On the Necessity of Real-Time Principles in GPU-Driven Autonomous Robots

Syed W. Ali[1*], Angelos Angelopoulos[1*], Denver Massey[1*], Sarah Haddix[2], Alexander Georgiev[2], Joseph Goh[1], Rohan Wagle[1], Prakash Sarathy[3], James H. Anderson[1], Ron Alterovitz[1]

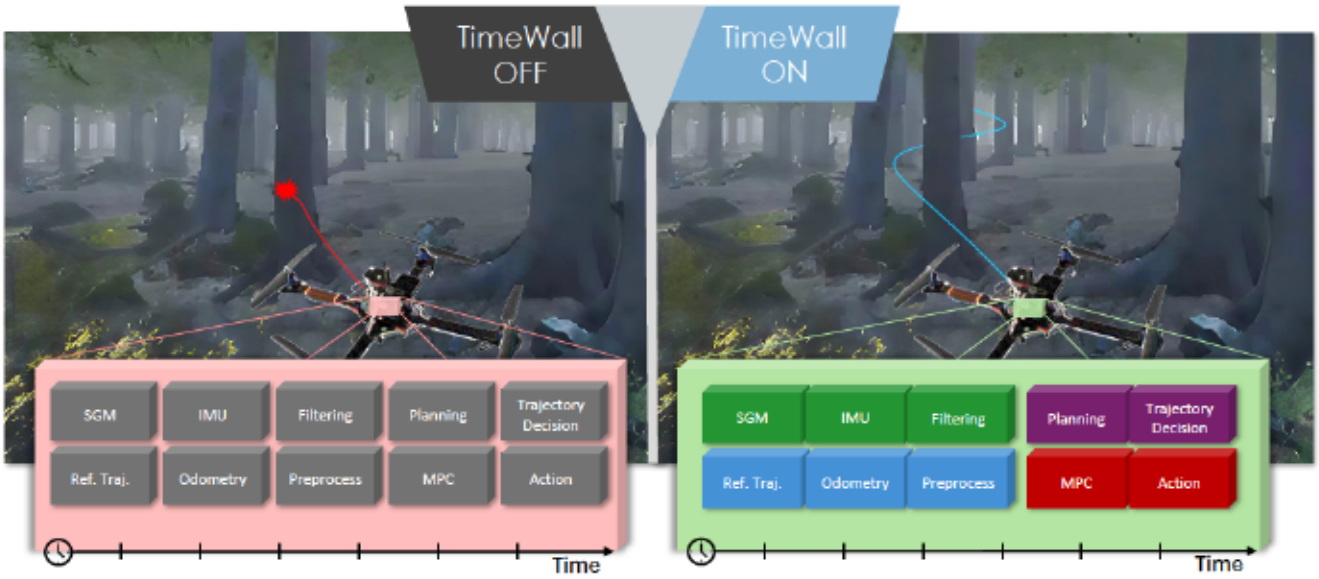

Fig. 1: Our real-time systems design approach is applied to a simulated drone autonomously navigating a forest at high speed. As interference from non-flight-critical workloads increases, the drone's reaction time becomes poorer, causing crashes. We apply TimeWall, our component-based real-time framework, to isolate and prioritize time-critical software components such as obstacle detection and control, ensuring the robot maintains its reaction times and avoids unnecessary crashes even as interference increases.

***Abstract*— Robot autonomy is driving an ever-increasing demand for computational power, including on-board multi-core CPUs and accelerators such as GPUs, to enable fast perception, planning, control, and more. Careful scheduling of these computational tasks on the CPU cores and GPUs is important to prevent locking up the finite computational capacity in ways that hinder other critical workloads; delays in computing time-critical tasks like obstacle detection and control can have huge negative consequences for autonomous robots, potentially resulting in damage, substantial financial loss, or even loss of life. In this paper, we leverage recent advances from real-time systems research. We apply TimeWall, a component-based real-time framework, to the computational components of an autonomous drone and experimentally show that the timeliness and safe operation properties of a drone are preserved even in the presence of increasing interfering computational processes.**

*Contributed equally.

†This work was supported by National Science Foundation (NSF) grants CPS 2038960, CPS 2038855, CNS 2151829, CPS 2333120, and RI 2008475.

