Scheduling Dynamic Software Updates in Mobile Robots

AHMED EL YAACOUB, Uppsala University, Sweden
LUCA MOTTOLA, Politecnico di Milano, Italy, and Uppsala University, Sweden
THIEMO VOIGT, Uppsala University, Sweden
PHILIPP RÜMMER, University of Regensburg, Germany, and Uppsala University, Sweden

We present NeRTA (Next Release Time Analysis), a technique to enable dynamic software updates for low-level control software of mobile robots. Dynamic software updates enable software correction and evolution *during* system operation. In mobile robotics, they are crucial to resolve software defects without interrupting system operation or to enable on-the-fly extensions. Low-level control software for mobile robots, however, is time sensitive and runs on resource-constrained hardware with no operating system support. To minimize the impact of the update process, NeRTA safely schedules updates during times when the computing unit would otherwise be idle. It does so by utilizing information from the existing scheduling algorithm without impacting its operation. As such, NeRTA works *orthogonal* to the existing scheduler, retaining the existing platform-specific optimizations and fine-tuning, and may simply operate as a plug-in component. To enable larger dynamic updates, we further conceive an additional mechanism called *bounded reactive control* and apply *mixed-criticality* concepts. The former cautiously reduces the overall control frequency, whereas the latter excludes less critical tasks from NeRTA processing. Their use increases the available idle times. We combine real-world experiments on embedded hardware with simulations to evaluate NeRTA. Our experimental evaluation shows that the difference between NeRTA's estimated idle times and the measured idle times is less than 15% in more than three-quarters of the samples. The combined effect of bounded reactive control and mixed-criticality concepts results in a 150+% increase in available idle times. We also show that the processing overhead of NeRTA and of the additional mechanisms is essentially negligible.

CCS Concepts: • **Computer systems organization → Embedded software**; **Real-time systems;**

Additional Key Words and Phrases: Dynamic software updates, mobile robotics, safety-critical systems, aerial drones

ACM Reference format:
Ahmed El Yaacoub, Luca Mottola, Thiemo Voigt, and Philipp Rümmer. 2023. Scheduling Dynamic Software Updates in Mobile Robots. *ACM Trans. Embedd. Comput. Syst.* 22, 6, Article 99 (November 2023), 27 pages.
https://doi.org/10.1145/3623676

1 INTRODUCTION

Dynamic software updates are performed without interrupting the software execution [9]. They are useful for applications that require frequent updates but must operate continuously, with no

Authors' addresses: A. El Yaacoub and T. Voigt, Uppsala University, Sweden; emails: {ahmed.el.yaacoub, thiemo.voigt}@angstrom.uu.se; L. Mottola, Politecnico di Milano, Italy, and Uppsala University, Sweden; email: luca.mottola@angstrom.uu.se; P. Rümmer, University of Regensburg, Germany, and Uppsala University, Sweden; email: philipp.ruemmer@ur.de.

1539-9087/2023/11-ART99
https://doi.org/10.1145/3623676

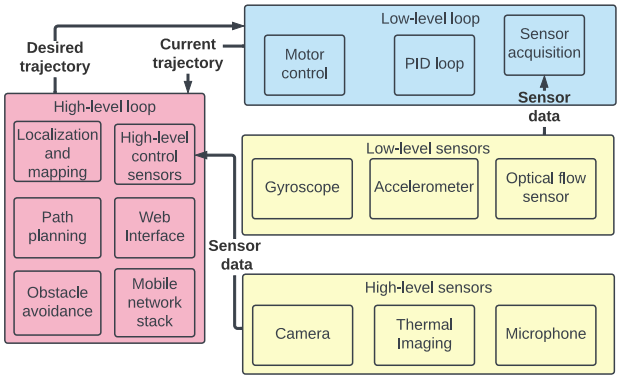


Fig. 1. Low-level and high-level robot control loops. Yellow indicates sensors. Pink indicates the high-level control loop. Blue indicates the low-level control loop. The lines indicate the dataflow between different components. Note that in reality, data flows out of each sensor individually.

perceivable downtime. In mobile robotics, dynamic software updates are useful for resolving software defects that may endanger the robot or its surroundings [27] and to extend the robot's capabilities by adding new features.

Mobile Robotics. Consider aerial drones as an example. Fixed-wing drones [4] may fly for hours while performing search-and-rescue missions. During this time, the need to patch the running software or to deploy new functionality may arise in response to unforeseen situations. For example, adjustments to the flight control software may become necessary or bugs may be discovered during the flight,[1][2][3] which can endanger the drone and the environment if left unresolved. The regular update procedure would require landing, updating the software, rebooting, and taking off again, disrupting the mission and wasting energy due to the necessary detour [29]. Dynamic software updates allow the system to continue a mission uninterrupted, saving energy and prolonging the system lifetime.

Mobile robots commonly feature a two-level control system [11], shown in Figure 1. The low-level control loop is responsible for direct control of the robot and primarily relies on two inputs: the desired trajectory in the form of inertial and angular velocities, and sensor data that determine robot attitude. The high-level control loop is responsible for advanced functionality such as localization and mapping [16]. Unlike the high-level control loop, the operation of the low-level loop is extremely time sensitive. Most existing systems employ custom implementations of the low-level loop running on resource-constrained hardware with no operating system [15, 20].

Problem and Idea. Several aspects are to be considered when performing dynamic software updates [24]. These include, for example, *what fraction of the software is replaced*, from the entire program to individual instructions; determining whether different parts of the program that are updated separately are possibly *dependent on each other*; and how to *transfer and/or modify state* from the older to the newer version. Orthogonal to these concerns is the *time of the update*—that is, *when* to perform the update while the system continues to run. We focus on how to perform the latter operation safely as it is crucial in low-level robot controllers [11]. The other operations we mention must also be performed safely; however, doing so depends on *what* is performed during and after the update process.

[1] https://github.com/ArduPilot/ardupilot/commit/6ebefbdb1680b61efc30d74ebcd95861f8adf02a
[2] https://github.com/ArduPilot/ardupilot/commit/8e37c93e7d7716885f97addd941fa0df3b47040d
[3] https://github.com/ArduPilot/ardupilot/commit/48881eeb5517d2804f929aa3d56b68f958f73772

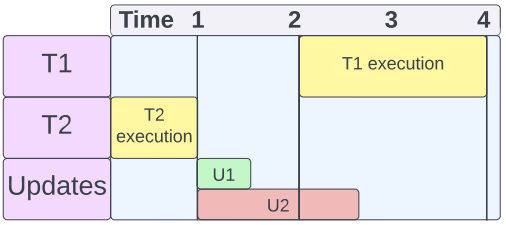


Fig. 2. An example of a schedule with two jobs, T1 and T2 (yellow), and two possible updates, a schedule-invariant update U1 (green) and a schedule-invariant update U2 (red). The schedule is divided into four time intervals indicated on top. Overlapping jobs indicate a scheduling conflict where one must be chosen, not parallel execution.

We specifically tackle the problem of when to perform a dynamic update of the *low-level control loop*. Unlike the high-level control loop, if the low-level control loop is unresponsive or delays the execution of crucial tasks, the robot loses control because its actuators receive late values compared to the robot state and desired setpoints [27]. Therefore, updating the low-level control loop dynamically, which is the focus of our work, is significantly more challenging than updating the high-level one.

We define *schedule-invariance* as a property that can guarantee the update process is performed at a safe time. *Schedule-invariance* states that the update must *begin and end* its execution without affecting when the robot's tasks execute. *Schedule-invariance* is linked to safety because deadline misses degrade the control loop's performance significantly and possibly cause situations that eventually lead to unsafe behaviors [34]. *Schedule-invariance* ensures the exact same schedule as if the update is not performed. Doing so ensures that the robot's physical behavior *during the update process* is identical to the behavior if the update had not been performed with respect to the timing properties.

Our fundamental idea is that the update can be made schedule-invariant by scheduling it completely when the computing unit running the low-level control loop would otherwise be idle—that is, when no tasks are running. We consider an update to be a time-bound task, whose worst-case execution time is known. Worst-case timing analysis [22] can be used to determine the worst-case execution time of the update, given that the necessary operations are known. Figure 2 shows two possible updates. Update U1 starts and completes when no tasks are running and therefore is schedule-invariant. Update U2 extends into the portion of time when T1 is to execute. Therefore, U2 necessarily delays the start of T1. Thus, U2 is not schedule-invariant. The availability of idle time is due to the scheduler leaving the interval between times one and two empty, which we utilize to perform a schedule-invariant update U1. Figure 2 also demonstrates why the update must be time-bound, since otherwise it would be impossible to determine when the update completes, and therefore whether it extends into the time interval when a task was scheduled to run.

NeRTA. To perform schedule-invariant dynamic software updates, we develop NeRTA (Next Release Time Analysis), a technique that dynamically and safely estimates the available idle times. We use the estimated idle time to check for the schedulability of the update—that is, it is schedule-invariant. We schedule an update using NeRTA only if the time taken to perform the update is less than or equal to the NeRTA estimate. These estimates are *conservative*—in other words, we guarantee they represent a lower bound compared to the actual idle times. This feature is essential to ensure that the time required for the update does not exceed the actual idle time available and thus remains schedule-invariant.

We conceive NeRTA to enable dynamic software updates in existing software, in contrast to designing software from the ground up that incorporates dynamic updates as a native

functionality, which is a related but entirely different problem. In this sense, schedule-invariant updating represents a *fundamental and intentional* design choice. The alternative would be, for example, to embed software updates as an additional task in the original scheduler design, allowing the system to adjust its application-level behavior to accommodate updates. Doing so would change the timing behavior of existing tasks, greatly complicating implementation, testing, and verification. In contrast, NeRTA operates *orthogonal* to the existing scheduler, retaining the extensive testing, platform-specific optimization, and fine-tuning of the existing scheduler. As long as the necessary information is available from the existing scheduler, as explained in Section 3, NeRTA operates as a plug-in component in existing systems to schedule updates while meeting schedule-invariance.

Increasing Idle Times. Low-level control loops of mobile robots typically utilize resource-constrained hardware and run several tasks at high frequencies, up to hundreds of hertz [11]. Therefore, the available idle time may be limited. We measure the idle times in the open source drone autopilot Hackflight [20] and find that no more than 3 ms are available, which might be insufficient for larger updates. Section 4 provides additional details on these measurements.

