Lab 2 C++ Profiler Report

Profiling System Writeup

Overall Architecture and Key Classes/Functions

The profiler System is designed to track and report the execution time of various code sections, helping to identify performance bottlenecks in applications. It offers a flexible and efficient way to track execution times of different code section

Architecture Overview

The system consists of several key components

1. Profiler Class: Singleton class managing the profiling process
2. TimeRecordStart and TimeRecordStop: Structures for recording timing information
3. ProfilerScopeObject: RAII object for automatic section entry/exit
4. ProfilerStats: Class for storing and calculating statistics for each profiled section
5. getTimeInSeconds: A helper function that uses chrono to measure high-resolution time. This is used throughout profiling system to capture performance.

Key Classes and Functions

- Profiler: This is the main class responsible for managing profiling sessions

  Key Functions

  - getInstance (): Singleton pattern implementation to ensure only one instance of the profiler exists
  - EnterSection (): Records the start time of a section
  - ExitSection (): Records the end time of a section and calculate elapsed time
  - calculateStats (): Aggregates profiling data to compute statistics
  - printStats (), printStatsToCSV (), printStatsToXML (), printStatstoJSON(): Output statistics in various formats
- TimeRecordStart: Holds the start time for a profiling section
  Key Members
  - sectionName (): Name of the section being profiled
  - secondsAtStart (): Time when the section started
- TimeRecordStop: Holds the stop time and metadata for a profiling section
  Key Members

  - sectionName, secondsAtStop, lineNumber, filename, functionName: various attributes to provide detailed profiling information

- ProfilerStats: Aggregates statistics for each section

  Key Members

  - sectionName, count, totalTime, averageTime, minTime, maxTime, lineNumber, filename, functionName: Used to store statistics about execution time

- ProfilerScopeObject: A RAII-style helper that automatically enters and exits profiling sections.

  Key Functions

  - Constructor and Destructor: Automatically calls EnterSection and ExitSection.

Data Flow

1. Initialization: the profiler instance is created or retrieved using getInstance ().
2. Profiling Sections: The user can profile a section of code by creating a ProfilerScopeObject, which automatically enters the section when constructed and exits it when destructed
3. Data Recording: When entering a section, the start time is recorded, and when exiting, the elapsed time is calculated and stored
4. Statistical Analysis: After profiling, the calculateStats () functions aggregates the collection data
5. Output: The profiler can output the collected data in different formats for further analysis

Performance Consideration

To ensure the Profiler system operates efficiently and does not introduce significant overhead we took some performance considerations:

- Efficient time measurement: Using chrono::high_resolution_clock ensures that the overhead from the time tracking is minimal
- Selective Profiling: Instead of profiling every part of code, the user can mark specific sections for profiling using PROFILER_ENTER (“section name”) and PROFILER_EXIT(“section name”) macros.
- Efficient Memory Usage: The Profiler uses vectors with reserved space for startTimes and elapsedTimes, minimizing memory reallocations during execution

- Low Overhead: Using the RAII with the ProfilerScopeObject ensures that sections are entered and exited without manual intervention, keeping code clean and reducing the risk of not exiting a section
- Data structure: Using the map for stats allows for average constant time complexity when accessing or modifying section statistics.
- Singleton pattern: Ensure minimal impact on the profiled code

Brief explanation of output formats

The user has the option of printing out stats to CSV, JSON and XML. However, to use the visualizations it needs the CSV file. The preferred formats to print out stats are CSV and JSON. Why?

CSV: For ease, speed and simplicity, CSV is the best choice. It is great for structured data, easy to read and write, and compatible with multiple tools like excel and data analysis libraries. Additionally, it is lightweight and straightforward.

JSON: More flexible than CSV and suitable for hierarchal data. It is also used in widely used web apps, but parsing can me a bit more complex than CSV

Despite XML being the last choice… It is better for extremely complex structure data but not ideal for simple stats which is what we are using it for. XML uses more verbose and is generally slower to parse compared to CSV and JSON.

