jUAV: A Java Based System for Unmanned Aerial Vehicles

Adam Czerniejewski, Shaun Cosgrove, Yin Yan, Karthik Dantu, Steven Y. Ko, Lukasz Ziarek
SUNY Buffalo
{adamczer, shaunger, yinyan, kdantu, stevko, lziarek}@buffalo.edu

ABSTRACT

Paparazzi UAV is one of the most popular open-source UAV platforms in use today. The code base of Paparazzi UAV has served as the basis for both C and Java open-source benchmarks for real-time embedded systems. These benchmarks, PapaBench and jPapaBench, respectively, have been widely accepted by their respective communities as they provide a representative and complex workload. Both PapaBench and jPapaBench, however, do not utilize a functional simulation in their execution, but rather focus on testing the contention between tasks. In this paper, we first detail our attempt at adapting jPapaBench to utilize the New Paparazzi Simulator (NPS) developed by the Paparazzi UAV team which and provides the underlying sensor models and allows for the use of different Flight Dynamic Models (FDM) which simulate actual environmental influence on the airframe [1]. We then compare jPapaBench performance to its C-based inspiration, Paparazzi UAV utilizing JSBSim as the FDM for both systems. Our results show that jPapaBench is quite different in implementation and performance from Paparazzi UAV. Following this finding, we target a more direct port of Paparazzi UAV to Java named jUAV using a one-component-at-a-time approach allowing for each component to be validated in isolation for correctness. The paper then compares the correctness of the initial prototype of jUAV consisting of a subset of the Paparazzi UAV code implemented in Java. In future work jUAV will be greatly beneficial to the real-time Java community since it will provide a means for benchmarking real-time JVMs with representative real-world workloads and allow for an open-source solution used for flying airframes using real-time Java.

1. INTRODUCTION

There is a growing need in the real-time Java community for additional benchmarks as well as representative open-source applications to compare, test, and validate virtual machine implementations, language specification implementations, as well as new research. While a number of benchmarks like CDx [13], jembench, and jPapaBench [14] [1] exist, to date there is no representative open-source application that can be deployed in simulation and experiment. Autonomous flight control is a good candidate for such a representative application, as it is comprised of many complex subsystems, heavily leverages I/O through sensor processing and actuator control, is usually coupled with a payload application and therefore has mixed-criticality requirements, and demands strict timeliness guarantees for correctness of execution.

This autonomous flight control or autopilot is generally part of larger system consisting of a ground station and the actual aerial vehicle. The autopilot module which stabilizes the aircraft must exist on the aerial vehicle as it must make adjustments based on the environment quickly to remain in flight. The route that is followed by the aircraft can be directed by a predefined in the embedded autopilot allowing for truly autonomous flight, or directed by the ground station. In most systems the ground station not only provides a means to collect data from the aircraft while it is in flight, but also issue commands for takeoff, ascend, descend, land, etc..

The real-time Java community has already explored flight control through the JAviator project [10] and jPapaBench [14]. The JAviator project provides a real-world UAV implementation. The ground control implementation is partially developed in real-time Java, but the onboard systems for the UAV are in C. Based on Paparazzi UAV [2], an autopilot for unmanned aerial vehicles (UAVs) and the PapaBench benchmark derived from Paparazzi UAV, jPapaBench is a complex benchmark workload consisting of many tasks encompassing an autopilot module, a fly-by-wire module, and a simulator module. As a benchmark, jPapaBench provides a way for the community to compare VMs as well as different versions and specifications of real-time Java, including the real-time specification for Java (RTSJ) [8] [3] and safety critical Java (SCJ) [11] [4]. It is particularly suited for a VM deployed on a resource constrained device, as it leverages multiple threads and a non-trivial amount of memory, thereby stressing the VM's scheduling infrastructure and the memory management implementation.

However, as it stands, jPapaBench has several shortcomings that do not make it suitable as a representative application for real-time Java. Since jPapaBench was developed as a benchmark, it makes a number of simplifications. First, it leverages pre-generated sets of sensor values for its workloads. Although these values have been generated by a simulator, they represent an idealized flight with no requirement of stabilization and adjustment of the flight path at runtime. Second, jPapaBench lacks the communication component for

[1] http://code.google.com/p/jpapabench/
[2] http://wiki.paparazziuav.org
[3] http://www.rtsj.org/docs/rtsj_1.0.2_spec.pdf
[4] http://download.oracle.com/otn-pub/jcp/safety_critical-0_94-edr2-spec/scj-EDR2.pdf

interaction with the ground station. Finally, it does not simulate full physics, and therefore represents a simplified flight dynamic model and simplified simulation of on-board sensors, both of which result in a drastically reduced computation and a flight control algorithm that is not directly usable on an actual UAV.

