A Comprehensive Study of Systems Challenges in Visual Simultaneous Localization and Mapping Systems

SOFIYA SEMENOVA, Computer Science and Engineering, University at Buffalo, Buffalo, United States
STEVEN KO, Computing Science, Simon Fraser University, Burnaby, Canada
YU DAVID LIU, Computer Science, SUNY Binghamton, Binghamton, United States
LUKASZ ZIAREK, Computer Science and Engineering, University at Buffalo, Buffalo, United States
KARTHIK DANTU, Computer Science and Engineering, University at Buffalo, Buffalo, United States

Visual SLAM systems are concurrent, performance-critical systems that respond to real-time environmental conditions and are frequently deployed on resource-constrained hardware. Previous work has identified three interconnected systems challenges to building consistent, accurate, and robust SLAM systems—*timeliness*, *concurrency*, and *context awareness*. In this article, we analyze three popular, state-of-the-art frameworks with varying system designs and optimization techniques, and we quantify the extent to which they are affected by the aforementioned system challenges. We find that all SLAM systems must balance the interconnected nature of *timeliness* and accuracy, and different system designs and optimization techniques uniquely address this tension. Global-map-based SLAM systems typically achieve the best performance but suffer in resource-constrained scenarios with increased *concurrency*. Across all SLAM systems, incorporating *context awareness* into decision-making would mitigate the impact of timeliness and concurrency on accuracy in resource-constrained scenarios.

CCS Concepts: • **Computer systems organization** → **Embedded and cyber-physical systems**; **Real-time systems**; • **Computing methodologies** → *Computer vision;*

Additional Key Words and Phrases: Visual simultaneous localization and mapping, mobile systems

ACM Reference Format:
Sofiya Semenova, Steven Ko, Yu David Liu, Lukasz Ziarek, and Karthik Dantu. 2024. A Comprehensive Study of Systems Challenges in Visual Simultaneous Localization and Mapping Systems. *ACM Trans. Embedd. Comput. Syst.* 24, 1, Article 2 (September 2024), 31 pages. https://doi.org/10.1145/3677317

1 Introduction

Several classes of applications such as mobile augmented reality and autonomous driving require a 3D map of the environment for accurate functioning. **Simultaneous Localization and**

This project is sponsored by NSF Awards CNS-1823260, CNS-1823230, CNS-1846320, and SHF-1749539.
Authors' Contact Information: Sofiya Semenova (Corresponding author), Computer Science and Engineering, University at Buffalo, Buffalo, New York, United States; e-mail: sofiyase@buffalo.edu; Steven Ko, Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada; e-mail: steveyko@sfu.ca; Yu David Liu, Computer Science, SUNY Binghamton, Binghamton, New York, United States; e-mail: davidl@binghamton.edu; Lukasz Ziarek, Computer Science and Engineering, University at Buffalo, Buffalo, New York, United States; e-mail: lziarek@buffalo.edu; Karthik Dantu, Computer Science and Engineering, University at Buffalo, Buffalo, New York, United States; e-mail: kdantu@buffalo.edu.

ACM 1539-9087/2024/09-ART2
https://doi.org/10.1145/3677317

Mapping (SLAM) is a software framework that builds a map of an environment from sensor data while traversing through the environment and simultaneously localizing the mobile device within the map. Modern SLAM systems are visual and use monocular, RGB-D, or stereo camera frames, potentially with inertial information. Over the past decade, several Visual SLAM systems have been proposed to improve map and localization accuracy, such as KinectFusion [36], ORB-SLAM2 [34], ORB-SLAM3 [10], OpenVINS [18], and Kimera [39]. There has been significant interest and rapid progress in the robotics and computer vision communities on *algorithmic* innovations for accurate localization and mapping. New systems such as ORB-SLAM3, OpenVINS, and Kimera have a similar pipeline for map construction—a demonstration of the algorithmic maturity in state-of-the-art SLAM frameworks.

Recently, there has been work on offloading SLAM [2, 3, 11, 12, 44] tasks to edge servers for performance. However, the view of SLAM frameworks as *performance-critical software systems* has not been systematically explored. This is unfortunate, because SLAM algorithms are most commonly deployed on software/hardware ecosystems with constrained resources and stringent performance requirements. In our previous work [41], we identified three systems challenges that cause suboptimal operation in SLAM frameworks—timeliness, concurrency, and context awareness. In this article, we extend our analysis to three popular, state-of-the-art SLAM systems (ORB-SLAM3 [10], Kimera [39], and OpenVINS [18]) with varying system designs and algorithmic implementations. We believe these challenges deserve exposure in the mobile systems community, whose shared knowledge may systematically address these challenges and strengthen next-generation SLAM-based software stacks.

1.1 Systems Challenges in SLAM

Timeliness. The mapping and localization information generated in SLAM systems is used to aid real-time applications such as rendering virtual objects in AR or avoiding obstacles during robot navigation; these applications therefore require SLAM to adhere to tight timeliness constraints. Typical SLAM pipelines process incoming images, track the device's location using the processed image data, insert new location information into a global map, and optimize the map structure. Within this pipeline, each task has timeliness requirements that affect the ability of the entire system to meet its timeliness constraints and generate accurate mapping and localization information. To keep up with real-time camera streams, SLAM systems must process incoming images as fast as they arrive. At a frame rate of 20 fps, this means that SLAM systems must process each image within 50 ms. To avoid localization error (and failure) and keep up with real-time mapping, SLAM systems must avoid dropping camera frames and must keep the global map up-to-date with new location information. To perform as quickly and accurately as possible, SLAM systems must perform map optimization, which refines the map by removing redundant data and rectifies error accumulation, frequently.

Concurrency. The SLAM pipeline is sequential, but many popular SLAM systems split the pipeline into *concurrent* modules for better performance. Depending on the system design and specific algorithms used for each module, SLAM systems may have a *global map* (a shared data structure containing the generated mapping data) that is frequently and primarily accessed and modified by all modules in the system. Systems with a global map manage concurrent shared memory accesses through course-grained locks and by implementing a drop mechanism wherein modules drop or minimally process tasks if other modules are accessing the map or if the module has too much queued work. These drops curtail the total load on the system but lead to missing map and localization data, which in turn result in lowered accuracy and unpredictable results. We observe that systems *with* a global map generally outperform those without, but suffer from

adverse performance in highly concurrent environments (e.g., faster frame rates, lower resources) due to poorly designed concurrent access to the global map.

Context Awareness. Further, the execution of SLAM systems *as a whole* is dependent on the conditions of the real world, which are highly variable and unknown in advance. These conditions include device velocity, whether movement is rotational or translational [9, 38], and the density of visual features in the current environment. Variations in real-time conditions drastically affect the operation of the system and the importance of each module/task. Despite this, current SLAM systems indiscriminately drop data that may affect performance and accuracy to maximize throughput of all modules. Instead, SLAM systems would benefit from a more "intelligent" way to incorporate real-world, real-time conditions into their decision-making, so *the right task is being performed at the right time*.

1.2 Contributions

Previous SLAM frameworks have focused on algorithmic advancements in the SLAM pipeline but have largely kept the structure of their systems unchanged. Through years of experimentation, we find that SLAM systems suffer from performance problems whose solutions may come from systems design.

Our previous work [41] quantitatively analyzed systems challenges faced by ORB-SLAM2 [34] but was limited in its scope. First, we performed the analysis entirely on one sequence (the 00 sequence of the KITTI [17] dataset). In this work, we expand our analysis using the 11 sequences in the EuROC dataset. Second, in our previous work, we performed our analysis on only one system (ORB-SLAM2). While Visual SLAM systems share many commonalities in their system design and pipeline construction, the optimization techniques they use can vary drastically, and no two system designs are exactly alike. In this article, we compare three state-of-the-art SLAM systems with varying system designs and algorithmic implementations. The first, ORB-SLAM3, is the evolution of ORB-SLAM2 that introduces multi-map reasoning, the ability to use inertial sensors, and a recovery mechanism in case of relocalization. Like ORB-SLAM2, it is a KeyFrame-based SLAM system with a global map that performs full smoothing, which causes high levels of concurrency but also very good accuracy. The second, Kimera, is a modular system that has a very similar pipeline to ORB-SLAM3 but *does not use a global map*, leading to no concurrent shared memory accesses (and therefore no adverse effects due to high concurrency) but lower accuracy in the "normal" (non-resource-constrained) case. The third, OpenVINS, is a single-threaded, minimal visual-inertial navigation system that is very fast but less accurate when the device re-visits old locations.

As a key theme, this article attempts to elucidate the interconnected relationship among the three challenges we identified earlier and the degree to which these challenges affect SLAM frameworks with different system designs. In a nutshell, all SLAM systems must meet *timeliness goals*. One way to achieve this is by implementing faster, less accurate optimization techniques or foregoing some optimizations entirely, which increases throughput at the expense of accuracy. Another approach is to parallelize the system, which is the technique taken by most SLAM systems. However, increased parallelization causes increased *concurrency*, which in turn causes lower accuracy, for systems with a global map. This is unfortunate, as systems with a global map typically outperform those without. However, these drawbacks can be mitigated through *context awareness*.

2 Experimental Setup

We ran our experiments on two devices—an Nvidia Jetson TX2 (Dual-Core NVIDIA Denver and Quad-Core ARM Cortex-A57 CPUs, 8 GB Memory), which we use as the resource-constrained

Table 1. Datasets

Dataset	Environment	FPS	Device Type
EuRoC [8]	Indoor	20	MAV
TUM VI [40]	Indoor/Outdoor	20	Handheld Camera
KITTI [17]	Outdoor	10	Car
Hilti-Oxford [47]	Indoor/Outdoor	40	5 Fisheye Cameras

(a) Dataset information.

	EuRoC	TUM VI	KITTI	Hilti-Oxford
ORB-SLAM3	✓	✓	✓	
Kimera	✓			
OpenVINS	✓	✓	✓	

(b) System support for each dataset.

mobile device, and a System76 Galago Pro laptop (Dual-Core Intel Core i7-7500U CPU, 32 GB Memory), which is representative of a service robot navigating inside a building. Both devices run Ubuntu 18.04 LTS.

In our prior work [41], we ran all experiments with the default CPU frequency settings, which dynamically adjust the CPU clock frequency based on past performance and balance power efficiency and speed. For the Jetson, the default is the `schedutil` governor and `MAXP*` mode [13]. For the laptop, the default is the `intel_pstate` driver with **hardware P-state (HWP)** disabled and the `powersave` governor, which performs similarly to `schedutil` when combined with the `intel_pstate` driver [14]. However, we found that these settings significantly increase SLAM modules' durations at very low frame rates, because the CPU experiences a lot of idle time between processing frames and thereby lowers the CPU frequency. This masks each module's true duration and makes it difficult to compare performance across frame rates. For the experiments in this article, we modified the device settings to run their CPU at max performance regardless of prior workload or future workload estimates. To do this, we use the `MAXN` mode on the Jetson and the `performance` governor [7] on both devices.

We used the EuRoC MAV dataset [8], which contains 11 indoor sequences, runs at 20 fps, and is officially supported by all three systems. While several other suitable datasets exist, we unfortunately could not get any others to work on all systems at once. Table 1 contains a summary of the available datasets and each SLAM system's support for them.

3 Visual SLAM Overview

While different systems can modify the algorithms used for each task in the Visual SLAM pipeline and/or add additional tasks, most modern systems follow a similar architecture and design for basic mapping and localization. In this section, we describe the essential functions of a basic Visual SLAM system, identify the corresponding modules in the three test systems, and briefly compare the algorithmic implementations of each of the test systems.

A Visual SLAM pipeline consists of three modules—*Tracking*, *Local Mapping*, and *Loop Closure*. The system diagrams for OpenVINS, Kimera (VIO and RPGO), and ORB-SLAM3 are shown in Figure 1. All described modules run in a continuous loop on input from their respective queues.

3.1 Tracking

The *Tracking* module runs on every image received from the camera and is responsible for reading images and IMU information (if available) from input video and inertial sensor streams, extracting features in the image, preintegrating IMU measurements, and using the combined sensor data to determine the odometry with regard to the map. Additionally, if the number of feature matches are low or enough time has passed, then the module inserts the frame into the map as a *KeyFrame*. If this decision is made, then it sends the KeyFrame and its extracted features to *Local Mapping* for further processing. This module corresponds to *Tracking* in ORB-SLAM3, *VIO Front-End* in Kimera, and *Feature Detection and Matching* in OpenVINS (Figure 1, blue).

ORB-SLAM3's specific implementation of the Tracking module differs significantly from that of Kimera and OpenVINS. ORB-SLAM3 first gets an initial pose estimate by matching the current frame's features with the previous frame's and performing motion-only bundle adjustment, then refines the pose by searching for similar KeyFrames in the map and minimizing the reprojection error of their MapPoints. As such, their implementation heavily relies on global map access. Conversely, Kimera and OpenVINS track the current frame entirely through frame-to-frame feature matching with the previous frame, which does not use the global map. Kimera and OpenVINS both perform feature detection at every KeyFrame—however, Kimera inserts a subset of frames as KeyFrames, and OpenVINS inserts *every frame* as a KeyFrame.

3.2 Local Mapping

The *Local Mapping* module inserts the KeyFrame into the map along with any *MapPoints* (prominent features in the KeyFrame with their own camera poses, also sometimes referred to as *Landmarks*) and performs a local optimization. This module corresponds to *Local Mapping* in ORB-SLAM3, *VIO Back-End* in Kimera, and *Propagation, EKF Update, Re-tri & Marg* in OpenVINS (Figure 1, purple).

SLAM optimization methods can be roughly split into three camps: *filtering*, *fixed-lag smoothing* (also referred to as *sliding-window bundle adjustment* and *local bundle adjustment*), and *full smoothing* (also referred to as *full bundle adjustment* or just *bundle adjustment*). Broadly, filters estimate only the latest pose and marginalize out earlier states, fixed-lag smoothers estimate a window of previous poses and marginalize states outside the window, and full smoothers estimate the complete trajectory. SLAM systems use these algorithms to optimize a *factor graph* that can be viewed as equivalent to the map, but some systems (such as ORB-SLAM3) keep a separate map data structure with which to construct the optimizable factor graph. OpenVINS uses the filtering algorithm from Mourikis et al. [33] with a modification to include *SLAM landmarks* (features that have been tracked for a long time) in the state vector to improve accuracy [27]. Kimera uses the fixed-lag smoothing algorithm from Forster et al. [16]. ORB-SLAM3 uses a fixed-lag smoothing algorithm, but notably *never marginalizes information from the map*. Instead, it constructs a factor graph of local KeyFrames and associated MapPoints using the global map.

