Cross-Language Process Execution Benchmarking

Student: Butas Rafael-Dorian

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

1.Introduction

1.1 Context

In modern computing, software performance is influenced by how efficiently processes such as memory allocation, memory access, thread creation, context switching, and thread migration are managed. These processes can significantly affect system performance, particularly in multi-threaded environments. Programming languages like C, C++, and Java offer distinct runtime environments, memory management models, and threading mechanisms, which impact performance differently.

This project aims to measure and compare the performance of key processes (memory management, thread operations, etc.) across different programming languages: C, C++, and Java. By understanding these performance differences, developers can make more informed decisions regarding language selection for performance-critical applications.

1.2 Objectives

The primary goal of this project is to create a tool that measures and compares the execution times of processes in three programming languages (C, C++, Java). The specific objectives include:

- Measuring the execution of memory management strategies in each language, focusing on static and dynamic memory allocation.
- Measuring and comparing the efficiency of thread creation, context switching, and thread migration.
- Using high-precision time measurement techniques in each language.
- Storing measurement results in JSON files for easy data analysis.
- Generating visual graphs from the results, representing process performance across languages.
- Generating textual explanations of the graphs and recommendations.
- Designing a GUI that enables the user to run code written in different languages and visualize the comparison results.

1.3 Project Proposal

This project will develop a cross-language benchmarking tool that measures the execution time of critical processes in three selected programming languages (C, C++, Java). The tool will focus on evaluating the performance of key operations such as memory allocation (static and dynamic), memory access, thread creation, context switching, and thread migration.

To achieve this, the project will implement the following:

- **Measurement Methods:** In C/C++, the <chrono> library will be used for high-resolution time measurement, and thread operations will be handled using pthread_create. In Java, nanoTime() will be used for time measurements and the Java Thread API for threading operations.

- **Data Storage:** The results will be stored in JSON format using libraries such as cJSON (C), nlohmann/json (C++), and Jackson/JSON-simple (Java). Each JSON file will contain detailed information about the process being measured (e.g., memory management or threading), the execution time, and the programming language used.

- **Visualization and Graphs:** Using the JFreeChart library, performance data will be extracted from the JSON files and visualized in graphical form. Each graph will represent the performance of different processes across languages. For example, one graph may compare memory allocation times between C, C++, and Java, while another compares the time taken for thread creation and migration.

- **Explanations and Recommendations:** Alongside the graphs, textual explanations will be provided to help interpret the data. These explanations will:

    - **Analyze Trends:** Highlight patterns in the graphs, such as which language performed better in specific scenarios (e.g., faster memory access or more efficient thread handling).

    - **Provide Insights:** Explain why certain languages excel in particular processes, based on their runtime environments or memory/thread management models.

    - **Offer Recommendations:** Based on the results, recommendations will be made for different use cases. For instance, if C++ shows superior performance in dynamic memory allocation, the tool will suggest using C++ for applications requiring heavy memory management. Conversely, if Java handles thread migration more efficiently, it will be recommended for systems with high concurrency needs.

- **Cross-Language Execution & GUI:** A Java-based GUI will be developed to run the benchmarking tests across C, C++, and Java programs. The GUI will utilize the Java Native Interface (JNI) to interact with C/C++ code, enabling cross-language execution.

Users will be able to select specific tests to run and receive both visual (graphs) and textual feedback (explanations and recommendations) on the results.

The GUI will feature:

- **Test Selection Panel:** Users can choose which performance aspects (e.g., memory, threads) to benchmark.
- **Graphical and textual Output:** The GUI will display graphs with corresponding textual explanations and recommendations appearing next to the graphs.
- **Export Functionality:** Users will be able to export the graphs and the textual analysis for future reference or reporting.

2. Bibliographic Research

2.1 Memory Management

Memory management is the process of controlling and coordinating a computer's main memory. It ensures that blocks of memory space are properly managed and allocated so the operating system, applications and other running processes have the memory they need to carry out their operations. [1]

- What are the 3 areas of memory management?
  - **Memory management at the hardware level** which is concerned with the physical components that store data (RAM chips, memory caches). The management at this level is done by the MMU (memory management unit) whose role is to translate the logical addresses (used by running processes) to physical addresses on the memory devices. [1]
  - **Memory management at the OS level** which involves allocation of specific memory blocks to individual processes. The OS moves these processes between memory and storage devices (HDD or SSD), while tracking each memory location. When the computer runs out of physical memory space, the OS turns to virtual memory (allocated from the storage device), but this can impact performance because storage device memory is slower than the computer's main memory. [1]
  - **Memory management at the application level** which ensures the availability of adequate memory for the program's objects and data structures. This is achieved through allocation. This can be manual (done by the developer) or automatic (done using an allocator). When the developer no longer needs the memory space, that memory is released for reassignment (recycling). [1]
- Memory management in C/C++
  - In C/C++, memory management falls on **your shoulders**. This can be hard to do, but it gives you the means to optimize the performance of the program. Understanding how to release memory when it's no longer needed and using just enough for the task at hand is crucial. [2]
  - **Static allocation (on the stack)** refers to the process where the memory for variables is allocated at **compile time**. The amount of memory required is determined then the program is compiled and that specific amount is reserved for the entire duration of the execution. [2]

- **Dynamic allocation (on the heap)** refers to the process where the memory is allocated at **runtime**. This allows programs to use memory more flexibly and efficiently. Dynamic memory allocation is managed using functions loke malloc (new in C++) (allocates a specified number of bytes and returns a pointer) ,calloc (allocates memory of an array of elements, initializes them to zero, and returns a pointer) , realloc (changes the size of previously allocated memory) and free (delete in C++) (deallocates the memory). It gives you complete control over the amount of memory used at any time. It can only be accessed via **pointers**. [2]
- A **memory leak** occurs then a program fails to release an allocated memory which is no longer needed. If this memory leaks accumulate, they will consume resources leading to worse performance, system instability/crashes. [2]

