INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT087-3-3

REAL TIME SYSTEMS

APU3F2402CS(DA)

HANDOUT DATE:	**20 MARCH 2024**
HAND-IN DATE:	**9 JUNE 2024**
WEIGHTAGE:	**60%**

NAME:	**SIAH YAO LIANG**
TP NUMBER:	**TP061006**
LECTURER:	**DR. IMRAN MEDI**
INTAKE:	**APU3F2402CS(DA)**

Table of Contents

Abstract

In todays security environments , missile technology has becoming modern and more complex, where causing such a big threat to the opponent . The creation of a real time missile attack and defence simulation system is very crucial for every country . In this project , A missile attack and defence simulation real time system will be developed by using Rust programming languages which excels at concurrency , memory safety and performance. The simulation system is designed to dynamically test different defence situations, optimize countermeasures, and enhance response times. The ultimate goal is to improve national security by giving the military a reliable and efficient tool for training and preparedness against missile attacks. Beside , in this report , I will also perform a benchmarking report to compare the performance with another missile attack and defence system that has done by my partner CHONG DIC SUM in order to compare the system efficiency , reliability and consistency.

1.0 Introduction

Air defense systems are complex networks of interconnected subsystems tasked with protecting airspace and fighting aerial threats in the military defense domain. Similar to the high stakes seen in commercial aviation, these technologies hold enormous responsibility for saving lives and national security. (Boyd, 2022). Real-time systems (RTS), which process and respond to events in real time, are the primary technical way of implementing a comprehensive defense system. Real-time functionality is particularly significant in missile defense. The time available to notice a missile launch and perform a counter-tactical measure is usually limited; thus, immediate detection is critical.

The primary goal of this project is to create a real-time military air defense system simulation using the Rust programming language, which captures the complexities of timely job execution, system predictability and reliability, and effective resource use. The investigation of the creation of such systems raises fundamental questions about the design factors that influence the delay between sensor detection and defensive response, as well as ways for improving system safety and dependability. Central concerns include the use of programming approaches that aim to reduce response times and improve overall system performance while also assuring resilience to unforeseen threats and hostile activities.

Key Questions:

I. How can using Rust concurrency characteristics improve the missile defense system's real-time responsiveness and effectiveness?

II. How do Rust's memory safety and error handling features improve defence system reliability and robustness against attacks and vulnerabilities?

Problem statement

In an era where global security threats are becoming more sophisticated, the need for advanced defence system is greater than ever. A good and effective real-time missile attack and defence simulation system is crucial for improving the country security and preparing defence military protocal for possible incoming attacks. Traditional defence strategies and simulations often lack the speed and accuracy required to counter modern missile threats, which are characterized by their rapid deployment and advanced evasion techniques. A real-time simulation system would offer a dynamic and interactive platform for testing different defence scenarios, determining response times, and optimizing countermeasures.

2.0 Literature Review

How rust helps in develop real time system

Rust is a powerful language for developing a real time system due to its ability to ensure predictability and performance which both of this are critical for real time sensitive applications. Beside , Rust has also address the some key aspect of a real time system which is time , robustness, reliability and efficiency through its combination of core features and language design.

Below are some of the reason why Rust is good in building real time system :

1. **Time and reliability**

   Execution time is first factor that should be consider in a real time system because these system need to respond to events within a strict and predictable time frame. Rust do well in its predictable time execution. This can be achieved by the static analysis which lead the compiler to identify the potential bottleneck and thus optimize memory access pattern at compiler time. beside that , the absence of garbage collection also make sure that rust has always enough space and has a expected execution time unlike other programming languages that has garbage collection causing the unpredictable pauses (Bugden et al., 2022).

2. **Robustness**

   Robustness refers to the system ability to withstand unexpected events errors and external pressure while still maintaining it functionalities is functioning and meet it requirements. Rust has its own unique system design to handle the robustness issue which is to ensure their memory safety through rust ownership and borrowing architecture. Unlike other programming language C and C++ which are prevalent in real time system but prone in memory error , Rust has enforces the memory safety through its ownership and borrowing system . This special system us the cornerstone of rust memory safety and performance guarantees. By using this , rust can prevents common memory related error like dangling pointers, user-after-free, and double free which lead to more stable and predictable program (Sharma et al., 2023).

3. **Efficiency**

   Efficient refers to the ability to utilize resource effectively in a real time system . A good real time system can manage resource effectively and deliver timely response within strict time constraints. In a word, its about maximizing performance while minimizing resource consumption. Rust has implement the Zero-Cost Abstraction and fine-grained control techniques which is a powerful abstraction for memory management without incurring runtime overhead. This action can avoid the need for explicit memory management like C , contributing to efficient resource usage. Moreover , rust also allow fine grained control over memory allocation and deallocation which enabling developer to optimize performance in resource - constrained real time environments (Gupta et al., 2023).

Rust Enhancing the Real Time System with Robust Security

Rust is now supported by PikeOS, a real-time operating system and hypervisor with strong industry security certifications. This integration enables Rust applications to run directly on PikeOS, eliminating the requirement for a guest OS or interfaces such as POSIX, improving resource efficiency and allowing certification against safety and security standards. PikeOS provides vital features such a Certifiable File System and communication ports, which are essential for applications in high-security contexts such as medical technology, aviation, automotive, rail, military and industrial industries. Rust's strong data type security and preventive security principles build upon this foundation, making it ideal for real-time systems that require both safety and security. This combination addresses the growing overlap between functional safety and cybersecurity, resulting in a strong platform where Rust's resistance to typical vulnerabilities such as buffer overflows improves system defense against cyber attacks (Klein-Winternheim, Rust now available for Real-Time Operating System and Hypervisor PikeOS, 2023).

3.0 Design and Methodology

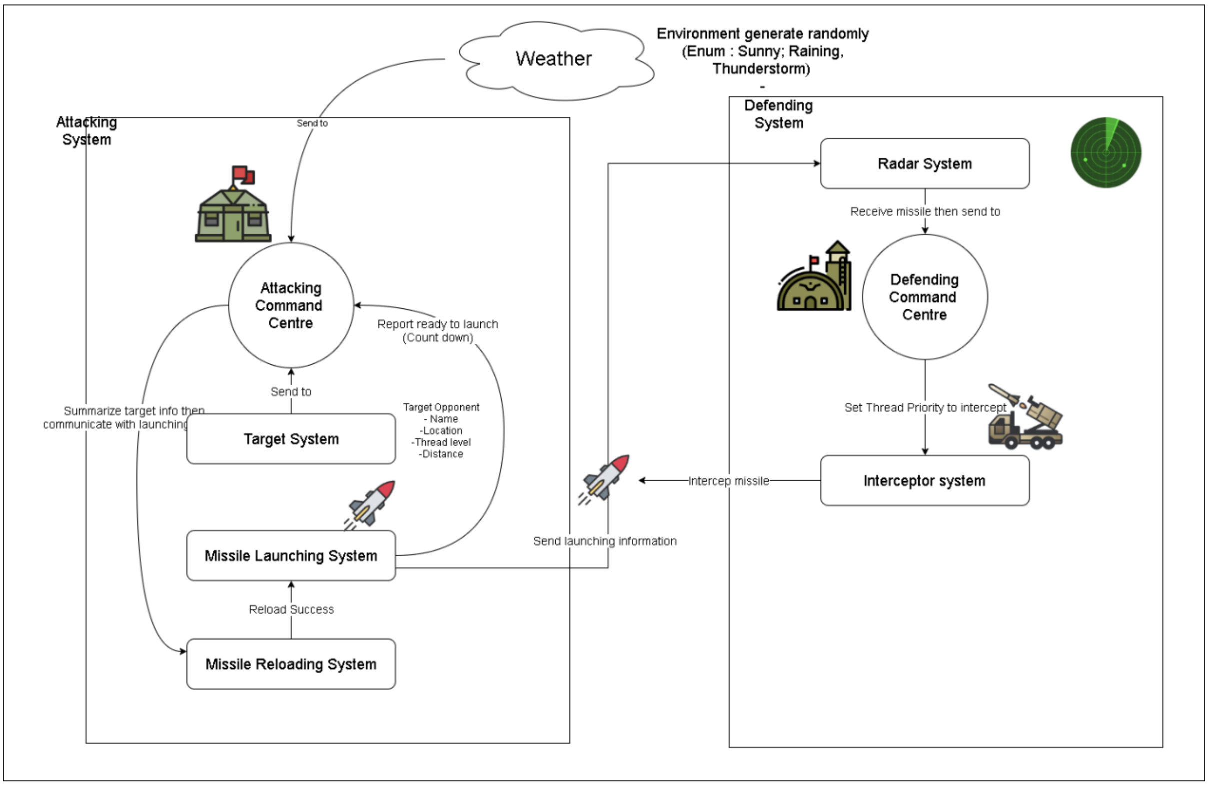


Figure 1 : Initial Design For Current Project

Weather System

- An Enum that contain weather type (Sunny , Rainy , Thunderstorm)
- Random generate weather type with respective temperature and humidity
- Create a thread to send generated weather information to **Attacking Command Centre** using channel**.**

Missile Attacking System

Targeting System

- Generate a target which contain opponent name, location (longitude and latitude), thread level and distance
- Communicate the target data to Attacking Command Centre using channel.

Attacking Command System

- Receive weather information from weather system.
- Receive targeting information from targeting system.
- Make missile decision based on target thread level
- If thread level is 3 then launch 2 (Cruise missile and hypersonic missile) missile ,otherwise launch 1 (Cruise missile) missile

- Send the decision to Missile Reloading system using channel

Missile Reloading System

- Receive decision from attacking command centre
- Prepare missile then send confirmation to Missile Launching System

Missile Launching System

- Receive confirmation from missile reloading system
- Start launching missile count down 10 second.
- Then launch missile and send message to radar system using RabbitMQ.

Missile Defending System

Radar System

- Receive missile launched message from missile launching system.
- Send missile information to defending command centre.

Defending Command Centre

- Receive missile information from radar system .
- Make decision to set thread priority
- Set priority to intercept hypersonic missile .
- Send decision to intercept system .
- Countdown for missile impact time
- Validate the missile status , if status = “intercepted” then break the threat otherwise continue the countdown.

Intercept System

- Receive decision from defending command centre
- Then set thread priority.
- Intercept missile .
- Change the missile status and update to defending command centre

4.0 Result and Discussion

4.1 Current Program Execution

StructFile.rs

```
#[derive(Debug)]
1 implementation
pub enum weather_type {
    Sunny,
    Rainy,
    Thunderstorm
}

impl Distribution<weather_type> for Standard {
    fn sample<R:Rng + ?Sized>(&self, rng:&mut R) -> weather_type{
        match rng.gen_range(0..=2){
            0 => weather_type::Sunny,
            1 => weather_type::Rainy,
            _ => weather_type::Thunderstorm
        }
    }
}

#[derive(Debug)]
2 implementations
pub struct WeatherData {
    pub weather: weather_type,
    pub temperature : u8,
    pub humidity :u8
}
impl WeatherData {
    pub fn new(weather: weather_type) -> Self{
        match weather {
            weather_type::Sunny =>Self{
                weather,
                temperature : 32,
                humidity : 83
            },
            weather_type :: Rainy =>Self{
                weather,
                temperature :30,
                humidity : 88
            },
            weather_type ::Thunderstorm =>Self{
                weather,
                temperature :30,
                humidity :85
            }
        }
    }

    pub fn randomWeather() ->Self {
        let mut rng: ThreadRng = rand::thread_rng();
        let weather: weather_type = rng.gen::<weather_type>();
        Self::new(weather)
    }
} impl WeatherData
```

Figure 2 : Struct.rs file

```
#[derive(Debug)]
#[derive(Clone)]

pub struct TargetInfomation {
    pub name : String,
    pub location: (f64,f64),
    pub threat_level: u8,
    pub distance : u64
}

#[derive(Debug,Deserialize,Serialize, Eq, PartialEq, Ord, PartialOrd

pub struct MissileStatus {
    pub name: String,
    pub status: String,  // Status might include "launched", "intercep
}
```

The structfile.rs is used to store all the data structures and related implementations for the weather ,target navigator and missile defense system. It includes weather_type, WeatherData, TargetInfomation, and MissileStatus structs. These structs represent weather conditions, target information, and missile status, with methods to generate random weather and initialize instances.