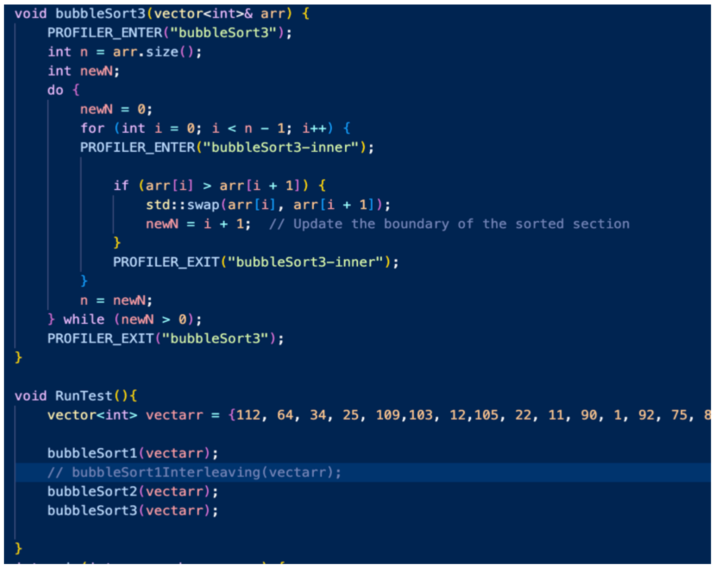
Profiler API

To use the Profiler API

1. Include the profiler header #include “profiler.hpp”
2. Get an instance of the Profiler
3. Wrap Code Section with Profiler Macros
   Example:
   Code
   ```
   {
     PROFILER_ENTER("SectionName");
     // Code to be profiled
     PROFILER_EXIT("SectionName");
   }
   ```
4. Visualize Results

5. Cleanup: At the end of your application, delete the profiler instance to free resources

Example:



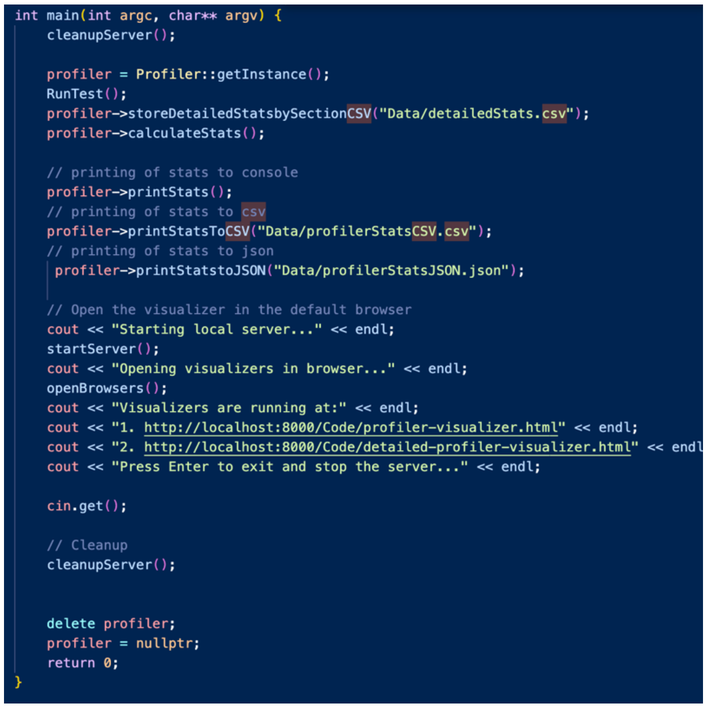
```
void bubbleSort3(vector<int>& arr) {
    PROFILER_ENTER("bubbleSort3");
    int n = arr.size();
    int newN;
    do {
        newN = 0;
        for (int i = 0; i < n - 1; i++) {
        PROFILER_ENTER("bubbleSort3-inner");

            if (arr[i] > arr[i + 1]) {
                std::swap(arr[i], arr[i + 1]);
                newN = i + 1;  // Update the boundary of the sorted section
            }
            PROFILER_EXIT("bubbleSort3-inner");
        }
        n = newN;
    } while (newN > 0);
    PROFILER_EXIT("bubbleSort3");
}

void RunTest(){
    vector<int> vectarr = {112, 64, 34, 25, 109,103, 12,105, 22, 11, 90, 1, 92, 75, 8

    bubbleSort1(vectarr);
    // bubbleSort1Interleaving(vectarr);
    bubbleSort2(vectarr);
    bubbleSort3(vectarr);

}
```



```
int main(int argc, char** argv) {
    cleanupServer();

    profiler = Profiler::getInstance();
    RunTest();
    profiler->storeDetailedStatsbySectionCSV("Data/detailedStats.csv");
    profiler->calculateStats();

    // printing of stats to console
    profiler->printStats();
    // printing of stats to csv
    profiler->printStatsToCSV("Data/profilerStatsCSV.csv");
    // printing of stats to json
     profiler->printStatstoJSON("Data/profilerStatsJSON.json");

    // Open the visualizer in the default browser
    cout << "Starting local server..." << endl;
    startServer();
    cout << "Opening visualizers in browser..." << endl;
    openBrowsers();
    cout << "Visualizers are running at:" << endl;
    cout << "1. http://localhost:8000/Code/profiler-visualizer.html" << endl;
    cout << "2. http://localhost:8000/Code/detailed-profiler-visualizer.html" << endl
    cout << "Press Enter to exit and stop the server..." << endl;

    cin.get();

    // Cleanup
    cleanupServer();


    delete profiler;
    profiler = nullptr;
    return 0;
}
```

Hierarchal and Interleaved Profiling

- Hierarchal Profiling: Hierarchal profiling is supported through nested sections. When a section is entered, it can contain other sections that also profiled. The EnterSection and Exit Section functions keep track of nested calls

- Interleaved Profiling: Interleaved profiling is possible by allowing the exit of the outer section while entering a new inner section. The profiler supports this by maintaining a stack of start times. When exiting a section, it searches for the matching entry point, allowing for accurate timing.
    - This was tested using the bubbleSort1Interleaving(vectarr); in RunTest() but was commented out in actual code.