[1,2]Department of Computer Science, University of North Carolina at Chapel Hill, NC 27599, USA

[1]{swali, aangelos, denmas22, jgoh, wagle, anderson, ron}@cs.unc.edu

[2]{sarahbh, ageorgiev}@unc.edu

[3]Northrop Grumman Aeronautics Systems, Redondo Beach, CA 90278, USA sriprakash.sarathy@ngc.com

I. Introduction

The quest for robot autonomy is driving an ever-increasing demand for computational power. As autonomous robots tackle increasingly sophisticated tasks in dynamic environments, the on-board compute hardware enabling their autonomy has grown complex in equal measure. These robots are incorporating a higher degree of heterogeneity in their compute hardware by employing a mixture of general-purpose multicore CPUs and hardware accelerators, commonly GPUs and FPGAs [1]. Accelerators are built to speed up certain computations, such as the linear algebra workloads required by modern AI applications, and are instrumental in pushing the boundaries of AI for robotics [2], [3].

Autonomous robots typically require running a variety of tasks, such as perception, planning, control, and mapping, on limited on-board compute hardware. Some tasks must be executed at high-frequency, like a perception module that detects dynamic obstacles or a controller that must generate the next robot control input. Other tasks may have lower frequency, like building maps or completing application-specific tasks. On shared hardware platforms, such tasks of varying criticality utilize the same on-board compute hardware.

Unfortunately, naïve utilization of CPU cores and GPUs can adversely affect the timely completion of tasks, diminishing reliability [4]. Tasks can experience undue scheduling delays when accessing busy compute hardware resources, like an on-board GPU, which increases application response times and creates potentially unsafe behavior. For example, if some unrelated subsystem of the robot utilizes a GPU required by a computer vision obstacle detection task, there may not be enough compute time available to identify a hazardous obstacle before the robot comes into contact with it [5]. The resultant violation of real-world timing constraints poses severe consequences for autonomous robots, potentially resulting in damage, substantial financial loss, or even loss of life.

In this paper, we leverage recent advances from the *real-time systems* research community, where the exact problem of guaranteeing bounded response times is a first-class concern. Such real-time systems encompass a *real-time scheduler* that ensures higher-priority work is guaranteed progress over lower-priority work. In a soft real-time system, code must complete within a bounded amount of time after its *deadline*. Scheduling tasks in a manner that satisfies deadlines can ensure high-frequency tasks like perception and control are completed in time to maintain a robot's safety and reliability. Additionally, *real-time locking protocols* ensure shared resource access is guaranteed within bounded time. A well-built real-time system is *predictable* and guarantees *timeliness*, ensuring that a robot's compute hardware completes its computational workloads on time even when the compute hardware is saturated.

We present a real-time approach toward predictable operation which assigns groups of related processes to modular *components*. We show how TimeWall [4], a component-based real-time framework, may be applied to robotics software executing on multicore+accelerator hardware platforms. Through the effective isolation of components, we realize the benefits of modularity, allowing components to be modified without invalidating the runtime assumptions made by other components.

We evaluate our approach on a high-speed drone navigation software stack [6], which we run in simulation (see Fig. 1). We task the drone to navigate through a forest using (1) the baseline implementation of the software [6] and (2) our modified implementation using a real-time scheduler and TimeWall with component partitioning. Both implementations are evaluated in the presence of interfering object detection workloads. We show that our modified real-time implementation is more stable and does not lead to increased drone collisions in the presence of increasing interference, contrary to the baseline implementation which breaks down and leads to many drone collisions.

II. Related Work

The Robot Operating System (ROS) is a component-based system framework widely used in robotics. Despite its ubiquity, ROS lacks proper real-time support. While ROS 2 has introduced features attempting to satisfy real-time requirements (*e.g.*, the rclc executor [7]), numerous shortcomings remain [8], [9].

Casini *et al.* [9] analyzed ROS 2 under a reservation-based scheduler and presented analysis to bound the end-to-end response times of processing chains (*i.e.*, source to sink). This work presents insights into ROS 2 execution, such as callback processing order and how executor callback ready queues are updated. This work also contributed the first steps towards developing an automated analysis tool compatible with the ROS 2 single-threaded executor. Such methods were later extended to the multi-threaded executor [10].

Blass *et al.* [11] proposed ROS-Llama, a user-level runtime system that automatically reduces the latency of user-specified processing chains. ROS-Llama inspects a ROS 2 system at runtime and dynamically adjusts scheduling parameters (*e.g.*, periods and execution budgets). Their experiments show that ROS-Llama obtains improved maximum latency under load when compared to the Linux CFS scheduler.

Choi *et al.* [12] proposed PiCAS, a user-level priority-driven scheduling method for ROS 2 which also reduces end-to-end processing chain latency by changing the execution order of callbacks. Some specific strategies employed are (1) non-timer callbacks later in the chain are given higher priority and (2) non-timer callbacks are prioritized over timer callbacks.