- Memory management in Java
  - Java **itself** manages the memory and no special interventions of programmers in needed as the Garbage Collector has the role of clearing (free in C) the objects that are no longer used. But the Garbage collector doesn't guarantee the deallocation of memory that is still referenced, which can lead to memory leaks that can't be managed by the Java Virtual Machine (JVM). [3]
  - **JVM Memory Structure:** it has 3 areas: Method Area (for methods, classes, constructors, can be fixed or dynamic), Heap Area (shared runtime data for instances and arrays), Stack Area (store method specific values that have a short lifetime, can be fixed or dynamic). [3]
  - **Java Garbage Collection:** the process of collecting memory occupied by unreferenced objects. The Garbage Collector is controlled by JVM, which chooses when to run it (when in detects low availability of memory resources). [3]

2.2 Threads

A thread is, fundamentally, a subject or a topic. A thread may exist in human communication, such as an email exchange. In technology, a thread is typically the smallest set of instructions that a computer can manage and execute; it is the basic unit of processor utilization. There are four elements used to control a thread: ID, program counter, register set, stack allocated to the thread. A CPU core works on one thread at a time. To optimize execution performance, the CPU can order the thread's instructions for efficiency. [4]

- **Multithreading** is a CPU core design that lets two or more threads execute simultaneously. The threads share resources, even if they don't interact at all. We

need to avoid conflicts when multithreading, such as race conditions or deadlocks. Multithreading can be parallel (cores actually handle more threads) or concurrent (cores only handle one thread at a time but the processor handles more threads. [4]

- Thread Creation

  - In C/C++: You can use the **pthread_create** function to create a new thread. The **pthread.h** header file includes its signature definition along with other thread-related functions.

  - In Java: There are two ways to create a thread in java, one by extending **Thread class** (using the start() method) and the other by implementing **Runnable/Callable** Interface (using the run()/call() method of it).

- Thread Context Switch: A thread context switch refers to the process of saving the current execution context of a running thread and restoring the execution context of another thread to allow it to run. In a multithreaded environment, multiple threads within a single process can execute concurrently, and the operating system performs thread context switches to switch the execution between different threads. They are fast and have lower overhead compared to process context switching. The 4 elements of the thread are saved, it allows to continue from where it left off. There is no need to switch the memory address space because it is shared and directly accessed. In order to measure thread context switch time we will create two threads, use a synchronization mechanism (mutex, condition variable, semaphores, etc.) where thread 1 signals thread 2 to run and vice versa. [5]

- Thread Migration: Thread migration is the process of moving or migrating a thread from one processor to a remote processor. This provides dynamic load balancing and failure recovery in a multi-threaded environment. One of the most difficult problems while migrating the state of a thread is dealing with pointers in the migrant thread. If the pointers refer to data in the thread's stack, then they will only remain correct if the stack is placed in the same memory location on the destination processor as on source processor. A similar situation exists for pointers that reference data in the heap. In order to measure thread migration time, we need to pin a thread to one CPU core and then move it to another. In order to set CPU **affinity** we will use pthread_setaffinity_np to force a thread to run on a specific core. Doing it in Java is difficult because Java does not have direct APIs for setting thread affinity, but we can use **Java Native Interface** to call native C function or we can use external tool such as **taskset** (Linux) to control on which CPU's core the Java process can run on. We can also use **Windows Performance Analyzer** (WPA) to monitor thread migration time. [6]

3. Analysis

3.1 Project Analysis

Measuring the execution time of program or of parts of it is sometimes harder than it should be, as many methods are often not portable to other platforms. Choosing the right method will depend largely on your operating system, your compiler version, and also on what you mean by 'time'. For example, the methods that measure full seconds are not good for our scope because we need precision since our measurements will not take long. [7]

- **Wall time vs CPU time:** wall time (clock time) is the total time elapsed during the measurement (like a stopwatch). CPU time refers to the time the CPU was busy processing the instructions of the program. [7]
- **Best methods for our purposes**: the best and most portable way to measure wall time on C++ is using <chrono>. The <chrono> library has access to a few different clocks in your machine, each of them with different purposes and characteristics. The best one is high_resolution_clock, it uses the clock with the highest resolution available. The best (and most complicated) way to measure CPU time on windows is using <processthreadsapi.h> and GetProcessTimes(), or we can use <time.h> and clock() in Linux if we want a less complicated way to measure CPU time. In C, clock_gettime() is used for high-resolution time measurement, providing precise timestamps suitable for benchmarking tasks. It retrieves the current time with nanosecond precision, typically using either CLOCK_REALTIME (system-wide clock) or CLOCK_MONOTONIC (time since an unspecified starting point, which is unaffected by system clock changes). [7]
- **In Java:** Because we need high precision, the best method from java for measuring time is nanoTime(). This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. If we want to measure wall time in java, we need to use currentTimeMillis() but this method is not entirely accurate for time below one second. [7]

The GUI will be created using Java as it was discussed above. But how can it run C/C++ code? We do actually need to use code that's natively compiled for a specific architecture. To achieve this, the JDK introduces a bridge between the bytecode running in our JVM and the native code (usually written in C or C++). The tool is called **Java Native Interface**. [8]