To run using Makefile

In terminal locate the location of the project folder. From the terminal 'make compile' and then 'make run' which will run code and additional scripts for visualizations

Visualization Tool Documentation

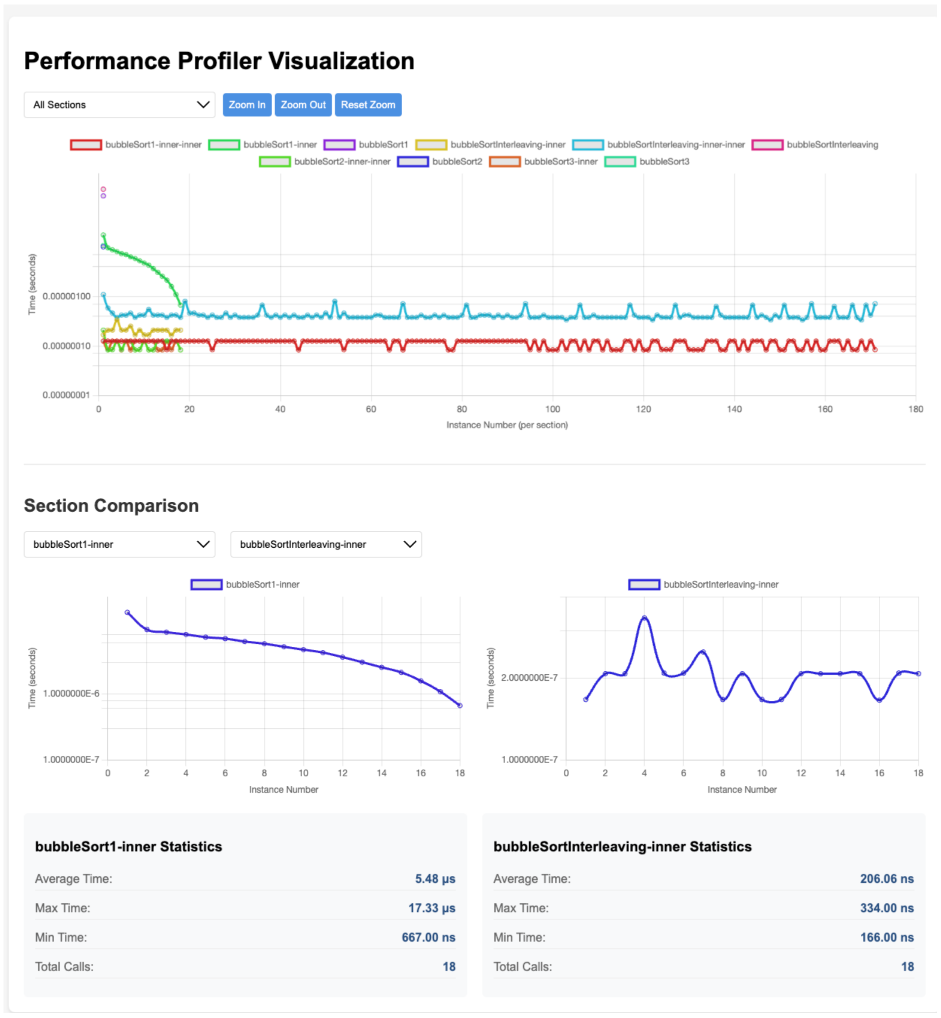
The profiler outputs its data in CSV, JSON and the additional option of XML, however only CSV data can be visualized at the moment. With the help of AI and using HTML, CSS, and D3.js for interactive charts and graphs. This was choosing due to:

1. D3.js allows for creating dynamic, data driven visualization, essential for representing hierarchal profiling data
2. Browser compatibility which makes visualizations accessible on any platform
3. Multiple chart and graph types
4. Chart.js has a straightforward API that allows for quick API implementation of complex charts.
5. Customization that the library offers various customization options.