One common approach to improving GPU timeliness guarantees is the use of GPU servers, often called "inference servers" in the AI domain, to centralize GPU access [13]. GPU servers provide managed access to the GPU by queuing requests from clients to run GPU work. However, critical workloads still suffer when enqueued behind ongoing lower priority work that cannot be aborted (*e.g.*, GPU code where the cost of preemption may exceed the remaining computation time). As such, the application-granularity scheduling offered by GPU servers has limits. To ensure high-priority work can complete on time, every time, a finer degree of control is desirable.

Prior work fails to provide hardware isolation guarantees for ROS where concurrently running tasks cause interference and execution delays. ROS employs a component-based system architecture wherein software components are organized as functionally independent nodes. This component-based structure lends itself well to the component-based scheduling in TimeWall [4], discussed in the next section, where isolation guarantees are achieved.

III. Real-Time System Model

A *real-time system* is subject to critical real-world timing constraints. While often misconstrued as "real-fast" systems, real-time systems must guarantee that their workloads complete on time (*i.e.*, meet deadlines) even in worst-case scenarios, rather than optimizing only for average execution times. In this section, we introduce our component-based real-time system model which forms the core of our system design and resource management approach.

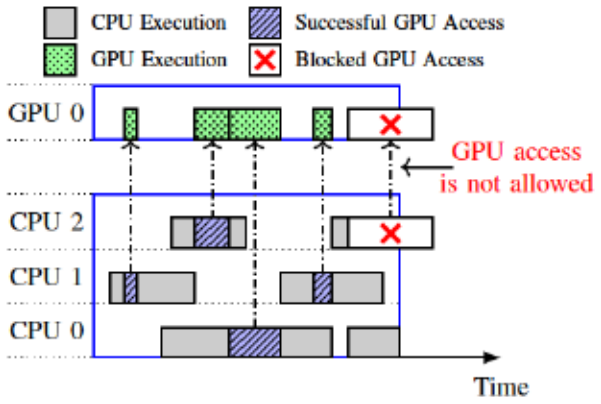

Fig. 2: Work running on CPUs requests time on the shared GPU, illustrated by the rising arrows. TimeWall postpones execution that extends past the component's time slice (outlined in blue), ensuring proper component isolation.

We begin by summarizing notation and terminology commonly used in real-time system analysis. A *task* is defined as periodically invoked work, with the i^{th} task denoted as τ_i. Tasks describe recurrent work such as sensor input processing, motion planning, and controller invocations. A single invocation of a task is called a *job*. Each task τ_i has three parameters, C_i, T_i, and D_i, expressing the *worst-case execution time* (WCET) of any of its jobs, the *period* separating job releases, and the *relative deadline* of each job, respectively. Each job of τ_i must be allowed an execution duration of up to C_i time units.

In *component-based* real-time systems, tasks are grouped into components where uncertainty is mitigated by ensuring each component has isolated access to its compute resources. While a divide-and-conquer approach to robot design is not new, a real-time component differs in that it provides analytical guarantees that the component will work reliably even when the rest of the system design changes. Such modularity is invaluable, as updating one component does not invalidate the real-time guarantees established in other components.

We define an arbitrary component $\Gamma = (\Theta, \Pi, \tau, \Upsilon)$. The set τ contains all tasks assigned to the component. The set Υ contains the resources utilized by τ, such as GPUs and processor cores. The parameters Θ and Π define a *periodic component reservation* (PCR) where Γ is guaranteed isolated access to the resources in Υ for an uninterrupted *time slice* duration of Θ. Lastly, Π denotes the exact time between the start of each time slice.

This component model is adopted from TimeWall (Time-Isolated Multicore Execution With AcceLerator Locking), a framework for multicore+accelerator platforms proposed by Amert *et al.* [4]. Such component-based frameworks are amenable to real-world avionics specifications such as ARINC 653 [14], which details real-time operating system design for enabling reliable access to compute hardware shared across tasks.

IV. Method

In this section, we detail how robotics software can be designed with real-time requirements in mind. First, we clearly describe the problems incurred by the lack of real-time scheduling guarantees. We then describe how existing robotics software design paradigms can be transformed to be compatible with real-time scheduling. Our approach is evaluated in the subsequent section by making the corresponding modifications to a ROS-based autonomous drone platform [6].

A. Problem Definition

Flight-critical functions of an autonomous drone include (1) ensuring that the drone stays in the air and (2) maintaining its integrity (*e.g.*, does not damage itself by colliding with obstacles). Such software may contain GPU code like neural networks and CPU code to process sensor data and output trajectory decisions. Lower-criticality features may be included with an autonomous drone software suite, for example, a drone performing a search or inspection task may perform object recognition. The computational demand of these additional tasks may result in a significant system-wide slowdown, affecting the performance of flight-critical components. Furthermore, the possibility of unexpected, disproportional computational demand stemming from software bugs must be accounted for.