Despite the difference in optimization algorithms and map construction, the process for Local Mapping is roughly the same across all systems. First, they construct or update a factor graph containing KeyFrames and MapPoints. OpenVINS inserts every frame as a KeyFrame, while Kimera and ORB-SLAM3 insert a subset of frames as KeyFrames (this is sometimes called *KeyFrame-based SLAM*). Then, they optimize the factor graph to minimize error. ORB-SLAM3 optimizes both KeyFrame and MapPoint poses, while Kimera and OpenVINS optimize only KeyFrame poses. Finally, they cull or marginalize old information from the map. ORB-SLAM3 culls redundant KeyFrames and MapPoints that are only seen by a few KeyFrames. OpenVINS and Kimera cull map information that is outside the time horizon.

3.3 Loop Closing

The final module, *Loop Closing*, searches for loops using the *place recognition database* (typically a bag-of-words representation of all KeyFrames) for every inserted KeyFrame after it has been processed by Local Mapping. When a loop is detected, the KeyFrames on either side of the loop are aligned and further optimized. This module corresponds to *Loop Closing* in ORB-SLAM3, *RPGO* in Kimera, and is not implemented for OpenVINS (Figure 1, green).

For loop detection, both Kimera and ORB-SLAM3 use a bag-of-words representation to store and query the place recognition database. However, Kimera performs the bag-of-words for each

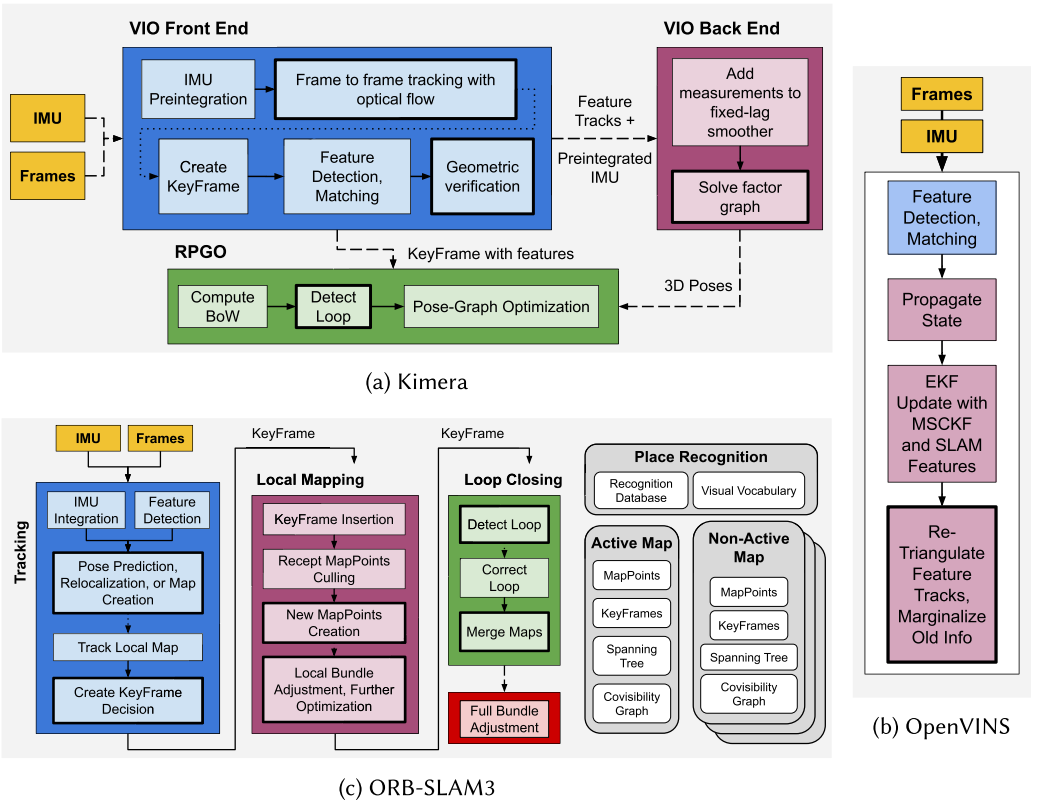


Fig. 1. System architectures for OpenVINS, Kimera, and ORB-SLAM3. Each module is implemented as a separate thread. Solid arrows indicate the next task, dotted arrows indicate enqueue and dequeue operations, and bold tasks indicate terminal states.

KeyFrame during the Loop Closing module, whereas ORB-SLAM3 performs it during Local Mapping. For map optimization, both systems perform an optimization over the global pose graph, which contains the entire set of KeyFrames for the camera trajectory. ORB-SLAM3 spawns this optimization step as a separate thread (*Full Bundle Adjustment* in Figure 1). Last, ORB-SLAM3 additionally includes a map merging step in the Loop Closing module as part of multi-map reasoning, which allows the system to create and maintain localization and map building across multiple, disjoint maps.

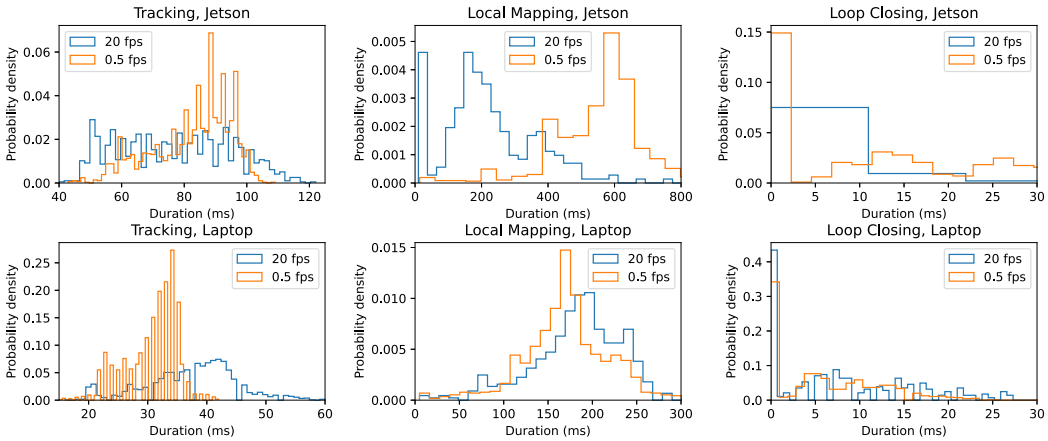
4 The Timeliness Challenge

The nature of the SLAM workload is sequential—control flows from Tracking to Local Mapping to Loop Closure. To improve overall timeliness, a typical pipeline executes these modules concurrently. This allows Tracking to work on the next image while Local Mapping is processing the previous KeyFrame. Systems without a loop closure module (such as OpenVINS) have a much faster processing time and can therefore run their pipeline sequentially without much impact on overall timeliness. While such pipelining provides overall efficiency, without proper scheduling the modules interfere with each other during shared access of the global map, resulting in inefficient operation.

Table 2. Execution Times of Each Module in ORB-SLAM3 on the Vicon Room 2 02 (V202) Sequence, for Two Frame Rates and Two Devices

ORB-SLAM3						
	Jetson, 20 fps			**Laptop, 20 fps**		
Module	Input Freq.	Throughput	Duration	Input Freq.	Throughput	Duration
Tracking	20.00 ± 0.00	9.64 ± 0.54	92.95 ± 15.23	20.00 ± 0.00	17.92 ± 4.01	46.07 ± 17.32
Local Mapping	2.16 ± 0.09	2.11 ± 0.08	393.21 ± 183.12	3.31 ± 0.49	3.31 ± 0.50	226.51 ± 90.57
Loop Closing	2.16 ± 0.09	2.10 ± 0.08	18.76 ± 28.97	3.31 ± 0.49	3.30 ± 0.51	8.79 ± 10.42
GBA	0.00 ± 0.01	0.00 ± 0.01	2,421.77 ± 249.01	0.01 ± 0.00	0.01 ± 0.00	746.68 ± 84.35
	Jetson, 0.5 fps			**Laptop, 0.5 fps**		
Module	Input Freq.	Throughput	Duration	Input Freq.	Throughput	Duration
Tracking	0.50	0.50	91.51 ± 10.08	0.50	0.50	34.17 ± 3.54
Local Mapping	0.17	0.17	611.88 ± 147.61	0.16	0.17	188.14 ± 50.52
Loop Closing	0.17	0.17	15.67 ± 16.45	0.16	0.16	6.30 ± 6.98
GBA	0.00	0.00	1,318.60	0.00	0.00	348.30

(a) *Input frequency* is the frequency of incoming data into each module's queue. Since modules can drop data, *throughput* is the frequency of data that has been processed.



(b) Duration variability of the Tracking, Local Mapping, and Loop Closing loops.

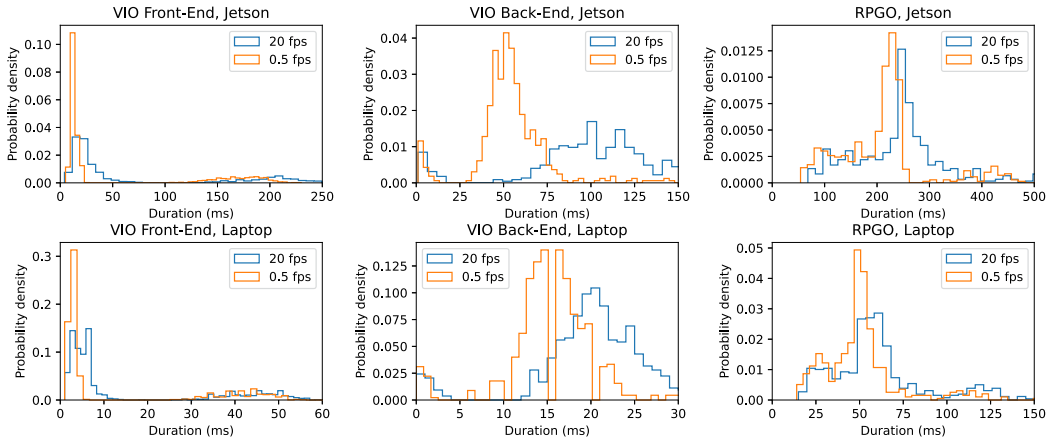
4.1 Timely Tracking

The Tracking module is responsible for reading incoming visual and IMU measurements, tracking the current frame, and optionally choosing to incorporate the frame into the map as a KeyFrame. Tracking runs in a loop, processing incoming camera images as quickly as possible. If incoming image frames arrive faster than Tracking can process them, then it will fall behind and start dropping images due to queue overflow. Dropping images can have varying impact on the localization and map accuracy. In feature-rich regions, loss of a few images does not largely affect the overall accuracy. In regions with fewer features or fast device movement, dropping even a single image could result in lower accuracy. In the worst case, the device can fail to localize the incoming image, resulting in tracking loss.

Table 3. Execution Times of Each Module in Kimera on the Vicon Room 2 02 (V202) Sequence, for Two Frame Rates and Two Devices

Kimera						
	Jetson, 20 fps			**Laptop, 20 fps**		
Module	Input Freq.	Throughput	Duration	Input Freq.	Throughput	Duration
VIO Frontend	20.00 ± 0.00	15.16 ± 0.11	65.12 ± 75.48	20.00 ± 0.00	20.00 ± 0.00	20.05 ± 27.45
VIO Backend	3.79 ± 0.03	3.78 ± 0.03	138.60 ± 38.67	5.00 ± 0.00	5.00 ± 0.01	25.99 ± 7.73
RPGO	3.26 ± 0.13	3.25 ± 0.12	307.52 ± 110.70	5.00 ± 0.00	5.00 ± 0.00	72.11 ± 26.70
	Jetson, 0.5 fps			**Laptop, 0.5 fps**		
Module	Input Freq.	Throughput	Duration	Input Freq.	Throughput	Duration
VIO Frontend	0.50	0.50	55.19 ± 64.88	0.50	0.50	12.02 ± 15.87
VIO Backend	0.12	0.12	74.11 ± 18.43	0.12	0.12	20.78 ± 5.09
RPGO	0.12	0.12	264.84 ± 78.68	0.12	0.12	62.62 ± 23.02

(a) *Input frequency* is the frequency of incoming data into each module's queue. Since modules can drop data, *throughput* is the frequency of data that has been processed.



(b) Duration variability of the VIO Front-End (Tracking), VIO Back-End (Local Mapping), and RPGO (Loop Closing) loops.