Screenshots and Explanation of each Visualization

detailed-profiler-visualization.html

- there is an error message but just press ok to get visualization



bubbleSort1-inner Statistics

Average Time:	5.48 μs
Max Time:	17.33 μs
Min Time:	667.00 ns
Total Calls:	18

bubbleSortInterleaving-inner Statistics

Average Time:	206.06 ns
Max Time:	334.00 ns
Min Time:	166.00 ns
Total Calls:	18

Description:

- The main chart displays the performance metrics(time) for each section across multiple instances. It uses a line chart to visualize how performance changes over time for different profiling sections. It has an additional section that allows users to compare the performance of two selected sections side by side. Each section has its own line chart making it easy to visualize and compare their respective performances

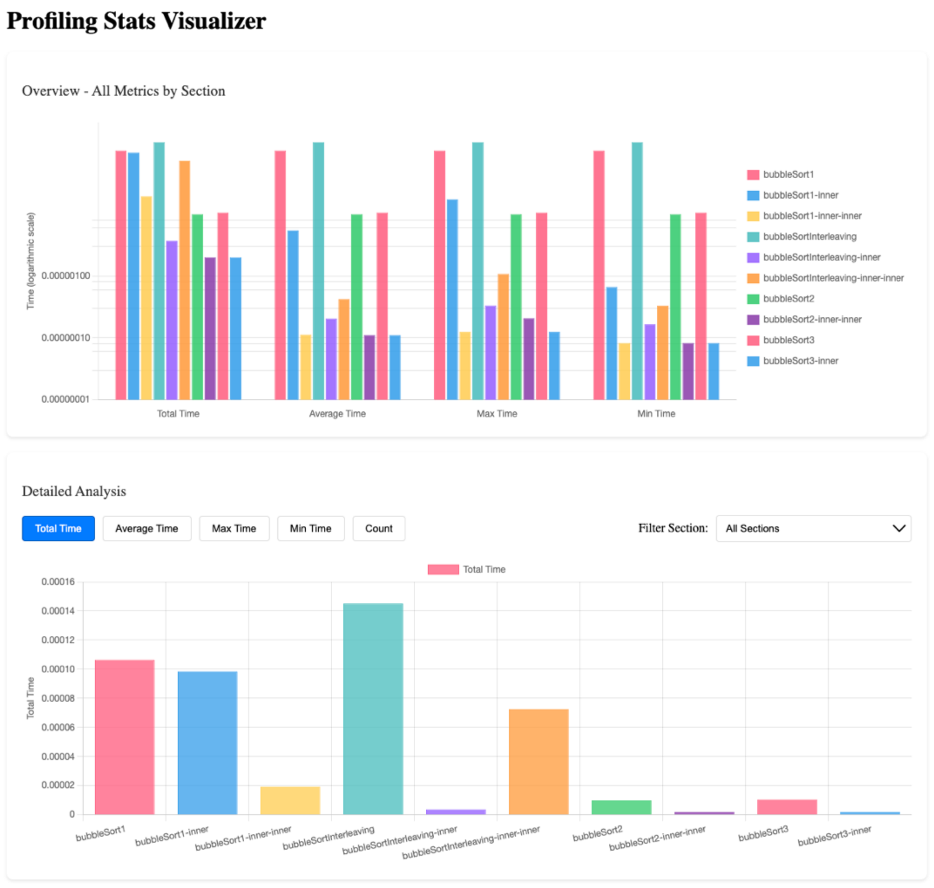
Features/ User Guide:

- Performance Profiler Visualization (Main Chart)
  - Logarithmic Scale: the Y-axis uses log scale to accommodate a wide range of time values, making it easier to visualize performance difference
  - Tooltips: Custom tooltips show the section name and exact time value when hovering over data points

- Options to zoom in, zoom out and reset zoom allowing you to box a portion of chart that you want to see further
- Color coded options to easily separate and visualize different sections

- Additional Section Comparison
    - Dynamic data loading: When a user selects a section the the corresponding data is loaded and visualized immediately
    - Statistics display: Below the charts, summary stats for each section provide quick insights
    - In drop down menu you have the option to select a section by name and compare side by side another section

Profiler-visualizer.html



Description:

- The overview chart provides an all metrics by section bar chart that displays key metrics like total time, average time, max time, min time and count for each section in a single view. In an additional section below is the details analysis chart that allows user to select specific metric and view it in detail per section. The dynamic buttons facilitate quick comparisons between metrics, enhancing user interactions.

