ECE 459: Programming for Performance
Assignment 4

Jinming Zhang

April 1, 2018

1 Profiling

At the beginning of this assignment, the original run time for the simulation of the Hackathon will be recorded as the base benchmark for the run time measurement of changes later on in this report. All of the run time in this report are obtained by taking the average of five runs using hyperfine with 1 warm-up. The original run time of the simulation of the Hackathon is shown in the table below.

	Time (s)
Run 1	1.905
Run 2	1.859
Run 3	1.873
Run 4	1.875
Run 5	1.883
Average	1.879

Table 1: Benchmark results for the original simulation of the Hackathon

Once the original run time is recorded, the rest of the report will be focused on explaining the changes I made in order to optimize the program. Before diving into the details for my changes, I have modified the PERF_FLAGS setting in the Makefile to make the flamegraphs more useful in the optimization process. The DWARF debugging format is used in this case.

```
1 PERF_FLAGS := -F 1000 -g --call-graph dwarf
2
```

1.1 Optimize the Container.h

In order to know where to start to optimize the program, we need to look at the flamegraph of the original program and find an interesting region for the optimization (The original flamegraph is included in the report folder, original_flamegraph.svg). In the original program's flamegraph, one of the highest fraction of samples taken within a function is the pushBack in the crossProduct. It takes 29.28% of the total samples in the flamegraph, which considered an interesting region for the optimization. The flamegraph for that region is shown below.
As we can see here, most of the samples for the crossProduct happens in the pushBack function in the Container.h. If we can optimize the pushBack function in the Container.h, we can minimize

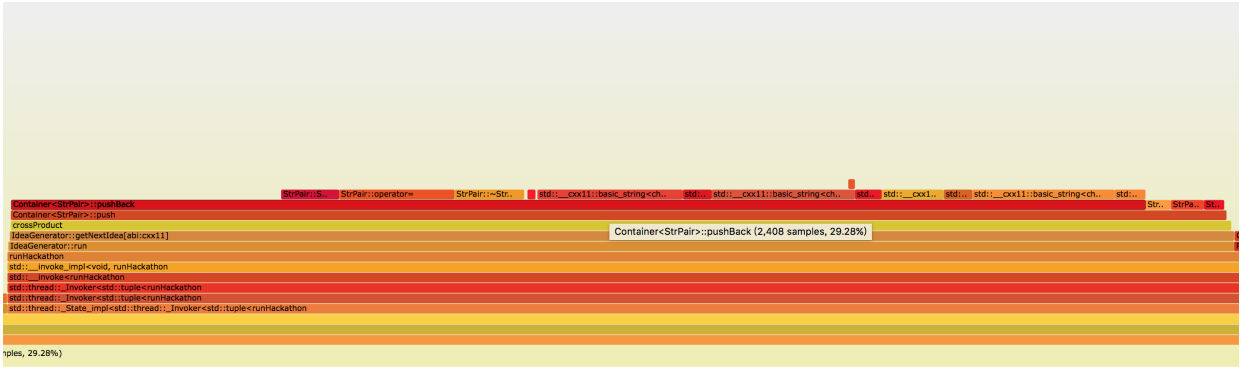


Figure 1: Flamegraph for pushBack in the crossProduct

the sampling taken place in the crossProduct function to result in a speed-up. The biggest problem with the pushBack and pushFront function in Container.h is that they are using array as their data structure. The array is not a suitable data structure to use in this case because the array has horrible performance when dealing will operations that changes the size of the array (like push and pop). During pushBack and pushFront, we need to create a new array with different size and loop through it to assign new values to it. These operations are certainly undermining the performance of the program. In order to optimize this part of the program, a queue like data structure (since we need to push and pop) is needed for the container. There are three most common queue like data structures in c++ which are std::vector, std::list, and std::deque. First, I have tried to use std::vector. std::vector is doing quite well for the pushBack functionality, however, since std::vector is a single-ended queue, the pushFront(insert) and popFront(std::vector doesn't even have this operation) will have bad performance, and it is even worst than the original implementation. std::list is a double-ended queue and can be able to perform the pushBack and pushFront quit well. But the downside for std::list is that you cannot access the element in the list using an index. You have to move the pointer of the list using std::advance or std::next to access the element. This is not ideal too. Finally, I have decided to use std::deque as the data structure for the container and it has good performance for pushBack, pushFront and popFront which is all we need. The run time for the program using std::deque for the Container.h is shown in the table below.

