A Quantitative Analysis of System Bottlenecks in Visual SLAM

Sofiya Semenova*
University at Buffalo
sofiyase@buffalo.edu

Steven Y. Ko
Simon Fraser University
steveyko@sfu.ca

Yu David Liu
SUNY Binghamton
davidl@binghamton.edu

Lukasz Ziarek
University at Buffalo
lziarek@buffalo.edu

Karthik Dantu
University at Buffalo
kdantu@buffalo.edu

Abstract

Visual SLAM systems are concurrent, performance-critical systems that respond to real-time environmental conditions and are frequently deployed on resource-constrained hardware. Previous SLAM frameworks have primarily focused on *algorithmic* advances and their *systems* core has largely remained unchanged. In turn, SLAM systems suffer from performance problems that could be alleviated with improved systems design. In this paper, we present a quantitative analysis of the systems challenges to building consistent, accurate, and robust SLAM systems in the face of concurrency, variable environmental conditions, and resource-constrained hardware. We identify three interconnected challenges on systems design — *timeliness*, *concurrency*, and *context awareness* – and clarify their effects on performance.

CCS Concepts

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computing methodologies** → *Vision for robotics.*

ACM Reference Format:
Sofiya Semenova, Steven Y. Ko, Yu David Liu, Lukasz Ziarek, and Karthik Dantu. 2022. A Quantitative Analysis of System Bottlenecks in Visual SLAM. In *The 23rd International Workshop on Mobile Computing Systems and Applications (HotMobile '22), March 9–10, 2022, Tempe, AZ, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3508396.3512882

1 Introduction

Several classes of applications such as mobile augmented reality and autonomous driving require a 3-D map of the environment for accurate functioning. Simultaneous Localization and 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 last decade, several Visual SLAM systems have been proposed to improve map and localization accuracy, such as KinectFusion [9], ORB-SLAM2 [8], ORB-SLAM3 [3], OpenVINS [5] and Kimera [11]. 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 [1, 12] tasks to edge servers for performance. However, the view of SLAM frameworks as *performance-critical software systems* is not systematically explored. This is unfortunate because SLAM algorithms are most commonly deployed on software/hardware ecosystems with constrained resources and stringent performance requirements. In this paper, we counterpose the algorithm-oriented SLAM research with a systems-oriented perspective. We identify a number of systems challenges that cause suboptimal operations of SLAM frameworks, based on our years of experimentation. 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 are 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. To avoid localization error (and failure), 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. Most popular SLAM systems split the pipeline into *concurrent* modules that correspond to the tasks listed above. However, these modules perform computationally heavy operations which frequently and primarily access, refine, and add to the shared data structures that comprise the global map. Current SLAM systems manage 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

*Contact author
†This project is sponsored by NSF Awards CNS-1823260, CNS-1823230, CNS-1846320 and SHF-1749539.

