Reactive Control of Autonomous Drones

Endri Bregu[†], Nicola Casamassima[†], Daniel Cantoni[†],
Luca Mottola[†*], and Kamin Whitehouse[‡]
[†]Politecnico di Milano (Italy), [*]SICS Swedish ICT, [‡]University of Virginia (USA)
Contact author: luca.mottola@polimi.it

ABSTRACT

Aerial drones, ground robots, and aquatic rovers enable mobile applications that no other technology can realize with comparable flexibility and costs. In existing platforms, the low-level control enabling a drone's autonomous movement is currently realized in a time-triggered fashion, which simplifies implementations. In contrast, we conceive a notion of *reactive control* that supersedes the time-triggered approach by leveraging the characteristics of existing control logic and of the hardware it runs on. Using reactive control, control decisions are taken only upon recognizing the need to, based on observed changes in the navigation sensors. As a result, the rate of execution *dynamically adapts* to the circumstances. Compared to time-triggered control, this allows us to: *i)* attain more timely control decisions, *ii)* improve hardware utilization, *iii)* lessen the need to over-provision control rates. Based on 260+ hours of real-world experiments using three aerial drones, three different control logic, and three hardware platforms, we demonstrate, for example, up to 41% improvements in control accuracy and up to 22% improvements in flight time.

1. INTRODUCTION

Aerial drones, ground robots, and aquatic rovers enable novel mobile applications. Compared to mobile phones and connected cars that only opportunistically sense or communicate, these platforms offer *direct control* over their movements. They can thus implement functionality that were previously beyond reach, such as collecting high-resolution imagery of civil infrastructures [10, 38], exploring near-inaccessible areas [16], or inspecting the sea floor to gain fine-grained environmental data [19, 54].

Existing platforms. Most existing drone platforms are architected as in Fig. 1. Two components are involved. Specialized software runs at a ground-control station (GCS) to let users configure mission parameters, such as the coordinates to cover through waypoint navigation and the action to take at each waypoint. The GCS is typically a standard

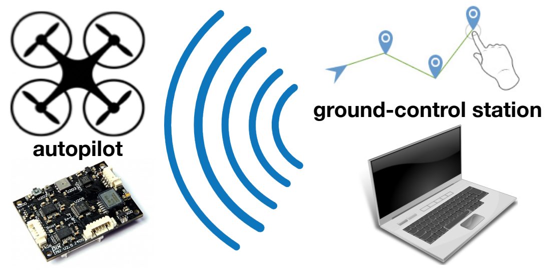


Figure 1: Software components in mainstream drone platforms. *The ground control station let users configure high-level mission parameters, the autopilot software implements the low-level motion control aboard the drone.*

computer that communicates with the drone using a long-range low-bandwidth radio.

Aboard the drone, the *autopilot* software implements the low-level control in charge of *autonomously* driving the drone to the next desired coordinate. The control loop processes various sensor inputs, such as accelerations and GPS coordinates, to operate actuators such as electrical motors that set the 3D orientation of the drone, also commonly termed as the drone's *attitude*. Because of size, cost, and energy concerns, autopilots run on resource-constrained embedded hardware. For example, ARM MCUs of the Cortex M series are often used as the main processing unit.

Together with the mechanical design, the low-level control determines the effectiveness of physical motion. For example, when using aerial drones in imagery applications, the low-level control directly influences the quality of the shots [37, 38]. Further, the low-level control is partly responsible for how the energy available from batteries is consumed. The drone's lifetime is often a result of how streamlined is its operation [11, 55].

Most existing autopilot implementations employ Proportional-Integral-Derivative (PID) [7] designs for low-level control. These controllers run in a time-triggered fashion: every T time units, sensors are probed, control decisions are computed, and commands are sent to the actuators. Such an approach enjoys the advantage of highly deterministic operation, which simplifies implementations.

Reactive control. Despite these advantages, current implementations overlook two essential aspects, as we further illustrate in Sec. 2:

1) Most PID controllers on the drones are tuned so that it is mostly the Proportional component to bear an influ-

MobiSys'16, June 25 - 30, 2016, Singapore, Singapore

ISBN 978-1-4503-4269-8/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2906388.2906410

ence. As long as the weights are balanced, the Derivative component can be kept to a minimum [13, 26]. A proper calibration of navigation sensors may also reduce the Integral component [13, 26, 44]. This yields executions where: *i)* small variations in the sensor inputs tend to correspond to small variations in the actuator settings, and *ii)* as long as the sensor inputs do not change, the actuator settings remain almost unaltered. Therefore, *in principle*, one may spare control executions that starts from the same or similar sensor inputs as the previous iteration, by simply maintaining the earlier actuator settings. We quantitatively verify this intuition in Sec. 3 through real-world experiments.

2) Autopilot software typically runs on hardware that closely resembles mobile phones. This especially applies to the sensing equipment. Many argue that without the push to improve sensors due to the rise of mobile phones, drone technology would have not emerged [17]. These sensors are especially designed to enable energy-efficient high-frequency sensing; for example, for tracking human activity [20, 35]. Many of them can also be programmed to return a value only upon verifying certain conditions [27, 49]; for example, when a threshold is passed, which is useful to implement functionality such as fall detection [22].

Reactive control builds upon these observations. Rather than periodically triggering the control logic, we constantly monitor the navigation sensors and run the control logic only upon recognizing *the need to*. As a result, reactive control *dynamically adapts* the control rate. When sensor inputs change often, control runs repeatedly, possibly even more frequently than the fixed rate of a time-triggered implementation. When sensor inputs exhibit small or no variations, the rate of control execution reduces, freeing up resources that may be needed at different times. Unlike other emerging forms of asynchronous control [6], however, we require no changes to the underlying control logic. The design methodologies of traditional control still apply [7].

Reactive control yields several advantages, for example: *i)* it enables more timely and adaptive control decisions, *ii)* it spares unnecessary processing, improving the utilization of the hardware, and *iii)* it lessens the need to overprovision control rates to handle extreme situations. As it exclusively works in software, reactive control also requires no hardware modifications. Nevertheless, we demonstrate that reactive control is applicable beyond waypoint navigation. Further use cases include, for example, active sensing functionality [14, 36, 51] and advanced motion control [1].

Challenges and solutions. Realizing reactive control is, however, non-trivial. Three issues are to be solved:

1) What is a "significant" change in the sensor input depends on several factors, including the accuracy of sensor hardware, the physical characteristics of the drone, the control logic, and the granularity of actuator output. This notion is thus difficult to generalize. Sec. 4 illustrates our probabilistic approach to tackle this problem, which *abstracts* from all these aspects by employing a form of auto-tuning of the conditions leading to running the control logic.
2) An indication for running the control logic may originate from different sensors, at different rates, and asynchronously with respect to each other. One problem is thus how to handle the possible interleavings. Moreover, not running the control loop for too long may negatively affect the drone's stability, possibly preventing to reclaim the correct behavior. Sec. 5 illustrates how we tackle these issues by only changing the *execution* of the control logic over time, rather than the logic itself.
3) Reactive control must run on resource-constrained embedded hardware. When implementing reactive control, however, the code quickly turns into a "callback hell" [25] as the operation becomes inherently event-driven. We experimentally find that, using standard languages and compilers, this negatively affects the execution speed, thus limiting the gains. Sec. 6 describes our custom realization of reactive programming (RP) techniques [8] to tackle this problem.

Sec. 7 illustrates the effectiveness of reactive control compared to the time-triggered approach. We consider the challenging case of aerial drone control. These require continuous attitude adjustments because of the potentially severe environment influence, for example, due to wind gusts in unpredictable directions. We report on 260+ hours of test flights in three increasingly demanding environments, using a combination of three aerial drones, three autopilot software, and three embedded hardware platforms.

Our results indicate that reactive control obtains up to 41% improvements in the accuracy of motion, which results in up to 22% extension of flight times. We also demonstrate that our RP-based implementation is fundamental to obtain this performance, in that a traditional implementation would only obtain about half of the improvements. Finally, we report on the benefits reactive control provides in two end-user applications beyond waypoint navigation, including 3D reconstruction and active sensing in search-and-rescue scenarios.

2. BACKGROUND

First, we survey efforts in related areas, showing that the goal we pursue sets us apart from existing literature. Next, to better understand our context, we describe the Ardupilot [5] autopilot and supported hardware.