- Java provides the *native* keyword that's used to indicate that the method implementation will be provided by a native code. Normally, when making a native executable program, we can choose to use static or shared libs. The latter is what makes sense for JNI as we can't mix bytecode and natively compiled code into the same binary file. Therefore, our shared lib will keep the native code

separately within its .dll file instead of being part of the java classes. [8]

- The native keyword transforms our method into a sort of abstract method with the main difference that instead of being implemented by another Java class, it will be implemented in a separated native shared library. [8]
  - **Components needed:** Java code, native code in C/C++, JNI header file for C/C++ (jni.h), C/C++ compiler. [8]
  - **C/C++ elements (defined within jni.h):** JNIEXPORT- marks the function into the shared lib as exportable so it will be included in the function table, and thus JNI can find it**.** JNICALL – combined with JNIEXPORT, it ensures that our methods are available for the JNI framework. JNIEnv – a structure containing methods that we can use our native code to access Java elements. JavaVM – a structure that lets us manipulate a running JVM (or even start a new one) adding threads to it, destroying it, etc. [8]

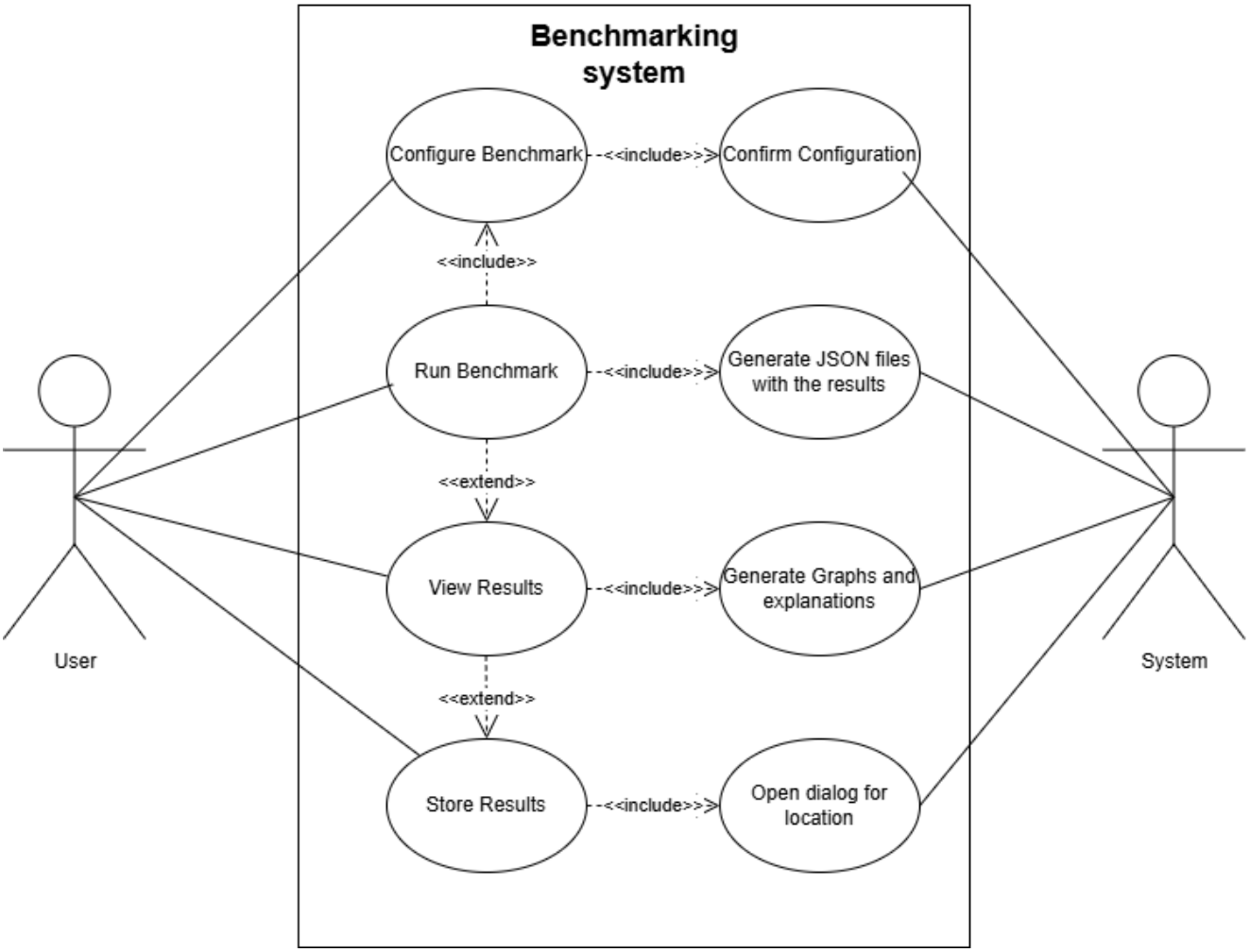
In performance benchmarking analyzing averages, standard deviations, and removing outliers is essential for obtaining meaningful, reliable insights. Here's how each metric and technique contributes to accurate measurement:

- **Averages**
  - **Mean (Arithmetic Average)**: The mean of multiple time measurements offers a straightforward central tendency, summarizing the performance of a particular operation. [9]
  - **Use in Benchmarking**: Calculating the average of multiple runs smooths out minor fluctuations due to unpredictable system behaviors, like background processes or I/O operations, which can briefly impact performance. [9]
- **Standard Deviation**
  - **Definition**: The standard deviation measures the spread of time measurements around the mean, indicating the variability or consistency in performance. A low standard deviation means the measurements are tightly clustered around the mean, suggesting consistent performance, while a high standard deviation shows variability. [9]
  - **Use in Benchmarking**: Inconsistent execution times may suggest underlying issues like memory allocation inefficiencies or unpredictable thread scheduling, which should be examined. [9]
- **Removing Outliers**
  - **Outliers**: Outliers are extreme measurements that deviate significantly from the rest, usually caused by unexpected interruptions or system anomalies. [9]
  - **Why Remove Outliers**: Outliers can skew the mean and inflate the standard deviation, leading to misleading results. For instance, a single long delay due to a system process can make the average appear slower than the true typical performance. We can remove outliers using the empirical rule. The empirical rule, or the 68-95-99.7 rule, tells you where most of the values lie in a normal distribution:
    - Around 68% of values are within 1 standard deviation of the mean.