HotMobile '22, March 9–10, 2022, Tempe, AZ, USA

ACM ISBN 978-1-4503-9218-1/22/03…$15.00
https://doi.org/10.1145/3508396.3512882

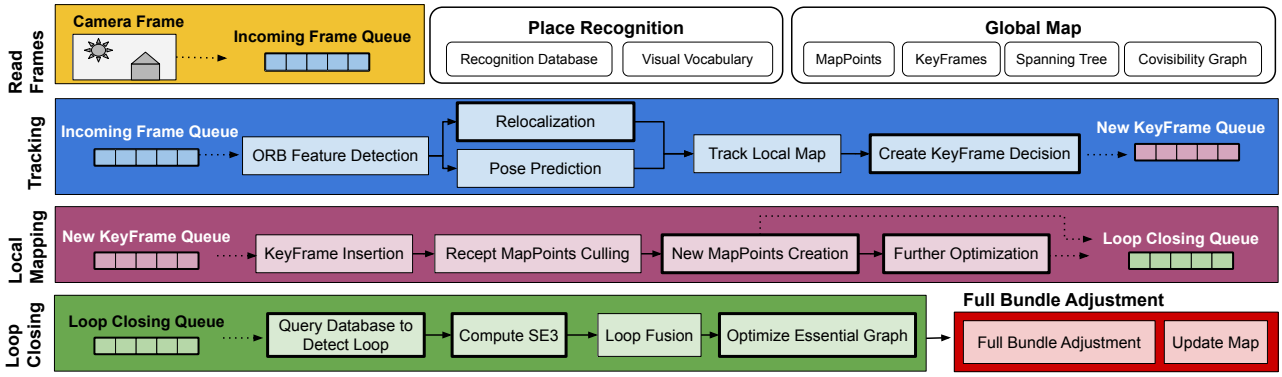

Figure 1: ORB-SLAM2 System Architecture. Each module (shown in a different color) is implemented as a separate thread. The visual vocabulary and recognition database are pretrained. All modules interact with the Global Map.

unpredictable results. As we observed, poorly designed concurrent access to the global map data structure is a major bottleneck in achieving performance scalability.

Context Awareness. Further, the execution of SLAM systems *as a whole* are dependent on the conditions of the real world, which are highly variable and unknowable in advance. These conditions include device velocity, whether movement is rotational or translational [2, 10] , and the novelty of the device's current location in the context of the entire explored 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 in order 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 that *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.

To the best of our knowledge, this paper is the first work to quantitatively and systematically demonstrate and analyze systems challenges faced in SLAM systems. It is our hope that this work can serve as a bridge, inspiring computer systems researchers to improve SLAM design for efficient operation. As a key theme, this paper attempts to elucidate the interconnected relationship among the three challenges we identified earlier. In a nutshell, both *concurrency* and *context awareness* have significant impact on *timeliness*, which in turn affects accuracy of function.

2 Experimental Setup

All data reported in this paper result from experiments with ORB-SLAM2 [8] on two devices. The first device is an Nvidia Jetson TX2 (Dual-Core NVIDIA Denver and Quad-Core ARM Cortex-A57 CPUs, 8GB Memory), which we use as the resource-constrained mobile device. The second device is a System76 Galago Pro laptop (Dual-Core Intel Core i7-7500U CPU, 32GB Memory), which is representative of a service robot navigating inside a building. Both devices run Ubuntu 18.04 LTS.

We used three datasets — two indoor sequences collected on our campus (Jarvis and Bell) and the 00 sequence of the KITTI dataset [4]. The Jarvis and Bell sequences primarily consist of slow, linear movement of a Turtlebot-2 through an indoor environment with occasional angular movement, and run at 15fps. The KITTI dataset consists of faster, varied vehicular movement through an outdoor environment (city streets), and runs at 10fps.

3 Visual SLAM Overview

In this section, we will describe the functionality of a typical Visual SLAM system. For this, we will refer mostly to ORB-SLAM2, a popular open-source Visual SLAM system that has been in use for a few years. While our focus is on ORB-SLAM2, most modern SLAM systems follow a similar architecture and design for basic mapping and localization. However, some newer systems provide additional functionality, such as semantic mapping in Kimera and multi-map reasoning in ORB-SLAM3. To this end, our description captures the essential functions of a basic Visual SLAM system.

A Visual SLAM pipeline consists of three modules – *Tracking*, *Local Mapping*, and *Loop Closing* – all performing work on the *map*. Figure 1 shows the ORB-SLAM2 architecture. Each module executes as a separate thread and all have shared access to the global map.

The *Tracking* module is responsible for reading the images from an input video stream, extracting ORB features from each image, matching these features to corresponding ones in the current *KeyFrame*, and using these correspondences to determine the odometry with regard to the KeyFrame. Additionally, if the percentage of correspondences is low, it adds the extracted features as a new KeyFrame in the map. If this decision is made, it sends the extracted features to *Local Mapping* for further processing. *This module should run on every image received from the camera.*

The *Local Mapping* module is responsible for further optimization of the map given an input KeyFrame by: culling recently-created MapPoints that are not visible from a significant number of KeyFrames, creating new MapPoints by triangulating features from KeyFrames connected to the new KeyFrame, performing *Local Bundle Adjustment* to optimize the poses of the new KeyFrame, its connected KeyFrames, and all MapPoints observed by them, and finally culling redundant KeyFrames. *Local Mapping runs for every added KeyFrame to the global map. This occurs roughly once every 6-10 images received from the camera on our dataset.*

Table 1: Effective execution times of each of the modules on the KITTI dataset, for two different frame rates. *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. Relocalization and Loop Closing do not occur periodically.