Our ultimate goal is to create a Java based UAV system for benchmarking JVMs on real-world UAV deployments and providing an open-source representative application for the real-time Java community. In this paper we introduce our prototype system called jUAV, consisting of just over 11,000 lines of code in its current state, as well as its design and preliminary experiments. jUAV is a more comprehensive version of jPapaBench that is capable of working with the Paparazzi UAV's framework. Specifically, our effort will augment jPapaBench to benchmark JVMs on more realistic application loads in three aspects:

- Task scheduling with explicit synchronization between tasks
- Adding communication based I/O for communicating telemetry to the ground station and visualization components of Paparazzi UAV.
- Adding sensor based I/O for communication to an advanced flight dynamic model with a full fledge physics simulator for precise flight simulation and sensor readings.
- Finally, we would like to have a benchmark that can be run both in simulation as well as on an actual UAV

In the remainder of this paper, we first detail the difference between jPapaBench and Paparazzi UAV, and explain the need for jUAV as a benchmark for the real-time JVMs. We then present jUAV, a port of Paparazzi UAV in Java, as a real-time Java benchmark application that runs on real hardware devices. The current stage of jUAV is a partial port of the Paparazzi UAV code that uses cyclic executive to control the execution of the autopilot logic, facilitating a hybrid jUAV consisting of both Java and Paparazzi UAV autopilot allowing for validation for the correctness of our ports as each component is transitioned to Java. Experiments are then presented comparing the component of the autopilot logic that performs the horizontal attitude stabilization of an airframe for both the original Paparazzi UAV and jUAV's Java implementation. Once the jUAV autopilot is completely running in Java we can leverage the knowledge of jPapaBench to translate the cyclic executive mode to the task-based model. The details of the development plan are discussed in Section 2.3.

2. BACKGROUND AND MOTIVATION

In this section, we first briefly discuss the architecture of Paparazzi UAV, PapaBench, and jPapaBench. We then explore jPapaBench's limitations as a real-time JVM benchmark. Finally, we outline the roadmap of jUAV project.

2.1 Paparazzi UAV

Paparazzi UAV [3] is an advanced and proven autopilot that handles flight autopilot logic such as navigation, flight control, sensor management, and wireless/radio communication. It provides a set of GUIs to configure autopilot hardware settings and flight course planning. The configuration files produced by the GUIs are then compiled into autopilot logic, and deployed to the embedded autopilot control board. Configuration includes hardware specific

[1]The boxes with color are the actual components for the real-time aircraft, the boxes without color are simulated components for testing and tuning.

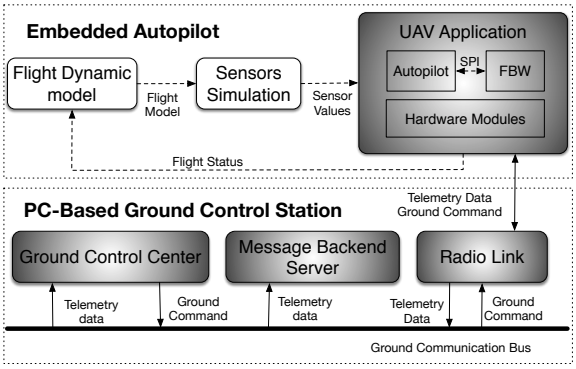


Figure 1: Paparazzi UAV Architecture[1].

configuration, where different types of sensors that are present on the target platform are selected. Additionally, Paparazzi UAV has support for both fixed-wing and rotor craft, including popular quadrotor UAVs. The type of airframe is also selected at configuration time.

Execution of the autopilot can be done in one of two modes: simulation or physical deployment. In simulation the autopilot is hooked up to a physics simulator and typically is executed on a PC. A physical deployment requires flashing the embedded board or micro controller on the hardware of the UAV to be deployed. During autopilot execution, Paparazzi UAV can monitor and record the status of the autopilot for the real-time visualization and off-line analysis at its PC-based ground control station. Communication is handled through a custom protocol, either executed on a PC in the case of simulation or via wireless communication in case of a physical deployment. Figure 1 shows an overview of the Paparazzi UAV architecture divided in two parts: (1) Ground Control Station (GCS) consists of a radio link, a ground control center and a backend message server; and (2) Embedded Autopilot, an Unmanned Aerial Vehicle (UAV) application, includes three sets of fundamental features.

- **Autopilot Logic** runs autonomous navigation and aircraft stabilization, and generates servo commands.
- **Fly−By−Wire** (FBW) module receives remote control commands and drives aircraft's servos based on the commands received and the low-level safety options.
- **Hardware modules** collect sensor values and provide control to aircraft servos.

The timing requirement of the autopilot logic, the ability to test it in simulation as well as in experiment on actual hardware, and its availability as open-source software makes Paparazzi UAV an ideal candidate for a real-time benchmark application. A simplified version of Paparazzi UAV, PapaBench [16], has been adapted to test real-time embedded systems. Although Paparazzi UAV's autopilot is a cyclic executive, PapaBench has modeled Paparazzi UAV's UAV application in Architecture Analysis and Design Language (AADL). It analyzed the original code base, and split the autopilot logic into separate tasks with timing constraints and dependencies.

2.2 jPapaBench: A Real-time Java Benchmark

jPapaBench is derived directly from PapaBench [16] and consists of roughly 10,000 lines of code. jPapaBench takes task configurations generated by PapaBench, and implements them as real-time periodic threads. The original PapaBench implementation was

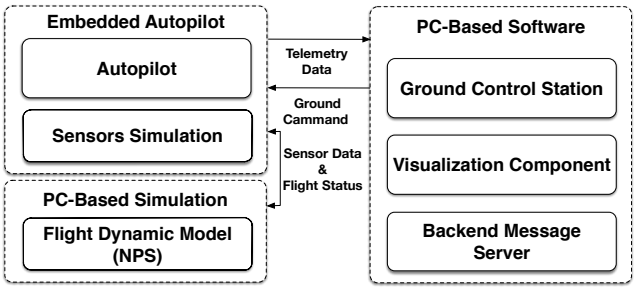


Figure 2: jUAV Architecture with Simulation.

not intended for functional execution (processing real data from hardware), but rather for evaluation of WCET analysis tools. Thus, the implementation of the autopilot logic in jPapaBench is significantly simplified, and mainly aimed to get its tasks to behave correctly with timing constraints. Specifically, there are two main design points that make jPapaBench different from the real-world context in Paparazzi UAV.