We define all such examples beyond the core flight and safety-critical operation of the drone as sources of *interference*. Interference occurs when concurrently running workloads contend for shared resources (such as CPU caches or GPUs) and slow each other down. Interference can occur even when software functions are introduced without malicious intentions. Thus, any software changes may induce new interference, requiring that the entirety of the software stack be reevaluated to determine whether timing guarantees will be met.

Now consider a drone whose software design is motivated by real-time component-level isolation. Following the design specifications of TimeWall [4], the core functionality of the drone may be encapsulated in a *component* that is guaranteed temporally isolated access to resources such as GPUs and CPU cores while executing its flight-critical functions.

By guaranteeing an isolated runtime environment, any added or modified components do not invalidate the correct operation of other existing components. So long as non-critical features are allocated an isolated component, core functionality will remain intact due to the runtime isolation guarantees provided by TimeWall. Overly inefficient software that overutilizes system hardware can be detected by simply applying a real-time schedulability test that determines whether the system can reliably meet deadlines [15]. When compared to larger, monolithic software stacks, the critical portion of code is easier to harden against interference due to the more modular code base.

Hence, we formulate the following problem definition. Consider interference from f non-flight-critical processes. As f increases, critical work such as trajectory planning may

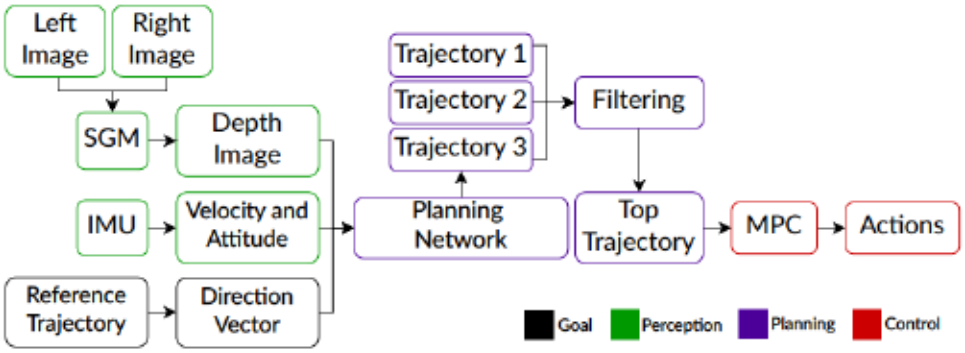

Fig. 3: Drone system functionality diagram. Planning functions are placed into the same component.

experience unwanted delays and subsequent failures. Using real-time system design principles, namely the TimeWall [4] component-scheduling framework, we wish to isolate critical work from such interference.

B. Real-Time Parameterization

In this section, we briefly discuss how the TimeWall framework [4] discussed in Sec. III can be integrated with existing robotics software such as [6]. For this work, we assume Global Earliest Deadline First (G-EDF) scheduling in a component where, intuitively, jobs with earlier deadlines are given higher priorities [16]. Additionally, we incorporate the OMLP [17], a real-time locking protocol that guarantees jobs exclusive access to resources, such as the GPU. Exclusive resource access is necessary for mitigating potential sources of interference for critical tasks [18], [19]. Such real-time locking protocols provide a bound on the duration a process can spend waiting to access a shared resource, denoted by X. We denote the maximum amount of time task τ_i spends holding exclusive access to a resource as B_i. The amount of time that TimeWall may delay τ_i's access to a resource (as illustrated in Fig. 2) is given by [15]:

$$b_i = X + \left\lceil \frac{X + B_i}{\Theta - B_i} \right\rceil \cdot B_i. \tag{1}$$

A component's task set τ may consist of various workloads, such as those of the drone system in Fig. 3. We define a real-time task as a timer or subscriber callback (*i.e.*, a ROS subscriber callback), and a job as a callback invocation. Let n be the number of such tasks. Each callback is now a real-time task τ_i, and therefore must be characterized by the parameters C_i, T_i, D_i. The component containing these callbacks must have a defined Θ and Π. This component must also be assigned a number of CPU cores, denoted as m. As long as the chosen values satisfy the following set of constraints, TimeWall will guarantee proper execution:

$$T_i \geq C_i + b_i, \tag{2}$$

$$\Theta \geq \max_i \{B_i\}, \tag{3}$$

$$1 \geq \frac{\Theta}{\Pi} \geq \frac{1}{m} \sum_{i=1}^{n} \frac{C_i + b_i}{T_i}. \tag{4}$$

The derivation of appropriate values for these parameters is determined by the response time requirements of the critical workload. Choosing too small of a frequency for a timer callback may result in the system being unable to respond to the environment, but too large of a frequency will overload the system. Moreover, if $\Pi - \Theta$ is too large, the component will experience extended intervals where it cannot access any compute resources, creating further delays.