Features/ User Guide:

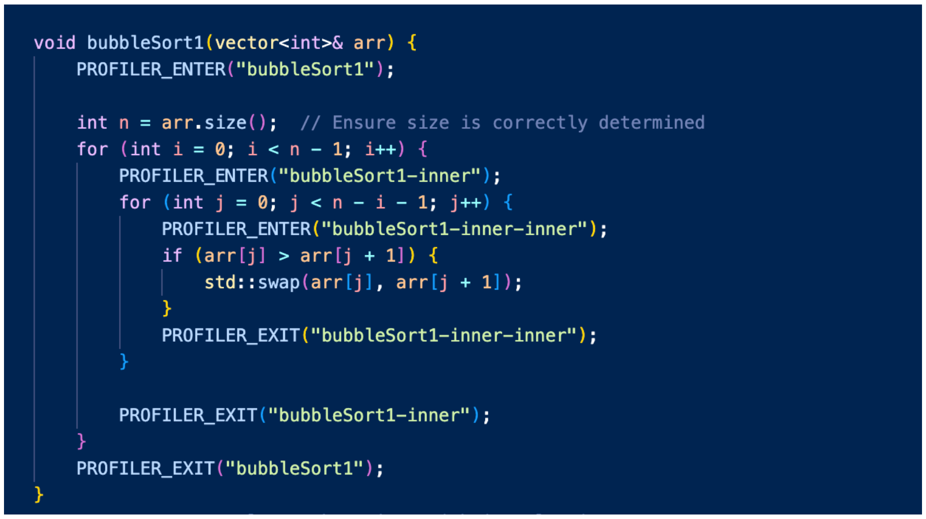
- Overview Chart Interaction
  - Tooltips: Hover over the bars to view specific values in a tooltip format, which shows the section menu and corresponding metric
- Detailed Analysis Interaction
  - Metric selection: Use the buttons to switch between metrics
  - Section filtering: Use dropdown menu labeled filter section to filter the data displayed in the detailed analysis char

Comprehensive Analysis and Demo

Describe initial Algorithm and each performance modification including rationale and expected impact

The algorithm initially profiled is a basic bubble sort. The performance of algorithm is analyzed three different times.

- Bubble Sort 1: Raw bubble sort implementation



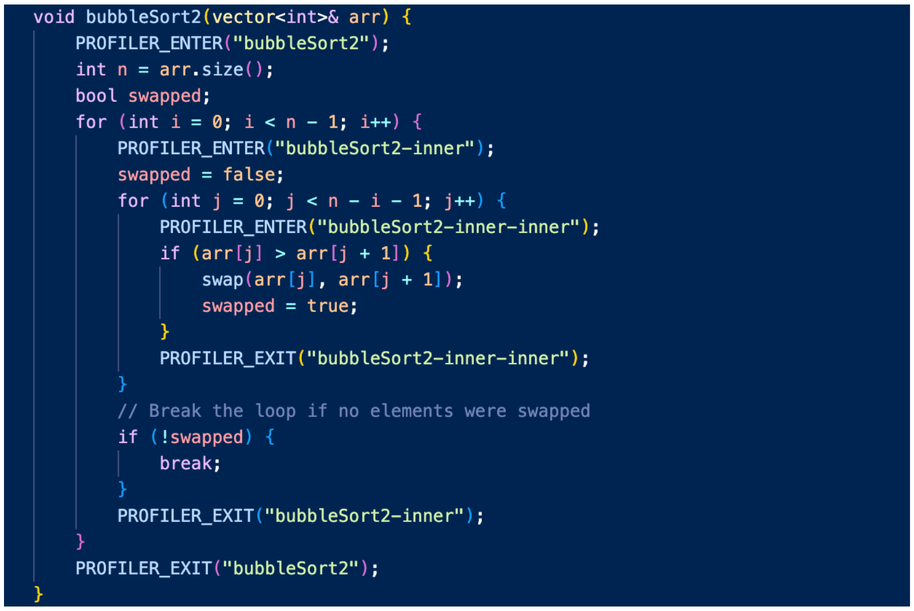
```cpp
void bubbleSort1(vector<int>& arr) {
    PROFILER_ENTER("bubbleSort1");

    int n = arr.size();  // Ensure size is correctly determined
    for (int i = 0; i < n - 1; i++) {
        PROFILER_ENTER("bubbleSort1-inner");
        for (int j = 0; j < n - i - 1; j++) {
            PROFILER_ENTER("bubbleSort1-inner-inner");
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
            PROFILER_EXIT("bubbleSort1-inner-inner");
        }

        PROFILER_EXIT("bubbleSort1-inner");
    }
    PROFILER_EXIT("bubbleSort1");
}
```

- The implementation has a time complexity of O(n^2) in all cases making it inefficient for large dataset