The EuRoC dataset runs at 20 fps, generating an image every 50 ms. Therefore, it is imperative for Tracking to process frames within 50 ms to keep up with real-time operation. Tables 2(a), 3(a), and 4 show the execution times, input frequency, and throughput of each module in ORB-SLAM3, Kimera, and OpenVINS. At the real-time rate of 20 fps, Tracking takes an average of 93 ms (Jetson) and 46 ms (laptop) in ORB-SLAM3, 65 ms and 20 ms in Kimera, and 14 ms and 6 ms in OpenVINS. While the Tracking portion of OpenVINS is very short compared to that of Kimera and ORB-SLAM3, it takes OpenVINS 53 ms (Jetson) and 16 ms (laptop) to process a frame because it runs the entire pipeline in a single thread. *At the real-time rate, all three systems on the Jetson cannot meet timeliness goals and thus experience frame drops.* We also run each system at 0.5 fps, to simulate an environment where the device has plenty of resources to meet all deadlines. At this frame rate, Tracking takes an average of 92 ms and 34 ms in ORB-SLAM3, 55 ms and 12 ms in Kimera, and 21 ms and 5 ms in OpenVINS. *All systems are able to meet timeliness goals at this*

Table 4. OpenVINS Execution Times on the Vicon Room 2 02 (V202) Sequence, for Two Frame Rates, Two Devices, and Single-threaded or Multi-threaded OpenCV

OpenVINS						
	Jetson, 20 fps					
	4 OpenCV threads			1 OpenCV thread		
Module	Input Frequency	Throughput	Duration	Input Frequency	Throughput	Duration
Tracking	20	–	14.51 ± 3.37	20	–	14.38 ± 3.74
Propagation	–	–	0.75 ± 0.23	–	–	0.74 ± 0.20
MSCKF Update	–	–	4.08 ± 4.98	–	–	4.01 ± 4.29
SLAM Update	–	–	18.00 ± 14.42	–	–	18.07 ± 13.95
Re-tri & Marg	–	16.12 ± 0.17	15.51 ± 0.56	–	16.30 ± 0.06	15.22 ± 0.58
TOTAL	20	16.12 ± 0.17	52.85 ± 16.04	20	16.30 ± 0.06	52.42 ± 15.58
	Laptop, 20 fps					
	4 OpenCV threads			1 OpenCV thread		
Module	Input Frequency	Throughput	Duration	Input Frequency	Throughput	Duration
Tracking	20	–	6.12 ± 1.13	20	–	8.55 ± 1.31
Propagation	–	–	0.21 ± 0.04	–	–	0.24 ± 0.03
MSCKF Update	–	–	1.31 ± 1.69	–	–	1.66 ± 2.11
SLAM Update	–	–	6.56 ± 3.84	–	–	8.28 ± 4.79
Re-tri & Marg	–	20.00 ± 0.00	2.24 ± 0.20	–	20.00 ± 0.01	2.52 ± 0.21
TOTAL	20	20.00 ± 0.00	16.43 ± 4.53	20	20.00 ± 0.01	21.25 ± 5.57
	Jetson, 0.5 fps					
	4 OpenCV threads			1 OpenCV thread		
Module	Input Frequency	Throughput	Duration	Input Frequency	Throughput	Duration
Tracking	0.50	–	20.59 ± 2.56	0.50	–	21.22 ± 2.50
Propagation	–	–	0.72 ± 0.09	–	–	0.71 ± 0.09
MSCKF Update	–	–	4.56 ± 5.36	–	–	4.41 ± 5.14
SLAM Update	–	–	22.70 ± 13.06	–	–	22.54 ± 12.87
Re-tri & Marg	–	0.50	15.33 ± 0.39	–	0.50	15.01 ± 0.34
TOTAL	0.50	0.50	63.91 ± 15.24	0.50	0.50	63.89 ± 14.99
	Laptop, 0.5 fps					
	4 OpenCV threads			1 OpenCV thread		
Module	Input Frequency	Throughput	Duration	Input Frequency	Throughput	Duration
Tracking	0.50	–	5.22 ± 1.19	0.50	–	7.83 ± 1.22
Propagation	–	–	0.21 ± 0.04	–	–	0.24 ± 0.04
MSCKF Update	–	–	1.33 ± 1.69	–	–	1.72 ± 2.22
SLAM Update	–	–	6.52 ± 3.84	–	–	8.52 ± 5.02
Re-tri & Marg	–	0.50	2.24 ± 0.19	–	0.50	2.55 ± 0.18
TOTAL	0.50	0.50	15.52 ± 4.60	0.50	0.50	20.87 ± 5.84

Because OpenVINS is single-threaded, each module has the same input frequency and throughput, and we report the information for the entire thread as "TOTAL."

frame rate. For Kimera, the Tracking module duration has a fairly large standard deviation because the duration has a bimodal distribution (Table 3(b)). The first peak corresponds to frames that are only tracked using optical flow, while the second peak corresponds to frames that are inserted as KeyFrames and thus need feature extraction.

Recovering from Tracking Loss. In the short-term, a system can ignore the tracking loss and continue trajectory estimation using IMU measurements. This step is skipped if a system does not

process IMU (ORB-SLAM2) or IMU is not available. While IMU-only odometry provides a short-term buffer to allow the system to get back on track without more serious repercussions, it is not very accurate and drifts quickly [25]. Therefore, it is imperative to re-integrate current visual information as soon as possible. All systems can begin feature detection immediately after tracking loss, but this poses another problem: how to connect the newly detected features to the existing map.

Kimera and OpenVINS adopt a simple solution that continually attempts to match the current frame's features to recently seen features in the existing map. If a match is successfully made, then the new KeyFrames can be added directly to the map and the only difference in the map would be missing visual measurements and potentially worse pose estimates during the period of tracking loss. However, this method fails when the device cannot track prior features—entering a previously, but not recently, seen area (e.g., turning a corner into a hallway that was seen before) or entering a completely new area. In both circumstances, this solution would indefinitely fail to relocalize the device and lead to a complete loss of localization and map building.

ORB-SLAM2 and non-IMU modes of ORB-SLAM3 instead perform *relocalization*, a technique to re-acquire the device's location by matching the features in the current frame to a database of all KeyFrames. Relocalization addresses the scenario where a device enters a previously but not recently seen area by creating *mid-* to *long-term data associations* (i.e., feature matches that span a longer range than just the most recently seen features). While relocalization can successfully re-acquire the device location when it enters a previously seen area, it still fails indefinitely when the device enters a completely new area, because there are no prior KeyFrames that the current frame can be compared to. Additionally, it is a separate, computationally intensive process that puts mapping on hold and leads to a further loss in map building. Such a cascading effect completely derails the SLAM process on a resource-constrained device, so it is very important for the device to manage its resources to avoid relocalization at all costs.

After 3 seconds of IMU-only odometry or relocalization attempts, all modes of ORB-SLAM3 employ a different technique—after a sufficient period of tracking loss, create a new map and merge maps in the future if they have overlapping regions. For cases where a device enters a previous but not recently seen area, the maps should be merged immediately, because the new map shares visual features with the old map. For cases where a device enters a completely new location, the maps can be merged in the future if the device enters an area that was seen in the old map. In addition to addressing both problematic cases, this solution also requires little to no immediate latency overheads. However, it requires additional overlapping region detection and map merge steps.

Effect of CPU Frequency and Scaling Governors. In our previous experiments [41], ORB-SLAM3's Tracking module had a longer duration at 0.5 fps than 20 fps, which we attributed to fewer frame and KeyFrame drops leading to a larger map size and larger search space for the localization step. However, in our current experiments, ORB-SLAM3's Tracking module is *shorter* for 0.5 fps on the laptop and around the same length on the Jetson (Table 2(b)), and Kimera's Tracking module is shorter at 0.5 fps for both Jetson and laptop (Table 3(b)). While part of the discrepancy between our results can be due to a difference in sequence/dataset (KITTI 00 vs. EuRoC Vicon Room 202) and a difference in system mode (Monocular vs. Stereo Inertial), they are more significantly affected by a change in max CPU frequency and scaling governors. As discussed in Section 2, the default settings adjust CPU clock frequency based on past performance and aim to balance power utilization and speed. While this is a good choice for most applications, we found that these settings artificially inflate each module's duration on the 0.5 fps experiments, making it difficult to compare the "real" performance across frame rates. To address this for our analysis, we run the CPU at max performance regardless of workload, which speeds up processing time at the cost of energy consumption. However, in real-world scenarios, choosing to always prioritize speed

above energy may not be desirable. *Fine-tuning the max/min CPU frequencies and governor/policies for the desired performance is a potential future optimization.*

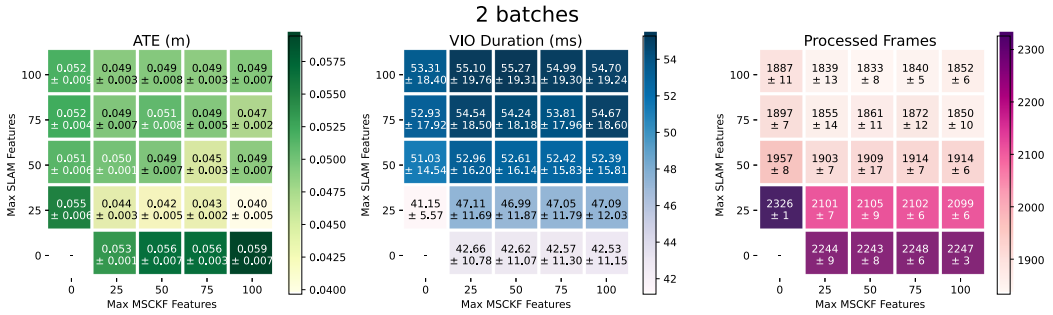
4.2 Timely Local Map Optimization

The Local Mapping module optimizes the map after each KeyFrame created in Tracking, primarily by culling redundant camera poses (KeyFrames) and visual information (MapPoints/Landmarks) and modifying the relative pose of map items to minimize overall error. Ideally, Local Mapping executes on every KeyFrame added to the map, making the map as efficient and accurate as possible. Ramifications for falling behind on or skipping Local Mapping processing depend on the system implementation and map design. For systems with a single global map (ORB-SLAM3), the Tracking and Loop Closure modules would interact with an un-optimized map, which is larger, less accurate, and more prone to tracking loss. For systems without a global map (Kimera and OpenVINS), map modifications essentially flow sequentially from Tracking to Local Mapping to Loop Closure, and maps later in the pipeline do not affect the operation of earlier modules. In these systems, Tracking would be unaffected (since it does not use the map at all) and Loop Closure would be skipped or delayed (since it needs Local Mapping results to function). However, frame-to-frame feature tracking is not very robust on its own, so these systems only report pose estimates *after* map optimization. Therefore, despite Tracking being unaffected by stalled or missing Local Mapping loops, it would still cause worse trajectory estimations.

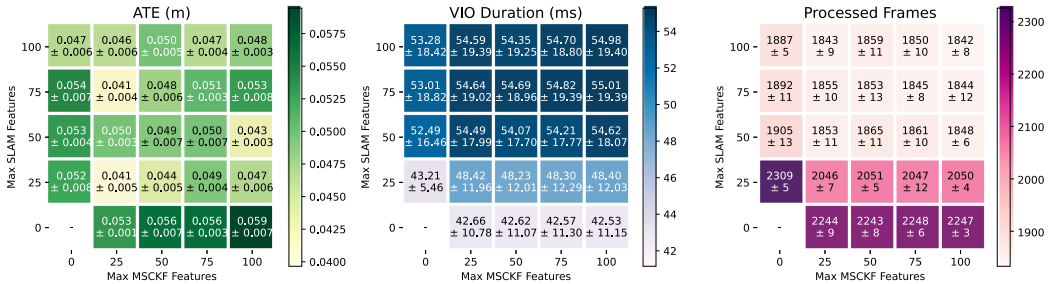
Our experiments show that at 20 fps, Local Mapping takes an average of 393 ms (Jetson) and 226 ms (laptop) for ORB-SLAM3 (Table 2(a)), and 139 ms and 26 ms for Kimera (Table 3(a)), and 38 ms and 10 ms for OpenVINS in multi-threaded mode (Table 4, the sum of "Propagation," "MSCKF Update," "SLAM Update," "SLAM Delayed," and "Re-tri & Marg"). For either frame rate, the duration of Local Mapping is not significantly affected when running OpenCV in single-threaded mode—it takes 38 ms and 12 ms at 20 fps, and 42.67 ms and 13.03 ms at 0.5 fps. As is the case for the Tracking module, the difference in performance for Local Mapping in the 0.5 fps experiments is affected by a difference in CPU frequency settings. Interestingly, after changing the CPU settings to prioritize performance, Kimera has a lower Local Mapping duration for 0.5 fps for both devices (Table 3(b)). Intuitively, we would expect the duration to be the same or higher for lower frame rates, because the system would experience fewer drops at lower frame rates, thus creating a larger map and requiring larger and longer map optimizations.

Accuracy/Latency Optimizations. OpenVINS introduces two modifications to the filtering approach in Mourikis et al. [33] that affect its performance with regard to accuracy and latency. First, they improve accuracy at the cost of increasing latency by introducing SLAM landmarks into the state vector, based on findings from Li et al. [27]. Then, they mitigate the increase in latency by performing a *sequential update* [1] of SLAM features to the state vector. This process involves updating the state vector multiple times with batches of SLAM landmarks, rather than updating the state with all landmarks at once.

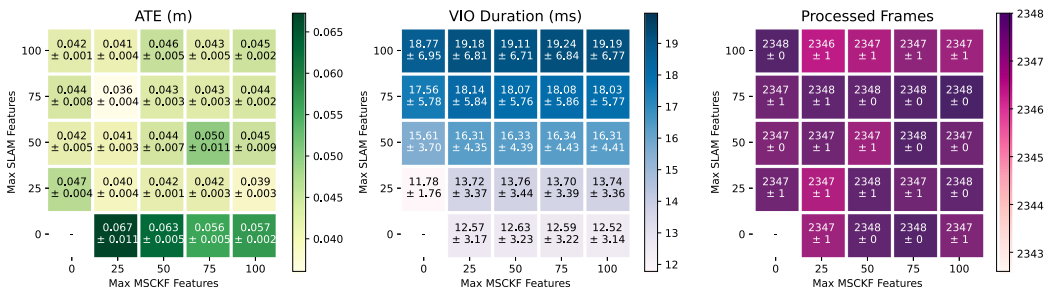
With the above modifications, the exact performance of the system depends on the number of SLAM and MSCKF (default) features in the state vector and the batch size for the sequential updates. OpenVINS allows the user to modify these variables, which we experiment with in Figure 2. We found that available resources on the device and its subsequent ability (or inability) to meet timeliness constraints strongly affected the actual outcome. Notably, increasing SLAM features only improves accuracy if the device has enough resources to handle the additional latency. This is because increasing SLAM features increases latency to such a degree that the system cannot meet timeliness goals and starts dropping more and more frames, undoing the accuracy benefit of increasing SLAM features in the first place. *Because resource constraints and timeliness so strongly affect the actual performance of the system, it is important to tailor optimizations for specific hardware.*



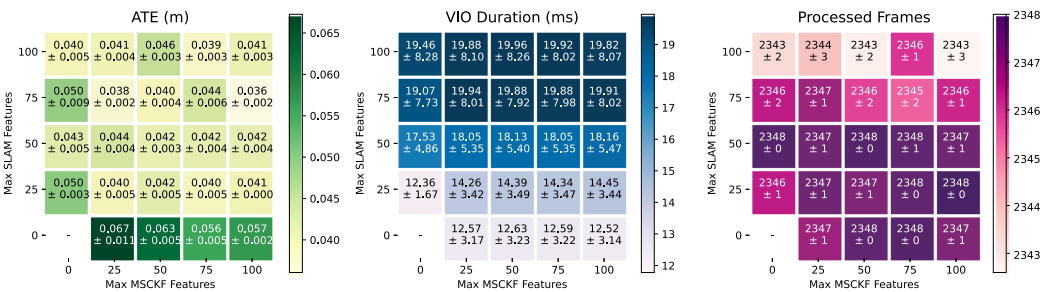
(a) Jetson, sequential updates with 2 batches.



(b) Jetson, no batching.



(c) Laptop, sequential updates with 2 batches.



(d) Laptop, no batching.

Fig. 2. The ATE, average duration, and total processed frames for OpenVINS when varying the maximum MSCKF features allowed (default 40), the maximum SLAM features allowed (default 50), and the batch size of SLAM features (default sequential updates with 2 batches). The sequence is V202.

