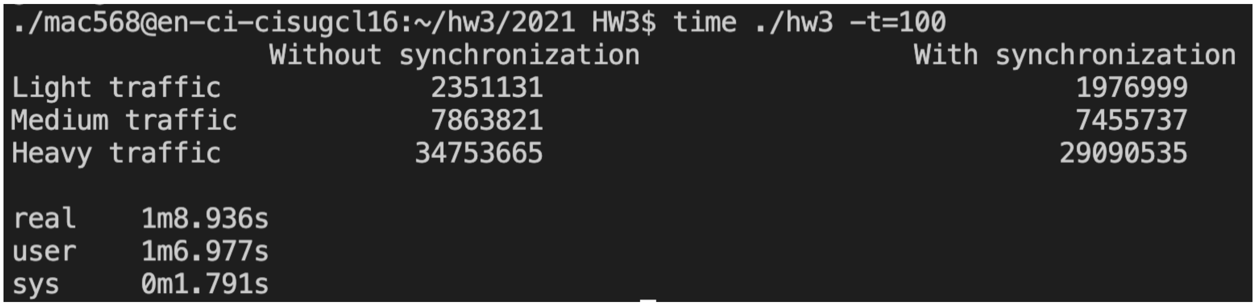
Overall (through naive arithmetic), the program spends close to 3 seconds just calling `stoi` and `stod` (when using optimizations, `-O3`). If we go back to our time for running the entire program, we see that it takes 19.161 seconds to run. This would mean that if `stoi` and `stod` *were* to show up on the `gprof` output page, it would show that `stoi` and `stod` are taking up about 15.78% of the program's running time. That is a big percentage!

Step 3: Switching from a priority queue to a vector of vectors

Switching over to a vector of vectors is a good way to reduce the overhead that we were experiencing with the standard priority queue. Let's re-run the same scenarios that we did in Step 1.

i. Compile with the flag `-g`:

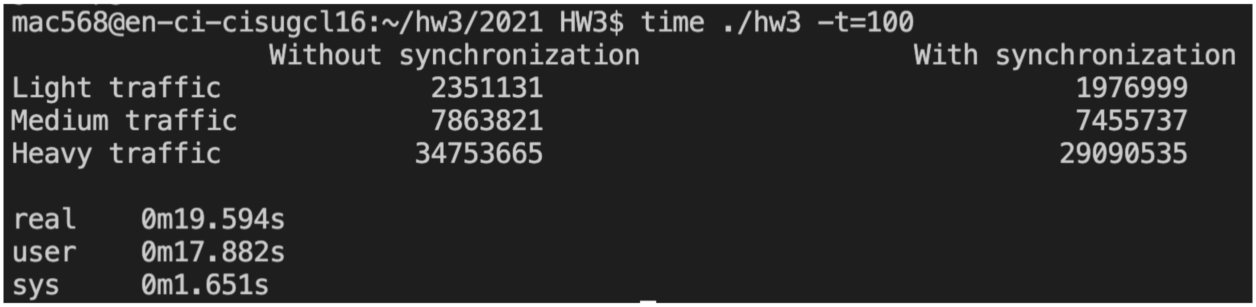
Time it takes running with debugger information:



```
./mac568@en-ci-cisugcl16:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic             2351131                                1976999
Medium traffic            7863821                                7455737
Heavy traffic            34753665                               29090535

real    1m8.936s
user    1m6.977s
sys     0m1.791s
```

ii. Compile with flags `-O3` and `-pg`:

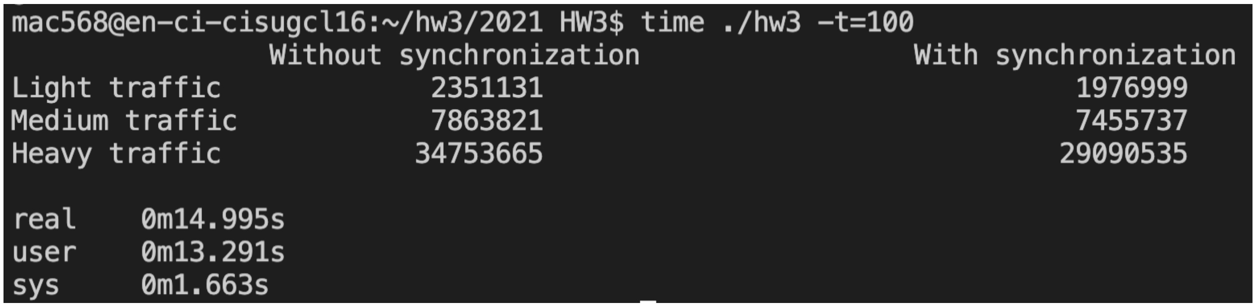
Time it takes running with optimization and profiling, but with no debugger information:



```
mac568@en-ci-cisugcl16:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic             2351131                                1976999
Medium traffic            7863821                                7455737
Heavy traffic            34753665                               29090535

real    0m19.594s
user    0m17.882s
sys     0m1.651s
```