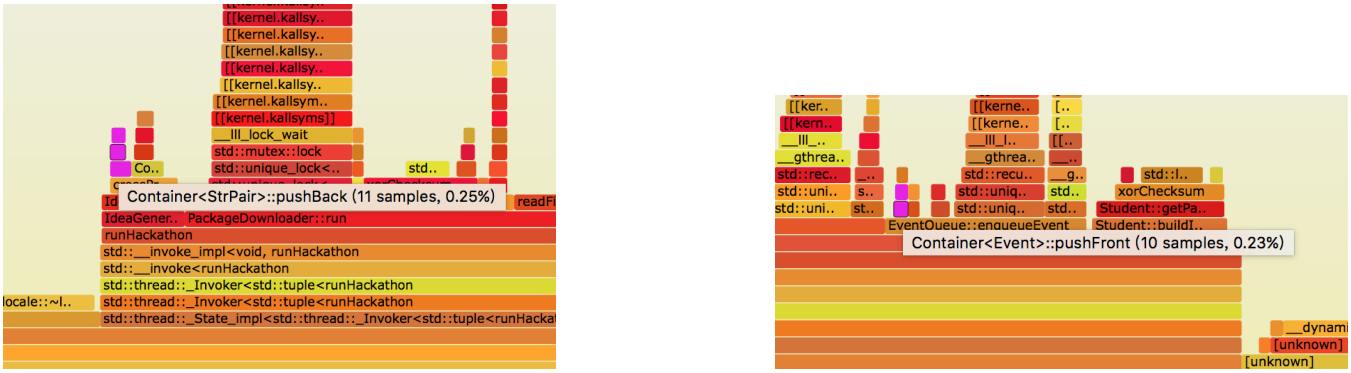
	Time (s)
Run 1	1.160
Run 2	1.157
Run 3	1.159
Run 4	1.180
Run 5	1.171
Average	1.1654

Table 2: Run time for the program using std::deque for the Container.h

The change on the data structure for Container.h results a 1.6x speed-up.

1.2 Optimize the xorChecksum

Let's look are the flamegraph after applying the change in the Container.h (This flamegraph is included in the report folder, perf_after_container_change.svg). After applying the change in the Container.h, the sample percentage for pushBack and pushFront is 0.25% and 0.20% respectively (shown below). This shows that the optimization in Container.h has certainly improved the performance of the program and justified that 1.6x speed-up.



(a) Flamegraph for pushBack after change

(b) Flamegraph for pushFront after change

Figure 2: Flamegraph after Container.h change

Now, we need to find the next interesting region for optimization. In the new flamegraph, one of the highest fraction of samples taken within a function is the xorChecksum in the ChecksumTracker <PackageDownloader>. It takes 4.76% of the total samples in the flamegraph, which considered an interesting region for the optimization. The flamegraph for that region is shown below.

Figure 3: Flamegraph for xorChecksum in the ChecksumTracker<PackageDownloader>

My first thought on how to optimize xorChecksum function in utils.cpp is to use OpenMP to parallelize the for-loop for appending and calculating the checksum in the xorChecksum and bytesToString function. However, it seems that using OpenMP to parallelize the for-loop is not a useful optimization. Since when appending and calculating the checksum, the order of operation in the for-loop matters. If append and calculate the checksum with a wrong order, it will result in a different checksum. So even with the OpenMP to parallelize the for-loop, most of the operations in the for-loop cannot execute concurrently, which makes this optimization ineffective. After the first unsuccessful attempt to optimize the xorChecksum function, I find out that most of the samples from

xorChecksum come with the hexStrToByte conversion during the checksum calculation showing on the flamegraph. The program needs to convert the baseLayer and newLayer from string to bytes and then back to a string. This is very time-consuming. My second thought on how to optimize xorChecksum function is to avoid the hexStrToByte conversion by using std::uint_8 throughout the whole checksum calculation in the xorChecksum. This attempt has successfully lowered the run-time. The run time for the program using std::uint_8 throughout the overall checksum calculation in the xorChecksum is shown in the table below (plus the container changes).

	Time (s)
Run 1	0.4903
Run 2	0.4925
Run 3	0.4927
Run 4	0.4952
Run 5	0.4918
Average	0.4925

Table 3: Run time for the program using std::uint_8 for the xorChecksum

The change in the data structure for Container.h and std::uint_8 for the xorChecksum results a 3.8x speed-up.

1.3 Optimize the readFileLine and readFile

Let's look are the flamegraph after applying the change in xorChecksum and the Container.h (This flamegraph is included in the report folder, perf_after_checksum_change.svg). After applying the change in the xorChecksum, the sample percentage for xorChecksum is negligence compare to other operations. This shows that the optimization in xorChecksum has certainly improved the performance of the program and justified the 3.8x overall speed-up.

Now, we need to find the next interesting region for optimization. In the new flamegraph, one of the highest fraction of samples taken within a function is the readFileLine in the PackageDownloader::run. It takes 1.23% of the total samples in the flamegraph. Although the 1.23% seems small, after the previous two changes, most of the run time dominant parts are resulted from std library and thread, mutex creation overhand. So this is one of the biggest tile I can find beside those kernel and library stuff. This makes readFileLine function as another interesting region for optimization. The flamegraph for that region is shown below.