We explore two optimizations to increase the size of the update that can be accommodated within idle times by relaxing the schedule-invariant condition, as explained in Section 5. Instead of being schedule-invariant, the optimizations must maintain *robust* operation of the robot. Robustness is defined as the system's ability to withstand disturbances while meeting performance goals [14]. A robust robot meets a set of *platform-specific requirements* within *well-defined environmental conditions* that allow the low-level controller to retain controllability of the robot. An example where this does not happen is an aerial drone no longer capable of hovering at a fixed position after an update [8].

The first such optimization, called **Bounded Reactive Control (BRC)**, cautiously reduces the overall control loop frequency, which yields a corresponding increase in the idle times available. Further, we apply *mixed-criticality* concepts and identify the safety-critical tasks that are mandated to keep running at the original frequency while excluding the others from NeRTA's computations. Doing so reduces the number of tasks NeRTA takes into account when finding the time to schedule an update, thus again increasing the idle times available.

Prototype and Performance. Although the design of NeRTA is orthogonal to the existing task scheduler, its implementation must be properly customized to the specific flight controller. We create a real-world prototype, described in Section 6, by integrating NeRTA in the task scheduler of Hackflight, an open source aerial drone flight controller [20]. We build a custom drone running Hackflight and use that, combined with a physics-accurate drone simulator [19], for evaluation.

Our experiments, reported in Section 7, reveal that NeRTA estimates, although conservative, are close to the actual idle times. The difference between NeRTA's estimated idle times and the measured idle times is less than 15% in more than three-quarters of the samples. All idle times larger than 0.6 ms show differences less than 15%. Moreover, our investigation reveals interesting insights into the combined use of BRC and mixed-criticality concepts, which together result in a 150+% increase in available idle times, without endangering the robust drone operation. We also demonstrate that the price to pay, represented as processing overhead, is negligible. We publish a replication package for our experiments[4] that may be instrumental to build upon our work.

We conclude the article in Section 8 by articulating threats to validity and limitations in our work, and with brief concluding remarks in Section 9. Before moving on to the technical matter, Section 2 provides a brief survey of related work.

[4]https://github.com/ahmadadam96/NeRTA-TECS-replication-package

2 RELATED WORK

To decide when to schedule an update, quiescent points should be identified and classified. Quiescent points are intervals in time when an update can be safely performed. Quiescent points can be inserted into the program by the programmer, by the compiler, or automatically identified at runtime [35, 37]. NeRTA combines both approaches by inserting potential update points at the end of each job, then automatically computing whether the update point is valid—that is, there is sufficient time to perform the update before the next job runs. Further, unlike the work of Wahler et al. [35], NeRTA provides estimates of how much idle time is actually available.

Dynamic update solutions based on selecting safe update points are explored in several studies [13, 23, 37]. Cazzola and Jalili [13] claim that the safety of an update point can be determined if (i) the state of the running application is available, (ii) the type of changes is known, and (iii) it is possible to predict the impact of the change during the dynamic update. Their work focuses on the safety of the update point based on source code information rather than on the capability of the update process to meet real-time requirements. Zhao et al. [37] elicit two requirements for update points: (i) timeliness, meaning that an update point is eventually reachable, and (ii) correctness, meaning that the program should behave correctly after the update. Unlike what we do, they do not explore the impact of the update process while the update is performed. Lounas et al. [23] focus on verifying that update points satisfy three objectives: being deadlock-free, activeness safety, and liveness. They use model checking to verify those properties for different update points. Their work is complementary to ours. Although those properties are important, they do not ensure that the update is performed at a time that ensures real-time requirements are met.

Seifzadeh et al. [30] split approaches determining the time of the update into two types: immediate and delayed. Immediate approaches apply the update as soon as it arrives. Delayed approaches postpone it until a suitable time. NeRTA is a delayed approach. They further split the delayed approaches into two type: techniques that determine an appropriate time, and techniques that modify the behavior of the program to make the time appropriate. NeRTA is of the first type; however, the optimizations in Section 5 are of the second type.

The work of Rommel et al. [28] considers multi-threaded systems and distinguishes between global and local quiescence. Global quiescence is when all threads are inactive, whereas local quiescence is when a thread is inactive. They propose updating different threads separately utilizing local quiescence. NeRTA targets single-threaded systems and therefore does not benefit from a local quiescence approach. However, NeRTA considers more than lack of active operation when determining whether an update point is safe. It also considers real-time requirements, which are key for mobile robotics.

Holmbacka et al. [17] target specifically runtime updating of embedded systems. Their work is compatible with FreeRTOS. They update tasks by creating a new FreeRTOS task with updated code and deleting the old task. This approach works because the tasks are compiled to be position independent and relocatable. They use a state transfer algorithm from the work of Wahler et al. [35]. Their work is complementary to ours. It focuses on the first three fundamental aspects of dynamic software updates, whereas our work focuses on the last aspect, the time of the update.

Most work on dynamic software updates do not analyze the possible vulnerabilities that may be introduced by implementing dynamic software updating. Security is studied in more depth with traditional (non-dynamic) over-the-air software updates. As an example, Nilsson et al. [25] identified and secured two stages of the update process where security may be compromised: (i) downloading a maliciously modified binary, and (ii) maliciously modifying a binary after download. Their work aims at updating vehicle ECUs that are connected to a central portal that performs the downloading, storage, and transmission of the binary. This is analogous to the typical mobile robot architecture, where the central portal is the processor running the high-level

control loop, and the ECU is the processor running the low-level control loop. Their work assumes the attacker has gained access to the portal but not to the ECUs, which is a fair assumption in our case too, since the low-level control loop's only communication with the outside is through radio, whereas the high-level control loop is usually connected through the mobile network. Therefore, the methods they propose can be reused in our context as well.

NeRTA has the explicit goal of scheduling updates while meeting schedule-invariance. Doing so results in the update process having no impact on the real-time requirements, because all tasks run when they would have with no update.

3 NeRTA: NEXT RELEASE TIME ANALYSIS

We describe the design of NeRTA, a technique to estimate the *duration* of idle times in an existing schedule and use it to schedule an update.

3.1 Target Systems

With NeRTA, we focus on enabling dynamic updates for pre-existing software over the alternative problem—that is, designing software with built-in update functionality. Therefore, we prioritize wide compatibility and ease of integration, and therefore we have limited ability to change the existing task scheduling algorithm. When tackling the alternative problem, the additional flexibility may be used, for example, to design a new task scheduling algorithm that meets both the real-time requirements and also guarantees that updates are scheduled quickly.

We target systems implemented with a fixed number of tasks, each of which is responsible for executing an unbounded number of computation jobs. The system is equipped with a basic task scheduler with a task scheduling algorithm that provides sufficient information to compute the *release time* of each upcoming job. The release time is the earliest time a job can start executing. The time between the end of one job and the start of a next job, when no task is running, is defined as *idle time*. To compute the length of the idle time, we must forecast when the next job of any task starts. A job is guaranteed not to execute before its release time. The release time is computed by the task scheduling algorithm. How the computation of the release time is achieved depends on the task scheduling algorithm. NeRTA is in principle orthogonal to such scheduling algorithm, as long as the release times are available. Moreover, NeRTA is a runtime technique in that it computes idle times as the software executes. Therefore, it is applicable to both dynamic and static (time-triggered) scheduling algorithms.

Task scheduling algorithms where release time information is available are commonly found in mobile robots and other embedded systems, due to the reliance on sensors that have fixed refresh rates and control loops that run at fixed frequencies [3, 20]. Therefore, NeRTA enjoys wide applicability, especially in mobile robotics. For example, to compute the release time of a job in a rate-monotonic scheduled system [18], the arrival time of the previously completed job is added to the period of the task the job belongs to [31]. Tasks may have scheduling dependencies and do not have to be independent. Task dependencies in NeRTA can be modeled using release times. If task T2 cannot execute prior to task T1, then the release time of T2 can be set to the maximum of the release time of T1 and the arrival time of T2.

NeRTA is not applicable to task scheduling algorithms without guaranteed inter-arrival delays. In these cases, a release time that is larger than the current time cannot be provided. For example, systems that use interrupts do not have guaranteed inter-arrival delays for tasks invoked by an interrupt. Therefore, they do not have guaranteed intervals of idle periods because an interrupt may fire at any moment. Another example is a scheduler where the job's release time depends solely on the availability of required inputs rather than on time. Without knowledge of when these inputs are available, a release time cannot be computed.

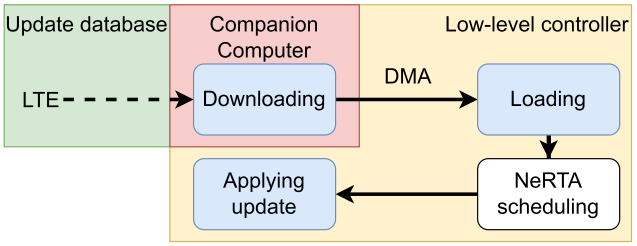


Fig. 3. Update stages (blue) with NeRTA used to schedule the applying update stage. Green represents the update database located externally. Red represents the powerful computer located on the device. Yellow represents the low-level controller being updated. Blue represents stages of the update model. A solid line is on-device communication, and a dashed line is off-device communication.

3.2 Update Model

We consider an update to be one or multiple aperiodic tasks where each task maps to a stage in the update process. Each task in the update must have a known worst-case execution time. All tasks in the update are non-preemptive. Only tasks that use up execution time on the low-level controller need to be scheduled with NeRTA before they are performed to ensure the update process is scheduling-invariant. The number, choice, and order of stages is a decision we leave to the update developer. We require that the system must be considered correct both before and after each individual stage is applied, and that by conducting all the stages, the program will be updated.

For example, we may categorize the update into a *loading* stage and an *applying* stage. The loading stage obtains the update files from an external source, and the applying stage applies the update. This is a valid categorization because the system operates correctly between the two stages. Therefore, the stages can be applied separately. An invalid categorization is, for example, splitting the update into a *machine code update* stage and a *state update* stage. If the stages are performed separately, the system may find itself in an invalid state because the new machine code may not be compatible with the old program state.

In the rest of the article, without loss of generality, we consider a specific (valid) update model, shown in Figure 3 and representative of a vast class of mobile robotics platforms [5–7]. We consider this model as a concrete example that meets the requirements we state while being suited for the type of architecture mobile robots have.

A *diff* file is generated externally including the necessary changes to convert the original version of the software to the new version. The changes are made on a block level. Each block represents a contiguous memory region. Each entry in the *diff* file specifies the starting memory address of the block to be replaced, the size of the block, and the new content of the block. By replacing all the blocks in the *diff* file, the machine code and program state are updated. How the *diff* file itself is actually created, and how it is processed, is orthogonal to our work. The machine code portion of the *diff* file may be created by comparing the binaries of both versions. The program state portion of the *diff* file is more complex and requires state transfer mechanisms [38].