Firstly, the design of jPapaBench does not take the simulation of the hardware architecture into account. In Paparazzi UAV, the autopilot logic involves sensor data readings and bi-directional data transfer via wireless communication (telemetry data and ground commands) between the aircraft and the ground station. In jPapaBench, the sensor simulation is simplified to copying simulated sensor values from the flight dynamic model of jPapaBench to the autopilot logic; the wireless communication is completely omitted. Similarly, the autopilot logic and FBW components are running on two different computational units communicating via a hardware bus in Paparazzi UAV. jPapaBench merges them into a single application that simulates the communication bus through a cyclic queue. However, the hardware interrupt handling and data buffering can affect the actual execution of the UAV application at runtime as these operations may introduce uncertainties during execution, such as task preemption, buffer overwriting *etc*., that a VM would need to deal with.

Secondly, since jPapaBench is based on PapaBench, its design also does not consider integration with other tools provided in Paparazzi UAV, such as the physics simulation provided by NPS and the ground station. According to our observation of the current jPapaBench source code, the simulated flight model in jPapaBench takes the current flight position and speeds and generates sensor readings for later autopilot logic, such as navigation or stabilization. Such simplicity results in unrealistic flight behaviors such as 1) the flight can only fly a straight line; 2) the flight is ever increasing altitude (i.e. the aircraft is always ascending), both of which are not representative. More detailed results will be discussed in Section 4. In addition, a subset of the tasks implemented in jPapaBench, especially in the FBW components, are quite different from their counterparts in Paparazzi UAV. For example, the safety checking and telemetric logic is either hard coded or not implemented, since it is not aimed to interact with the ground station.

Lastly, jPapaBench, does not provide any integration with the configuration tools in Paparazzi UAV, including tracing and debugging support. Such lack of integration impedes further development as it makes debugging and testing more difficult.

2.3 Design Goals in jUAV

This paper is a first step in a larger effort that aims to create a fully working Java autopilot (jUAV) running on real-world hardware. In this paper, we show our current prototype of jUAV, which has ported a subset of the autopilot logic for rotor-craft UAVs in Paparazzi UAV. The resulting Java application has three major parts running in cyclic executive mode: (1) A portion of the stabilization algorithm from the original Paparazzi UAV. (2) A set of JNI functions that interface the Java-based stabilization with flight model simulation and ground station in Paparazzi UAV. (3) a set of auxiliary functions for conversion between different coordination systems, arithmetic and geometric computation with respect to rotation and motion. The resulting artifact of the current prototype jUAV provides us a way to prove the viability of our porting methodology described in section 3.2, and verify the correctness of our Java-based autopilot logic against the ground truth — the original simulation in Paparazzi UAV.

The next step of jUAV is to completely port autopilot logic in Java, convert the cyclic executive mode to task-based mode, and isolating the Java-based autopilot logic from flight model simulation. We will split the current prototype into two executing units: one is a simulation-based autopilot application written in Java, another one is purely a flight model simulation as it is in Paparazzi UAV. Such isolation requires extra communication channels from the embedded autopilot to the PC-based simulation and the ground station, as shown in Figure 2. Such communication introduces extra I/O operations for on-board hardware/simulation, as the communication is enabled via TCP/IP or serial channels. Table 1 provides a side-by-side comparison among the simulated autopilot in original Paparazzi UAV, jPapaBench, and jUAV.

Finally, if the simulation-based autopilot in jUAV can fly with the PC-based simulation as the autopilot in Paparazzi UAV, it will give us confidence in its correctness and a transition to an actual UAV can follow. It is at this point that jUAV can be refactored to work with not only standard JVMs but those that implement the RTSJ or SCJ standards. Once complete, engineering time will be spent on the integration with the hardware drivers required to fly on actual hardware.

3. IMPLEMENTATION EXPERIENCE

As discussed in Section 2.3, jUAV aims to implement a Java version of Paparazzi UAV that can be executed in real-time and also on a hardware platform. The first step is to create a Java version of the Paparazzi UAV autopilot running on a traditional JVM, and enable interaction between the Java autopilot application, NPS simulator and the ground station in Paparazzi UAV. The immediate question to ask is "Why not implement jUAV using jPapaBench as the starting point?", since it is a well-accepted real-time benchmark for real-time JVMs. In this section, we first report our experience in extending jPapaBench to interact with the ground station in Paparazzi UAV, and discuss the reason why we chose to implement jUAV by porting the native source code of Paparazzi UAV. Finally, we present our system design with details in Section 3.2.

3.1 Integrating jPapaBench with Ground Station in Paparazzi UAV

The autopilot logic of jPapaBench is much simpler than the current implementation found in Paparazzi UAV, since jPapaBench [14] is derived from PapaBench [16], which itself is a watered down version of a four-year-old snapshot of the embedded autopilot code in Paparazzi UAV. We have conducted an experiment that compares the recorded flight paths between the various versions and present them later in Section 4.2. In this section we discuss two major differences between the jPapaBench and Paparazzi UAV implementation in detail to provide a better intuition as to why the flight paths vary significantly.

