# NeRTA: Enabling Dynamic Software Updates in Mobile Robotics

Ahmed El Yaacoub*, Luca Mottola*†, Thiemo Voigt*, Philipp Rümmer*
*Uppsala University; †Politecnico di Milano
{ahmed.el.yaacoub; luca.mottola; thiemo.voigt; philipp.ruemmer}@it.uu.se

## Abstract

We present NeRTA (Next Release Time Analysis), a technique to schedule dynamic software updates of the low-level control loops 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 loops of mobile robots, however, are time sensitive and run 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. Our experimental evaluation shows that NeRTA estimates are within 15% of the actual idle times in more than three-quarters of the cases. We also show that the processing overhead of NeRTA is essentially negligible.

## Categories and Subject Descriptors

Embedded and cyber-physical systems [**Embedded systems**]: Embedded software

## General Terms

Minimal effect, updates, bug-fixes, software evolution

*Keywords*

Dynamic software updates, mobile robotics, safety-critical systems, aerial drones

## 1 Introduction

Dynamic software updates are performed without interrupting the software execution [7]. They are useful for applications that require frequent updates but must operate continuously, with no perceivable downtime. In mobile robotics, dynamic software updates are useful for resolving software defects that may endanger the robot or its surroundings [16] and to extend the robot's capabilities by adding new features.

**Mobile robotics.** Consider for example aerial drones. Fixed-wing drones [1] may fly for hours performing search-and-rescue missions. During this time, the need to patch the running software or to deploy a new functionality may arise in response to unforeseen situations requiring adjustments to the flight software or bugs discovered during the flight [18, 19, 20], each of 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 and the additional landing and take-off [17]. 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 [8]. 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 [11]. Unlike the high-level control loop, the operation of the low-level one is extremely time sensitive. Existing systems employ custom implementations running on resource-constrained hardware with no operating system [10, 13].

**Problem and fundamental idea.** We tackle the problem of *when* to perform a dynamic update of *the low-level control loop* without being detrimental to the dependable robot operation. Unlike the high-level control loop, if the low-level control loop is unresponsive for some time or delays the execution of crucial tasks, the robot loses control because its actuators receive incorrect or late values compared to the robot state [16]. 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.

Our fundamental idea is that a safe time to schedule dynamic updates with *minimal effect* is when the computing unit running the low-level control loop would otherwise be idle, that is, when no tasks are running. Minimal effect means that tasks running on the mobile robot maintain the same exact scheduling as if the update did not take place.

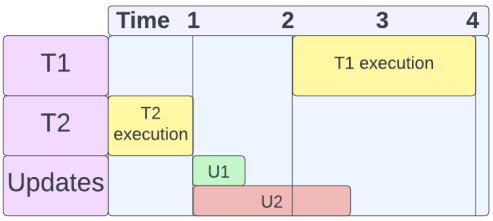

Figure 1: An example of an update U1 with minimal effect and an update U2 that violates the minimal effect condition.

We consider an update to be a non-preemptive and time-bound task, whose worst-case execution time is known. We show an update that meets the minimal effect condition and one that does not in Fig. 1. Update U1 starts and completes when no tasks are running. Update U2 extends into the portion of time when T1 is to execute. As a result, update U2 necessarily delays the start of T1. The availability of idle time is due to the scheduler leaving the interval between times one and two empty, which we utilize to perform an update U1 that meets the minimal effect condition.

**NeRTA.** To perform dynamic software updates with minimal effect we develop NeRTA (Next Release Time Analysis), a technique that dynamically estimates the available idle times. We use the estimated idle time to check for the schedulability of the update according to the minimal effect requirement. We schedule an update using NeRTA only if the time taken for the update is less than or equal to the NeRTA estimate. These estimates are *conservative*, that is, 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, thus retaining the minimal effect condition.

This feature represents a *fundamental and intentional* design choice. The alternative would be, indeed, to embed software updates as an additional task in the original scheduler design. However, such a task would be aperiodic, unlike most other tasks in low-level control of mobile robots, and rare compared to regular operation. These factors would greatly complicate the scheduler design, implementation, and testing. Instead, we schedule the update with NeRTA. NeRTA operates *orthogonal* to the existing scheduler, retaining the extensive testing, verification, 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 Sec. 3, NeRTA operates as a plug-in component in existing systems, while satisfying the condition of minimal effect.

