Automatic Memory Management in ORB SLAM-3

by

Shreyas Athreya Venkatesh

12th January 2024

A thesis submitted to the
faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science
Department of Computer Science and Engineering

Acknowledgements

I would like thank my advisor and mentor, Dr. Lukasz Ziarek, who provided me with the opportunity to pursue a thesis and guiding me throughout this research journey. His inclusivity not only instilled confidence in me to develop research ideas but also motivated me to become a better individual. I would also like to thank Dr. Kathik Dantu for the suggestions for improving this thesis.

I would also like to thank Mr. Nitin Vinod, my colleague, for the constant brainstorming sessions during the past year and without his support, this thesis would not be possible.

Finally, I would like to thank my friends and family for all their support.

Abstract

Advances in visual simultaneous localization and mapping (VSLAM) systems profoundly impact mobile robotics, augmented reality, and virtual reality domains [1] by enabling them to create accurate map representations of their surroundings and precisely locate themselves in it. VSLAM applications are frequently deployed in an embedded system where they often encounter operational challenges such as insufficient memory, restrictions on processing power due to resource constraints and adherence to stringent timing requirements etc. The limitations imposed by finite memory and the unbounded growth of dynamic memory represent critical issues that could restrict the scalability of SLAM systems in large-scale environment [2] that involve longer operational times and increased map complexity. This thesis addresses this issue in ORB-SLAM-3 by presenting an automatic memory management strategy using the reference counting scheme. The testing and profiling on the EuRoC dataset demonstrate the viability of such a scheme with minimal execution overhead. Additionally, an alternative keyframe redundancy marking scheme is presented in this thesis with a time complexity analysis.

TABLE OF CONTENTS

List of Tables

List of Figures

1. Introduction

Over the last decade, there has been a significant surge in the advancement of autonomous mobile robotics, augmented reality, and virtual reality. This is primarily credited to the accessibility of affordable hardware. Simultaneous Localization and Mapping (SLAM) algorithms are ubiquitous among the autonomous mobile robot and AR/VR domain [3]. SLAM algorithm addresses the challenge of generating a map in an unknown environment using sensors like Lidars, sonars, or cameras while simultaneously determining the position/location of the host, typically a mobile robot or a virtual reality headset/controller, within the generated map. Visual SLAM systems, which rely on cameras as their primary input source, remain a popular choice for SLAM researchers because of their simple sensor configuration [3], resulting in increased accuracy and robustness.

Despite these notable advancements in SLAM algorithms, more progress is needed, to improve the SLAM system design focusing on efficient memory storage, computational efficiency, and safety. Moreover, these systems are usually operated under strict timing and resource constraints setting [1]. From a memory management perspective, real-world and large-scale implementation of SLAM, such as VineSLAM, LeGO-LOAM, and LOAM [4], often entail extended hours of operations, resulting in larger memory requirement due to the possibility of an unbounded growth of the generated map. Additionally, SLAM researchers have been exploring multimodal SLAM setups like MIMOSA [5] for increased accuracy and robustness under visual constraints, adding an extra memory overhead. Consequently, managing memory in SLAM systems remains an important open problem. This thesis primarily focuses on leveraging idea from programming languages such as reference counting to manage memory in ORB-SLAM-3 by reducing the resident memory footprint.

1.1 Thesis Contributions:

This thesis addresses the problem of memory management on ORB-SLAM-3 by providing a framework for automatic garbage collection of dynamic memory objects such as keyframes and map points. We utilize a reference counting strategy for implementing safe deletion. Furthermore, we present an optimization over the mutex-based reference counting using a compare and swap-based strategy. Additionally, we benchmark the performance of both references counted approaches using the profiling tool Tracy. Contributions comprising of, implementation of reference counting, compare and swap and profiling and benchmarking of the system, i.e section 4 represent joint work with Mr. Nitin Vinod. The thesis further delves into two experiments. The first hypothesis states that a relation exists between the deletion of a heap object and its number of references, thereby providing a mechanism to infer similar object lifetimes. The second hypothesis examines the viability of replacing a redundancy in the Local Mapping thread.

1.2 Thesis Structure

Section 2 overviews the existing SLAM system design, identifies the similarities among popular visual SLAM systems, and briefly describes memory management strategies, particularly in real-time systems, followed by background on reference counting. Section 3 gives a detailed description of the ORB SLAM architecture. Section 4 (joint work with Mr. Nitin Vinod) presents our garbage collection experiments and a reference counting-based memory management strategy for ORB-SLAM-3. We offer an optimization over the reference counting-based solution with the compare and swap technique. We conclude this section by discussing the performance profile for reference counting and comparing and swapping garbage collection. Section 5 explores an alternative framework for marking keyframes as bad(garbage) in the local mapping thread. Section 6 summarizes the thesis and any future work.

2. Background and Related Work

The challenge of meeting the stringent timeliness requirements [1] arises due to the real-time nature of the ORB SLAM 3 system. We begin this section by introducing real-time systems and design considerations for real-time systems, then understand SLAM systems, particularly graph SLAM systems followed by a typical Visual SLAM system. Subsequent sections explain memory management, types of memory management strategies, C++ memory model and some C++ features such as smart pointers as they are used in thesis in later sections.

2.1 Real-Time Systems

Systems that rely on the completion of computations and logical soundness are known as real-time computing systems [10]. These systems could have a function or a collection of functions that must adhere to strict deadlines to operate correctly. Real-time systems are required to control the system's behavior resulting from an external stimulus within strict temporal deadlines. In the case of a SLAM system, this involves mapping (creating a map of an environment) and localization (calculating the pose (position and orientation) of a robot within a map) using camera images. The timing requirements for processes/functions in designing SLAM systems are categorized as soft and hard. Soft requirements are the processes in a real-time system that can be performed even after the deadline has passed. On the other hand, processes that cannot miss a deadline are called hard requirements. Subsequently, real-time systems with hard requirements are called **hard real-time systems**, and those without hard requirements are termed **soft real-time systems**.

According to Kopetz & Steiner et. al [9], a well-designed real-time system must satisfy functional and temporal requirements. We briefly go over each of these requirements below.

2.1.1 Functional requirement:

Functional requirements are the tasks/functions a real-time system must perform every cycle. The author [9] categorizes them into,

- **Data collection requirement**

  Real-time systems are required to complete the task of data collection and the signal conditioning algorithm to ensure that the system behavior is appropriate even in rare transient overloads.

- **Direct digital control requirement**

  Real-time systems require the execution of the control algorithms to complete in time to provide the actuating variables, such as the speed of a vehicle or the actuating speed of a motor.

- **Man-machine interaction requirement**

  In typical safety-critical applications, the real-time system needs to relay the system's state through data logging.

2.1.2 Temporal requirements:

Temporal requirements in a real-time system result from control loops of an actual time process, such as behavior control of an automotive engine that requires stringent temporal demands. These requirements are termed as explicit requirements [10]. Real-time systems designed for human interaction have less stringent temporal demands as the human perception delay is estimated to be 50-100ms [9]. These requirements are termed implicit requirements and correlate to soft deadlines. A system designer should also account for the following: context-switching overhead, dealing with limited priority levels, and jitter, which is the variation in completing a periodic task while designing a real-time system.

2.1.3 Real-Time Challenges in SLAM System

For a SLAM system to be functional, it is essential for its localization and map-building modules to work in real-time as, a robot's essential functions such navigation, and interaction with surrounding environment directly depends on its latency. Below we discuss various components of SLAM systems as real time systems.

- Sensor Data Processing:
    - Latency Concerns and Timing Concerns: The data acquisition process in SLAM systems using the information from sensors such as cameras, LiDARs, IMU sensors etc. needs to be processed in real-time. Delays in processing of the data acquisition can potentially cause outdated maps and inaccurate localization. Hence timing of sensor data processing needs to meet the real-time constraints.
- Map Building and Optimization Updates:
    - Latency Concerns and Timing Concerns: The SLAM system after processing the sensory information creates an intermediate representation of the sensory information in the form of a map. This map needs to be real-time as it effects the localization task and effects accuracy.
- Actuation Control and decision making:
    - Latency Concerns and Timing Concerns: The localization information that the SLAM system provides, is used for the decision making of the trajectory and the actuation system. The inability of the SLAM pipeline to meet the timeliness constraints effects the robot's ability to respond to the environment quickly.

Hence latency and timing challenges faced by SLAM systems need to be addressed for a robot to be successful in its operation.

2.2 Understanding SLAM and Visual SLAM systems

The SLAM problem attempts to solve two separate tasks, localization, and mapping.

Localization problem can be categorized into three separate problems,

- **Position Tracking (local localization):**

  Given a robot's initial pose, the algorithm would keep estimating the robots position within the map using the sensor information [50]. In this problem the uncertainty related to the robot is limited to the region surrounding the robot.

- **Global Localization:**

  Global localization task is to estimate the position of the robot in the map without knowing the initial coordinates of the robot [50]. The amount of uncertainty in this problem is much greater than position tracking and thus global localization is a much more challenging.

- **Kidnapped Robot Problem:**

  This Localization is the same as global localization where it finds its position on the map with an exception that the robot could randomly move from its current location and switched to a new location and the robot should recover its location [51].

In general, localization is the challenge of estimating the robot's pose (position and orientation) in a (given) mapped environment [52]. Possible solutions to the localization problem are probabilistic methods such as monte Carlo localization (MCL), extended Kalman filter, and machine learning techniques such as Convolutional neural network (CNN) used in conjunction with MCL for estimating the robots pose [52].

Mapping problem involves the process of producing a map of the environment, provided we are given the robots pose and has access to the sensor information and movement of the robot. The mapping task is challenging as it uses finite state variables of the robot such as robots' position

and sensor information to create a map which lies in the continuous space. As a result, there are infinitely many variables used to describe the map.

In **Simultaneous Localization and Mapping (SLAM)**, the robot performs two separate tasks simultaneously, i) localization which means a robot, given its map tries to locate itself in it, ii) mapping is the task of generating a map of its surroundings using the onboard sensors. In real-world scenarios we do not have the robots poses nor do we have the map and hence we use SLAM algorithms to solve this problem. The uncertainty in the map and the robot's pose due to noise in the robot's motion and sensor readings causes a correlation between the errors in the robot's pose and the map [7]. Applications of SLAM algorithms include robot vacuum cleaner such as Roomba, where the environment of the vacuum keeps changing and the robot equipped with its sensors maps the room and the robots pose. Other application are the Self-driving vehicles with its environment being on the roads, underground mines, aerial surveillance, or mars rovers where we do not have the position of the robot nor do we have the map of the environment.

SLAM researchers divide the SLAM problem [6] into the **front-end** processes: Transforming the raw sensor data into intermediate representation such as probability distribution of a landmark (distinctive identifiable points in an image) based on the sensor information and **backend** processes: Using the intermediate representation to perform the optimization and state estimation problem. Based on the mathematical model, filtering approaches and optimization techniques SLAM algorithms are classified into five categories [7],

- EKF SLAM (Extended Kalman Filter SLAM)
- SEIF SLAM (Sparse Extended Information Filter SLAM)
- EIF SLAM (Extended Information Filter SLAM)
- FastSLAM

- GraphSLAM

We will dive deeper into GraphSLAM algorithms as ORB-SLAM 3 falls under this category. The diagram below is how a simple graph is supposed to look like.

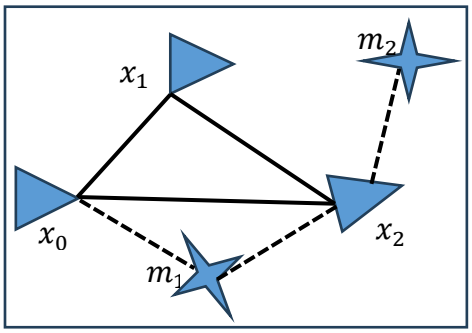


Figure 1: Simple Pose Graph

In the above graph the blue triangles represent a pose at their respective time steps. The solid line connecting two nodes(pose's) is called a soft motion constraint, where soft indicates uncertain measurement. There are two types of soft constraints namely motion constraint which connects two robot poses and measurement constraint which is a dashed line connecting the robots pose and a feature with a dashed line. A star represents a feature. As the robot moves in the environment the graph size keeps growing. The constraints move the robot pose based on the certainty of certain measurements. The goal is to find the best node configuration and minimize the error in the graph. The final graph finds the robot's most likely path across the environment and displays every position and landmark that it encountered, together with an environment map [7]. The GraphSLAM's front-end tasks involves the construction of the pose graph using the odometry and sensor measurements. The front-end of SLAM algorithms greatly differ based on the type of sensor used to perform the task. The back-end of the GraphSLAM is where the optimization of this graph happens and the result is supposed to be the most probable robot poses. The back-end of SLAM algorithms largely remains the same among different applications.

Another way to represent this graph instead of robot's pose is by using a factor graph which is beyond the scope of this thesis. The most popular back-end libraries are g2o, g2sam and ceres. ORB-SLAM-3 uses g2o for its back-end computations. In visual SLAM algorithms bundle adjustment is the popular choice for optimization of the reprojection errors in the pixel coordinates.

GraphSLAM has several **advantages** over other algorithm techniques, such as decreased onboard processing capacity requirements and increased accuracy [7]. GraphSLAM methods estimate an environment's complete path and map rather than simply its most recent pose and map because they handle the full SLAM problem [7]. This makes it possible to incorporate dependencies between the current and past positions, increasing accuracy and facilitating the handling of loop closure. [7]. Furthermore, in improves time and memory complexity over EKF techniques [7]. Despite these advantages the complexity of the graph increases with increase in operational time(large-environment).

2.2.1 Cameras used in SLAM systems

Visual SLAM uses cameras to solve the localization and map building problem. The camera captures a continuous stream of images at a prefixed rate. The type cameras used can be divided in to three categories and we will the advantages and disadvantages of each camera model,

1. **Monocular**

Monocular cameras are single camera setup used for SLAM [6]. The benefits of using a monocular setup lies in its easy setup and low cost. The disadvantage is that we get a low-resolution depth information of a scene. The depth information is calculated from a scene using motion i.e. multiple images are used to determine the relative depth from disparity calculation.

2. **Stereo**

A stereo camera setup is preferred over the monocular setup, to overcome the low resolution in the depth information (section 1.1.2 [6]). The stereo setup is inspired from the binocular vision of the human eyes in determining the depth information. The drawback of stereo system is that it is complicated to setup. Furthermore, it requires a complicated calibration process to maintain accuracy.

3. **RGB-D**

Depth camera setup is based on the design of laser scanner and works on the principle of time-of-flight infrared nature of the light to determine the depth information (section 1.1.2 [6]). This setup however suffers from narrow field of view and works in indoors setting.

2.3 Typical Visual SLAM structure.

As mentioned in the previous sections, the SLAM task is a complex algorithm, and can be broken down into two sections for easier understanding and processing. These sections include the front-end used to process the sensory information to come up with the intermediate representation, and we have the back-end used to perform the optimizations on the intermediate representation. The diagram below [6] provides a simple understanding of visual SLAM structure.

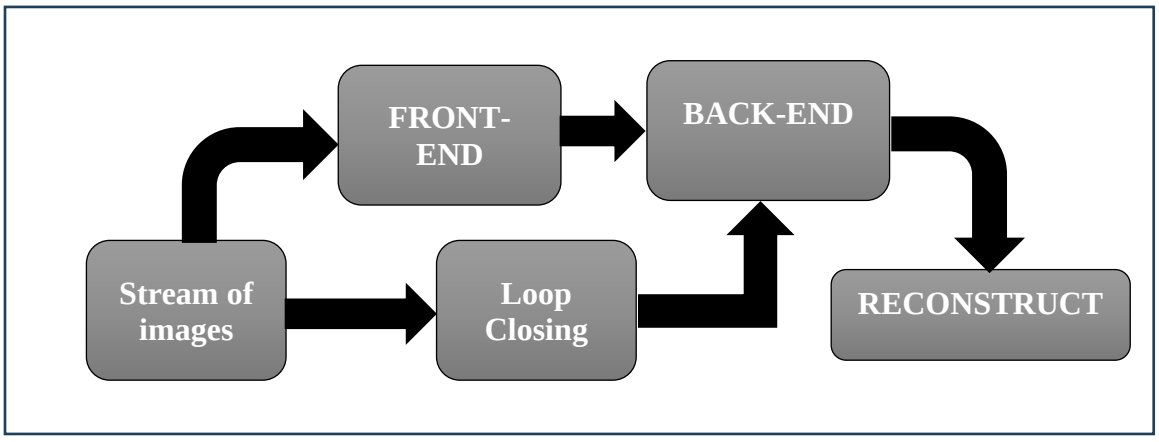


Figure 2: Visual SLAM Structure [6] (figure taken from the introduction to Visual SLAM book fig1.7)

We will look each block of the above diagram in detail.

2.3.1 Front-End

Data Collection and input pre-processing

Data collection can be done using any of the above stated cameras in section. The front-end task processes the incoming stream of images, to find correspondences in the environment and estimate the motion and form an intermediate representation for further processing by the back-end. Servières et al [8] defines the data collection process, as an input search problem, where the aim is to search for meaningful information from the input image stream. There are two ways to perform data acquisition as follows,

Direct Methods

Direct search method stated in Servières et al [8], which uses raw pixels intensities as features which are used to create pixel maps. A pixel map is a three-dimensional representation of the environment, where each pixel value is assigned a depth value. The computational demands for using direct methods and performing mapping often requires parallelization and GPU support [8]. This method directly processes the position and the structure of the environment using numerical optimization techniques. Pose-graph optimization, loop closure and keyframe management are common modules present in both direct and indirect methods [24].

Feature Based methods

Feature based approaches, encode information from an image, to leverage the easily recognizable geometric points of interests, such as, edges, corners, or curve segments etc. Feature based approaches utilize geometric constraints for matching descriptors such as Harris corner detectors [17], SURF [15], SIFT [14], ORB [16]. They are robustness to noise, illumination, rotation, and other inaccuracies in the image and are most commonly type in SLAM systems.