Different Coordinate Systems: In practice, there are several co-

	Paparazzi UAV	jPapaBench (PapaBench)	Current Prototype jUAV	Simulation-based jUAV
Flight Dynamic Model (FDM)	NPS in a single application	Self-contained computation in a single application	NPS via JNI	NPS via JNI in the simulation application
Sensor Simulation	Simulated devices with local data			Simulated devices connecting to the autopilot application
Completeness of FBW	Simulated radio link with GCS; Failsafe checking	placeholder tasks	via JNI	Same as Paparazzi UAV
Completeness of Autopilot	Same as the UAV application on hardware	close to Paparazzi UAV	Horizontal Stabilization, Remainder via JNI	Same as Paparazzi UAV
Execution Mode	Cyclic executive	Task based	Cyclic executive	Cyclic executive or Task based

Table 1: Autopilot application comparison among Paparazzi UAV simulation, jPapaBench and jUAV

ordinate systems used to represent the navigation, control and positioning of an aircraft [4]. For example, an aircraft is normally modeled with body coordinates in Euler angles or a rotation matrix [15]. For autopilot navigation, two systems are used: North-East-Down (NED) system for GPS and Earth-center, Earth-fixed (ECEF) system, and an inter-mediate system that transforms NED to Universal Transverse Mercator (UTM) for displaying in ground station. However, jPapaBench utilizes longitude, latitude, and altitude (LLA) for its internal representation and does not implement any transformations between the aforementioned coordinate systems. This simplified coordinate system prevents any interaction between the autopilot of jPapaBench and the ground station of Paparazzi UAV, such as dynamic guidance to the autopilot or displaying the autopilot position.

Fly-By-Wire (FBW) Logic for FailSafe: Paparazzi UAV has several built-in failsafe features ranging from automatic navigation mode to manual control mode. The FBW logic includes the basic functionality for failsafe, including reading commands sent via remote control, reading the autopilot commands, driving the servos etc. However, jPapaBench only provides a shell implementation of the FBW logic. According to our observation, most of the failsafe checks in jPapaBench are hard coded or omitted. Since some of the FBW logic is incomplete and mostly inconsequential, this dead code may be eliminated via compilation, changing the workload and affecting task scheduling at runtime.

3.2 Java Based Autopilot

As one can imagine, the code base of a UAV platform like Paparazzi UAV is enormous. As discussed in Section 2.1, we notice that Paparazzi UAV can be broken into two main components, the Ground Control Station and the onboard autopilot. For the purposes of jUAV, there is no need to reimplement the ground station as it merely provides a means for the user to monitor and control the air frame in both simulation and actual flight. Like both PapaBench and jPapaBench, jUAV focuses on the onboard autopilot. However, jUAV will integrate directly to the existing Paparazzi UAV Ground Station and NPS simulation engine for testing. This ability to focus on porting just the autopilot will not only facilitate a means of creating a benchmark that utilizes proven physics simulations, but also allow us to reuse the well-tested Paparazzi UAV user components.

From investigation of the NPS simulation found in the Paparazzi UAV code base, it is noted that NPS is essentially an extension to the normal cyclic executive that facilitates communication between the core Paparazzi UAV system and the simulation engine. This extension allows for the simulated sensor values to be used in the autopilot and rotor commands produced to be returned to the simulator. This extension is depicted in Figure 4. We can see the overall flow of the cyclic executive that is run in the NPS simulation. In the initial port, the atmosphere characteristics are omitted since their influence only allows the user to test environmental stimuli transitively on the airframe via the physics simulation.

To facilitate the use of the Flight Dynamic Model, an underlying physics simulator (JSBSim) and to allow for a fully functioning test environment, these components are currently triggered via JNI calls in the cyclic executive. In the following block, each of the sensors take on the form of a specialized periodic task `ISensor<SensorReading>`. All periodic tasks provide two main functions: an `init` and `execute` method. The sensors job is to get the values from their respective hardware during execution if a predefined interval has passed and convert any values as needed. The hardware is simulated by the JSBsim and the values are copied into their respective `SensorReading` data structures. In the next block, the NPS's autopilot extension executes checks to determine if there are new readings in the sensors, performs any required computations, and populates the autopilot's native data structures via JNI calls.

For the prototyping of jUAV discussed in this paper, the Java autopilot implementation was scoped to the horizontal attitude stabilization portion of the autopilot. To facilitate the ability to focus on such a specific portion of the autopilot while ensuring a fully functioning simulation, a series of JNI calls were introduced to the original Paparazzi UAV code to maintain the remaining functionality.