4.3 Timely Loop Detection and Closure

The Loop Closing module detects and closes loops in the trajectory to correct accumulated trajectory drift. In ORB-SLAM3, the Loop Closing module also merges duplicate, disjoint maps. Executing Loop Closing as soon as possible results in a globally optimized, minimized, efficient map with minimal error. This is important for accurate tracking and mapping, especially when the tracking results need to be used in real-time. However, Loop Closing is also the largest module in the entire system. The loop detection step involves a fairly large read operation on the map during which no new map data can be added. For systems with a global map, this requires a halt on execution of other modules, which has ramifications for overall performance, as described earlier. After a loop is detected, the loop closing step involves an optimization of *the entire map* (a very lengthy process) and modifies a fairly large portion of poses, the latter of which also halts the execution of other modules. Therefore, while the Loop Closing module greatly increases accuracy, it may not be advantageous to perform it as soon as possible. Identifying how long Loop Closing can be delayed, or identifying an idle time in map access when the execution of Loop Closing does not affect other modules, could greatly improve overall SLAM performance.

For all systems, Loop Closing has by far the longest duration of any of the other modules. In Kimera, Loop Closing takes 308 ms (Jetson) and 72 ms (laptop) at 20 fps. Because Kimera does not use a global map, the only downside of Loop Closing's long duration is that it uses computational resources. In ORB-SLAM3, we report both the Loop Closing and **global bundle adjustment (GBA)** duration in Table 2(a), because they run in separate threads. The Loop Closing thread performs loop detection and map merging, then spawns the GBA thread to perform the optimization step. To control access to shared memory and mitigate the queue of KeyFrames for Local Mapping to process, ORB-SLAM3 halts the execution of other threads during the Loop Closing thread. However, *this is not the case for the GBA thread*, so Tracking and Local Mapping can continue during global bundle adjustment. Because new KeyFrames may have been inserted by Tracking and Local Mapping during GBA, the GBA thread must merge new map data into the optimized map after the optimization is complete. Allowing Tracking and Local Mapping to continue during GBA is beneficial, because at 20 fps, Loop Closing takes 19 ms (Jetson) and 9 ms (laptop), while GBA takes 2,422 ms and 745 ms! This is much larger than Kimera's Loop Closing module, because ORB-SLAM3 optimizes both MapPoint and KeyFrame poses (Kimera only optimizes KeyFrame poses), which is a much larger optimization task but leads to much more accurate results. Last, both systems have slightly shorter durations at 0.5 fps compared to 20 fps, but the difference is well within a standard deviation and likely not due to any difference in system operation at slower frame rates (Tables 2(b) and 3(b)).

5 The Concurrency Challenge

Each module in a SLAM system operates on a loop, processing data that arrives in its input queue. The first module, Tracking, accepts images as input from the camera. To avoid falling behind real-time, the Tracking queue is only allowed to hold 1 frame in its queue and drops any additional frames that arrive while it is busy processing a previous frame. In our experiments, we found that *all three systems* experience extreme frame drops on the Jetson, moderate frame drops on the laptop, and very few frame drops on either device when running at a very slow frame rate (Tables 6–8). Increased frame drops on resource-constrained devices can occur for several reasons, such as increased Tracking durations or longer lock wait times, which ultimately cause the system to be unable to meet Tracking timeliness goals. Because frame drops lead to weaker tracking (fewer feature matches between the current and previous frames) and missing map information (frames that are not processed cannot be inserted into the map), frame drops negatively affect the **ATE**

Table 5. For ORB-SLAM3, the Amount of Created KeyFrames in the Jetson and Laptop at 20 fps as a Percentage of the "Ideal" Amount of KeyFrames for the Sequence

Sequence	Jetson	Laptop	Sequence	Jetson	Laptop
MH01	60%	81%	V102	33%	51%
MH02	56%	78%	V103	23%	40%
MH03	36%	58%	V201	54%	77%
MH04	26%	47%	V202	32%	53%
MH05	28%	52%	V203	27%	57%
V101	50%	70%			

The ideal occurs in either the Jetson or laptop 0.5 experiments (whichever is higher).

(average trajectory error). Tables 6–8 show the relationship between device resources, frame drops, and ATE for ORB-SLAM3, Kimera, and OpenVINS, respectively.

However, despite all three systems dropping more frames on the Jetson at 20 fps, ORB-SLAM3 experiences the greatest impact on its trajectory error. This is because frame drops (due to Tracking failing timeliness goals) accounts for only a portion of the increased ATE on resource-constrained devices. SLAM systems that use a global map for localization and map building (such as ORB-SLAM3) are additionally affected by *increased concurrency*, while systems without a global map (Kimera, OpenVINS) are not, even if they are multi-threaded. In this section, we will discuss the map design of each system and the subsequent relationship between decreased performance, data drops, and resource contention.

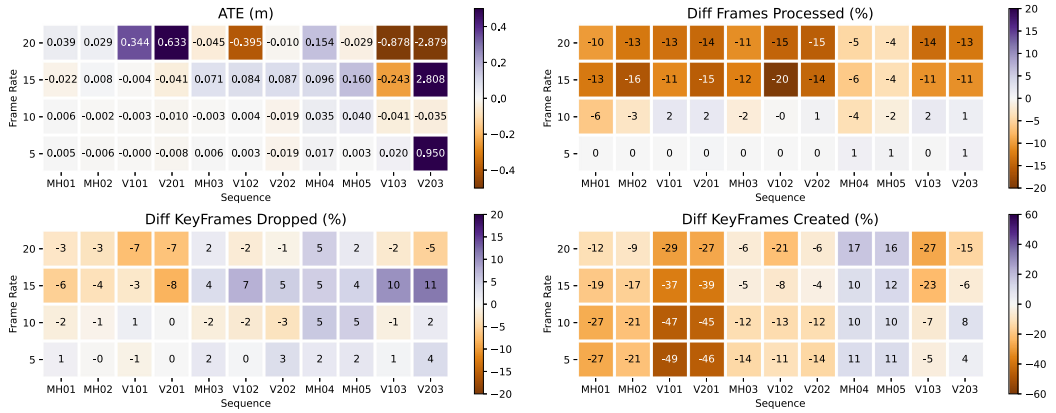
5.1 Concurrency in ORB-SLAM3

All modules in ORB-SLAM3 are implemented in multiple concurrently executing threads that jointly perform work on a *shared global map*, with a high frequency of reads and writes and a wide variety of short to long-range queries. ORB-SLAM3 manages shared memory access using locks, allowing modules sequential access. Table 2(a) shows the *effective* execution time of each module, including the time spent waiting on lock acquisitions. However, we found that blocks due to lock acquisitions accounted for very little of the total execution time. This is because ORB-SLAM3 was designed to drop or minimally process KeyFrames rather than wait on lock acquisitions to aid each module in meeting its timeliness goals. Unfortunately, KeyFrame drops result in missing map and localization data, which in turn may result in additional workload (relocalization), a dramatic reduction in performance, and even a complete halt in KeyFrame creation. In our experiments (Table 6), we observe that more resource-constrained environments suffer from higher ATE (average trajectory error), which we attribute to an increased amount of KeyFrame drops due to concurrency.

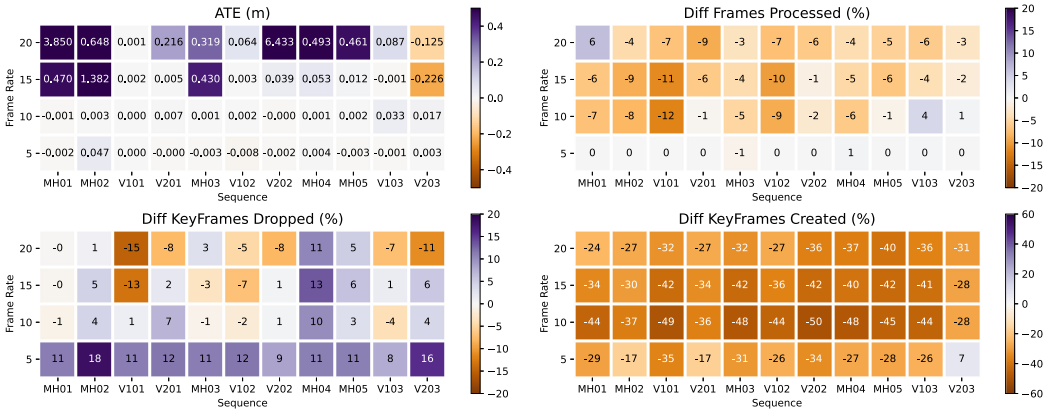
KeyFrame Drops. Local Mapping operates on a loop, processing KeyFrames inserted into its queue by Tracking. To avoid processing stale data and falling behind real-time, ORB-SLAM3 bounds the queue to 3 KeyFrames for non-monocular modes and 1 for all other modes. To prevent drops in the queue, Local Mapping skips additional optimizations ("New MapPoints Creation" and "Local Bundle Adjustment, Further Optimization" in Figure 1(c)) if there are any items in its queue. This has ramifications for the accuracy (due to a sub-optimal map) and timeliness (due to a larger map) of localization and mapping results. Further, because map optimization modifies a large part of the map, ORB-SLAM3 does not allow Tracking and Local Mapping to occur at the same time. When a new KeyFrame is created, the system chooses either to interrupt Local Mapping or to discard the new KeyFrame and allow Local Mapping to complete. The latter is a KeyFrame drop,

Table 6. The Total Tracked Images, Created and Dropped KeyFrames, and ATE for Both Devices at the Target Frame Rate (20 fps) and a Very Slow Frame Rate (0.5 fps) for the EuRoC Dataset

ORB SLAM3					
Sequence	Experiment	Tracked Images	Created KeyFrames	Dropped KeyFrames	ATE (m)
MH01	Laptop, 20 fps	3,540 ± 63	500 ± 2	2,036 ± 69	0.02 ± 0.00
	Jetson, 20 fps	1,688 ± 74	374 ± 24	812 ± 23	2.78 ± 4.12
	Laptop, 0.5 fps	3,682	619	0	0.02
	Jetson, 0.5 fps	3,682	612	0	0.02
MH02	Laptop, 20 fps	2,994 ± 12	440 ± 2	1,636 ± 52	0.04 ± 0.00
	Jetson, 20 fps	1,409 ± 26	313 ± 18	707 ± 19	2.03 ± 3.00
	Laptop, 0.5 fps	3,040	537	1	0.02
	Jetson, 0.5 fps	3,040	561	0	0.02
MH03	Laptop, 20 fps	2,651 ± 5	440 ± 5	1,704 ± 14	0.03 ± 0.00
	Jetson, 20 fps	1,283 ± 12	273 ± 3	735 ± 59	1.23 ± 0.79
	Laptop, 0.5 fps	2,700	757	0	0.03
	Jetson, 0.5 fps	2,700	753	1	0.03
MH04	Laptop, 20 fps	1,992 ± 9	380 ± 2	1,474 ± 5	0.05 ± 0.00
	Jetson, 20 fps	1,008 ± 8	211 ± 1	685 ± 10	0.23 ± 0.19
	Laptop, 0.5 fps	2,031	801	0	0.04
	Jetson, 0.5 fps	2,032	792	2	0.04
MH05	Laptop, 20 fps	2,242 ± 4	411 ± 8	1,438 ± 23	0.05 ± 0.00
	Jetson, 20 fps	1,152 ± 10	225 ± 5	684 ± 60	0.55 ± 0.59
	Laptop, 0.5 fps	2,272	757	1	0.06
	Jetson, 0.5 fps	2,272	798	0	0.04
V101	Laptop, 20 fps	2,824 ± 32	391 ± 1	1,871 ± 45	0.03 ± 0.00
	Jetson, 20 fps	1,160 ± 13	280 ± 1	500 ± 34	0.07 ± 0.06
	Laptop, 0.5 fps	2,911	560	0	0.03
	Jetson, 0.5 fps	2,911	544	0	0.03
V102	Laptop, 20 fps	1,678 ± 6	290 ± 7	1,016 ± 16	0.01 ± 0.00
	Jetson, 20 fps	824 ± 39	186 ± 11	295 ± 40	3.55 ± 4.31
	Laptop, 0.5 fps	1,708	551	0	0.01
	Jetson, 0.5 fps	1,708	566	0	0.01
V103	Laptop, 20 fps	2,145 ± 2	408 ± 1	1,335 ± 14	0.02 ± 0.00
	Jetson, 20 fps	1,121 ± 43	236 ± 11	494 ± 112	0.75 ± 1.33
	Laptop, 0.5 fps	2,149	1,014	0	0.02
	Jetson, 0.5 fps	2,149	1,009	2	0.02
V201	Laptop, 20 fps	2,275 ± 3	324 ± 2	1,242 ± 28	0.03 ± 0.00
	Jetson, 20 fps	1,018 ± 28	226 ± 8	455 ± 27	1.42 ± 2.41
	Laptop, 0.5 fps	2,279	420	0	0.03
	Jetson, 0.5 fps	2,280	412	1	0.01
V202	Laptop, 20 fps	2,333 ± 7	415 ± 7	1,578 ± 27	0.02 ± 0.02
	Jetson, 20 fps	1,131 ± 63	253 ± 10	559 ± 57	3.24 ± 3.46
	Laptop, 0.5 fps	2,348	774	0	0.01
	Jetson, 0.5 fps	2,348	787	6	0.01
V203	Laptop, 20 fps	1,916 ± 2	613 ± 9	1,035 ± 18	0.07 ± 0.08
	Jetson, 20 fps	1,235 ± 23	297 ± 9	472 ± 123	0.06 ± 0.04
	Laptop, 0.5 fps	1,921	1,061	0	0.03
	Jetson, 0.5 fps	1,921	1,081	0	0.03



(a) Stereo.



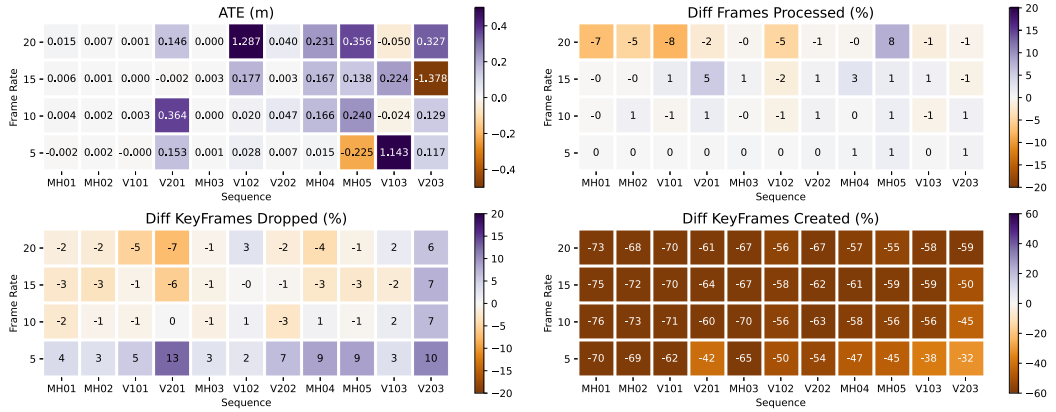
(b) Stereo Inertial.