**Performance.** While 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 by integrating NeRTA in the task scheduler of Hackflight, an open-source aerial drone flight controller [13]. We build a custom drone running Hackflight on embedded hardware and use that to evaluate our work. Our prototype is described in Sec. 4.

Our experiments, reported in Sec. 5, reveal that NeRTA estimates, while being conservative, are close to the actual idle times. In more than three-quarters of the cases we measure, NeRTA estimates are less than 15% from the actual idle times. We also demonstrate that the price to pay, represented in terms of processing overhead, is negligible.

Before moving on to the technical matter, Sec. 2 provides a brief survey of related work.

## 2 Related Work

Six fundamental aspects are to be considered in dynamic software updates [15]:

1) *What fraction of the software is replaced*, from the entire program to individual instructions;
2) *Dependency analysis*, that is, determining whether two components are dependent, and what to do if so;
3) *State transfer*, that is, how to transfer and/or modify state from the older to the newer version, if necessary;
4) *Cleaning*, that is, how the leftover state of the older version should be disposed of, if necessary;
5) *Rollback*, that is, the ability to dynamically undo an update should there be a need for that;
6) *Time of the update*, that is, when to perform the update while the system keeps running.

We focus on 6) which is crucial in time-sensitive software implementations, such as low-level robot controllers [8]. Here, an additional consideration is also possible, which is the physical effect of the update process on the robot and its surroundings [12]. We seek to minimize this by fulfilling the *minimal effect condition*.

To decide when to schedule an update, quiescent points should be classified and identified. Quiescent points are defined as intervals in time when an update can be safely performed. Quiescent points can either be inserted into the program by the programmer or compiler, or identified automatically during runtime [21, 23]. 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. [21], 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 [9, 14, 23]. Cazzola and Jalili [9] 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 from a source code perspective, rather than on the physical effects of the update on the device: a robot in our case.

Lounas et al. [14] 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. While those properties are important, they do not ensure that the update is performed safely from a physical standpoint.

## 3 NeRTA: Next Release Time Analysis

We describe the design of NeRTA (Next Release Time Analysis), a technique to estimate the *duration* of idle times in an existing schedule.

### 3.1 Target Systems

We target systems implemented with a fixed number of tasks, each of which is responsible for executing an unbounded number of computation jobs. 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. NeRTA is therefore applicable to systems where the *release time* can be computed for each job. A job is guaranteed not to execute before its release time. The release time is computed by the task scheduling algorithm. Once a job executes, a new release time is computed for the same task. 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. NeRTA is a runtime technique in the sense that it computes idle times as the controller 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, 13]. Therefore, NeRTA enjoys wide applicability, especially in mobile robotics.

In contrast, NeRTA is not applicable to task scheduling algorithms without guaranteed inter-arrival delays. Without guaranteed inter-arrival delays, a release time that is larger than the current time cannot be provided. NeRTA is also not applicable when interrupts are enabled and tasks may be preempted. The release time for an interrupt is generally to be considered as the current time, because interrupts may fire at any point. In both cases, a guaranteed idle time cannot be computed and therefore NeRTA cannot be used.

### 3.2 Update Model

We consider an update to be an aperiodic task composed of at least one job. Each job in the update must have a known worst case execution time. All jobs in the update are non-preemptive. Only jobs that use up execution time on the low-level controller need to be scheduled with NeRTA before they are performed, so that the minimal effect requirement is met. Each job in the update task can be considered a stage in the update process. The number, choice, and order of stages is a decision we leave to the update developer, with the only requirement that the system must be considered correct both before and after each individual stage is applied.