iii. Compile with `-O3` and nothing else:

Time it takes running with optimization, but with no profiling and no debugger information:

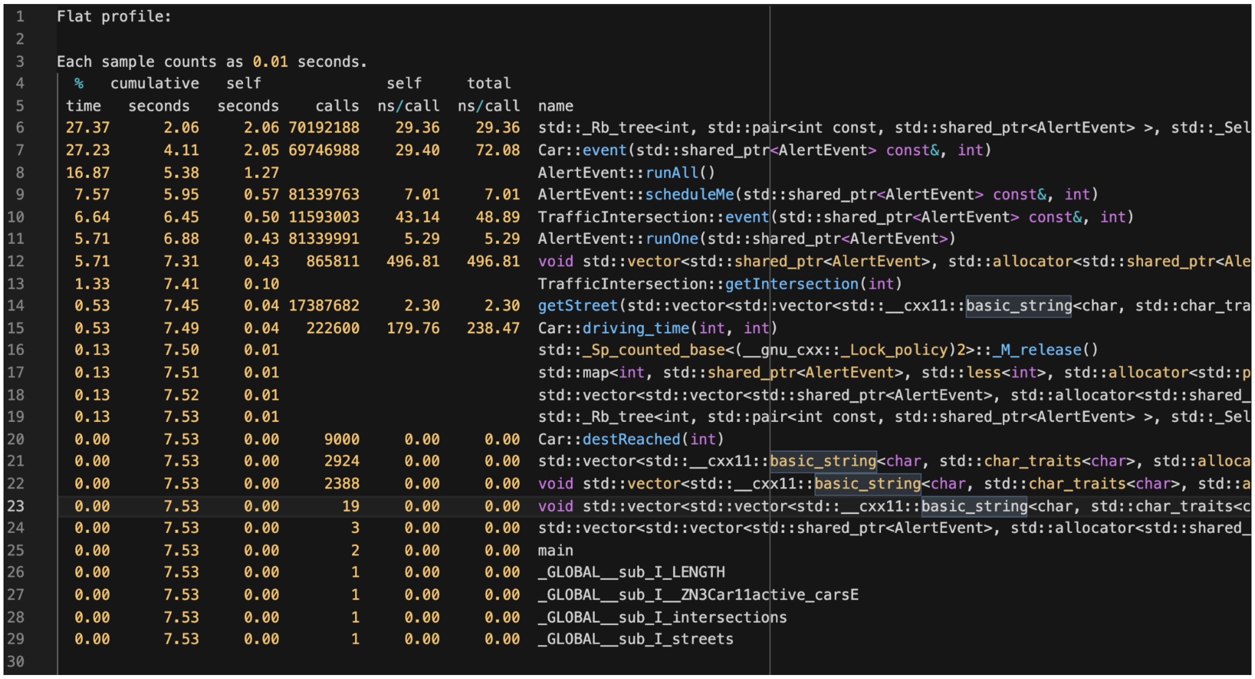


```
mac568@en-ci-cisugcl16:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic             2351131                                1976999
Medium traffic            7863821                                7455737
Heavy traffic            34753665                               29090535

real    0m14.995s
user    0m13.291s
sys     0m1.663s
```

Discussion of Step 3:

When running with just debugger information (the `-g` flag), we see that we improved from 3 minutes and 33 seconds to 1 minute and 7 seconds. This is a 68.5% improvement in running time! When running with optimization and profiling (the `-O3` and `-pg` flags), but with no debugger information, we see that we improved from 25.841 seconds to 17.882 seconds. This is a 30.8% improvement in running time! When running with optimization, but with no profiling and no debugger information (the `-O3` flag), we see that we improved from 19.161 seconds to 13.291 seconds. This is a 30.6% improvement in running time!