Weather

Weather.rs

```rust
pub fn generate_weather_data(tx:Sender<WeatherData>){
    let random_weather: WeatherData = WeatherData::randomWeather();
    println!("[Weather Analyser] - First Starting analyse current wather .");
    for mut i: i32 in 0..3{
        println!("Analysing weather ...");
        i+=1;
        thread::sleep(dur: Duration::from_secs(1));
    }
    println!("[Weather Analyser] - Data collected : Weather: {:?}, Temperature: {}C, Humidity: {}%",
        random_weather.weather, random_weather.temperature, random_weather.humidity);
    println!("[Weather Analyser] - Sending weather data to Command Centre. Please wait ... " );
    thread::sleep(dur: Duration::from_secs(3));
    tx.send(random_weather).unwrap();

}
```

Figure 3 : Generate Weather Data File

Image above is a generate weather data function located in weather.rs file. It uses a thread to simulate weather data analysis and transmission. It generates random weather data, prints progress messages while analysing over 3 seconds, and sends the collected data to a command center through a channel, pausing briefly during the process.

Missile Attacking System

Target_navigation.rs

```rust
pub fn targetOpponent(target:TargetInfomation){
    let mut rng: ThreadRng = rand::thread_rng(); // Create a random number generator

    println!("[Target navigator] - Name: {}", target.name);
    thread::sleep(dur: Duration::from_millis(rng.gen_range(0..1000))); // Sleep for 0 to 999 milliseconds randomly

    println!("[Target navigator] - Location: (Latitude ({}), Longitude ({}))", target.location.0, target.location.1);
    thread::sleep(dur: Duration::from_millis(rng.gen_range(0..1000)));

    println!("[Target navigator] - Threat Level: {}", target.threat_level);
    thread::sleep(dur: Duration::from_millis(rng.gen_range(0..1000)));

    println!("[Target navigator] - Distance: {} km", target.distance);
    thread::sleep(dur: Duration::from_millis(rng.gen_range(0..1000)));
}
```

```
pub fn generateTarget (tx: Sender<TargetInfomation>){
    let target: TargetInfomation = TargetInfomation{
        name :"Robert Enemy base".to_string(),
        location: (3.0554,101.7006),
        threat_level: 3,
        distance : 4500
    };
    println!("[Target Navigator] - Collecting Target Infomation ...");
    thread::sleep(dur: Duration::from_secs(2));
    targetOpponent(target.clone());
    println!("[Target Navigator] - Data collected :Name :{}, Location: ({},{}), Threat Level : {}, Distance :{} Km",
    target.name, target.location.0,target.location.1 ,target.threat_level, target.distance);
    println!("[Target Navigator] - Sending target data to Command Centre. Please wait ... " );
    thread::sleep(dur: Duration::from_secs(3));

    tx.send(target).unwrap();
}
```

Figure 4: Target Generation File

The code defines two functions, targetOpponent and generateTarget, to simulate the process of collecting and sending target information in a threaded environment. The generateTarget function initializes a TargetInfomation struct with details of a target, including its name, location, threat level, and distance. It prints messages indicating the start of target data collection and simulates a delay. The targetOpponent function further processes this target information, printing each attribute with random sleep intervals to mimic the time taken for each step. After processing, the generateTarget function sends the collected target data to a command center through a channel, simulating thread to sleep for 3 seconds.

Atk_command_centre.rs

```
pub fn reciveData(rx_weather:Receiver<WeatherData>,rx_target:Receiver<TargetInfomation>,tx_decison:Sender<bool>) {
    let mut weather_received: bool = false;
    let mut target_received: bool = false;

    while !weather_received || !target_received {
        if !weather_received {
            match rx_weather.try_recv() {
                Ok(weather_info: WeatherData) => {
                    println!("[Attack Command Centre] - Received weather data: {:?}", weather_info);
                    weather_received = true; // Set the flag to true as weather data has been received
                },
                Err(_) => {
                    println!("[Attack Command Centre] - Weather data has not been received yet.");
                }
            }
        }
        if !target_received {
            match rx_target.try_recv() {
                Ok(target_info: TargetInfomation) => {
                    println!("[Attack Command Centre] - Received target information: {:?}", target_info);
                    target_received = true; // Set the flag to true as target data has been received
                    make_decision(target_info,tx_decison.clone());
                },
                Err(_) => {
                    println!("[Attack Command Centre] - Target information has not been received yet.");
                }
            }
        }
        // Sleep briefly to prevent this loop from consuming too much CPU
        std::thread::sleep(dur: std::time::Duration::from_secs(3));
    }
} fn reciveData
```

```
pub fn make_decision (target_info:TargetInfomation,tx_decison:Sender<bool>){
    // let (tx_command,rx_command) = channel::<bool>();
    let attack_priorities :bool = match target_info.threat_level {
        3 => true,
        _ => false
    };
    if attack_priorities {
        println!("[Attack Command Centre] - Attack priority is high.");
    }else {
        println!("[Attack Command Centre] - Normal threat level. Stadard attack protocol.");
    }
    tx_decison.send(attack_priorities).unwrap()
}
```

Figure 5 : Attack Command Centre Code

The reciveData function continuously checks and receives weather and target information from respective channels. Once both data types are received, it prints the details and triggers the make_decision function. The make_decision function evaluates the threat level based on the target information. If the threat level is high, it sends a high-priority attack decision to the missile reloading system through a channel, otherwise, it sends a standard protocol decision.

Missile_reload_system.rs

```
0 implementations
pub struct Missile {
    name : &'static str,
    speed_mph :u32,
    attack_damage :&'static str
}
// Implement the reloading functionality for missiles
1 implementation
pub struct MissileReloadingSystem {
    tx_launcher: Sender<String>,  // Channel to communicate with the missile launcher system
}
impl MissileReloadingSystem{
    pub fn new(tx_launcher: Sender<String>) -> Self {
        MissileReloadingSystem { tx_launcher }
    }
    pub fn prepareMissiles(&self, rx_decision:Receiver<bool>){
        let hypersonic_missile: Missile = Missile {
            name: "Hypersonic missile",
            speed_mph: 3800,
            attack_damage: "High",
        };
        let cruise_missile: Missile = Missile {
            name: "Cruise missile",
            speed_mph: 767,
            attack_damage: "Low",
        };
```

```
        match rx_decision.recv() {
            Ok(decision: bool) => {
                if decision{
                    println!("[Missile Reloading System]: High threat detected. Preparing all missiles.");
                    self.tx_launcher.send(format!("Prepare {}", hypersonic_missile.name)).unwrap();
                    self.tx_launcher.send(format!("Prepare {}", cruise_missile.name)).unwrap();
                }else {
                    println!("[Missile Reloading System]: Standard threat detected. Preparing cruise missile.");
                    // Send a message to the missile launcher system to launch the cruise missile
                    self.tx_launcher.send(format!("Prepare {}", cruise_missile.name)).unwrap();
                }
            },
            Err(_) => {println!("[Missile Reloading System]: Error in receiving decision.")}
        }
    } fn prepareMissiles
} impl MissileReloadingSystem
```

Figure 6 : Missile Reloading Code

The MissileReloadingSystem struct manages missile preparation based on received threat levels. It includes a method, prepareMissiles, which listens to a decision channel. If a high-priority threat is detected (receives true), it prepares both the hypersonic and cruise missiles, sending preparation commands to the missile launcher system. If the threat is standard (false), it only prepares the cruise missile. The system communicates with the missile launcher through a channel, ensuring that the appropriate missiles are ready to launch based on the received threat assessment from the command center.

Missile_launcher.rs

```
pub fn missile_launcher(rx_launcher: Receiver<String>) {
    println!("[Missile Launcher System] - Ready to receive launch commands.");

    let mut handles: Vec<JoinHandle<()>> = Vec::new();
    // Listen for incoming preparation commands from the Missile Reloading System
    for command: String in rx_launcher {
        let handle: JoinHandle<()> = thread::spawn(move || { // Ensure to capture the JoinHandle correctly
            match command.as_str() {
                "Prepare Hypersonic missile" => {
                    println!("[Missile Launcher System] - Preparing Hypersonic missile...");
                    // thread::sleep(Duration::from_secs(3));

                    count_down_missile(missile_name: "Hypersonic Missile");
                },
                "Prepare Cruise missile" => {
                    println!("[Missile Launcher System] - Preparing Cruise missile...");
                    count_down_missile(missile_name: "Cruise Missile");
                },
                _ => {
                    println!("[Missile Launcher System] - Unknown command received: '{}'", command);
                }
            }
        });
        handles.push(handle);
    }
    for handle: JoinHandle<()> in handles {
        handle.join().unwrap();
    }
} fn missile_launcher
```

```
pub fn send_missile_info_to_radar(missile_name:&str){
    // Create a missile status struct with the information
    let missile_info: MissileStatus = MissileStatus {
        name: missile_name.to_string(),
        status: "launched".to_string(),
    };

    // Serialize the missile info to JSON
    let serialized_info: String = serde_json::to_string(&missile_info).unwrap();

    // Send the serialized information to the radar queue
    send(msg: serialized_info, queue_addr: "missile_notifications").unwrap();
}
```

Figure 7 : Missile Launcher System Code

The missile_launcher function receives missile preparation commands from the MissileReloadingSystem via a channel. Upon receiving a command, it spawns a thread to handle the countdown and preparation for each missile in parallel. The count_down_missile function initiates a 10-second countdown, logging each second. After the countdown, it announces the missile launch. The send_missile_info_to_radar function then creates a MissileStatus struct with the missile's name and status set to "launched", serializes this information to JSON, and sends it to the radar system by using RabbitMQ send function. This ensures that missile preparations and launches are efficiently managed and communicated.

Missile Defending System

Radar_system.rs

```
pub fn radar_system() {
    println!("[Defense Radar System] - Radar has activated. ");
    let mut detected_missiles: i32 = 0;

    loop {
        // Use the existing `receive` function which returns a string
        let notification: String = receive(queue_name: "missile_notifications");
        let missile_status: Result<MissileStatus, Error> = from_str::<MissileStatus>(&notification);

        // Check if notification is not empty then attempt to deserialize
        if !notification.is_empty() {
            match missile_status{
                Ok(missile: MissileStatus) => {println!("[Defense Radar System] - Scanning... Incoming Missile Detecte
                send_to_command_centre(missile_info: missile);
                detected_missiles += 1;
                if detected_missiles == 2 {
                    break;
                };
            },
                Err(e: Error) => println!("[Defense Radar System] - Failed to deserialize missile data: {}", e),
            }
        } else {
            println!("[Defense Radar System] - Scanning... No threat found.");
        }
        // Pause for 5 seconds before the next scan.
        thread::sleep(dur: Duration::from_secs(3));
    }
} fn radar_system
```

```
pub fn send_to_command_centre(missile_info : MissileStatus){
    println!("[Defense Radar System] - Sending missile data to Command Centre... {:?}", missile_info);
    let serialized_info: String = serde_json::to_string(&missile_info).unwrap();
    // Send the serialized information to the radar queue
    send(msg: serialized_info, queue_addr: "missile_to_command_centre").unwrap();
}
```

Figure 8 : Radar Function Code

The radar system runs as a continuous thread, waiting for messages from the missile launcher system via RabbitMQ. When it receives a missile notification, it processes the message to detect potential risks. If a missile is spotted, the radar system analyses it and sends the information to the defense command center. This ensures fast notifications and responses to any incoming missile threats while maintaining constant awareness.