In the rest of the paper, without loss of generality, we consider a specific (valid) update model, shown in Fig. 2 and representative of a vast class of mobile robotics platforms [4, 5, 6]. The downloading job is performed on a companion computer, for example, through a cellular connection to download a diff file. The loading job loads the diff file onto the low-level controller using DMA. Once loading is complete, NeRTA is invoked to ensure that the last stage in the process, which updates machine code and program state, meets the minimal effect requirement. We update the entire machine code and program state in one go to simplify safety checks, since updating parts of the software means each intermediate version must be verified to be safe. Both the downloading job and the loading job, since it uses a DMA, are performed in parallel with the regular execution, and therefore they do not require scheduling with NeRTA. Only the final stage uses execution time on the low-level controller device and therefore can impact the control tasks. This is therefore the only job scheduled with NeRTA.

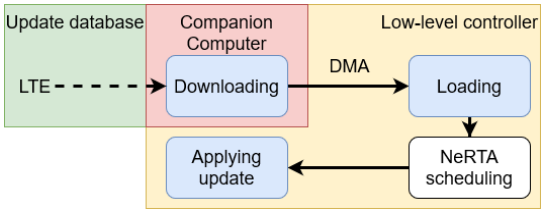


Figure 2: Update stages with NeRTA used to schedule the stage requring execution time from low-level controller.

**Security considerations.** The update model we consider presents key security considerations that must be taken into account to securely update the software. One is protection against man-in-the-middle attacks where an attacker would modify the diff file sent to the companion computer, or impersonates the machine sending the diff file to send an invalid one. To protect against man-in-the-middle-attacks, we can load a private key during initial offline programming of the companion computer, which is only known to the computer sending the diff file. The key is used to encrypt the binary before transmission, and decrypt it upon reception. A checksum encrypted with the same key can be packaged with the binary to verify its integrity.

### 3.3 NeRTA

We start with an example to provide the basic intuition behind how NeRTA estimates idle times, and describe its generalization next.

**Example.** Fig. 3 helps understand NeRTA's design. We consider three tasks, T1, T2, and T3. In a low-level control loop for drones, for example, T1 may be the flight control task, T2 the receiver task processing commands from a remote controller, and T3 the sensor task that gathers data from the IMU. As in existing low-level robot controllers [2, 10, 13], 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 reached 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 completes, we add five time units for T2's jobs, and seven time units for T3's jobs. Those values are a property of the task scheduling algorithm and are chosen to satisfy real-time requirements. In this example, we are currently at time four in Fig. 3, therefore one job from each task executed.

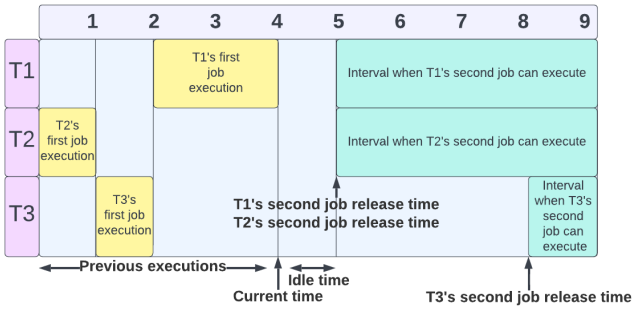


Figure 3: Example demonstrating release times of jobs in a system. The current time is four.

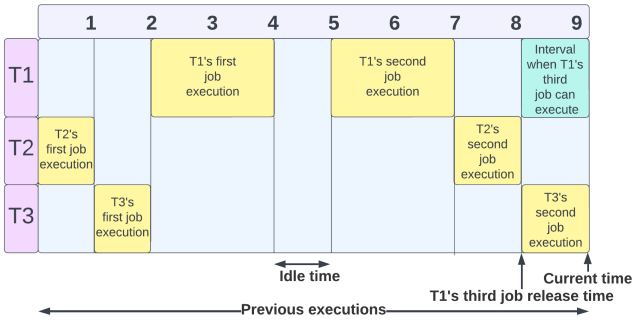


Figure 4: The system of Fig. 3 after the execution of the second job of all tasks. Current time is nine.

Fig. 4 shows the actual execution of the system as observed until time nine. At time five, 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 one time unit to when its second job completes, resulting in a new release time of eight. At time seven, 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 its second job executes. At time eight, 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.

Fig. 4 shows an idle time worth one time unit between time four and five. This happens as the earliest release time of any task in Fig. 3 is at time five. Between the current time and the earliest release time, no jobs can possibly execute, thus 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 preexisting jobs would be affected. As a result, updates performed during this interval meet the condition of minimal effect *by construction*. NeRTA generalizes this reasoning, as explained next.