The downloading stage transfers the *diff* file and is performed on a companion computer, for example, through a cellular connection. The loading stage loads the *diff* file onto the low-level controller using DMA (Direct Memory Access). Once loading is complete, NeRTA is invoked to ensure that the last stage in the process, which updates the machine code and program state, is schedule-invariant. We update the entire machine code and program state in one go to simplify the verification process, since updating parts of the software means each intermediate version must be verified to meet a range of application-specific requirements. Both the downloading stage, since it

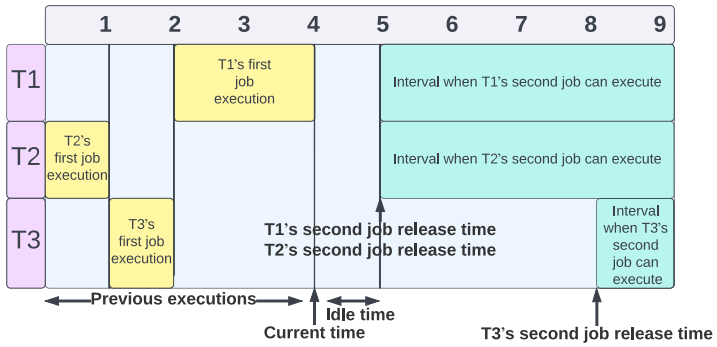


Fig. 4. Example demonstrating release times of jobs in a system. Current time is 4. Already executed jobs are yellow. Intervals where execution is possible are cyan.

is performed on a companion computer, and the loading stage, since it uses DMA, are performed in parallel with the regular execution and do not require scheduling with NeRTA. Only the final stage uses execution time on the low-level controller and therefore can impact the control tasks. This is the only stage scheduled with NeRTA.

Ensuring that the *diff* file the companion computer receives is a legitimate one represents an orthogonal problem that may be addressed, for example, using a public/private key encryption schema to check authenticity and checksums to check integrity. The companion computer performs these operations *before* the *diff* file is loaded on the low-level controller; therefore, they do not impact scheduling the update.

3.3 NeRTA

We start with an example to provide the basic intuition behind NeRTA and describe its generalization next.

Example. Figure 4 exemplifies NeRTA's design. We consider three tasks: T1, T2, and T3. In a low-level control loop for drones, for example, T1 is the sensor task that gathers data from the IMU (Inertial Measurement Unit), T2 is the flight control task, and T3 is the receiver task processing commands from a remote controller. As in existing low-level robot controllers [1, 15, 20], tasks are not preemptable. Each job has a release time that is computed by adding a fixed number of time units to the time the prior job of the same task starts its execution.

For simplicity, we assume that whichever job reaches its release time executes immediately, unless either another job is running already or two or more jobs reached their release times simultaneously, in which case one is randomly chosen to execute. To compute the release time of the T1 jobs, we add three time units to the time the previous job of T1 starts, we add five time units for T2's jobs, and we add seven time units for T3's jobs. These numbers are for illustration purposes. If the time units used are milliseconds, then these numbers accurately represent typical times observed in drone autopilots such as Hackflight. In this example, we are currently at time 4 in Figure 4, and therefore one job of each task has been executed.

Figure 5 shows the actual execution of the system as observed until time 9. At time 5, the task scheduler randomly chooses T1's second job over T2's second job due to the overlap in release times. When T1's second job completes its execution, the task scheduler computes the release time of T1's third job by adding three time units to when its second job starts, resulting in a new release time of 8. At time 7, T2's second job then executes because neither T1's third job nor T3's second job reach their release times. The scheduler computes a release time for T2's third job after

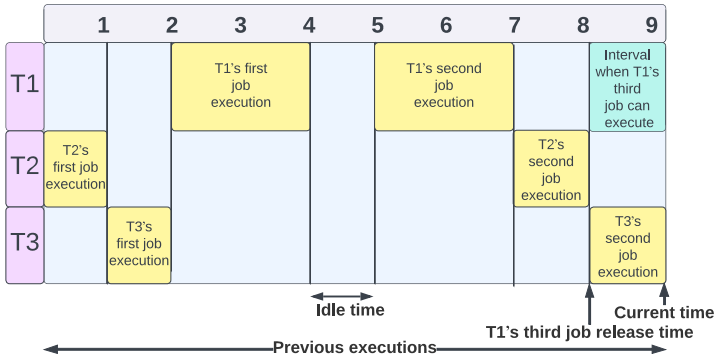


Fig. 5. The system of Figure 4 after the execution of the second job of all tasks. Current time is 9. The colors mean the same as shown in Figure 4.

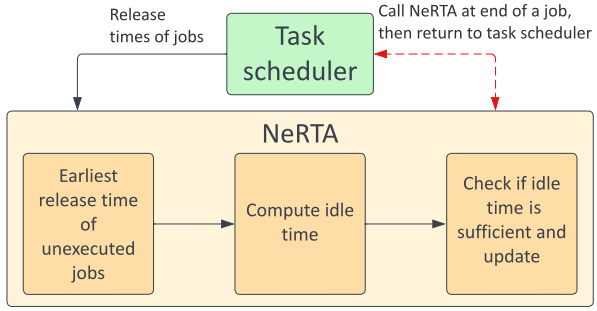


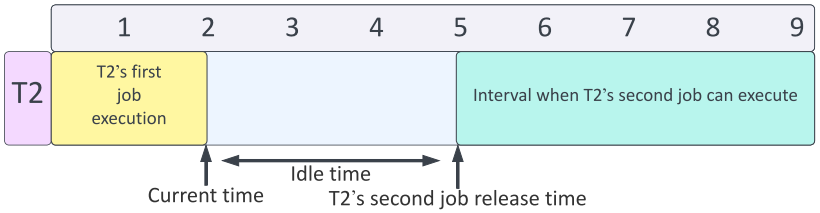
Fig. 6. Interface between NeRTA and the task scheduler. Steps taken by NeRTA are orange. The task scheduler is green. The solid line indicates dataflow. The dashed line indicates control flow when the task scheduler calls NeRTA, which returns to the task scheduler.

its second job executes. At time 8, the release times of T1's third job and T3's second job align. The scheduler randomly executes T3's second job over T1's third job.

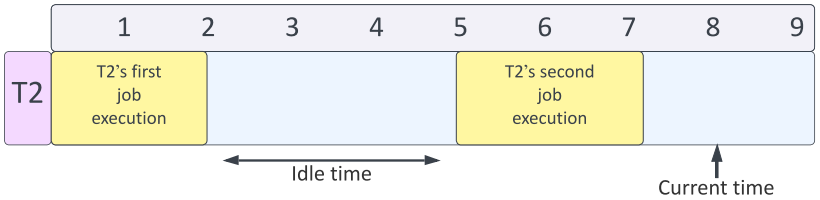
Figure 5 shows an idle time worth one time unit between times 4 and 5. This happens as the earliest release time of any task in Figure 4 is at time 5. Between the current time and the earliest release time, no jobs can possibly execute: the computing unit is guaranteed to be idle. This is a perfect time to perform an update job that does not take more than one time unit to complete, as none of the pre-existing jobs would be affected. Updates performed during are schedule-invariant *by construction*. NeRTA generalizes this reasoning, as explained next.

General Case. Figure 6 shows how NeRTA interfaces with the task scheduler. NeRTA requires two inputs: an update job with a specified worst-case execution time, and the release times of the earliest unexecuted job of each task in the system. We have an infinite number of jobs, because tasks repeat indefinitely. However, tasks have a guaranteed inter-arrival time between one job and the next. We only need the release times of the earliest job of each task, which is guaranteed to be smaller than the release time of latter jobs of the same tasks.

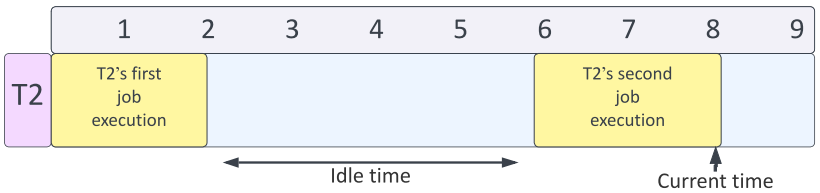
The task scheduling algorithm computes the release time of each job at the appropriate time and sends it to NeRTA. Once NeRTA has up-to-date release times for all jobs, it can estimate the available idle time. If an update job is waiting to be scheduled, the task scheduler calls NeRTA after any job completes its execution. Calling NeRTA after the execution of a job simplifies the computation of idle times because only the release time of upcoming jobs is needed to determine



(a) Task 2's schedule at time 2. The colors mean the same as Fig. 4.



(b) Possible task 2 schedule at time 8. The colors mean the same as Fig. 4.



(c) Alternative task 2 schedule at time 8. The colors mean the same as Fig. 4.

Fig. 7. Possible executions given a release time interval.

the idle time. NeRTA computes the estimated idle time by subtracting the current time from the smallest release time of any job. The current time is measured immediately prior to the subtraction to obtain the most accurate estimate possible. For an update job to be scheduled successfully, the estimated idle time must be greater than the time needed to perform the update. If the update cannot be scheduled right then, the task scheduler waits until the next job completes and calls NeRTA again with updated release time information. In Section 7.4, we find the overhead of NeRTA to be negligible, and since we measure the current time immediately prior to computing the idle time, NeRTA's overhead is already incorporated into the estimated idle time. To schedule updates composed of several jobs, NeRTA is invoked sequentially for each job that needs execution time from the low-level controller. Once the first job is scheduled successfully, the second job waits to be scheduled.

The idle time NeRTA computes is a *conservative* estimate of the actual idle time, meaning that NeRTA estimates cannot be after the actual idle time. This is because release times indicate the *earliest* time a task can start executing a job, which cannot be larger than the time the execution actually starts.