- Around 95% of values are within 2 standard deviations of the mean.
- Around 99.7% of values are within 3 standard deviations of the mean [9]

3.2 Use cases

A use case diagram is a graphical representation of the interactions between users (actors) and a system, depicting the various use cases or functionalities of the system and how users interact with them. It provides a high-level view of the system's behavior from the perspective of its users. Here's a use case diagram representing the cross-language benchmarking project:



- **Configure Benchmark**
  - **Description**: The user selects a specific type of benchmark (e.g., Memory Allocation, Memory Access, Thread Migration) that they wish to analyze and a specific programming language in which that benchmark will run.
  - **Primary Actor**: User
  - **Preconditions**: The system presents available benchmark types and programming languages to the user.
  - **Basic Flow**:
    - User chooses a programming language from a list.

        - User chooses a benchmark type a list.
        - The system confirms the selection and prepares to run the benchmark.
    - **Postconditions**: The system is configured to execute the chosen benchmark type in the chosen programming language.

- **Run Benchmark**
    - **Description**: The user selects a programming language (C, C++, or Java) and a benchmarking type (e.g., memory access or thread creation). The system then runs the selected benchmark and records the execution time.
    - **Primary Actor**: User
    - **Preconditions**: The system is running, and the available languages and benchmark types are visible to the user.
    - **Basic Flow**:
        - User selects the programming language.
        - User chooses the benchmark type
        - User initiates the "Run Benchmark" action.
        - The system executes the benchmark, either directly in Java or through JNI for C/C++.
        - The system collects the benchmark data in JSON files.
    - **Postconditions**: The benchmark results are stored for further analysis.
    - **Alternate Flow**: If a selected benchmark or language fails to execute, an error message is shown to the user.

- **View Results**
    - **Description**: After a benchmark is completed, the user can view the results as a graph with corresponding textual explanations and recommendations appearing next to the graphs for easy analysis.
    - **Primary Actor**: User
    - **Preconditions**: A benchmark has been run and results are available for viewing.
    - **Basic Flow**:
        - User selects the "View Results" option.
        - The system generates a graph based on stored benchmark data.
        - The graph together with the explanations are presented to the user.
    - **Postconditions**: The user sees a visual and textual representation of benchmark results, aiding in performance comparison.
    - **Alternate Flow**: If no benchmark has been run, an error message informs the user that there are no results to view.

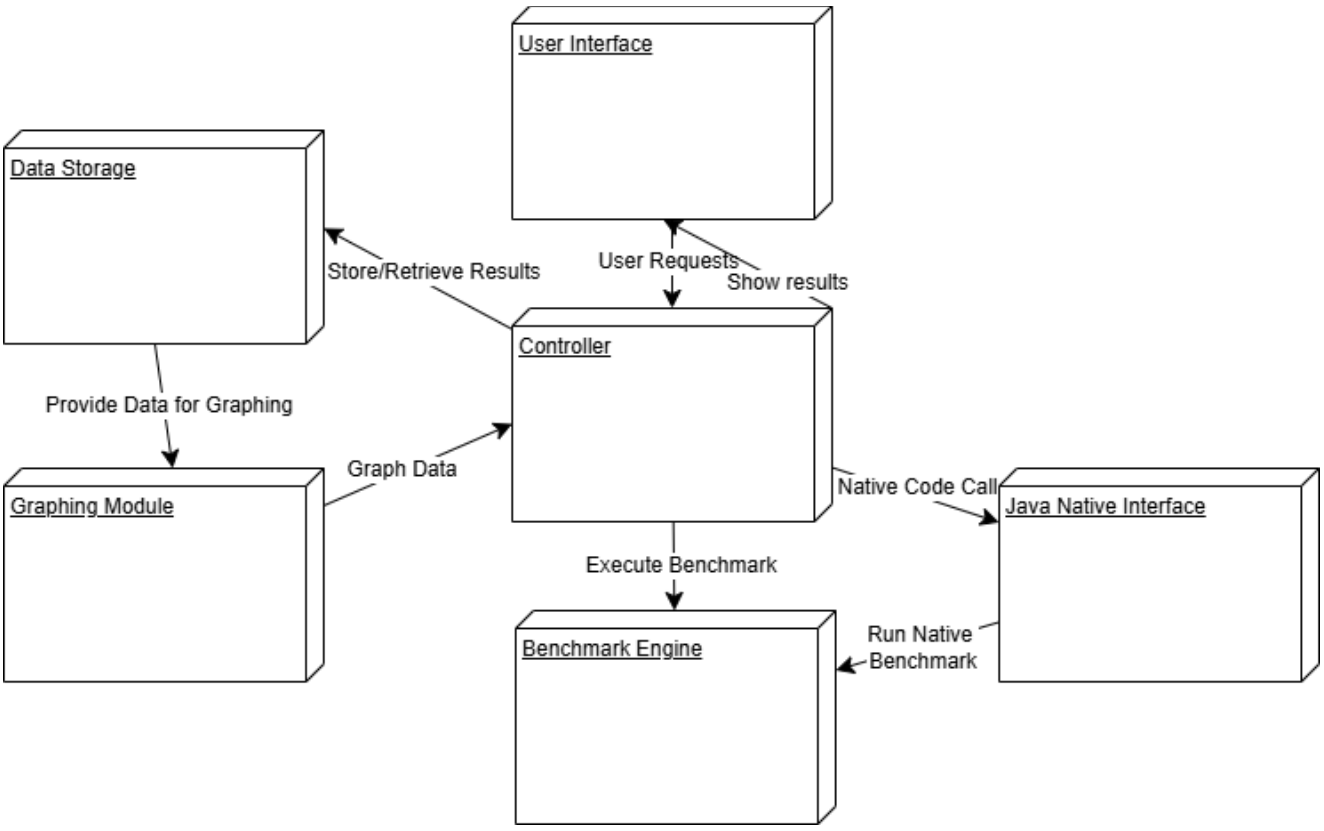
- **Export Results**
    - **Description**: The user exports the benchmark results as a JSON file and as photos with the graphs for documentation or further use.
    - **Primary Actor**: User
    - **Preconditions**: Benchmark results exist in the system.