2.1 Related Work

The drones we target in this work can be regarded as a cruder form of modern robotics [17]. In this field, control operates at two levels. *High-level* control implements advanced navigation functionality, such as simultaneous localization and mapping [23], path planning [9], and obstacle avoidance [30]. In our context, this is the job of the GCS, even though it is certainly realized in a simplified form here.

The output of high-level control may be a waypoint or a trajectory, given as input to the *low-level* control that operates the robot actuators. In the majority of the cases, PID controllers are used for this [13]. The rate to run the controller is statically set to find a trade-off between accuracy and resource consumption, based on a few "rules of thumbs" [56]. In contrast, our goal is to run the PID controller in a reactive fashion, dynamically adapting to the circumstances dictated by, for example, the instantaneous environment influence.

In the field of aerial drones, demonstrations exist showing motion control in tasks such as throwing and catching balls [43] and poles [15], building architectural structures [32], flying in formation [50], and carrying large payloads [34]. In

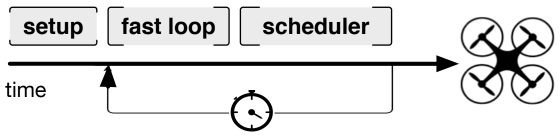


Figure 2: Ardupilot's control loop. *The time for a single iteration of the loop is split between* fast loop*, which only includes critical motion control functionality, and an application-level* scheduler *that runs non-critical tasks.*

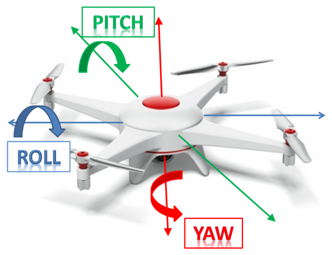


Figure 3: Attitude control with raw, pitch, and yaw.

these settings, the low-level control does not operate aboard the drone. At 100 Hz or more, a powerful computer receives accurate localization data [52], runs sophisticated control algorithms based on drone-specific mechanical models often expressed through differential equations, and sends actuator commands back to the drones. Differently, we aim at improving the performance of mainstream low-level control running on embedded hardware, targeting mobile sensing applications that operate in the wild.

On the surface, reactive control may resemble the notion of event-based control [6]. In the latter, however, the control logic is expressly redesigned for settings different than ours; for example, in distributed control systems to better cope with limited communication bandwidth. This requires an entirely different theoretical framework [6]. Our work aims at re-using existing control logic, whose properties are well understood. Different than event-based control, however, reactive control is mainly applicable only to PID-like controllers where the Proportional component dominates.

2.2 Autopilots

Autopilots provide critical functionality for the efficient and dependable operation of drone platforms. Their implementation is also often coupled with dedicated hardware.

Among available autopilot implementation, Ardupilot[1] is a mature open-source project that provides reliable autopilot functionality for aerial drones and ground robots [5]. The project is at the basis of many commercial products, including those of 3DRobotics [2] and many others, and boasts a large on-line community. Nonetheless, our evaluation in Sec. 7 demonstrates the applicability of reactive control with autopilots other than Ardupilot and dedicated hardware.

Software. Fig. 2 shows the execution of Ardupilot's control loop. Following the initial setup, the control loop is split in two parts. The *fast loop* only includes critical motion control functionality. The time left from the execution of *fast loop* is given to an application-level *scheduler* that distributes it among non-critical tasks that may not execute at every iteration, such as logging. The scheduler operates in a *best-effort* preemptable manner based on programmer-provided priorities. Many autopilots share similar designs [18, 39].

Initially, *fast loop* blocks waiting for a new value from the Inertial Measurement Unit (IMU), which provides an indication of the forces the drone is subject to in the three dimensions. This is obtained by combining the readings of accelerometers, gyroscopes, magnetometers, and barometers. Once a new value is available, IMU information is combined with GPS readings to determine updated attitude control by minimizing the error between the desired and actual pitch, roll, and yaw, shown in Fig. 3. Multiple PID controllers inside *fast loop* are used to this end. Their output is converted into commands sent to the motors to orient the drone.

Ardupilot well exemplifies the state of the art. Control is periodic and proceeds sequentially, which simplifies the implementation. Most importantly, the control rate is statically set as discussed in Sec. 2.1. For example, Ardupilot runs at a *fixed* 400 Hz on the hardware we describe next. This rate is not necessarily the maximum the hardware supports. The 400 Hz of Ardupilot, for example, are thought to leave enough room—on average—to the scheduler. In short bursts, control may run much faster than 400 Hz, as long as some resources are eventually allocated to the scheduler; for example, at times when control does not need to run that frequently. By recognizing the situations when control does need to run—or not—reactive control enables precisely this kind of dynamic adaptation of control rate.

Hardware. Ardupilot runs on various embedded hardware. A primary example is the Pixhawk board [41], which features a Cortex M4 core at 168 MHz and a full sensor array for navigation, including a 16-bit gyroscope, a 14-bit accelerometer/magnetometer, a 16-bit 3-axis accelerometer/gyroscope, and a 24-bit barometer. Most often, at least a sonar and a GPS are added to the built-in sensor array to provide positioning and altitude information, respectively.

The sensors on Pixhawk have similar capabilities as those on modern mobile phones. They support energy-efficient high-frequency sampling and often provide interrupt-driven modes to generate a value upon verifying certain conditions. The ST LSM303D [49] on the Pixhawk, for example, can be programmed to generate an SPI interrupt based on three thresholds. While useful, for example, in crowdsensing applications for specific functionality such as fall detection [35], these features are rarely exploited in autopilots.

The commands sent from the autopilot to the actuators are encoded in Pulse-Width Modulation (PWM). These are converted to a current flowing into the motors through dedicated Electronic Speed Controllers (ESC): tiny embedded boards that govern the current flow to the motors according to some analog input. Typically, ESCs hold the output value as long as the input does not change. Because of size and cost, ESCs are extremely primitive; most of them employing 8-bit Atmel ATMega MCUs. The granularity of their inputs/outputs is also often quite coarse. This further limits the degrees of changes to the actual motor operation.

3. MOTIVATION

Experimentally verifying that some iterations of the control loop are, in fact, unnecessary is a challenge per se. One issue is that of *visibility*: the most un-biased measure we may get is the current flowing into the motors, namely, the output of the ESC. Next is a problem of *realism*: one would

[1]At the start of the project, Ardupilot ran on Arduino hardware. However, developers eventually moved to more capable hardware while retaining the name.

(a) Quadcopter. (b) Hexacopter.

Figure 4: Custom aerial drones used to gain evidence of the opportunities for reactive control.

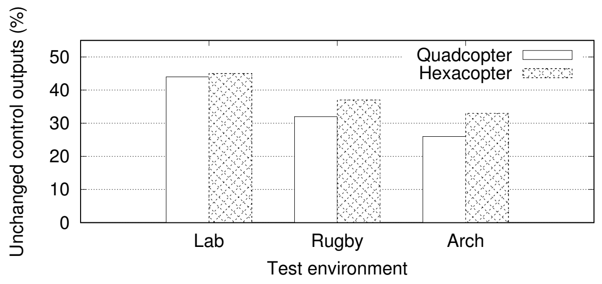


Figure 5: Preliminary experiments: fraction of time the control outputs do not change. *The less influence the environment plays, the more the control decisions tend to remain the same. The hexacopter shows higher values as it is more resilient to the environment.*

need to gather the measurements during actual operation. Finally, high-frequency logging is *expensive* on embedded hardware, and may affect the processing of the control loop.

Setup. To address the issues above, we instrument two custom-built aerial drones, shown in Fig. 4, with Hall-effect current sensors [47] placed between the ESC and the motors. We choose aerial drones because they represent a challenging case for autopilots, as they require continuous attitude adjustment. By contrast, for example, a ground robot may simply not run any control loop once it reaches the destination, as in most cases it naturally maintains the position. The Hall-effect sensors are sampled at the same rate of the control loop from an independently-powered Waspmote [31] aboard the drones. Data is dumped on an SD card. The measurement subsystem is thus completely decoupled from the drone, and does not affect its operation but for the minimal added weight.