Dfd_command_centre.rs

```
pub fn dfd_command_centre(){
    // let notification = receive("missile_to_command_centre");
    // let missile_status = from_str::<MissileStatus>(&notification);
    let mut detected_missiles: i32 = 0;

    loop {
        // Use the existing `receive` function which returns a string
        let notification: String = receive(queue_name: "missile_to_command_centre");
        let missile_status: Result<MissileStatus, Error> = from_str::<MissileStatus>(&notification);

        // Check if notification is not empty then attempt to deserialize
        if !notification.is_empty() {
            match missile_status{
                Ok(missile_status: MissileStatus) => {
                    println!("[Defense Command Centre] - Missile Information Received, Analyzing Missile ....");
                    // detected_missiles += 1;
                    let serialized_info: String = serde_json::to_string(&missile_status).unwrap();
                    send(msg: serialized_info, queue_addr: "missile_to_interceptor").unwrap();

                    thread::spawn(move||{count_down_missile_arrived(missile_status); });
                },
                Err(_)=> println!("[Defense Command Centre] - Failed to deserialize missile data"),
            }
        }
        // Pause for 5 seconds before the next scan.
        thread::sleep(dur: Duration::from_secs(5));
    };

} fn dfd_command_centre
```

```
pub fn count_down_missile_arrived(missile_status :MissileStatus){
    static INTERCEPTED_COUNT: AtomicUsize = AtomicUsize::new(0);
    const TOTAL_MISSILES: usize = 2; // Total number of missiles
    const intercepted:usize = 0;
    let count_down_time: i32 = if missile_status.name == "Hypersonic Missile" {30}else{50};
    for i: i32 in (1..=count_down_time).rev() {
        println!("[Command Centre] - Estimated {} seconds until impact for {}.", i, missile_status.name);
        thread::sleep(dur: Duration::from_secs(1));
        let notification: String = receive(queue_name: "missile_status_updates");
        let updated_status_result: Result<MissileStatus, Error> = from_str::<MissileStatus>(&notification);
        if !notification.is_empty(){
            match updated_status_result {
                Ok(MissileStatus: MissileStatus) =>{println!("[Command Centre] - {:?}",MissileStatus);
                if MissileStatus.status == "intercepted" {
                    println!("[Defense Command Centre] - Missile intercepted! Countdown stopped for {}.", missile_sta
                    // break;
                    INTERCEPTED_COUNT.fetch_add(val: 1, order: Ordering::SeqCst)+1;
                    if INTERCEPTED_COUNT.load(order: Ordering::SeqCst) == TOTAL_MISSILES {
                        println!("All missiles intercepted. Terminating the program.");
                        std::process::exit(code: 0); // Terminate the program
                    }
                    return; // Terminate the thread
                };
            },
                Err(_) =>{println!("errrrr")}
            }
        }
    }
    println!("[Defense Command Centre] - Missile impact! Base hit by {}.", missile_status.name);
} fn count_down_missile_arrived
```

Figure 9 : Defending Command Centre Code

The defense command center operates in a continuous loop, waiting for missile information from the radar via RabbitMQ. Upon receiving a missile notification, it sends the missile data to the interceptor system for interception. Simultaneously, it spawns a thread to execute the count_down_missile_arrived function, which counts down the missile's arrival time. The countdown will continue unless an update indicating the missile has been intercepted is received, at which point the countdown stops and the missile is marked as intercepted. If no interception occurs, the countdown completes, and a message is printed indicating that the base has been hit.

Interceptor_system.rs

```rust
pub fn interceptor_system(){

    let priority_queue: Arc<Mutex<PriorityQueue<MissileStatus, …>… = Arc::new(Mutex::new(PriorityQueue::new()));

    loop {
        // Receive missile information from command center or radar system
        let notification: String = receive(queue_name: "missile_to_interceptor");
        let missile_status: Result<MissileStatus, Error> = from_str::<MissileStatus>(&notification);
        if !notification.is_empty() {
            match missile_status {
                Ok(missile_status: MissileStatus) => {
                    println!("[Interceptor System] - Interceptor recieve {} data",missile_status.name);
                    // Determine priority (higher priority value for hypersonic missiles)
                    let priority: i32 = if missile_status.name == "Hypersonic Missile" { 10 } else { 1 };

                    // Add missile status to the priority queue
                    let mut pq: MutexGuard<PriorityQueue<…, …>> = priority_queue.lock().unwrap();
                    pq.push(item: missile_status, priority);
                },
                Err(e: Error) => {
                    println!("[Interceptor System] - Failed to deserialize missile data: {}", e);
                }
            }
        }
```

```rust
        // Process missiles based on priority
        {
            let mut pq: MutexGuard<PriorityQueue<…, …>> = priority_queue.lock().unwrap();
            while let Some((mut missile_status: MissileStatus, _)) = pq.pop() {
                // Perform interception logic
                println!("[Interceptor System] - Intercepting missile: {:?}", missile_status);
                thread::sleep(dur: Duration::from_secs(5)); // Simulate time taken to intercept
                missile_status.status = "intercepted".to_string();

                // Send updated missile status back to the command center or radar system
                let serialized_missile: String = serde_json::to_string(&missile_status).unwrap();
                // send(serialized_missile, "missile_status_updates").unwrap();
                if let Err(e: Error) = send(msg: serialized_missile, queue_addr: "missile_status_updates") {
                    println!("[Interceptor System] - Failed to send updated missile status: {}", e);
                } else {
                    println!("[Interceptor System] - Updated missile status sent.");
                }
            }
        }

        // Pause for a short duration before the next scan.
        thread::sleep(dur: Duration::from_secs(1));
    }

} fn interceptor_system
```

Figure 10 : Interceptor Function

The interceptor system continuously runs in a loop, waiting for missile information from the command center or radar system via RabbitMQ. Upon receiving missile data, it determines the priority based on the missile type (higher priority for hypersonic missiles) and adds it to a priority queue. The system then processes missiles from the queue based on priority, simulating a 5-second interception time. After interception, it updates the missile status to "intercepted" and sends this update back to the command center .This ensures high-priority threats are intercepted first, and the status is communicated effectively.

4.2 Criterion Benchmark

Criterion Report

Criterion.rs Benchmark Index

See individual benchmark pages below for more details.

- Attack Command Centre System
  - Generate Target Information
  - Missile Launcher System Receive Launcher Data and Send To Radar
  - Missile Reloading System Receive Threat Level
  - Receiving Weather and Target Information
- Defence Command Centre System
  - Defense Command Centre Receive Incomming Missile Data
  - Defense Command Centre Receive The Missile Updated Status and Stop Countdown
  - Interceptor System Receive and Intercept Incomming Missile Data
  - Radar System Receive Threat Information
- Overall Attack System
  - Overall Attack System
- Overall Defence System
  - Overall Defence System
- Overall System
  - Overall System
- Weather System
  - Weather System Generated weather data

This report was generated by Criterion.rs, a statistics-driven benchmarking library in Rust.

Figure 11 : Criterion Report Index For Current System

In this report I have use criterion to benchmark the index for all the function that implement in our missile attack and defence simulation. This report Provides a detailed overview of performance standards for numerous systems, such as the Attack Command Centre, Defence Command Centre, Overall System, and Weather Systems. Each category provides particular functions that were tested for throughput and latency, providing detailed insights into system performance .

Benchmark.rs

```rust
struct LatencyBenchmark {
    times: Vec<Duration>,
}

impl LatencyBenchmark {
    fn new() -> Self {
        LatencyBenchmark { times: Vec::new() }
    }

    fn op_start(&self) -> Instant {
        Instant::now()
    }

    fn op_finish(&mut self, start_time: Instant) {
        let duration: Duration = start_time.elapsed();
        self.times.push(duration);
    }

    fn print(&self) {
        if !self.times.is_empty() {
            let total: Duration = self.times.iter().sum();
            let avg: Duration = total / self.times.len() as u32;
            let min: &Duration = self.times.iter().min().unwrap();
            let max: &Duration = self.times.iter().max().unwrap();
            println!(
                "Latency (ns) avg: {:.6}, min: {:.6}, max: {:.6}",
                avg.as_nanos(), min.as_nanos(), max.as_nanos()
            );
        } else {
            println!("No latency data recorded.");
        }
    }
} impl LatencyBenchmark
```

Figure 12 : BMA Latency Function Code

We have implemented two functions to evaluate latency: op_start captures the start time, and op_finish calculates and stores the elapsed duration. These functions enable accurate latency measurement.

```rust
fn criterion_benchmark_attack_command_centre_receive_data(c: &mut Criterion) {…

fn criterion_benchmark_generate_weather_data(c: &mut Criterion) {…

fn criterion_benchmark_generate_target_data(c: &mut Criterion) {…

fn criterion_benchmark_missile_reloading_system(c: &mut Criterion) {…

fn criterion_benchmark_missile_launcher_system(c: &mut Criterion) {…

fn criterion_benchmark_radar_system(c: &mut Criterion) {…

fn criterion_benchmark_dfd_command_centre_system(c: &mut Criterion) {…

fn criterion_benchmark_interceptor_system(c: &mut Criterion) {…

fn criterion_benchmark_dfd_command_centre_updated(c: &mut Criterion) {…

fn criterion_overall_attack_system(c: &mut Criterion) {…

fn criterion_overall_defence_system(c: &mut Criterion) {…

fn criterion_overall_system(c: &mut Criterion) {…
```

```rust
criterion_group!(benches, criterion_benchmark_attack_command_centre_receive_data,
    criterion_benchmark_generate_weather_data,
    criterion_benchmark_generate_target_data,
    criterion_benchmark_missile_reloading_system,
    criterion_benchmark_missile_launcher_system,
    criterion_benchmark_radar_system,
    criterion_benchmark_dfd_command_centre_system,
    criterion_benchmark_interceptor_system,
    criterion_benchmark_dfd_command_centre_updated,
    criterion_overall_attack_system,
    criterion_overall_defence_system,
    );
criterion_main!(benches);
```

Figure 13 : Criterion Benchmark For Different Function

In the benchmark.rs file , I have call the criterion function to run the function inside the criterion_group(). Then I will run "cargo bench" in the terminal to evaluate the performance index of each function.

Weather system

Benchmark # 1 : Weather System Generate Random Weather (Current System vs Competition System)

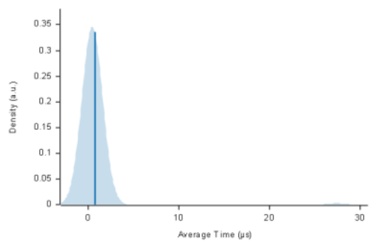
Code

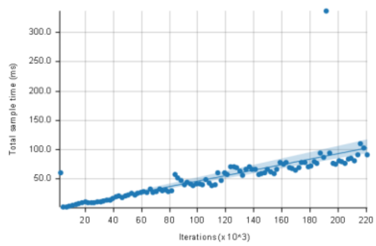
```rust
fn criterion_benchmark_generate_weather_data(c: &mut Criterion) {
    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Weather System ");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Weather System Generated weather data", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            generate_weather_data_main();
            lb.op_finish(start_time);

        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the generate_weather_data_main() function by measuring throughput with Criterion and latency with a custom LatencyBenchmark, collecting start and end times, and printing the results.

Result

```
Weather System /Weather System Generated weather data
                        time:   [428.94 ns 467.29 ns 536.15 ns]
                        thrpt:  [1.8651 Melem/s 2.1400 Melem/s 2.3313 Melem/s]
                 change:
                        time:   [-67.872% -19.554% +101.86%] (p = 0.77 > 0.05)
                        thrpt:  [-50.461% +24.308% +211.26%]
                        No change in performance detected.
Found 4 outliers among 100 measurements (4.00%)
  2 (2.00%) high mild
  2 (2.00%) high severe

Latency (ns) avg: 380, min: 200, max: 1052900
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	428.94 ns	**467.29 ns**	536.15 ns
Throughput	1.8651 Melem/s	**2.1400 Melem/s**	2.3313 Melem/s
R^2	0.0203088	**0.0209971**	0.0189290
Mean	441.64 ns	**729.28 ns**	1.2856 µs
Std. Dev.	55.044 ns	**2.6950 µs**	4.6175 µs
Median	417.17 ns	**432.70 ns**	451.32 ns
MAD	37.726 ns	**55.059 ns**	69.403 ns

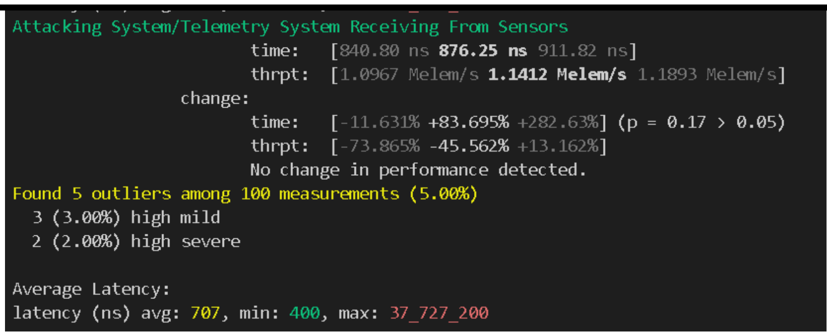
Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

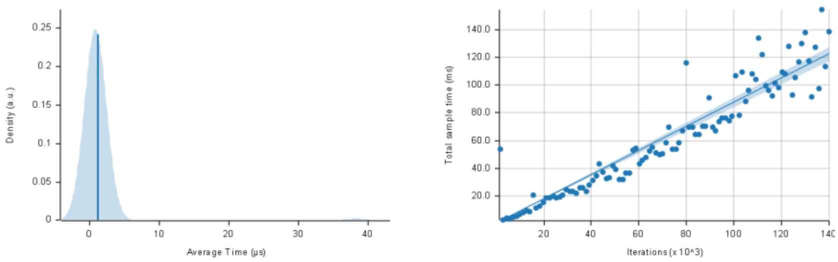
Analysis

The benchmark shows an average throughput of 2.1400 Melem/s and an average latency of 380 nanoseconds. Four outliers were found among 100 measurements, indicating occasional high latency. The overall performance is consistent with no significant change detected. Additional statistics and plots provide detailed insights into the function's performance.