For the call-graph in Figure 3 the shaded sections shown have been ported into the jUAV code as Java, the remainder of the blocks are facilitated using a series of encapsulating JNI calls to allow normal execution. This approach of using JNI to port one section of the call graph at a time ensuring that each component can be tested in isolation ensuring correctness prior to moving on. The porting process consists of identifying the component to be ported, horizontal attitude stabilization in the case of this paper, and beginning as close to the cyclic executive as possible. Then one function at a time is ported to Java and any calls deeper are filled in with JNI. Once the ported logic has been tested by executing the Paparazzi UAV simulation verifying correctness the process is repeated until the targeted leaf call has been verified correct. After a module has been completed and uses no JNI the code may then be refactored to be more object oriented. Section 4.2 and Section 4.3 detail perfor-

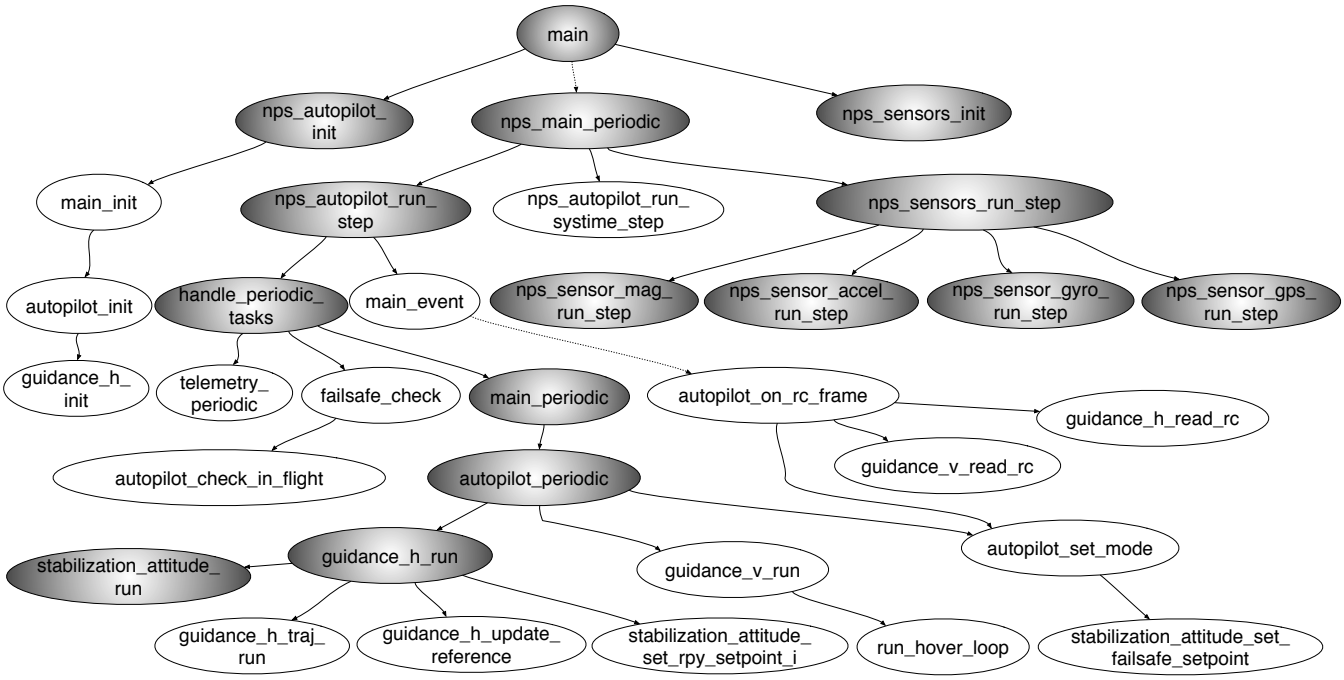


Figure 3: Simplified Program Call Graph for Paparazzi UAV

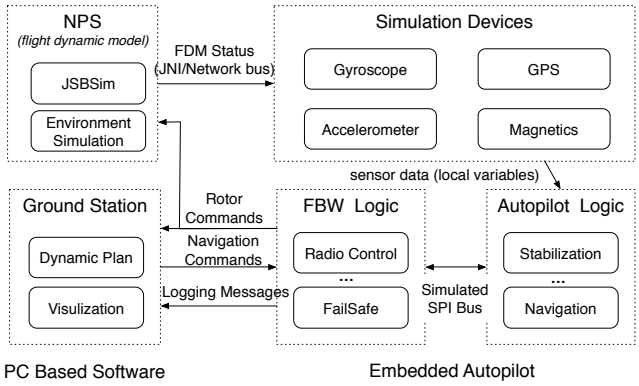


Figure 4: jUAV Data Flow Chart

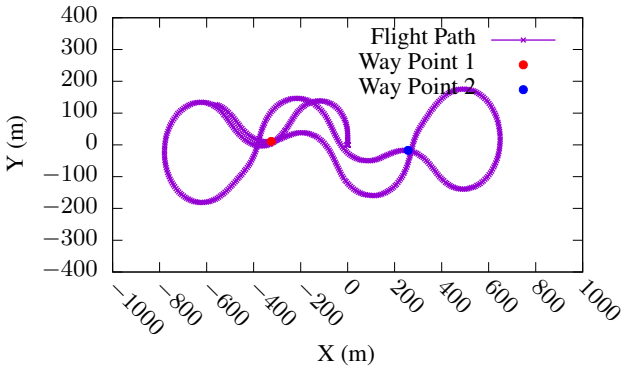
mance and correctness comparison between jUAV and Paparazzi UAV will be discussed. In `stabilization_attitude_run` the values required for computation are fetched from the native Paparazzi UAV computed on and the results are returned to the native Paparazzi UAV data-structures.