We have now specified all necessary parameters for a component. Once these parameters have been set, isolation is guaranteed as TimeWall ensures processes in one component are unaffected by other components.

V. Evaluation

In this section, we describe the implementation of our proposed method in an autonomous drone software stack and present experimental results showing that TimeWall successfully isolates safety-critical workloads from interference.

A. Robot System Description

We build upon an existing end-to-end drone software stack developed by Loquercio *et al.* [6] which performs high-speed navigation with obstacle avoidance in unknown environments. The core of the navigation software is a neural network with the following inputs: (1) a 2D depth image generated via Semi-Global Matching (SGM) from a stereo camera, (2) the velocity and attitude (rotation) of the drone, and (3) a direction vector pointing to the closest point of a reference trajectory. The reference trajectory is a global trajectory from a start pose to a goal pose, which by default is a straight line. The outputs of the neural network are (1) three trajectories consisting of ten drone poses each and (2) a collision cost for each trajectory. A filtering step follows inference with the network. The trajectories are projected to an order-5 polynomial space to ensure continuity and dynamic feasibility. Then, the trajectory with the lowest predicted collision cost and input cost [20] is selected. The chosen trajectory is executed using an MPC controller. These functions are illustrated in Fig. 3.

We use neural network weights provided in [6], trained by Loquercio *et al.* on 90,000 samples using imitation learning, where a "privileged expert" with ground-truth knowledge of

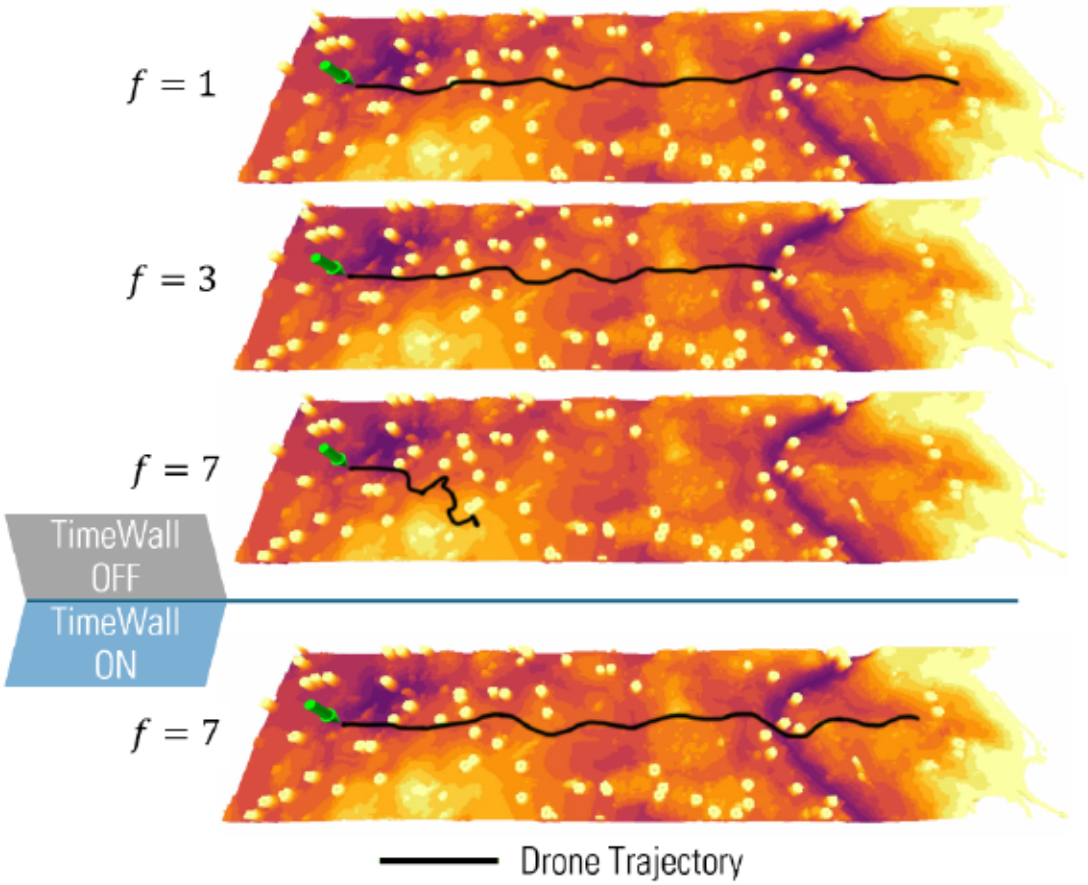

Fig. 4: We show representative simulations of the drone, starting at the green marker and moving along the black path toward a goal on the right while avoiding obstacles shown in the yellow/red depth map, operating under different interference levels. When increasing f, the number of interference processes, the drone typically crashes earlier. When using the TimeWall scheduler, the drone successfully executes its flight even under high levels of interference.