- **Basic Flow**:
  - User selects "Export Results."
  - The system opens a dialog for the user to choose the file location.
  - Results are saved as a JSON file and as photos at the chosen location.
- **Postconditions**: The benchmark results are saved externally as a JSON file and as photos.
- **Alternate Flow**: If no results are available, an error message is shown to the user.

4. Design

4.1. Block Diagram

The block diagram shows the architectural flow of the system, detailing how components interact to accomplish the benchmarking tasks. It's more about the high-level, functional software architecture and dependencies that are easy to digest and understand:



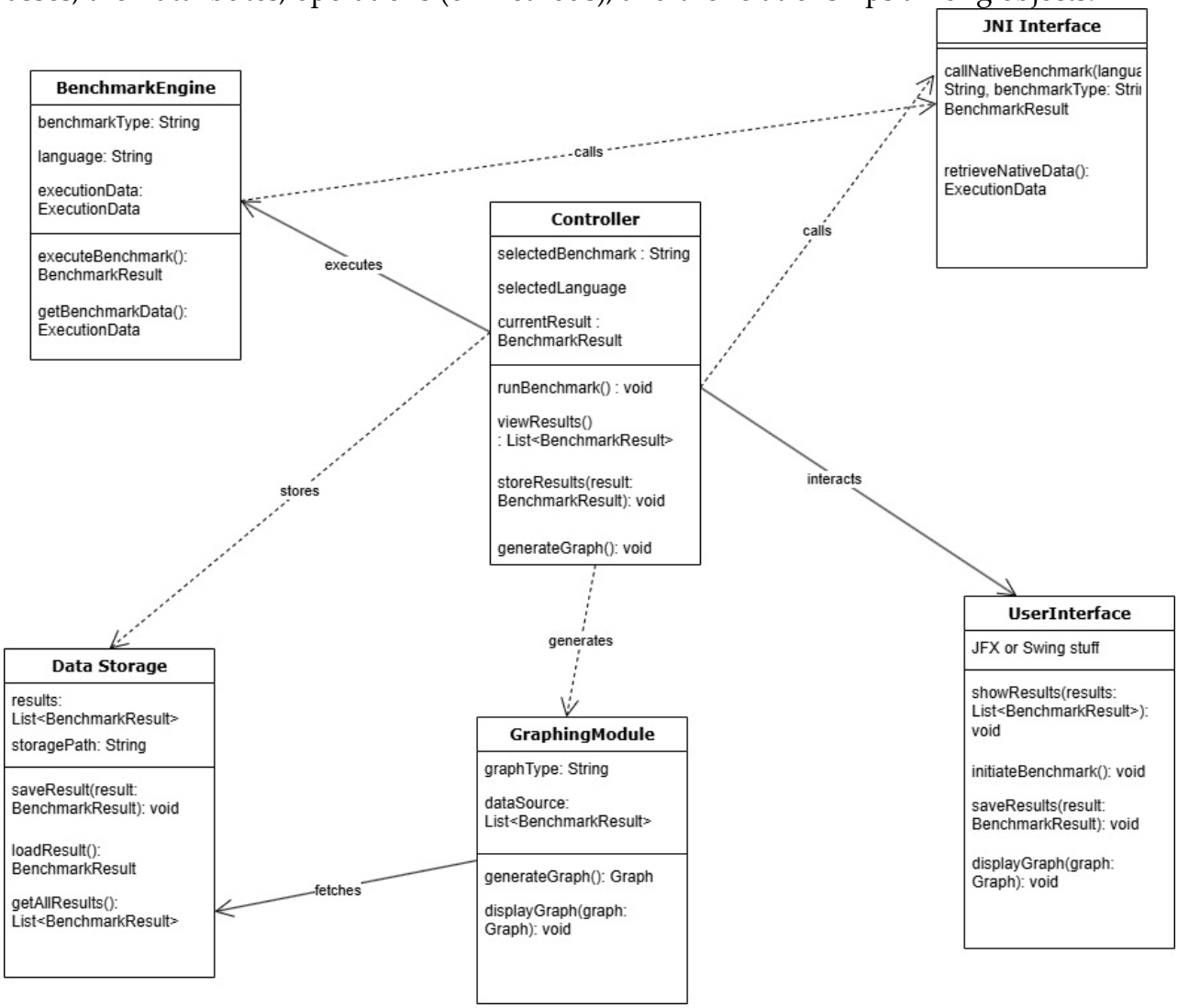
- **User Interface (UI)**: Represents the front-end where users interact with the system, select benchmarks, and view results.
- **Controller (MeasurementController)**: The main logic that coordinates benchmark selection, execution, and data retrieval.
- **Benchmark Engine**: This component houses the different benchmarking scripts or executables for each language (C, C++, Java). It receives instructions from the Controller and runs the specified benchmark.
- **JNI (Java Native Interface)**: Used to bridge Java and native code (C/C++). This layer

facilitates communication between Java GUI/Controller and native C/C++ benchmarks.