Fig. 3. The difference between Jetson and laptop performance for Stereo and Stereo Inertial modes and 4 frame rates on every EuRoC sequence. Each reported value is the Jetson value minus the laptop value. % KeyFrames Dropped and % KeyFrames Created are relative to *total frames processed by each device.*

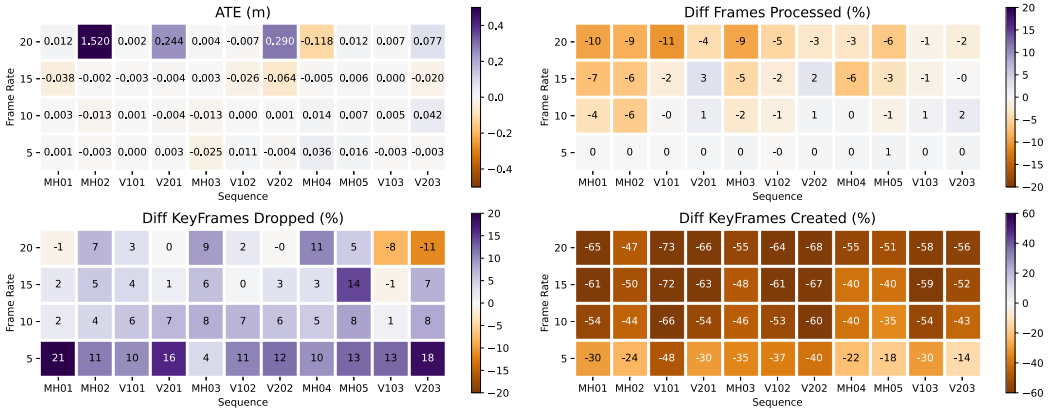
and we identify it as the primary effect of concurrency on performance in resource-constrained devices. This behavior is similar for Loop Closing—Tracking and Local Mapping will check that Loop Closing is not running before they attempt to process incoming data. However, KeyFrame drops due to concurrency with the Loop Closing thread are rare, because its duration is extremely short (< 5 ms) when a loop is not found and moderately short (<20 ms) when a loop *is* found, which occurs infrequently.

The extent to which resource contention affects KeyFrame creation can be seen in Table 5, which shows the amount of created KeyFrames for both devices at 20 fps *as a percentage of the "ideal" amount of KeyFrames for the sequence.* The ideal scenario occurs in either the Jetson or laptop 0.5 fps experiments, where there are no dropped KeyFrames at all. Across all sequences, both devices create fewer KeyFrames at 20 fps than at 0.5 fps, and the Jetson creates much fewer KeyFrames than the laptop.

Figures 3 and 4 show the difference in performance between the Jetson and laptop for Stereo and Monocular modes, respectively, for four frame rates and every sequence in the EuRoC dataset. The reported values refer to the Jetson value minus the laptop value—thus, a *positive* value (purple)



(a) Mono.



(b) Mono Inertial.

Fig. 4. The difference between Jetson and laptop performance for Monocular and Monocular Inertial modes and 4 frame rates on every EuRoC sequence.

indicates that the Jetson has a higher ATE, processes more frames, or creates/drops more KeyFrames, and vice versa for negative values (brown). The Jetson drops a significant amount of frames for frame rates faster then 10 fps for both stereo modes, some frames for frame rates faster than 10 fps for Monocular Inertial mode, and almost no frames in Monocular mode. This is intuitive, because Stereo modes require the system to process twice the amount of images.

% KeyFrames Dropped and % KeyFrames Created are relative to the *total frames processed by each device*—this means that the Jetson can process far fewer frames than the Laptop (e.g., Stereo mode at 20 fps) and have fewer KeyFrames than the laptop (because it has fewer opportunities to create them) but still create more KeyFrames relative to the amount of frames processed. We count a KeyFrame drop when it occurs *because of concurrency*—that is, the KeyFrame would have been inserted had Local Mapping not been busy. On average, the Jetson creates fewer and drops more KeyFrames compared to the laptop, but not always. While the Jetson experiences more concurrency due to its lowered resources, concurrency can also be induced through increased work. Finding the sweet spot between enough KeyFrames to reach the target accuracy and few enough to avoid increasing KeyFrame drops due to concurrency is a potential optimization.

Viewing CPU usage as a measure of the degree of concurrency experienced by a system, we found that the laptop uses slightly less CPU than the Jetson, on average, but both are within a standard deviation of each other (Figures 5(a) and 5(b), middle). Additionally, both devices experience similar spikes of CPU usage over time (Figures 5(a) and 5(b), top). However, these small differences in CPU usage translate to vastly different outcomes, as can be seen in Figures 5(a) and 5(b) (bottom). When a frame arrives from the camera, it can either be processed or dropped ("Dropped F," hashed blue). For every processed frame, the system can either create a KeyFrame ("Created KF," solid orange), drop a KeyFrame due to concurrency ("Dropped KF," hashed orange), or choose not to make a KeyFrame because the frame is not a good addition to the map ("Tracked F," solid blue). Across both datasets and modes, the Jetson drops more frames than the laptop for the same frame rate. While the laptop usually drops more KeyFrames, this is because it processes more frames overall. When the Jetson does not drop frames (5 fps, all modes and datasets), it drops more KeyFrames..

The Jetson has a worse ATE than the laptop, on average, but not always and not always to the same degree. This is caused by two reasons: (1) The laptop occasionally experiences more concurrency than the Jetson, because it can process more frames and therefore create more KeyFrames, and (2) Not all frame and KeyFrame drops equally affect mapping and localization accuracy, which we will discuss in the next section. Additionally, the performance of the Jetson relative to the laptop is far worse for non-inertial modes, which is intuitive, because inertial measurements can help stabilize the trajectory estimation in cases with no visual measurements.

5.2 Concurrency in Kimera and OpenVINS

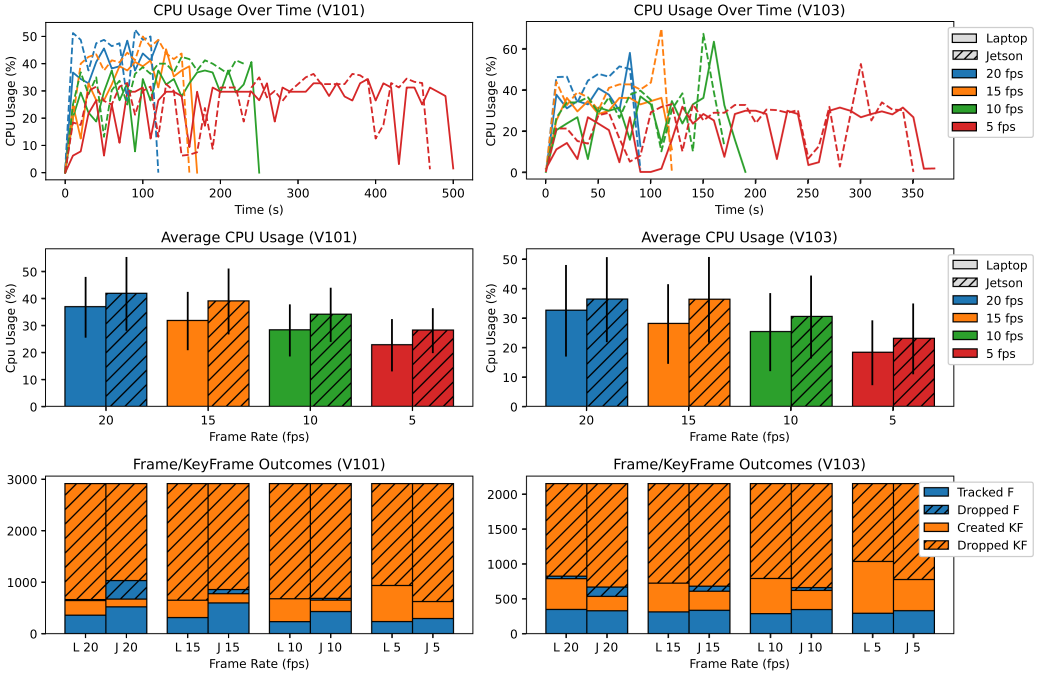
Unlike ORB-SLAM3, Kimera and ORB-SLAM3 have no shared, global map. Below, we discuss how the map data structures and pipeline are designed to avoid shared global map access and the ramifications of this design on performance.

Kimera Design. In Kimera, each module iteratively builds its own relevant data structures using information from modules earlier in the pipeline. This information flows from Tracking (VIO Front-End) to Local Mapping (VIO Back-End) to Loop Closing (RPGO), and the data structures from modules later in the pipeline are never used by modules earlier in the pipeline.

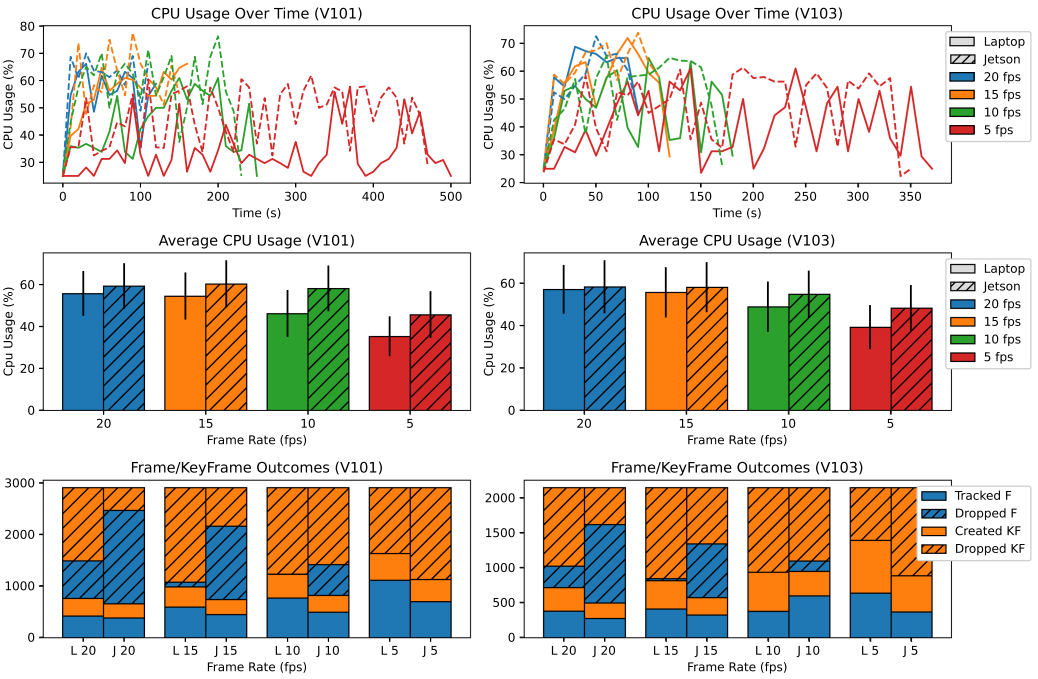
Kimera's Tracking module (VIO Front-End) performs frame-to-frame tracking instead of localizing the current robot position in the generated SLAM map. It stores frames (which can become KeyFrames) and their associated features. Modules later in the pipeline can modify KeyFrames by optimizing their pose, but updated poses are not necessary for frame-to-frame tracking and are thus not propagated back to Tracking. In our experiments, we observed that the majority of shared memory contentions in ORB-SLAM3 are between Tracking and Local Mapping. These memory contentions are eliminated with Kimera's method of tracking.

Kimera's Local Mapping module (VIO Back-End) optimizes a factor graph containing feature tracks, preintegrated IMU measurements, and KeyFrames. The factor graph contains only information within a time horizon of 30 KeyFrames, and older information is marginalized out. The factor graph is updated and optimized for every KeyFrame inserted by the Tracking module and then sends the optimized 3D poses to Loop Closing. In Table 7, we report the ATE of these poses as "ATE (VIO)."

Kimera's Loop Closing module contains two main map data structures—a *pose graph* and *place recognition database*. The pose graph (a factor graph) contains all poses generated by the Local Mapping module, is globally optimized by Loop Closing to generate more accurate pose estimations, and is most similar to ORB-SLAM3's global map (but does not include landmarks/MapPoints). Every pose graph optimization generates another estimated pose for the KeyFrame, but this information does not need to propagate upstream, because it is not required for the local optimizations



(a) Mono.



(b) Stereo Inertial.

Fig. 5. ORB-SLAM3 CPU usage and corresponding KeyFrame/frame outcomes (Tracked or dropped frames, created or dropped KeyFrames) for two datasets and two system types.