the environment (*i.e.*, full point cloud) predicts trajectories with Metropolis-Hastings sampling, which the network then learns to imitate. Training was performed in the Flightmare [21] simulator with custom environments containing randomly placed obstacles (*e.g.*, trees), the results of which were demonstrated to be transferable to the real world without fine-tuning [6]. Flightmare uses the Unity game engine for rendering and Gazebo with the RotorS [22] plugin for physics.

Our experiments were conducted on two Dell Precision 7920 2-socket motherboards with an eight-core 2.10GHz Intel Xeon Silver 4110 processor per socket. Both machines used LITMUS^{RT} [23], [24], a real-time testbed based on the Linux 5.4 kernel, with ROS 2 Humble Hawksbill. The neural network and interference workloads were run on one machine equipped with an NVIDIA Titan V GPU. The Flightmare simulation was run on the other machine, which was equipped with an NVIDIA GTX 1080 Ti GPU.

Modifications. We require all ROS nodes to be initialized with a unique callback queue [25]. Then, each timer and subscriber callback must be registered on its own node. Finally, a thread is created for each node, and that thread registers itself as a real-time task with the scheduler such that the thread is only scheduled when a job exists (*i.e.*, the callback can be invoked).

When a real-time thread is scheduled, the associated callback is serviced if it is ready in the callback queue. Afterward, the thread suspends until the next invocation specified by the task period T_i. In the case of a timer task, the callback is always serviced.

Because the real-time scheduler meticulously controls the scheduling of threads in the system, we found that Python inhibits the ease of translation from best-effort ROS to real-time. In `rospy` [26], each subscriber is given a thread that spins on the callback queue, but there does not exist an elegant way to convert the Python thread to a real-time task while also using real-time locking protocols for predictable resource access. Therefore, in our evaluation, we converted all Python code from [6] to C++ so that real-time threads and locks can be properly utilized.

The real-time locking protocols provided by TimeWall [15] are essential when working in component-based systems. Unlike non-real-time locking, TimeWall requires that the worst-case access duration is specified when a resource is locked. A job is only given access to that resource if no other job is accessing the resource and if the resource will be relinquished before the component is no longer scheduled,

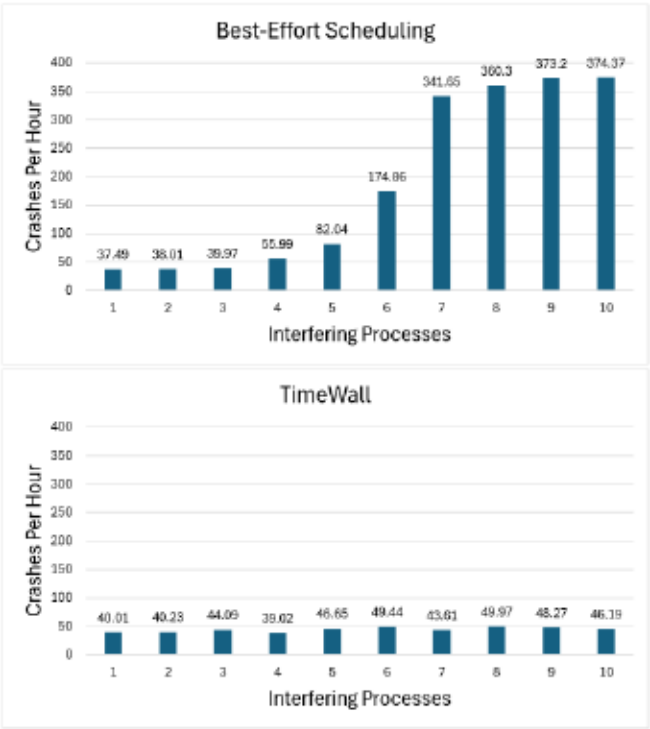

Fig. 5: Number of crashes per hour given f, the number of interfering processes. As interference increases, best-effort scheduling (top) does not mitigate unnecessary crashes. TimeWall (bottom), however, ensures flight-critical workloads meet their deadlines, preventing an increase in crashes.

as in Fig. 2. The latter requirement is necessary as GPU access must end before the start of another component that may require the GPU. Thus, jobs must lock a GPU before using it to ensure isolation guarantees across components.