Initialization

The initialization performs the setup of the local map and camera pose for a visual slam system. The initialization also generates the first global world frame for the system. The next task involves triangulation of key points. In stereo camera the two-view reconstruction is employed to triangulate points where as in monocular setup this step is delayed until two keyframes are processed. Triangulation of points is a process performed to reconstruct the three-dimensional (3D) coordinates of a scene from their (2D) projections. The common algorithms implemented by SLAM systems include five-point algorithm, and eight-point algorithm. The essential matrix is computed for a stereo image and as soon as two keyframes are processed the essential matrix for the monocular camera is also computed. Essential matrix relates the 2D image points of a scene observed in two different images. Furthermore, SLAM systems also use approaches to relate the homograph matrix used to compute transformations in planar scenes and fundamental matrix used to compute transformations for general scenes.

Feature Matching/Pose tracking

Computing the matches between two successive images is the purpose of the tracking/feature matching step. The tracking phase can be performed in three ways, depending on the dimension of the extracted features, i) A 2D matching is performed that utilizes techniques such as pair wise Euclidean distance square or normalized cross correlation, and choosing the ones with the lowest values. These methods however are very computationally expensive with quadratic time complexity. These methods are seen in systems that perform pure visual odometry [8] ii) 2D -3D matching [8] is commonly used technique in VSLAM system that are purely monocular because of lack of depth information from single images. These processes estimate the coordinates of a 3D point from successive images and use projection geometry to make 2 D correspondences with the

new image. iii) The third method is used when the SLAM system has a stereo camera setup that provides depth information instantly to make the 3D-to-3D correspondences. However, these techniques are prone to reprojection errors and are still open research areas.

Mapping:

The matched features or pixels are supposed to be mapped onto the 2D or a 3D reconstruction graph of the environment. Based on the method of intermediate representation, there exits two ways to build graphs, sparse graph representation, and dense graph representation, as per Servières et al [8]. Sparse graph representations are preferred when an application demand is to acquire the most accurate trajectory of the robot. Dense graph representation is preferred when a reconstruction of the environment is desired. A combination of both dense methods and sparse method is also stated in [8, 27], in which only specific required areas of a graph are dense.

The SLAM systems that use monocular camera setup require some time for processing landmark matches. To solve this PTAM [18] suggested a partial computation and placement of poses. The direct methods of creating maps use the 3D world coordinate and map it voxels in the map. However, pixel mapping has been found to inaccurate due to lack of gaussian probability factorization. Alternative methods to provide accurate representation and accounting for nonlinearity have been suggested in “homogeneous point (HP), anchored homogeneous point (AHP), and inverse depth parameterization (IDP)” [8].

ORB SLAM - 3 system builds a sparse graph representation using its ORB features. Map building can also be performed with a collection of spatial points representing the environment or a map of trajectory. Xiang et. al [6] categories maps into metrical maps and topological maps.

Metric maps [6]

Metrical map is supposed to replicate the exact metrical location of objects found in the map. These maps are further classified as sparse maps or dense maps [6]. Sparse metric maps perform selective representation of the environment. A similar categorization of the map by Servières et al [8] is known as semi dense representation of the map. The dense map on the other hand is supposed to represent as many details as possible about its environment. The dense map can be a 2D dense map in the form of an occupancy grid or it could be represented in a 3D form know as voxel grids. Metric maps are known to be storage expensive and continues to be an active area of research.

Topological Maps [6]

A topological map is a graph generated by the mapping algorithms with nodes and edges. These relational maps are supposed to provide efficient representation for connectivity explanations.

Outlier Management in graph:

Due to the nature of estimation in the intermediate representation creation, i.e., converting real world environment, which continuous in nature, to a discrete representation, causes outliers to be added to the graph. These outliers in SLAM systems are typically removed using a function estimation technique known as Random sample consensus (RANSAC). The RANSAC technique, picks randomly possible observation candidates, and estimates a function, following which a consensus is performed to find the best fit. This RANSAC technique is applied over the camera motion model. Most of the SLAM algorithms also add more candidate nodes than necessary (keyframe or mappoints) to maintain robustness of the system. However, these additions add a considerable amount of memory cost to the system. Therefore, redundancy detection, loops are added in SLAM architectures to detect redundancy and delete such observations from the map.

Loop Closing:

The loop closure, is an algorithm, applied to a robot's graph or trajectory, when the robot revisits a known environment. The algorithm applied aims to apply corrections wherever necessary to the graph, where there is an accumulation of drift. Loop closing step is performed in two steps, i) Loop detection: given the current keyframe features or pixel values, searching through the intermediate representation graph for similarities. This is also known as place recognition. In situations where robot makes sudden movements, can lead the robot losing its position in the graph. Place recognitions is used in similar situations to resocialize the robot. Most SLAM systems use Bag-of-words representation, provided by the C++ library DBoW2 [43] to find matches, ii) Loop Closure: Upon detection of a loop, the total accumulated drift for graph is computed, this total amount of drift is then distributed appropriately for each node of the graph to complete the loop closure.

Relocalization:

The relocalization is similar to loop closure where, if the robot has lost its position in the graph then the place recognition algorithm similar to loop closing is applied. However, unlike loop closure, relocalization is only triggered when tracking is lost. In majority of the implementation, loop closure is performed in separate thread due to its computational complexity.

2.3.2 Back-end

Primarily, the back-end components of a SLAM system are tasked with processing the noise and uncertainty present in the graph, and optimizing it, and to uniformly distributing the weights of these uncertainties among each node(pose/keyframe). The uncertainty could include the robot's trajectory or the map environment.

Bundle Adjustment (BA):

BA is a state estimation technique, that is used to estimate 3D points in the environment based on the extracted features and camera images [48]. It particularly estimates the precise orientation of the camera poses in the graph. Based on the prior/existing estimates of the position of the 3 D point coordinates and the camera parameter matrix (essential matrix/ fundamental matrix) a reprojection of the actual points in the world are introduced on to the graph. An error parameter know as reprojection error is used by bundle adjustment to find the best possible estimates. This problem is known as the Large least squares problem which requires solving large system of equations to correct the reprojection error. The resulting solution is proposed to be a statistical optimal solution, with an assumption of the noise to be gaussian.

Graph Optimization:

The graph used in most SLAM systems is a factor graph (bipartite graph) representing the factorization of a function [49]. This reduces a large function into product of its component function. In SLAM systems this function is a joint probability distribution function of the poses estimated by the front end of the SLAM system. This joint probability function is then applied to algorithms to provide inferences of the position estimates.

2.4 Memory Management and Concurrency Control

Efficient management of critical resources in a computing system is crucial, especially in real-time systems. Because memory is finite in a computer, it is common for programs to exceed the size of the onboard memory. Hence, we need methods to reuse memory and we need to perform reclamation of memory safely. Programming languages provide developers with the necessary mechanisms to manage memory in their application. Selecting a programming language is a critical decision that influences the success of a software project. It plays a crucial role in performance of the application, development efficiency, scalability, and long-term sustainability.

Programming languages are designed in two ways to address the task of memory management: automatic memory management and manual memory management [20]. Manually managed languages such as C and C++ provide developers with explicit control over the allocation and deallocation of memory. On the other hand, automatically managed languages, such as Java and Python have mechanisms such as garbage collection that automates the task of manual memory management. These languages mark the memory that is no longer in use, and reclaim all marked memory locations.

Real-time systems are generally written in C/C++, where memory is managed manually, leading to the following potential challenges like, inability to ensure type safety when converting void pointers to any pointer, passing raw pointers across different threads, improper ownership transfer, and unintended extension of the lifetimes of objects present notable challenges in a C/C++ codebase. Hence, establishing methods for efficient memory reuse and implementing secure memory reclamation processes are essential steps to enhance systems robustness and reliability.

For a language to exhibit memory safety the Arthur et .al [19] states that it offers spatial safety (bounds checking), and temporal safety (every memory block allocated should have a unique identifier). An ideal programming language should [19] i) raise errors due to improper memory access, ii) enforce only unique identifiers to memory locations and enforce immutability to these identifies, iii) restriction in revealing the identity of an identifier preventing any memory misuse, iv) enforcing initialization of every new memory location preventing accesses to information in a previously freed memory location.

Before we dive deeper into the types of automatic memory management strategies let us look at a typical program and data layout.

2.4.1 Types of Allocations and Typical program layout

Most of the modern programming languages provide static, stack and heap allocations [27]. The typical data layout in memory is given by the following diagram.

Figure 3: Memory Layout

Static Allocation [27]

In static allocation the data static data objects are stored and its layout is determined at compile time. Additionally, all the static variables are bound to the initial memory locations. The advantage of the static section is that it is very fast and does not require memory management as it does not support runtime allocations. However, the size of allocations and variables needs to be fixed at compile time leading to limited functionality. Another disadvantage is that it does not support recursion because the value of recursive variable is always fixed thus forbidding recursion.

Stack Allocation [27]

The stack provides a way to perform recursion, by utilizing a mechanism called as activation records or stack frames. In this allocation type, for every new subroutine call, a new stack frame is created, and pushed on to the stack. The return of a subroutine triggers the deallocation of the stack frame. The stack model also allows parameters of different size, which allow creation of variable stack frames sizes. In programming languages like C/C++ or java the stack memory management is fully automatic. The disadvantage of stack allocation is that the size of a stack needs to be determined at compile time causing a limitation in providing runtime allocation capabilities.

Heap Allocation [27]

The heap memory introduces runtime capabilities which allows creation of objects at runtime and returning a pointer of the allocated type. Heap memory also allows variable size data structures to be created at runtime. Allocating data in the heap, also avoids causing stack overflow. The design of heap however, changes the responsibility of managing memory from the language to the user. This resulted in issues like memory leaks and dangling pointers. In the next section we shall look at manual memory management and automatic memory management.

2.4.2 Manual Memory Management

Manual memory management provides the developer with the capability to manage dynamic memory(heap) directly. The new construct in C++ provides developers with the capability of heap allocation and delete to deallocate memory. However, manual memory management causes two critical issues i) memory leaks ii) dangling pointers

2.4.3 Automatic memory management: Garbage collection

The contents of the following sections are based on The Art of Automatic Memory Management by Jones et. al [20], Memory Management and Garbage Collection CS 4120 Spring 2023[21] and Dmitry Soshnikov blog on writing memory allocator [22].

Garbage collection automates the issue of memory management at a language level and solves the two critical issues from the manual memory management namely memory leaks and dangling pointers. However, it should be noted that, any memory automation comes with a tradeoff such as we storage for speed or speed for storage. In garbage collection, we store meta information on each object which is known as the object header. For example, if a user were to request 5 bytes from a managed language, then the resultant object returned could have a size of approximately 14 bytes to 24 bytes [23, 22] or even more.

The contents of the object header could have the following, a mark bit/flag bit, reference counter etc. depending of the method of garbage collector we use. The location of the object header could be at the before the user data or it could be at the end.

The figure below provides a visual example of the 5-byte allocation.

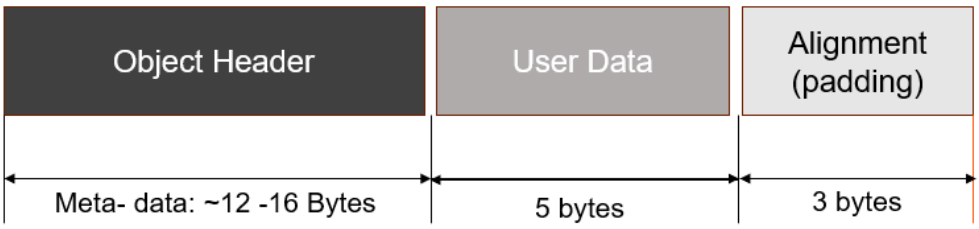


Figure 4: Five-byte allocation with object header

Every garbage collector can be described by, three components namely, the mutator, the collector and the allocator according to Jones et. al [20].

Mutator

For a garbage collector, a mutator is the main program or the user program. The user program operates on the heap allocated objects. However, the mutator does not allocates the heap objects, another module known as the allocator performs the allocation. The mutator in a multi-threaded application can have more than one thread manipulating the heap.

Allocator

The allocator directly manipulates the heap by acquiring the resource from the operating system, and it also assigns the object header. Allocation for memory is performed by a sequential allocator (bump allocation) or a free list allocator[31]. The sequential allocation method is used by the mark-compact, and generational garbage collector etc. whereas the free-list allocator is used by the mark and sweep and the reference counting garbage collectors [31].

Collector

The collector or the garbage collector, is responsible for reclaiming the memory and it also directly manipulates the heap. In a multi-threaded implementation, the collector can be implemented using multiple threads. The mutator is put in a "Stop the world" Jones et. al [20] state where all mutator threads are blocked when the collector processes garbage on the heap. Such garbage collection cycles are termed as "GC pauses" [30]. It should be noted that not all systems can have such pause cycles, for example real-time systems, hardware drivers and transactional processing cannot tolerate GC pause cycles [34]. The garbage collectors are classified as tracing collectors and direct collectors. The garbage collectors that traverse the heap to classify live objects are called the tracing collectors. Tracing collectors requires a GC pause cycles to perform heap traversal and deletion. Direct collectors do not require gc pause cycles and examples of direct collection include reference counting discussed in greater detail in the section.

Before we explore garbage collector in a greater detail, let us understand a few terminologies

Root: A root is considered as the starting point of the object graph or the references that are live and accessible from the stack.

Liveness: A object in the heap is alive if it is accessible from the mutator

Reachability: An object is considered reachable if we can traverse the object graph from the root node at reach that object.

2.4.3.1 Mark and Sweep Garbage collector [20]

The mark and sweep Garbage collector are a tracing garbage collector, that searches and marks live objects on the heap [20]. The mark and sweep collector work in two phases, i) mark phase that marks live objects, ii) the sweep phase that reclaims the garbage. The mark bit, is set when the object is alive in the object header. All nodes that are marked as garbage are freed and added to the free-list for future consumption. Another important to note is that the mark sweep algorithm is a non-moving collector. This means that the objects after the GC cycle stay at the same location. Mark and sweep algorithm cause heap fragmentation that causes increased cache misses.

2.4.3.2 Compact Collector

Mark Compact

Mark compact garbage collector is supposed to provide a better cache locality and faster memory allocation than the mark and sweep algorithm. The mark compact collector works in two phases, i) mark phase which traces the heap and sets the mark bit of the live objects. ii) the compact phase moves the live objects. The objects in a mark compact collector, have a forwarding address field in its header which denotes where an object is moved. This is performed to reduce fragmentation. Another benefit is that we get to use the bump allocator which results in faster allocations. The

mark compact collector, however is a slower garbage collector as it may require multiple traversals of the heap.

Copying collector

The copying collector offers a fasters garbage collector than the mark compact collector. Furthermore, it provides bump allocation which results in a faster allocation. The copy collector however trades storage (half of the heap reserved for collection) for speed. The heap is divided into two equal parts, i) from space (area for allocation) and ii) to space (area reserved for garbage collection). The copying collector works in four stages, i) The first phase, is the tracing stage or copying stage where we traverse the object graph, copy all the live object pointers from the from heap to the to heap. This process is fast as the bump allocator is use in the to section of the heap for allocation. ii) Then we have the forwarding address phase, where the object header of the heap object in the from section is added the forwarded address of the copied object present in the to portion of the heap. iii) the next phase is the child pointer fixing stage, where all the child pointers of the previously allocated space are transferred to the new section. This process is repeated until all child objects are moved to the from section of the heap. iv) The final step is the swapping step where the sections from, and to are swapped by changing the bits of each section.

2.4.3.3 Generational Collector [38]

The generational garbage collector is based on the hypothesis, which states that most objects die young [38]. These typically include local temporary variables whose lifetime is like that of the local stack variables in the subroutine, however, these variables are allocated on the heap. The generational garbage collector works by having two separate heap sections i) young (Eden) generation and ii) old (Tenured) generation. The objects in the young regions are copied to the old section when they survive several cycles heap collection. The young generation section is garbage

collected more frequently than the old generation section. The garbage collection for the young section is known as the minor cycle whereas the one for the old section is called the major cycle. In situations where we have a pointer from an object in the old region to an object in the young region, we term it as an intergenerational pointer. In situations where the root link to such an object in the young region is lost, however if it still contains an intergenerational link, then such an object needs to be saved during the GC cycle of the young region. This is done by a write barrier which saves the object in the young section from deletion.

2.4.3.4 Immix Collector [39]

The immix garbage collector also known as mark region garbage collector is a modern garbage collection algorithm [39]. The immix GC is a tracing collector which attempts to provide better cache locality compared to the methods seen thus far, reclaim memory faster than the methods that we have seen yet. Immix GC also tries to overcome the challenges faced by the copying GC. The immix garbage collector partitions the heap granularly. It begins by creating blocks in the heap. The bifurcation can be predetermined or can be done on demand when current block is exhausted. Further, the block is divided into lines. A block is said to be free if all lines in a block are free. A block has several lines to be free and rest occupied or the whole block can be occupied. The garbage collection begins with the marking phase which is done during tracing. The marking is done in three ways, a region(block) is checked if the alive bit is set, if it is not set no traversal is needed. However, if a block is set to alive, then the lines inside the block is checked. If a pointer from the root exists to a line, then traversal is performed to mark all live objects. Then all the lines that are not set to alive are freed. This completes the mark-region and sweep phase. Then the copying of all live objects is done to a freed block, by copying and forwarding the child address

like the copying collector. This finally results in a defragmented heap and the previous block is reclaimed by the mutator.

2.4.4 Reference Counting

The reference counting collector is a direct collect. The garbage collector strategies that we have seen in the above sections, traverse the object graph to identify live objects. In case of a reference counting collector, are directly work on the object to determine garbage. The reference counting scheme works on a invariant which is that an object is garbage if and only if its reference count is zero and in all other cases it is a live object. For the reference counting scheme to work every object must have slot in its object header that represent the reference count.

A simple reference counting scheme is given below taken from Jones et. al [20],

```
REFERENCECOUNTING()
addReferece(ref):
    If ref ≠ null
      rc(ref)++
deleteReferece(ref):
    If ref ≠ null
      rc(ref)--
      If(rc(ref) == 0)
          free(ref)
```

Now we shall look at the methods of reference counting found in the Art of Automatic Memory Management by Jones et. al [20].