Table 7. The Total Tracked Images, Created and Dropped KeyFrames, and ATE for Both Devices at the Target Frame Rate (20 fps) and a Very Slow Frame Rate (0.5 fps) for the EuRoC Dataset

Kimera					
Sequence	Experiment	Tracked Images	Created KeyFrames	ATE VIO (m)	ATE Final (m)
MH01	Laptop, 20 fps	3,676 ± 5	920 ± 0	0.52 ± 0.05	0.52 ± 0.05
	Jetson, 20 fps	3,104 ± 286	776 ± 71	0.63 ± 0.03	0.71 ± 0.03
	Laptop, 0.5 fps	3,680	920	0.49	0.49
	Jetson, 0.5 fps	3,680	920	0.59	0.58
MH02	Laptop, 20 fps	3,038 ± 0	759 ± 0	0.14 ± 0.00	0.10 ± 0.00
	Jetson, 20 fps	2,655 ± 14	664 ± 3	0.13 ± 0.01	0.09 ± 0.02
	Laptop, 0.5 fps	3,038	759	0.14	0.09
	Jetson, 0.5 fps	3,038	759	0.14	0.09
MH03	Laptop, 20 fps	2,697 ± 0	674 ± 0	0.17 ± 0.02	0.13 ± 0.04
	Jetson, 20 fps	2,217 ± 124	554 ± 31	0.21 ± 0.03	0.17 ± 0.02
	Laptop, 0.5 fps	2,697	674	0.16	0.12
	Jetson, 0.5 fps	2,622	656	3.34	3.34
MH04	Laptop, 20 fps	2,028 ± 0	507 ± 0	0.17 ± 0.00	0.12 ± 0.03
	Jetson, 20 fps	1,903 ± 10	476 ± 2	0.17 ± 0.02	0.08 ± 0.01
	Laptop, 0.5 fps	2,029	507	0.16	0.14
	Jetson, 0.5 fps	2,029	507	0.21	0.21
MH05	Laptop, 20 fps	2,268 ± 1	567 ± 0	0.23 ± 0.02	0.22 ± 0.02
	Jetson, 20 fps	2,104 ± 13	526 ± 3	0.17 ± 0.00	0.12 ± 0.00
	Laptop, 0.5 fps	2,269	567	0.19	0.13
	Jetson, 0.5 fps	2,270	567	0.17	0.15
V101	Laptop, 20 fps	2,910 ± 1	727 ± 0	0.07 ± 0.01	0.06 ± 0.03
	Jetson, 20 fps	2,460 ± 210	615 ± 52	0.05 ± 0.01	0.06 ± 0.02
	Laptop, 0.5 fps	2,910	727	0.07	0.04
	Jetson, 0.5 fps	2,910	727	0.08	0.06
V102	Laptop, 20 fps	1,708 ± 0	427 ± 0	0.09 ± 0.00	0.08 ± 0.01
	Jetson, 20 fps	1,288 ± 3	322 ± 1	0.09 ± 0.00	0.09 ± 0.02
	Laptop, 0.5 fps	1,708	427	0.09	0.09
	Jetson, 0.5 fps	1,708	427	0.08	0.04
V103	Laptop, 20 fps	2,146 ± 0	536 ± 0	0.16 ± 0.00	0.12 ± 0.03
	Jetson, 20 fps	1,678 ± 6	419 ± 2	0.19 ± 0.01	0.18 ± 0.01
	Laptop, 0.5 fps	2,143	536	0.16	0.09
	Jetson, 0.5 fps	2,147	536	0.20	0.11
V201	Laptop, 20 fps	2,277 ± 0	569 ± 0	0.04 ± 0.00	0.04 ± 0.01
	Jetson, 20 fps	1,988 ± 13	497 ± 3	0.05 ± 0.01	0.03 ± 0.00
	Laptop, 0.5 fps	2,278	569	0.04	0.04
	Jetson, 0.5 fps	2,278	569	0.06	0.04
V202	Laptop, 20 fps	2,131 ± 429	532 ± 107	0.12 ± 0.02	0.09 ± 0.01
	Jetson, 20 fps	1,662 ± 399	415 ± 100	0.08 ± 0.02	0.08 ± 0.02
	Laptop, 0.5 fps	2,346	586	0.09	0.07
	Jetson, 0.5 fps	2,346	586	0.10	0.09
V203	Laptop, 20 fps	1,918 ± 0	580 ± 0	0.18 ± 0.00	0.19 ± 0.02
	Jetson, 20 fps	1,643 ± 7	484 ± 2	0.17 ± 0.00	0.18 ± 0.01
	Laptop, 0.5 fps	1,919	581	0.19	0.20
	Jetson, 0.5 fps	1919	581	0.20	0.18

"ATE VIO" is the ATE from poses in VIO Back-End, "ATE Final" is the ATE from poses in RPGO.

Table 8. The Total Tracked Images, MSCKF and SLAM Features, and ATE for Both Devices at the Target Frame Rate (20 fps) and a Very Slow Frame Rate (0.5 fps) for the EuRoC Dataset

OpenVINS					
Sequence	Experiment	Tracked Images	MSCKF Features	SLAM Features	ATE (m)
MH01	Laptop, 20 fps	3,679 ± 4	22.03 ± 32.30	39.04 ± 8.78	0.10 ± 0.01
	Jetson, 20 fps	2,943 ± 20	35.53 ± 56.62	37.60 ± 9.72	0.09 ± 0.01
	Laptop, 0.5 fps	3,682	29.28 ± 43.28	40.21 ± 8.52	0.08
	Jetson, 0.5 fps	3,681	32.36 ± 46.80	39.42 ± 8.30	0.11
MH02	Laptop, 20 fps	3,038 ± 3	19.12 ± 30.12	39.33 ± 9.21	0.11 ± 0.01
	Jetson, 20 fps	2,419 ± 22	24.23 ± 27.48	37.49 ± 10.04	0.10 ± 0.01
	Laptop, 0.5 fps	3,040	26.87 ± 40.43	39.75 ± 8.95	0.09
	Jetson, 0.5 fps	3,039	19.60 ± 31.87	39.46 ± 8.82	0.12
MH03	Laptop, 20 fps	2,699 ± 3	17.69 ± 22.87	38.65 ± 10.43	0.15 ± 0.02
	Jetson, 20 fps	2,114 ± 12	28.61 ± 36.90	33.28 ± 11.74	0.14 ± 0.01
	Laptop, 0.5 fps	2,700	20.20 ± 25.18	38.92 ± 10.23	0.10
	Jetson, 0.5 fps	2,700	31.53 ± 52.45	38.71 ± 9.86	0.14
MH04	Laptop, 20 fps	1,679 ± 702	32.45 ± 65.42	0.15 ± 1.40	96.38 ± 48.16
	Jetson, 20 fps	2,026 ± 2	38.12 ± 61.09	0.29 ± 2.43	134.28 ± 8.39
	Laptop, 0.5 fps	2,032	21.73 ± 18.63	0.03 ± 0.36	73.68
	Jetson, 0.5 fps	2,032	32.80 ± 48.10	0.44 ± 2.82	145.90
MH05	Laptop, 20 fps	2,271 ± 1	32.67 ± 50.20	21.22 ± 20.53	34.70 ± 42.30
	Jetson, 20 fps	2,176 ± 191	32.72 ± 49.00	4.84 ± 12.74	59.31 ± 31.22
	Laptop, 0.5 fps	2,273	22.94 ± 26.47	38.46 ± 10.74	0.17
	Jetson, 0.5 fps	2,273	32.61 ± 44.16	0.00 ± 0.00	77.78
V101	Laptop, 20 fps	2,912 ± 1	21.42 ± 31.37	44.00 ± 6.48	0.05 ± 0.00
	Jetson, 20 fps	2,080 ± 12	23.95 ± 26.43	39.05 ± 9.13	0.06 ± 0.01
	Laptop, 0.5 fps	2,912	18.24 ± 17.89	44.29 ± 5.96	0.05
	Jetson, 0.5 fps	2,912	18.00 ± 18.03	44.16 ± 6.59	0.05
V102	Laptop, 20 fps	1,710 ± 0	23.58 ± 20.92	34.20 ± 10.75	0.05 ± 0.00
	Jetson, 20 fps	1,387 ± 5	23.88 ± 19.56	25.62 ± 11.79	0.06 ± 0.01
	Laptop, 0.5 fps	1,709	21.82 ± 19.31	34.31 ± 10.83	0.05
	Jetson, 0.5 fps	1,709	22.45 ± 20.16	34.40 ± 10.91	0.06
V103	Laptop, 20 fps	2,149 ± 1	40.35 ± 71.17	25.91 ± 13.86	0.07 ± 0.01
	Jetson, 20 fps	1,792 ± 10	35.34 ± 49.38	18.95 ± 13.00	0.06 ± 0.01
	Laptop, 0.5 fps	2,149	26.31 ± 20.20	26.06 ± 13.81	0.06
	Jetson, 0.5 fps	2,149	29.85 ± 30.16	25.90 ± 13.84	0.06
V201	Laptop, 20 fps	2,278 ± 1	24.65 ± 36.21	42.23 ± 10.51	0.05 ± 0.00
	Jetson, 20 fps	1,661 ± 13	30.46 ± 51.83	37.26 ± 12.15	0.05 ± 0.00
	Laptop, 0.5 fps	2,279	27.42 ± 40.77	42.39 ± 10.54	0.05
	Jetson, 0.5 fps	2,279	23.44 ± 28.15	41.92 ± 11.25	0.06
V202	Laptop, 20 fps	2,347 ± 1	25.54 ± 25.15	32.48 ± 11.99	0.05 ± 0.01
	Jetson, 20 fps	1,892 ± 20	30.48 ± 42.62	25.05 ± 11.42	0.05 ± 0.01
	Laptop, 0.5 fps	2,348	29.66 ± 46.47	32.53 ± 12.07	0.04
	Jetson, 0.5 fps	2,348	24.29 ± 17.54	32.58 ± 11.93	0.04
V203	Laptop, 20 fps	1,920 ± 1	32.32 ± 38.28	18.64 ± 14.13	0.11 ± 0.00
	Jetson, 20 fps	1,737 ± 6	32.92 ± 40.07	14.10 ± 11.82	0.11 ± 0.02
	Laptop, 0.5 fps	1,921	32.69 ± 37.11	18.05 ± 13.79	0.09
	Jetson, 0.5 fps	1,921	32.52 ± 38.86	18.27 ± 13.86	0.12

of Local Mapping. In Table 7, we report the ATE of these poses as "ATE (Final)." Additionally, Loop Closing uses KeyFrame features from the Tracking module to compute a bag-of-words representation of the KeyFrame, which is entered into the visual place recognition database and used for loop detection. If a loop is detected, then the full pose graph mentioned earlier is connected at the loop and optimized again. Information inside the place recognition database does not need to be propagated earlier into the pipeline, because it is not used anywhere else.

OpenVINS Design. OpenVINS runs in a single-thread and thus also has no concurrency. If it is necessary to parallelize OpenVINS to meet timeliness goals, then the pipeline could be split after the "Tracking" stage in Figure 1. This split could be done without introducing concurrency, because OpenVINS, like Kimera, does not use the map for tracking. The OpenVINS map is most similar to Kimera's VIO Back-End map. It is a factor graph containing KeyFrames and SLAM landmarks as *optimizable variables* and IMU measurements and MSCKF features as *constraining factors* for a sliding window of 11 frames. SLAM landmarks and MSCKF features are tracked in the current window and marginalized out when lost.

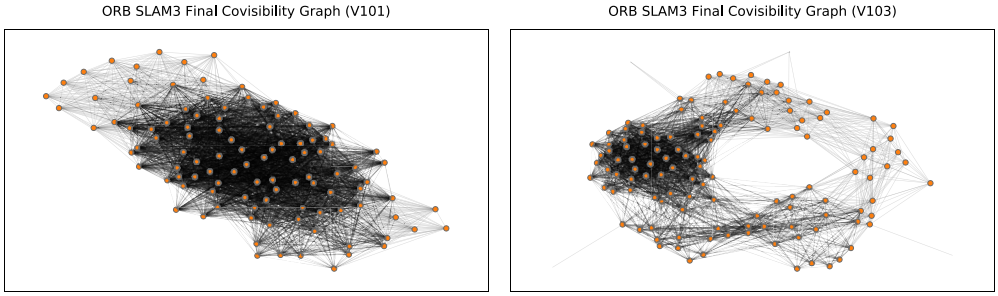
Downsides. The aforementioned system designs avoid concurrency challenges but come with several downsides. First, neither Kimera nor OpenVINS can build on an existing, previous map, or use it for localization-only mode, because Tracking does not use a map for localization. Second, while neither system suffers from additional concurrency-related errors, we found that Kimera and OpenVINS do not perform as well as ORB-SLAM3 in the non-resource-constrained case (Tables 6 and 7). This is in line with prior research [43] that finds BA-based SLAM systems tend to be more accurate than filtering-based approaches. Last, neither system generates an optimized map that can be used later by other SLAM systems or applications.

5.3 Viability of Fine-grained Concurrency

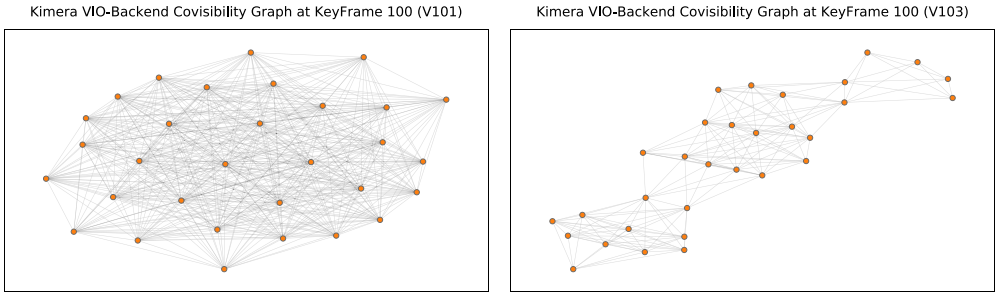
With fine-grained concurrency, we can reduce the number of drops and skipped optimizations by allowing modules to run concurrently if they do not modify the same subsection of the map. In this section, we discuss the possibility for fine-grained concurrency given the current structure of the shared global map in ORB-SLAM3.

The ORB-SLAM3 map is composed of five data structures: KeyFrames, MapPoints, the place recognition database, the spanning tree, and the covisibility graph (Figure 1(c)). The latter two are generated from KeyFrames and MapPoints to speed up several computations. The *covisibility graph* describes the KeyFrame connections—each KeyFrame is a node and an edge exists between two KeyFrames if they observe the same MapPoints. When Tracking inserts a new KeyFrame, the covisibility graph is updated to include the new KeyFrame and connect it to the appropriate, older KeyFrames.

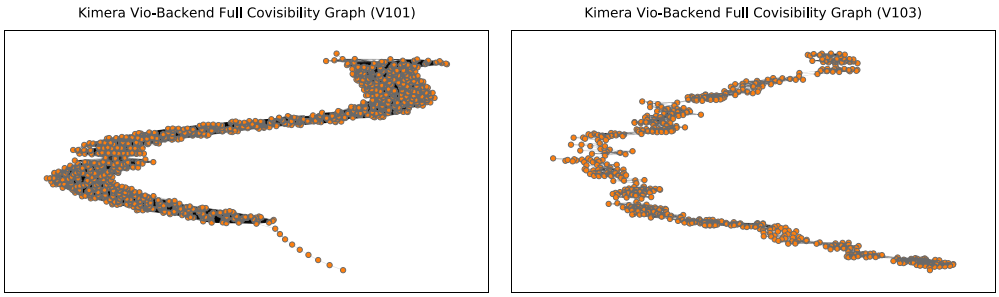
Figure 6(a) shows the ORB-SLAM3 covisibility graphs for two EuRoC sequences—V101 "easy," with slower movement and easier tracking, and V103 "difficult," with faster movement and fewer features to track. The graph is denser when the device sees many features in the environment and travels at a reasonable speed without little/slow rotational movement, and sparser in circumstances with more difficult tracking. This density can likewise be seen in Kimera's maps, even though it does not have a global map. Figure 6(b) shows the covisibility graphs for Kimera's Local Mapping module in the middle of the sequences. Because Kimera marginalizes out Landmarks and KeyFrames outside the time horizon, the size of this graph remains constant throughout execution. However, the density of the graph does not change throughout execution. This can be seen in Figure 6(c), which shows the covisibility graphs at the end of two sequences if the system did not marginalize out any KeyFrames. This graph contains no loops, because the Local Mapping module in Kimera is not aware of loop closures.