We test several flight paths at a maximum speed of 3 m/s and 5 m altitude, each with 8 random waypoints in a loop and lasting at least 20 min, in three environments: *i)* a 20x20 m lab where drones localize themselves using on-drone visual techniques [42], termed Lab; *ii)* a rugby field in front of our department termed Rugby, using GPS; and *iii)* an archaeological site in Aquileia (Italy) termed Arch, the size of almost four soccer fields [36], stll using GPS. The sites exhibit increasing environment influence. Lab only suffers from air conditioning from the ceiling. Rugby is protected on two sides by trees. Arch lies in an area with average wind speeds of 8+ knots. We record a total of 10 flight hours for each drone in every site.

Results. Fig. 5 shows the fraction of samples from the Hall-effect sensor within the sensor accuracy of the previous sample. This quantity can be understood as the fraction of time the control decisions at one iteration are the same as the previous one. This figure is highest for Lab, because of the little environment influence. By contrast, the values are smallest for Arch: sudden wind gusts may require unanticipated attitude corrections. The physical design also plays a role: the hexacopter, which is heavier and mounts more powerful motors, is more resilient. Thus, the control logic outputs the same decisions more often than the quadcopter.

Fig. 5 shows that there would be ample margin to spare a fraction of control executions, even in the most challenging environment, *if only one were able to recognize these unnecessary iterations beforehand.* By freeing resources when they are not needed, we may make them available when there is a need to promptly *react*, thus overcoming the limitations of a fixed control rate. Sparing unnecessary executions of the control logic may also create room for running additional functionality on the same hardware, or yield an opportunity for downsizing the hardware itself.

The rest of the paper describes how reactive control provides these benefits by changing the temporal execution of *existing* control logic. Three issues are to be addressed. Sec. 4 describes how to recognize, based only on sensor inputs and with minimal overhead, when there is a need for running the control logic. Next, Sec. 5 describes how to regulate the executions in a setting where different sensors may require running the control logic asynchronously with respect to each other. Sec. 6 describes how to implement these functionality efficiently on resource-constrained hardware.

4. CONDITIONS FOR REACTING

In principle, one may simply assume that control decisions do not change whenever the inputs do not change.

Problem. Although correct, such a reasoning is, in fact, quite naive. The control logic used to govern the drones is not at all trivial. It involves combining several readings from diverse sensors with varying precision and generated at different rates across multiple steps of processing. Moreover, the control logic runs on embedded resource-constrained hardware and using simplified numerical libraries, which affects the precision of processing. Thus, it is often the case that control decisions remain the same even when the sensor inputs *do change.*

The issue is thus to distinguish those changes in the sensor inputs that would alter the control decisions from those that would not. Whenever a sensor input changes in a way that—all other sensor inputs staying the same—the control decisions change as well, we say we record a *trigger* on that sensor. It is difficult to determine beforehand the conditions to search in the sensor readings to identify a trigger. Factors such as the accuracy and calibration of sensors, the physical characteristics of the drone, the control logic, and the granularity of actuator outputs all concur to determine these conditions.

Approach. We opt for a lightweight on-line approach that *self-adapts* to different instances of the aspects above *without* assuming any specific knowledge on any of them, including the control logic. This makes our approach applicable to different implementations, as we demonstrate in Sec. 7.

Despite the control logic is deterministic, we abstract away the underlying complexity by considering *a change in the control output* as a random phenomena. The input to this phenomena is the difference between consecutive samples of the same sensor; the output is a binary value indicating whether the control decisions change. An accurate statisti-

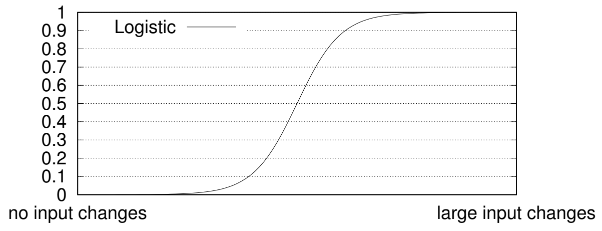


Figure 6: Example logistic function.

cal estimator of such phenomena would allow us to take an informed decision on whether to run the control logic.

Among statistical estimators with a *binary* dependent variable, *logistic regression* [28] is both simple and closely matches the intuition on when one would need to execute the control logic. Fig. 6 shows its probability distribution. For small changes in the sensor inputs, the probability of changes in the control outputs is small. When changes in sensor inputs are large, a change in the control decisions becomes (almost) certain. The logistic function we employ is expressed as:

$$L(x) = \frac{1}{1 + e^{-(\beta_1 + \beta_2 x)}} \qquad (1)$$

where $L(x)$ is the probability of new control decisions compared to the previous iteration, given a change of x in the inputs of a navigation sensor. The problem thus becomes how to estimate parameters β_1 and β_2 at run-time. Existing results indicate that, for $0 < L(x) < 1$, eq. (1) can be re-written in *generalized linear form* [28]. Barring the cases $L(x) = 0$ and $L(x) = 1$, this allows us to employ simple estimation models to determine β_1 and β_2, such as ordinary least squares [28].

Detecting triggers. We employ one logistic regression model per navigation sensor. Given a change x in the sensor readings from one iteration to the next, we compute the probability $L(x)$ that the change corresponds to new control decisions. If this is greater than a threshold P_{run}, we detect a trigger on that sensor and execute the control logic, with all other inputs set to the most recent value; otherwise, we maintain the earlier output to the actuators.

This approach assumes that changes in a sensor's inputs at different times are statistically independent. This is justified because the time-dependent I, D components of the PID controllers bear little influence in our setting, as discussed earlier. Moreover, maintaining the earlier output to the actuators is possible only as long as the control setpoint does not change in the mean time. This is most often the case when drones hover or perform waypoint navigation, but rarely happens in applications such as aerial acrobatics, where this approach would probably be inefficient.

Parameter P_{run} offers a knob to trade computational resources against the tightness of control. Large values of P_{run} spare a significant fraction of control executions. However, the drone may require drastic corrections whenever the control loop does run; in a sense, motion becomes more "nervous". Small values of P_{run} limit the processing gains. However, control runs more often, ensuring the drone smoothly maintains the right attitude. Sec. 7 illustrates that gains over time-triggered control are seen for many different settings of P_{run}. Also, P_{run} is no threat to dependable operation, in that we make sure there is a minimum frequency the control loop runs anyways, as described in Sec. 5.

Run-time operation. The least square estimation has two operating modes. At start-up, control runs in a time-triggered fashion for T_{boot}. For each execution of the control logic within T_{boot}, we record whether that run was necessary. This provides an initial data set for the least square estimation to compute preliminary values for β_1 and β_2. The value of T_{boot} can be small; for the experiments of Sec. 7, we use $T_{boot} = 30$ sec.

After T_{boot}, we start identifying the triggers by comparing $L(x)$ with P_{run}. Upon occurrence of a false positive, that is, a value of x recognized as a trigger that does *not* change the control decisions, we feed back the value of x to the estimation of β_1 and β_2. This occurs by adding x to the data set for least square estimation and by periodically re-evaluating β_1 and β_2. The parameter estimation runs with medium priority in the scheduler part of the control loop, shown in Fig. 2.

This design considers the execution up to T_{boot} as representative of the rest of the flight. Should this not be the case, for example, whenever after T_{boot} a drone would enter an area with significantly different environment conditions, the initial parameter estimation would need to be recomputed. Moreover, Sec. 5 shows there may also be cases when we recognize false negatives. From the perspective of parameter estimation, we treat these occurrences in the same way as false positives. Overall, this processing implements a simplified form of *auto-tuning* [56] that improves the accuracy of logistic regression as the system runs.

5. TRIGGERS OVER TIME

Existing control logic, such as the various PID controllers used in autopilots, is designed under the assumption that sensors are sampled simultaneously according to a known frequency [7]. Removing such assumptions changes the control problem and requires a different conceptual framework [6]. In contrast, we aim at re-using existing control logic, and thus intend to approximate its timing assumptions.

Problems. Because of the above, how to process triggers deserves some thoughts:

1) We should be careful when multiple sensors record triggers close in time. It is unclear if these triggers deserve separate executions of the control logic as they are "too far" in time, or rather a single iteration of the control logic suffices because the triggers are "sufficiently close" to approximate simultaneity.
2) The way we recognize triggers is essentially probabilistic, as described in Sec. 4. Therefore, to ensure dependable operation we must consider the unlucky case when we miss a large number of consecutive triggers, and avoid running the control logic for too long. Otherwise, when we finally recognize a trigger, it may be too late to reclaim the drone's stability.