	Jetson, 10 fps			**Laptop, 10 fps**		
Module	Input Frequency	Throughput	Duration	Input Frequency	Throughput	Duration
Tracking	10 fps	9 ± 1 fps	103 ± 21 ms	10 fps	9.5 ms ± 0.7 fps	41 ± 17 ms
Relocalization	-	-	178 ± 90 ms	-	-	71 ± 23 ms
Local Mapping	1.1 ± 1.1 KFps	1.1 ± 1.1 KFps	417 ± 235 ms	2.7 ± 1.7 KFps	2.7 ± 1.7 KFps	205 ± 103 ms
Loop Closing	-	-	1306 ms	-	-	858 ± 202 ms
	Jetson, .5 fps			**Laptop, .5 fps**		
Module	Input Frequency	Throughput	Duration	Input Frequency	Throughput	Duration
Tracking	0.5 fps	0.5 fps	138 ± 221 ms	0.5 fps	0.5 fps	69 ± 46 ms
Relocalization	-	-	-	-	-	-
Local Mapping	0.3 ± 0.5 KFps	0.3 ± 0.5 KFps	1757 ± 772 ms	0.4 ± 0.5 KFps	0.4 ± 0.5 KFps	566 ± 301 ms
Loop Closing	-	-	3611 ± 6147 ms	-	-	4367 ± 1322 ms

The final module, *Loop Closing*, searches for loops for every inserted KeyFrame. When a loop is detected, it finds the accumulated drift in the loop, aligns both sides of the loop, fuses duplicate points, and optimizes the *pose graph* by evenly distributing the accumulated drift across the length of the loop. Lastly, it spawns a temporary thread to perform full bundle adjustment to further optimize the map. *Loop closing occurred once in the 30min dataset on the Jetson and twice on the laptop for the target frame rate (10fps).*

All described modules run in a continuous loop on input from their respective queues. In Figure 1, within each module, the solid arrows indicate the next task, dotted arrows indicate enqueue and dequeue operations, and bolded tasks indicate terminal states, after which the module will start again on new data.

4 The Timeliness Challenge

The nature of the SLAM workload is sequential — control flows from Tracking to Local Mapping to Loop Closing. 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. 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.

Timely Frame Processing. The tracking thread is responsible for reading incoming video stream data, localizing the incoming frame against the map, 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, 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. This is a catastrophic event and triggers *relocalization*, a separate, computationally more intensive process that puts the mapping on hold while it attempts to reacquire the device location in the current map. Since mapping is put on hold, relocalization leads to a further loss in map building. Such a cascading effect completely derails the SLAM process on a resource-constrained device. Given these observations, it is very important for the device to manage its resources to avoid relocalization at all costs.

Our experiments show that Tracking takes an average of 103ms on the Jetson and 41ms on the laptop when run at the target frame rate (Table 1, top). A video stream at 10fps generates an image every 100ms. However, Tracking run at a very low frame rate (.5 fps) takes an average of 138ms on the Jetson and 69ms on the laptop (Table 1, bottom). The increased duration results from the relationship between available resources, map size, and module execution length. When run with a large amount of resources, SLAM systems are able to create more KeyFrames and MapPoints, which increases the map size, which increases the search space for tasks such as feature matching.

Timely Map Optimization. The Local Mapping module optimizes the map after each KeyFrame created in Tracking, primarily by culling redundant KeyFrames and MapPoints and modifying the relative pose of MapPoints to minimize overall error. Ideally, Local Mapping executes on every Keyframe added to the global map, making the map as efficient and accurate as possible. If this module is behind on processing, the Tracking and Loop Closure modules would interact with an un-optimized map, which has three ramifications. First, without culling redundant KeyFrames and MapPoints, all other modules will operate on a larger map, taking longer to accomplish the same task. Second, without an optimized map, the accuracy of the map suffers. Third, without the creation of new MapPoints, Tracking has a higher chance of entering relocalization.