2.4.4.1 Deferred reference counting

Deferred reference counting was introduced to mitigate the cost associated with manipulating the reference counts. The algorithm postpones reference counting of local variables, such as registers or stack slots, as most pointer loads are to these variables. However, this may introduce in accuracies in the maintained reference counts. To maintain accuracy the deferred reference

counting (RC), the deferred RC has pauses cycles to correct the inaccuracies and maintain precise reference counts.

The deferred RC uses a Zero count Table (ZCT), which keeps track of the objects whose reference count goes to zero. Objects placed in the ZCT are not deleted immediately due to possible inaccuracies in the deferred operations. As soon as the pause cycle is introduced the objects in the ZCT are checked for their true count by traversing the roots and marking all referenced objects.

This reference counting method although reduces the cost associated with reference counting local variables, this strategy introduces the pause cycles where object headers must be updated and must be done atomically. The cost associated with performing atomic updates are expensive and should be accounted when planning to use a deferred counting strategy.

2.4.4.2 Coalesced reference counting

The coalesced reference counting was introduced to reduce the cost associated with deferred reference counting's atomic updates. It is based on the observation that when an object goes into intermediate stages, the reference count of such stages can be coalesced into two stages, the before stage and the after stage. The intermediate increments are canceled out with the decrement reference counting counts. The author provides an example as follows, let us assume an object X that refers to an object O_0. The object O_0 in turn refers to objects O_1,O_2,… O_n. The reference count of object O_1 increment is cancelled by its decrement and so on, which can be omitted. The method places eliminate such counts by copying the objects to a local log before an intermediate modification. This local log is also known as a local buffer. The method begins by placing a clean object (object whose pointer fields is not modified) onto the local log. This means that the objects address and the pointers that it possesses are placed onto the buffer. To avoid duplicate entries, the

source object address is checked if it is dirty or not. If not, dirty it is logged. This algorithm also guarantees thread safety where every thread local buffer is supposed to hold the same information.

This reference counting scheme also requires a stop the world event to process the local buffers to ensure consistency. During the pause cycle the local log is synchronized among all the threads, and any duplicate entry, due to concurrency may exist, but is ignored if it has been accounted for. Before the references count is updated the algorithm check for dirty entries. If such dirty entries exist the count of the children are incremented followed by a decrement. Any reference count at this stage if it reduces to zero, then the object is freed and its entry is removed from the log to avoid double frees.

2.4.4.3 Handling Cyclic structure using reference counting

The major disadvantage with reference counting is that it cannot deleted cyclic structures. Common ways to deal with such cyclic structures can be done by combining reference counting with a tracing collector. The cycles for the tracing collector do not need to be very frequent and can be performed only to delete cyclic structures and all other objects are deleted by reference counting. Another popular method to reference counting cyclic structure involves having different types of references. We could define weak references and strong references where we use a weak reference to close a cycle. A weak reference by itself would not increase its reference count. However, a reference counting scheme based on weak references quickly becomes vulnerable to memory leaks and premature deletion if not placed correctly. The most widely used strategy to deal with cyclicity is the trial detection algorithm [20]. Its algorithm takes advantage of the observation that in using a tracing collector in conjunction with the reference counting scheme, the tracing collector does not require scanning the whole object graph, rather it can accomplish

detecting cycles by tracing the portion of the graph that has the highest probability of being in the graph.

Advantages of Reference Counting

Since the managing of pointers, to objects, are not concentrated at specific time instances, the time cost for collecting garbage is distributed through the program lifetime. Memory is also recycled as soon as the reference count drops to zero leading to quick deletion as opposed to garbage collection where deletion happens only during GC cycles. According to the author Jones et. al [20] reference counting has found widespread adoption in languages like objective-c, swift and in C++ it is used in smart pointer implementations and applications like photoshop. Furthermore, it is also widely used in file manager of operating systems.

Disadvantages of Reference Counting

To maintain consistency in the reference counting atomicity of the increment and decrement operations are required. Adding atomic increment and decrement operation on every heap with references adds a considerable overhead on the mutator int terms of time and computational cycles. In contrast garbage collected programs do not impose such costs on the user program or the mutator. As seen from the above examples reference counting alone cannot be used for deleting cyclic structures. Since reference counting requires an integer to be maintained for counting, it adds a storage overhead on every heap object allocated.

The next section explains how language developers define memory models to control concurrency in the applications.

2.5 Memory model

A memory model provides the basis for how threads should interact with shared data and highlights the possible read and write operations in a concurrent program, thereby offering semantics to

shared variables [29]. Furthermore, it accounts for possible reordering done by the processor, memory system and compiler. According to Manuel et. al [28] reasoning of a memory model requires distinguishing three different components i) the program a developer writes, ii) the code the compiler generates and iii) the operation the CPU performs while executing the code. A typical workflow of a program C++ program begins with a collection of logical statements written by a developer, which is compiled, optimized. The optimization process could possibly involve reordering of statements which are not supposed to change the semantics of the program. These side effect could result in unintended results such as security vulnerabilities and memory issues etc. [33], especially in a multi-threaded system. The role of a memory model is to place restriction on these reordering's so that the developer can reason through the likely behavior of a multi-threaded program.

A memory model must define legal reads, writes to shared memory location simultaneously Sarita et.al [29]. It should also provide visibility of the changes made between different threads when changes are made to the program by the hardware or the software. A memory model serves as a template for coordination between a developer, compiler, and the hardware. Java developed a concurrent memory model around 1995[The Java Memory Model (umd.edu)] attempting to provide type safety and security guarantees which severed as a basis for C++ 11 memory model. C++ 11's memory model is a part of the core language and $std::thread$, the primary mode for multithreading in C++ is also part of this standard library. In summary the C++ memory model should provide, sequential consistency guarantees.

2.5.1 Sequential consistency Memory Model

The sequential memory model borrows its ideas, from the memory access mechanics of a single threaded application. In a Single threaded application any read instruction, results in a fetch

instruction, of the most recent write value to a memory location [28, 29]. Most recent means the last write instruction according to the program text [28, 29]. In a multi-threaded program, if we were to follow similar mechanics, where we combine all the instructions from the program, and order them precisely(interleave) into a single order [28, 29], then the resulting multi-threaded program would follow the Sequential consistency Memory Model. The author Sarita et.al [29] states that such a memory model suffers from deficiencies, i) The practical implementation of such a model is not feasible as it is very computationally expensive and as per Manuel et.al [28] 2018, none of the hardware architectures provide a complete sequential memory model. ii) Another noted deficiency stated by Savita et.al [29] is that the current, hardware and the compilers often make memory access visible to other threads that may not be in order thus violating the sequential consistency rule.

We shall now look at a hardware model and the C++ memory model which are taken from Manuel et. al [28].

2.5.2 Relaxed/weaker Memory Model

1. **x86** [28]

The X86 is a memory consistency model used by the intel x86 architecture and it describes the interaction between different hardware threads and the memory. The memory model can be simplified as given below,

- **Storage Subsystem:** The storage subsystem includes a shared memory, a global lock and write buffers per hardware thread and each thread has its own write buffer [28].
- **Global lock:** indicates when a particular hardware thread has access to memory [28].
- **Store Buffer**: It refers to a First in First out structure. Whenever a particular thread tries to access its store buffer, it is supposed to fetch the most recent buffered writes. If a threads

store buffer is empty then the information is fetched from the shared memory. Store buffers are flushed using the mfence instruction [28].

- **Read-modify-Write Instruction**: Whenever hardware thread executes a read-modify-write instruction, it must first acquire a global lock. Once it successfully acquires a global lock, necessary writes are performed, and the thread buffer is flushed followed by a release of the lock [28].
- **Propagation of Buffered Writes**: A threads buffered writes can propagated to the shared memory at any time except when other threads have acquired a lock.

The below figure, is a simplified x86 block diagram [28]

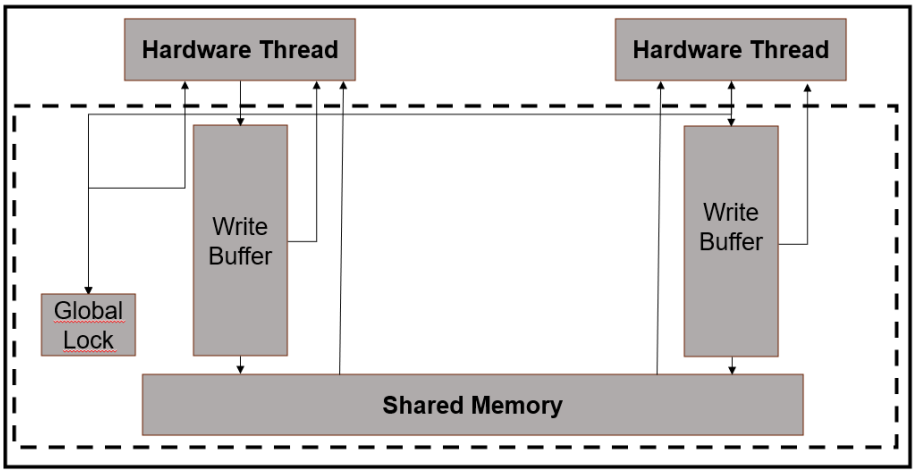


Figure 5: Simplified x86 block diagram is taken from "C/C++ memory models" by Arthur et.al [28]

2. C++ Memory Model

The C++ memory model is providing constructs such as C++ atomics and mutexes to provide visibility among threads and maintain synchronization operations on memory locations. C++ also presents a synchronization construct fence, that have similar function to a mutex where threads can acquire and release fences. Fences are provided for achieving synchronization of operations that do not involve memory locations.

The execution of a C++ program is defining two types of computations,

- Value computation: The program either steps to a value (that needs to be stored at memory location) or returns a value at a memory location.
- Side effect: All other operations such as reads/writes to a volatile object, writing to a memory location, calling a library function and IO's are side effects.

C++ compiler however is not required to abide by the C++ standard stipulated, if a similar behavior of the program is observable after code transformations.

The C++ standard also defines the definition of a data race,

A data race is supposed occurs in a multithreaded C++ program, if two (or more) conflicting threads, are in the condition, where neither of the two threads happens-before the other and at least one of the threads performs a non-atomic operation [37].

Conflicting thread action is supposed to occur when one thread performs an expression evaluation that results in modifying a memory location while thread evaluates an expression that results in either modification of the same memory location or accessing the same memory location [37]

The data race definition mentions, happens-before relation, which means that, if one thread performs an expression evaluation that mutates a memory location then another thread executing another expression on the same memory location can see the mutation result of the first threads execution. If two threads are running in such order then they are supposed to follow the happens before relation[28]

Now we shall look at the mechanisms, provided by C++ to achieve thread safety during a data race condition.

C++ Atomics

An operation is termed as atomic, when the operation is guaranteed to execute as a single transaction. Furthermore, other threads will see the state of the memory location either before this operation has begun or after the operation has been completed. The benefit of this guarantee is that there is no intermediate state visible to any thread, thus abiding by the happens-before relation.

C++ provides a library for atomic memory operations. Atomics in C++ provides a faster means of managing concurrency when compared to mutex locks. However, it should be noted that atomic operations themselves do not provide faster operations when compared to mutexes. A main reason for such a behavior lies in the size of the critical section. The C++ standard states that if the size of the atomic class is less than or equal to the size of a pointer the operations such as increment, decrement store add and compare and exchange are atomic. However, if the size exceeds the size of the pointer, then the operation becomes lock based.

C++ compare and exchange.

The atomic compare and swap are a conditional exchange. An atomic exchange is a swap operation consisting of read-modify-write operations done atomically. The syntax for a compare and swap operation is given by,

$$x.compare\_exchange\_strong(expected\_value, new\_value)$$

Where,

x is an atomic variable declared and initialized by std::atomic<int>x(0)

expected_value is an int

new_value is also an int

The above expression returns a Boolean true if the swap was successful, else it returns a false.

The compare and swap loop of a typical C++ program that performs an increment operation is given by,

$std :: atomic < int > x(0);$

$int\ x0 = x;$

$while(!x.compare\_exchange\_strong(x0, x0 + 1)$

This loop works by first reading the value of x and storing in a local variable. If none of the other threads change the value of x, then the current thread changes the value of x by incrementing. This loop continues until the increment is successful. Another point to note is that x0 gets the value of the new x if the compare and swap fails.

We shall now conclude this section with a few C++ features that would be used in the implementation of memory management in later sections.

2.6 Smart Pointers in C++

Raw pointers in C++ allow variables to store memory addresses. However, unlike higher-level abstractions, raw pointers lack automatic memory management features, such as deletion upon going out of scope, which can result in memory leaks. C++ raw pointers also allows sharing of memory which means that any thread or part of the program can manipulate a memory location. Such sharing, lacks ownership (responsibility over a memory location) of memory, and improper use of raw pointers can potentially cause erroneous or unpredictable behavior such as data corruption, race conditions. Smart pointers namely unique pointer and shared pointer were added to C++ to handle these issues. Smart pointers are classes that mimic the behavior of a raw pointer, but also provide automatic memory management when the destructor for the smart pointer object is called.

2.6.1 Unique Pointers

The unique pointer is a part of the memory library in C++, is a scoped pointer which means it automatically deletes itself once the pointers scope ends. Unique pointers are also not allowed to

be copied (no data sharing or ownership sharing). However, unique pointers allow ownership transfers.

2.6.2 Shared Pointers

Shared pointers allow sharing ownership of a memory location and are implemented using reference counting. The last shared pointer object to a jointly owned memory location handles the deletion of the object. We shall now see an example of how shared pointers are implemented in greater detail. This example is taken from David Kieras [32] tutorial on using smart pointers in C++11.

Let m be an object created in the heap and it is pointed by a shared pointer $ptr1$. The constructor for the ptr1 object also creates a "manager object" [32] which is also dynamically allocated, which in turn points to the heap object. This "manager object" [32] is already present, when we make pointer copies. The "manager object" [32] carries two integers, one the number of pointers that are pointing to the heap object and another the number of weak pointers pointing to the object. In this example we restrict to only shared pointer counts. The figure below illustrates this process.

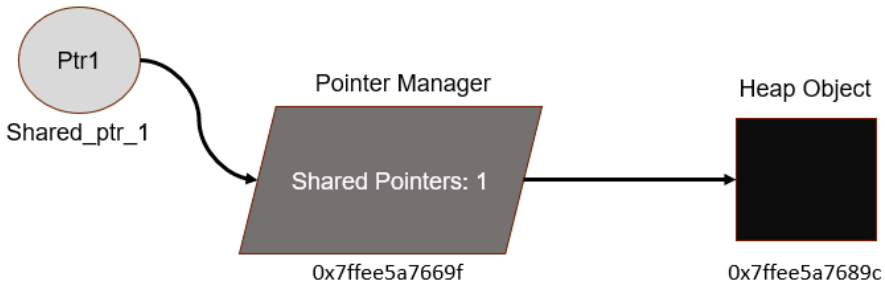


Figure 6: Shared Pointer Example

If we were to make two more pointers to the heap object, by copy or by assignment, then the "manager object's" [32] count would increase to three as show.

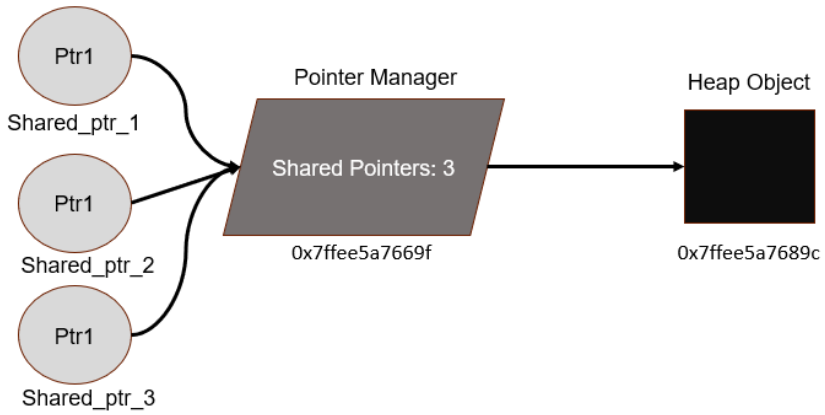


Figure 7: Example of Shared Pointer

As the scope of each shared pointer has reached the end, the reference count keeps decrementing, and the shared pointer with the longest scope is the one that deletes the manager object and the heap object. If we were to have a cyclic structure, then shared pointers would not be able to delete the heap object thus leading to a memory leak.

2.6.2.1 Weak Pointers

Weak pointer is also known as an observer pointer, that is a not-owning a reference. Weak pointers provide a way to break a cyclic structure. Unlike shared pointers weak pointers do not have the ability to create a manager object. When all the shared pointers to a object have gone out of scope the and if we still have a weak pointer then, this weak pointer has the ability to retain the manager object. The manager object as mentioned in the previous paragraph also has a integer count of the number of weak pointers pointing to it. If the weak pointer references to the manager object goes below zero then the weak pointers destructor performs the deletion of the manager object. Weak pointers are different from a regular pointer, where we can query a weak pointer to check if the manager object (heap object) still exists or not. One distinction to note is that the heap object will be deleted as soon as the last shared pointer to it goes out of scope however the manager object is deleted either by the last shared pointer if no weak pointers exist or by the last weak pointer. The

weak pointer does not have the ability to access the heap object directly, they can only create new shared pointers or check if the heap object exists or not. If the heap object is deleted and we create a shared pointer from a weak pointer we would just get a null pointer.

2.7 Related work

There have been proposals to reorder map information in the form of long-term and short-term memory [2] [12,13]. Long-term and Short-term memory representation is utilized in [7] RTAB-Map to offload map information to a SQL database, which has a lower likelihood of being used by loop closing and retaining rest of the valuable information. The SQL model is noted to limit the extension in large-scale map-making. Edge-SLAM employs an edge computing infrastructure to offload local mapping and loop closing while keeping track of the mobile device. C2TAM [2,11] provides a distributed cloud computing framework where the tracking and relocalization are performed on the client device while rest of the modules are sent to the cloud service in addition to the map storage. Li et al. [2] present a redesign of the VSLAM system based on a cloud-based solution where map storage, map fusion, and real-time components are offloaded to a cloud, keeping the remaining components on the robot.