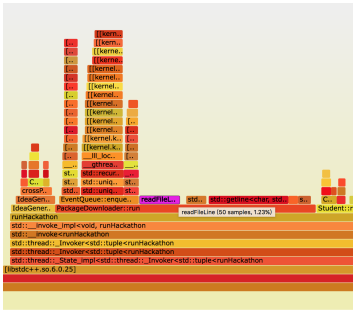


Figure 4: Flamegraph for readFileLine in the PackageDownloader::run

The problem I find out for readFileLine function (used in PackageDownloader.cpp) is that this function only returns one line from the input file. However, PackageDownloader::run needs multiple lines in the "data/packages.txt" file. This will result in multiple read from the file and read operation is a slow operation (I/O), which undermines the performance of the program. So in order to have an efficient read, I have changed the readFileLine function to readFileLines and returns a list of lines needed by PackageDownloader::run at once. This will minimize the time wasted on reads. The run time for the program using readFileLines in PackageDownloader::run to return multiple necessary lines is shown in the table below (plus all previous changes).

	Time (s)
Run 1	0.0762
Run 2	0.0762
Run 3	0.0660
Run 4	0.0761
Run 5	0.0735
Average	0.0736

Table 4: Run time for the program using readFileLines for the PackageDownloader::run

The change in the data structure for Container.h, std::uint_8 for the xorChecksum and readFileLines for the PackageDownloader::run results a 25.5x speed-up.

It seems that the speed-up has already satisfied the requirement. Just in case it fluctuates, I will introduce one more change for the program to the IdeaGenerator::getNextIdea function that uses readFile. It suffers the same problem as the readFileLine function previously stated, which triggers too many unnecessary reads that slows down the performance. My change to IdeaGenerator::getNextIdea is instead of calling readFile every time the for-loop loops it, do a readFile before the for-loop to load all the necessary lines. This will minimize the read count to one and boost the performance. The run time for the program using IdeaGenerator::getNextIdea in IdeaGenerator::run to read-only once is shown in the table below (plus all previous changes).

	Time (s)
Run 1	0.0613
Run 2	0.0534
Run 3	0.0606
Run 4	0.0549
Run 5	0.0608
Average	0.0582

Table 5: Run time for the program using IdeaGenerator::getNextIdea for the IdeaGenerator::run

The change on the data structure for Container.h, std::uint_8 for the xorChecksum, readFileLines for the PackageDownloader::run and IdeaGenerator::getNextIdea for the IdeaGenerator::run results a 32.3x speed-up.

1.4 Final Flamegraph

A part of the final flamegraph after the optimization is shown below (the full version is included in the report folder, final_flamegraph.svg).

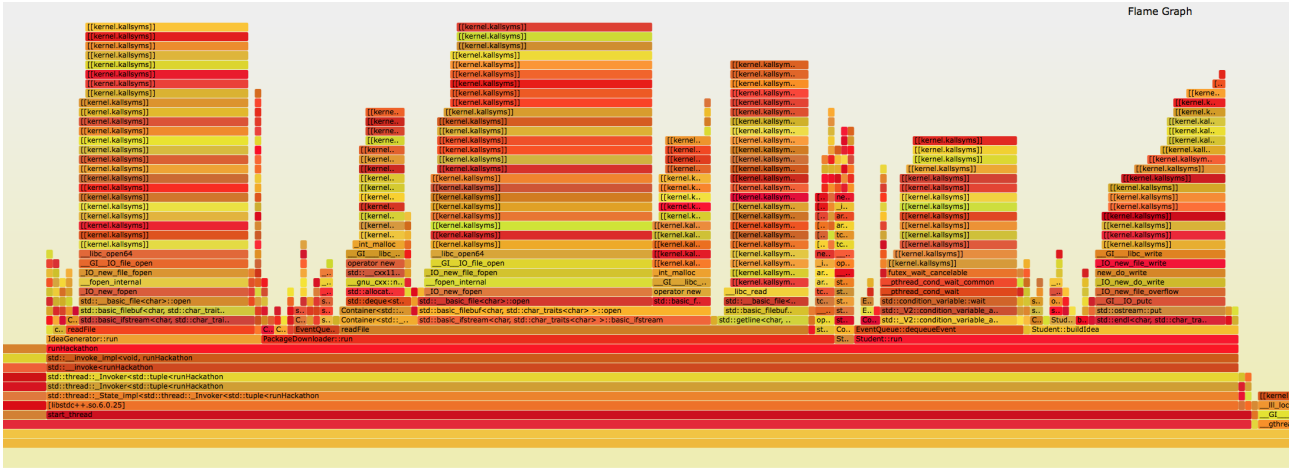


Figure 5: Final Flamegraph

In this final flamegraph, there are not many regions that can be further optimized. Most of the samples are coming from kernel and gthreads. The readFile and readFileLine tile in the final flamegraph seem wider, and that is because the overall simple because thinner. The tower for the Hackerton program is even thinner than the tower for the gthread. This means the dominant sample in this flamegraph comes from the gthread and mutex creations, which are not really interesting for optimization. This shows that the program is well optimized. The overall speed-up for all the optimization is 32.3x.