Several aspects are to be taken into account to find appropriate solutions to these problems.

Sampling frequency. Ideally, one would aim at continuously sampling the environment waiting for triggers. This way, they would be promptly recognized. In reality, such a sampling only occurs at discrete times.

To recognize triggers as early as possible, we sample every sensor at the highest frequency allowed by the hardware. Using sensors like those of Pixhawk boards, this is usually not a problem. They are expressly designed for energy-efficient high-frequency sampling, while the major energy drain aboard the drones is anyways due to the motors. Moreover, whenever the sensors permit, as in the case of the ST LSM303D [49], we compute the inverse of eq. (1) and program the sensor to return an interrupt only when a change x would return $L(x) > P_{run}$.

Hyperperiod triggers. To match the assumptions of existing control logic, we must approximate the simultaneous sampling of all input sensors. However, the time of sampling, and therefore of possibly recognizing a trigger, is not necessarily aligned across sensors. Drastic changes in the sensor inputs may also be correlated. For example, when the accelerometers record a sudden increase, it may be because environmental factors such as wind gusts; in these cases, a gyroscope also likely records changes. A time-triggered implementation would likely process these inputs together.

We take a conservative approach to address these issues. Based on the sampling frequency of every sensor in the system, we compute the system's *hyperperiod* as the smallest interval of time after which the sampling of all sensors repeats. Upon recognizing a trigger, we wait until the current hyperperiod completes before running the control logic. This allows us to "accumulate" all triggers possibly recognized on different sensors, giving the most up-to-date inputs to the execution of the control logic at once.

Note that all such executions are not taken into account as feedback to the regression models described in Sec. 4. With multiple triggers from different sensors, it is difficult to determine which sensors are responsible for the change of control decisions. This would require an understanding of the underlying control laws, which we wish avoid to foster general applicability of reactive control.

Failsafe. There may be unlucky cases when false negatives in the trigger conditions happen in a row. This may hurt dependability: it may be too late to reclaim the drone's stability when a trigger is recognized.

To cater for these extreme cases, we run the control logic anyhow every $T_{failsafe}$. Whenever this happens after a row of false negatives, the drone most likely applies some significant corrections that make sensor inputs change drastically. In turn, this triggers reactive control already at the immediately following iteration. In other words, $T_{failsafe}$ prompts reactive control to execute as soon as possible whenever there is a need to. In the experiments of Sec. 7, including those in ARCH with winds blowing at 8+ knots, we use $T_{failsafe} = .1$ sec. This is negligible compared to a control logic that normally runs at 400 Hz.

For all executions due to $T_{failsafe}$ resulting in a change of actuator settings, if the logistic regression models of Sec. 4 return a false negative for every sensors, the last changes to the sensor inputs are fed back for the next round of least square estimation of β_1 and β_2. We do this to reduce false negatives. We also demonstrate that the setting of $T_{failsafe}$ bears little influence on the final performance. $T_{failsafe}$ should be set to ensure the drone's stability in worst-case conditions. Determining this value analytically would require an accurate model of the environment to identify these worst-case conditions. In this work, we take a practical approach—as in the vast majority of operational control systems [4]—and employ a setting of $T_{failsafe}$ that is most likely overprovisioned, but safely ensures the dependable system operation without appreciably impacting the performance.

6. IMPLEMENTATION

For simplicity, Sec. 4 and Sec. 5 describe the control logic as a single processing unit. In reality, things are more complex. As a matter of fact, the control logic is composed of multiple processing steps arranged in a complex multi-branch pipeline. Depending on what trigger occurs, different slices of the control logic may need to run while other parts may not. Moreover, each such processing step may—in addition to producing an output *immediately* useful—update global state used *at a different iteration* elsewhere in the control pipeline.

In this setting, using triggers makes the processing event-driven not just because of the triggers themselves, but also because of asynchronous updates to global state. Thus, *every* processing step—not just those that directly process the sensors' inputs—might potentially need to execute upon recognizing a change in the inputs. Employing standard programming techniques to this end quickly turns implementations into a "callback hell" [25]. This fragments the program's control flow across numerous syntactically-independent fragments of code, hampering compile-time optimizations. As shown in Sec. 7, this causes a processing overhead that limits the benefits of reactive control.

We tackle this issue using a technique called *reactive programming* (RP) [8], illustrated in Sec. 6.1. This technique is rarely employed in embedded computing, because of resource constraints. We thus create an RP implementation tailored to the hardware we target, described in Sec. 6.2, with additional custom semantics to better support reactive control. Next, Sec. 6.3 describes the use of our realization of RP to implement reactive control.

6.1 Reactive Programming

RP is increasingly employed to realize applications that maintain a continuous interaction with the environment [8]. In these settings, it is generally impossible to predict the time of arrival of interesting events. Programmers thus implement the required processing through asynchronous callbacks and by manually propagating the relevant updates throughout the application's global state. Besides being error-prone, this is usually also inefficient. The control flow is hidden in the interactions across callbacks, which typically happens through side-effects and updates to global state variables. Thus, compilers and linkers have little control-flow information to optimize the execution.

RP provides abstractions to automatically manage data dependencies in programs where updates to variables happen unpredictably. Consider the example:

```
a= 2;
b= 3;
c= a + b;
```

In sequential programming, variable `c` retains the value 5 regardless of any future update to variable `a` or `b`. Updating `c` requires an explicit assignment following the changes in `a` or `b`. It becomes an issue to determine *where* to place such an assignment without knowing when `a` or `b` might change.

Using RP, one declaratively describes the data dependencies between variables `a`, `b`, and `c`. As variables `a` and `b` change, the value of `c` is constantly kept up-to-date. Then, variable `c` may be input to the computation of further state variables. Conceptually, the data dependencies take the form of an (acylic) graph, where the nodes represent individual values, and edges represent input/output relations.

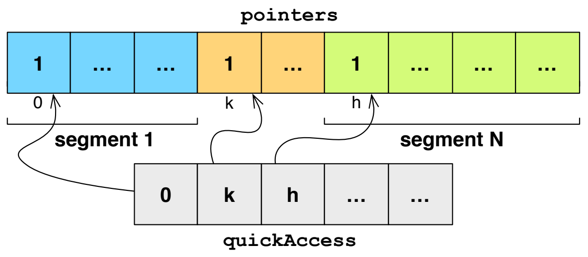


Figure 7: Data structures used in RP-Embedded to store the data dependency graph. *Using statically dimensioned vectors helps reduce memory consumption at the price of reduced flexibility.*

Together with the value, every node is also associated to a processing function that determines the value based on the inputs, executed whenever any of the input nodes changes. This resembles synchronous programming [12], yet with the fundamental difference that the arrival of events remains generally unknown.

The RP run-time support traverses the data dependency graph every time a data change occurs, stopping whenever a variable does not change its value as a result of changes in its inputs. Any further processing would be unnecessary because the other values in the graph would remain the same.

6.2 RP-Embedded

Run-time support for RP is provided through specialized libraries embedded in mainstream languages. Autopilots are generally written in C/C++, so we focus primarily on these.

Motivation. Existing C++ RP libraries [21, 46, 48] implement the propagation of changes in the data dependency graph using complex functionality to speed up the operation. One example is the use of Intel's Threading Building Blocks (TBB) [29]: a C++ library to handle concurrent events in a multi-core system. These functionality are not applicable to resource-constrained MCUs. Existing RP libraries also pay little attention to memory consumption, employing dynamic and even redundant data structures. These provide flexibility at run-time and reduce execution times with a large number of inputs.

Our setting is different. First, the data dependency graph encodes the control logic; therefore, its layout is known at compile-time. Further, memory consumption must be limited, as we use resource-constrained hardware. The sensors we wish to map to behaviors are also only a handful. Finally, the highest frequency of data changes is known; for each input sensors, we are aware or can safely approximate the highest sampling frequency. Based on these considerations, we design and implement RP-Embedded: a C++ library to support RP on embedded resource-constrained hardware.

Data structures. Unlike existing libraries, we employ two statically allocated vectors to encode the graph, as shown in Fig. 8. The `pointers` vector is split in as many segments as the maximum number N of nodes in the dependency graph. Each segment stores a sequence of object pointers that indicate what other nodes in the dependency graph use the value of that node as input. The `quickAccess` vector serves to speed up random accesses to the `pointers` vector. Using these structures, memory occupation is greatly reduced, especially compared to the container classes of the STD library used in many C++ RP libraries. For the same reason, the traversal of the graph requires fewer pointer dereferences and indirection operations, which tend to be costly on embedded hardware.