The current control mechanism in the jUAV code consists of a cyclic executive who's initialization function creates the periodic tasks and assembles them into a list. The set of periodic tasks that currently exist within jUAV are the JNI calls to FDM, a Java implantation of each of the sensors, and the hybrid autopilot task consisting of both Java and JNI as seen in Figure 3. These tasks then continually run in the order they were added to the list during execution. Such task scheduling to make a cyclic executive will easily allow the control loop to be changed to a multi threaded task based controller. Transitioning to a task based controller will allow the autopilots tasks to operate with different importances the ability to test mixed-criticality requirements.

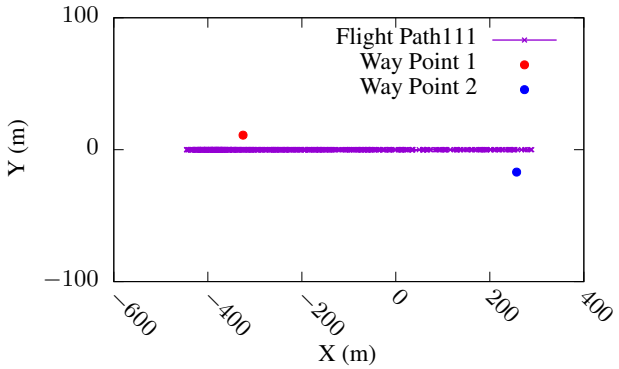
3.3 Physics Simulation

To enable more realistic simulation, we integrate the Java-based autopilot with the NPS simulation in Paparazzi UAV. The original autopilot is coupled with the NPS simulation, and runs both autopilot logic and simulation computation in a single process with cyclic executive. The simulated sensor devices are a logical division that can be used to split the NPS simulation and autopilot logic, as the data structure they compute on are identical, i.e., the autopilot logic uses the data structure that represents the status of autopilot and the NPS simulation utilizes its own flight model data structure. In the cyclic execution, the status generated in the FDM is fed into the simulated sensors for the autopilot logic, and the autopilot logic produces servo commands that are sent back to the FDM.

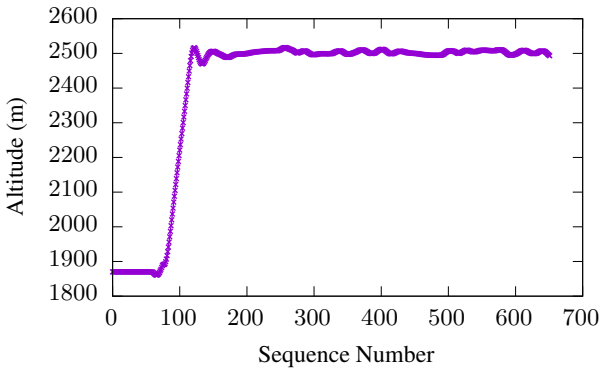
In the prototype phase, we focused on allowing the C based implementation to be utilized in Java. The interfaces between the Java autopilot and the NPS simulation have been implemented as a set of JNI calls. In the completed implementation, the use of a system network bus will replace the JNI calls, since the autopilot logic and the NPS simulation will run on separate computational units. A major hurdle for the communication is to ensure the messages are formed correctly and passed appropriately through different channels. It will be imperative to ensure appropriate processing of messages while maintaining the scheduling of the autopilot logic, transitioning data between the PC based NPS simulation and the embedded autopilot. As Figure 4 shows, to keep the realistic nature of our simulated Java autopilot, a simulated SPI bus with a two socket pipeline that enables bi-directional communication between the autopilot logic and FBW logic will be implemented. The communication between the PC based software and embedded autopilot can leverage either Ethernet or serial as simulated multi-channel radio communication. According to our study, the downlink data in Paparazzi UAV communicates the motor commands and autopilot position information to the ground station, which are critical to ensure the correctness of the autopilot logic. In the real-world autopilot in a deployed version of Paparazzi UAV, these messages



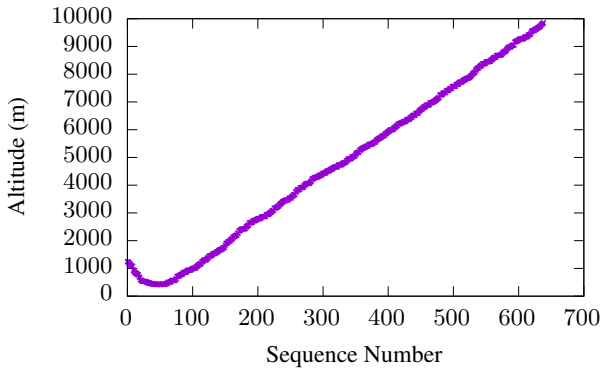
(a) X, Y GPS on Paparazzi UAV



(b) X, Y GPS on jPapaBench



(c) Altitude on Paparazzi UAV



(d) Altitude on jPapaBench

Figure 5: Flight Path Comparison on Simulated Microjet Between Paparazzi UAV and jPapaBench

can use dedicated channels for radio communication where as the rest of the logging messages share the same channel. In jUAV, we will simulate this behaviour with different TCP/IP communication channels.

4. EVALUATION

To evaluate how close to reality jPapaBench simulations are, we show a comparison of flight paths between Paparazzi UAV and jPapaBench. To show the correctness of the jUAV prototype, we present a flight path comparison between jUAV and Paparazzi UAV. The performance of jUAV is compared Paparazzi UAV by measuring the time cost associated with the horizontal attitude stabilization components computation for both Paparazzi UAV and jUAV's Java implementation. All of the experiments are conducted using Oracle Java version 1.7.0_80, on a standard personal computer, equipped with Intel i7-4700 Processor and 16GB RAM running Ubuntu 14.04 LTS.