**General case.** NeRTA requires two inputs, an update job with a specified worst case execution time, and the release times of jobs that belong to various tasks in the system. 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 the idle time. NeRTA computes the estimated idle time by subtracting the current time from the smallest release time of any job. 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 now, the task scheduler waits until the next job completes and calls NeRTA again with updated release time information.

The idle time NeRTA computes is a *conservative* estimate of the actual idle time of the concrete execution. Conservative means that NeRTA estimates cannot be larger than 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. Therefore, if the release times provided are correct, then by construction the idle time computed from those release times is conservative. The conservative nature of NeRTA estimates is extremely important because if the estimated idle time was larger than the actual idle time, then performing the update in this time span would necessarily postpone the execution of jobs, and therefore violate the minimal effect condition.

## 4 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, otherwise, the drone would react to changes too slowly and may destabilize or even crash.

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.

**Hackflight.** We use Hackflight [13] as a concrete instance of low-level robot controller. Hackflight is a C++ based autopilot developed by Levy [13] that is designed to be well-structured, powerful, and extensible.

Hackflight 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; 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. A sensor

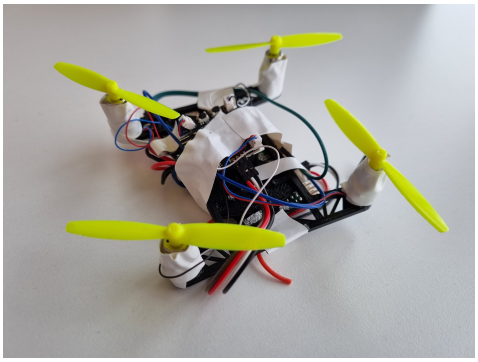

Figure 5: Custom drone with Ladybug flight controller.

to measure orientation is mandatory.

Scheduling in Hackflight is performed on a per-task basis; each task must meet certain requirements before it can run. Tasks in Hackflight have timing requirements that are dependant on whether the release time of the task is met or not. Task release times are computed in the exact same way as the example from Sec. 3.3, by adding a fixed value to the previous time the task started execution. Once the current time exceeds the release time, the task can execute.

**Hardware.** Autopilots run on resource-constrained flight controllers, and therefore prioritize efficiency to meet real-time deadlines to safely control the drone. An example is the Ladybug flight controller, featuring a 32-bit STM32L432KC microcontroller with an 80MHz single-core ARM M4F processor, 64 KB of SRAM, and 256 KB of non-volatile Flash memory [22]. The ARM M4F processor comes with an on-board floating-point unit (FPU), which is particularly important for autopilots due to the prevalence of floating-point operations. We use the Ladybug aboard a custom drone we build, shown in Fig. 5.

# 5 Evaluation

Our evaluation is three-pronged. In Sec. 5.1, we measure idle times observed in Hackflight and later use it for comparison with the other parts of the evaluation. Sec. 5.2 reports on how close NeRTA's (conservative) estimates of idle time are to actual executions. We conclude by measuring the run-time overhead of NeRTA in Sec. 5.3.

The results we obtain lead to two key conclusions:

1. NeRTA estimates are within 15% of the actual idle times in more than 75% of the cases;
2. the run-time overhead of NeRTA is essentially negligible compared to the timing dynamics at stake.

## 5.1 Idle Time Analysis

NeRTA conservatively estimates the available idle times in an existing task schedule. To reason about the accuracy of the idle times predicted, and the run-time overhead of NeRTA, we measure the idle times observed in Hackflight. Using an existing autopilot implementation and real hardware, we can measure the idle times that are present in a real-world mobile robot platform, and provide a reference for our results in other parts of the evaluation.