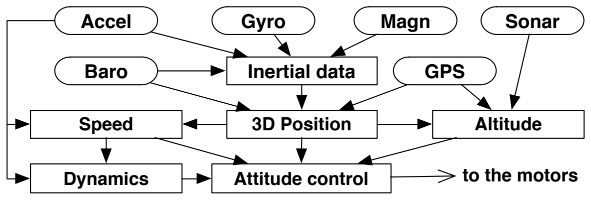


Figure 8: Ardupilot's control loop for copters after refactoring to use RP-Embedded. *Squashed rectangles indicate triggers possibly signalled by sensors, squared rectangles indicate global state information.*

These features come at the cost of limited flexibility. Programmers must place an upper bound on the number of nodes N and on their fan-out. At run-time, the data dependency graph can only change within these bounds, yet these needs should be rare in autopilots.

Hyperperiods. From the discussion in Sec. 5, we know it makes sense to await the possible signaling of all triggers within the same hyperperiod.

Besides having an effect on control decisions, this may impact the processing overhead. Say we implement a classical RP semantics and propagate any data change independent of any other. The propagation caused by triggers from a high-frequency sensor may simply be immediately superseded by another trigger caused by another sensor within the same hyperperiod. The control decisions that become effective, however, are only those of the second propagation.

To avoid this unnecessary processing, RP-Embedded allows the initial input values to be classified based on their maximum rate of change. Based on this information, RP-Embedded computes the system's hyperperiod and, for all downstream nodes in the dependency graph, it recursively determines their maximum rate of update. Every time a value is updated in the graph, RP-Embedded knows the maximum time to wait before propagating the change, as further updates in other inputs may occur. As far as we know, such a semantics is not available in any RP implementation, regardless of the language.

6.3 Using RP-Embedded

Employing RP-Embedded for reactive control requires to reformulate the *implementation* of the control logic—not the logic itself—in the form of a data dependency graph. The sensor inputs, as well as the outputs to the motors, remain the same as in the original time-triggered implementation. Triggers generated by sensors when applying reactive control become the initial inputs to the dependency graph, whereas global state variables are explicitly associated to the functions that compute their values. The implementation of trigger functionality is straightforward, as it boils down to computing $L(x)$ and comparing the output against P_{run}. The run-time estimation of β_1 and β_2 simply consists in running an ordinary least square estimation [28].

The problem above is, in essence, a problem of code refactoring. We currently perform this transformation manually, using code inspection tools such as Understand [45]. Software engineering, however, offers a wide literature on code refactoring that can be leveraged to further ease this transformation [33]. Even in the absence of dedicated support, our experience indicates that the needed transformations can be implemented with very little effort. Fig. 8 shows the data dependency graph of the Ardupilot control loop for copters, which a single person on our team realized and tested in *three* days of work. Ardupilot is one of the most complex autopilot implementations. The other autopilots we test in Sec. 7 are simpler. It took from one to two work days to refactor them. We plan to perform a complete study to better quantify the efforts and benefits due to implementing control logic in a reactive fashion as opposed to a time-triggered approach.

7. EVALUATION

We assess the effectiveness of reactive control along several dimensions, using aerial drones as a challenging use case. We compare the performance of implementations using reactive control against the original time-triggered versions.

In the following, Sec. 7.2 reports on the improvements in navigation and flight time. Sec. 7.3 discusses a series of micro-benchmarks investigating the influence of RP and parameter settings. Sec. 7.4 illustrates the impact of reactive control in two end-user applications, demonstrating the benefits beyond waypoint navigation. The next section illustrates the experimental setup common to all experiments.

7.1 Setup

Drones. We use the two drones of Fig. 4 plus a Y6 drone from 3D Robotics [2]. The latter is peculiar; it is composed of three arms with two co-axial motor-propeller assemblies at each end. One propeller faces up and "pulls"; the other propeller faces down and "pushes". This configuration obtains almost the same thrust of a hexacopter, but with fewer components and a lighter frame, which allows it to carry larger payloads.

Using a Y6 configuration, however, control is more complex. The autopilot needs to set the desired attitude by carefully tuning the motor settings on the same arm while counteracting the gyro effect. Due to this, not all autopilots support this configuration.

Autopilots. Besides Ardupilot and Pixhawk, we consider two other autopilots and corresponding hardware. We implement reactive control on both as described in Sec. 6, even though the results of the refactoring generally differ from that of Ardupilot, shown earlier in Fig. 8.

OpenPilot [39] is an open-source community-driven autopilot tightly co-designed with the underlying hardware. It generally offers fewer navigation features than Ardupilot. The corresponding CC3D board mounts a Cortex M3 MCU clocked at 72 MHz, coupled to a 16-bit 3-axis accelerometer/gyroscope, a 24-bit barometer, and a 12-bit magnetometer. *Cleanflight* [18] is the youngest of the three and is expressly designed for Cortex M MCUs. It focuses on the robustness of the implementation at the cost of only supporting copter drones . We use the SP Racing F3 (SPRF3) board for Cleanflight. It features a 32-bit Cortex M4 core running at 72 MHz, together with a 16-bit 3-axis accelerometer/gyroscope, a 24-bit barometer, and a 12-bit magnetometer.

The implementations of OpenPilot and Cleanflight resemble the design of Ardupilot shown in Fig. 2, but the control logic differs substantially in both sophistication and tuning. This aspect is likely to bear an impact on the effectiveness of reactive control. Further, the boards we employ differ in processing capabilities and sensor equipment, which is instrumental to understand the general applicability of reactive control. We test OpenPilot and Cleanflight by replacing Ardupilot and the Pixhawk board on either the quadcopter or the hexacopter of Fig. 4; however, only Ardupilot supports the Y6. Besides the sensors mounted directly on the boards themselves, the remainder of the equipment on the drones remains the same.

Metrics and setting. To study the accuracy of flight control, we measure the *attitude error*, that is, the difference between the desired and actual attitude. The former is determined by the autopilot to reach the next waypoint and changes seldomly, for example, when reaching a waypoint and turning to the next one The actual attitude is recorded through the on-board sensors. Their difference is the figure the control logic aims at minimizing. If the error was constantly zero, the control would attain perfect performance; the larger this figure, the less effective is the autopilot.

The attitude error is measured along the tree axis as the error between desired and actual raw, pitch, and yaw, shown in Fig. 3. When navigating from one waypoint to the next, all three are subject to the environment influence. To measure the error in a minimally invasive way, we connect a separately-powered RaspberryPI board to the I2C or GPIO interfaces of the boards we test. The modifications to the autopilot to output these information are minimal; the impact of this processing and the added weight of the RaspberryPI and its battery are negligible and equally affect time-triggered and reactive control.

To understand how the accuracy of flight control impacts the drone lifetime, we also record the *flight time* as the time between the start of an experiment and the time when the battery falls below a 20% threshold. For safety, most GCS implementations instruct the drone to return to the launch point upon reaching this threshold. In general, the lifetime of aerial drones is currently extremely limited. State of the art technology usually provides at most half an hour of operation [24]. As a result, this aspect is widely perceived as a major hampering factor.

We test 18 different flight paths in the three environments of Sec. 3. Each path is composed of 8 random waypoints covered at a maximum speed of 3 m/s and 5 m altitude. We repeat every run at least 3 times with every drone. The path loops as long as the battery stays above 20%. Note that it is extremely difficult for RUGBY and ARCH to ensure that the comparisons are performed in the exact same conditions. We use a 5 m tall digital anemometer in the middle of the fields to record the wind speed during an experiment, and consider the conditions comparable within ± 1 knot. The results we present next are based on more than 260 flight hours, and exhibit a variance constantly within 5% of the average value given comparable environment conditions. Running these experiments took almost 8 months, as we often had to wait for the right conditions to run the tests in comparable conditions.

7.2 Navigation

Based on empirical tests and our own experience in developing reactive control, we set $P_{run} = 0.6$ and $T_{failsafe} = .1$ sec. We study the impact of different settings for these parameters in Sec. 7.3.