3. ORB-SLAM-3[40]

ORB-SLAM-3 is like the previously discussed Visual SLAM architecture in sections above. The tasks that it performs can be divided into the following,

- Front-End tasks
    - Feature Detection
    - Feature Matching
    - Pose Estimation
    - Adding data associations to the co-visibility graph
- Back-End tasks
    - Bundle adjustment
    - Map optimization

Apart from the above stated task the ORB-SLAM-3, also does loop closure, that is responsible detecting loops and correcting them. To speed up the system, ORB-SLAM-3 runs of three threads, namely tracking, local mapping and loop closing [41]. The structure of ORB-SLAM is based on

the Parallel Tracking and Mapping (PTAM)'s parallelization of mapping and localization architecture, with an addition of a loop closing thread [41].

We shall now note the following definitions,

Mappoint: A mappoint corresponds to the three-dimensional coordinate in the world and is derived from the ORB-Features extracted [42].

Keyframe: A keyframe represents the pose (position and orientation) of a camera at a certain time instance [42].

Map: The map is the collection of mappoints and keyframes [42]

Covisibility Graph: This is a weighted graph of connected keyframes, where an edge in the graph represents connection between two keyframes and the graphs serve as nodes [42].

The figure below gives an overview of the ORB_SLAM-3 system

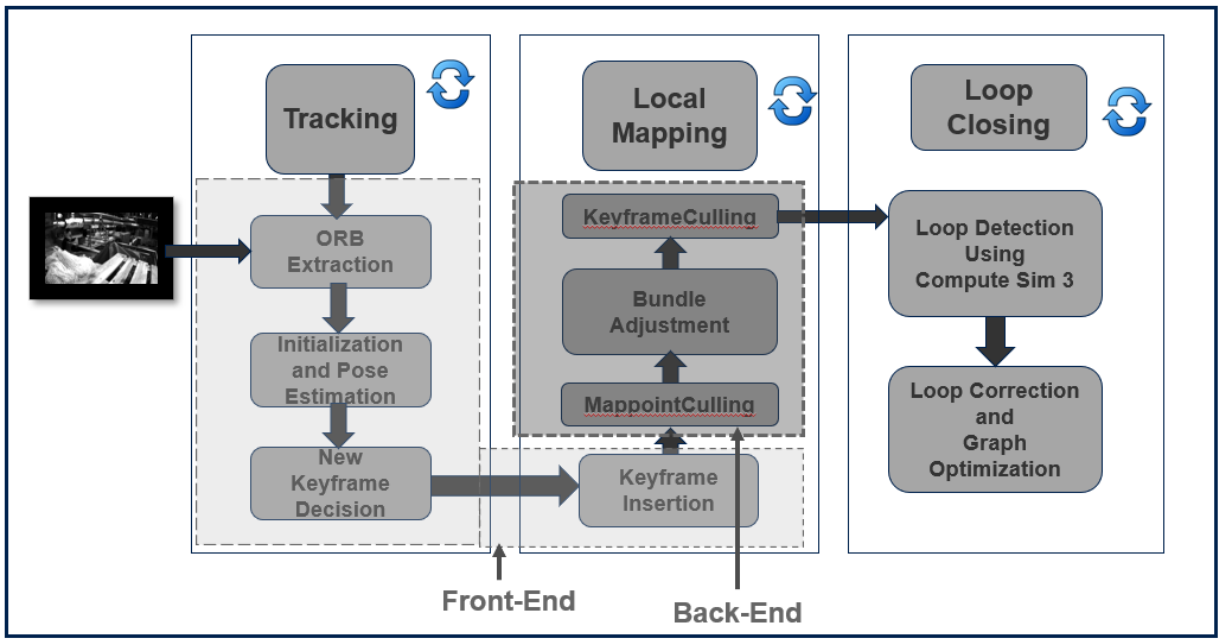


Figure 8: ORB-SLAM-3 Structure

3.1 Tracking Thread:

The tracking thread begins with **extracting ORB features** by generating an image pyramid of the image being processed by the tracking thread. An Image pyramid is an image processing technique, used for image reconstruction among others. The incoming image is sampled at various scales by repeatedly dividing the resolution by half and applying a gaussian blur. Now we have a representation of the current tracking image at different scales. Now the image representation at different levels, is passed through a function to compute the FAST (Features from Accelerated Segment Test) corners [44]. FAST is a corner detection technique, that is rotation invariant and computational easier(faster) to compute than other corner detection algorithm. The final step is to compute the ORB feature descriptor, which is an extension of the computed FAST descriptors with an exception of adding intensity centroids [35] and the rotated d Binary Robust Independent Elementary Features BRIEF feature descriptor. The BRIEF descriptor is used because it has low memory footprint and fast to compute [45]. ORB SLAM-3 then uses these computed features and maps them corresponding to the image positions based on the camera parameters provided by the system. This feature extraction step is performed over every incoming image and stored in a temporary object known as the frame corresponding to its location coordinate.

As images are added, we perform feature matching, which is performed by using three different methods namely, search by projection, bag-of-words, and similarity transforms, and then, the strongest matches are retained to be added to the map [41]. To place matches in a map, a critical task of **initialization** is performed to create an initial map provided at least a threshold number of features is detected. The very first frame is compared against incoming frames for feature matching. All features detected in the first frame is compared with the current frame of the tracking thread. If matching is successful the algorithm processed to compute the required transformation matrices such as the homography matrix and the fundamental matrix depending on the matching

of the initial frame and the current frame using RANSAC [46, 47]. The optimum homography and the fundamental matrices are fixed to finally determine the depth of the features onto the world coordinates. Then the first frame, and the matched frame are both converted to a keyframe all corresponding mappoints are added to the map. This marks the completion of initialization, and since we have estimated nodes present in the graph, we proceed by performing a bundle adjustment of the graph. The tracking then continues to perform feature matching with every incoming image. Tracking also performs **relocalization** which was introduced in the background section in case of sudden movements or drop in tracking performance.

Code Structure Details:

The Driver code which is also the tracking thread initializes a system object, which in turn spawns the necessary threads, local mapping, loop closing, and viewer(optional). Once necessary initialization is completed, the tracking thread loops over all the images of the dataset or could run on a live stream of images. Each image after resizing, calls the function $system.TrackMonocular(image)$ (based on the camera model, which in this case is monocular) that performs a couple of system and timing checks for synchronization and then triggers the function $GrabImageMonocular(image)$. This function converts the image in grayscale and creates a Frame object. The constructor for the frame object, calls $ExtractORB(imageGray)$ performs feature extraction. At the end of the $GrabImageMonocular()$ function the function, $track()$ is called. The track function performs necessary initialization of the following objects,

- Atlas: Collection of Maps (done only once)
- Map: Collection of Keyframes and Mappoints

The track function further calls the $TrackReferencekeyframe()$ that does the matching of the current keyframe and the previous keyframe. The function also converts the features in the current frame into a bag of words representation. Furthermore, it tries to use the computed bag of Words to find possible matches with the previous keyframe. $track()$ further calls $TrackWithMotionModel()$, which does the feature matching by a function called $searchbyprojection()$. The function $trackLocalMap()$ called by $track()$ also performs feature matching in the local map (subset of keyframes from the complete map). The $track()$ functions at the end, checks if a new keyframe is necessary, if yes creates a new keyframe. Additionally, whenever the map is added with keyframes or mappoints, a $poseOptimization()$ is performed using the g2O library.

System and Timing challenges:

The performance of the tracking thread is crucial in maintaining the accuracy and overall functioning of the system. If the tracking thread processes keyframes slower, it can lead to a drop in the keyframes throughout the system. It has been noted in Sofiya et al [1]. that the effect of dropping keyframe in regions with sparse feature density or fast camera movements can lead to the system calling the $relocalization()$ function. This function stops the other threads and performs a place recognition throughout the keyframe database to find matches and relocalize. This action is noted to be computational expensive leading and possibly leading a failure of the system [1]. As per Sofiya et. al [1] tracking on an average requires 17 ms when running on the KITTI dataset and images are streamed at 10 frames per second.

3.2 Local Mapping Thread:

Any new keyframe processed by tracking is added to the local mapping threads keyframe queue. The local mapping thread checks and updates all the nodes and connections of the added keyframe.

As mentioned in the background section, any addition to the graph requires a optimization and updation check to make sure that the graph is as optimum as possible. Then spurious and redundancies are removed based on heuristic criteria for both, MapPoint and keyframe. New mappoints are added to the graph based on the new keyframe added and the existing covisible keyframes. At the end of the local mapping thread Bundle adjustment is performed on the graph to adjust the estimates where ever necessary.

Code Structure Details:

The local-thread spawned in the tracking phase, creates a local mapping object. This object has a function $run()$, which is a while true loop, that runs all the local-mapping tasks. Firstly, the function $SetAcceptKeyFrames()$ that informs the tracking thread of the local mapping status of whether it is processing keyframes or not. Next the local mapping thread maintains a queue to maintain incoming keyframes which are added by tracking thread. The next function call is $ProcessNewKeyFrame()$. This function processes the incoming keyframes and inserts them into the existing map. Any addition of keyframes cause all the links of the covisibility graph to the updated. $MappointCulling()$ is the next function that is executed which is responsible to remove mappoints that are redundant or mappoints which are considered outliers. The $createNewMappoint()$ function is then executed that triangulates matches on the current keyframes. This is done with the help of querying for the covisible keyframes to the current frame and checking for matches. Then the $SearchInNeighbors()$ function is called that checks and updates connections in the covisibility graph. The $keyframeCulling()$ function is then called on the current keyframe (first in the queue). The $keyframeCulling()$ function, checks for redundancy of a keyframe, and marks a keyframe bad. Finally, $localbundleAdjustment()$ is

performed to optimize the local map, followed by adding the current keyframe to the loop closing threads queue.

System and Timing challenges:

The effect of slower processing of local mapping has indirect effect on the system. Both tracking and loop closing, would end up searching larger maps, which is a computationally expensive task, which in turn could cause the tracking thread to get slower that as seen above could end leading to relocalization. As per Sofiya et. al [1] local mapping thread on an average requires a 103 ms when running on the KITTI dataset and images are streamed at 10 frames per second.

3.3 Loop Closing Thread:

The loop closing thread performs two task loop detection and loop closure. Newly added keyframes to system that start from tracking pass through local mapping for adding in map and finally checked for possible loops i.e. checking for previously visited places. This is done by iterating through the covisibility graph. Every keyframe in the covisibility graph is matched with the incoming keyframe, followed my periodically checking for keyframe in the atlas. The ORB-SLAM-3 system also maintains a keyframe database where periodically a query is passed, for detecting matches with every keyframe every added. If a loop is finally detected and based on the methods mentioned in the background section the loop is closed.

Code Structure details:

The loop closing thread performs loop detection and correction if a drift occurs in a loop. Similar to the local mapping, the system thread creates the loop closing object. The loop closing function $run()$ is passed as an argument to the newly spawned thread loop closing. The loop closing thread performs detection of similarities of the complete graph and the atlas based on the newly added keyframe. This is done by calling the function $NewDetectCommonRegions()$. The function

validates the prospective matches, and if it finds two different maps having similarities it will merge them into one and performs a complete bundle adjustment. The $correctLoop()$ function performs updation of all connected keyframes and mappoints based on the detected loop. Based on the accumulated drift it also performs optimization on the essential graph followed by running the $globalbundleadjustment()$ initiating bundle adjustment on all the components of the graph.

System and Timing challenges:

The loop closing thread, if detects a loop, locks the whole map data structure, which means connected keyframes and mappoints are locked. This locking of the whole map is done to perform loop closure, which distributes offset corrections throughout the graph. However, the mapping of the whole graph results in performance drop on all the threads. According to Sofiya et. al [1], the loop closing thread requires approximately 202 ms when images are streamed at 10 frames per seconds.

3.4 Experimental Setup

The experiments performed in the following sections was based on the following setup,

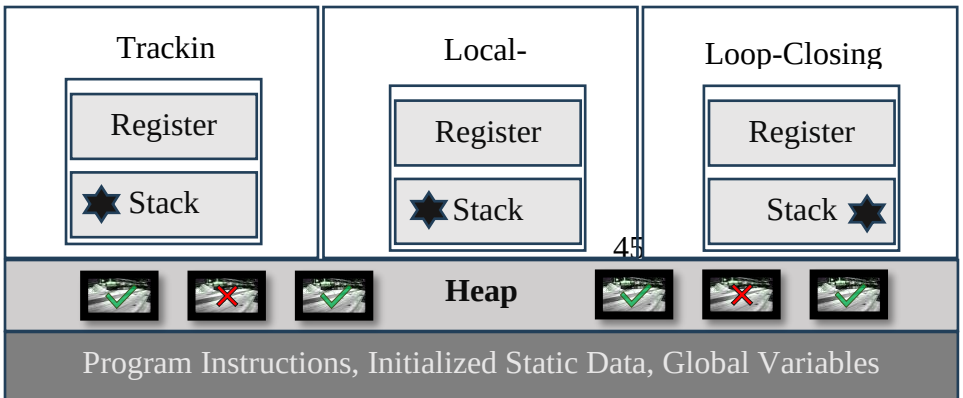
CPU: i5-8300H CPU @2.30GHz

RAM: 32 GB

System Type: Ubuntu 22.04.3 LTS, x64

4. [Joint work] Automatic garbage collection of keyframes and MapPoints

The figure below describes a conceptual view of the memory layout for the ORB-SLAM-3 system.



Each thread, tracking, local mapping, and loop closing are represented with their own registers and stack. We have a common heap followed by the space in memory for program instructions and uninitialized static data and global variables. The images in the heap show in figure are supposed to represent keyframes (keyframes are not images; they are features from the images as mentioned in previous sections) and we could also assume that mappoints to be present in the heap. The stars on the stacks each stack is supposed to represent references to the heap allocations. The green ticks on keyframes are supposed to represent good heap allocations allowed to live and the ones crossed in red are supposed to be bad heap allocations required to be culled. In the following figures, we will be looking at the figure shown above; however, we will only represent the crossed ones as they are the topic of discussion for deletion.

Now we go over each deletion scheme, which begins with an explanation of the experiment performed followed by the results in the form of a experimental outcomes and issue that need addressing.

4.1 1st Deletion Attempt: Direct deletion and collecting reference and deletion

Experiment: Direct Deletion of the heap allocation in the local mapping thread, specifically in the functions, $LocalMapping::MapPointCulling()$ or $LocalMapping::KeyframeCulling()$, when the heap objects are marked as bad. Another minor attempt involved in saving the marked

bad references and deleting them in the next iteration. This also results in a segmentation fault as we have local copies of reference that may not be possible to forceable remove from containers.

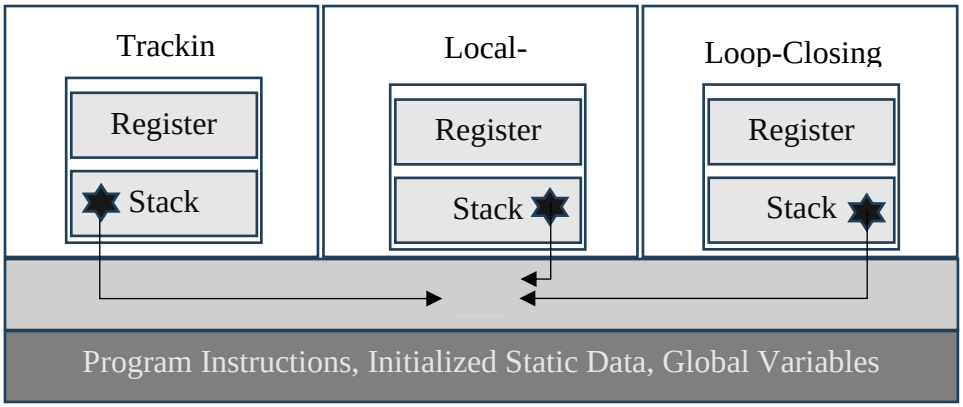


Figure 10: Experimental result of Direct Deletion

Result: Failure, as there are references to the heap allocations present, that results in any of the stack objects dereferencing the heap allocation resulting in a segmentation fault.

Experiment outcomes: This exercise does indeed confirm that that keyframes and mappoints references are being used by the other threads.

Issues that need addressing: No deletion of heap allocations.

4.2 2nd Deletion Attempt: Using Shared Pointers

Experiment: In this experiment, instead of using raw pointers, we utilize shared pointers. Shared pointers, as mentioned in the background section, works on the principle of reference counting. This attempt involves, replacing the left-hand side of all the new instances of keyframes and mappoints with, shared pointers with the same types.

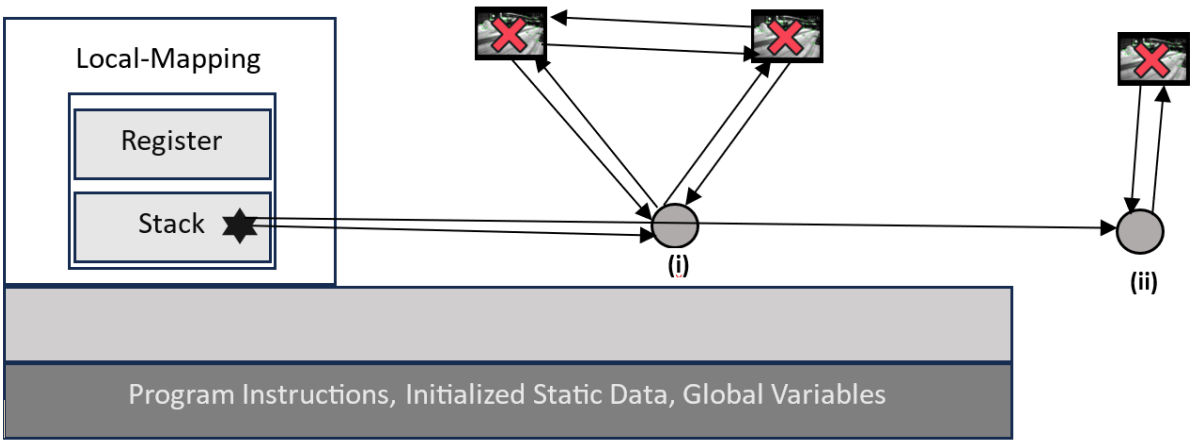


Figure 11: Cyclic References Due to Shared Pointers

Result: Failure to delete any heap allocation. The nature of heap allocations such as mappoints and keyframes in ORB-SLAM-3 are cyclic. This cyclicity results in the reference count never reducing to zero, and hence none of the heap allocations are deleted leading to memory leaks. The figure shows one of the possible scenarios for the cyclicity to arise. For simplicity the illustration below refers one thread (local mapping) pointing to the possible scenarios. i) we have a reference of a MapPoint, and since this mappoint is common between the two bad keyframes, we could have all the following unordered pair of relations denoted by arrows. So, none of the pointers ever go out of scope.