(a) ORB-SLAM3. The full global map that is generated by the end of the sequence.



(b) Kimera. The covisibility graphs in Local Mapping in the middle of the sequences.



(c) Kimera. The covisibility graph at the end of the sequence *had no information been marginalized out.*

Fig. 6. The final covisibility graphs for the EuRoc V101 (easy) and V103 (hard) sequences for Kimera and ORB-SLAM3. Graph nodes (KeyFrames) are connected if they share visible MapPoints/Landmarks.

Thus, graph density is a function of KeyFrame selection criteria more so than the underlying pipeline or map design in the SLAM system. For systems with a global map such as ORB-SLAM3, this opens an opportunity to increase concurrency through fine-grained locking or other granular synchronization mechanisms.

6 The Context Challenge

SLAM systems must perform computationally intensive operations on high-frequency sensor data streams to achieve acceptable accuracy at a target frame rate. As the mobile device traverses a physical environment, these operations are not done in a vacuum—SLAM systems must react to real-world conditions, which can be highly variable and unpredictable. For example, throughout the course of execution, the mobile device may start and stop (change velocities), take turns or go straight (rotational vs. translational movement), and enter environments with a high or low density

of features. These changing conditions are the *application context*—external (i.e., in the real world) and internal (i.e., prior and current system state) conditions that underlie the operation of the SLAM system.

Variations in application context lead to changing workloads for the SLAM system and therefore drastically affect the operation of the system and the importance of each module/task. For example, regions with feature sparsity and/or fast device movement necessitate timely tracking (regardless of whether KeyFrames are made) more than feature-rich and slow regions do. Likewise, missing many frames in a row heightens the urgency of processing the next frame, so relocalization does not occur and the map is sufficiently dense. Additionally, prior literature [9, 38], as well as our empirical evidence, shows that some KeyFrames are more important to accuracy and overall execution than others. For example, KeyFrames made during feature-sparse regions, fast device movement, and rotational movement (corresponding to a greater change in scene) all have fewer tracked MapPoints and therefore lower connectivity with the rest of the map. If the system chooses to drop a KeyFrame whose connectivity is low, then it might lead to a loss of tracking and trigger relocalization, whereas dropping a KeyFrame from a dense region might not affect the overall function or localization accuracy.

The key takeaway from these observations is that *application context is very influential on the overall function of the system*. Therefore, we cannot treat every concurrent access equally. Depending on context, the system needs to variably prioritize KeyFrame addition, map optimization, or loop closure. Despite this, current SLAM systems use the standard OS scheduler to allocate processing time and indiscriminately drop data that may affect performance and accuracy to maximize throughput of all modules. Instead, SLAM systems would benefit from a more "intelligent" way to incorporate real-world, real-time conditions into their decision-making, so *the right task* is being performed at *the right time*.

Being able to make intelligent, context-aware resource allocation decisions benefits SLAM systems in two scenarios. First, depending on the amount of available resources (CPU, memory, etc.), some devices will encounter an inevitable degree of overload when running a SLAM system. For example, in our experiments, all three SLAM systems are overloaded on the Jetson at 20 fps—ORB-SLAM3's Tracking module is very overloaded, and ORB-SLAM3's Local Mapping Module, Kimera's Tracking module, and OpenVINS are all slightly overloaded (Tables 2(a), 3(a), and 4). Second, due to the concurrency challenges described earlier, resource-constrained devices will necessarily need to drop more data compared to resource-rich devices, even if they are not overloaded. In both circumstances, the SLAM system *must* drop some data and tasks, so it would be better to drop those that are less crucial for performance.

The impact of *not* taking context into account can be seen in Table 9. All three SLAM systems display inconsistent average trajectory errors when evaluated on the same dataset multiple times, because indiscriminate data drops lead to a higher variability in performance. This variability is seen in both devices but is especially pronounced in the Jetson, because higher resource constraints lead to more data drops, increasing the likelihood of "bad" data drops. This variability makes it difficult to rely on SLAM systems, because they do not perform consistently on the same hardware and do not predictably and consistently degrade with lowered resources.

Including knowledge about the external, physical environment and the generated map into scheduling decisions will have three effects. First, for all devices, context-aware scheduling will lower the performance variability and lead to more predictable results. More predictable results, in turn, will mitigate the effects of overloading on resource-constrained devices by allowing them to gracefully and reliably degrade their performance. Last, resource-rich devices will be able to avoid unnecessary tasks that do not affect accuracy so they can run at a faster frame rate, have

Table 9. Consistency of ATE across Multiple Runs of Each Sequence on the Jetson (J) and Laptop (L) at the Target Frame Rate (20 fps)

	OpenVINS		Kimera		ORB-SLAM3			OpenVINS		Kimera		ORB-SLAM3	
Seq	J	L	J	L	J	L	Seq	J	L	J	L	J	L
MH01	0.08	0.08	0.70	0.48	0.25	0.02	V102	0.07	0.05	0.13	0.07	0.13	0.01
	0.08	0.10	0.70	0.62	0.02	0.03		0.06	0.05	0.08	0.09	6.25	0.01
	0.11	0.10	0.78	0.49	0.45	0.03		0.06	0.05	0.09	0.08	10.79	0.01
	0.08	0.10	0.69	0.49	10.85	0.02		0.07	0.05	0.09	0.09	0.01	0.01
	0.12	0.10	0.69	0.49	2.33	0.02		0.06	0.05	0.09	0.05	0.56	0.01
MH02	0.10	0.11	0.10	0.10	0.25	0.04	V103	0.06	0.07	0.19	0.14	3.41	0.02
	0.12	0.11	0.10	0.09	7.93	0.04		0.06	0.07	0.18	0.11	0.03	0.02
	0.09	0.11	0.05	0.09	1.60	0.04		0.05	0.06	0.19	0.16	0.03	0.02
	0.12	0.09	0.10	0.09	0.05	0.04		0.07	0.07	0.19	0.09	0.17	0.02
	0.09	0.11	0.10	0.10	0.35	0.03		0.07	0.07	0.18	0.09	0.13	0.02
MH03	0.15	0.12	0.19	0.08	2.76	0.03	V201	0.06	0.05	0.03	0.04	0.02	0.03
	0.13	0.16	0.18	0.20	0.76	0.03		0.06	0.05	0.04	0.03	0.62	0.03
	0.12	0.16	0.19	0.11	0.82	0.03		0.06	0.05	0.04	0.03	0.08	0.02
	0.14	0.16	0.13	0.12	0.61	0.03		0.05	0.05	0.04	0.06	0.15	0.02
	0.15	0.16	0.19	0.11	1.20	0.03		0.05	0.05	0.03	0.03	6.22	0.03
MH04	142.93	122.06	0.09	0.17	0.20	0.04	V202	0.06	0.04	0.09	0.09	0.17	0.01
	121.93	122.06	0.08	0.11	0.06	0.04		0.05	0.04	0.08	0.09	1.00	0.01
	144.40	0.19	0.08	0.11	0.58	0.05		0.05	0.06	0.10	0.08	8.40	0.05
	130.22	122.06	0.08	0.10	0.06	0.05		0.05	0.06	0.04	0.09	6.37	0.01
	131.92	115.52	0.08	0.10	0.23	0.04		0.05	0.06	0.07	0.10	0.24	0.01
MH05	81.72	0.13	0.12	0.25	0.30	0.05	V203	0.12	0.11	0.17	0.19	0.06	0.02
	78.82	0.18	0.12	0.21	0.44	0.04		0.08	0.11	0.17	0.21	0.03	0.03
	0.29	0.18	0.12	0.24	1.71	0.04		0.11	0.11	0.19	0.19	0.03	0.23
	54.43	86.52	0.12	0.24	0.07	0.05		0.13	0.11	0.18	0.15	0.07	0.05
	81.29	86.52	0.13	0.18	0.25	0.04		0.10	0.11	0.18	0.19	0.13	0.04
V101	0.05	0.06	0.05	0.12	0.03	0.03							
	0.06	0.05	0.04	0.04	0.19	0.03							
	0.05	0.05	0.04	0.05	0.03	0.03							
	0.07	0.05	0.10	0.05	0.04	0.03							
	0.05	0.05	0.05	0.04	0.04	0.03							

less expensive physical hardware, and/or free up computational time for additional on-device applications.

7 Conclusion and Future Directions

7.1 A Summary of Findings

We have highlighted three systems challenges that affect real-world deployment of Visual SLAM systems based on an analysis of the performance of ORB-SLAM3, Kimera, and OpenVINS—three SLAM systems with similar pipelines but very different map designs, concurrency challenges, optimization techniques, and performance. The challenges faced by SLAM systems can be summarized as follows: All SLAM systems have strict *timeliness* requirements with significant ramifications to accuracy if missed. Implementing more accurate and robust algorithms increases the accuracy of the system, but usually at the cost of increasing latency and complicating *concurrency* design. This

Table 10. Summary of Performance of All Systems

System	Latency	Robustness to tracking loss	Accuracy (best case)	Accuracy (difficult sequences)	Accuracy (resource-constrained)
ORB-SLAM3	High	High	Best	Best	Worst
Kimera	Medium	Low	Worst	Average	Average
OpenVINS	Low	Low	Average	Worst	Best

impacts timeliness and causes indiscriminate data drops in resource-constrained devices, which lowers accuracy. To some extent, resource contentions are unavoidable, but the worst of their effects can be mitigated by *context awareness*.

Table 10 illustrates this relationship for all three systems. We found that ORB-SLAM3 performs the most work of the three systems, followed by Kimera, followed by OpenVINS. In general, performing less work results in *lower* accuracy in typical non-resource-constrained cases ("Accuracy (best case)") and difficult non-resource-constrained cases like those with fast movement, loop closures, and few features ("Accuracy (difficult sequences)"). However, systems with less work perform relatively *better* than high-work systems in resource-constrained cases ("Accuracy (resource-constrained)"), because less work results in fewer resource contentions, so the system experiences fewer drops.

Three design decisions in ORB-SLAM3 greatly improve its accuracy in non-resource-constrained scenarios but negatively impact latency. First, ORB-SLAM3 is the only system to use a global map for tracking and mapping, which achieves greater accuracy but introduces concurrency. Second, ORB-SLAM3 implements a robust tracking recovery mechanism that is less prone to indefinite tracking loss but takes much longer to run. Last, ORB-SLAM3 optimizes KeyFrames *and* MapPoints in Local Mapping and Loop Closing, which increases accuracy but takes much longer than the KeyFrame-only optimization in Kimera and OpenVINS. For all these reasons, *ORB-SLAM3 has best performance with ample resources, but the worst performance under resource constraints.*

Kimera and OpenVINS do not use a global map, implement less robust tracking recovery mechanisms, and implement smaller map optimizations. All these design decisions negatively impact accuracy but also greatly lower latency, so they are able to meet timeliness goals even under resource constraints. *Kimera has moderate results on all sequences and all hardware, making its design a good choice for unknown or varied use-cases and hardware.* Additionally, OpenVINS does not implement Loop Closing, which frees up a lot of computational resources for the other modules. *OpenVINS' minimal implementation has the best performance in resource-constrained cases, but the worst performance in "normal" cases, especially when loop closure is needed.*

Based on these observations, we propose three design modifications that we believe will help SLAM systems achieve state-of-the-art performance in resource-rich scenarios *and* better, consistent performance in resource-constrained scenarios. First, decreasing module processing time will allow the system to meet timeliness goals even under resource constraints. Second, improving shared access to the global map will allow the system to achieve better accuracy without introducing unfeasible amounts of concurrency. Third, incorporating context-aware priorities into scheduling tasks will allow the system to drop nonessential information in circumstances where drops are unavoidable. When implementing these or any other large changes to the SLAM system design, it is important to be mindful that increasing work (e.g., adding frames, KeyFrames, and/or MapPoints, performing larger optimizations, using a global map) increases accuracy *only when*

there are enough computational resources to meet timeliness goals. Below, we detail each of these suggestions for future research.

7.2 Decreasing Processing Time

In our experiments, we found that *all three systems* fail to meet timeliness goals for their Tracking modules at 20 fps on resource-constrained devices, causing frame drops. Additionally, increased rates of concurrency on resource-constrained devices cause module execution to be skipped (via KeyFrame drops) for systems with global maps. Failure to meet timeliness goals has multiple ramifications—a decrease in accuracy due to IMU-only tracking, a temporary loss of map building, a delay in optimization, and possibly a complete and irrecoverable loss of tracking. Some of these ramifications are much more impactful to ATE than others, and their impact additionally depends on their context. While all three systems challenges are interconnected, addressing the problem of timeliness is a good first approach to tackling subpar and inconsistent performance on resource-constrained devices. Below, we detail a few options for addressing timeliness by increasing module throughput. Whichever techniques one uses, it is imperative to consider the overall system design, because increasing the throughput of one module can likewise increase concurrency.

Task Parallelization. ORB-SLAM3 and Kimera already parallelize the SLAM pipeline, which helps the system as a whole meet timeliness goals. An additional parallelization that would help lower frame drops is splitting the Tracking module into two threads—one that performs feature extraction on a frame, and another that performs feature matching and localization. We have tested this previously with ORB-SLAM2 and found that we could *nearly double* the number of tracked frames, because feature extraction takes around half the entire duration of Tracking. For single-threaded system designs like OpenVINS, splitting the system into two threads (Tracking and Local Mapping) could similarly dramatically increase throughput, since its tracking portion alone can easily keep up with 20 fps. Another simple option could be to use the parallelization techniques built into external libraries, such as OpenCV, GTSAM, and g2o.