- **Data Storage**: Handles storing and retrieving results in a structured format like JSON.
- **Graphing Module**: Generates graphical and textual representations of benchmark results.

4.2. Class Diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is **a type of static structure diagram** that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.



The class diagram above represents the main components and interactions in the Cross-Language Process Execution Benchmarking system. This system is designed to facilitate benchmarking across multiple programming languages, providing a graphical interface to view results and compare performance metrics. The key classes and their roles are as follows:

- **Controller**: Serves as the central coordinator, connecting the User Interface with the other components. The Controller initiates benchmarking through the BenchmarkEngine and JNI Interface, manages data storage, and triggers the graph

generation. Key methods in the Controller include runBenchmark(), which initiates the benchmarking process, and generateGraph(), which calls on the GraphingModule to visualize results.

- **BenchmarkEngine**: Executes the benchmarking process within the Java environment. It is responsible for running benchmarks based on selected languages and benchmark types, storing raw execution data for further processing. The Controller communicates with this class to obtain benchmark data and manage its execution.
- **JNI Interface**: Acts as a bridge to allow Java to interface with native C or C++ code, facilitating cross-language benchmarking. The Controller calls methods within this interface to execute benchmarks in other languages, with callNativeBenchmark() serving as the primary method for initiating native code benchmarks.
- **DataStorage**: Manages the persistent storage of benchmark results, including saving, loading, and retrieving results as needed. The Controller stores results in DataStorage for later retrieval and presentation.
- **GraphingModule**: Generates and displays graphs based on benchmark data. After retrieving benchmark data from DataStorage, the Controller calls generateGraph() to create visualizations. The displayGraph() method then presents these graphs to the User Interface.
- **User Interface**: The main entry point for users to interact with the application. It allows users to initiate benchmarks, view results, and see visualized performance comparisons. Methods in this class include initiateBenchmark() for starting the benchmarking process, and showResults() for displaying saved results.

5. Implementation

The Cross-Language Process Execution Benchmarking project comprises various functions for benchmarking system performance across C, C++, and Java. These functions measure key aspects of memory and thread management, including memory allocation, deallocation, memory access, thread creation, context switching, and thread migration. This section provides a detailed explanation of each function, its purpose, and how it was implemented.

5.1. Memory Benchmarking Functions

Memory benchmarking functions measure the efficiency of memory allocation, deallocation, and access. These metrics reveal how each language's memory model handles these processes.

5.1.1. Memory Allocation

- **Function Name**: `measureMemoryAllocation`
- **Purpose**: Measures the time required to allocate a block of memory of a specified size.
- **Implementation Details**:
    - **C Implementation**: In C, memory allocation is done using `malloc`, which dynamically allocates a specified number of bytes on the heap and returns a pointer to the start of

this memory block. The `clock_gettime()` function, which provides nanosecond resolution, is used to record start and end times with `CLOCK_MONOTONIC` (to avoid inconsistencies due to system time changes). Timing begins immediately before the `malloc` call and stops right after it, and the allocation duration is calculated by subtracting the start time from the end time. This gives an accurate measurement of the allocation overhead.

- **C++ Implementation**: Similar to C, the C++ version uses `new` to dynamically allocate memory. Timing in C++ is handled by `<chrono>`, specifically using `std::chrono::high_resolution_clock::now()`. The time difference between allocation (`new`) and release (`delete`) gives the allocation duration. `delete` is used after each allocation to prevent memory leaks. This setup allows an accurate, comparable allocation measurement between languages.
- **Java Implementation**: In Java, memory allocation is managed by the JVM. Here, `System.nanoTime()` provides a high-precision timestamp immediately before and after creating an integer array. This function mimics C/C++ manual allocation by creating an array of integers. Unlike C/C++, where the developer has control over deallocation, Java relies on garbage collection. By isolating this allocation process, the function allows a direct comparison with C/C++.

5.1.2. Memory Deallocation

- **Function Name**: `measureMemoryDeallocation`
- **Purpose**: Measures the time taken to free or deallocate previously allocated memory.
- **Implementation Details**:
  - **C Implementation**: In C, `free` is used to release dynamically allocated memory. Timing starts immediately before calling `free` and stops immediately after, with `clock_gettime()` capturing the duration in nanoseconds. This duration represents the deallocation time, which can be directly compared to Java's garbage collection.
  - **C++ Implementation**: C++ uses `delete` to deallocate memory, following the same timing logic as in C. The `<chrono>` library calculates the deallocation time, providing a comparable duration by timing the difference between allocation and deallocation.
  - **Java Implementation**: Java does not allow explicit memory deallocation, as memory is automatically managed by the JVM's garbage collector. However, `System.gc()` is called to request garbage collection, though it may not trigger immediately. `System.nanoTime()` measures around the garbage collection request to approximate memory cleanup time in Java. This is not an exact deallocation time but provides a rough comparison with manual deallocation in C/C++.