Figure 7(a) shows one job of task T2 with a release time of 5, which we observe from time 2. Because the release time indicates the earliest time a job can execute, the time the job actually executes may be different if the task scheduler delays its execution. Figure 7(b) and (c) demonstrate two possible executions observed from time 8, with one execution starting at time 5 and another at time 6. Despite the differences in the executions, using the release time to compute the idle

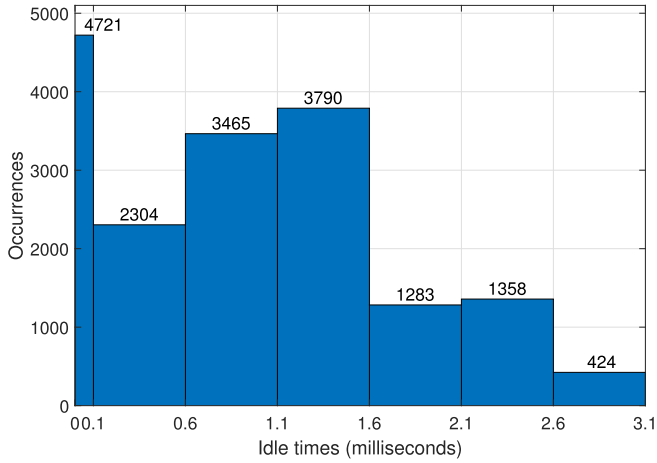


Fig. 8. Idle times in Hackflight; note the prevalence of idle times between 0.6 and 1.6 ms.

time provides a guaranteed idle time of at least three time units, computed at time 2, because it is impossible for the job to execute prior to time 5.

As long as the release times provided are correct, the idle time computed from those release times is necessarily conservative. The conservative nature of NeRTA estimates is fundamental because if the estimated idle time is larger than the actual idle time, performing the update in this time span may postpone the execution of jobs and therefore is no longer schedule-invariant.

4 IDLE TIME EMPIRICAL EVALUATION

NeRTA conservatively estimates the available idle times in an existing task schedule. Whether an update may be completed in the available idle time is an orthogonal question that deserves further investigation and motivates our work in Section 5 toward increasing the available idle times. We provide a quantitative basis to reason upon in this section.

4.1 Idle Times in Hackflight

We run experiments to determine the idle times available on Hackflight [20], a low-level flight controller for aerial drones running on embedded resource-constrained hardware [36]. Using an existing autopilot implementation and real hardware, we can measure the idle times that are present in a real-world mobile robot platform. We run the experiment on a stationary but armed drone—that is, the motors are spinning as they would normally do but the propellers are removed so the drone does not lift off.

By instrumenting the code, we record every time a task begins its execution and every time a task ends its execution. By aligning the traces obtained this way over time, we can derive the available idle times *post-facto*. Figure 8 shows an aggregate view on the results we obtained across 17,345 samples of idle time. The largest idle time we observe is just under 3.0 ms, yet only about 1.8% of the samples show idle times larger than 2.7 ms. However, 46.1% of the samples are larger than 1.0 ms. Overall, we find a median idle time of 0.8 ms and an average of 0.9 ms.

Note that in Figure 8 the size of the bins is not homogeneous. In the 0 to 0.1-ms interval, we obtain numerous small samples because two or more jobs have overlapping release times and therefore can execute one right after the other.

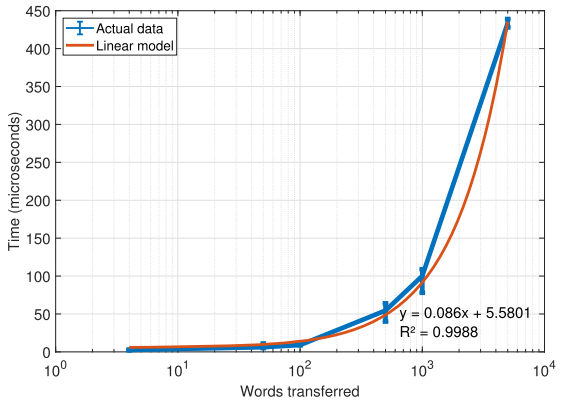


Fig. 9. Time taken to transfer different numbers of words from Flash to SRAM as one block. A linear model approximates the data well. The equation for the linear model along with the coefficient of determination are displayed. The figure includes the average, upper, and lower bounds of the measurements.

4.2 Update Time Durations

Idle times may not be meaningful per se; it is rather what update we can fit within those times that matters. The time taken for performing an update is dominated by operations on non-volatile memory to change the machine code. We run experiments to relate the preceding idle times with the size of the memory operations to perform an update.

We consider the STM32L432KC MCU [33], a 32-bit ARM Cortex M processor as used by the hardware platform running Hackflight [36]. It uses Flash as the non-volatile memory and SRAM as the volatile memory. To update the program, both the state stored in the data memory, SRAM in our case, and the machine code in program memory, using Flash, must be updated. We perform a memory benchmark to gauge the time required to update the program.

4.2.1 State Updates. We evaluate the time needed to update the state. We assume that the update is a *diff* file that is composed of a series of commands. Each command states to replace block X with block Y, and blocks X and Y have size W. Block X is found in data memory starting at memory address Z while block Y is found in the *diff* file stored in the Flash. Each command in the *diff* file contains the necessary changes that must be made in SRAM to convert the old version to the new version. The *diff* file already contains all necessary changes to the program state, and therefore the update process only has to execute the commands to transfer blocks from the *diff* file to the specified location in SRAM. The *diff* file is given. Therefore, determining how the *diff* file is created is outside the scope of this work.

To estimate how long an update to SRAM takes, we measure the amount of time taken to transfer blocks of various sizes with Flash as the source and SRAM as the destination. We measure the time to transfer one block of increasing sizes from 4 words (4 bytes in each word) to 5,000 words, as shown in Figure 9. We perform the measurements six times and plot the average, upper, and lower bounds. Note that we plot the number of words transferred on a logarithmic scale.

Figure 9 demonstrates that we can transfer 5,000 words in around 430 μs. We also observe that the transfer time follows a linear relation as shown in Figure 9. We use the equation to calculate that the time to update 256 words, corresponding to 1 KB, in SRAM is around 28 seconds, and that we can overwrite the entire SRAM of a typical embedded microcontroller like the STM32L432KC [33] in 1.4 ms, which is feasible in the idle time we show in Figure 8.

We also found no significant difference in the total time taken between writing to locations in SRAM that are contiguous, such as one large block of words, or writing to locations that are not

Table 1. Update Time Required for Different Kinds of Updates to Hackflight

Code modification	Words changed	Time (ms)
Modify input to PID controller	1	0.00025
Modify one PID computation	2	0.0005
Remove an entire PID controller	20,209	5.05
Modify formula for computing Euler angles	7,882	1.97

contiguous, such as several small blocks of words with different destinations. Therefore, the time to update is only impacted by how many words must be copied, not by whether the words are copied to contiguous locations in memory.

4.2.2 Machine Code Updates. Next we evaluate the time required to update the program memory—that is, the machine code stored in non-volatile memory. We originally consider Flash memory as non-volatile memory because it is prevalent among microcontrollers used for mobile robots. However, we find Flash memory to be too slow for the task at hand, due to the necessity to erase an entire page to only modify a single word in the same page. The datasheet for the STM32L432KC processor, used to run Hackflight, mentions that the time to erase a page of size 2 KB (512 words) is 22.02 ms and the time to write an entire page is 20.91 ms [33]. Therefore, updating a previously programmed page takes 42.93 ms. Considering we have a maximum 2.9 ms of idle time, it means we would need roughly 20 times the idle time we currently have to update just one page.

An alternative form of non-volatile storage is **Ferroelectric RAM (FRAM)**, which does not require an erase procedure to overwrite data. The only microcontroller family available with onboard FRAM is the MSP430FR family, which sports a 16-bit computing unit able to write a 16-bit word in 125 ns [2]. We estimate that an onboard FRAM with a 32-bit processor would be able to write a 32-bit word in double the time, which is 250 ns. To determine how many words must be modified for different updates, we compile Hackflight for the ARM Cortex M4F processor using the GNU Arm Embedded Toolchain. We apply different updates to Hackflight and recompile. We use the machine code produced by compiling the unmodified Hackflight source as the default and compare all other binaries with it. We measure the number of the words that we must modify to convert the default machine code to the updated machine code. The comparison is made at address-level. For example, if address 12 contains the one-word integer 100 before the update and contains the one-word integer 200 after the update, we consider the word in that address as modified.

We consider selected modifications to the Hackflight codebase, summarized in Table 1. We select these modifications to include both minor and major changes. We also choose them to model realistic updates for a drone autopilot. The first one models parameter adjustments. The second and fourth model bug fixes. The third one models a feature removal intended to improve execution time. Using these modifications enables us to observe the variation across update times.

Compared to the idle times available in Hackflight, shown in Figure 8, the first two modifications are easily accommodated. The third update needs to modify a large number of words, due code shifting location in memory: it is unfeasible in the available idle times. The fourth update also results in code shifting; however, the total number of words that must be overwritten is significantly smaller than the third update. The fourth update can be performed in the idle time available, provided that the update to the state does not exceed 1 ms.

To summarize, performing both a large state update, needing 1.4 ms, and a small machine code update, needing as much as 0.5 second, is doable within the available idle times. Performing the same large state update and a medium-sized machine code update, needing 1.97 ms, is not doable. Performing the medium-sized machine code update is doable provided the state update does not

exceed 1 ms. These results motivate developing further solutions to increase the available idle times, increasing the chances that larger updates may be accommodated.

5 INCREASING IDLE TIMES

Depending on the number and nature of changes to be performed, it may not be possible to schedule large updates. With NeRTA, we target updating pre-existing software and therefore prioritize maintaining the application-level behavior by schedule-invariant updates. For larger updates, this approach is no longer sufficient. Therefore, we develop optimizations (separate from NeRTA) that increase the available idle times. They do so by relaxing schedule-invariance and instead require that software updates only maintain the *robust* operation of the robot.

To increase available times, in Section 5.1 we apply mixed-criticality concepts [12], which retain the schedule-invariant properties only for critical tasks. Next, in Section 5.2 we present a notion of BRC [11], which judiciously reduces the frequencies of the control loop.

5.1 Mixed Criticality

Mixed criticality is an approach to recognize that not all tasks are of equal criticality in a time-sensitive system, where criticality refers to the task's importance to maintain robust operation [12]. We apply this concept to the low-level robot controllers by defining two criticality levels: *high* and *low*. The former includes tasks that are necessary to maintain the robust operation of the mobile robot; *low* criticality tasks are all others.

The classification of low and high criticality is application-specific. However, it needs to take into account the environment as well as the capability of the software. A task would be deemed low criticality only if the robot is able to operate for an extended period of time in the given environment without it.

Consider a low-level robot controller with three different tasks: a *communication* task that receives control inputs from a ground station, a *mobility control* task that controls the robot's actuators based on control inputs, and a task to *update values* of on-board sensors. The sensor task and the mobility control task are of *high* criticality. The former ensures that the robot detects significant changes in its orientation and in the surrounding environment, such as unintended angle changes detected by the robot's accelerometer, whereas the mobility control task is needed to react to those changes, such as ensuring that the robot can rebalance itself.