Experiment Outcomes: This experiment shows that shared pointer implementation of ORB-SLAM-3 is possible, however it should be noted that there is no deletion.

Issues that need addressing: The lack of deletion of heap allocations using shared pointers indicate that reference counting exposes the cyclicity in ORB-SLAM-3 codebase.

4.3 3rd Deletion Attempt: Using custom reference counting scheme.

Experiment: In this attempt we, perform a simple reference counting strategy as described in the above sections. Due to the scale of the ORB-SLAM-3 system, we performed the reference

counting of keyframes and mappoints incrementally. By incrementally, we mean we identified the containers into which these heap allocations are added such as the following,

- $mvpOrderedConnectedKeyframes$,
- $mConnectedKeyFrameWeights$
- $mObservations$.

We perform an increment, whenever a keyframe or a mappoint is used and perform a deletion whenever, this count goes to zero. The figure below illustrates, how increment and decrement is performed in a local scope. For simplicity we only shown this for one thread, however, this mechanism can be extended for all the threads. We also have a simplified program counter as shown in the figure for thread Local mapping. Furthermore, we have the Functions A, B and C that need to be executed. So, whenever any container of MapPoint or keyframe pointer is utilized, we perform an increment shown in green plus in the illustration and perform a decrement before the scope ends shown in blue minus sign.

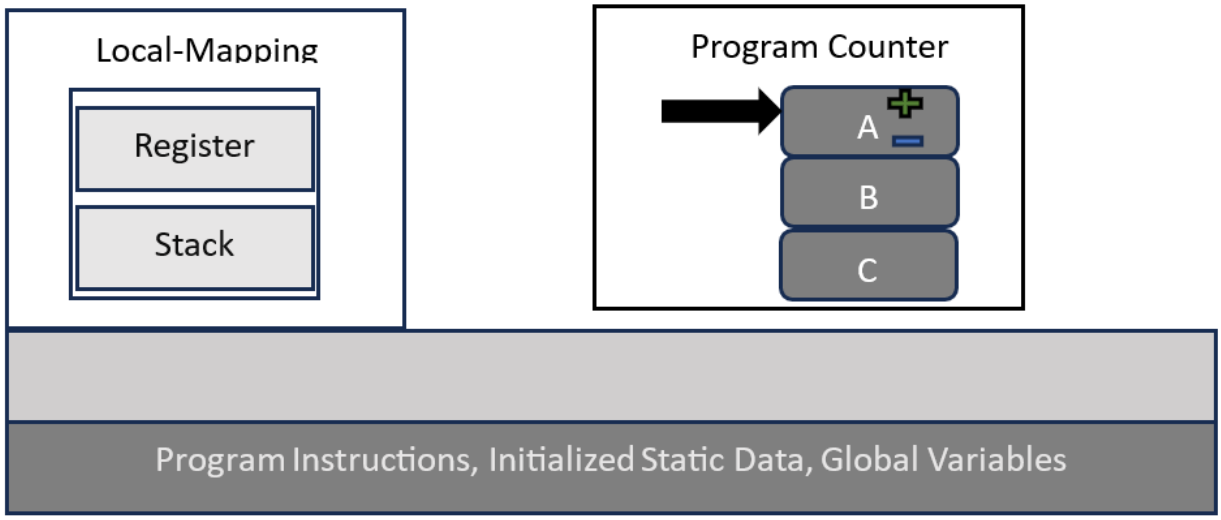


Figure 12: Custom reference counting for subrountines

Through our experiment we have identified that reference counting the containers $mvpOrderedConnectedKeyframes$, $mObservations$ was sufficient for performing a deletion

of keyframes that did not result in a memory issues. The ORB-SLAM-3 system marks keyframes as bad in the keyframe culling loop in the local mapping thread. We also observed from the previous experiments that, the reference count of keyframes do not reduce to zero at the end of the keyframe culling function. The possible explanation for this behaviour can be attributed to live references of the marked bad keyframes in use by the other threads. To tackle this problem, we collect the references in a set (choice of set was chosen due to possibility of repeated keyframes references being collected). We loop through this container at the end of the keyframe culling function every iteration, and check for a zero count of the keyframes. We delete the keyframe at zero. For mappoints we follow a similar strategy of reference counting. We collect the mappoint references that are marked bad in a set. At the end of the mappoint culling function in the local mapping thread, we perform the deletion of mappoints at reference count of zero.

Dealing with cyclicity: As noted in the shared pointer scheme of reference counting, we need to break the cycle manually. We do this by adding the deletion

Result: The experiment results in successful deletion of mappoints and keyframes. We show the results of the deletion on the EuRoC micro aerial vehicle datasets and the experiments were conducted for ten iterations on each dataset and compare them to Vanilla ORB SLAM 3.

Keyframe Deletion Statistics:

The statistics below is a cumulative average across eleven EuRoC micro aerial vehicle datasets

Table 1: Keyframe Deletion Statistics on the Euroc dataset

Number of Keyframes marked to be deleted	94
Number of Keyframes deleted	65
Memory saved in percentage	**69%**
Average Execution time	125 Seconds

Approximate size of each Keyframe	4720 Bytes
Raw Memory marked for deletion	0.44 MB
Raw Memory deleted	0.30 MB
Raw memory saved	0.14 MB

MapPoint Deletion Statistics:

The statistics below is a cumulative average across eleven EuRoC micro aerial vehicle dataset

Table 2: Mappoint Statistics on the Euroc dataset

Number of MapPoints marked to be deleted	13,183
Number of MapPoints deleted	12,833
Memory saved in percentage	**97%**
Average Execution time	125 Seconds
Approximate size of each Keyframe	752 Bytes
Raw memory marked for deletion	9.91 MB
Raw memory deleted	9.65 MB
Raw memory saved	0.26 MB

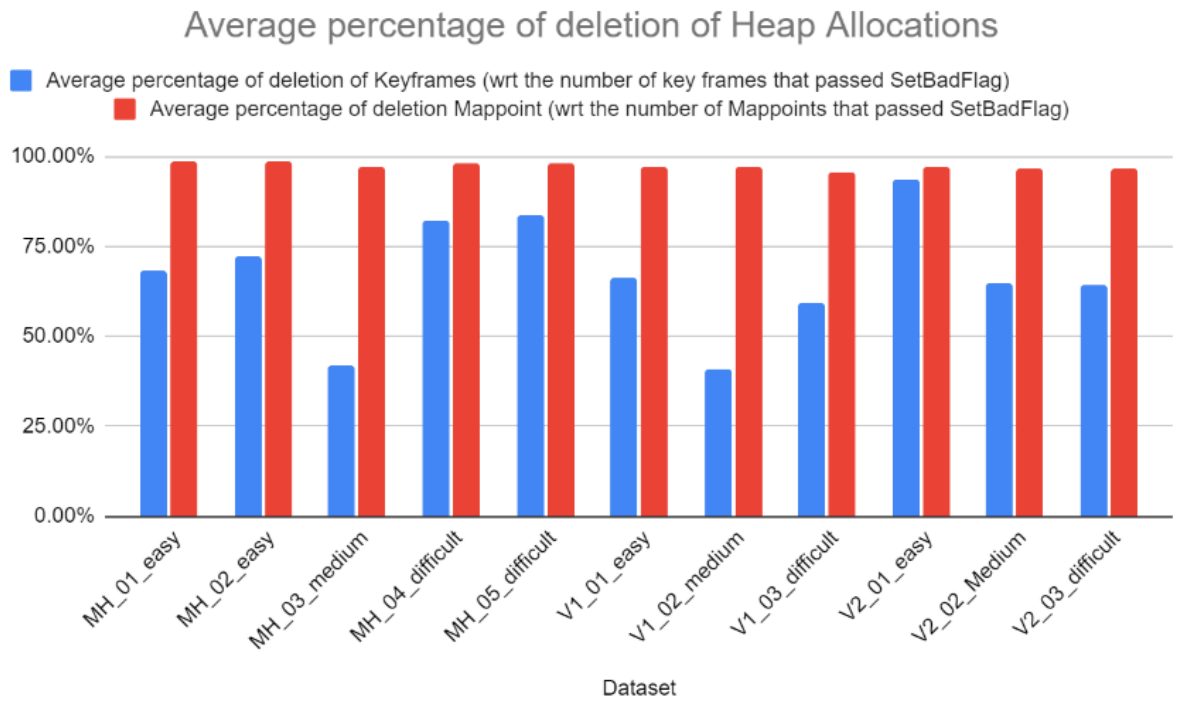


Figure 13: Deletion percentage using mutexs

The above is a summary of the deletion statistic.

Execution time statistics and overhead:

The statistics below is a cumulative average across eleven EuRoC micro aerial vehicle dataset

Local Mapping Thread

Table 3: Local Mapping execution statistics on the euroc data set

Vanilla ORB-SLAM-3	~ 228 ms
Reference counted ORB-SLAM-3	~ 255 ms
Drop-in execution time due to reference counting	~ 11.18% ~ 27ms slower

Tracking Thread

Table 4: Tracking execution statistics on the euroc data set

Tracking thread Vanilla ORB-SLAM-3	~ 19.88 ms
Tracking thread Reference counted ORB-SLAM-3	~21.15 ms
Drop-in execution time due to reference counting	~6.19% ~ 1.27ms slower

Processing Overhead:

Keyframes

Table 5: Processing overhead in terms of keyframes

Vanilla ORB-SLAM-3	~ 406
Reference counted ORB-SLAM-3	~ 376
Percentage drop in keyframe processing	~7.6% ~ 30 fewer keyframes processed

Mappoints

Table 6: Processing overhead interms of mappoints

Vanilla ORB-SLAM-3	~ 15170
Reference counted ORB-SLAM-3	~ 13189
Percentage drop in Mappoints processing	~13.96% ~ 1980 Mappoints unprocessed

Experiment Outcomes: This experiment provides a successful deletion of heap allocations in the ORB-SLAM-3 codebase. Reference counting is used for deletion with mutex locks for maintaining invariance of the integer responsible for reference counting. The statistics of the

deletion results in a processing overhead of 11% and 6% increase for local mapping and tracking thread respectively. We also process 30 fewer keyframes when compared to vanilla ORB-SLAM-3 which could be attributed to the use of mutexes for maintaining sequential consistency.

Issues that need addressing: The overhead due to addition of mutexes which results in processing of fewer keyframes could result in backlog of keyframes in the tracking thread, which could in turn result in relocalization call. As stated in [1], relocalization incurs a computational overhead that could result in the SLAM system crashing.

4.4 4th Deletion Attempt: Deletion using compare and swap.

Experiment: In the background section, we explained lock-less synchronization method compare and swap. This experiment is supposed to be an optimization over the regular reference counting framework. In this experiment we replace all the reference counting mechanism that employs mutex locks, and instead we use $compare\_exchange\_strong$ a C++ construct discussed in detail in the above sections. The result of this experiment is shown below.

Keyframe Deletion Statistics:

The statistics below is a cumulative average across eleven EuRoC micro aerial vehicle dataset

Table 7:Keyframes statistics for compare and swap on the Euroc dataset

Number of Keyframes marked to be deleted	98
Number of Keyframes deleted	70
Memory saved in percentage	**71%**
Average Execution time of the program	125 Seconds
Approximate size of each Keyframe	4720 Bytes
Raw Memory marked for deletion	0.46 MB
Raw Memory deleted	0.33MB

Raw memory saved	0.13 MB

MapPoint Deletion Statistics:

The statistics below is a cumulative average across eleven EuRoC micro aerial vehicle dataset

Table 8: Mappoint deletion Statistics for compare and swap on the Euroc dataset

Number of MapPoints marked to be deleted	13305
Number of mappoints deleted	12939
Memory saved in percentage	**97%**
Average Execution time	125 Seconds
Approximate size of each Keyframe	752 Bytes
Raw memory marked for deletion	10.00 MB
Raw memory deleted	9.73 MB
Raw memory saved	0.27 MB

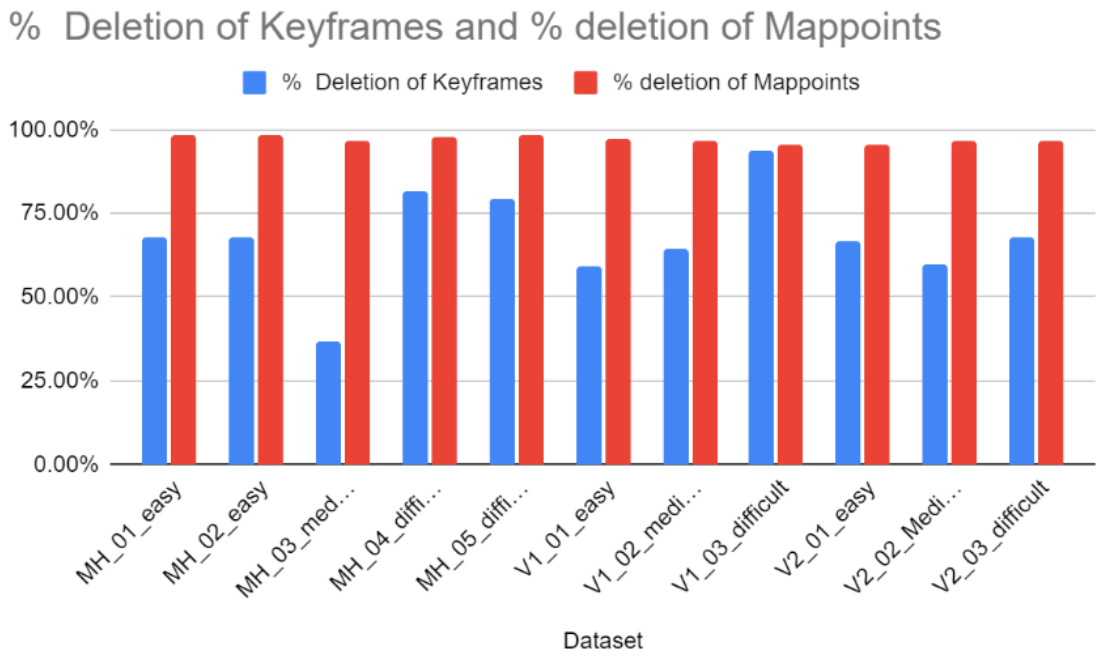


Figure 14:Deletion percentage for Compare and Swap

The above is a summary of the deletion statistic.

Execution time statistics and overhead:

The statistics below is a cumulative average across eleven EuRoC micro aerial vehicle dataset

Local Mapping Thread

Table 9: Local Mapping execution statistics, Compare and Swap on the Euroc dataset

Vanilla ORB-SLAM-3	~ 228 ms
Compare and Swap Reference counted ORB-SLAM-3	~ 247 ms
Drop-in execution time due to reference counting	~ 8% ~ 19 ms slowers

Tracking Thread

Table 10: Tracking execution statistics, Compare and Swap on the Euroc dataset

Vanilla ORB-SLAM-3	~ 19.88 ms
Compare and Swap Reference counted ORB-SLAM-3	~ 20.68 ms
Drop in execution time due to reference counting	~ 3.94% ~0.8 ms slower

Processing Overhead:

Keyframes

Table 11: Keyframe overhead, Compare and Swap on the Euroc dataset

Vanilla ORB-SLAM-3	~ 406
Compare and Swap Reference counted ORB-SLAM-3	~ 383
Percentage drop in keyframe processing	~5.83% ~ 23 fewer keyframes processed

Mappoints

Table 12: Mappoint overhead, Compare and Swap on the Euroc dataset

Vanilla ORB-SLAM-3 processed	~ 15170
Compare and Swap Reference counted ORB-SLAM-3 processed	~ 13305
Percentage drop in Mappoints processing	~13.09% ~ 1865 Mappoints unprocessed

Experiment Outcomes: This experiment provides another successful deletion of heap allocations in the ORB-SLAM-3 codebase. Reference counting is used for deletion with compare and swap

loops for maintaining invariance of the integer responsible for reference counting. The statistics of the deletion results in a processing overhead of 8% and 3% increase for local mapping and tracking thread respectively. We also process 23 fewer keyframes when compared to vanilla ORB-SLAM-3 which could be attributed to the use of compare and swap instructions for maintaining sequential consistency. When compared with the mutex based reference counting, compare and swap performs slightly better because, mutex based locks according to the C++ standards, put a thread to sleep, if it fails to acquire a lock. Since the critical section for reference counting is one instruction i.e either increment or decrement, compare and swap on the other hand works similar to a spin lock, which means it keeps trying to increment. Thus, it could be noted that the mechanism required to wake a thread from sleep in this experiment could be costlier than continuedly check for synchronized reads.

Issues that need addressing: The possible presence of memory leak because of keyframes and mappoints due to subset reference counting must be addressed, as the current scheme reference counts containers that do not result in segmentation fault on the Euroc data set. A thorough reference counting for every container where keyframes and mappoints are added would be necessary for not only guaranteeing safety (potential of deleted references being dereferenced) but also eliminating the possibility of memory leaks due to keyframes and mappoints.

4.5 Comparison of both deletion scheme with vanilla ORB-SLAM-3:

In this section we summarize the results from the two deletion strategies

Deletion Statistics:

The below figure gives Keyframes deletion statistics.