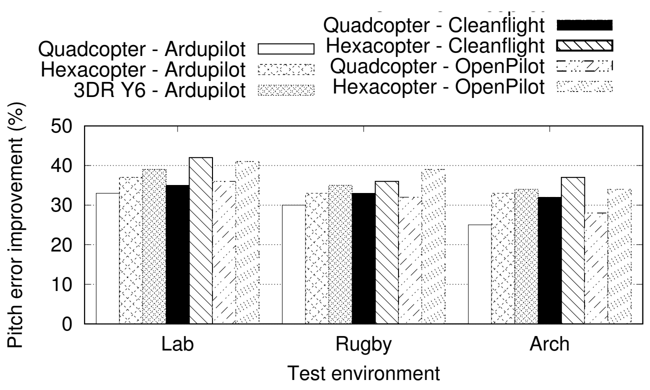


Figure 9: Average improvement in pitch error. *The absolute values range from a 41% improvement with Cleanflight in* LAB *to a 27% improvement with Ardupilot in* ARCH. *The improvements for the latter are proportional to the time control outputs the same decisions, shown in Fig. 5; it is the opportunity to spare iterations of the control loop that enables more accurate control decisions.*

Test environment	Yaw improvement (min/max)	Roll improvement (min/max)
LAB	29%/41%	31%/41%
RUGBY	31%/37%	30%/39%
ARCH	22%/35%	24%/37%

Figure 10: Improvements for yaw and roll across all autopilots and drone types.

Results: attitude error. Fig. 9 shows the average improvements in pitch error enabled by reactive control. The improvements are significant, ranging from a 41% reduction with Cleanflight in LAB to a 27% reduction with Ardupilot in the challenging ARCH. We obtain very similar results, sometimes better, for yaw and roll, summarized in Fig. 10.

The improvements for Ardupilot are proportional to the time the control logic outputs the same decisions, shown in Fig. 5. This confirms that it is the opportunity to spare iterations of the control loop that enables more accurate control decisions. Not running the control loop unnecessarily frees resources, increasing their availability whenever there is actually the need to use them. In these circumstances, reactive control dynamically increases the rate of control, possibly beyond the pre-set rate of time-triggered control. This means reactive control does not necessarily decreases the utilization of resources, but rather moves that utilization at times where the demands are more crucial to satisfy.

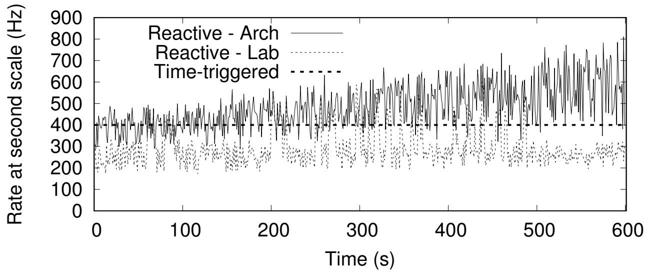


Figure 11: Average rate of control at second scale in two example Ardupilot runs. *Reactive control adapts the rate of control executions both in the short and long term, and according to the perceived environment influence.*

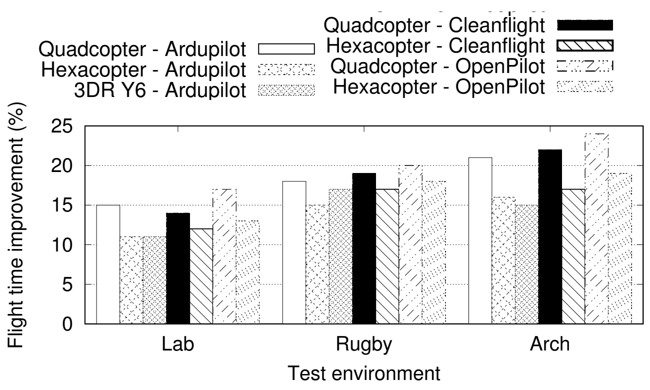


Figure 12: Improvement in flight time. *More accurate motion control reduces the energy overhead in attitude corrections. As a result, battery utilization improves and flight times increase. The improvements are higher in the more demanding settings.*

Evidence of this is shown in Fig. 11, showing an example trace that indicates the average control rate at second scale using Ardupilot and the hexacopter. In ARCH, reactive control results in rapid adaptations of the control rate in response to the environment influence. On average, the control rate starts slightly below the 400 Hz used in time-triggered control and slowly increases. Our anemometer confirms that the average wind speed is growing during this experiment.

In contrast, Fig. 11 shows reactive control in LAB exhibiting more limited short-term adaptations. The average control rate stays below the rate of time-triggered control, with occasional bursts above the rate of time-triggered control whenever corrections are needed to respond to environmental events, for example, when passing close to a ventilation duct. The trends in Fig. 11 demonstrate reactive control's adaptation abilities both in the short and long term.

Still in Fig. 9, the improvements of reactive control apply to the Y6 as well; in fact, these are highest in a given environment. This cannot be attributed to its structural robustness; the Y6 is definitely the least "sturdy" of the three. We conjecture that the different control logic of the Y6 offers additional opportunities to reactive control. This provides evidence of the general applicability of reactive control independent of the control logic. This is also demonstrated by the results for Cleanflight, still shown in Fig. 9. Being the youngest of the autopilot we test, it is fair to expect the control logic to be the least refined. Reactive control is still able to drastically improve the pitch error, by a 32% (37%) factor with the quadcopter (hexacopter) in ARCH.

The results for OpenPilot in Fig. 9 also show how reactive control leads to better hardware utilization. All other settings being equal, the improvements for OpenPilot are higher than with Ardupilot, yet the OpenPilot hardware is the least powerful we test. Sparing unnecessary iterations of the control loop is thus increasingly important, and the effect of reactive control amplifies. This testifies the opportunity for downsizing the hardware, as reactive control can utilize it better and, most importantly, at the right times.

Results: flight time. The improvements in attitude error translate into more accurate motion control. This reduces the energy overhead in attitude corrections, for example, due to operating the motors at higher rates to adjust the trajectory. As a result, battery utilization improves and flight times increase.

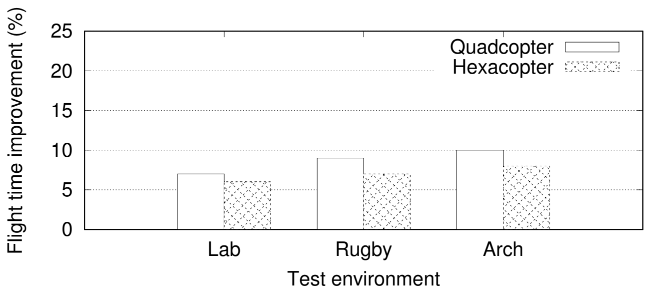


Figure 13: Improvement in flight time without using RP-Embedded, with Cleanflight. *The improvements are still appreciable, but almost halved. Part of the resources freed by reactive control are eaten up by an inefficient implementation.*

Fig. 12 shows the results we obtain in this respect. Reactive control reaches up to a 24% improvement compared to time-triggered control, with a worst-case improvement of 11%. This means flying more than 27 min instead of 22 min with OpenPilot in ARCH, that is, covering 6 to 8 additional waypoints: almost an entire additional lap. This figure is crucial for aerial drones, as discussed earlier. The improvements reactive control enables in this respect are thus extremely valuable.

Perhaps most importantly, these improvements are higher in the more demanding settings. Compared to the results of Fig. 9, the trends are indeed opposite. The better resource utilization enabled by reactive control becomes more important as it is more urgent to react. Similarly, the quadcopter shows higher improvements than the hexacopter. The mechanical design of the latter already makes it physically resilient. Differently, the quadcopter offers more ample margin to cope with the environment influence in software.

7.3 Micro-benchmarks

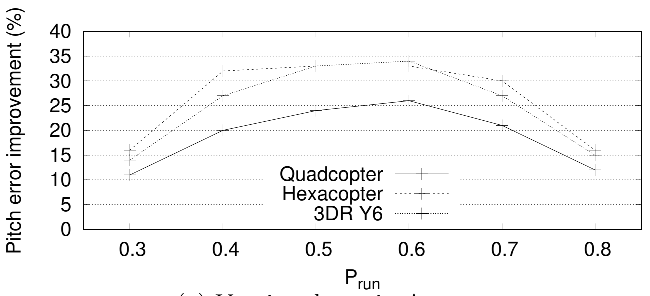
Based on the results of the previous section, we study the impact of employing RP-EMBEDDED for implementing reactive control and the influence of P_{run} and $T_{failsafe}$.