B. Experimental Results

We created f GPU-using processes to contend with flight-critical workloads. Each interfering process performs inference with a separate instance of the neural network used by the drone at 15Hz. We executed our modified drone software alongside these interference processes. For each $f \in [1, 10]$, we performed approximately one hour of simulated flight under best-effort scheduling with the default Linux scheduler. These experiments were then repeated with TimeWall enabled. Shown in Fig. 4 are representative flights of the drone at various values of f.

If the drone travels far enough without crashing into trees, then the flight successfully finishes. If the drone crashes into a tree or the ground, the flight ends and a crash is recorded. The number of crashes per hour is derived by running flights and then normalizing the number of crashes observed to an hour of airtime. The results are plotted in Fig. 5.

Observation 1. As f increases, under best-effort scheduling, the drone becomes less capable of making timely decisions. Fig. 4 shows the drone failing to react to trees as trajectory decisions are delayed due to blocked and interrupted GPU accesses. Such delays caused trajectory decision tasks to miss deadlines where a deadline was implicitly defined as the point in time the next decision needed to be computed. The result is a marked increase in the number of crashes per hour as seen in the top graph of Fig. 5.

Observation 2. Increasing f has little effect when TimeWall arbitrates CPU execution and GPU access. Shown in the bottom graph of Fig. 5, the number of crashes per hour is stable and does not increase significantly as f increases. This is due to interfering processes being prevented from utilizing the GPU while the flight-critical components are guaranteed isolated access to it.

Observation 3. Increasing reliability in high-interference scenarios comes with a small performance cost. Because the overhead incurred by TimeWall's scheduling decisions is under $10\mu s$ [4], the capacity loss is instead due to the scheduling parameters of the component. When compared to best-effort scheduling, the periodic component reservation reduces the time spent executing to that specified by the reservation's period and time slice duration (Π and Θ respectively). The effects of such a reduction are shown where the crashes per hour in Fig. 5 under TimeWall is slightly higher when compared to a system under best-effort scheduling with few interfering processes. Additionally, the performance degradation can also be attributed to delaying GPU work at a component's time slice boundary, as shown in Fig. 2.

However, these effects can be mitigated by optimizing Θ and Π, where Θ can be increased to execute more often, and Π can be optimized around the critical workload's period to avoid attempting to execute in a forbidden zone. Any remaining capacity loss is justified given the observed benefits of greatly increased reliability with TimeWall's real-time guarantees. Such reliable execution times are instrumental in the quest for certifiably safe, GPU-using autonomous systems.

VI. Conclusion

We have demonstrated the importance and benefits of real-time system design principles for the safe operation of autonomous robots. Using TimeWall as an exemplar component-based real-time framework, we have described how actual robotics systems, particularly those utilizing multicore+accelerator platforms, may be modified to adhere to such principles while also enabling certifiable, modular system components.

We have verified the efficacy of our approach by implementing the proposed modifications in an actual autonomous drone software stack. Our experiments show that TimeWall guarantees timely, isolated access to compute resources for flight-critical workloads and thus prevents unnecessary crashes where a naïve implementation cannot.

Future work may explore how the modularity of TimeWall components may be customized and further enhanced for publish/subscribe systems commonly used in robotics. We also hope to validate our approach with a physical drone platform in the real world.

REFERENCES

[1] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris, "Robot Motion Planning on a Chip," in *Robotics: Science and Systems XII*. Ann Arbor, Michigan: Robotics: Science and Systems Foundation, June 2016. [Online]. Available: http://www.roboticsproceedings.org/rss12/p04.pdf

[2] N. Nayakanti, R. Al-Rfou, A. Zhou, K. Goel, K. S. Refaat, and B. Sapp, "Wayformer: Motion Forecasting via Simple & Efficient Attention Networks," Jul. 2022, arXiv:2207.05844 [cs]. [Online]. Available: http://arxiv.org/abs/2207.05844

[3] "Isaac ROS cuMotion — isaac_ros_docs documentation." [Online]. Available: https://nvidia-isaac-ros.github.io/repositories_and_packages/isaac_ros_cumotion/index.html

[4] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. D. Smith, and J. H. Anderson, "TimeWall: Enabling Time Partitioning for Real-Time Multicore+Accelerator Platforms," in *2021 IEEE Real-Time Systems Symposium*. Dortmund, DE: IEEE, Dec. 2021, pp. 455–468. [Online]. Available: https://ieeexplore.ieee.org/document/9622375/