Our experiments show that Local Mapping takes 417ms on average on the Jetson and 205ms on the laptop when run at the target frame rate. When run at the low frame rate, Local Mapping takes 1757ms on the Jetson and 566ms on the laptop. This is because a larger *and denser* map leads to a larger search space for optimization tasks, as well as more map data that needs to be optimized.

Timely Loop Detection and Closure. The Loop Closing module detects and closes loops in the trajectory to correct accumulated trajectory drift. It is difficult to specify the timeliness requirements of the Loop Closure module as its effects are indirect. Executing as

Table 2: The tracked images, created KeyFrames, dropped KeyFrames, and ATE (average trajectory error) for the four experiments in Table 1. The sequence contains 4539 images.

Experiment	Tracked Images	Created KeyFrames	Dropped KeyFrames	ATE
Jetson, 10 fps	4240	532	652	15.71m
Laptop, 10 fps	4529	1400	733	7.41m
Jetson, .5 fps	4532	2933	399	5.38m
Laptop, .5 fps	4532	3731	10	6.56m

soon as a loop is discovered results in a globally optimized, efficient map that minimizes error. However, execution of this module locks the map data structure and halts the execution of Tracking and Local Mapping, which has ramifications for overall performance as described in those modules. 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.

Loop Closing is significantly more computationally expensive than the previous two modules: 1306ms on the Jetson and 858ms on the laptop for the target frame rate. For the low frame rate, the module took 1757ms on the Jetson and 566ms on the laptop because a larger and denser map leads to a larger search space for loop closures and more KeyFrames and MapPoints that need to be modified if a loop is detected.

5 The Concurrency Challenge

As described earlier, the SLAM modules are implemented in three concurrently executing threads and need to frequently access the shared global map. Shared memory access is managed using locks, allowing the modules sequential access to the global map. Table 1 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-SLAM2 was designed to drop or minimally process data rather than wait on lock acquisitions, in order to aid each module in meeting its timeliness goals. However, 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, we observe that more resource-constrained environments suffer from higher ATE (average trajectory error) and more relocalizations, which we attribute to an increased amount of KeyFrame drops due to concurrency (Table 2). In this section, we will discuss the relationship between decreased performance, data drops, and resource contention.

Frame Drops. Each module picks up work from its queue when it is ready to process the next input. The Tracking queue only contains the latest frame that arrived from the camera. Thus, a frame drop occurs when multiple frames arrive while Tracking is busy. Our experiments show that the laptop at 10fps, the Jetson at .5fps, and the laptop at .5fps track roughly the same amount of images (Table 2). The Jetson at 10fps tracks significantly fewer images because it experiences an increased amount of relocalizations, during which no images are tracked at all. However, *before* any relocalizations, the Jetson at 10fps and the laptop at 10fps track nearly the same amount of images: 1054 and 1057, respectively. While Tracking runs

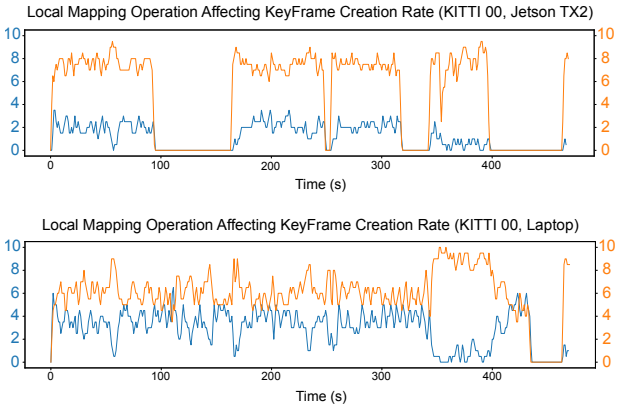

Figure 2: Left (blue) axis indicates rate of KeyFrame creation, while the right (orange) axis indicates rate of Keyframe drop due to concurrency. A higher blue line and lower orange line indicates better performance.