Hardware Acceleration. Another way to decrease processing time is to offload tasks to dedicated hardware. The OpenCV library offers CUDA GPU acceleration for some vision tasks, which is compatible with the Jetson TX2 but only affects a small amount of tasks in the entire SLAM system. Some recent work [20, 29, 48] focuses on delegating other parts of the SLAM pipeline to the GPU. Incorporating multiple of these techniques into one system could drastically increase throughput. Last, the performance on the Jetson TX2 in particular might suffer from using an ARM-based processor. OpenCV offers Tengine-based acceleration for ARM but this is for deep learning tasks only, and g2o and GTSAM do not have similar modules. Customizing the SLAM system for performance on ARM is another potential solution to the timeliness problem.

7.3 Improving Shared Access to the Global Map

The joint challenges of concurrency and timeliness can be improved by providing better shared access to the global map. There are several approaches to this problem. The first solution is providing finer resolution locking on the global map to allow for concurrent access to non-overlapping regions of the global map. A second approach is the incorporation of concepts from concurrent data structures [32] that could provide an alternative to locking as a mechanism for shared access. A third solution is to create multiple thread-local versions [30] of the map and incorporate an on-demand mechanism to synchronize between these versions.

Concurrent Data Structures. Lock-based shared memory control falls prey to three problems. First, numerous and course-grained locks introduce sequential bottlenecks if only one part of the

system can make meaningful work at one time. Second, locks introduce overheads to accessing memory by increasing memory contention on the underlying hardware. Third, delayed threads that hold a lock block other threads from making progress. Work on concurrent data structures addresses some or all of the above problems [32]. Fine-grained locking can reduce memory contention and sequential bottlenecks but fails for programs with many concurrent accesses to the same part of the underlying data structure. Data structures like the *combining tree* are a viable solution to this problem but easily lead to deadlocks if designed incorrectly. An alternative technique, ***software transactional memory* (STM)**, converts a thread's multiple memory modifications into one atomic operation, allowing the system to forego locks entirely and simplify program development [22–24, 42].

Multi-version Concurrency Control. Another solution to the problem of concurrent long-running queries over quickly changing data structures is to allow threads to modify *local* versions of the shared memory, then "commit" those versions to a global version when all edits have been completed. Within this paradigm, threads also "update" their local versions when the global version has been updated, typically either at programmer-specified synchronization points or by default at synchronization primitives. This memory model allows for concurrent accesses to "shared memory" segments without the additional timing overheads inherent with traditional synchronization techniques like mutual exclusion, as threads can perform concurrent work on their local copies without needing to synchronize with other threads.

Multi-version concurrency control (MVCC) implements this architectural pattern but primarily with databases as a use case [6, 37]. However, some recent work has focused on in-memory MVCC for main memory segments [26, 30, 35]. Two considerations need to be taken when applying MVCC to SLAM systems. First, the implementation of the versioning system should prevent needless copying, since the map is a large data structure. Second, the conflict resolution strategy for multiple threads trying to commit to the global data structure needs to be domain-specific *and* responsive to real-world conditions. Typical conflict resolution strategies define an atomic "unit" as a minimum amount of bytes that need to be committed together, take all non-conflicting units between the two versions, and default to the thread with priority for resolving conflicts. However, for SLAM applications, the correct unit size and the prioritized thread may change, depending on program and real-world conditions.

7.4 Context-aware Priorities

A third problem to be solved is the dynamic changes in priority based on application context. While we observe this problem in Visual SLAM, we conjecture that it can be observed more broadly in sensing and control applications where the sensing affects the overall performance indirectly through inaccurate plans for control (e.g., autonomous driving, where inaccurate maps could lead to inefficient or dangerous paths). In all these contexts, the priority of processing sensor data (in our case, the Tracking module and its creation of KeyFrames) is context-dependent. Therefore, there needs to be a mechanism for applications to provide feedback about runtime priority to the scheduling to achieve efficient execution.

Real-time Sensing. Some recent work in real-time sensing incorporates real-time conditions to make scheduling decisions that improve accuracy, as defined on the application-level. Reference [31] considers the problem of sensor inferences in redundant multi-device environments such as a person using a phone and wearing a smartwatch and earbud simultaneously. Each device has unique characteristics that affect inference accuracy and vary due to environmental conditions. Given this, their system uses runtime information to select the device with the best inference accuracy. LEO [19] minimizes energy use of multiple concurrent sensor apps by monitoring runtime

information and varying the placement of the underlying algorithms across different hardware (CPU, co-processor, GPU, the cloud). ApproxDet [45], NestDNN [15], and MCDNN [21] work with multiple vision CNN/DNNs with various resource-accuracy tradeoffs and use real-time information about resource contention to make informed decisions about which model to use.

Deterministic and Stable Multithreading. A cause of program nondeterminism in traditional, synchronization-based multithreaded programs is the lack of enforcement over possible *schedules* (thread interleavings). Work on **deterministic multithreading (DMT)** approaches nondeterminism by effectively enforcing a subset of allowable schedules [4, 5, 28]. However, these approaches tend to either incur high latency overheads or fail in the case of data races. More recently, work on **Stable Multithreading (StableMT)** foregoes the focus on determinism in lieu of *stability*, which they define as robustness against input and program variations [46]. In StableMT, each allowable schedule is reused on a wide variety of inputs, thus drastically reducing the set of schedules and decreasing overhead. However, as discussed previously, even the overhead associated with StableMT is not necessary if the goal is to achieve responsiveness to real-time, real-world conditions.

References

[1] Brian D. O. Anderson and John B. Moore. 2012. *Optimal Filtering*. Courier Corporation.

[2] Ali J. Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. 2020. Edge-SLAM: Edge-assisted visual simultaneous localization and mapping. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys'20)*. Association for Computing Machinery, New York, NY, USA, 325–337. DOI: https://doi.org/10.1145/3386901.3389033

[3] Ali J. Ben Ali, Marziye Kouroshli, Sofiya Semenova, Zakieh Sadat Hashemifar, Steven Y. Ko, and Karthik Dantu. 2022. Edge-SLAM: Edge-assisted visual simultaneous localization and mapping. *ACM Trans. Embed. Comput. Syst.* 22, 1, Article 18 (Oct. 2022), 31 pages. DOI: https://doi.org/10.1145/3561972

[4] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic process groups in DOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, USA, 177–191.

[5] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe multithreaded programming for C/C++. *SIGPLAN Not.* 44, 10 (Oct. 2009), 81–96. DOI: https://doi.org/10.1145/1639949.1640096

[6] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion concurrency control–theory and algorithms. *ACM Trans. Datab. Syst.* 8, 4 (1983), 465–483.

[7] Dominik Brodowski, Nico Golde, Rafael J. Wysocki, and Viresh Kumar. 2023. CPU Frequency and Voltage Scaling Code in the Linux Kernel. Retrieved from: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt#:~:text=The"schedutil"governoraimsat,therecentload[1]

[8] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W. Achtelik, and Roland Siegwart. 2016. The EuRoC micro aerial vehicle datasets. *The International Journal of Robotics Research* 35, 10 (2016), 1157–1163.

[9] Alvaro Parra Bustos, Tat-Jun Chin, Anders Eriksson, and Ian Reid. 2019. Visual SLAM: Why bundle adjust? In *Proceedings of the International Conference on Robotics and Automation (ICRA'19)*. IEEE, 2385–2391.

[10] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José M. M. Montiel, and Juan D. Tardós. 2021. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics* 37, 6 (2021), 1874–1890.

[11] Timothy Chase, Ali J. Ben Ali, Steven Y. Ko, and Karthik Dantu. 2022. PRE-SLAM: Persistence reasoning in edge-assisted visual SLAM. In *Proceedings of the IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS'22)*. 458–466. DOI: https://doi.org/10.1109/MASS56207.2022.00071

[12] Rodrigo Chaves, Paulo Rezeck, and Luiz Chaimowicz. 2019. SwarMap: Occupancy grid mapping with a robotic swarm. In *Proceedings of the 19th International Conference on Advanced Robotics (ICAR'19)*. IEEE, 727–732.

[13] NVIDIA Corporation. 2019. Power Management for Jetson TX2 Series Devices. Retrieved from https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/TegraLinuxDriverPackageDevelopmentGuide/power_management_tx2_32.html

[14] Linux Kernel Documentation. 2017. Intel Pstate CPU Performance Scaling Driver. Retrieved from https://www.kernel.org/doc/html/v4.19/admin-guide/pm/intel_pstate.html

[15] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. 115–127.

[16] Christian Forster, Luca Carlone, Frank Dellaert, and Davide Scaramuzza. 2016. On-manifold preintegration for real-time visual–inertial odometry. *IEEE Trans. Robot.* 33, 1 (2016), 1–21.

[17] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are we ready for autonomous driving? The KITTI vision benchmark suite. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR'12)*.

[18] Patrick Geneva, Kevin Eckenhoff, Woosik Lee, Yulin Yang, and Guoquan Huang. 2020. OpenVINS: A research platform for visual-inertial estimation. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

[19] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. 2016. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. 320–333.

[20] Shishir Gopinath. 2023. Improving the performance of bundle adjustment for on-device SLAM using GPU resources. (2023). https://ieeexplore.ieee.org/document/10160499

[21] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 123–136.

[22] Tim Harris and Keir Fraser. 2014. Language support for lightweight transactions. *ACM SIGPLAN Not.* 49, 4S (2014), 64–78.

[23] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A practical multi-word compare-and-swap operation. In *Proceedings of the International Symposium on Distributed Computing*. Springer, 265–279.

[24] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*. 92–101.

[25] Manon Kok, Jeroen D. Hol, and Thomas B. Schön. 2017. Using inertial sensors for position and orientation estimation. *Found. Trends® Signal Process.* 11, 1-2 (2017), 1–153.

[26] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *arXiv preprint arXiv:1201.0228* (2011).

[27] Mingyang Li and Anastasios I. Mourikis. 2013. High-precision, consistent EKF-based visual-inertial odometry. *Int. J. Robot. Res.* 32, 6 (2013), 690–711. DOI: https://doi.org/10.1177/0278364913481251 .

[28] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. Association for Computing Machinery, New York, NY, USA, 327–336. DOI: https://doi.org/10.1145/2043556.2043587

[29] Tianji Ma, Nanyang Bai, Wentao Shi, Xi Wu, Lutao Wang, Tao Wu, and Changming Zhao. 2021. Research on the application of visual SLAM in embedded GPU. *Wirel. Commun. Mob. Comput.* 2021 (2021), 1–17.

[30] Timothy Merrifield and Jakob Eriksson. 2013. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. Association for Computing Machinery, New York, NY, USA, 127–139. DOI: https://doi.org/10.1145/2465351.2465365

[31] Chulhong Min, Alessandro Montanari, Akhil Mathur, and Fahim Kawsar. 2019. A closer look at quality-aware runtime assessment of sensing models in multi-device environments. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. 271–284.

[32] Mark Moir and Nir Shavit. 2018. Concurrent data structures. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 741–762.

[33] Anastasios I. Mourikis and Stergios I. Roumeliotis. 2007. A multi-state constraint Kalman filter for vision-aided inertial navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*. 3565–3572. DOI: https://doi.org/10.1109/ROBOT.2007.364024

[34] Raúl Mur-Artal and Juan D. Tardós. 2017. ORB-SLAM2: An open-source SLAM system for monocular, stereo and RGB-D cameras. *IEEE Trans. Robot.* 33, 5 (2017), 1255–1262. DOI: https://doi.org/10.1109/TRO.2017.2705103

[35] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 677–689.

[36] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. 2011. KinectFusion: Real-time dense surface mapping and tracking. In *Proceedings of the 10th IEEE International Symposium on Mixed and Augmented Reality*. 127–136. DOI: https://doi.org/10.1109/ISMAR.2011.6092378

[37] Christos H. Papadimitriou and Paris C. Kanellakis. 1984. On concurrency control by multiple versions. *ACM Trans. Datab. Syst.* 9, 1 (1984), 89–99.

[38] Christian Pirchheim, Dieter Schmalstieg, and Gerhard Reitmayr. 2013. Handling pure camera rotation in keyframe-based SLAM. In *Proceedings of the IEEE International Symposium on Mixed and Augmented Reality (ISMAR'13)*. 229–238. DOI: https://doi.org/10.1109/ISMAR.2013.6671783

[39] Antoni Rosinol, Marcus Abate, Yun Chang, and Luca Carlone. 2020. Kimera: An open-source library for real-time metric-semantic localization and mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'20)*. IEEE, 1689–1696.

[40] D. Schubert, T. Goll, N. Demmel, V. Usenko, J. Stueckler, and D. Cremers. 2018. The TUM VI benchmark for evaluating visual-inertial odometry. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS'18)*.

[41] Sofiya Semenova, Steven Y. Ko, Yu David Liu, Lukasz Ziarek, and Karthik Dantu. 2022. A quantitative analysis of system bottlenecks in visual SLAM. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*. 74–80.

[42] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distrib. Comput.* 10, 2 (1997), 99–116.

[43] Hauke Strasdat, José M. M. Montiel, and Andrew J. Davison. 2012. Visual SLAM: Why filter? *Image. Vis. Comput.* 30, 2 (2012), 65–77.

[44] Jingao Xu, Hao Cao, Danyang Li, Kehong Huang, Chen Qian, Longfei Shangguan, and Zheng Yang. 2020. Edge assisted mobile semantic visual SLAM. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'20)*. 1828–1837. DOI: https://doi.org/10.1109/INFOCOM41043.2020.9155438

[45] Ran Xu, Chen-lin Zhang, Pengcheng Wang, Jayoung Lee, Subrata Mitra, Somali Chaterji, Yin Li, and Saurabh Bagchi. 2020. ApproxDet: Content and contention-aware approximate object detection for mobiles. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 449–462.

[46] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. 2014. Making parallel programs reliable with stable multithreading. *Commun. ACM* 57, 3 (2014), 58–69.

[47] Lintong Zhang, Michael Helmberger, Lanke Frank Tarimo Fu, David Wisth, Marco Camurri, Davide Scaramuzza, and Maurice Fallon. 2023. Hilti-Oxford dataset: A millimeter-accurate benchmark for simultaneous localization and mapping. *IEEE Robot. Autom. Lett.* 8, 1 (2023), 408–415. DOI: https://doi.org/10.1109/LRA.2022.3226077

[48] Dhruv Kumar, Shishir Gopinath, Karthik Dantu, and Steven Y. Ko. 2024. JacobiGPU: GPU-Accelerated numerical differentiation for loop closure in visual SLAM. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 1687–1693. DOI: https://doi.org/10.1109/ICRA57147.2024.10611512

Received 4 September 2023; revised 4 September 2023; accepted 7 May 2024