Comparison System Result



```
Attacking System/Telemetry System Receiving From Sensors
                        time:   [840.80 ns 876.25 ns 911.82 ns]
                        thrpt:  [1.0967 Melem/s 1.1412 Melem/s 1.1893 Melem/s]
                 change:
                        time:   [-11.631% +83.695% +282.63%] (p = 0.17 > 0.05)
                        thrpt:  [-73.865% -45.562% +13.162%]
                        No change in performance detected.
Found 5 outliers among 100 measurements (5.00%)
  3 (3.00%) high mild
  2 (2.00%) high severe

Average Latency:
latency (ns) avg: 707, min: 400, max: 37_727_200
```



Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	840.80 ns	876.25 ns	911.82 ns
Throughput	1.0967 Melem/s	1.1412 Melem/s	1.1893 Melem/s
R^2	0.2666489	0.2767341	0.2665889
Mean	797.57 ns	1.1940 µs	1.9617 µs
Std. Dev.	123.57 ns	3.7497 µs	6.4282 µs
Median	760.15 ns	790.86 ns	823.50 ns
MAD	86.014 ns	108.57 ns	149.41 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Metric	Current System	Comparison System
Average Latency	**380 ns**	**707 ns**
Min Latency	**200 ns**	**400 ns**
Max Latency	**152900 ns**	**37_727_200 ns**
Throughput	**2.140 Melem/s**	**1.142 Melem/s**
Average Time taken	**467.29 ns**	**876.25 ns**
Outlier	**4**	**5**
Analysis	The telemetric system function works the same function as my current weather system which is generate weather . The comparison between the current and comparison systems reveals that the current system outperforms the comparison system in several key metrics. It boasts lower average and minimum latency, significantly lower maximum latency, and higher throughput. Additionally, the current system has a faster average processing time, despite both systems having a similar number of outliers. Overall, the current system offers improved performance and efficiency in handling operations.	

Missile Attack System

Benchmark # 2 : Target Navigator Generate Target Information (Current System vs Competition System)

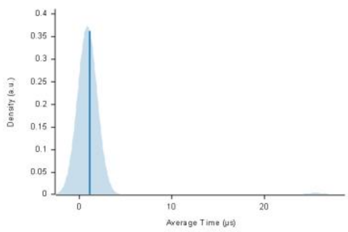
Code

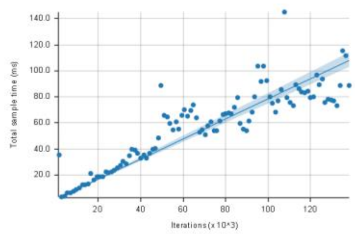
```
fn criterion_benchmark_generate_target_data(c: &mut Criterion) {
    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Attack Command Centre System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Generate Target Information", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            generateTarget_main();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the generateTarget_main() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Attack Command Centre System/Generate Target Information
                        time:   [664.42 ns 698.02 ns 733.63 ns]
                        thrpt:  [1.3631 Melem/s 1.4326 Melem/s 1.5051 Melem/s]
                 change:
                        time:   [-56.861% -0.5475% +119.35%] (p = 0.99 > 0.05)
                        thrpt:  [-54.412% +0.5505% +131.81%]
                        No change in performance detected.
Found 2 outliers among 100 measurements (2.00%)
  1 (1.00%) high mild
  1 (1.00%) high severe

Latency (ns) avg: 598, min: 300, max: 26909500
```





Additional Statistics:

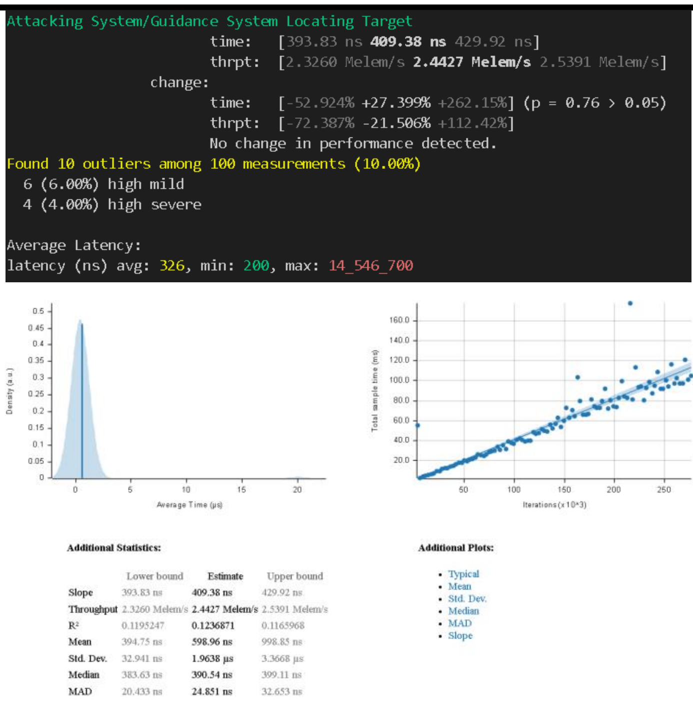
	Lower bound	Estimate	Upper bound
Slope	745.55 ns	782.44 ns	826.20 ns
Throughput	1.2104 Melem/s	1.2781 Melem/s	1.3413 Melem/s
R^2	0.0430935	0.0448435	0.0424180
Mean	845.55 ns	1.1196 µs	1.6359 µs
Std. Dev.	157.76 ns	2.4711 µs	4.2317 µs
Median	824.88 ns	855.00 ns	878.46 ns
MAD	123.94 ns	174.28 ns	210.66 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Analysis

The benchmark for generate_target_information shows an average throughput of 1.4326 Melem/s and an average latency of 598 nanoseconds. With only two outliers among 100 measurements, the function demonstrates stable performance. The latency distribution and scatter plot show constancy, without frequent spike. Overall, the function maintains reliable execution times with no significant performance changes detected, suggesting its

<table>
<tr><td></td><td>suitability for time-sensitive operations within the Attack Command Centre System.</td></tr>
<tr><td>Comparison System Result</td><td>
</td></tr>
</table>

Metric	Current System	Comparison System
Average Latency	**598 ns**	**326 ns**
Min Latency	**300 ns**	**200 ns**
Max Latency	**269500 ns**	**14_546_700 ns**
Throughput	**1.4326 Melem/s**	**2.4427 Melem/s**
Average Time taken	**698.02 ns**	**409.38 ns**
Outlier	**2**	**10**
Analysis	This section compare both target generating function in both system. The comparison system have better benchmark than the current one in terms of throughput (2.4427 Melem/s) and average time taken (409.38 µs). However, it has higher maximum latency (14,546,700 ns) and more outliers (10). The current system is more stable, with fewer outliers (2), a lower maximum latency (269,500 ns), but a larger average latency (598 ns) and a lower throughput (1.4326 Melem/s).	

Benchmark # 3 : Attack Command Centre Receiving weather and Target Information (Current System vs Competition System)

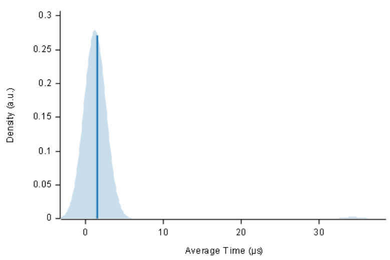
Code

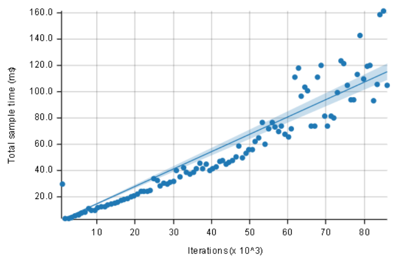
```
fn criterion_benchmark_attack_command_centre_receive_data(c: &mut Criterion) {
    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Attack Command Centre System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Receiving Weather and Target Information", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            attack_cc_receive_data();
            lb.op_finish(start_time);

        });
    });
    group.finish();
    lb.print();

}
```

This function benchmarks the attack_cc_receive_data() function by measuring throughput with Criterion and latency with a custom LatencyBenchmark, collecting start and end times, and printing the results.

Result

```
Attack Command Centre System/Receiving Weather and Target Information
                        time:   [1.4248 µs 1.4729 µs 1.5199 µs]
                        thrpt:  [657.94 Kelem/s 678.93 Kelem/s 701.83 Kelem/s]
                 change:
                        time:   [-36.664% +10.796% +85.715%] (p = 0.79 > 0.05)
                        thrpt:  [-46.154% -9.7438% +57.888%]
                        No change in performance detected.
Found 2 outliers among 100 measurements (2.00%)
  1 (1.00%) high mild
  1 (1.00%) high severe

Latency (ns) avg: 1374, min: 800, max: 4645400
```





Additional Statistics:

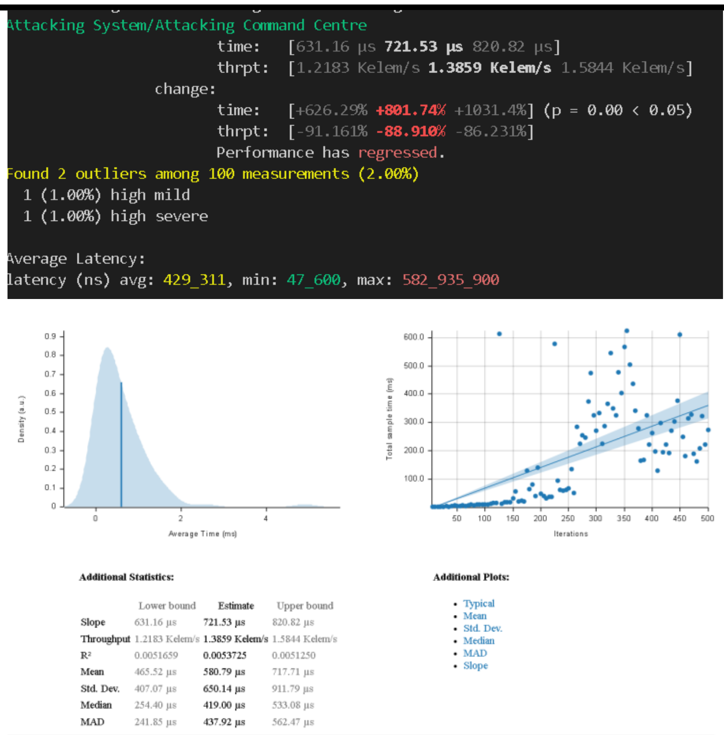
	Lower bound	Estimate	Upper bound
Slope	1.2675 µs	1.3387 µs	1.4110 µs
Throughput	708.74 Kelem/s	747.01 Kelem/s	788.96 Kelem/s
R^2	0.1087830	0.1155815	0.1085860
Mean	1.1960 µs	1.5602 µs	2.2522 µs
Std. Dev.	193.24 ns	3.3167 µs	5.6818 µs
Median	1.1032 µs	1.1469 µs	1.2165 µs
MAD	101.94 ns	159.56 ns	229.00 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Analysis

The benchmark demonstrates that the attack_cc_receive_data function runs consistently, with an average throughput of 678.93 Kelem/s and an average latency of 1.374 microseconds. The presence of a few outliers indicates

	unexpected latency, but the function still performs consistently within expected parameters.
Comparison System Result	