5.1.3. Memory Access (Static and Dynamic)

- **Function Names**: `measureStaticMemoryAccess`, `measureDynamicMemoryAccess`
- **Purpose**: Measures the time taken to access elements in statically and dynamically allocated arrays.
- **Implementation Details**:
  - **Static Access (C/C++)**: Static arrays are allocated at compile-time. In the function, an array is accessed sequentially within a loop that sums up its elements. Timing is recorded around this loop using `clock_gettime()` (C) or `<chrono>` (C++), and the

total time is divided by the number of accesses to calculate an average per-access time. For large static arrays, `ulimit -s` is used in Linux to increase stack size, ensuring the array can be stored without segmentation faults.

- **Dynamic Access (C/C++)**: Dynamic arrays are allocated with `malloc` in C or `new` in C++. After allocation, a loop initializes the array, then a second loop accesses and sums its elements. Timing starts just before the access loop and stops immediately after. Using `clock_gettime()` in C and `<chrono>` in C++ captures the access time overhead specific to heap-allocated memory.
- **Java Implementation**: Java arrays are managed within the heap, whether declared final (static) or not. Arrays are accessed in a loop where each element is added to a sum variable. Timing starts and stops around this loop with `System.nanoTime()`, yielding a comparable access time metric. The final modifier in Java prevents size changes at runtime, aligning a bit with C/C++ static arrays.

5.2. Thread Management Functions

Thread management functions measure the performance of thread creation, context switching, and thread migration, demonstrating each language's concurrent capabilities.

5.2.1. Thread Creation Time

- **Function Name**: `measureThreadCreationTime`
- **Purpose**: Measures the time required to create and join a thread.
- **Implementation Details**:
  - **C Implementation**: Threads in C are created with `pthread_create`. The function starts timing just before `pthread_create` and stops right before `pthread_join`, which blocks until the thread finishes. This timing includes the overhead of thread initialization and creation.
  - **C++ Implementation**: C++ uses the `<thread>` library to handle threading. A new `std::thread` object is instantiated, and timing is captured before calling `start()` and right before `join()`, which joins the thread with the main execution.
  - **Java Implementation**: In Java, a `Thread` object is created, and timing is captured with `System.nanoTime()` around `start()` and `join()`. This function measures Java's thread creation efficiency in comparison to C/C++.

5.2.2. Context Switch Time

- **Function Name**: `measureContextSwitchTime`
- **Purpose**: Measures the average time taken for a context switch between two threads.
- **Implementation Details**:
  - **C Implementation**: Context switching is achieved by creating two threads synchronized using a mutex and condition variable. Each thread alternates control, signaling the other to run. A shared atomic counter increments with each switch, and timing is recorded between the first and last switch. The total time is divided by the number of switches to calculate an average switch time.
  - **C++ Implementation**: Using `<mutex>` and `<condition_variable>`, context switching in C++ is synchronized similarly to C, using a shared boolean flag. An atomic counter

records the number of switches, and the total duration is divided by this counter to yield the average context switch time.
    - **Java Implementation**: In Java, `ReentrantLock` and `Condition` classes synchronize two threads for context switching. Timing is recorded at each switch using `System.nanoTime()`, and the final time is divided by the total switches to give an average switch time. Two functions are needed: one for each thread that alternately signals the other, evaluating Java's context switching in comparison to pthreads.

5.2.3. Thread Migration Time

- **Function Name**: `measureThreadMigrationTime`
- **Purpose**: Measures the time required to migrate a thread between CPU cores.
- **Implementation Details**:
    - **C Implementation**: Migration is implemented using `pthread_setaffinity_np`, which binds the thread to a specific CPU core. Timing starts right after setting the initial affinity and stops after switching to another core. This captures the migration time.
    - **Java Implementation (JNI Integration)**: Java lacks a built-in method for setting thread affinity. A native function is created with JNI to handle affinity setting in C, using `pthread_setaffinity_np`. The JNI function follows naming conventions (`Java_ClassName_MethodName`) and includes `JNIEXPORT` and `JNICALL` specifiers to ensure compatibility. A `.h` file is generated for the C library, and `.so` file is compiled for use in Java. The JNI function is invoked through Java using System.loadLibrary() to load the C library (e.g., .so or .dll (for windows) file) into the JVM.

5.3. Data Storage Functions

Data storage functions ensure benchmarking results are structured and saved in JSON format, making them accessible for further analysis.

5.3.1. Saving Results to JSON

- **Function Name**: `saveResultsToJSON`
- **Purpose**: Saves benchmark results (e.g., average time, standard deviation) in JSON files, including metadata.

Implementation Details:

- **Metadata:**
    - Metadata includes:
        - **process_measured**: Name or description of the process being benchmarked (e.g., memory allocation).
        - **number_of_tests**: Total number of tests conducted.
        - **passed_tests**: Number of tests that passed.
        - **outlier_threshold**: Threshold for outlier detection in the results.
        - **programming_language**: The language used for the benchmark (e.g., C, C++, Java).
        - **array_size**: Size of the array used in the benchmark (if applicable).
        - **average_time**: The average execution time across tests.

        - **std_deviation**: Standard deviation of the times measured.