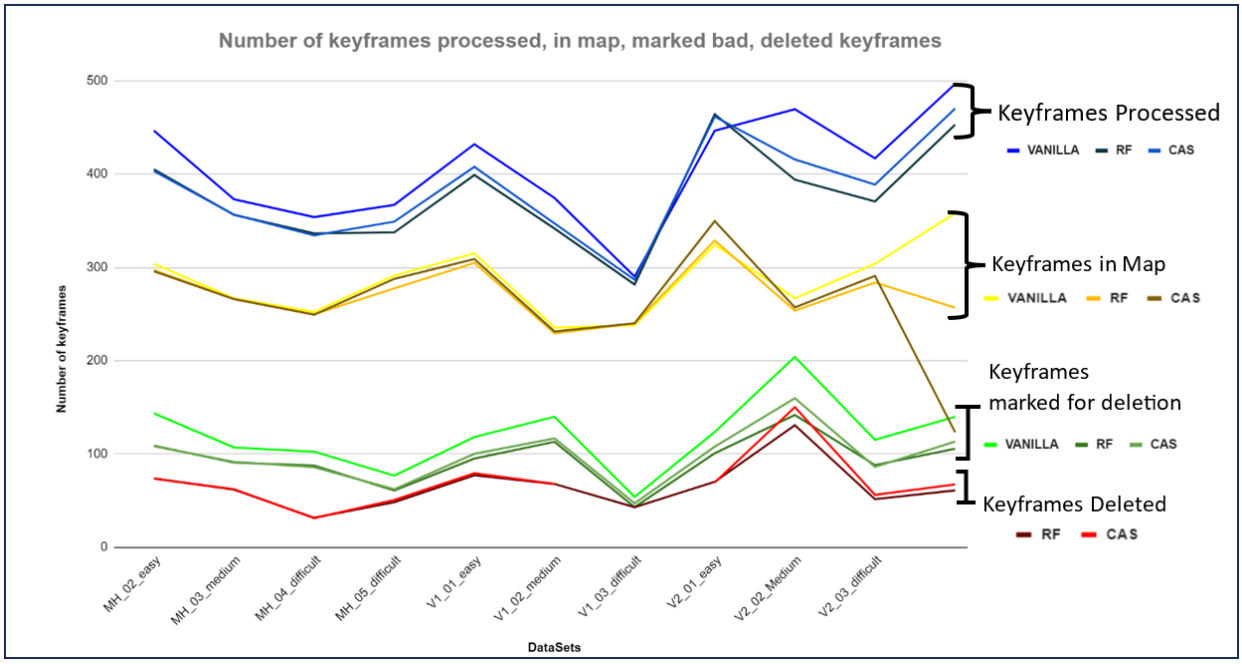


Figure 15:Keyframe Statistics

The trend lines in the figure represents the summary of keyframe statistics that was calculated using the tables show in the appendix section. The number of keyframes processed, keyframes retained in map, keyframes marked for deletion and the actual deletion of keyframes are shown together.

The Blue themed trendline shows the processed keyframes with vanilla ORB_SLAM-3 leading the trend with 7% greater keyframes when compared to the mutex based reference counting (RF) and 5.8% greater keyframes processed as compared to the atomic compare and swap operation (CAS). There is no significant difference in the processing of the RF and CAS implementation. The figure provides a summary of comparisons.

The yellow themed trendline represents the keyframes in the map. We note a slight decrease in the number of keyframes in the map as shown in the figure, where we RF and CAS have approximately 2% fewer keyframes than the vanilla ORB-SLAM -3 implementation.

The green themed trendline also follows a similar trajectory where RF and CAS have fewer keyframes than vanilla.

Finally, the red themed lines represent deletion with CAS having approximately 8% greater deletion compared to RF.

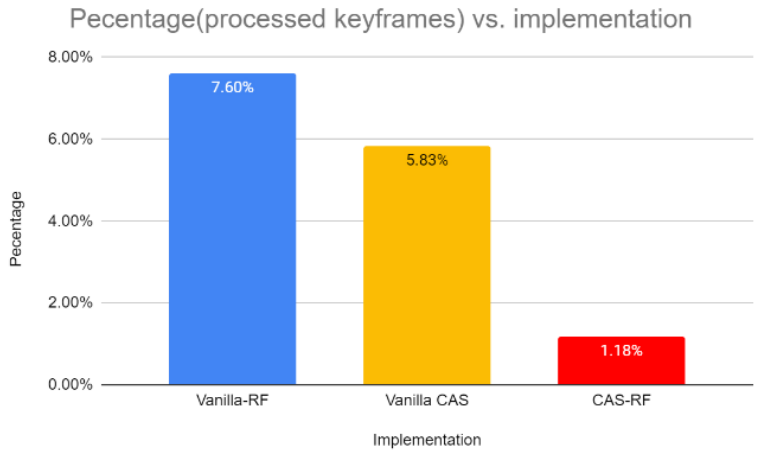


Figure 16: Keyframe Deletion vs Implementation

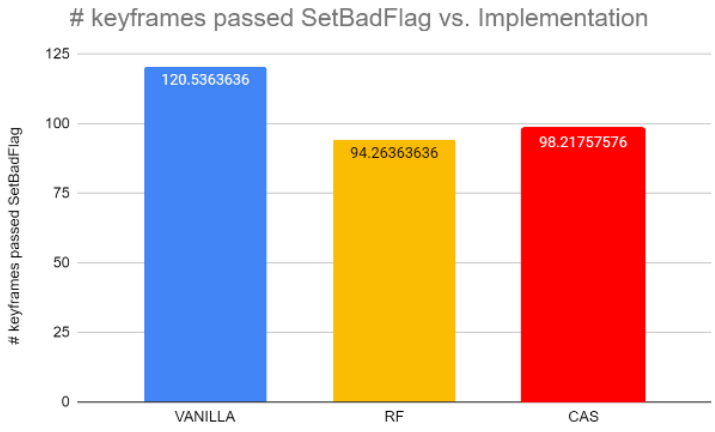


Figure 17: Keyframes marked bad in all implementations

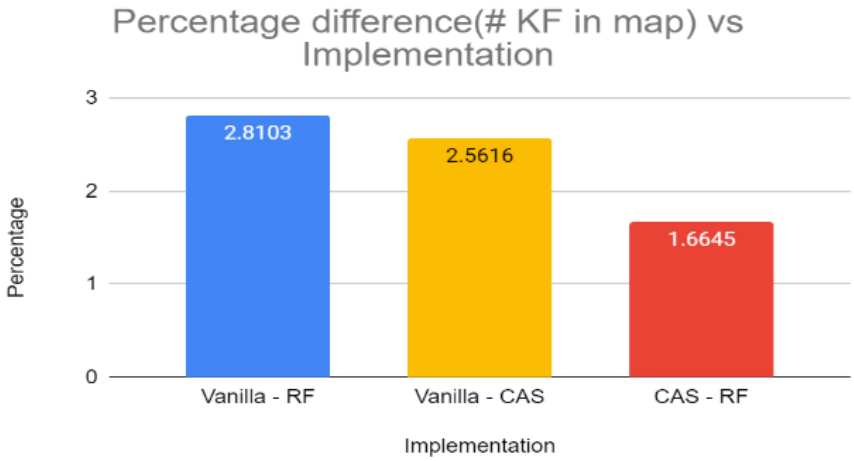


Figure 18: Percentage difference among deletions

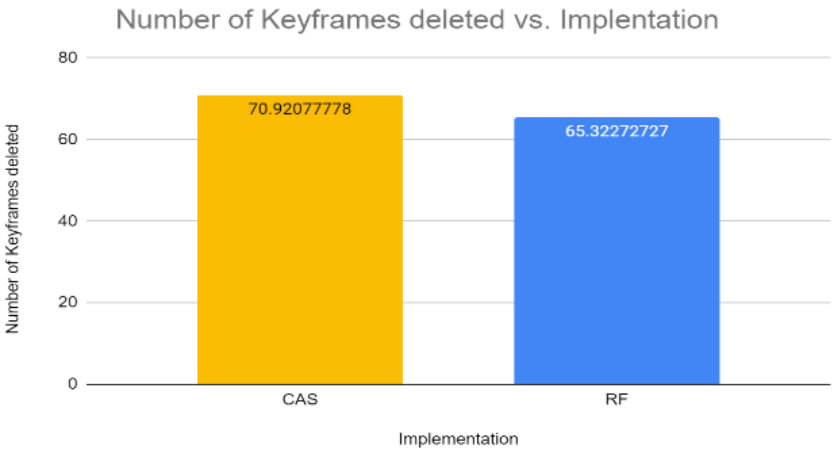


Figure 19: Total Number of keyframes deleted

The line graph shown in figure represents a summary like the keyframe statistic, with an of deletion shown in a separate trendline graph for better visibility. From the graph we can see that vanilla ORB_SLAM-3 allocates approximately 13% greater mappoints as compared to the other counterparts. The mappoints in map are similar in the three implementations with 4% to 6% greater mappoints in map in the Vanilla ORB-SLAM-3. The deletion statistics for mappoints among keyframes and mappoints are identical with negligible difference. The bar charts are accompanied to provide easier statistic difference among the trendlines.

The below figure gives Mappoints deletion statistics.

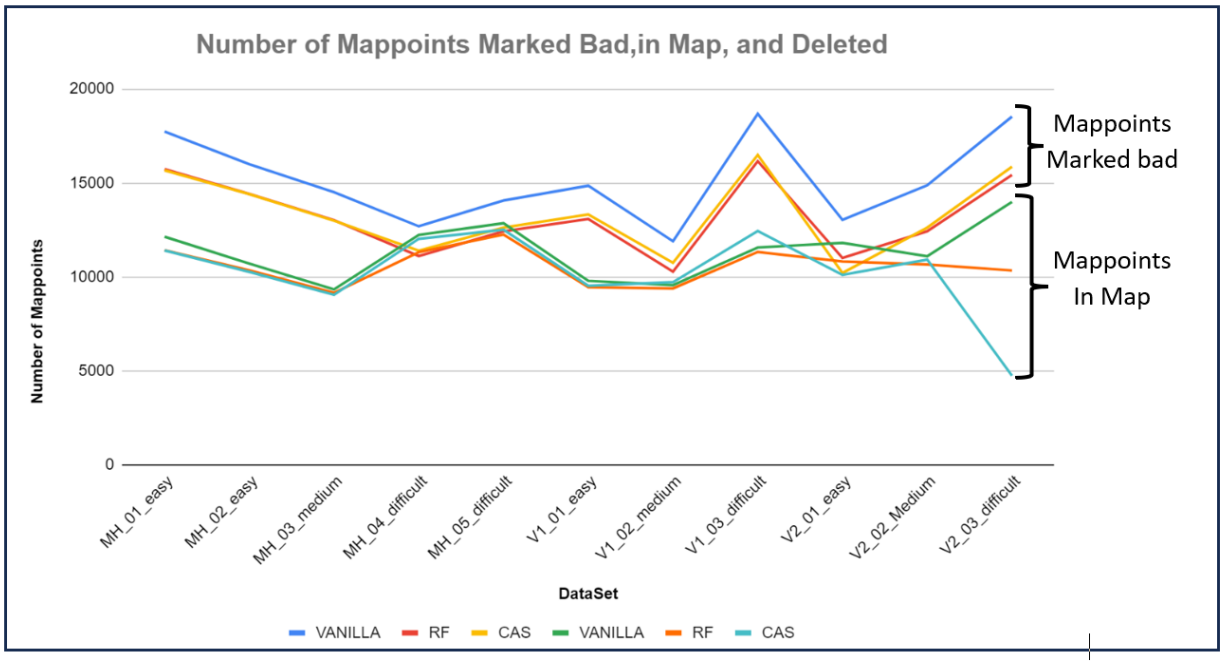


Figure 20: MapPoint Statistics

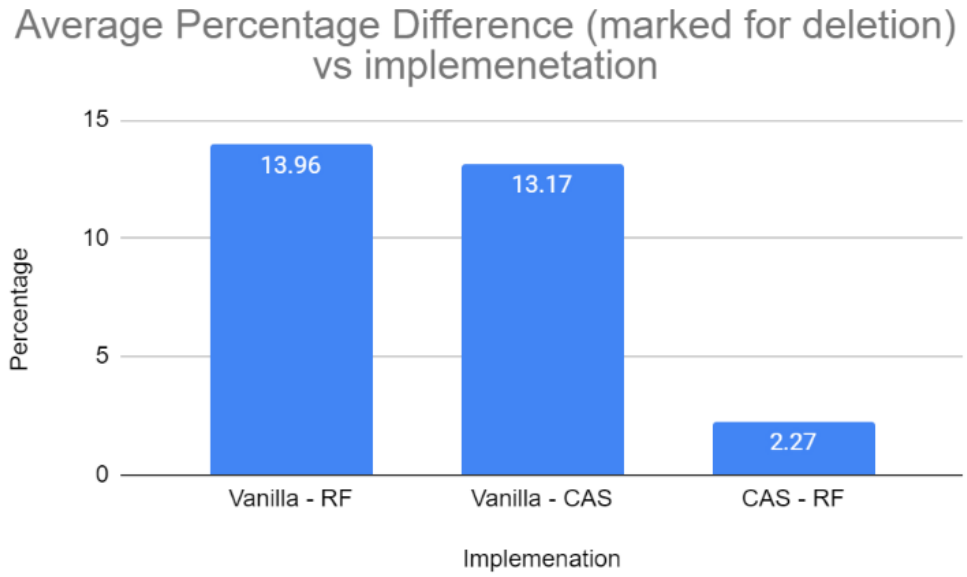


Figure 19: Percentage difference of mappoints marked for deletion in difference implementations

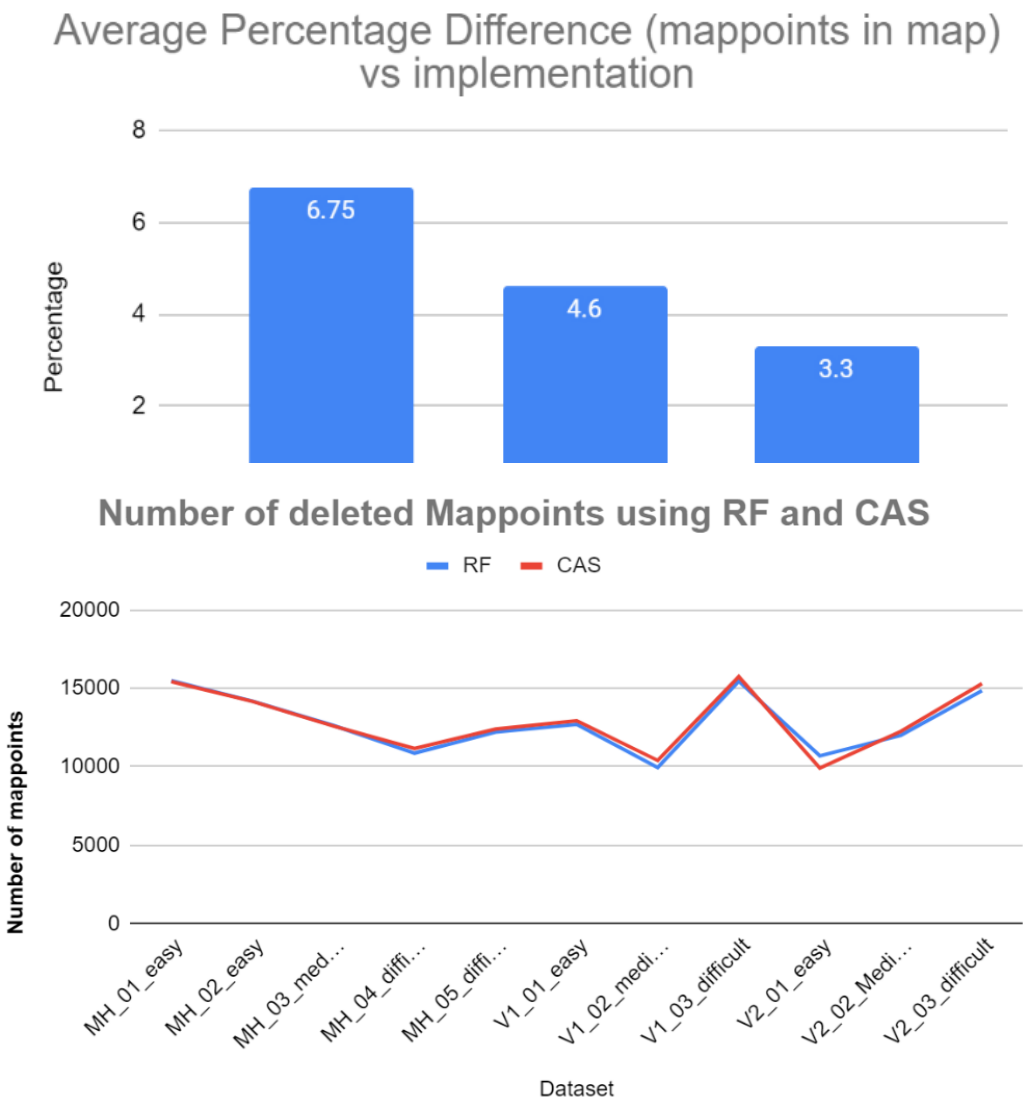


Figure 21:Mappoint deletion in different scheme

Execution Statistics and Overhead statistics:

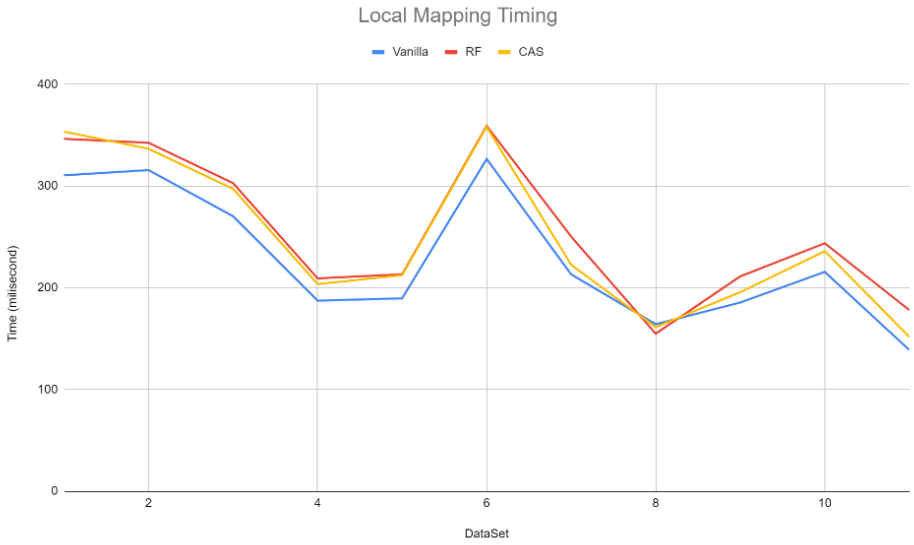


Figure 22: Local Mapping Execution statistics

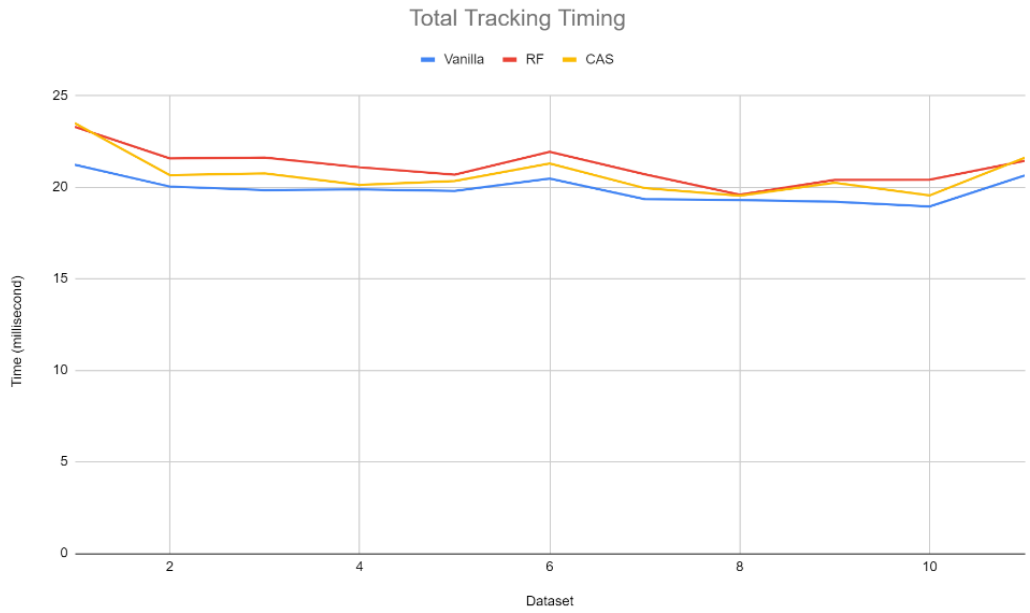


Figure 23: Tracking Thread Execution Statistics

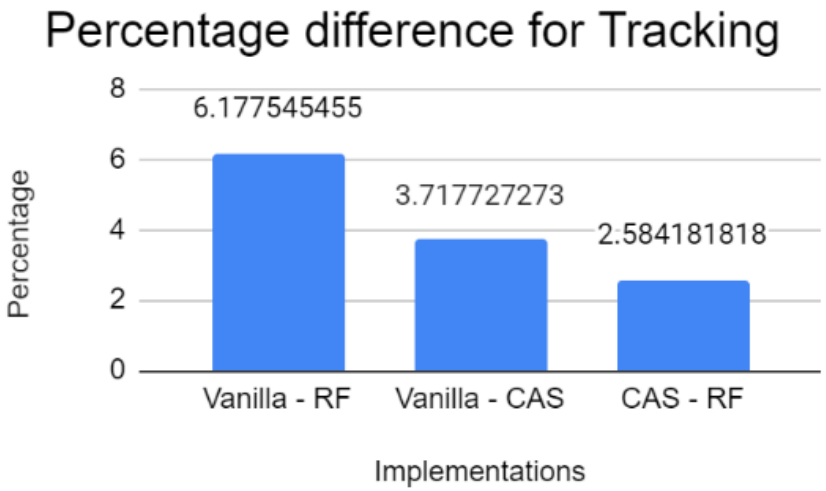


Figure 24: Percentage difference of local mapping timing

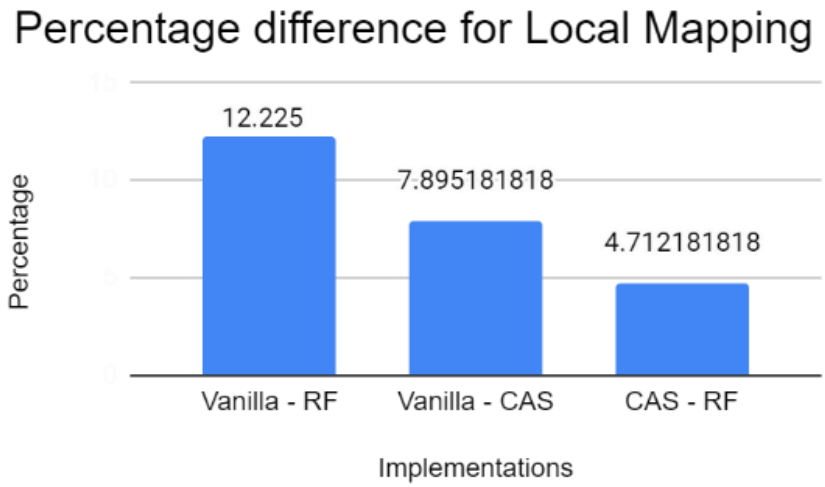


Figure 25: Percentage difference of Execution statistics for the tracking thread

The summary of the execution statistics show that both tracking and local mapping threads run faster in vanilla ORB-SLAM 3. We see a 12% drop in execution timing in RF implementation as compared to vanilla orb slam and 7% drop in CAS execution timing. CAS however is faster than RF statistically, by 4% faster execution in the local mapping thread. The tracking behavior among the different implementations follow an identical trend with only a 6% overhead for RF

compared to vanilla and 3% compared to CAS. RF and CAS differ by 2% faster execution by the CAS implementation.

5. Experiments for improving ORB SLAM-3 Keyframe Culling

In this section we go over an experiment with the following hypothesis,

The computation necessary to compute redundancy of a keyframe in the keyframe culling function of the local mapping thread can be replaced with an alternative mechanism that is possibly more efficient.

Before exploring the mechanism for the above-mentioned hypothesis, let us first examine the current marking ($setBadFlag$) scheme, its time complexity. Let us consider a keyframe $\boldsymbol{k}$ to be redundant, then the following expression evaluates to *true*.

$$nRedundantObservations > redundant\_th * nMPs$$

Where,

$nRedundantObservations$: integer count of redundant mappoints in

$redundant_{th}$: $\left.\begin{matrix} 0.9: \ if\ monocular \\ 0.5: \ all\ other\ camera\ model \end{matrix}\right\}$

$nMPs$: number of mappoints present in keyframe k

Both $nMPs$ and $redundant_{th}$ do not require additional computation.

The figure below illustrates when a count for $nRedundantObservations$ is incremented,

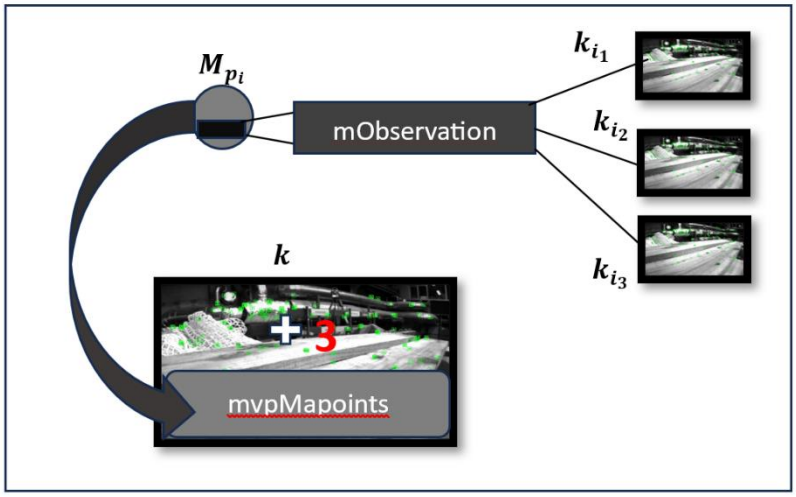


Figure 26:Incrementing reference counts

Let mappoint M_{p_i}be added to a keyframe k. If M_{p_i} is present at the same scale or at a finer scale in the keyframes $k_{i_1}, k_{i_2}$and k_{i_3}then this mappoint M_{p_i} is increments $nRedundantObservations$ for keyframe k. The condition for mappoint redundancy is as follows,

$$\boldsymbol{scaleLevel_{i_1} <= scaleLevel + 1}$$

Where,

$scaleLevel_{i_1}$is the similarity index of keyframe k_{i_1} for the mappoint M_{p_i}

$scaleLevel$ is the similarity index of a keyframe k for the mappoint M_{p_i}

In the local mapping thread, of Vanilla ORB-SLAM-3, the loop responsible for counting redundant keyframes, employs the above-mentioned check for every keyframe in the set of covisible keyframes. Let κ_{c_i} be the set of covisible keyframes present at time t_i, and $\boldsymbol{n_{\kappa_{c_i}}}$, the number of keyframes in κ_{c_i}. Let k_i be a keyframe in κ_{c_i}, M_i be the set of mappoints in k_i and $\boldsymbol{n_{M_i}}$ be the size of this set. Let m_i be a mappoint in M_i that has a set $\boldsymbol{\pi_{m_i}}$ of connected keyframes of size $\boldsymbol{n_{\pi_{m_i}}}$. The figure and the psuedo code below illustrate the computation or work required to

complete one iteration of the keyframe culling function in the local mapping thread. the complexity of the keyframe culling loop can be given by $\boldsymbol{O}(\boldsymbol{n}_{\boldsymbol{\kappa}_{c_i}} * \boldsymbol{n}_{\boldsymbol{M}_i} * \boldsymbol{n}_{\boldsymbol{\pi}_{m_i}})$.

Alternatively, we can compute $nRedundantObservations$ while adding mappoints to a keyframe. Let us understand the alternative mechanism by looking at an example that goes over all the steps required for marking a keyframe as bad. We look at this example in the reverse order, beginning with a keyframe that is marked bad, and work our way through incrementing $nRedundantObservations$ and decrementing *nRedundantObservations*.

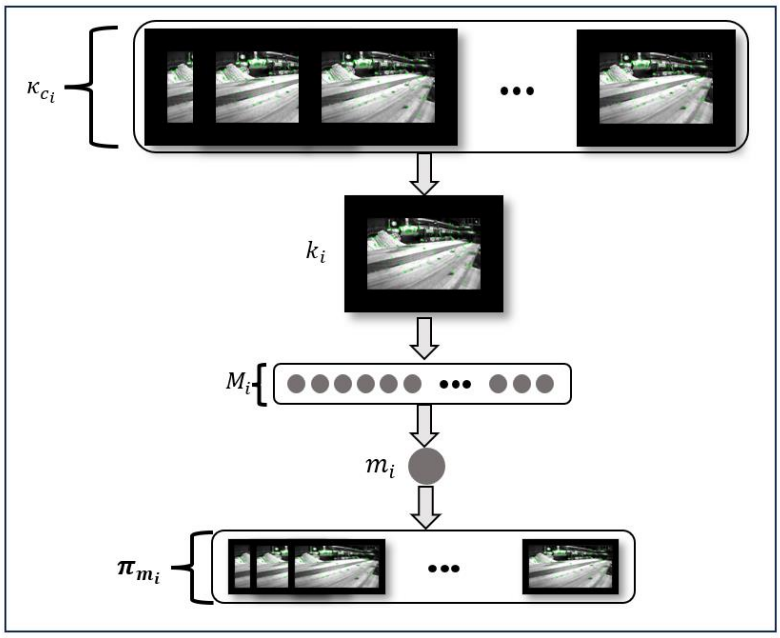


Figure 27: multi-loop structure, where black frame represents keyframes and the gray dots represent mappoints.

KeyframeCulling()

```
⋮
for every k_i ∈ κ_{c_i} do
    nRedundantObservations ← 0
    for every m_i ∈ M_i
        for every k_i ∈ π_{m_i}
            If m_i present in greater than or equal to three keyframes
                nRedundantObservations++
```

end if
end for
end for
end for

Let us assume a keyframe k, that is marked for deletion. This means that keyframe k evaluates $nRedundantObservations > redundant\_th * nMPs$ to true as a result of $nRedundantObservations$ going from an initialized value of zero to passing the threshold number of redundant mappoints. There are two ways to mutate $nRedundantObservations$,

- We increment $nRedundantObservations$ when we find at least three keyframes with the same mappoint present in keyframe k at either the same scale or at a finer scale.
- We decrement $nRedundantObservations$, when a mappoint is removed from a keyframe that was connected to three or more keyframe.

To maintain the correctness of this implementation, we add a data structure as follows,

- $Redundant\_mappoint\_connections$: A container (hash map) the size of $mvpmappoints$(container with all mappoints in a keyframe), that maintains a list of connections made with a mappoint.

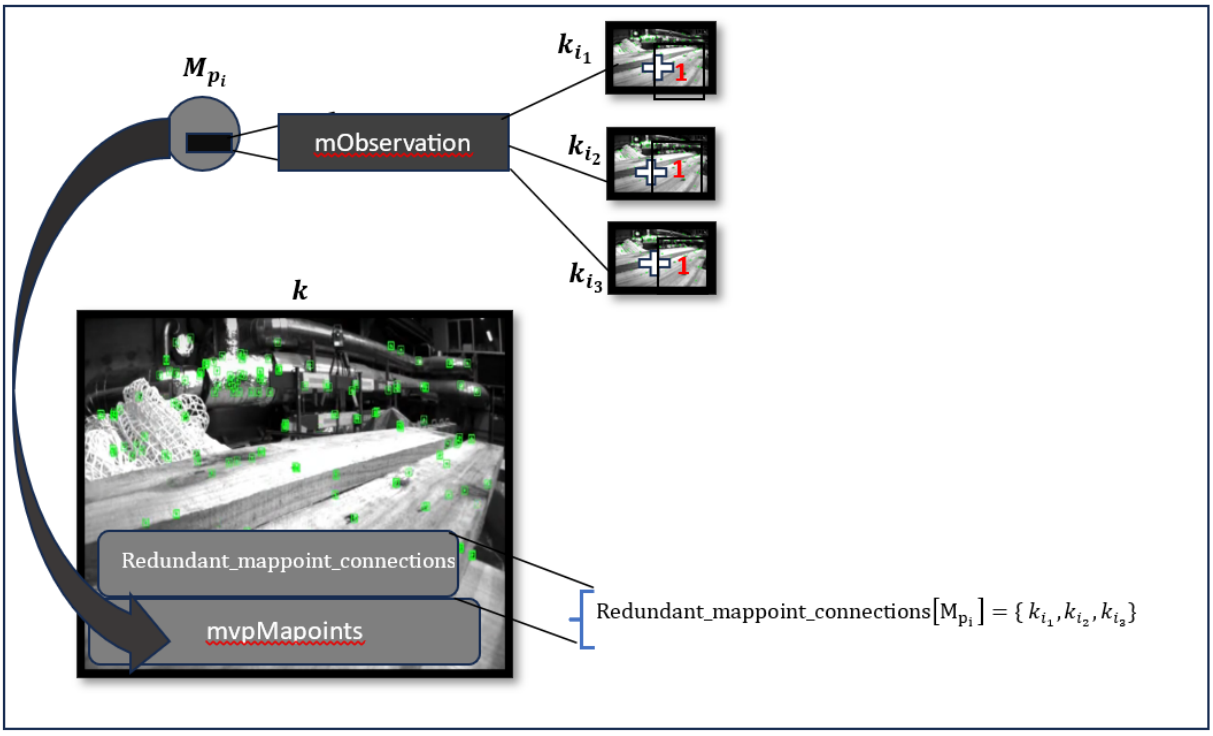


Figure 28:Alternative incrementing of reference count

The above figure shows an example case of the **increment** operation due to the addition of a mapppoint $\boldsymbol{M_{p_i}}$ to the keyframe $\boldsymbol{k}$. In this example, mappoint $\boldsymbol{M_{p_i}}$ was already connected to keyframes $\boldsymbol{k_{i_1}}$, $\boldsymbol{k_{i_2}}$, and $\boldsymbol{k_{i_2}}$. We iterate through the $\pi_{\mathrm{M_{p_i}}}$ (container with connected keyframes to $\boldsymbol{M_{p_i}}$) i.e connected keyframes and check similarity ($scaleLevel_{i_1} <= scaleLevel + 1$). In this example, we assume the condition to be true for the keyframes k_{i_1}, k_{i_2}, and k_{i_2} and we add three connected keyframes to Redundant_mappoint_connections$[\mathrm{M_{p_i}}]$ for the current keyframe k. Since the size of Redundant_mappoint_connections$[\mathrm{M_{p_i}}]$ is three we perform an increment of $nRedundantObservations$ for keyframe k. Additionally, also add k to k_{i_1}, k_{i_2}, and $k_{i_2}{}'s$ Redundant_mappoint_connections$[\mathrm{M_{p_i}}]$ and if their size of Redundant_mappoint_connections$[\mathrm{M_{p_i}}]$ should go beyond **three**, which in this example is

true, we perform an increment of $nRedundantObservations$ for $\boldsymbol{k_{i_1}}$, $\boldsymbol{k_{i_2}}$, and $\boldsymbol{k_{i_2}}$ respectively.

The algorithm below illustrates this increment operation.

ADDMAPPOINT()

⋮

for every $\mathcal{k}_i \in \pi_{\mathrm{M_{p_i}}}$ (where, $\mathcal{k}_{\mathrm{i}}$ is a keyframe, $\pi_{\mathrm{M_{p_i}}}$ is the container with connected keyframes to $\mathrm{M_{p_i}}$)

  If $\mathrm{M_{p_i}}$ is present in $\mathcal{k}_{\mathrm{i}}$ at same scale or below ($scaleLevel_{i_1} <= scaleLevel + 1$) **then**

    Add $\mathcal{k}_i$ to $k's\ Redundant\_mappoint\_connections[\mathrm{M_{p_i}}]$

    Add k to $\mathcal{k}_i's\ Redundant\_mappoint\_connections[\mathrm{M_{p_i}}]$

    If $Redundant\_mappoint\_connections[\mathrm{M_{p_i}}]$ size greater than or equal to three **then**

      $k \rightarrow nRedundantObservations{+}{+}$

    end if

    If $\mathcal{k}_i's\ Redundant\_mappoint\_connections[\mathrm{M_{p_i}}]$ *is* size greater than or equal to three **then**

      $\mathcal{k}_i \rightarrow nRedundantObservations{+}{+}$

    end if

  end if

end for

The time complexity of this operation would be $O(\boldsymbol{n_{\pi_{\mathrm{M_{p_i}}}}})$

where,

$n_{\pi_{\mathrm{M_{p_i}}}}$ is the size of $\pi_{\mathrm{M_{p_i}}}$

$\pi_{\mathrm{M_{p_i}}}$ data structure representing the connected keyframes to mappoint $\mathrm{M_{p_i}}$.