Metric	Current System	Comparison System
Average Latency	**1374 ns**	**429_311 ns**
Min Latency	**800 ns**	**47_600 ns**
Max Latency	**4645400 ns**	**582_935_900 ns**
Throughput (Melem/s)	**1.4729 Kelem/s**	**1.3859 Kelem/s**
Average Time taken	**698.02 µs**	**721.53 µs**
Outlier	**2**	**2**
Analysis	This section compare benchmark performance of both attack command centre system . The current system show a lower average latency that the comparison which perform faster. Beside the time taken for current system (698.02) is shorter than the comparison system (721.53). The current system's throughput (1.4729) has slightly more than the comparison system (1.3859). Both system have the same number of outlier	

Benchmark # 4 : Missile Reloading Receive Threat Level (Current System vs Competition System)

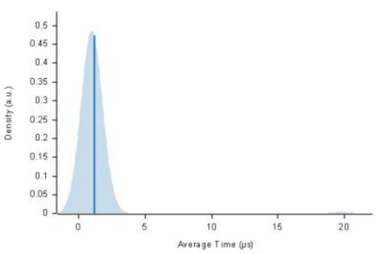
Code

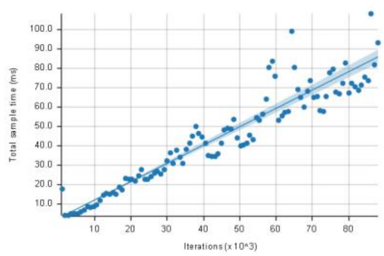
```
fn criterion_benchmark_missile_reloading_system(c: &mut Criterion) {
    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Attack Command Centre System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Missile Reloading System Receive Threat Level", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            missile_reloading_system_main();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the missile_reloading_system_main() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Attack Command Centre System/Generate Target Information
                        time:   [728.82 ns 758.10 ns 790.68 ns]
                        thrpt:  [1.2647 Melem/s 1.3191 Melem/s 1.3721 Melem/s]
                 change:
                        time:   [-58.342% +23.151% +251.81%] (p = 0.77 > 0.05)
                        thrpt:  [-71.575% -18.799% +140.05%]
                        No change in performance detected.
Found 1 outliers among 100 measurements (1.00%)
  1 (1.00%) high severe

Latency (ns) avg: 687, min: 300, max: 8079800
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	940.14 ns	977.36 ns	1.0197 µs
Throughput	980.70 Kelem/s	1.0232 Melem/s	1.0637 Melem/s
R^2	0.2502009	0.2599243	0.2474935
Mean	992.55 ns	1.2070 µs	1.6041 µs
Std. Dev.	144.94 ns	1.8753 µs	3.2067 µs
Median	938.46 ns	978.78 ns	1.0392 µs
MAD	115.33 ns	143.16 ns	192.97 ns

Additional Plots:

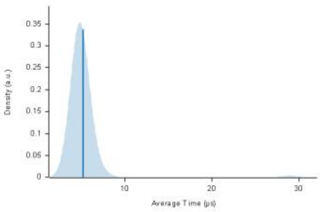
- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

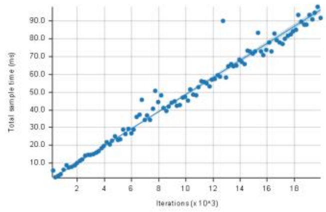
Analysis

The system has shows an average throughput of 1.3191 Melem/s and an average latency of 697 nanosecond. Form the result , seem like there are only one outlier are found among 100 iteration which depict a high degree of stability .The density plot has show a tight latency distribution. Beside , the scatter plot indicates a consistent performance trend . the presence of outlier does not significantly impact to the overall performance suggesting that the function is well-optimized and reliable for the intended operations within the Attack Command Centre System, maintaining efficiency and consistency.

Comparison System Result

```
Attacking System/Missile Reloading System
                        time:   [4.7651 µs 4.8331 µs 4.9184 µs]
                        thrpt:  [203.32 Kelem/s 206.91 Kelem/s 209.86 Kelem/s]
                 change:
                        time:   [+22.918% +39.853% +59.537%] (p = 0.00 < 0.05)
                        thrpt:  [-37.318% -28.496% -18.645%]
                        Performance has regressed.
Found 11 outliers among 100 measurements (11.00%)
  5 (5.00%) high mild
  6 (6.00%) high severe

Average Latency:
latency (ns) avg: 4_871, min: 4_100, max: 616_600
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	4.7651 µs	4.8331 µs	4.9184 µs
Throughput	203.32 Kelem/s	206.91 Kelem/s	209.86 Kelem/s
R^2	0.4837867	0.4918379	0.4792609
Mean	4.8917 µs	5.2057 µs	5.7495 µs
Std. Dev.	385.34 ns	2.4496 µs	4.1432 µs
Median	4.7570 µs	4.8346 µs	4.9038 µs
MAD	180.39 ns	237.98 ns	335.04 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Metric	Current System	Comparison System
Average Latency	**686 ns**	**4_871 ns**
Min Latency	**300 ns**	**4_100 ns**
Max Latency	**8079800 ns**	**616_600 ns**
Throughput	**1.3191 Melem/s**	**206.91 Kelem/s**
Average Time taken	**758.10 ns**	**4.8331 µs**
Outlier	**1**	**11**
Analysis	This section compare benchmark performance of both Missile Reloading System. The current system excels with significantly lower average latency (686 ns vs. 4,871 ns) and higher throughput (1.3191 Melem/s vs. 206.91 Kelem/s). It also demonstrates more stability with only one outlier compared to eleven in the comparison system. Despite a slightly higher average time taken, the current system is superior due to its overall faster response times, greater consistency, and better performance.	

Benchmark # 5 : Missile Launcher System Receive Launcher Data And Send To Radar (Current System vs Competition System)

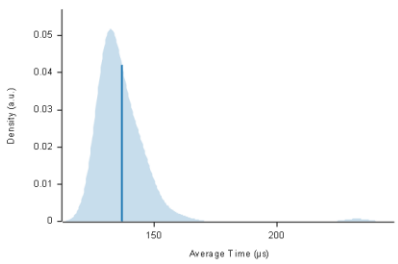
Code

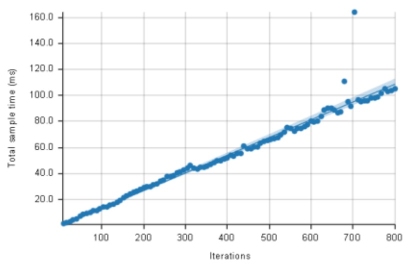
```rust
fn criterion_benchmark_missile_launcher_system(c: &mut Criterion) {
    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Attack Command Centre System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Missile Launcher System Receive Launcher Data and Send To Radar", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            missile_launcher_main();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the missile_launcher_main() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Attack Command Centre System/Missile Launcher System Receive Launcher Data and Send To Radar
                        time:   [132.61 µs 135.89 µs 141.29 µs]
                        thrpt:  [7.0777 Kelem/s 7.3588 Kelem/s 7.5408 Kelem/s]
                 change:
                        time:   [-32.904% -29.020% -25.296%] (p = 0.00 < 0.05)
                        thrpt:  [+33.861% +40.885% +49.040%]
                        Performance has improved.
Found 3 outliers among 100 measurements (3.00%)
  2 (2.00%) high mild
  1 (1.00%) high severe

Latency (ns) avg: 134895, min: 93900, max: 27163100
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	132.61 µs	**135.89 µs**	141.29 µs
Throughput	7.0777 Kelem/s	**7.3588 Kelem/s**	7.5408 Kelem/s
R^2	0.3053659	**0.3143308**	0.2911879
Mean	134.78 µs	**136.77 µs**	139.41 µs
Std. Dev.	6.0185 µs	**11.917 µs**	18.171 µs
Median	131.65 µs	**132.72 µs**	135.01 µs
MAD	2.8064 µs	**4.1254 µs**	7.2518 µs

Additional Plots:

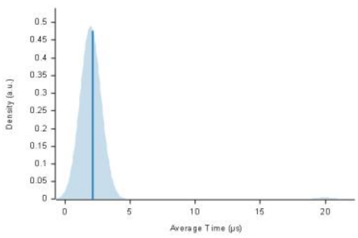
- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

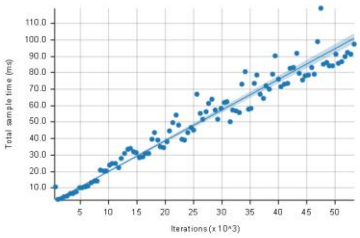
Analysis

This benchmark function has depict an average throughput of 7.3588 kelem/s and am average latency of 134895 nanosecond. The results indicate improved performance with a significant reduction in time (-29.020%) and increased throughput (+40.885%). There are 3 outlier found among 100 measurement showing a stable performance as indicated by tight latency distribution and consistent trend in the scatter plot. These findings confirm that the function is well-optimized for real-time operations.

Comparison System Result

```
Attacking System/Missile Launching System
                        time:   [1.8398 µs 1.8888 µs 1.9463 µs]
                        thrpt:  [513.80 Kelem/s 529.44 Kelem/s 543.53 Kelem/s]
                 change:
                        time:   [-12.891% +27.098% +78.841%] (p = 0.18 > 0.05)
                        thrpt:  [-44.084% -21.321% +14.799%]
                        No change in performance detected.
Found 1 outliers among 100 measurements (1.00%)
  1 (1.00%) high severe

Average Latency:
latency (ns) avg: 1_826, min: 1_300, max: 1_490_000
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	1.8398 µs	**1.8888 µs**	1.9463 µs
Throughput	513.80 Kelem/s	**529.44 Kelem/s**	543.53 Kelem/s
R^2	0.2926462	**0.3025652**	0.2890467
Mean	1.9403 µs	**2.1572 µs**	2.5484 µs
Std. Dev.	228.10 ns	**1.8127 µs**	3.0921 µs
Median	1.8238 µs	**1.8992 µs**	2.0227 µs
MAD	161.49 ns	**251.01 ns**	334.15 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Metric	Current System	Comparison System
Average Latency	**134895 ns**	**1_826 ns**
Min Latency	**93900 ns**	**1_300 ns**
Max Latency	**27163100 ns**	**1_490_000 ns**
Throughput	**7.3588 Kelem/s**	**529.44 Kelem/s**
Average Time taken	**135.89 µs**	**1.8888 µs**
Outlier	**3**	**1**
Analysis	This section compare benchmark performance of both Missile Launcher System. The comparison system has faster average time (1.8888 µs vs. 135.89 µs) and lower average latency (1,826 ns vs. 134,895 ns), indicating quicker response times. However, the current system significantly outperforms in throughput (7.3588 Kelem/s vs. 529.44 Kelem/s), making it more efficient for handling high volumes of operations despite the higher latency. Additionally, the current system has more outliers (3 vs. 1), which might suggest occasional performance inconsistencies. Overall, the comparison system is better for quick, low-latency responses, while the current system excels in throughput.	