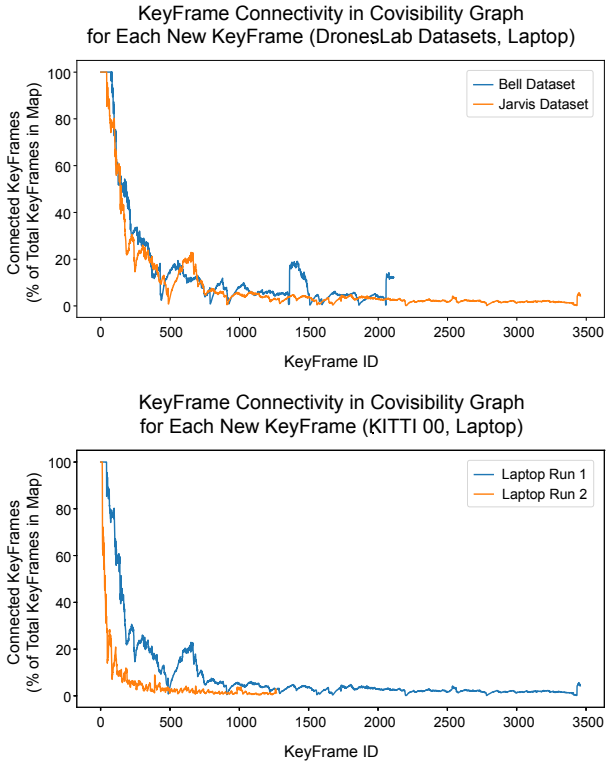

Figure 3: The KeyFrame connectivity in the covisibility graph for each new KeyFrame, as a percentage of the total KeyFrames.

at a slower rate on the Jetson than the laptop, both devices can run the Tracking task faster than the rate of incoming images, so very few images are dropped. The key takeaway is that, while frame drops *can* contribute to lowered mapping and localization results, they do not contribute to the lower accuracy and more frequent relocalizations seen in our experiments.

KeyFrame Drops. The Local Mapping module has an unbounded queue. To avoid processing stale data and falling behind real-time,

the queue length is mitigated in two ways. First, Local Mapping will skip additional optimizations ("New MapPoints Creation" and "Further Optimization" in Figure 1) if there are any items in its queue. This has ramifications for the accuracy (due to a suboptimal map) and timeliness (due to a larger map) of localization and mapping results. Second, because map optimization modifies a large part of the map, ORB-SLAM2 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, and we identify it as the primary effect of concurrency on performance in resource-constrained devices.

In the first 100 seconds (before any relocalizations), the Jetson creates **232** KeyFrames and the laptop creates **372** KeyFrames, despite both devices tracking a similar amount of frames in that time. The difference in the number of created KeyFrames is because the Jetson experiences more KeyFrame drops. Figure 2 illustrates the rate at which Tracking inserts a KeyFrame (left axis, blue) and the rate at which KeyFrames are dropped due to resource contention with Local Mapping (right axis, orange), for the *entire* KITTI 00 sequence for both devices. Areas of the graph where the KeyFrame creation rate goes to 0 indicate that the device is in relocalization mode. Taken together, the gap between the two rates indicates the extent to which resource contention affects KeyFrame creation.

Impact of Loop Closure. Loop Closing also has an unbounded queue, and the other two modules will check that it is not running before they attempt to process incoming data. The effect of these drops are limited because the duration of Loop Closing is very short (< 5 ms) when a loop is not found and loops are found infrequently. However, Loop Closing is a much more computationally intensive process than the other two and will execute for much longer. Thus, the amount of drops and skipped optimizations would be heightened when a loop closure occurs.

5.1 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.