Impact of RP-Embedded. Conceptually, reactive control is not necessarily tied to any specific implementation technique. The question then arises as to what is the role of RP-EMBEDDED in the performance of Sec. 7.2. To answer this, we run 20 hours of experiments with an implementation of reactive control that exclusively employs standard C/C++ and mainstream libraries. Code is always compiled using the GNU GCC toolchain.

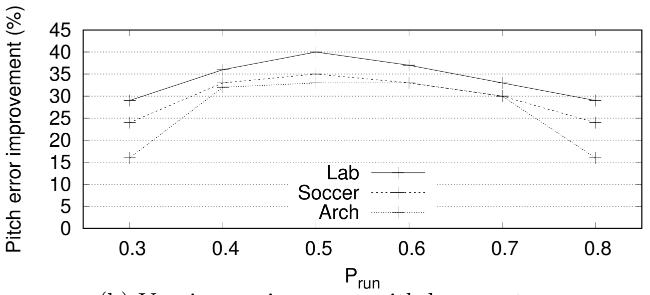
Fig. 13 exemplifies the performance in flight time improvements with this implementation, using Cleanflight running on SPRF3. Compared to Fig. 12, the improvements are still appreciable, but are almost halved now. Basically, at least part of the resources freed by reactive control are eaten up by an inefficient implementation. These results demonstrate that provisioning an efficient implementation is essential to fully harvest the advantages of reactive control, and our RP-EMBEDDED implementation of RP is effective to this end.

Parameter setting. We run another 20 hours of test flights to understand the influence of P_{run} and $T_{failsafe}$ on the performance of Sec. 7.2.

Initially, we vary the setting of P_{run} from 0.2 to 0.8 and keep a fixed $T_{failsafe} = .1$ sec. As example, Fig. 14 depicts the pitch error improvement we observe using Ardupilot. Worth noticing is that for all values and settings we test, the results



(a) Varying drone in ARCH.



(b) Varying environment with hexacopter.

Figure 14: Improvement in pitch error with different P_{run}, using Ardupilot. *Some improvements are obtained for* all *values and settings we test.*

are positive, that is, some improvements are *always* obtained regardless of the parameters' values.

The performance shown in Fig. 14(a) demonstrates the robustness of reactive control against different settings for P_{run} in the most challenging environment. A setting in the $[0.4, 0.7]$ range offers a performance close to the best attainable. Within this range, the probability of false positives in logistic regression as described in Sec. 4 is also very small: it is about 7% right after T_{boot}, and quickly reaches a steady 2%[2]. In Fig. 14(a), the performance only degrades when P_{run} takes extreme values. When P_{run} is too large, control happens too seldom so corrections tend to be abrupt. When P_{run} is too small, the execution approximates time-triggered control, which sub-optimally uses resources. Again, the structural robustness of the hexacopter yields the least sensitivity to different P_{run}: its curve is almost flat in the $[0.4, 0.7]$ interval.

We show the performance with varying P_{run} across the three environments in Fig. 14(b), using the hexacopter. We obtain similar results with the quadcopter and the Y6. We observe how the range of robust values for P_{run} tends to become larger as the environment plays less influence. The curve for ARCH is the most concave, the curve for LAB tends to flatten, and the one for RUGBY lies in the middle.

Based on these results, and without relying on any knowledge on the environment, values in a $[0.5, 0.6]$ range should be preferred. The more the drone can sustain the environment influence, the more it is safe to increase P_{run}, as observed in Fig. 14(a). Similarly, if the environment is ex-

[2]Note that computing the general probability of false negatives would, in principle, require to run the control loop anyways and on the same hardware of reactive control. This would affect the timing of the following iterations and thus the whole execution. We offer next some insights into the probability of false negatives for failsafe executions, which do occur anyways.

pected not to play any strong influence, it makes sense to increase P_{run}, as the range of robust values is probably large as observed in Fig. 14(b).

In a different set of experiments, we fix $P_{run} = 0.6$ and test $T_{failsafe} \in \{.01, .05, .1, .2, .3\}$ sec, again using Ardupilot with all three drones across all environments. The performance in attitude error turns out largely insensitive of the value of $T_{failsafe}$, demonstrating how this is in fact only a safety measure. Moreover, for all failsafe executions, the probability of a false negative compared to the output of logistic regression turns out negligible. The only exception is for $T_{failsafe} = .01$ sec, which yields a slight performance degradation. With this setting, the frequency of failsafe executions starts interfering with reactive control.

7.4 End-user Applications

We demonstrate the benefits of reactive control in end-user applications beyond waypoint navigation.

7.4.1 3D Reconstruction

We perform 3D reconstruction of static objects using aerial pictures [38]. This is a paradigmatic aerial drone application of great societal interest [53]. The procedure requires gathering a predetermined set of aerial pictures, followed by a post-processing step with 3D reconstruction software, such as Photoscan [3].

Setup. In LAB, we attempt the 3D reconstruction of a puppet on a table in the middle of the area. In RUGBY, our target is a car kindly provided by one of the authors[3]. We are interested in the *quality* of the 3D reconstructions. The more stable is the drone while hovering for taking a picture, the less blurred or distorted is the picture. In turn, the better are the input pictures, the higher quality is the resulting 3D reconstruction. The latter is usually measured based on the *density of the point cloud* [38], that is, the number of image points the reconstruction software found to be the same across pictures. This is the information that enables the transition from a 2D plane to a 3D surface.

We use Ardupilot and the hexacopter equipped with a Nikon D5500+ camera. We refactor the PID controllers for Ardupilot's *Hovering* mode, which are different than those for waypoint navigation, to use reactive control. We determine a total of 30 target coordinates where to take a picture, and repeat the experiment 3 times using either reactive or time-triggered control at 400 Hz. Because the number of pictures input to Photoscan is the same in both cases, any improvement due to reactive control is only due to the quality of individual pictures.

Results. Photoscan returns that the point cloud is 23% or 29% more dense when using reactive control in LAB or RUGBY, respectively. Reactive control thus translates into tangible benefits also in a challenging end-user application. Coherently with the earlier observations, these advantages are greater in the more challenging environment. The higher environment influence in RUGBY amplifies the improvements of reactive control compared to LAB.

7.4.2 Search-and-rescue

We demonstrate how reactive control is amenable to implement active sensing functionality.

[3]It was not possible to obtain the needed authorization to place an object of proper size in ARCH; thus, we could not perform any 3D reconstruction experiment there.

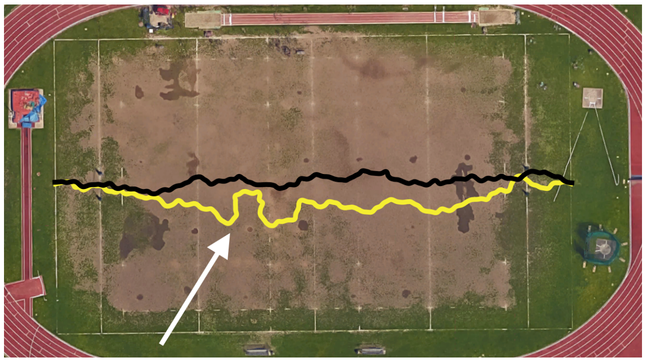


Figure 15: Example of ARVA-driven navigation when using reactive control (black) and time-triggered processing (yellow). *Time-triggered control occasionally produces highly inefficient paths, whereas we never observe similar behaviors with reactive control.*

Setup. Professional skiers are recommended to carry a device called "Appareil de Recherche de Victimes en Avalanche" (ARVA) [40], that is, a radio transceiver operating at 457 KHz and specialized for the purpose of finding people under snow. Normally, the device emits a low-power beacon. Following an avalanche, a rescue team can use another ARVA device as a direction finding device, searching for signals from trapped persons. The searching device generates a "U-turn" signal whenever it detects the owner starts moving away from the victim. Modern ARVA devices reach a 5 m accuracy in locating a transmitter under 10 m of snow [40].

We integrate a Pieps DSP PRO [40] ARVA transceiver with the Pixhawk board through the UART port. A custom PID controller drives the drone's yaw to align it with the direction pointed by the ARVA. Roll and pitch, instead, are determined to fly at constant 1 m/s along the direction indicated by the ARVA. The resulting control does not make any use of GPS information; navigation is entirely determined by the ARVA inputs. We implement this controller both using reactive control by probing the ARVA device as fast as possible, and with time-triggered control at 400 Hz.

