Poster Abstract: Scheduling Dynamic Software Updates in Safety-critical Embedded Systems - the Case of Aerial Drones

Ahmed El Yaacoub, Luca Mottola, Thiemo Voigt, Philipp Rümmer
Uppsala University
Sweden

ABSTRACT

Dynamic software updates enable software evolution and bug fixes to embedded systems without disrupting their run-time operation. Scheduling dynamic updates for safety-critical embedded systems, such as aerial drones, must be done with great care. Otherwise, the system's control loop will be delayed leading to a partial or even complete loss of control, ultimately impacting the dependable operation. We propose an update scheduling algorithm called NeRTA, which schedules updates during the short times when the processor would have been idle. NeRTA consequently avoids the loss of control that would occur if an update delayed the execution of the control loop. The algorithm computes conservative estimations of idle times to determine if an update is possible, but is also sufficiently accurate that the estimated idle time is typically within 15% of the actual idle time.

1 INTRODUCTION

Safety-critical embedded systems differ from other embedded systems because errors or failures in their operation can endanger people and physical objects in their vicinity [4]. As a result, safety-critical embedded systems are subject to stricter requirements than embedded systems that are not safety-critical.

Aerial drones represent a challenging case of safety-critical embedded systems, because their mobility is part of the application logic. Therefore, drones are an appropriate case study for safety-critical embedded systems because an incorrect program directly affects the device physical mobility and therefore reduces its safety.

Mobile computing with drones enjoys a wide variety of possible applications. As an example, Mayer et al. [5] outlined different ways drones are used in search and rescue. With dynamic updates, a drone can be reprogrammed dynamically through an update to deliver necessary supplies such as food to a lost boat that it found, while the search and rescue personnel arrive. Without dynamic updates, the drone would have to land, then perform the update, which may not be feasible in search and rescue missions due to hazardous conditions. On-the-fly software updates provide the flexibility to make unforeseen and necessary adjustments regardless of the environmental conditions and landing opportunities.

Drone control is categorized into low-level control, and high-level control [2]. Low-level control uses sensor data and control setpoints to compute motor throttle values that keep the drone stable. Advanced drones feature high-level control that implements more advanced features such as path planning. High-level control generates control setpoints to feed into low-level control. It is not uncommon that low-level and high-level control are implemented on different boards, each of which communicates with the other through the use of a messaging protocol such as MAVLink [1].

We focus on updating the low-level control software, known as the autopilot. The autopilot is safety-critical because low-level control is necessary to keep the drone in a safe state and updating it is therefore a more difficult problem than updating the high-level control software. Autopilots are typically composed of several tasks that run hundreds of times per second. The higher the frequency of the tasks, the faster the drone can react to environmental changes. If the frequency is too low, the drone's response may be sufficiently delayed such that the drone enters an unsafe state and will crash.

2 PROBLEM AND BACKGROUND

Applying dynamic software updates during an inappropriate time can result in the delayed execution of other tasks. This is caused by the update process occupying the processor when a task would otherwise be executing. Therefore, the time an update is performed must be chosen carefully to avoid delaying others tasks, increasing the probability of entering an unsafe state.

Wahler et al. [6] explored possible update points for dynamic updates and determined that the idle time was a possible choice of time to perform an update. However, they did not explore determining how much idle time is indeed available. Zhao et al. [8] elicited two requirements for update points, *i)* timeliness, meaning that an update point is reachable within a reasonable amount of time, and *ii)* correctness, meaning that the program should behave correctly after the update. However, they did not explore the impact of the update process while the update is performed. Key for us is the impact on the physical device's behavior, which can enter an unsafe state if an update delays the execution of other tasks.

To bridge the gap in the literature, we develop NeRTA, an algorithm that schedules updates by taking into account that the program has strict timing requirements that must be met, thus minimizing the impact of the update process on the device being updated. NeRTA preserves the physical behavior of the device by scheduling updates when the processor would have been idle.

3 NeRTA: NEXT RELEASE TIME ALGORITHM

NeRTA is an update scheduling algorithm targeted towards systems composed of a fixed number of tasks that repeat indefinitely. Such a configuration is common in drone autopilot software [3]. NeRTA uses the next release time for each task as input data. The next release time is defined as the earliest time when the next instance of a task can execute and is a property of each task. NeRTA uses the next release times to compute a conservative estimate of the idle time available after a task completed its execution. Conservative estimate means that we guarantee that the idle time computed is smaller than or equal to the actual idle time, but cannot be larger.

We emphasize that the next release time is *not* the exact time a task starts executing, but the earliest time that a task *can possibly* start execution. If a task is released while another task is executing, then the released task will not execute until the executing task

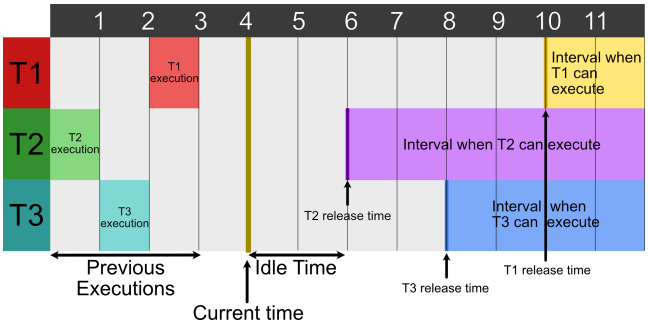

Figure 1: Example demonstrating the next release times of three tasks.