Missile Defence System

Benchmark # 6 : Radar System Receive Threat Information (Current System vs Competition System)

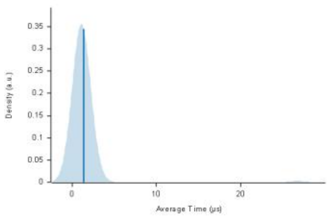
Code

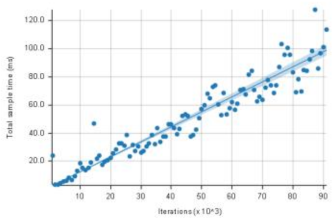
```rust
fn criterion_benchmark_radar_system(c: &mut Criterion) {
    let mut group = c.benchmark_group("Defence Command Centre System");
    let mut lb = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function("Radar System Receive Threat Information", |b| {

        b.iter(|| {
            let start_time = lb.op_start();
            radar_system_main();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the radar_system_main() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Defence Command Centre System/Radar System Receive Threat Information
                        time:   [1.0502 µs 1.0889 µs 1.1300 µs]
                        thrpt:  [884.96 Kelem/s 918.33 Kelem/s 952.19 Kelem/s]
                 change:
                        time:   [-35.216% +7.2469% +76.805%] (p = 0.80 > 0.05)
                        thrpt:  [-43.440% -6.7572% +54.360%]
                        No change in performance detected.
Found 4 outliers among 100 measurements (4.00%)
  2 (2.00%) high mild
  2 (2.00%) high severe

Latency (ns) avg: 1019, min: 600, max: 20017800
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	1.0502 µs	1.0889 µs	1.1300 µs
Throughput	884.96 Kelem/s	918.33 Kelem/s	952.19 Kelem/s
R^2	0.3191307	0.3302857	0.3177920
Mean	1.1108 µs	1.4058 µs	1.9515 µs
Std. Dev.	177.05 ns	2.5707 µs	4.3929 µs
Median	1.0576 µs	1.1165 µs	1.1525 µs
MAD	150.73 ns	197.11 ns	231.68 ns

Additional Plots:

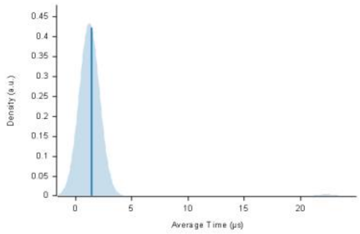
- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

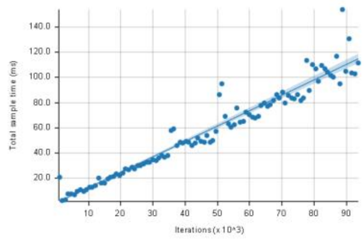
Analysis

shows an average throughput of 918.33 Kelem/s and an average latency of 1,019 nanoseconds. Despite four outliers among 100 measurements, the performance remains stable, with a tight latency distribution and a consistent trend in the scatter plot. The latency ranges from 600 nanoseconds to 20,017,000 nanoseconds, indicating occasional spikes. No significant changes in performance were detected, suggesting the function maintains its efficiency and reliability for real-time operations.

Comparison System Result

```
Defending System/Radar Detection System
                        time:   [1.1903 µs 1.2274 µs 1.2704 µs]
                        thrpt:  [787.15 Kelem/s 814.71 Kelem/s 840.16 Kelem/s]
                 change:
                        time:   [+8.9346% +57.859% +136.31%] (p = 0.01 < 0.05)
                        thrpt:  [-57.682% -36.652% -8.2018%]
                        Performance has regressed.
Found 12 outliers among 100 measurements (12.00%)
  6 (6.00%) high mild
  6 (6.00%) high severe

Average Latency:
latency (ns) avg: 1_082, min: 800, max: 560_200
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	1.1903 µs	**1.2274 µs**	1.2704 µs
Throughput	787.15 Kelem/s	**814.71 Kelem/s**	840.16 Kelem/s
R^2	0.2702327	**0.2811409**	0.2667476
Mean	1.2089 µs	**1.4448 µs**	1.8870 µs
Std. Dev.	135.14 ns	**2.1111 µs**	3.6135 µs
Median	1.1634 µs	**1.1805 µs**	1.2050 µs
MAD	62.237 ns	**83.058 ns**	106.69 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Metric	Current System	Comparison System
Average Latency	**1019 ns**	**1_082 ns**
Min Latency	**600 ns**	**800 ns**
Max Latency	**21007800 ns**	**560_200 ns**
Throughput	**918.33 Kelem/s**	**814.71 Kelem/s**
Average Time taken	**1.0889 µs**	**1.2274 µs**
Outlier	**4**	**12**
Analysis	This section compare benchmark performance of both radar system. The current system has a shorter average time (1.0889 µs vs. 1.2274 µs) and higher throughput (918.33 Kelem/s vs. 814.71 Kelem/s), indicating better performance in processing tasks quickly and efficiently. However, it also has more outliers (4 vs. 12) and higher maximum latency (2,100,7800 ns vs. 560,200 ns). Overall, the current system is better due to its higher throughput and faster average time.	

Benchmark # 7 : Défense Command Centre Receive Incoming Missile Data (Current System vs Competition System)

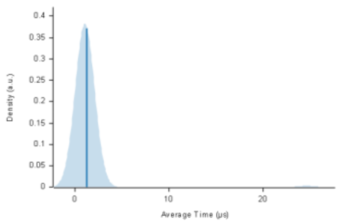
Code

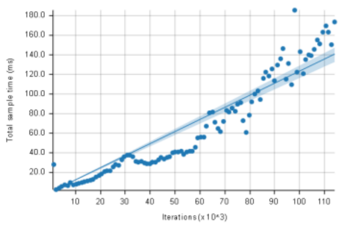
```
fn criterion_benchmark_dfd_command_centre_system(c: &mut Criterion) {
    let mut group = c.benchmark_group("Defence Command Centre System");
    let mut lb = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function("Defense Command Centre Receive Incomming Missile Data", |b| {

        b.iter(|| {
            let start_time = lb.op_start();
            dfd_command_centre_main();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the dfd_command_centre_main() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Defence Command Centre System/Defense Command Centre Receive Incomming Missile Data
                        time:   [1.1670 µs 1.2319 µs 1.2884 µs]
                        thrpt:  [776.16 Kelem/s 811.73 Kelem/s 856.92 Kelem/s]
                 change:
                        time:   [-36.593% +5.0432% +72.529%] (p = 0.81 > 0.05)
                        thrpt:  [-42.039% -4.8011% +57.711%]
                        No change in performance detected.
Found 1 outliers among 100 measurements (1.00%)
  1 (1.00%) high severe

Latency (ns) avg: 962, min: 600, max: 2962600
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	1.1670 µs	**1.2319 µs**	1.2884 µs
Throughput	776.16 Kelem/s	**811.73 Kelem/s**	856.92 Kelem/s
R^2	0.2663035	**0.2784043**	0.2691638
Mean	1.0180 µs	**1.2921 µs**	1.7978 µs
Std. Dev.	241.84 ns	**2.3717 µs**	4.0548 µs
Median	923.42 ns	**1.0574 µs**	1.1565 µs
MAD	262.67 ns	**373.69 ns**	413.33 ns

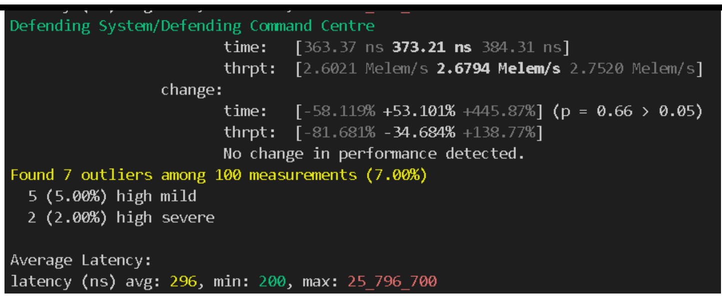
Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

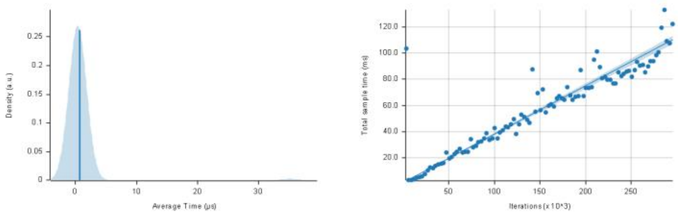
Analysis

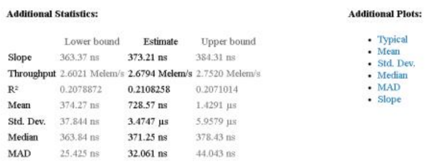
The benchmark for Defense Command Centre Receive Incoming Missile Data shows an average throughput of 811.73 Kelem/s and an average latency of 962 nanoseconds. Despite only one outlier has found among 100 measurements, the function demonstrates stable performance. The latency distribution is tight, with minimal deviation, and the scatter plot indicates consistent performance trends. There is no significant performance changes were detected, confirming the function's

<table>
<tr><td></td><td>reliability and efficiency in processing incoming missile data for real-time operations.</td></tr>
<tr><td>Comparison System Result</td><td>



```
Defending System/Defending Command Centre
                        time:   [363.37 ns 373.21 ns 384.31 ns]
                        thrpt:  [2.6021 Melem/s 2.6794 Melem/s 2.7520 Melem/s]
                 change:
                        time:   [-58.119% +53.101% +445.87%] (p = 0.66 > 0.05)
                        thrpt:  [-81.681% -34.684% +138.77%]
                        No change in performance detected.
Found 7 outliers among 100 measurements (7.00%)
  5 (5.00%) high mild
  2 (2.00%) high severe

Average Latency:
latency (ns) avg: 296, min: 200, max: 25_796_700
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	363.37 ns	373.21 ns	384.31 ns
Throughput	2.6021 Melem/s	2.6794 Melem/s	2.7520 Melem/s
R^2	0.2078872	0.2108258	0.2071014
Mean	374.27 ns	728.57 ns	1.4291 µs
Std. Dev.	37.844 ns	3.4747 µs	5.9579 µs
Median	363.84 ns	371.25 ns	378.43 ns
MAD	25.425 ns	32.061 ns	44.043 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

</td></tr>
</table>

Metric	Current System	Comparison System
Average Latency	**962 ns**	**296 ns**
Min Latency	**200 ns**	**200 ns**
Max Latency	**2962600 ns**	**25_796_700 ns**
Throughput	**811.73 Kelem/s**	**2.6794 Melem/s**
Average Time taken	**1.2319 µs**	**373.21 ns**
Outlier	**4**	**7**
Analysis	This section compare benchmark performance of both Missile Defending system. The comparison system excels with lower average latency (296 ns vs. 962 ns) and significantly higher throughput (2.6794 Melem/s vs. 811.73 Kelem/s) It also has a shorter average time taken (373.21 ns vs. 1.2319 µs), indicating faster performance. However, it has more outliers (7 vs. 4) and a higher maximum latency (25,796,700 ns vs. 2,926,600 ns). Overall, the comparison system is better due to its superior latency and throughput, despite more outliers.	

Benchmark # 8 : Interceptor System Receive Threat Information (Current System vs Competition System)

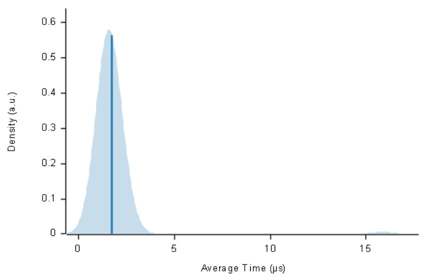
Code

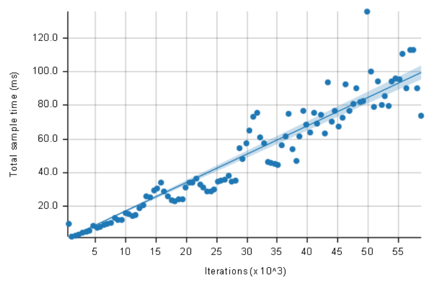
```rust
fn criterion_benchmark_interceptor_system(c: &mut Criterion) {

    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Defence Command Centre System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Interceptor System Receive and Intercept Incomming Missile Data", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            interceptor_system_main();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the interceptor_system_main() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Defence Command Centre System/Interceptor System Receive and Intercept Incomming Missile Data
                        time:   [1.6277 µs 1.6995 µs 1.7747 µs]
                        thrpt:  [563.46 Kelem/s 588.39 Kelem/s 614.37 Kelem/s]
                 change:
                        time:   [-59.953% -52.149% -41.608%] (p = 0.00 < 0.05)
                        thrpt:  [+71.258% +108.98% +149.70%]
                        Performance has improved.
Found 2 outliers among 100 measurements (2.00%)
  1 (1.00%) high mild
  1 (1.00%) high severe

Latency (ns) avg: 1609, min: 1000, max: 1377700
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	1.6277 µs	1.6995 µs	1.7747 µs
Throughput	563.46 Kelem/s	588.39 Kelem/s	614.37 Kelem/s
R²	0.0537817	0.0568580	0.0535068
Mean	1.5583 µs	1.7451 µs	2.0678 µs
Std. Dev.	262.45 ns	1.4578 µs	2.4689 µs
Median	1.4963 µs	1.5794 µs	1.6450 µs
MAD	258.77 ns	345.08 ns	419.86 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

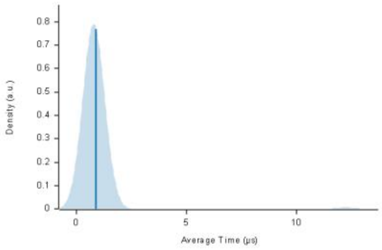
Analysis

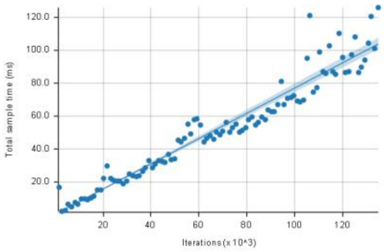
The benchmark shows an average throughput of 588.39 Kelem/s and an average latency of 1,609 nanoseconds. With two outliers among 100 measurements, performance has significantly improved, as indicated by a 52.149% reduction in time and a 108.98% increase in throughput. The latency distribution and scatter plot indicate consistent and reliable performance, despite occasional spikes. These improvements confirm

	the system's enhanced efficiency in real-time missile interception operations.

Comparison System Result