The global map is concurrently accessed by all three SLAM modules, with a high frequency of reads and writes and a wide variety of short to long-range queries. It is comprised of four data structures: KeyFrames, MapPoints, the spanning tree, and the covisibility graph (Figure 1). 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 3 show the percentage of all the KeyFrames that an inserted KeyFrame is connected to (i.e., *the KeyFrame connectivity*) for two DronesLab datasets and the KITTI dataset. As the map grows, the percentage of KeyFrames the incoming KeyFrame connects to reduces, which is expected. Figure 4 shows the covisibility graph for the KITTI dataset. The KeyFrames are denser when the device

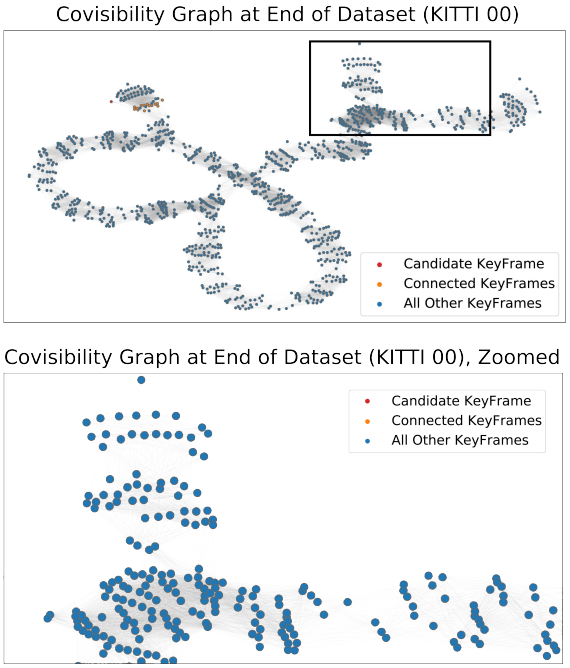

Figure 4: The covisibility graph at the end of the dataset, and a zoomed-in portion of the same graph.

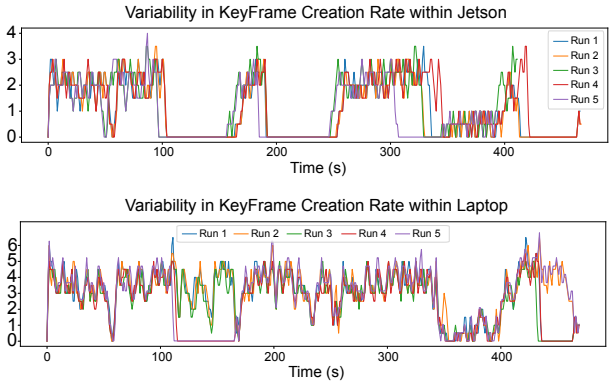

Figure 5: Variability in the KeyFrame creation rate *within* a device for the KITTI dataset.

is traveling at a reasonable speed without turns, and connectivity is sparser when there are turns or faster movement. 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 quickly-arriving input data and achieve acceptable accuracies at a target frame rate. 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, Table 1 (top) shows that the Tracking and Local Mapping modules are slightly overloaded for the Jetson at 10fps. Additionally, 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.

Table 3: Average Trajectory Error for 5 runs of the KITTI dataset for the Jetson and laptop, both at 10fps.

Jetson	15.71 ± 8.56 m	17.38 ± 8.39 m	18.94 ± 9.22 m
	21.45 ± 9.07 m	16.66 ± 8.72 m	
Laptop	7.41 ± 3.16 m	6.44 ± 3.53 m	7.54 ± 3.45 m
	17.41 ± 7.8 m	7.14 ± 2.88 m	

Both circumstances lead to higher variability in performance. Figure 5 shows the variability in Keyframe creation rate in five runs on both the Laptop at 10fps and the Jetson at 10fps. While both platforms exhibit variability, it is more pronounced on the Jetson. The resultant localization and map accuracies (Table 3) also vary highly, especially when relocalization is triggered in some runs. For both devices, the degree of variability makes it difficult to rely on SLAM systems, since lowering or raising device resources does not predictably lower or raise performance. Ideally, the system should gracefully degrade with lowered resources.