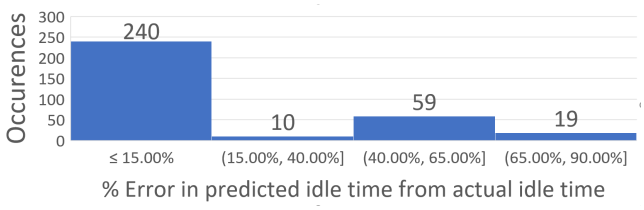

Figure 2: Histogram of errors between actual idle times and NeRTA predicted idle times. Note that a significant number of samples have an error of less than 15%.

completes its execution. Therefore, the time a task actually starts executing and the release time may differ significantly. However, the release time is always less than or equal to the time the execution starts, which makes it useful as a guarantee that a task will not execute before a specific time, allowing us to provide a conservative estimate of the idle time available.

Figure 1 shows an example of using NeRTA in a system of three tasks. Assume we are currently at time four and that Tasks 1, 2, and 3 executed at time two, zero, and one respectively and that each task has an execution time of one time unit.

In the example, the next release times provided for Tasks 1, 2, and 3 are ten, six, and eight respectively. Therefore, Tasks 1, 2, and 3 will not execute again before time ten, six, and eight respectively. We update the figure with this information by marking out when each task can execute again, which is any time greater than or equal to the release time. We extend those blocks indefinitely because we only know the earliest time that a task can start execution.

Using the next release times, we determine how much idle time we have by taking the minimum of all the three next release times. This is the earliest time that any task can execute again. We conservatively estimate how much idle time we have by computing the difference between the minimum of all future release times and the current time. Since we are currently at time four, and the next time any task can execute is time six, we have at least two time units of idle time available and can perform an update that takes at most two time units without delaying any of the other tasks.

4 EXPERIMENT

We conduct an experiment to compare the idle times predicted by NeRTA and the actual idle times observed. We run Hackflight on a flight controller called Ladybug [7]. Ladybug integrates an onboard motion sensor, which is sampled by Hackflight at regular intervals. We do not connect Ladybug to any motors, but still run all the tasks that would be present in a drone. We measure the predicted and actual idle times for 0.66 seconds, starting at ten seconds after boot and capturing 587 samples. The data is captured into a buffer, which is sent through a serial connection after data collection is complete.

We exclude from the results cases when the predicted idle time is negative or zero. These cases occur in 260 of the 587 samples and happen if one of the next release times is less than the current time and therefore we cannot guarantee any idle time. At run-time, these situations are readily detected and only affect how many iterations of NeRTA we need before an update is scheduled successfully, not the maximum size of the update that can be scheduled.

Figure 2 demonstrates the results of the experiment. No bin had an error larger than 90%. In most samples, the error is less than 15%. Of the 240 samples that have less than 15% error, 143 of those have less than 5% error. Therefore, NeRTA estimates idle times that are not significantly different from the actual idle times making it a reasonable estimate of the idle time available that is both conservative and also sufficiently close to the actual idle time.

We measure the performance impact of running NeRTA with the same hardware configuration. Compared with the idle times observed that are in the order of 1 ms, the time required to check schedulability with NeRTA is roughly 0.08 ms, which is negligible.

5 OUTLOOK

For a complete dynamic update solution, other problems must be considered as well. First, solving the problem of state transfer, i.e., how to transform the state of the program to be compatible with and support the new version of the code, which is particularly difficult because the state changes during runtime.

Second, solving the problem of how to generate a program binary to optimize for dynamic update speed. Binaries are typically a contiguous block of machine code, which may not be an ideal structure for inserting code during dynamic updates because all the code located after the newly inserted code would be shifted, requiring changes to instruction and memory references.

REFERENCES

[1] 2022. MAVLink Developer Guide. https://mavlink.io/en/
[2] Endri Bregu, Nicola Casamassima, Daniel Cantoni, Luca Mottola, and Kamin Whitehouse. 2016. Reactive Control of Autonomous Drones *(MobiSys '16)*. ACM. https://doi.org/10.1145/2906388.2906410
[3] Zhuoqun Cheng, Richard West, and Craig Einstein. 2018. End-to-End Analysis and Design of a Drone Flight Controller. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* (Nov. 2018).
[4] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze. 2017. Metamorphic Model-Based Testing of Autonomous Systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*.
[5] Sven Mayer, Lars Lischke, and Pawel W. Woźniak. 2019. Drones for Search and Rescue. In *1st International Workshop on Human-Drone Interaction*. ENAC, Glasgow, United Kingdom. https://hal.archives-ouvertes.fr/hal-02128385
[6] Michael Wahler, Stefan Richter, and Manuel Oriol. 2009. Dynamic software updates for real-time systems *(HotSWUp '09)*. ACM, New York, NY, USA. https://doi.org/10.1145/1656437.1656440
[7] Kris Winer. 2021. Ladybug Flight Controller by Tlera Corp on Tindie. https://www.tindie.com/products/TleraCorp/ladybug-flight-controller/
[8] Zelin Zhao, Xiaoxing Ma, Chang Xu, and Wenhua Yang. 2014. Automated recommendation of dynamic software update points: an exploratory study *(INTERNETWARE 2014)*. ACM, New York, NY, USA. https://doi.org/10.1145/2677832.2677853