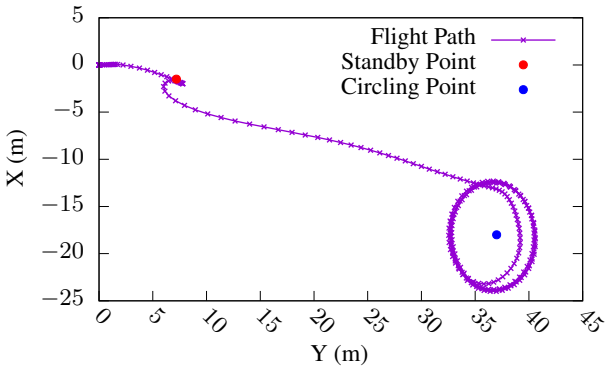
4.1 Flight Path: Paparazzi UAV v.s jPapaBench

To show the difference between jPapaBench and Paparazzi UAV, we have conducted an experiment comparing the recorded flight path on both of their UAV systems. Since jPapaBench can only fly a fixed-wing flight with its built-in flight plans that guide the flight to pass different way-points, we configure the simulated autopilot in Paparazzi UAV to mirror the same flight path. Thus, both of the flight plans are configured to travel between two way-points, i.e., the autopilot is first launched from its start position, climbs in altitude and then navigates to the first way-point and then to the second.

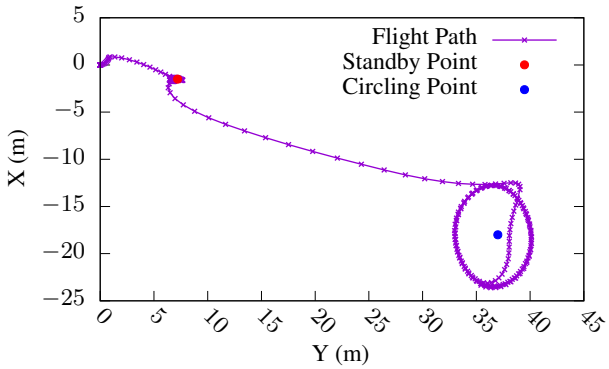
The coordinates and altitude are derived from messages the UAV sends to the ground station during simulation. Each flight path graph is offset such that the plane starts at the [0,0] position in the N and E plane. As the coordinate system for the GPS messages is North East Down (NED), these are the values we chose to represent in the graph but for simplicity, they can be viewed as a distance from the initial position in X and Y coordinates. Figure 5a and Figure 5b shows the comparison of the flight path for Paparazzi UAV and jPapaBench. The change in altitude over time is given in Figure 5c and Figure 5d. In Paparazzi UAV, the flight makes dynamic adjustments in its flight path until it precisely reaches the way-point. The flowing nature of the graphs represent the interaction of the simulated physical plane, the physics of the planes of motion in the atmosphere and the autopilot trying to meet its targets. However, inspection of the flight path created in jPapaBench highlights the fact that the simulated flight model is much more simplistic and just computes the route for aircraft by directly targeting the next way point with no realistic dynamics. It is evident that the lack of simulation of physics results in a simplified flight trace as seen in Figure 5b.

4.2 Flight Path: Paparazzi UAV v.s jUAV

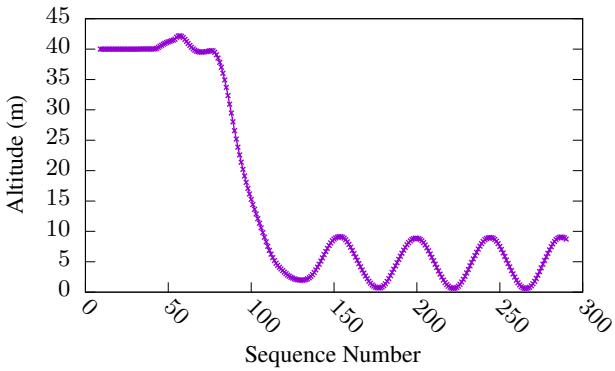
To show the correctness of our Java Based autopilot implementation in jUAV, we have repeated the above experiment on Paparazzi UAV and jUAV. However, in this case, our ultimate airframe is a



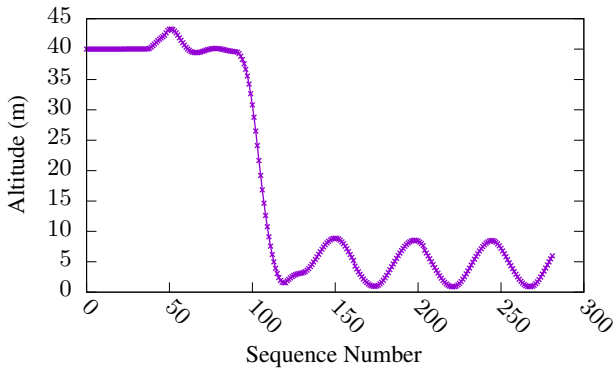
(a) X, Y GPS on Paparazzi UAV



(b) X, Y GPS on jUAV



(c) Altitude on Paparazzi UAV



(d) Altitude on jUAV

Figure 6: Flight Path Comparison on Simulated Quad_LisaM_2 Between Paparazzi UAV and jUAV