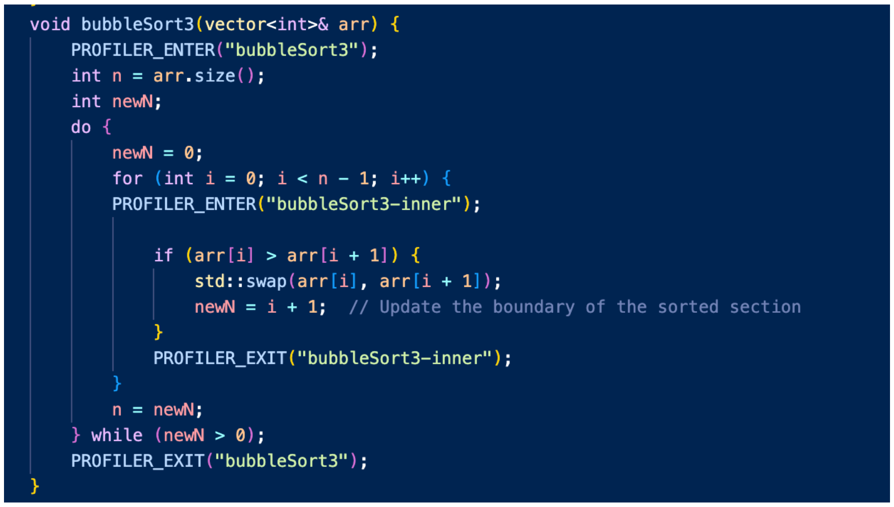
- Bubble Sort 2: Early exit in the inner loop is no swaps occurred. This reduces unnecessary iteration



```
void bubbleSort2(vector<int>& arr) {
    PROFILER_ENTER("bubbleSort2");
    int n = arr.size();
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        PROFILER_ENTER("bubbleSort2-inner");
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            PROFILER_ENTER("bubbleSort2-inner-inner");
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
            PROFILER_EXIT("bubbleSort2-inner-inner");
        }
        // Break the loop if no elements were swapped
        if (!swapped) {
            break;
        }
        PROFILER_EXIT("bubbleSort2-inner");
    }
    PROFILER_EXIT("bubbleSort2");
}
```

  - Rationale: The optimization adds a flag to check if any swaps occurred in a pass. If no swaps are made, the data structure is already sorted, and the algorithm can exit early
  - Expected Impact: This should significantly reduce the number of iterations for partially sorted arrays potentially improving performance

- Bubble Sort 3: A Cocktail Shaker Sort (bidirectional bubble sort) approach that arranges a list of elements in a specified order typically ascending



```
void bubbleSort3(vector<int>& arr) {
    PROFILER_ENTER("bubbleSort3");
    int n = arr.size();
    int newN;
    do {
        newN = 0;
        for (int i = 0; i < n - 1; i++) {
        PROFILER_ENTER("bubbleSort3-inner");

            if (arr[i] > arr[i + 1]) {
                std::swap(arr[i], arr[i + 1]);
                newN = i + 1;  // Update the boundary of the sorted section
            }
            PROFILER_EXIT("bubbleSort3-inner");
        }
        n = newN;
    } while (newN > 0);
    PROFILER_EXIT("bubbleSort3");
}
```

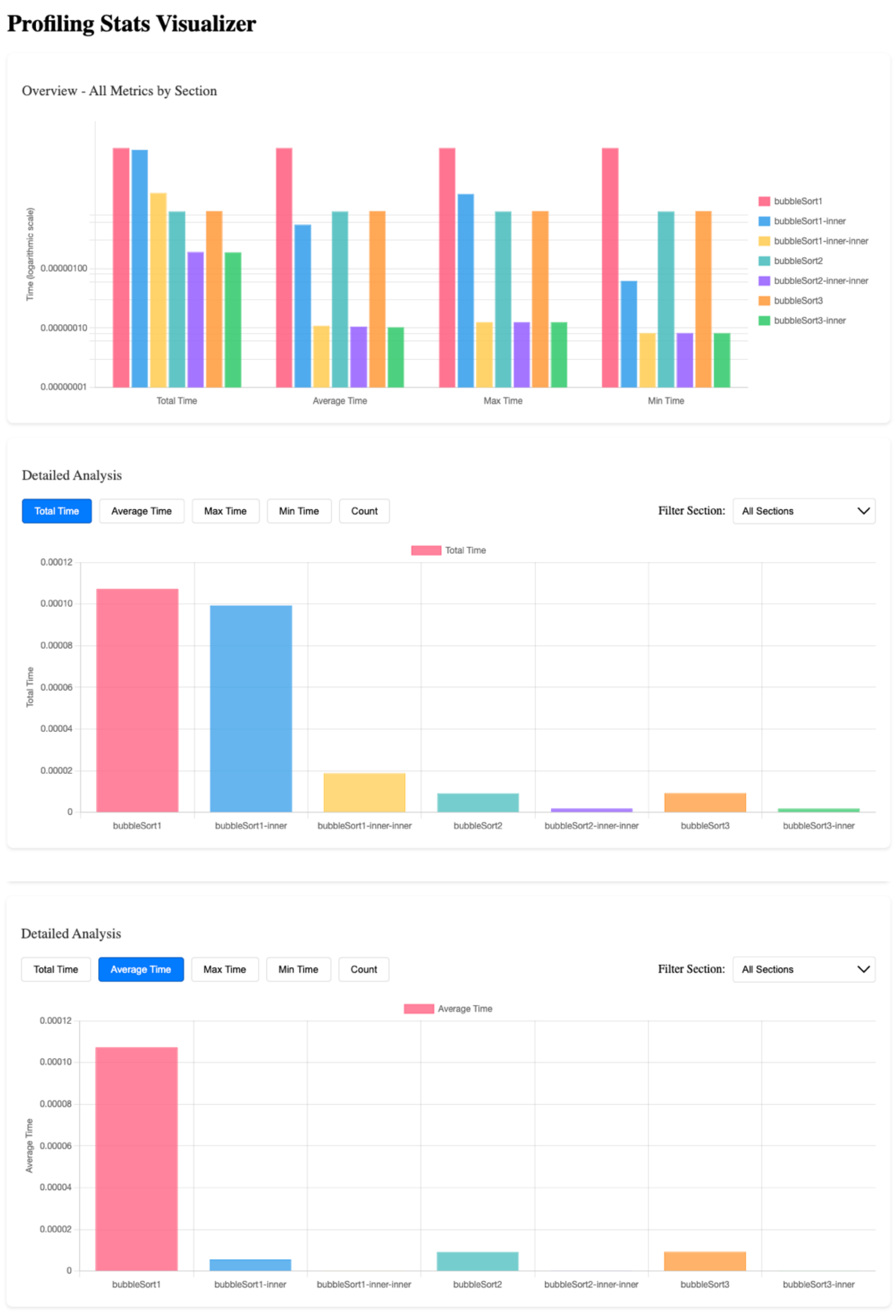
  - Rationale: this variant, alternates between forward and backward passes through the array