```
Defending System/Interceptor System
                        time:   [743.47 ns 768.80 ns 795.81 ns]
                        thrpt:  [1.2566 Melem/s 1.3007 Melem/s 1.3450 Melem/s]
                 change:
                        time:   [-31.435% +19.534% +97.853%] (p = 0.65 > 0.05)
                        thrpt:  [-49.457% -16.342% +45.846%]
                        No change in performance detected.
Found 5 outliers among 100 measurements (5.00%)
  3 (3.00%) high mild
  2 (2.00%) high severe

Average Latency:
latency (ns) avg: 695, min: 500, max: 927_800
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	743.47 ns	768.80 ns	795.81 ns
Throughput	1.2566 Melem/s	1.3007 Melem/s	1.3450 Melem/s
R^2	0.4349343	0.4494022	0.4330225
Mean	758.02 ns	890.34 ns	1.1352 µs
Std. Dev.	98.501 ns	1.1515 µs	1.9686 µs
Median	713.31 ns	727.34 ns	755.08 ns
MAD	52.021 ns	71.413 ns	104.05 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Metric	Current System	Comparison System
Average Latency	**1609 ns**	**695 ns**
Min Latency	**1000 ns**	**500 ns**
Max Latency	**137600 ns**	**927_800 ns**
Throughput	**500.39 Kelem/s**	**1.3007 Melem/s**
Average Time taken	**1.6995 µs**	**768.80 ns**
Outlier	**2**	**5**
Analysis	This section compare benchmark performance of both Intercepting system. The comparison system outperforms the current system with lower average latency (695 ns vs. 1,609 ns), higher throughput (1.3007 Melem/s vs. 500.39 Kelem/s), and shorter average time taken (768.80 ns vs. 1.6995 µs). Despite having more outliers (5 vs. 2) and higher maximum latency, the comparison system is better due to its superior efficiency and faster performance.	

Benchmark # 9 : Defence Command Centre The Missile Updated Status and Stope Countdown. (Current System vs Competition System)

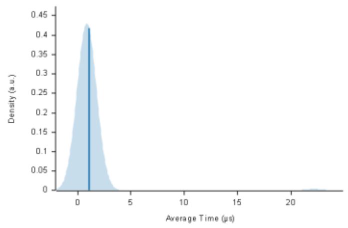
Code

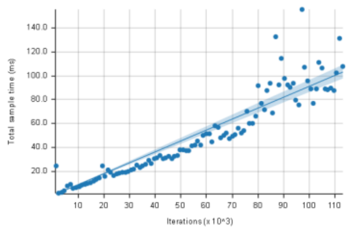
```
fn criterion_benchmark_dfd_command_centre_updated(c: &mut Criterion) {

    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Defence Command Centre System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Defense Command Centre Receive The Missile Updated Status and Stop Countdown", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            count_down_missile_arrived_main();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the count_down_missile_arrived_main() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Defence Command Centre System/Defense Command Centre Receive The Missile Updated Status and Stop Cou...
                        time:   [865.62 ns 914.45 ns 966.43 ns]
                        thrpt:  [1.0347 Melem/s 1.0935 Melem/s 1.1552 Melem/s]
                 change:
                        time:   [-45.161% -9.0471% +55.954%] (p = 0.79 > 0.05)
                        thrpt:  [-35.878% +9.9470% +82.353%]
                        No change in performance detected.
Found 7 outliers among 100 measurements (7.00%)
  4 (4.00%) high mild
  3 (3.00%) high severe

Latency (ns) avg: 813, min: 500, max: 15076200
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	865.62 ns	914.45 ns	966.43 ns
Throughput	1.0347 Melem/s	1.0935 Melem/s	1.1552 Melem/s
R^2	0.1552446	0.1633529	0.1542296
Mean	826.84 ns	1.0663 µs	1.5147 µs
Std. Dev.	150.57 ns	2.1373 µs	3.6579 µs
Median	765.02 ns	778.20 ns	822.65 ns
MAD	73.158 ns	94.059 ns	142.28 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Analysis

The benchmark shows an average throughput of 1.0935 Melem/s and an average latency of 813 nanoseconds. Even though seven outliers has found among 100 measurements, but performance still remains stable with no significant changes detected. The latency distribution is tight, and the scatter plot shows consistent performance trends. These results indicate the system's reliability and efficiency in real-time missile status updates, with occasional outliers not significantly impacting overall performance

Comparison System Result

NA in comparison system

Benchmark # 10: Overall Attack System (Current System vs Competition System)

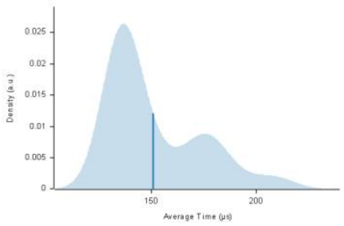
Code

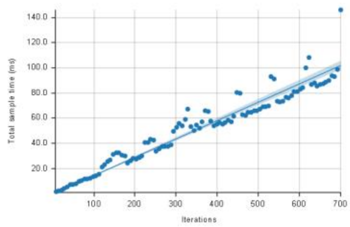
```rust
fn criterion_overall_attack_system(c: &mut Criterion) {

    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Overall Attack System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Overall Attack System", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            attack_system();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the attack_system() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Overall Attack System/Overall Attack System
                        time:   [141.13 µs 145.62 µs 151.31 µs]
                        thrpt:  [6.6091 Kelem/s 6.8673 Kelem/s 7.0858 Kelem/s]
                 change:
                        time:   [-2.8825% +0.0014% +3.0588%] (p = 1.00 > 0.05)
                        thrpt:  [-2.9680% -0.0014% +2.9681%]
                        No change in performance detected.

Latency (ns) avg: 153421, min: 101200, max: 10274000
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	141.13 µs	145.62 µs	151.31 µs
Throughput	6.6091 Kelem/s	6.8673 Kelem/s	7.0858 Kelem/s
R²	0.5681223	0.5806816	0.5607659
Mean	146.74 µs	150.86 µs	155.26 µs
Std. Dev.	18.402 µs	21.904 µs	24.857 µs
Median	137.01 µs	138.64 µs	142.96 µs
MAD	5.5504 µs	8.1453 µs	14.766 µs

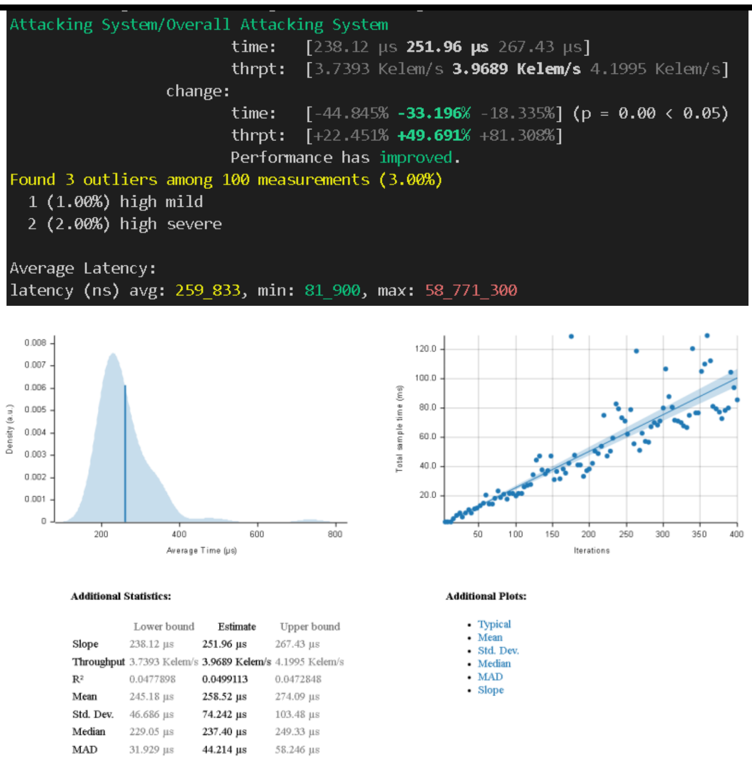
Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Analysis

The benchmark for the Overall Attack System, which combines all previously analysed attack functions, shows an average throughput of 6.8673 Kelem/s and an average latency of 153,421 nanoseconds. The results indicate no significant performance changes, with a minimal impact from outliers. The latency distribution is relatively wide, suggesting variability in execution times, yet the scatter plot shows a consistent performance trend. These findings confirm the combined attack functions' reliability and efficiency, maintaining consistent execution times and throughput, essential for real-time operations in the overall attack system.

Comparison System Result



Metric	Current System	Comparison System
Average Latency	**153421 ns**	**259_833 ns**
Min Latency	**101200 ns**	**81_900 ns**
Max Latency	**10274000 ns**	**58_771_300 ns**
Throughput	**6.8673 Kelem/s**	**3.9689 Kelem/s**
Average Time taken	**145.62 µs**	**251.96 µs**
Outlier	**NA**	**3**
Analysis	This section compare benchmark performance of both overall attacking system. The current system outperforms with higher throughput (6.8673 Kelem/s vs. 3.9689 Kelem/s) and shorter average time taken (145.62 µs vs. 251.96 µs). Beside , the current system also have the lower average latency (153421 vs 259833) which mean to higher efficiency and faster performance . In summary , the current overall attacking system and perform better performance than the comparison system due to the lower average time taken (145.62 **µs** vs 251.96 **µs**)	

Benchmark # 11: Overall Defence System (Current System vs Competition System)