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

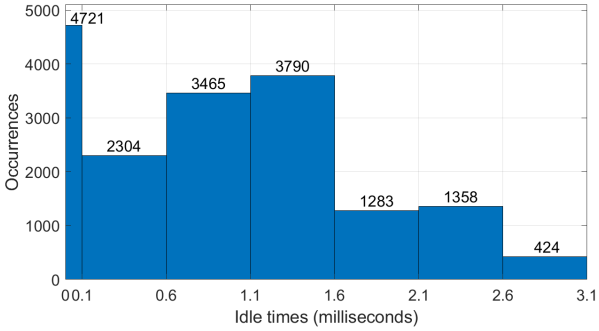


Figure 6: Idle times in our experiments with Hackflight; note the prevalence of idle times between 0.6 ms and 1.6 ms.

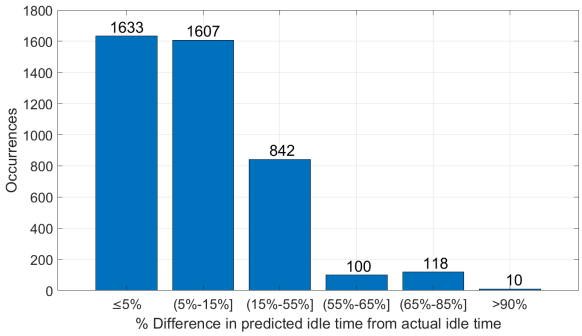


Figure 7: Histogram of differences between actual idle times and NeRTA predicted idle times; note that a significant number of samples have a difference less than 15%.

can derive the available idle times *post-facto*. Fig. 6 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. On the other hand, 10.9% of the samples are larger than 2.0 ms, and 46.1% are larger than 1.0 ms. Overall, we find median and average idle times of 0.8 ms and 0.9 ms respectively.

Note that in Fig. 6 the size of the bins is not homogeneous. In the 0-0.1 ms interval, we obtain numerous small samples because two or more jobs have overlapping release times, and therefore execute one right after the other.

## 5.2 NeRTA Estimates

We quantitatively determine how conservative are NeRTA estimations of the available idle times. We do so by comparing the idle time NeRTA computes with the idle time that we measure during actual execution.

**Setup.** We use the prototype of Sec. 4 without the motors attached and in stable conditions. 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 the 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 run-time, these situations are readily detected and only affect how many iterations of

Table 1: Time required for different stages of the update scheduling process

| Stage of update scheduling process | Average time (µs) | Max time (µs) |
|---|---|---|
| *Release time update* | 1.86 | 9 |
| *NeRTA scheduling* | 1.44 | 6 |

NeRTA we need before an update is scheduled successfully.

**Results.** Fig. 7 shows the results of the experiment. We vary the widths 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%.; 75% of the samples have an error of less than 15%. We also compute the largest difference in idle times to be 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. Despite some large percentage-wise differences, those large differences were observed only for small idle times, since the largest absolute difference was only 86 µs between the predicted and actual idle times, which is within 15% of the median idle time of 0.8 ms. This shows that for idle times larger than the median, the percentage-wise differences are small, i.e., less than 15%. It is more important to be accurate for larger idle times to be able to schedule updates that need more time, than to be accurate for small idle times that are not that useful for scheduling larger updates even if predicted accurately. This justifies our design choices and makes NeRTA an accurate estimator of the idle times. Its conservative nature allows NeRTA to meet the minimal effect condition by design.

## 5.3 Overhead

NeRTA's implementation has two components, each of which consumes valuable CPU time at regular intervals. We measure the average and maximum CPU time used for each component. The first component computes release times and runs at the start of each task regardless of whether an update is pending. By running regardless of whether an update is pending, NeRTA attempts to schedule the update in the first iteration after we issue it. The second component is the update scheduling portion that runs at the end of each task only when there is an update pending.

**Setup.** We use the same setup as Sec. 5.2. We schedule an update that takes 2 ms to complete. We choose 2 ms because such an update is possible in the idle times available from Sec. 5.1. We separately measure the average processing times of each component. We take 13,348 samples of the first component, and 847 samples of the second component.

**Results.** Tab. 1 reports the average and max processing times observed for each component. The median idle time is 0.8 ms as indicated in Sec. 5.1. The table demonstrates that the max times for either stage are over two orders of magnitude smaller than the median idle time. Based on these results, we conclude that the processing overhead of NeRTA is arguably negligible.