- **C Implementation**:
    - The results (such as `average_time`, `std_deviation`, `number_of_tests`, `passed_tests`, etc.) are stored in a C JSON object.
    - Data is written to a file using `fprintf` to ensure that the JSON format is correctly structured. If the file already exists, the new results are appended; otherwise, a new file is created. If isFirstEntry is false, a comma is added before appending the next JSON object to the file. (the array format is hardcoded, in main function the [ ] are appended to the start and end of the file) .
- **C++ Implementation**:
    - The C++ implementation uses the `nlohmann::json` library (referred to as `ordered_json` in the provided code) to create structured JSON objects. It reads an existing JSON file (if available) and appends the new results in an ordered manner keeping the order of insertion and not the alphabetical order. The result object is added to the existing JSON array (json.push_back(result)), whether the file was previously empty or not.
    - The function checks if the JSON file exists. If it does, the data is read and parsed into a JSON object. If not, an empty array is initialized. Then, the new benchmarking data is appended to this JSON object (array), and the file is saved using `std::ofstream`.
- **Java Implementation**:
    - In Java, the implementation makes use of `org.json.JSONObject` and `org.json.JSONArray` to create structured JSON objects.
    - The function creates a `JSONObject` for each benchmark result, including the metadata.
    - Results are saved in individual JSON files (e.g., `static_access.json`, `dynamic_access.json`) using the `FileWriter` class. These files are organized by process type (static or dynamic memory access), and each file contains the benchmark results in a structured format that makes it easy to retrieve and analyze later.

5.3.2. Combining JSON Files

- **Function Name**: `combineJSONFiles`
- **Purpose**: Consolidates multiple JSON files into a single output file for unified analysis.

Implementation Details:

- **C Implementation**:
    - The C implementation would require reading each input JSON file (using a JSON parsing library like `cJSON`) and extracting the contents.
    - The contents from each file are read into a C data structure (an array of JSON objects).
    - After all files are processed, the combined results are written to an output file using `fprintf`. This file will contain all the results from the individual files in a unified JSON array.
- **C++ Implementation**:
    - The C++ version follows a similar approach to the C implementation but uses the `nlohmann::json` library for easier handling of JSON data.

    - It iterates through each input file (`filenames`), reads the contents into a `ordered_json` object, and appends the entries from each file into the `combinedResults` array.
    - The `ordered_json::array()` is used to store the combined results. After reading and appending all the entries from the files, the combined data is written into a new JSON file (`outputFilename`), formatted for easy readability using the `dump(4)` method.
- **Java Implementation**:
    - The Java implementation reads each individual JSON file using `FileReader` and `org.json.JSONTokener`.
    - For each file, it parses the content into a `JSONArray`. Each element of the array represents an individual result entry.
    - After reading all files, the entries from all the `JSONArray` objects are merged into a single `JSONArray` that holds all benchmark results.
    - The combined results are then saved into a new output file (`results.json`) using `FileWriter` and the `toString(4)` method to pretty-print the JSON with an indentation of 4 spaces.

How it combines the files:

- For each file in the list of input filenames:
    - The file is opened and read into a `JSONArray`.
    - Each entry (a `JSONObject`) from the input file is added to the `combinedResults` array.
    - After all files are processed, the combined array is dumped into a new output file.
- The resulting `results.json` file contains a complete, unified dataset, with each benchmark result from every input file included as individual entries in the array. This consolidated file is used by the `GraphingModule` to generate visual comparisons of the benchmark data.

5.4. Statistical Analysis Functions

Statistical functions calculate averages, standard deviations, and remove outliers, providing reliable data for performance comparisons.

5.4.1. Average Calculation

- **Function Name**: `calculateAverage`
- **Purpose**: Computes the mean time across multiple test iterations.
- **Implementation Details**:
    - **Implementation in All Languages**: An array of recorded times is summed, then divided by the array's length to get the mean. This simple average smooths out minor fluctuations, yielding a stable metric for each process measured.

5.4.2. Standard Deviation Calculation

- **Function Name**: `calculateStandardDeviation`

- **Purpose**: Determines variability in time measurements, providing insight into performance consistency.
- **Implementation Details**:
    - **Implementation in All Languages**: Using the mean as a baseline, each value's deviation from the mean is squared and averaged. The square root of this average yields the standard deviation, indicating whether timing is consistent or varies widely due to factors like system load.

5.4.3. Outlier Removal

- **Function Name**: `removeOutliers`
- **Purpose**: Filters out extreme values to prevent skewing of results.
- **Implementation Details**:
    - **Implementation in All Languages**: After calculating the mean and standard deviation, values beyond a specified threshold from the mean are removed from the data set. This ensures anomalies like sudden system interruptions do not distort the benchmark results.

This detailed implementation covers each major function developed, describing its purpose, language-specific details, and impact on the overall benchmarking process. Each function works together to capture, process, and store precise performance data across languages, offering a comprehensive cross-language benchmarking solution.

Bibliography

1. Sheldon, R. (2022). *Memory Management*. Available at: https://www.techtarget.com/whatis/definition/memory-management
2. Frank, D. (2024). *C-Memory Management & Allocation*. Available at: https://medium.com/@frankokey469/c-memory-management-allocation-99ccb4f36386
3. SoftYoi LLP. (2024). *Memory Management in Java*. Available at: https://www.linkedin.com/pulse/memory-management-java-softyoi-llp-wxrsf/
4. Bigelow, S. J. (2019). *Threads*. Available at: https://www.techtarget.com/whatis/definition/thread
5. Kumar, P. (2023). *Difference between Thread Context Switch and Process Context Switch*. Available at: https://www.tutorialspoint.com/difference-between-thread-context-switch-and-process-context-switch
6. Balakrishman, S., & Pattabiraman, K. (2013). *Migration of Threads Containing Pointers in Distributed Memory Systems*. Available at: https://pages.cs.wisc.edu/~saisanth/papers/hipc00.pdf
7. Ferreira, C. R. (2020). *8 Ways to Measure Execution Time in C/C++*. Available at: https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9
8. Baeldung. (2023). *Guide to JNI (Java Native Interface)*. Available at: https://www.baeldung.com/jni
9. Barbara Illowsky & Susan Dean (2013). *Introductory Statistics.*