Now we look at an example case, for **decrementing** $nRedundantObservations$ due to the removal of a mapppoint $\boldsymbol{M_{p_i}}$ form keyframe $\boldsymbol{k}$. The figure below illustrates an example of the decrement operation.

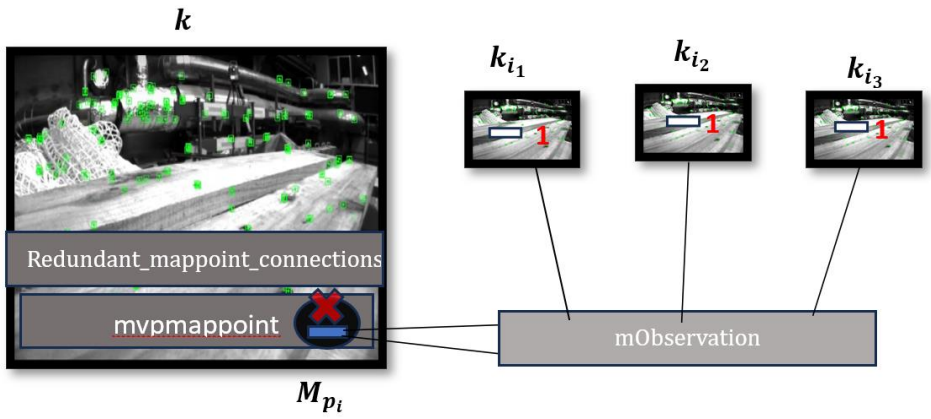


Figure 29:Decrementing reference count

Since, M_{p_i} is connected to keyframes $\boldsymbol{k_{i_1}}$, $\boldsymbol{k_{i_2}}$, and $\boldsymbol{k_{i_2}}$ i.e. connected to three or more keyframes we perform a decrement of $nRedundantObservations$. Furthermore, we iterate through $\pi_{\mathrm{M_{p_i}}}$ and remove k from its $Redundant\_mappoint\_connections[\mathrm{M_{p_i}}]$ and if the size of this container should fall below three, we perform a decrement of $nRedundantObservations$ for all such keyframes. The algorithm given below illustrates the decrement operation.

ERASEMAPPOINTMATCH(), REPLACEMAPPOINTMATCH()

⋮

If $k \rightarrow Redundant\_mappoint\_connections[M_{p_i}]$ size is greater than three
  $nRedundantObservations$- -
end if
Clear the $Redundant\_mappoint\_connections[M_{p_i}]$ *container in* k
for every $\mathcal{k}_i \in \pi_{\mathrm{M_{p_i}}}$
  If k present in $\mathcal{k}_i \rightarrow Redundant\_mappoint\_connections[M_{p_i}]$
    Remove k from $\mathcal{k}_i \rightarrow Redundant\_mappoint\_connections[M_{p_i}]$
    If *size of* $\mathcal{k}_i \rightarrow Redundant\_mappoint\_connections[M_{p_i}]$ drop below three
      $\mathcal{k}_i \rightarrow nRedundantObservations$- -
    end if
  end if
end for

The time complexity of the above functions would be $O(\boldsymbol{n_{\pi_{\mathrm{M_{p_i}}}}})$,

where,

$n_{\pi_{M_{p_i}}}$ is the size of $\pi_{M_{p_i}}$

$\pi_{M_{p_i}}$ data structure representing the connected keyframes to mappoint M_{p_i} .

Controlling concurrency:

Since we could potentially have all the three threads namely tracking, local mapping and loop closing calling the functions $addMappoint()$, $eraseMappoinMatch$ (), and $replaceMappointMatch()$, this could lead in a data race condition.

We use a mutex lock to protect the $Redundant\_mappoint\_connections$ container and make the $nRedundantObservations$ variable atomic.

Relevant calculations and Experimental results,

In summary, we reduce the time complexity of keyframe culling function, in a local mapping thread, which has a time complexity of $\boldsymbol{O(n_{\kappa_{c_i}} * n_{M_i} * n_{\pi_{m_i}})}$ to $\boldsymbol{O(n_{\kappa_{c_i}})}$**.**

However, we incur an increased time complexity in $addMappoint()$, $eraseMappoinMatch$ () , and $replaceMappointMatch()$ which was $O(1)$ to $O(\boldsymbol{n_{\pi_{M_{p_i}}}})$.

Another method to compare the two schemes can be, the amount of work that a keyframe does to maintain state and the amount of time a keyframe is repeated in a keyframe culling.

Based on statistical results, we find that a keyframe is repeated in keyframe culling approximately 30 times. The amount of work required would be 30* $\boldsymbol{O(n_{\kappa_{c_i}} * n_{M_i} * n_{\pi_{m_i}})}$**.**

On the other hand, we find the count of the number of times a mappoint undergoes state changes through its lifetime is ~10,000 times. Furthermore, we find that the maximum number of mappoints a keyframe can have been ~1024. So, the amount of work a keyframe undergoes to maintain state can be approximately given by $10{,}000 * 1024 * O(\boldsymbol{n_{\pi_{M_{p_i}}}})$.

Finally, we propose that work required for one keyframe in the alternative scheme would be lesser than the current marking scheme in Vanilla ORB-SLAM-3,

$10{,}000 * 1024 * O(\boldsymbol{n_{\pi_{M_{p_i}}}}) < 30*1024* \boldsymbol{O(n_{\kappa_{c_i}} * n_{\pi_{M_{p_i}}})}$

The summary of the time complexities for each function call is given below,

	Previous Marking Scheme	Current Marking Scheme
KeyframeCulling()	$O\left(\boldsymbol{n_{\kappa_{c_i}} * n_{M_i} * n_{\pi_{m_i}}}\right)$	$\boldsymbol{O(n_{\kappa_{c_i}})}$
AddMappoint( )	$\boldsymbol{O(1)}$	$\boldsymbol{n_{M_i} * O\left(n_{\pi_{M_{p_i}}}\right)}$
ReplaceMapPoint( )	$\boldsymbol{O(1)}$	$\boldsymbol{n_{M_i}} * O(\boldsymbol{n_{\pi_{M_{p_i}}}})$
DeleteMappoint( )	$\boldsymbol{O(1)}$	$\boldsymbol{n_{M_i}} * O(\boldsymbol{n_{\pi_{M_{p_i}}}})$

6. Summary:

In this thesis we introduce a mutex based reference counting scheme for implementing deletion in the ORB-SLAM-3. We then provide a lock-free implementation of the reference counting scheme. In comparison to mutex-based reference counting, compare and swap exhibits slightly superior performance. This could possibly be because mutex-based locks, as mandated by C++ standards, force a thread to sleep if it is unable to obtain a lock. Comparing and swapping, on the other hand, functions more like a spin lock and since the critical section of reference counting consists of an increment or a decrement, a compare and swap could perform better than mutex locking. The reference counting scheme works successfully work on the "EuRoC micro aerial vehicle datasets" with reference counting a subset of the codebase. However, this does not provide safe deletion on other data sets or on other architectures. For running the reference counting scheme on other datasets, we would require completing the reference counting for the complete ORB_SLAM-3 codebase rather than a subset.

Furthermore, this thesis introduces an alternative strategy for marking keyframes and mappoints for deletion which is potentially possible to reduce redundant computation. The theoretical analysis suggests a possible improvement in performance in local mapping thread. However, it should also be noted that an improvement in local mapping does not imply improvement in the SLAM performance as the strategy could add an overhead on the tracking thread.

References

[1] Semenova, S., Ko, S. Y., Liu, Y. D., Ziarek, L., & Dantu, K. (2022). A quantitative analysis of system bottlenecks in visual SLAM. HotMobile '22. https://doi.org/10.1145/3508396.3512882

[2] Li, F., Yang, S., Yi, X., & Yang, X. (2018). Towards Visual SLAM with Memory Management for Large-Scale Environments. In Lecture Notes in Computer Science (pp. 776–786). https://doi.org/10.1007/978-3-319-77383-4_76

[3] Barros, A., Michel, M., Moline, Y., Corre, G., & Carrel, F. (2022). A comprehensive survey of visual SLAM algorithms. Robotics, 11(1), 24. https://doi.org/10.3390/robotics11010024

[4] Xue, G., Wei, J., Li, R., & Cheng, J. (2022). LEGO-LOAM-SC: an improved simultaneous localization and mapping method fusing LEGO-LOAM and Scan context for underground coalmine. Sensors, 22(2), 520. https://doi.org/10.3390/s22020520

[5] MIMOSA: A Multi-Modal SLAM Framework for Resilient Autonomy against Sensor Degradation. (2022, October 23). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9981108

[6] Zhang, T., & Gao, X. (2021). Introduction to Visual SLAM. In Springer eBooks. https://doi.org/10.1007/978-981-16-4939-4

[7] S. Das, "Simultaneous localization and mapping (SLAM) using RTAB-map," arXiv.org, https://arxiv.org/abs/1809.02989 (accessed Jan. 5, 2024).

[8] Servières, M., Renaudin, V., Dupuis, A., & Antigny, N. (2021). Visual and Visual-Inertial SLAM: state of the art, classification, and experimental benchmarking. Journal of Sensors, 2021, 1–26. https://doi.org/10.1155/2021/2054828

[9] Kopetz, H., & Steiner, W. (2022). Real-Time systems. In Springer eBooks. https://doi.org/10.1007/978-3-031-11992-7

[10] The concise handbook of real-time systems - UFPE. (n.d.-b). https://www.cin.ufpe.br/~svc/str/rthandbook.pdf

[11] L. Riazuelo, J. Civera, and J. M. Montiel, “C2TAM: A Cloud framework for cooperative tracking and mapping,” Robotics and Autonomous Systems, vol. 62, no. 4, pp. 401–413, Apr. 2014, doi: 10.1016/j.robot.2013.11.007.

[12] F. Dayoub, G. Cielniak, and T. Duckett, “Long-term experiments with an adaptive spherical view representation for navigation in changing environments,” Robotics and Autonomous Systems, vol. 59, no. 5, pp. 285–295, May 2011, doi: 10.1016/j.robot.2011.02.013.

[13] “An adaptive appearance-based map for long-term topological localization of mobile robots,” IEEE Conference Publication | IEEE Xplore, Sep. 01, 2008. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4650701

[14] T. Lindeberg, “Scale invariant feature transform,” Scholarpedia, vol. 7, no. 5, p. 10491, Jan. 2012, doi: 10.4249/scholarpedia.10491.

[15] H. Bay, T. Tuytelaars, and L. Van Gool, “SURF: Speeded up robust features,” in Lecture Notes in Computer Science, 2006, pp. 404–417. doi: 10.1007/11744023_32.

[16] E. Rublee, V. Rabaud, K. Konolige and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," 2011 International Conference on Computer Vision, Barcelona, Spain, 2011, pp. 2564-2571, doi: 10.1109/ICCV.2011.6126544.

[17] C. G. Harris, “A combined corner and edge detector,” 1988. https://www.semanticscholar.org/paper/A-Combined-Corner-and-Edge-Detector-Harris-Stephens/6818668fb895d95861a2eb9673ddc3a41e27b3b3

[18] G. Klein and D. Murray, “Parallel tracking and mapping for small AR workspaces,” in IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR), Nara, Japan, November 2007, pp. 225–234

[19] A. A. De Amorim, C. Hriţcu, and B. C. Pierce, “The meaning of memory safety,” in Lecture Notes in Computer Science, 2018, pp. 79–105. doi: 10.1007/978-3-319-89722-6_4.

[20] R. Jones, A. L. Hosking, and J. E. B. Moss, The Garbage Collection Handbook. 2023. doi: 10.1201/9781003276142.

[21] “CS 4120 Spring 2023.” https://www.cs.cornell.edu/courses/cs4120/2023sp/notes.html?id=gc

[22] D. Soshnikov, “Writing a memory allocator,” Dmitry Soshnikov, Jan. 04, 2024. http://dmitrysoshnikov.com/compilers/writing-a-memory-allocator/#allocator-interface

[23] “JEP 450: Compact Object Headers (Experimental).” https://openjdk.org/jeps/450

[24] S. Chandrachary, “A brief introduction to GraphSLAM - Shiva Chandrachary - Medium,” Medium, Nov. 11, 2022. [Online]. Available: https://shivachandrachary.medium.com/a-brief-introduction-to-graphslam-4204b4fce2f0#:~:text=One%20of%20the%20major%20benefits,to%20find%20the%20optimal%20solution.

[25] N. L. Large, F. Bieder, and M. Lauer, “Comparison of different SLAM approaches for a driverless race car,” Tm-technisches Messen, vol. 88, no. 4, pp. 227–236, Mar. 2021, doi: 10.1515/teme-2021-0004.

[26] B. Mishra, R. J. Griffin, and H. E. Sevil, “Modelling software architecture for visual simultaneous localization and mapping,” Automation, vol. 2, no. 2, pp. 48–61, Apr. 2021, doi: 10.3390/automation2020003.

[27] Engelen, R. A. van. (n.d.). Names, scopes, and Bindings. Computer Science, FSU. https://www.cs.fsu.edu/~engelen/courses/COP402001/notes5.html

[28] M. Pöter, “Memory models for C/C++ programmers,” arXiv.org, Mar. 12, 2018. https://arxiv.org/abs/1803.04432

[29] D. Padua et al., “Memory models,” in Springer eBooks, 2011, pp. 1107–1110. doi: 10.1007/978-0-387-09766-4_419.

[30] Berger, E. (2009, Fall). Operating Systems [Lecture 18]. https://people.cs.umass.edu/~emery/classes/cmpsci377/current/notes/lecture_21_gc.pdf

[31] “Memory allocators.” https://www.cs.nmsu.edu/~ekerriga/presentation/index2.html

[32] D. Kieras, Using C++11’s smart pointers - websites.umich.edu, https://websites.umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf (accessed Jan. 7, 2024).

[33] “CON33-C. Avoid race conditions when using library functions - SEI CERT C Coding Standard - Confluence.” https://wiki.sei.cmu.edu/confluence/display/c/CON33-C.+Avoid+race+conditions+when+using+library+functions

[34] “An Introduction to Real-Time Java Technology: Part 1, The Real-Time Specification for Java (JSR 1).” https://www.oracle.com/technical-resources/articles/javase/jsr-1.html

[35] Paul L Rosin. “Measuring Corner Properties”. In: Comput. Vis. Image Underst. 73.2 (1999), pp. 291–307. issn: 1077-3142. doi: 10.1006/cviu.1998.0719.

[36] “Garbage collection,” Garbage Collection (GC), https://www.ibm.com/docs/en/sdk-java-technology/8?topic=management-garbage-collection-gc (accessed Jan. 6, 2024).

[37] C++ Working draft - open-std.org, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4892.pdf (accessed Jan. 7, 2024).

[38] T. Domani, E. K. Kolodner, and E. Petrank, “A generational on-the-fly garbage collector for Java,” ACM, May 2000, doi: 10.1145/349299.349336.

[39] S. M. Blackburn and K. S. McKinley, “Immix,” Sigplan Notices, vol. 43, no. 6, pp. 22–32, May 2008, doi: 10.1145/1379022.1375586.

[40] “ORB-SLAM3: an accurate Open-Source library for Visual, Visual–Inertial, and Multimap SLAM,” IEEE Journals & Magazine | IEEE Xplore, Dec. 01, 2021. https://ieeexplore.ieee.org/document/9440682

[41] “ORB-SLAM: a versatile and accurate monocular SLAM system,” IEEE Journals & Magazine | IEEE Xplore, Oct. 01, 2015. https://ieeexplore.ieee.org/document/7219438

[42] “Detectsurffeatures,” MathWorks, https://www.mathworks.com/help/vision/ug/visual-simultaneous-localization-and-mapping-slam-overview.html (accessed Jan. 7, 2024).

[43] “Bags of binary words for fast place recognition in image sequences,” IEEE Journals & Magazine | IEEE Xplore, Oct. 01, 2012. https://ieeexplore.ieee.org/document/6202705

[44] E. Rosten and T. Drummond, “Machine Learning for High-Speed Corner Detection,” in Lecture Notes in Computer Science, 2006, pp. 430–443. doi: 10.1007/11744023_34.

[45] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, “BRIEF: Binary Robust Independent Elementary Features,” in Lecture Notes in Computer Science, 2010, pp. 778–792. doi: 10.1007/978-3-642-15561-1_56.

[46] 12.2 Essential Matrix - CMU School of Computer Science, https://www.cs.cmu.edu/~16385/s17/Slides/12.2_Essential_Matrix.pdf (accessed Jan. 8, 2024).

[47] M. A. Fischler and R. C. Bolles, “Random sample consensus,” Communications of the ACM, vol. 24, no. 6, pp. 381–395, Jun. 1981, doi: 10.1145/358669.358692.

[48] Y. Chen1, Y. Chen1, and G. Wang1, “Bundle Adjustment Revisited.” Accessed: May 27, 2023. [Online]. Available: https://arxiv.org/pdf/1912.03858.pdf

[49] “Bipartite graph of factors and nodes - MATLAB,” www.mathworks.com. https://www.mathworks.com/help/nav/ref/factorgraph.html (accessed Jan. 08, 2024).

[50] L. Romero, E. F. Morales, and L. E. Sucar, “Solving the global localization problem for indoor mobile robots,” in Lecture Notes in Computer Science, 2003, pp. 416–423. doi: 10.1007/978-3-540-24586-5_51.

[51] “Introduction.” https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume11/fox99a-html/node1.html

[52] S. Huang and G. Dissanayake, “Robot Localization: An Introduction,” Wiley Encyclopedia of Electrical and Electronics Engineering, pp. 1–10, Aug. 2016, doi: 10.1002/047134608x.w8318.