We place an ARVA transmitter at one end of RUGBY, and set up the quadcopter at 100 m distance facing opposite to it. Even though GPS does not provide any inputs for navigation, we use it to track the path until the first time the ARVA device generates the "U-turn" signal. We compare the duration and length of the flight when using reactive or time-triggered control. The results are obtained from 20 repetitions in comparable conditions.

Results. On average, reactive control results in a 21% (11%) reduction in the duration (length) of the flight. Besides the averages, time-triggered control shows higher variance in the results, occasionally producing highly inefficient paths. Fig. 15 reports a visual example. The path followed by reactive control appears fairly smooth. In contrast, time-triggered control shows a convoluted trajectory at about one-third of the distance, where the drone's yaw is basically $\pm 90°$ compared to the target.

By looking at the logs, we find the reason to be in the inability of time-triggered control to promptly react. Probably because of sudden wind gusts, at some point the drone gains a lateral momentum. Time-triggered control is unable to react fast enough; a higher than 400 Hz rate would probably be needed in this case, so the drone turns almost 90°. At this point, time-triggered control takes some seconds to remedy the situation. We never observe this behavior with reactive control, which better manages available resources against environment influences.

8. CONCLUSION

Reactive control replaces time-triggered control by governing the execution of the low-level control logic based on changes in the navigation sensors, rather than time. This allows the system to dynamically adapt the control rate to varying environment dynamics. We described how we enable reactive control by conceiving a probabilistic approach to trigger the execution of the control logic, by carefully regulating the control executions over time, and by provisioning an efficient implementation on resource-constrained hardware. We then extensively demonstrated the benefits reactive control provides, including higher accuracy in motion control, longer operational times, and better performance in two diverse end-user applications.

Acknowledgments. We thank Marco Zimmerling for his feedback on earlier versions of this paper. The work was partly supported by the Projects "Zero-energy Buildings in Smart Urban Districts" (EEB), "ICT Solutions to Support Logistics and Transport Processes" (ITS), and "Smart Living Technologies" (SHELL) of the Italian Ministry for University and Research.

9. REFERENCES

[1] 3D Robotics. Advanced flight modes. goo.gl/CeGp6Y.

[2] 3D Robotics. UAV Technology. goo.gl/sBoH6.

[3] Agisoft. Photoscan. goo.gl/1gcwbs.

[4] K. H. Ang, G. Chong, and Y. Li. PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4), 2005.

[5] Ardupilot. Home. goo.gl/x2CHyM.

[6] K. J. Åström. Event based control. In *Analysis and Design of Nonlinear Control Systems*. Springer Verlag, 2007.

[7] K. J. Åström and T. Hägglund. *Advanced PID control*. ISA - The Instrumentation, Systems, and Automation Society, 2006.

[8] E. Bainomugisha et al. A survey on reactive programming. *ACM Comput. Surv.*, 45(4), 2013.

[9] J. Barraquand, et al. B. Langlois, and J. C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man and Cybernetics*, 22(2), 1992.

[10] BBC News. Disaster drones: How robot teams can help in a crisis. goo.gl/6efliV.

[11] G. A. Bekey. *Autonomous robots: From biological inspiration to implementation and control*. The MIT Press, 2005.

[12] A. Benveniste et al. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), 2003.

[13] S. Bouabdallah, A. Noth, and R. Siegwart. PID vs LQ control techniques applied to an indoor micro quadrotor. In *Proceedings of IROS*, 2004.

[14] M. Brambilla et al. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence*, 2013.

[15] D. Brescianini, M. Hehn, and R. D'Andrea. Quadrocopter pole acrobatics. In *Proceedings of IROS*, 2013.

[16] W. Burgard et al. Collaborative multi-robot exploration. In *Proceedings of ICRA*, 2000.

[17] C. Anderson. How I accidentally kickstarted the domestic drone boom. goo.gl/SPOIR.

[18] Cleanflight. Home. goo.gl/uCGmr4.

[19] CNN.com. Your personal $849 underwater drone. goo.gl/m1JRuD.

[20] S. Consolvo et al. Activity sensing in the wild: A field trial of Ubifit garden. In *Proceedings of ACM CHI*, 2008.

[21] CPPReact. A reactive programming library for C++11. goo.gl/Q78cRK.

[22] J. Dai, X. Bai, Z. Yang, Z. Shen, and D. Xuan. PerFallD: A pervasive fall detection system using mobile phones. In *Proceedings of IEEE PERCOM*, 2010.

[23] A. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings of the IEEE International Conference on Computer Vision*, 2003.

[24] DJI. Phantom Drone. goo.gl/wLbVic.

[25] J. Edwards. Coherent reaction. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009.

[26] R. M. Faragher et al. Captain Buzz: An all-smartphone autonomous delta-wing drone. In *Workshop on Micro Aerial Vehicle Networks, Systems, and Applications (with ACM MOBISYS)*, 2015.

[27] Good-ARK Semiconductors. eCompass modules with 3D accelerometer and 3D magnetometer. goo.gl/D0CH6y.

[28] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[29] Intel. Threading building blocks. goo.gl/Wlvp44.

[30] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1), 1986.

[31] Libelium Inc. WaspMote. www.libelium.com.

[32] Q. Lindsey, D. Mellinger, and V. Kumar. Construction with quadrotor teams. *Autonomous Robots*, 33(3), 2012.

[33] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004.

[34] N. Michael et al. Cooperative manipulation and transportation with aerial robots. *Autonomous Robots*, 30(1), 2011.

[35] E. Miluzzo et al. Sensing meets mobile social networks: The design, implementation and evaluation of the CenceMe application. In *Proceedings of ACM SENSYS*, 2008.

[36] L. Mottola, M. Moretta, C. Ghezzi, and K. Whitehouse. Team-level programming of drone sensor networks. In *Proceedings of ACM SENSYS*, 2014.

[37] E. Natalizio et al. Filming sport events with mobile

camera drones: Mathematical modeling and algorithms. goo.gl/v7Qo80, 2012. Technical report.

[38] F. Nex and F. Remondino. UAV for 3D mapping applications: A review. *Applied Geomatics*, 2003.

[39] OpenPilot. Home. goo.gl/D89lkb.

[40] Pieps. ARVA Transceivers. goo.gl/tPywra.

[41] PixHawk.org. PX4 autopilot. goo.gl/wU4fmk.

[42] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Real-time computer vision with OpenCV. *Communications of the ACM*, 55(6), 2012.

[43] R. Ritz et al. Cooperative quadrocopter ball throwing and catching. In *Proceedings of IROS*, 2012.

[44] I. Sadeghzadeh and Y. Zhang. Actuator fault-tolerant control based on gain-scheduled PID with application to fixed-wing unmanned aerial vehicles. In *IEEE International Conference on Control and Fault-Tolerant Systems*, 2013.

[45] SciTools. Code analysis with Understand. goo.gl/0VZGAc.

[46] Sodium. Functional reactive programming (FRP) for C++. goo.gl/vwJH1P.

[47] SparkFun. Low current sensor breakout - ACS712. goo.gl/VPV3HW.

[48] SpiraFRP. Yet another C++ reactive programming library. goo.gl/HfDQsT.

[49] ST Microelectronics. Ultra compact high performance e-compasses. goo.gl/TLrdy5.

[50] M. Turpin, N. Michael, and V. Kumar. Decentralized formation control with variable shapes for aerial robots. In *Proceedings of ICRA*, 2012.

[51] U.S. Environmental Protection Agency. Air Pollutants. goo.gl/stvh8.

[52] Vicom. Motion capture systems. goo.gl/Vh5Q4c.

[53] J. Villasenor. Observations from above: Unmanned Aircraft Systems. *Harvard Journal of Law and Public Policy*, 2012.

[54] P. Wardle. Personal submersible drone for aquatic exploration. goo.gl/XTVgQF, 2015. US Patent App. 14/143,713.

[55] M. Yim et al. Modular self-reconfigurable robot systems. *IEEE Robotics Automation Magazine*, 14(1), 2007.

[56] M. Zhuang and D. Atherton. Automatic tuning of optimum PID controllers. *IEEE Proceedings on Control Theory and Applications*, 140(3), 1993.