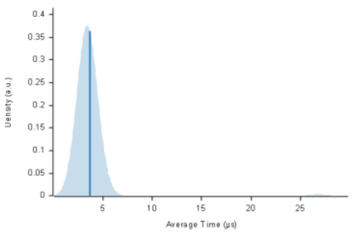
Code

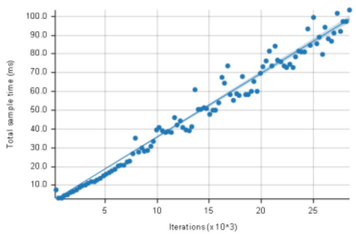
```rust
fn criterion_overall_defence_system(c: &mut Criterion) {

    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Overall Defence System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Overall Defence System", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            defence_system();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the defence_system() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Overall Defence System/Overall Defence System
                        time:   [3.4069 µs 3.4688 µs 3.5339 µs]
                        thrpt:  [282.98 Kelem/s 288.29 Kelem/s 293.52 Kelem/s]
                 change:
                        time:   [-18.687% -5.5036% +11.925%] (p = 0.57 > 0.05)
                        thrpt:  [-10.654% +5.8241% +22.981%]
                        No change in performance detected.
Found 7 outliers among 100 measurements (7.00%)
  5 (5.00%) high mild
  2 (2.00%) high severe

Latency (ns) avg: 3409, min: 2700, max: 997100
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	3.4069 µs	**3.4688 µs**	3.5339 µs
Throughput	282.98 Kelem/s	**288.29 Kelem/s**	293.52 Kelem/s
R^2	0.5613294	**0.5730675**	0.5600789
Mean	3.4001 µs	**3.6856 µs**	4.2002 µs
Std. Dev.	291.21 ns	**2.3594 µs**	4.0193 µs
Median	3.3354 µs	**3.3961 µs**	3.4555 µs
MAD	200.56 ns	**268.91 ns**	364.52 ns

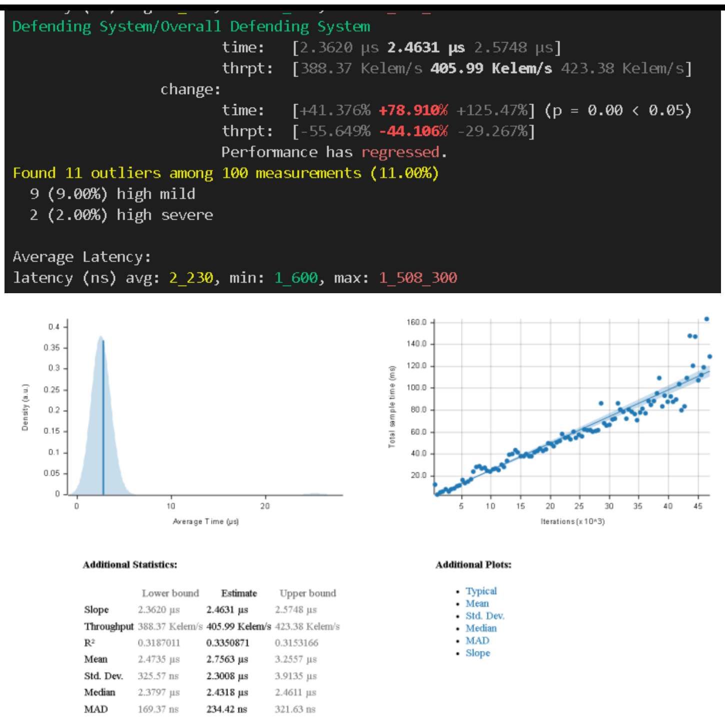
Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Analysis

The benchmark for the Overall Defence System, which combines all defence functions, shows an average throughput of 288.29 Kelem/s and an average latency of 3,409 nanoseconds. Since this is a combined function of all defence function, hence there are seven outliers among 100 measurements, the system maintains stable performance with no significant changes detected. The latency distribution is tight, and the scatter plot indicates consistent performance. These results confirm the overall defence functions' reliability and efficiency in real-time operations, with occasional outliers not significantly impacting overall performance.

Comparison System Result



```
Defending System/Overall Defending System
                        time:   [2.3620 µs 2.4631 µs 2.5748 µs]
                        thrpt:  [388.37 Kelem/s 405.99 Kelem/s 423.38 Kelem/s]
                 change:
                        time:   [+41.376% +78.910% +125.47%] (p = 0.00 < 0.05)
                        thrpt:  [-55.649% -44.106% -29.267%]
                        Performance has regressed.
Found 11 outliers among 100 measurements (11.00%)
  9 (9.00%) high mild
  2 (2.00%) high severe

Average Latency:
latency (ns) avg: 2_230, min: 1_600, max: 1_508_300
```


Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	2.3620 µs	2.4631 µs	2.5748 µs
Throughput	388.37 Kelem/s	405.99 Kelem/s	423.38 Kelem/s
R^2	0.3187011	0.3350871	0.3153166
Mean	2.4735 µs	2.7563 µs	3.2557 µs
Std. Dev.	325.57 ns	2.3008 µs	3.9135 µs
Median	2.3797 µs	2.4318 µs	2.4611 µs
MAD	169.37 ns	234.42 ns	321.63 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Metric	Current System	Comparison System
Average Latency	**3409 ns**	**2_230 ns**
Min Latency	**2700 ns**	**1_600 ns**
Max Latency	**997200 ns**	**1_508_300 ns**
Throughput	**288.29 Kelem/s**	**405.99 Kelem/s**
Average Time taken	**3.4688 µs**	**2.4631 µs**
Outlier	**7**	**11**
Analysis	This section compare benchmark performance of both overall defending system. The comparison system outperforms the current system with lower average latency (2,230 ns vs. 3,409 ns), higher throughput (405.99 Kelem/s vs. 288.29 Kelem/s), and shorter average time taken (2.4631 µs vs. 3.4688 µs). Despite having more outliers (11 vs. 7) and higher maximum latency (1,508,300 ns vs. 997,200 ns), the comparison system's superior efficiency and faster performance make it better overall. The current system shows fewer outliers and better maximum latency, but the comparison system's advantages in latency, throughput, and average time taken outweigh these factors.	

Benchmark # 12: Overall System (Current System vs Competition System)

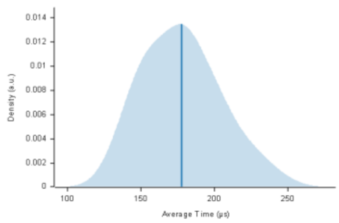
Code

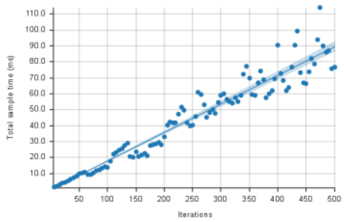
```rust
fn criterion_overall_system(c: &mut Criterion) {

    let mut group: BenchmarkGroup<WallTime> = c.benchmark_group(group_name: "Overall System");
    let mut lb: LatencyBenchmark = LatencyBenchmark::new();
    group.throughput(Throughput::Elements(1));
    group.bench_function(id: "Overall System", f: |b: &mut Bencher<WallTime>| {

        b.iter(routine: || {
            let start_time: Instant = lb.op_start();
            overall_system();
            lb.op_finish(start_time);
        });
    });
    group.finish();
    lb.print();
}
```

This function benchmarks the overall_system() function by measuring throughput with Criterion and latency with a custom Latency Benchmark, collecting start and end times, and printing the results.

Result

```
Overall System/Overall System
                        time:   [172.82 µs 179.11 µs 185.90 µs]
                        thrpt:  [5.3793 Kelem/s 5.5832 Kelem/s 5.7863 Kelem/s]
                 change:
                        time:   [-2.2394% +2.2843% +6.8870%] (p = 0.32 > 0.05)
                        thrpt:  [-6.4433% -2.2332% +2.2906%]
                        No change in performance detected.

Latency (ns) avg: 191264, min: 106500, max: 76824900
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	172.82 µs	**179.11 µs**	185.90 µs
Throughput	5.3793 Kelem/s	**5.5832 Kelem/s**	5.7863 Kelem/s
R^2	0.1541277	**0.1623949**	0.1528403
Mean	172.36 µs	**177.49 µs**	182.71 µs
Std. Dev.	23.014 µs	**26.592 µs**	29.710 µs
Median	168.63 µs	**177.47 µs**	182.71 µs
MAD	21.601 µs	**29.712 µs**	34.514 µs

Additional Plots:

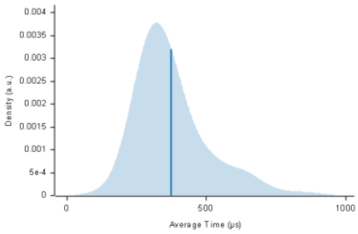
- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

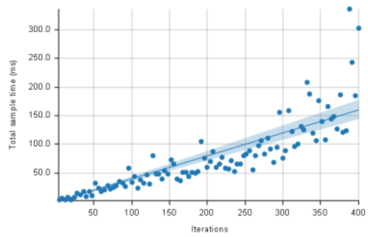
Analysis

The benchmark for the Overall System, which combines all functions from attack to defence until the missile interception, shows an average throughput of 5.5832 Kelem/s and an average latency of 191,264 nanoseconds. Despite minor performance changes, the results indicate stable execution times with consistent throughput. The latency distribution is relatively tight, and the scatter plot demonstrates a consistent performance trend. These findings confirm the combined system's reliability and efficiency in maintaining consistent execution times and throughput, essential for real-time operations across the entire defence workflow.

Comparison System Result

```
Overall Simulation/Overall Simulation
                        time:   [361.05 µs 400.67 µs 444.34 µs]
                        thrpt:  [2.2505 Kelem/s 2.4958 Kelem/s 2.7697 Kelem/s]
                 change:
                        time:   [+401.30% +436.93% +475.03%] (p = 0.00 < 0.05)
                        thrpt:  [-82.610% -81.376% -80.052%]
                        Performance has regressed.
Found 9 outliers among 100 measurements (9.00%)
  8 (8.00%) high mild
  1 (1.00%) high severe

Average Latency:
latency (ns) avg: 324_080, min: 81_800, max: 55_953_900
```





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	361.05 µs	**400.67 µs**	444.34 µs
Throughput	2.2505 Kelem/s	**2.4958 Kelem/s**	2.7697 Kelem/s
R^2	0.3010616	**0.3170257**	0.2978423
Mean	350.77 µs	**374.81 µs**	400.82 µs
Std. Dev.	103.23 µs	**129.07 µs**	152.89 µs
Median	323.66 µs	**341.85 µs**	368.41 µs
MAD	69.766 µs	**87.890 µs**	115.54 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

Metric	Current System	Comparison System
Average Latency	**191264 ns**	**324_080 ns**
Min Latency	**106500 ns**	**81_800 ns**
Max Latency	**76824900 ns**	**55_953_900 ns**
Throughput	**5.5831 Kelem/s**	**2.4958 Kelem/s**
Average Time taken	**179.11 µs**	**400.67 µs**
Outlier	**NA**	**9**
Analysis	This section compare benchmark performance of both overall system. The current system outperforms the comparison system with significantly higher throughput (5.5831 Kelem/s vs. 2.4958 Kelem/s) and shorter average time taken (179.11 µs vs. 400.67 µs), indicating more efficient processing. Beside, the comparison system has higher average latency (324,080 ns vs. 191,264 ns) and minimum latency (81,800 ns vs. 106,500 ns), suggesting quicker response times. Despite these advantages, the comparison system has more outliers (9) and much higher maximum latency (55,953,900 ns vs. 7,682,4900 ns). Overall, the current system is better due to its higher	

throughput and faster average processing time, crucial for handling high volumes of operations efficiently.

Summary of Finding

System Name	Reliability		Efficiency		Consistency	
	Current	**Comparison**	**Current**	**Comparison**	**Current**	**Comparison**
Weather System	✓		✓		✓	
Target Navigator System	✓			✓		✓
Attack Command Centre	✓		✓		✓	
Missile Reloading System	✓		✓			✓
Missile Launcher System		✓		✓	✓	
Radar System	✓		✓			✓
Défense Command Centre	✓			✓		✓
Interceptor System	✓			✓		✓
Overall Attacking System	✓		✓		✓	
Overall Defending System	✓			✓	✓	
Overall System	✓		✓		✓	

5.0 Conclusion

The development of a real-time missile attack and defence simulation system in this project marks a significant step forward in country security operations. Using Rust's rich concurrency, memory safety, and performance features, this project successfully handles the problems of timely job execution and system predictability. The combination of dynamic and interactive simulations enables thorough evaluation of numerous defence situations, resulting in optimized response strategies and increased awareness. The extensive benchmarking undertaken confirms the system's robustness and efficiency, indicating its suitability for real-time operations. Finally, this simulation system serves as a critical tool for defence forces, allowing them to respond quickly and effectively to missile threats, considerably improving the overall country security.

6.0 References

Boyd, I. (18 OCT, 2022). *A game of numbers: How air defense systems work and why Ukraine is eager for more protection*. Retrieved from https://theconversation.com/: https://theconversation.com/a-game-of-numbers-how-air-defense-systems-work-and-why-ukraine-is-eager-for-more-protection-192487

Bugden, W., Alahmar, A., & Author. (2022). *Rust: The Programming Language for Safety and Performance.*

Gupta, P., Rahar, R., Yadav, R. K., Singh, A., Ramandeep, & Kumar, S. (2023). *Combining Forth and Rust: A Robust and Efficient Approach for Low-Level System Programming †.*

Klein-Winternheim. (21 Feb, 2023). *Rust now available for Real-Time Operating System and Hypervisor PikeOS*. Retrieved from www.sysgo.com: https://www.sysgo.com/press-releases/rust-now-available-for-real-time-operating-system-and-hypervisor-pikeos

Klein-Winternheim. (21 FEB, 2023). *Rust now available for Real-Time Operating System and Hypervisor PikeOS*. Retrieved from www.sysgo.com: https://www.sysgo.com/press-releases/rust-now-available-for-real-time-operating-system-and-hypervisor-pikeos

Rust. (2024). *Rust A language empowering everyone*. Retrieved from rust-lang.org: https://www.rust-lang.org/

Sharma, A., Sharma, S., Torres-Arias, S., & Machiry, A. (11, 2023). *Rust for Embedded Systems: Current State, Challenges and Open Problems*. Retrieved from http://arxiv.org/abs/2311.05063