Further, our experiments at the very low frame rate suggest that simply increasing the amount of available resources (either by using a faster device or lowering the frame rate) might not be the ideal solution to unpredictable performance and/or overload. For very low frame rates where the system has an ample amount of time to perform all computations, ORB-SLAM2 takes *longer* to run each module than in cases where the system needs to drop data (Table 1, bottom). This is because lower resource contention leads to fewer KeyFrame drops, resulting in a much larger and denser map. A map with more KeyFrames and MapPoints creates a larger search space for all tasks, and a denser map with higher KeyFrame connectivity means that more of the map will need to be optimized and operated on during each task. The resulting ATEs for the low frame rate experiments are also not significantly better than the ATE for the laptop at 10fps, despite such a dramatic frame rate reduction (Table 2). This suggests that the low frame rate experiments are doing an abundance of tasks that do not significantly increase performance. If the unimportant tasks could be eliminated, the entire system could run at a faster frame rate or with worse hardware.

Including context into scheduling decisions addresses both the problem of reliable graceful degredation and the problem of culling unnecessary tasks in resource-rich devices. ORB-SLAM2, like most other SLAM systems, uses the standard OS scheduler to allocate processing time. However, the importance of each task (and the percentage of resources it should be allowed to use) changes depending on environmental conditions. Prior literature [2, 10], as well as our empirical evidence, shows that some KeyFrames are more important to accuracy and overall execution than others. This is related to how many features are seen in a given image, how fast the device is moving in the environment (which in turn affects how many common features it sees across images), and if the movement is rotational or translational (rotation results in greater changes in scene, corresponding to fewer overlapping features across images). This can be seen in the covisibility graph (Figure 4, bottom) where there is high variability in the connectivity of the graph. If we choose to drop a KeyFrame whose connectivity is low, it might lead to a loss in tracking and trigger relocalization, whereas deciding to drop 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.* Further, because timeliness is incredibly important for the system to achieve its best performance, small changes in scheduling have large and long-lasting effects. Therefore, we cannot treat every concurrent access equally. Depending on context, we need to variably prioritize KeyFrame addition, map optimization, or loop closure.

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

7 Future Directions

We have highlighted three systems challenges that affect real-world deployment of Visual SLAM systems based on an analysis of the performance of ORB-SLAM2. We believe that these challenges can be addressed through two design modifications: improving shared access to the global map and incorporating context-aware priorities into scheduling tasks.

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 [7] that could provide an alternative to locking as a mechanism for shared access. A third solution is to create multiple thread-local versions [6] of the map and incorporate an on-demand mechanism to synchronize between these versions.

Context-Aware Priorities. A second 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.

Further SLAM Characterization. Finally, we focused our analysis on ORB-SLAM2 and three sequences across two datasets. Most modern SLAM systems follow a similar architecture to ORB-SLAM2. Because of this, we believe our results will likely extend to other popular SLAM systems. However, future work could expand on this analysis to include additional SLAM systems and datasets.

References

[1] 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* (Toronto, Ontario, Canada) *(MobiSys '20)*. Association for Computing Machinery, New York, NY, USA, 325–337. https://doi.org/10.1145/3386901.3389033

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

[3] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. 2021. ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM. *IEEE Transactions on Robotics* (2021).

[4] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[5] Patrick Geneva, Kevin Eckenhoff, Woosik Lee, Yulin Yang, and Guoquan Huang. 2020. OpenVINS: A Research Platform for Visual-Inertial Estimation. In *Proc. of the IEEE International Conference on Robotics and Automation*. Paris, France.

[6] 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* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 127–139. https://doi.org/10.1145/2465351.2465365

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

[8] Raúl Mur-Artal and Juan D. Tardós. 2017. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. *IEEE Transactions on Robotics* 33, 5 (2017), 1255–1262. https://doi.org/10.1109/TRO.2017.2705103

[9] 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 *2011 10th IEEE International Symposium on Mixed and Augmented Reality*. 127–136. https://doi.org/10.1109/ISMAR.2011.6092378

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

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

[12] Jingao Xu, Hao Cao, Danyang Li, Kehong Huang, Chen Qian, Longfei Shangguan, and Zheng Yang. 2020. Edge Assisted Mobile Semantic Visual SLAM. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 1828–1837. https://doi.org/10.1109/INFOCOM41043.2020.9155438