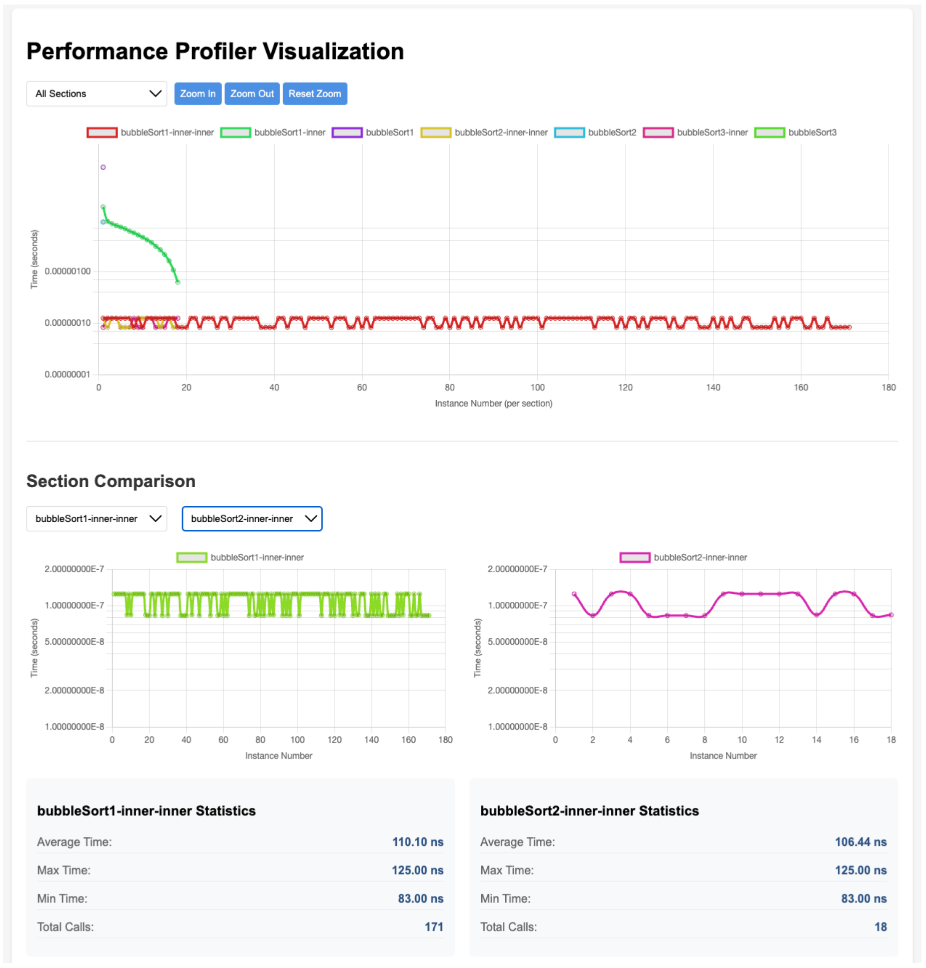
        - Expected impact: It can be faster for certain input distributions, particularly when small elements need to move to the beginning of array/vector quickly.