Analyzing the new `gprof` output:



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ns/call  ns/call  name    
 27.37      2.06     2.06 70192188    29.36    29.36  std::_Rb_tree<int, std::pair<int const, std::shared_ptr<AlertEvent> >, std::_Sel
 27.23      4.11     2.05 69746988    29.40    72.08  Car::event(std::shared_ptr<AlertEvent> const&, int)
 16.87      5.38     1.27                             AlertEvent::runAll()
  7.57      5.95     0.57 81339763     7.01     7.01  AlertEvent::scheduleMe(std::shared_ptr<AlertEvent> const&, int)
  6.64      6.45     0.50 11593003    43.14    48.89  TrafficIntersection::event(std::shared_ptr<AlertEvent> const&, int)
  5.71      6.88     0.43 81339991     5.29     5.29  AlertEvent::runOne(std::shared_ptr<AlertEvent>)
  5.71      7.31     0.43   865811   496.81   496.81  void std::vector<std::shared_ptr<AlertEvent>, std::allocator<std::shared_ptr<Ale
  1.33      7.41     0.10                             TrafficIntersection::getIntersection(int)
  0.53      7.45     0.04 17387682     2.30     2.30  getStreet(std::vector<std::vector<std::__cxx11::basic_string<char, std::char_tra
  0.53      7.49     0.04   222600   179.76   238.47  Car::driving_time(int, int)
  0.13      7.50     0.01                             std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release()
  0.13      7.51     0.01                             std::map<int, std::shared_ptr<AlertEvent>, std::less<int>, std::allocator<std::p
  0.13      7.52     0.01                             std::vector<std::vector<std::shared_ptr<AlertEvent>, std::allocator<std::shared_
  0.13      7.53     0.01                             std::_Rb_tree<int, std::pair<int const, std::shared_ptr<AlertEvent> >, std::_Sel
  0.00      7.53     0.00     9000     0.00     0.00  Car::destReached(int)
  0.00      7.53     0.00     2924     0.00     0.00  std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
  0.00      7.53     0.00     2388     0.00     0.00  void std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::a
  0.00      7.53     0.00       19     0.00     0.00  void std::vector<std::vector<std::__cxx11::basic_string<char, std::char_traits<c
  0.00      7.53     0.00        3     0.00     0.00  std::vector<std::vector<std::shared_ptr<AlertEvent>, std::allocator<std::shared_
  0.00      7.53     0.00        2     0.00     0.00  main
  0.00      7.53     0.00        1     0.00     0.00  _GLOBAL__sub_I_LENGTH
  0.00      7.53     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN3Car11active_carsE
  0.00      7.53     0.00        1     0.00     0.00  _GLOBAL__sub_I_intersections
  0.00      7.53     0.00        1     0.00     0.00  _GLOBAL__sub_I_streets
```

We can see that the methods at the top are no longer associated with the priority queue. The methods taking up the most CPU time are the `std::_Rb_tree...()` method for the vector of vectors of `AlertEvent` shared pointers, the `event()` function for the `Car` objects, and the `runAll()` function for the `AlertEvent` objects. The most expensive method is still associated with our event "queue" (now vector), but now it only takes up 27.37% of the program's run time, which is 2.06 seconds — as opposed to 32.65% and 4.39 seconds — a big improvement! The `Car::event()` method takes up 27.23% of the program's run time, which is 2.05 seconds; this method bumped up to a higher percentage of the program's run time, which is actually a good thing, because it takes the same amount of seconds as before. The `AlertEvent::runAll()` function takes up 16.87% of the program's run time, which is 1.27 seconds; this went up from previously being 0.45% of the program's run time and 0.06 seconds due to the vector event list processing now associated with `runAll()`; this was necessary for the overall improvement in "event queue" overhead.

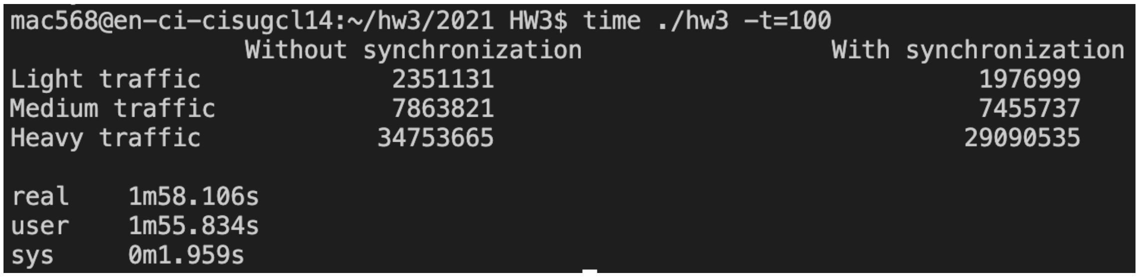
Step 4: Preventing repeat calls of `stoi` and `stod`

I decided to take an approach that would ensure that `stoi`, parsing the `POINT` string, and `stod` are *never called more than once for the same thing*. What I did was, I created two unordered maps that hold all `stoi` and `stod` conversions ever introduced in the program; the first map maps cnn strings to cnn integers and the second map maps cnn integers to pairs of doubles, where each pair holds the longitude and latitude for the given cnn. This way, whenever I need a cnn integer or a set of coordinates, I just check the two maps to see if the program ever used a certain cnn before or a certain set of coordinates before; if it did, then I can just reuse the conversion value without having to call `stoi` or `stod` again (also I avoid parsing the `POINT` string again); if it is not in the map, then I just call `stoi` or `stod` for the first time within that if-block.

Here is the before/after impact of the change after implementing my solution:

i. Compiling with the flag `-g`:

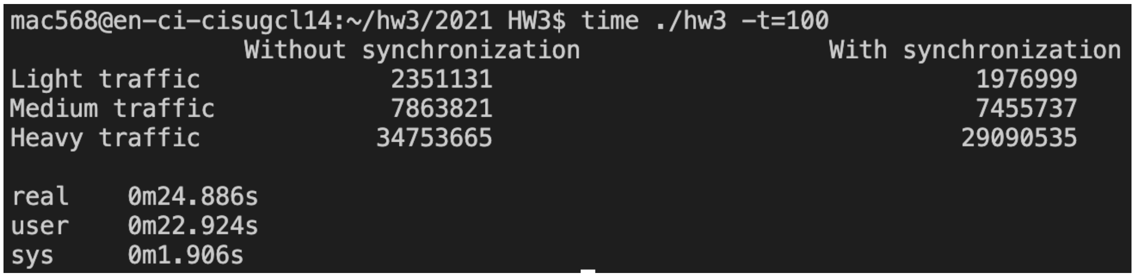
Time it takes running with debugger information:



```
mac568@en-ci-cisugcl14:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic            2351131                                  1976999
Medium traffic           7863821                                  7455737
Heavy traffic           34753665                                 29090535

real    1m58.106s
user    1m55.834s
sys     0m1.959s
```

ii. Compiling with flags `-O3` and `-pg`:

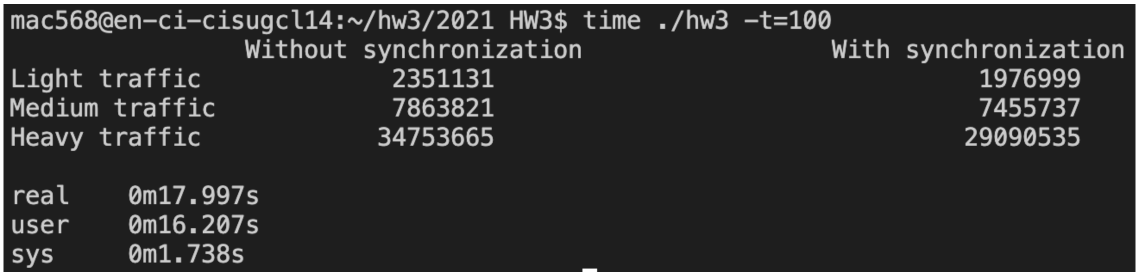
Time it takes running with optimization and profiling, but with no debugger information:



```
mac568@en-ci-cisugcl14:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                 With synchronization
Light traffic            2351131                                  1976999
Medium traffic           7863821                                  7455737
Heavy traffic           34753665                                 29090535

