UNIVERSITY OF
WATERLOO

ECE 459 - Assignment 4

Author:
David Chau

Student Number:
20623345

Date: March 30, 2020

Profiling

The task for this assignment was to optimize the performance of a given simulation of a Hackathon. Analysis was done using flamegraphs to identify bottlenecks in the program and corresponding corrections to the code were implemented. After a correction was made, a new flamegraph was generated to be used as the next starting point of investigation. In the end, the following areas were investigated and corrected:

1. `Container` class in `Container.h`
2. `xorChecksum()` function in `utils.cpp`
3. `readFileLine()` used by `PackageDownloader::run()`
4. `getNextIdea()` used by `IdeaGenerator::run()`

The first problem to fix was the `Container` class. This was primarily indicated by the `crossProduct` function which showed that its usage of `Container<StrPair>::push` accounted for a significant amount of CPU time. To optimize this, the internal storage being used by the `Container` class was replaced by `std::deque`, which provided more efficient access to stored data. The results of this, relative to `hackathon_slow`, can be seen in Table 1.

Table 1: Results from optimizing the `Container` class

	# Samples	CPU Time	Runtime (s)
Before	3027	33.79%	1.837
After	28	0.50%	1.007

The second problem was identified as the inefficiency of the algorithm used by `xorChecksum`. Initially, it was performing 32 iterations, which processed 8 bits at a time. A solution was attempted in which only 4 iterations were required, processing 64 bits at a time. The speedup was deemed insufficient, with the bottleneck being the usage of `std::stringstream`. Finally, a solution which avoids using `std::stringstream` was implemented. The results from this, relative to the last fix, can be seen in Table 2.

Table 2: Results from optimizing `xorChecksum()`

	# Samples	CPU Time	Runtime (s)
Before	1465	26.08%	1.007
After	34	1.18%	0.3994

The next problem identified was the usage of `readFileLine()` in `PackageDownloader::run()`. Essentially, every time a package name needed to be read from file, the file had to be re-opened and a sequential search for the specified line needed to be performed. This was changed by reading in the entire file once per thread and storing it in a `Container`. From here, package names were accessed through the `Container` using index manipulation. The results of this, relative to the last fix, can be seen in Table 3.

Table 3: Results from optimizing `PackageDownloaer::run()`

	# Samples	**CPU Time**	**Runtime (s)**
Before	1647	56.79%	0.3994
After	67	15.58%	0.2427

The last problem identified was caused by `getNextIdea()` in `IdeaGenerator::run()`. In essence, each time an idea needed to be created, all the products and customers were read in and a cross product between them was performed. To correct this, each thread was made to only read in the products and customers once, with a following cross product also only being performed once. The results of this, relative to the last fix, can be seen in Table 4.

Table 4: Results from optimizing `IdeaGenerator::run()`

	# Samples	**CPU Time**	**Runtime (s)**
Before	75	17.44%	0.2427
After	14	3.00%	0.0570

The aggregated results of testing using `hyperfine` with the default program arguments can be seen below in Table 5. After all the optimizations were applied, the runtime of the program went from 1.837s to 0.057s, which is a 32x increase. As such, it can be concluded that the minimum required speed up was achieved.

Table 5: Benchmark results for `hackathon` executions on `ecetesla0`

Test Case	**Mean (s)**	σ **(s)**	**Min (s)**	**Max (s)**
`hackathon_slow`	1.837	0.0360	1.792	1.892
`Fix #1`	1.007	0.032	0.968	1.060
`Fix #2`	0.3994	0.0703	0.3217	0.5459
`Fix #3`	0.2427	0.0765	0.1435	0.4382
`Fix #4`	0.057	0.0088	0.0384	0.0794

Correspondingly, the final flamegraph can be seen in Figure 1. For the purpose of analysis, only call stacks above `runHackathon` were considered. In `hackathon_slow`, the `runHackathon` function accounted for 7265 samples and 81.10% of CPU time, whereas the final version of `hackathon_fast` only accounted for 241 samples and 51.72% of CPU time. This is validating of the fact that performance improvements were made between the two versions. With respect to further improvements, the final flamegraph indicates that the next areas to investigate are related to things such as file I/O and locking mechanisms.

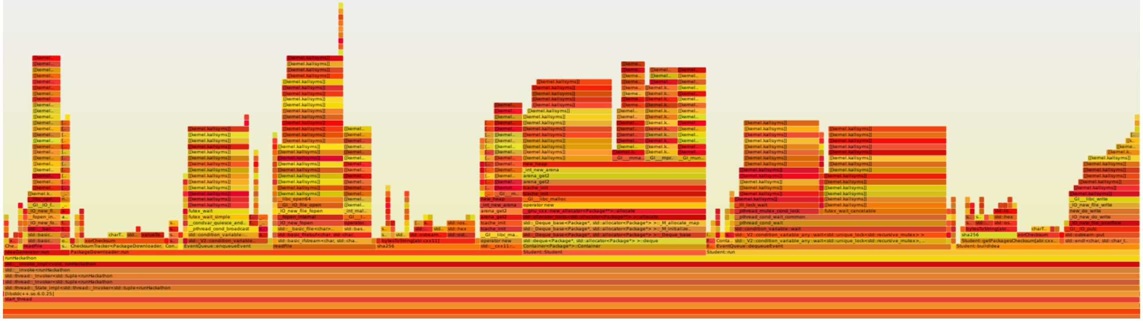

Figure 1: Relevant portion of final flamegraph showing `runHackathon`