Expected Observations:

1. bubbleSort1 will likely show the highest total time due to its lack of optimizations
2. bubbleSort2 should demonstrate improved performance over bubbleSort1, due to partially sorted inputs, due to early exit feature
3. bubbleSort3 may show varied results depending on the input distribution. It could outperform the other.

Results and Impact Analysis of each



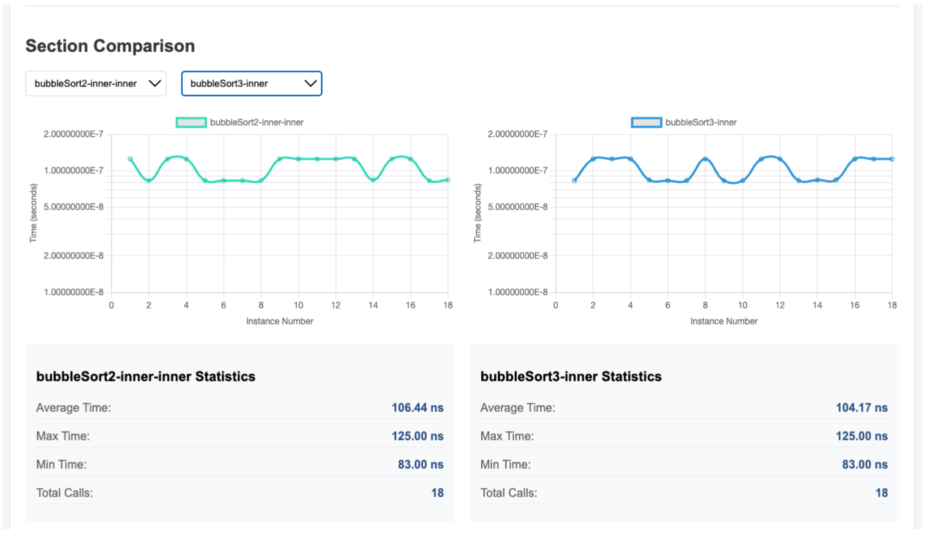


bubbleSort1-inner-inner Statistics

Average Time:	110.10 ns
Max Time:	125.00 ns
Min Time:	83.00 ns
Total Calls:	171

bubbleSort2-inner-inner Statistics

Average Time:	106.44 ns
Max Time:	125.00 ns
Min Time:	83.00 ns
Total Calls:	18



bubbleSort2-inner-inner Statistics

Average Time:	106.44 ns
Max Time:	125.00 ns
Min Time:	83.00 ns
Total Calls:	18

bubbleSort3-inner Statistics

Average Time:	104.17 ns
Max Time:	125.00 ns
Min Time:	83.00 ns
Total Calls:	18

BubbleSort1 (base, -inner -inner-inner)

- Shows highest total and average time as expected
- Performance metrics
  - Highest peaks in time measurements

    - No early termination evident from the graphs
    - Most consistent but poorest performance
- The -inner and -inner-inner profiling showed
    - Higher number of total calls (171 for -inner-inner)
    - More variable performance pattern
- This aligns with expected observation that their is a lack of optimization that leads to highest total time

BubbleSort2 (-base, -inner, -inner-inner)

- Shows clear improvement over bubbleSort1
- Key observations
    - Significantly fewer calls (18 vs 171)
    - More stable performance pattern
    - Average time: 106.44 ns for inner-inner variant
- Matches second expected observation about helpful early exiting feature
    - Reduced number of calls suggest early termination on partially sorted inputs
    - More consistent performance pattern indicates better handling of partially sorted sequences
    - Clear evidence of early exit optimization working as intended

BubbleSort3(base, -inner)

- Performance characteristics
    - Slightly better average time than bubbleSort2
    - Same number of calls as bubbleSort2
    - More consistent performance across points
- Aligns with third expected observation
    - Shows varies results compared to bubbleSort2
    - Did outperform other slightly in average time
    - Performance improvements is modest but measurable

The data strongly supports expected observations:

1. ✓ bubbleSort1 was indeed the slowest due to lack of optimizations

2. ✓ bubbleSort2 showed clear improvement due to early exit feature (evidenced by reduced call count)

3. ✓ bubbleSort3 did show varied results and managed to slightly outperform others

The most interesting insight is how accurately the expected observation about bubbleSort2's early exit feature was reflected in the dramatic reduction in total calls (from 171 to 18) while maintaining better performance metrics. This suggests the early termination optimization was particularly effective for this data.