[5] M. J. Slaby, "One dead after vehicle hits firetruck parked on I-70." [Online]. Available: https://www.indystar.com/story/news/2019/12/29/one-dead-after-tesla-hits-parked-fire-truck-70/2771593001/

[6] A. Loquercio, E. Kaufmann, R. Ranftl, M. Müller, V. Koltun, and D. Scaramuzza, "Learning high-speed flight in the wild," *Science Robotics*, vol. 6, no. 59, p. eabg5810, Oct. 2021, publisher: American Association for the Advancement of Science. [Online]. Available: https://www.science.org/doi/10.1126/scirobotics.abg5810

[7] W. Woodall, D. Thomas, E. Fernandez, J. Staschulat, R. Lange, J. B. Ortega, P. G. Sánchez, A. Cuadros, L. Dauphin, and B. Downing, "rclc." [Online]. Available: https://github.com/ros2/rclc

[8] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance," in *2021 IEEE Real-Time Systems Symposium*, Dec. 2021, pp. 41–53, iSSN: 2576-3172. [Online]. Available: https://ieeexplore.ieee.org/document/9622336/?arnumber=9622336

[9] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 6:1–6:23. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2019.6

[10] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors," in *2020 IEEE Real-Time Systems Symposium*, Dec. 2020, pp. 231–243, iSSN: 2576-3172. [Online]. Available: https://ieeexplore.ieee.org/document/9355523/?arnumber=9355523

[11] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, May 2021, pp. 264–277, iSSN: 2642-7346. [Online]. Available: https://ieeexplore.ieee.org/document/9470451/?arnumber=9470451

[12] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2021, pp. 251–263, iSSN: 2642-7346. [Online]. Available: https://ieeexplore.ieee.org/document/9470466/?arnumber=9470466

[13] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar, "A server-based approach for predictable GPU access control," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug. 2017, pp. 1–10, iSSN: 2325-1301. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8046309

[14] P. J. Prisaznuk, "ARINC 653 role in Integrated Modular Avionics (IMA)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, Oct. 2008, pp. 1.E.5–1–1.E.5–10, iSSN: 2155-7209. [Online]. Available: https://ieeexplore.ieee.org/document/4702770/?arnumber=4702770

[15] S. Voronov, S. Tang, T. Amert, and J. H. Anderson, "AI Meets Real-Time: Addressing Real-World Complexities in Graph Response-Time Analysis," in *2021 IEEE Real-Time Systems Symposium*. Dortmund, DE: IEEE, Dec. 2021, pp. 82–96. [Online]. Available: https://ieeexplore.ieee.org/document/9622398/

[16] J. W. S. W. Liu, *Real-Time Systems*, 1st ed. USA: Prentice Hall PTR, 2000.

[17] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 49–60.

[18] N. Otterness and J. H. Anderson, "Exploring amd gpu scheduling details by experimenting with "worst practices"," in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, ser. RTNS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 24–34. [Online]. Available: https://doi.org/10.1145/3453417.3453432

[19] J. Bakita and J. H. Anderson, "Demystifying nvidia gpu internals to enable reliable gpu management," in *2024 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, may 2024, pp. 294–305.

[20] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 2520–2525, iSSN: 1050-4729. [Online]. Available: https://ieeexplore.ieee.org/document/5980409

[21] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A Flexible Quadrotor Simulator," May 2021, arXiv:2009.00563 [cs]. [Online]. Available: http://arxiv.org/abs/2009.00563

[22] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "RotorS—A Modular Gazebo MAV Simulator Framework," in *Robot Operating System (ROS): The Complete Reference (Volume 1)*, A. Koubaa, Ed. Cham: Springer International Publishing, 2016, pp. 595–625. [Online]. Available: https://doi.org/10.1007/978-3-319-26054-9_23

[23] B. B. Brandenburg, "Scheduling and Locking in Multiprocessor Real-Time Operating Systems," PhD thesis, The University of North Carolina at Chapel Hill, December 2011.

[24] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *2006 27th IEEE International Real-Time Systems Symposium*. Rio de Janiero, Brazil: IEEE, 2006, pp. 111–126. [Online]. Available: http://ieeexplore.ieee.org/document/4032341/

[25] "roscpp/Overview/Callbacks and Spinning - ROS Wiki." [Online]. Available: https://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning

[26] K. Conley, D. Thomas, and J. Perron, "rospy." [Online]. Available: https://wiki.ros.org/rospy