# 6 Conclusions

NeRTA estimates the available idle times in existing task schedules of low-level robot controllers to accommodate dynamic software updates with *minimal effect* on the robot operation. Its operation is orthogonal to the existing scheduler, retaining the existing platform-specific optimizations and fine-tuning, while its estimates are cautiously conservative. Our evaluation shows that NeRTA estimates are within 15% of the actual idle times in more than three-quarters of the cases we measure. We also show that the processing overhead of NeRTA is essentially negligible.

**Acknowledgements.** Work partially funded by the Knut and Alice Wallenberg Foundation through project UPDATE.

# References

[1] Albatross uav : Long range drone. URL https://www.appliedaeronautics.com/albatross-uav.

[2] Ardupilot homepage, Jul 2021. URL https://ardupilot.org/.

[3] 2022. URL https://ardupilot.org/dev/docs/learning-ardupilot-threading.html.

[4] Px4, May 2022. URL https://px4.io/.

[5] Pixhawk, May 2022. URL https://pixhawk.org/.

[6] Turtlebot3, May 2022. URL https://www.turtlebot.com/turtlebot3/.

[7] B. H. Ahmed, S. P. Lee, M. T. Su, and A. Zakari. Dynamic software updating: a systematic mapping study. 2020. doi: 10.1049/iet-sen.2019.0201.

[8] E. Bregu, N. Casamassima, D. Cantoni, L. Mottola, and K. Whitehouse. Reactive Control of Autonomous Drones. 2016. URL https://doi.org/10.1145/2906388.2906410.

[9] W. Cazzola and M. Jalili. Dodging Unsafe Update Points in Java Dynamic Software Updating Systems. 2016. doi: 10.1109/ISSRE.2016.17.

[10] D. Clifton. Cleanflight, Apr 2017. URL http://cleanflight.com/.

[11] Davison. Real-time simultaneous localisation and mapping with a single camera. 2003. doi: 10.1109/ICCV.2003.1238654.

[12] V. Kangunde, R. S. Jamisola, and E. K. Theophilus. A review on drones controlled in real-time. 2021. URL https://doi.org/10.1007/s40435-020-00737-5.

[13] S. D. Levy. Hackflight homepage, Jun 2021. URL https://github.com/simondlevy/Hackflight.

[14] R. Lounas, N. Jafri, A. Legay, M. Mezghiche, and J.-L. Lanet. A Formal Verification of Safe Update Point Detection in Dynamic Software Updating. pages 31–45, Cham, 2017. doi: 10.1007/978-3-319-54876-0_3.

[15] D. Mlinarić. Challenges in Dynamic Software Updating. *TEM Journal*, 9(1):13, 2021.

[16] A. Patelli and L. Mottola. Model-based Real-time Testing of Drone Autopilots. pages 11–16, June 2016. URL https://doi.org/10.1145/2935620.2935630.

[17] A. Seewald et al. Mechanical and Computational Energy Estimation of a Fixed-Wing Drone. In *2020 IRC*, pages 135–142, 2020. doi: 10.1109/IRC.2020.00028.

[18] A. Tridgell. Ardupilot adding a check for pitch considering current airspeed, Mar 2022. URL https://github.com/ArduPilot/ardupilot/commit/6ebefbdb1680b61efc30d74ebcd95861f8adf02a.

[19] A. Tridgell. Ardupilot rudder control fix, Apr 2022. URL https://github.com/ArduPilot/ardupilot/commit/8e37c93e7d7716885f97addd941fa0df3b47040d.

[20] A. Tridgell. Ardupilot yaw fix when disarming rudder, Mar 2022. URL https://github.com/ArduPilot/ardupilot/commit/48881eeb5517d2804f929aa3d56b68f958f73772.

[21] M. Wahler, S. Richter, and M. Oriol. Dynamic software updates for real-time systems. HotSWUp, 2009. URL https://doi.org/10.1145/1656437.1656440.

[22] K. Winer. Ladybug flight controller, Jun 2021. URL https://www.tindie.com/products/TleraCorp/ladybug-flight-controller/.

[23] Z. Zhao, X. Ma, C. Xu, and W. Yang. Automated recommendation of dynamic software update points: an exploratory study. INTERNETWARE, 2014. URL https://doi.org/10.1145/2677832.2677853.