However, the communication task can be assigned a *low* criticality provided certain requirements are met. In the absence of newer control inputs, the robot controller may default to specific setpoints, for example, to retain the current position. The communication task is of *low* criticality if the drone is capable of retaining the current position safely for an extended period of time without user input and the environment does not endanger a stationary drone. For example, say birds may be flying around the drone, then a stationary drone with no automated collision avoidance would not hover safely in this environment. In that case, the communication task is of *high* criticality because the pilot should intervene to counteract the situation.

By classifying tasks into two distinct criticality levels, we increase the amount of idle time available to perform an update by only taking into account *high*-criticality tasks when running NeRTA. Therefore, mixed criticality is not schedule-invariant toward *low*-criticality tasks but is toward high-criticality tasks. Therefore, a low-criticality task may be delayed in its execution because of an update. This is considered acceptable because, by definition, the robot can safely operate in the current environment without the low-criticality task.

After the update is complete, it is possible that some low-criticality jobs are past their release time, since they are delayed by the update. Therefore, we must ensure that these jobs do not further delay the high-criticality tasks. To do so, we require the worst-case execution time of all

low-criticality tasks. Once the update is applied, we disable all released low-criticality tasks. We add new stages to our update model from Figure 3, one for each disabled low-criticality task, which are to be scheduled by NeRTA, called *re-enabling* low-criticality tasks After each job executes, we use NeRTA to compute the idle time taking into account only enabled tasks. We utilize the idle time to schedule as many ready and disabled low-criticality jobs as can fit during the idle time based on worst-case execution time information. The ones that execute are then re-enabled. Over time, all low-criticality tasks are re-enabled without delaying high-criticality tasks, and the update process is finally considered complete. We assume that the system was schedulable prior to disabling the low-criticality tasks. Therefore, the capacity to re-enable the low-criticality tasks is guaranteed.

5.2 Bounded Reactive Control

A complementary approach is to adapt a technique called *reactive control* [11] to make it compatible with NeRTA. We call this *bounded reactive control* (BRC).

Reactive Control. Bregu et al. [11] present a technique to dynamically modify the frequency of the low-level control loop in mobile robots. Techniques that modify the control loop's frequency are not schedule-invariant. The fundamental insight of Bregu et al. [11] is that the outputs of the low-level robot control loop should change more often when the environment surrounding the robot is more dynamic, whereas they may be updated less often in calm conditions. The specific situation is determined by the amount of change in the values of on-board sensors from one measurement to the next [11], which is taken as indication of how dynamic the environment is.

Using reactive control as-is would not work with NeRTA, as the latter requires a release time for each task as input. Conversely, *at any given time instant*, reactive control dynamically determines whether the low-level robot control loop should run, based on sensor changes [11]. Therefore, we would not know whether the control loop actually runs until the very moment we perform the reactive control computation, and therefore cannot compute release times.

We adopt the same principles of reactive control—that is, that the low-level robot control loop should run at high frequencies in difficult conditions and at low frequencies in calm conditions. Instead of determining on the spot whether the control loop should run, we change the frequency of the control loop among a limited set of candidate settings, depending on the robot and environment dynamics. By setting a frequency for the control loop beforehand rather than dynamically, the task scheduler can compute release times for the relevant tasks and use them as input to NeRTA.

Bounded Reactive Control. We perform a training phase to determine the boundary conditions that ensure (or do not ensure) robust robot operation, as a function of the frequency of low-level control. The boundary conditions are constants that are compared with the values of on-board sensors to choose a control frequency for which the robot is robust. For three frequency settings based on one sensor's readings, for example, there are two boundary conditions: one for crossing from the lowest to the middle frequency, and one for crossing from the middle to the highest frequency.

Our training phase consists of a series of tests with specific mobility patterns while checking the robot's capability to meet a specific level of robustness. The goal is to determine parameter ranges to attain robust behavior at a specific control loop frequency, and thereby determine the boundary conditions for the chosen frequency settings. As an example, using a certain control loop frequency, we program a ground robot to move along a straight line at a given speed and check whether it can avoid an obstacle in front of it. The outcome of the experiment tells us whether the specific combination of speed setting and control frequency is within (or outside) the parameter range of robust operation. This information is then used at runtime, whereby the robot dynamically selects the lowest control loop frequency that retains robust operation as a function of current speed.

Table 2. Maximum Speed Where Hovering Is Achieved for Different Control Frequencies

Control frequency (Hz)	Maximum speed (m/s)
100	1
200	16
300	Otherwise

Training. We consider aerial drones as a proof of concept. The robustness requirement is that the drone can stabilize in a hovering configuration—that is, slow down within a limited speed interval and eventually maintain a given position, all within a set amount of time. We perform training on a physics-accurate drone simulator called *MultiCopterSim* [19] using the implementation described in Section 6. The simulator is based on a detailed drone model [10]. The drone simulated is a DJI Phantom. It weighs 1.38 kg and has a frame length of 35 cm.

We start with a control frequency of 100 Hz, which is a third of the default control frequency, thus significantly increasing the time between one job and the next. We fly the drone to an altitude of 20 m, then externally apply full roll until we reach a set speed. The roll setpoint is reset, and we wait for the drone to reach a hovering state—that is, speed is between 0 and 0.5 m/s. If this happens in less than 15 seconds, we consider control frequency robust for the set speed. The speed we use is the magnitude of the velocities in the x, y, and z directions.

We perform experiments in MultiCopterSim by using a script that introduces small timing variations through the different steps of the simulation, providing a degree of non-determinism in the training that models real-world flight dynamics. We repeat each experiment 50 times and only consider a frequency robust when the drone can successfully hover in every such repetition.

Outcomes. Table 2 shows the results, which we use to program BRC to set the control frequency to 100 Hz when the drone's speed is within 1 m/s, to 200 Hz when the speed is between 1 and 16 m/s, and to 300 Hz when the speed is higher than 16 m/s. BRC utilizes incremental reduction and instantaneous increase of the control loop frequency. A decrease of the control frequency due to a low speed steps down from 300 Hz to 200 Hz first, then to 100 Hz. An increase of the control frequency due to a high speed instantly steps from 100 to 300 Hz. This ensures prompt reactions.

Note that we also use BRC to modify the execution frequency of the sensor tasks as well, since they are upper-bound by the control frequency. If BRC reduces the control frequency, we can correspondingly lower the sensor's frequency too.

Generality. We only use the drone's speed to determine the boundary conditions as a tradeoff between training complexity and versatility. Using the speed as a boundary condition is an obvious choice since a larger speed means a larger distance covered in the same time period and therefore a larger distance per iteration of the control loop. This can be counteracted by increasing the control frequency at larger speeds, which is what BRC does.

We can use more metrics to improve the drone's robustness such as its altitude, orientation, acceleration, and angular velocity. In case multiple metrics are used, BRC should choose its frequency based on the metric that results in the highest control frequency. We should repeat the training phase for each metric independently. For angular velocity, we can induce an angular velocity in one direction, then measure the time to reach a stable hovering configuration, and ensure the required time is below a specific value. The process can be repeated for other directions as well. Alternatively, we can combine different metrics together, such as speed and altitude.

In addition to additional metrics, more frequencies can be used in training to enable finer control of frequency and idle times. The drone may also use GPS coordinates to determine its environment and choose from a set of pre-programmed configurations. For example, a drone operating in a city

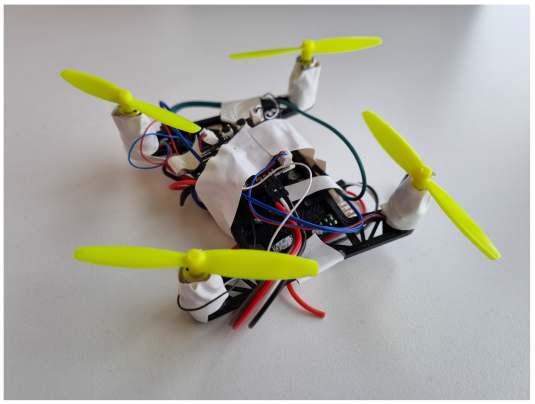


Fig. 10. Custom drone with the Ladybug flight controller.

may err on the side of caution and switch to a higher frequency at lower speeds than a drone operating in an open field.

6 PROTOTYPE

We demonstrate the operation of NeRTA in a system used for low-level control of aerial drones. We describe next the existing autopilot platform and the corresponding NeRTA implementation.

Autopilots. A drone autopilot is an embedded software library that includes the functionality needed for low-level control of an aerial drone. To control a drone while airborne, drone autopilots implement a control loop that uses as input data from on-board sensors and produces as output specific motor settings. The control loop must run at high enough frequencies, as otherwise the drone would react to changes too slowly and may destabilize or even crash.

Autopilots are generally written in low-level languages such as C [15] and C++ [1, 20] due to the limited computational resources available on typical flight controller hardware and the presence of strict real-time deadlines [26]. A large number of autopilots [1, 15, 20] are developed as open source software through a collaborative process.

Flight control is typically implemented through the use of **Proportional-Integral-Derivative (PID)** controllers that control the drone's motors depending on data from the sensors and the pilot's setpoints, which are sent through a **Radio Control (RC)** transmitter. Advanced autopilots such as ArduPilot [1] include flight modes, which provide varying amounts of automatic control, ranging from keeping the drone steady and stable to fully autonomous navigation.

Communication between the drone and the pilot is typically performed through radio transmission. A radio receiver is connected to the flight control hardware and is paired with a radio transmitter used by the pilot. RC protocols such as SBUS and CPPM [21] are used in drone autopilots.

Autopilots run on resource-constrained hardware and therefore prioritize efficiency to meet real-time deadlines. An example is the Ladybug flight controller, featuring a 32-bit STM32L432KC microcontroller with an 80-MHz single-core ARM M4F processor, 64 KB of SRAM, and 256 KB of non-volatile Flash memory [36]. The ARM M4F processor comes with an onboard **Floating-Point Unit (FPU)**, which is particularly important for autopilots due to the prevalence of floating-point operations. We use the Ladybug with a custom drone we build, shown in Figure 10.