real    0m24.886s
user    0m22.924s
sys     0m1.906s
```

iii. Compiling with `-O3` and nothing else:

Time it takes running with optimization, but with no profiling and no debugger information:

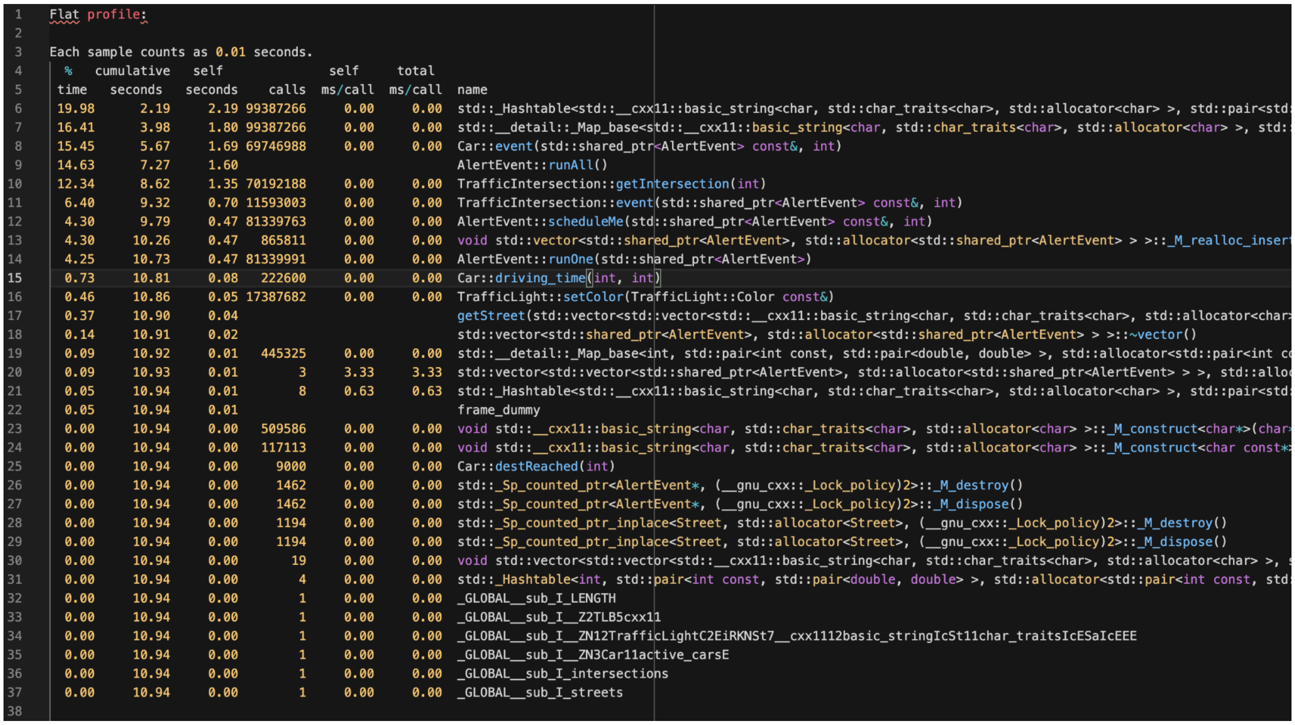


```
mac568@en-ci-cisugcl14:~/hw3/2021 HW3$ time ./hw3 -t=100
                Without synchronization                   With synchronization
Light traffic            2351131                                  1976999
Medium traffic           7863821                                  7455737
Heavy traffic           34753665                                 29090535