Hackflight. We use Hackflight [20] as a concrete instance of a low-level robot controller. Hackflight is C++-based autopilot developed by Levy [20] that is designed to be well-structured,

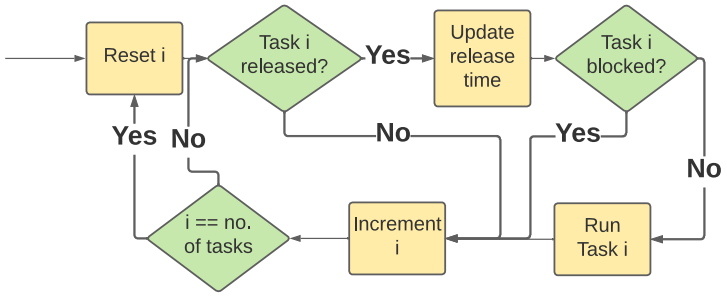


Fig. 11. Hackflight's scheduling logic demonstrating how Hackflight chooses which task to execute. Yellow indicates the executed action, and green indicates a binary decision to be taken.

powerful, and extensible. It is composed of at least three tasks: receiver, PID, and sensors. The receiver task processes data received through RC transmission and updates the drone's setpoints based on this. The PID task performs three different steps: it obtains the receiver setpoints; it runs the setpoints through all the PID controllers; and then it sends the setpoints generated by the final PID controller to the mixer, which converts the setpoints to individual motor speeds and sends them to the motors. Each sensor in Hackflight has its own task used to obtain data from it.

Figure 11 shows task scheduling in Hackflight. Tasks are ordered, which means the scheduler checks whether they can run in a specific order, which is receiver, then PID, then the sensor tasks. First, the scheduler checks if the current receiver job is released, and if so, it computes an updated release time. Job release times in Hackflight are computed similarly to the example in Section 3.3, by adding a fixed value to the time the prior job of that task started execution or was blocked. Blocking occurs when a task is refused to execute for reasons other than the task not being released. For example, the receiver task is blocked if there is no new frame received. No other tasks have blocking conditions. After updating the release time, the scheduler checks if the job is blocked. If the job is not blocked, then it executes. Then the scheduler repeats the same process with the PID task, then the sensor tasks. Once all the sensor tasks have gone through the process, a scheduling cycle is complete and the next cycle begins with the receiver task again.

NeRTA. Our implementation of NeRTA gathers necessary information about each task from Hackflight—that is, the time the previous job of the task started executing; the inter-arrival delay; the next release time, which is computed at runtime by summing the former two; and the task's criticality.

When implementing BRC, a key parameter is the number of possible frequencies. We strike a balance between a quick training procedure and a reactive system with three possible frequencies, defined as *high*, *medium*, and *low*. The *high* frequency is equal to the frequency of the control loop without BRC and is thus not part of the training. The *medium* and *low* frequencies are defined using the training procedure.

A key aspect is how to choose which optimization to apply. The procedure to do so is shown in Figure 12 and is repeated after the execution of each job, until the update is successfully scheduled. Initially, our implementation attempts to schedule the update with no optimizations. If scheduling fails, we attempt to schedule the update by only applying mixed-criticality settings and only consider high-criticality tasks. If scheduling fails again, we switch to BRC, which is used in the next cycle. BRC adjusts the frequency of the control loop depending on the speed of the drone. It starts off with the high-frequency setting for robustness reasons and reduces the frequency if the speed is sufficiently low. Cycles with BRC enabled begin slightly differently by invoking the BRC

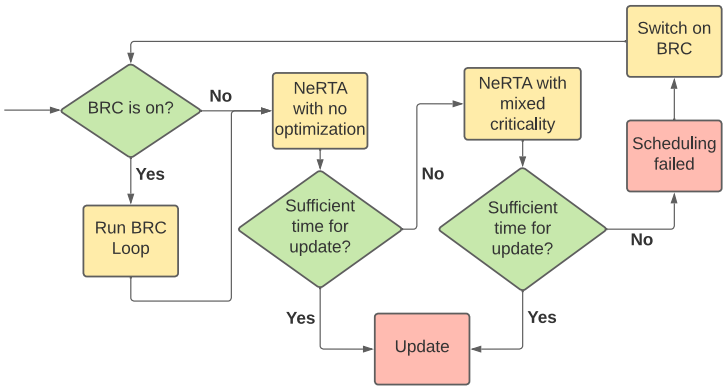


Fig. 12. Choice of optimization to apply. The color choices are the same as in Figure 11 with the addition of red for the final action taken of whether to schedule the update.

loop that updates the frequencies and computes new release times for the control loop and sensor tasks.

7 EVALUATION

Our evaluation is four-pronged. Section 7.1 compares NeRTA conservative estimates of idle time with actual executions. Section 7.2 quantifies the impact of the additional mechanisms we design to increase available idle times, whereas Section 7.3 investigates their impact on robust robot operation. We measure NeRTA's runtime overhead in Section 7.4.

The results we obtain lead to five key conclusions:

(1) The difference between NeRTA's estimates and actual idle times is less than 15% in more than 75% of the samples.
(2) All idle times larger than 0.6 ms exhibit differences less than 15%.
(3) Applying mixed-criticality concepts and BRC more than doubles the available idle times.
(4) The impact of these mechanisms on the robust robot operation is limited.
(5) The runtime overhead of NeRTA is essentially negligible compared to the dynamics at stake.

Our methodology combines real-world experiments on embedded hardware and simulations. We use our NeRTA prototype, described in Section 6, that embeds NeRTA in the aerial drone flight controller Hackflight, and MultiCopterSim described in Section 5.2.

7.1 NeRTA Estimates

We quantitatively determine how conservative NeRTA estimations are by comparing the idle time NeRTA computes with the idle time we measure during actual execution.

Setup. As in Section 4, we use the prototype of Section 6 while the drone is stationary but armed. We instrument our implementation to gain information on the actual task scheduling in the absence of non-deterministic environment influence, using Hackflight's default parameters for task scheduling.

We capture 4,881 samples of NeRTA estimates and of actual idle times. We compute the difference between corresponding samples as a measure of error. We exclude 571 samples from the analysis because they represent cases when one of the next release times is lower than the current time, and hence idle time is zero. At runtime, these situations are readily detected and only affect how many iterations of NeRTA we need before an update is scheduled successfully.

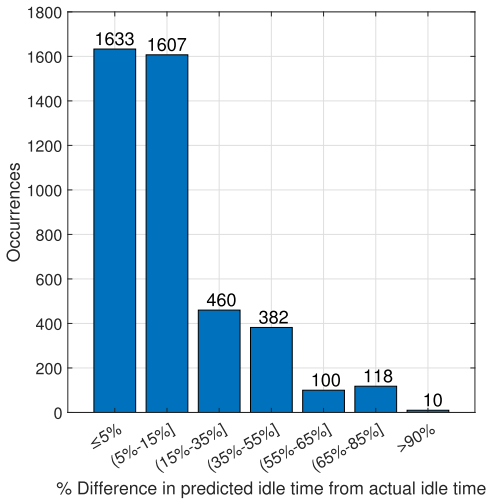


Fig. 13. Histogram of differences between actual idle times and NeRTA predicted idle times; note that a significant number of samples have a difference less than 15%.

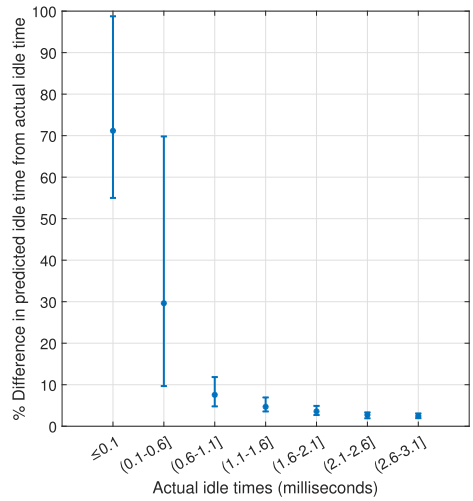


Fig. 14. Min, max, and average differences between actual idle times and NeRTA predicted idle times, grouped by actual idle times; note decreasing differences as idle times increase.

Results. Figure 13 shows the results of the experiment. We vary the size of the bins to highlight key results, namely the sample with the largest error has an error of 98.75%, 1,633 of the 4,310 samples that are left have an error of less than 5%, and three-quarters of the samples have an error of less than 15%. The largest difference in idle times is 86 μs.

The experiment demonstrates that, despite the conservative nature of NeRTA estimates, its measures of available idle times are close to the actual idle times. Figure 14 shows that large percentage-wise differences are observed only for small idle times. All idle times larger than 0.6 ms show differences below 15%.

Generally, it is more important to be accurate for larger idle times to schedule larger updates than to be accurate for small idle times that are not that useful per se, even if predicted accurately. This justifies our design choices and makes NeRTA an accurate estimator of idle times. Its conservative nature allows NeRTA to be schedule-invariant by design.

7.2 Impact of Additional Mechanisms

We measure the impact of applying mixed-criticality concepts and of BRC on available idle times.

Setup. We use the same setup as in the previous experiment. We emulate the deployment of an update 5 seconds after boot and provide NeRTA with a maximum of 60 seconds to find idle times sufficient to schedule the update. We repeat the experiment by varying the time required by the update stepwise by 0.1 ms. We repeat the experiment until we reach an update that cannot be scheduled within 60 seconds. We experimentally verify that increasing this time would not lead to different results. For each configuration, we record the largest available idle time NeRTA estimates.

Results. Figure 15 depicts the results. It shows that using only mixed-criticality concepts yields no improvement in the maximum available idle time compared with the regular configuration. BRC yields an improvement of around 14%. Applying both optimizations yields a significant improvement of around 155% over the regular configuration.

The reason for the results with mixed-criticality concepts is that the idle times are dominated by the task for the orientation sensor, which runs at 330 Hz. The receiver task is the only task

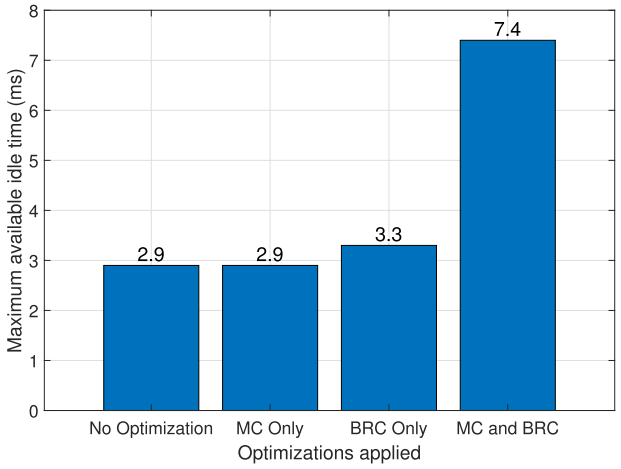


Fig. 15. The maximum available idle times with different additional mechanisms. MC refers to the application of mixed-criticality concepts.

affected by applying mixed-criticality concepts. It runs at 300 Hz, which is less than the 330 Hz of the orientation sensor. The idle times are generally determined by the task with the highest frequency because that task places an upper bound on the idle time equal to the interval between two of its jobs, which is shorter with higher frequency. Therefore, excluding the receiver task does not increase the idle times significantly because it is not the one that ultimately determines them.

BRC reduces the frequency of the orientation sensor task and of the control task, so the highest frequency task now becomes the receiver task. It has a frequency of 300 Hz, which is lower than the highest frequency task when applying only mixed-criticality concepts, which was 330 Hz for the orientation sensor task. This explains the slight improvement in the available idle times.

When we apply both optimizations, applying mixed-criticality concepts excludes the receiver task from NeRTA estimates, whereas the orientation sensor task and the control tasks are set to the *low* frequency of 100 Hz. The receiver task still runs at 330 Hz, but NeRTA does not account for that anymore. Therefore, the highest-frequency task taken into account now has a frequency of 100 Hz, which explains the significant improvement when we apply both mechanisms.

7.3 Impact of BRC on Robustness

We run experiments to determine the effect of BRC on robustness. Applying mixed-criticality concepts is robust by design, if the choice of less critical tasks is judicious.

Setup. We use the physics-accurate drone simulator MultiCopterSim [19], enabling controlled experiments that involve applying external forces to the drone. The drone simulated is a DJI Phantom, which is used in the BRC training phase in Section 5.2. We fly the drone in the simulator to an altitude of 20 m. We apply an external roll force for 1 second at 20 seconds after the start of the simulation. The product of the roll force and arm length, which is constant, is the torque. Since the arm length is constant, the torque is proportional to the force applied. We sample the drone speed 250 times per second as it attempts to hover and return to a speed of zero and record how reactive control behaves at various points in time. The speed is the magnitude of the velocities in x, y, and z directions.

We increase the intensity of the roll force by 1 N if the drone successfully hovers without crashing in an experiment, and keep doing so until the drone fails to hover. We call maximum roll force

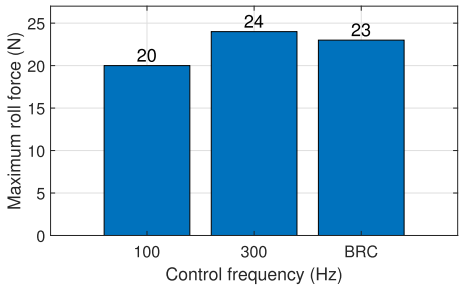


Fig. 16. The maximum roll force from which a successful hovering is achieved.

the largest force the drone successfully counteracts, eventually reaching a hovering configuration. We compare the maximum roll force between different configurations, including BRC, the default Hackflight static frequency of 300 Hz, and a static frequency of 100 Hz. Note that 300 Hz and 100 Hz are the upper and lower bounds of BRC's (dynamic) frequency range, respectively.

The force we apply models an intense pressure gradient differential [8] caused by the rotors on one side of the drone producing more thrust than on the other side, due to a difference in the air density. In practice, a pressure gradient would produce a smaller force than what we apply; our experimental setting provides a worst-case estimate of the control capabilities of BRC. The force applied is large but is only applied for 1 second, which is short enough that the drone can recover from, provided the control loop is fast enough.

Results. Figure 16 shows the results of these experiments. It demonstrates that BRC can successfully hover from a maximum roll force close to the maximum roll force that the static 300-Hz setting can hover from.

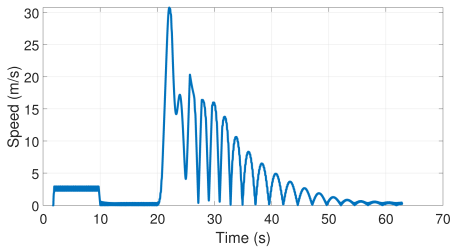
Figure 17 provides a deeper insight into the system behavior by showing the drone speeds observed when applying a 20-N roll force, which is the largest roll force that all three configurations successfully manage. Figure 17(a) demonstrates that the drone experiences significant oscillations when using a 100-Hz control frequency, meaning that it is close to the limit in terms of handling the roll force to regain a hovering configuration. Figure 17(b) and (c), however, are very similar to each other, showing that BRC has robustness similar to the static 300-Hz configuration, but can throttle down tasks when possible.

Figure 17(d) indeed demonstrates that with BRC the drone spends a significant amount of time using the 100-Hz frequency. Comparing Figure 17(b) and (d), it is possible to note that the periods running at 100-Hz frequency correspond to the lowest drone speeds, indeed, when a higher control frequency is not needed. Therefore, BRC allows NeRTA to find significantly larger idle times with a limited impact on robustness.

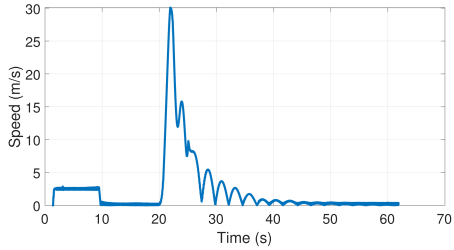
7.4 Overhead

NeRTA's implementation has two components. The first component computes release times and runs at the start of each task regardless of whether an update is pending. By doing so, NeRTA can attempt to schedule the update in the first iteration right after we issue the update. The second component is the update scheduling portion that runs at the end of each task only when there is an update pending.

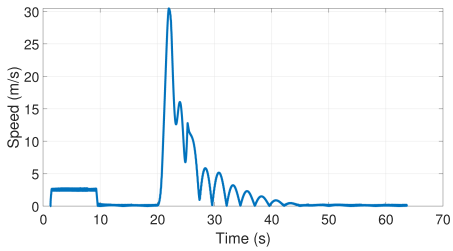
Both components have a performance impact on Hackflight because they require processing time. We measure how large this processing time is and compare that with the idle time estimates of NeRTA, to make sure the overhead does not significantly reduce the idle time. For similar reasons, we also measure the processing time of the reactive control loop, which dynamically chooses a



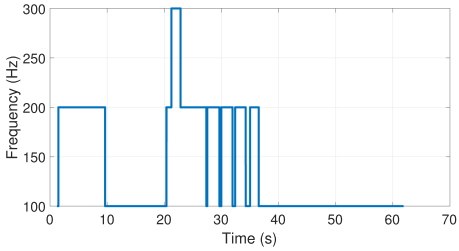
(a) Drone speed over time with a control frequency of 100 Hz.



(b) Drone speed over time with BRC enabled.



(c) Drone speed over time with a control frequency of 300 Hz.



(d) Control frequencies set by BRC over time.

Fig. 17. Different frequency configurations showing the effect of a 20-N force applied at the 20-second mark.

Table 3. Time Required for Different Stages of the Update Scheduling Process

Stage of update scheduling process	Average time (μs)	Max time (μs)	Samples
Release time update	1.76	27	212,328
NeRTA scheduling without mixed criticality	2.53	14	21,914
NeRTA scheduling with mixed criticality	8.71	24	88,057
Reactive control loop	7.42	29	126,221
Reactive control changing state	4.43	16	104,222
MC mitigation turning receiver task on	2.16	14	34,979
MC mitigation keeping receiver task off	2.20	28	35,024

control frequency for the current environment, and the mitigation to ensure low-criticality tasks are re-enabled safely.

NeRTA's processing overhead, indeed, is essentially subtracted from the idle times and therefore detrimental to the ability to accommodate updates. It should consequently be as low as possible.

Setup. We use the same setup as Section 7.1. Without mixed criticality, we schedule updates that take 1.5 ms to complete. We choose 1.5 ms because such an update can be performed without any additional mechanisms. For the rest of the measures, we schedule updates that take 4 ms to complete. We choose 4 ms because such an update requires both BRC and mixed-criticality concepts to be applied, allowing us to measure the performance impact of the additional mechanisms. We measure the overhead of the mixed-criticality mitigation both when it re-enables a low-criticality task (the receiver task) and when it does not (due to insufficient idle time to execute it).

Results. Table 3 reports the average and max processing times, as well as the number of measured samples, for each component. In the absence of additional mechanisms, most idle times are in the 0.6 to 1.6-ms range as indicated in Figure 8. The table demonstrates that the maximum time

overhead of all stages is around two orders of magnitude smaller than the common idle times. Based on these results, we conclude that the processing overhead of NeRTA is arguably negligible.

As expected, the results show that the overhead is negligible. The operations needed for different steps within NeRTA are extremely simple even for a microcontroller. Updating the release time is a simple addition, plus reading the current time if necessary. Scheduling without mixed criticality reads the release time of each task to find the smallest one, subtracts the current time from the minimum value, and compares the result with the update time required. Scheduling with mixed criticality takes a little longer because it actually checks twice: once taking into account all tasks, and once taking into account only the high-criticality ones. The reactive control loop performs a square root to compute the drone's speed, which is fast due to the FPU on the target microcontroller. Then it compares the speed with pre-defined values. Changing the state computes and updates task frequencies using multiplications and divisions of floating-point numbers, which is fast due to the FPU. The mixed-criticality mitigation performs similar operations to the NeRTA scheduling and has similar overhead.

8 VALIDITY, THREATS, AND LIMITATIONS

We outline key considerations and limitations of our design and methodology.

Simulation for BRC Training and Robustness. We perform the evaluation in the same environment where BRC is trained. This is in line with the original design of reactive control, where the training procedure happens online rather than offline. The training procedure must accurately reflect the features of the target environment [11]. As with the original reactive control, our results do not demonstrate how well BRC behaves in environments it was not trained for.

Our training procedure is simple with only three possible frequency settings and only one parameter—that is, speed—that determines the frequency. A more complex training procedure can help in case the training approach presented here is ineffective. For example, one may increase the number of possible frequencies and include additional parameters to determine the frequency setting.

The simulator models the drone's dynamics and the environment. However, the simulator does not simulate the drone's computing capabilities. The control tasks are executed on the machine running the simulator, which is significantly more powerful than the microcontroller aboard the drone. Therefore, task execution times in the simulation are significantly smaller than in reality. This is not a significant threat to our results, since the impact of the BRC execution times is negligible even on the drone microcontroller.