real    0m17.997s
user    0m16.207s
sys     0m1.738s
```

Discussion of Step 4:

When running with just debugger information (the `-g` flag), we see that we slowed down from 1 minute and 7 seconds to 1 minute and 56 seconds. When running with optimization and profiling (the `-O3` and `-pg` flags), but with no debugger information, we see that we slowed down from 17.882 seconds to 22.924 seconds. When running with optimization, but with no profiling and no debugger information (the `-O3` flag), we see that we slowed down from 13.291 seconds to 16.207 seconds. Overall, preventing repeat calls of `stoi` and `stod` actually led to a slow down in performance. This was not ideal! Although we saved a lot of time and computations by omitting multiple calls of `stoi` and `stod` for the same thing, the method in doing this was very expensive. It is likely that the map operations used to keep track of `stoi` and `stod` conversions had a lot of overhead and contributed to a lot of the running time. We can explore this further with `gprof`.

Here is the new `gprof` output:



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 19.98      2.19     2.19 99387266     0.00     0.00  std::_Hashtable<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::pair<std:
 16.41      3.98     1.80 99387266     0.00     0.00  std::__detail::_Map_base<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::
 15.45      5.67     1.69 69746988     0.00     0.00  Car::event(std::shared_ptr<AlertEvent> const&, int)
 14.63      7.27     1.60                             AlertEvent::runAll()
 12.34      8.62     1.35 70192188     0.00     0.00  TrafficIntersection::getIntersection(int)
  6.40      9.32     0.70 11593003     0.00     0.00  TrafficIntersection::event(std::shared_ptr<AlertEvent> const&, int)
  4.30      9.79     0.47 81339763     0.00     0.00  AlertEvent::scheduleMe(std::shared_ptr<AlertEvent> const&, int)
  4.30     10.26     0.47   865811     0.00     0.00  void std::vector<std::shared_ptr<AlertEvent>, std::allocator<std::shared_ptr<AlertEvent> > >::_M_realloc_insert
  4.25     10.73     0.47 81339991     0.00     0.00  AlertEvent::runOne(std::shared_ptr<AlertEvent>)
  0.73     10.81     0.08   222600     0.00     0.00  Car::driving_time(int, int)
  0.46     10.86     0.05 17387682     0.00     0.00  TrafficLight::setColor(TrafficLight::Color const&)
  0.37     10.90     0.04                             getStreet(std::vector<std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
  0.14     10.91     0.02                             std::vector<std::shared_ptr<AlertEvent>, std::allocator<std::shared_ptr<AlertEvent> > >::~vector()
  0.09     10.92     0.01   445325     0.00     0.00  std::__detail::_Map_base<int, std::pair<int const, std::pair<double, double> >, std::allocator<std::pair<int c
  0.09     10.93     0.01        3     3.33     3.33  std::vector<std::vector<std::shared_ptr<AlertEvent>, std::allocator<std::shared_ptr<AlertEvent> > >, std::allo
  0.05     10.94     0.01        8     0.63     0.63  std::_Hashtable<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::pair<std:
  0.05     10.94     0.01                             frame_dummy
  0.00     10.94     0.00   509586     0.00     0.00  void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<char*>(char
  0.00     10.94     0.00   117113     0.00     0.00  void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<char const*
  0.00     10.94     0.00     9000     0.00     0.00  Car::destReached(int)
  0.00     10.94     0.00     1462     0.00     0.00  std::_Sp_counted_ptr<AlertEvent*, (__gnu_cxx::_Lock_policy)2>::_M_destroy()
  0.00     10.94     0.00     1462     0.00     0.00  std::_Sp_counted_ptr<AlertEvent*, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
  0.00     10.94     0.00     1194     0.00     0.00  std::_Sp_counted_ptr_inplace<Street, std::allocator<Street>, (__gnu_cxx::_Lock_policy)2>::_M_destroy()
  0.00     10.94     0.00     1194     0.00     0.00  std::_Sp_counted_ptr_inplace<Street, std::allocator<Street>, (__gnu_cxx::_Lock_policy)2>::_M_dispose()
  0.00     10.94     0.00       19     0.00     0.00  void std::vector<std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, s
  0.00     10.94     0.00        4     0.00     0.00  std::_Hashtable<int, std::pair<int const, std::pair<double, double> >, std::allocator<std::pair<int const, std:
  0.00     10.94     0.00        1     0.00     0.00  _GLOBAL__sub_I_LENGTH
  0.00     10.94     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z2TLB5cxx11
  0.00     10.94     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN12TrafficLightC2EiRKNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
  0.00     10.94     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN3Car11active_carsE
  0.00     10.94     0.00        1     0.00     0.00  _GLOBAL__sub_I_intersections
  0.00     10.94     0.00        1     0.00     0.00  _GLOBAL__sub_I_streets
```

The methods taking up the most CPU time are the `std::_Hashtable…()` method, the `std::__detail::_Map_base...()` method, the `Car::event()` function, the `AlertEvent::runAll()` function, and the `TrafficIntersection::getIntersection()` function. The two most expensive methods are now associated with the hash map; the first hash table method takes up 19.98% of the program's run time, which is 2.19 seconds, and the second map base method takes up 16.41% of the program's run time, which is 1.80 seconds. This confirms that the method used to prevent repeat calls of `stoi` and `stod` was actually quite expensive! The other expensive methods are similar to what we had in the last run of the program, `Car::event()` and `AlertEvent::runAll()`. These continue to have some considerable overhead due to event list processing.

Although I was able to keep `stoi` and `stod` calls to a minimum, I wasn't able to optimize this way. Next time, I might consider doing something related to lookups on the `SC` table itself, rather than creating two new tables.