Moreover, it is worth noting that the robustness experiment only evaluates the impact of BRC, not of NeRTA as a whole or of the application of mixed-criticality concepts. The former is not a significant limitation since the runtime overhead of NeRTA is negligible, and therefore the effect of NeRTA's operations on task scheduling is negligible as well. The latter is not an acute issue either, since the application of mixed-criticality concepts only affects the receiver task in our setting, which by the definition of low-criticality tasks is not needed for robust operation.

Hackflight. Focusing on Hackflight for the experimental evaluation may, in principle, limit the generality of our results. The results for the accuracy of the NeRTA's idle times depend on the length of time between the release time of a job and on when the job begins its execution. In scheduling techniques where the entire schedule is determined statically, however, the release time and start of execution time are equal. Therefore, NeRTA's idle times would be more accurate in this case than for a dynamic schedule, such as that of Hackflight.

In other dynamic scheduling techniques, such as earliest deadline first, the differences in idle times depend on the system utilization. Hackflight shows a fairly low system utilization, so tasks often execute close to their release time. The same applies to other systems where the system

utilization is low, provided the scheduling algorithm is priority-based. A priority-based scheduler with low system utilization likely has few tasks released at the same time [32], and therefore the response time of the tasks is low. The release times are then close to the start of execution times. The validity of our results extend to these systems too.

In contrast, we expect our results not to apply to systems with high utilization. In these cases, the idle times are expected to be so small that scheduling updates in idle times is unfeasible regardless. Our results do not generalize to time slice based schedulers such as round-robin scheduling either: as we mention in Section 3.1, NeRTA is not applicable to algorithms that do not provide guaranteed inter-arrival times. Round-robin scheduling can provide a release time for tasks outside the current time slice, but it cannot do so for the task inside the current time slice.

Update Model. We do not perform an actual update during our evaluation. Instead, we model the update as a time-bound task whose worst-case execution time is known. Since our focus is *when* to schedule the update rather than performing the update itself, this approach does not restrict the validity of our results.

We only schedule single-stage updates in the evaluation. In Section 3.2, nonetheless, we discuss how an update can be split into multiple stages if the system meets a set of application-specific requirements between each stage. Because scheduling multi-stage updates simply uses the same scheduling process for each stage, if the scheduling process is schedule-invariant for one stage, then it is schedule-invariant for more than one.

9 CONCLUSION

NeRTA estimates the available idle times in existing task schedules of low-level robot controllers to accommodate schedule-invariant dynamic software updates. Its operation is orthogonal to the existing scheduler, retaining the existing platform-specific optimizations and fine-tuning, whereas its estimates are cautiously conservative. To accommodate larger updates, we applied mixed-criticality concepts and developed additional mechanisms, such as BRC. Our evaluation showed that the difference between NeRTA's estimated idle times and the measured idle times is less than 15% in more than three-quarters of the samples, whereas the combined effect of BRC and mixed-criticality concepts yields a 150+% increase in available idle times. We also showed that the processing overhead of our techniques is essentially negligible.

REFERENCES

[1] ArduPilot. 2021. ArduPilot Home Page. Retrieved September 18, 2023 from https://ardupilot.org/

[2] Texas Instruments. 2021. MSP430FR6007x Datasheet. Retrieved September 18, 2023 from https://www.ti.com/lit/ds/symlink/msp430fr6007.pdf?ts=1635260016102

[3] ArduPilot. 2022. Threading. Retrieved September 18, 2023 from https://ardupilot.org/dev/docs/learning-ardupilot-threading.html

[4] Applied Aeronautics. 2022. Albatross: The Affordable, Long Range Drone. Retrieved September 18, 2023 from https://www.appliedaeronautics.com/albatross-uav

[5] Pixhawk. 2022. Pixhawk Home Page. Retrieved September 18, 2023 from https://pixhawk.org/

[6] PX4. 2022. PX4 Home Page. Retrieved September 18, 2023 from https://px4.io/

[7] TurtleBot. 2022. TurtleBot3. Retrieved September 18, 2023 from https://www.turtlebot.com/turtlebot3/

[8] Mikhail Afanasov, Alessandro Djordjevic, Feng Lui, and Luca Mottola. 2019. FlyZone: A testbed for experimenting with aerial drone applications. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'19)*. ACM, New York, NY, 67–78. https://doi.org/10.1145/3307334.3326106

[9] Babiker Hussien Ahmed, Sai Peck Lee, Moon Ting Su, and Abubakar Zakari. 2020. Dynamic software updating: A systematic mapping study. *IET Software* 14, 5 (2020), 468–481. https://doi.org/10.1049/iet-sen.2019.0201

[10] S. Bouabdallah, P. Murrieri, and R. Siegwart. 2004. Design and control of an indoor micro quadrotor. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation (ICRA'04)*. 4393–3498. https://doi.org/10.1109/ROBOT.2004.1302409

[11] Endri Bregu, Nicola Casamassima, Daniel Cantoni, Luca Mottola, and Kamin Whitehouse. 2016. Reactive control of autonomous drones. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)*. ACM, New York, NY, 207–219. https://doi.org/10.1145/2906388.2906410

[12] Alan Burns and Robert I. Davis. 2019. *Mixed Criticality Systems—A Review*. University of York, York, UK

[13] Walter Cazzola and Mehdi Jalili. 2016. Dodging unsafe update points in Java Dynamic Software Updating systems. In *Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE'16)*. 332–341. https://doi.org/10.1109/ISSRE.2016.17

[14] Antoine Chaillet, Antonio Loria, and Rafael Kelly. 2006. Robustness of PID-controlled manipulators with respect to external disturbances. In *Proceedings of the 45th IEEE Conference on Decision and Control*. 2949–2954. https://doi.org/10.1109/CDC.2006.377362

[15] Dominic Clifton. 2017. Cleanflight Home Page. Retrieved September 18, 2023 from http://cleanflight.com/

[16] Davison. 2003. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings of the 9th IEEE International Conference on Computer Vision*. 1403–1410. https://doi.org/10.1109/ICCV.2003.1238654

[17] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. 2013. Lightweight framework for runtime updating of C-based software in embedded systems. In *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades (HotSWUp'13)*. https://www.usenix.org/conference/hotswup13/workshop-program/presentation/holmbacka

[18] J. Lehoczky, L. Sha, and Y. Ding. 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 1989 Real-Time Systems Symposium*. 166–171. https://doi.org/10.1109/REAL.1989.63567

[19] Simon Levy. 2022. MulticopterSim. Retrieved September 18, 2023 from https://github.com/simondlevy/MulticopterSim

[20] Simon D. Levy. 2021. Hackflight Home Page. Retrieved September 18, 2023 from https://github.com/simondlevy/Hackflight

[21] Oscar Liang. 2021. RC Protocols Explained. Retrieved September 18, 2023 from https://oscarliang.com/rc-protocols/

[22] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. 1995. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering* 21, 7 (1995), 593–604. https://doi.org/10.1109/32.392980

[23] Razika Lounas, Nisrine Jafri, Axel Legay, Mohamed Mezghiche, and Jean-Louis Lanet. 2017. A formal verification of safe update point detection in dynamic software updating. In *Risks and Security of Internet and Systems*. Lecture Notes in Computer Science, Vol. 10158. Springer, 31–45. https://doi.org/10.1007/978-3-319-54876-0_3

[24] Danijel Mlinarić. 2021. Challenges in dynamic software updating. *TEM Journal* 9, 1 (2021), 13.

[25] Dennis K. Nilsson, Lei Sun, and Tatsuo Nakajima. 2008. A framework for self-verification of firmware updates over the air in vehicle ECUs. In *Proceedings of the 2008 IEEE Globecom Workshops*. 1–5. https://doi.org/10.1109/GLOCOMW.2008.ECP.56

[26] Tammy Noergaard. 2013. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Elsevier.

[27] Andrea Patelli and Luca Mottola. 2016. Model-based real-time testing of drone autopilots. In *Proceedings of the 2nd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use (DroNet'16)*. ACM, New York, NY, 11–16. https://doi.org/10.1145/2935620.2935630

[28] Florian Rommel, Christian Dietrich, Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. 2020. From global to local quiescence: Wait-free code patching of multi-threaded processes. 651–666. https://www.usenix.org/conference/osdi20/presentation/rommel

[29] Adam Seewald, Hector Garcia De Marina, Henrik Skov Midtiby, and Ulrik Pagh Schultz. 2020. Mechanical and computational energy estimation of a fixed-wing drone. In *Proceedings of the 2020 4th IEEE International Conference on Robotic Computing (IRC'20)*. 135–142. https://doi.org/10.1109/IRC.2020.00028

[30] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. 2013. A survey of dynamic software updating. *Journal of Software: Evolution and Process* 25, 5 (2013), 535–568. https://doi.org/10.1002/smr.1556

[31] Wei Kuan Shih, J. W. S. Liu, and C. L. Liu. 1993. Modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines. *IEEE Transactions on Software Engineering* 19, 12 (Dec. 1993), 1171–1179. https://doi.org/10.1109/32.249662

[32] Spuri and Buttazzo. 1994. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 1994 Proceedings Real-Time Systems Symposium*. 2–11. https://doi.org/10.1109/REAL.1994.342735

[33] STMicroelectronics. 2021. STM32L432KC Datasheet. Retrieved September 18, 2023 from https://www.st.com/en/microcontrollers-microprocessors/stm32l432kc.html

[34] Nils Vreman, Anton Cervin, and Martina Maggio. 2021. Stability and performance analysis of control systems subject to bursts of deadline misses. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems (ECRTS'21)*.

[35] Michael Wahler, Stefan Richter, and Manuel Oriol. 2009. Dynamic software updates for real-time systems. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades (HotSWUp'09)*. ACM, New York, NY, 1–6. https://doi.org/10.1145/1656437.1656440

[36] Kris Winer. 2021. Ladybug Flight Controller. Retrieved September 18, 2023 from https://www.tindie.com/products/TleraCorp/ladybug-flight-controller/

[37] Zelin Zhao, Xiaoxing Ma, Chang Xu, and Wenhua Yang. 2014. Automated recommendation of dynamic software update points: An exploratory study. In *Proceedings of the 6th Asia-Pacific Symposium on Internetware on Internetware (INTERNETWARE'14)*. ACM, New York, NY, 136–144. https://doi.org/10.1145/2677832.2677853

[38] Ze-Lin Zhao, Di Huang, and Xiao-Xing Ma. 2022. TOAST: Automated testing of object transformers in dynamic software updates. *Journal of Computer Science and Technology* 37, 1 (2022), 50–66.

Received 29 November 2022; revised 24 May 2023; accepted 21 August 2023