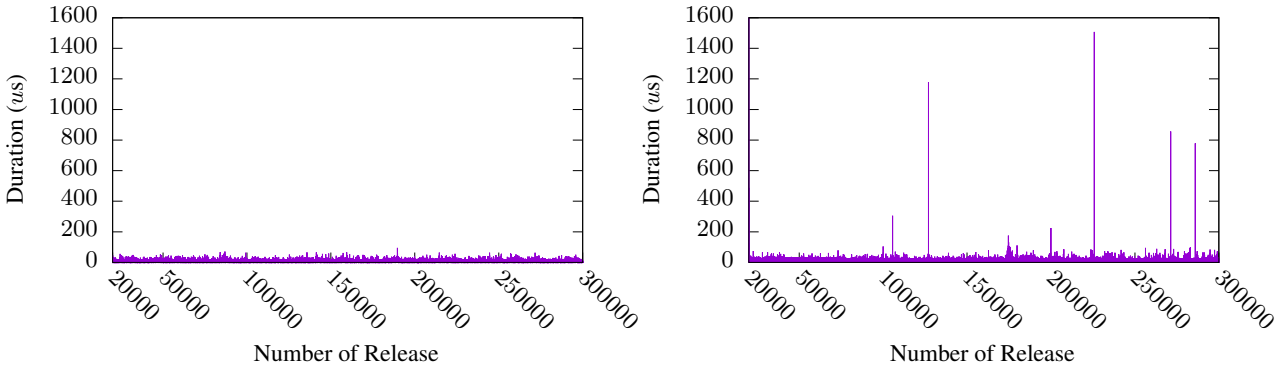
quad-copter, and the comparison is based on a rotor-craft UAV simulation for the (`Quad_LisaM_2`). During these experiments, both autopilots are guided by the ground station to follow the same flight plan. This plan begins with normal take off procedures followed by an ascent. Upon ascent, the air frame stabilizes at a standby position near the point of take-off. Shortly after the airframe then moves towards a point that it will circle for five iterations.

The coordinates and altitude are the same as the previous comparison between Paparazzi UAV and jPapaBench, but we do not need to calibrate the origin of the coordinates with the starting point, as both Paparazzi UAV and jUAV are using the same starting point and way points. Figure 6a and Figure 6b show the comparison of the flight path between Paparazzi UAV and jUAV. The change in altitude over time is given in Figure 6c and Figure 6d. As the figures indicate, the flight path of jUAV is nearly identical to the one in Paparazzi UAV. This makes sense since the Java based autopilot in jUAV mirrors the exact same autopilot logic in Paparazzi UAV. There is some deviation between the flight paths of Paparazzi UAV and jUAV. This deviation likely comes from the fact that during the simulation Paparazzi UAV and our derivative work add a random noise to the sensor reading as it enters the sensor data-structures to simulate the precision of actual sensor readings. Though there are differences it is clearly observed that both UAVs flight paths once stabilized circle the Circling Point in identical circle.

4.3 Timing Performance: Paparazzi UAV v.s jUAV

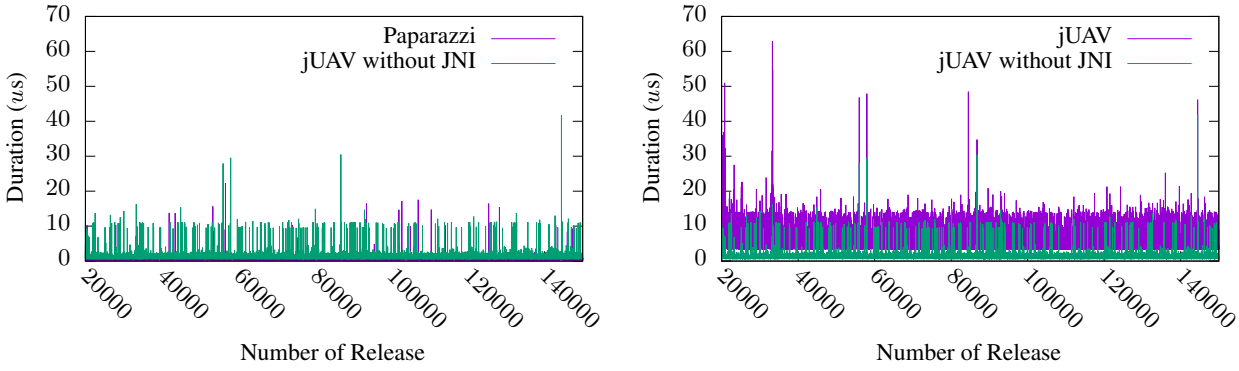
To evaluate the timing performance and verify the viability of the Java based UAV autopilot, we have collected the time duration for periodically released logic including: 1) the entire autopilot logic, 2) the stabilization logic, which is one of the critical computations in the autopilot logic. The experiments are executed over 5 minutes on both Paparazzi UAV and jUAV. To warm up the JVM for jUAV, we discard the first 20,000 samples, and compare the time duration for each iteration on the remaining samples between both experiments.

Figure 7 shows the comparison of time durations for autopilot logic. All time durations on Paparazzi UAV are ranging from 15 μs to 200 μs. Most of time durations are in the same range on jUAV except a few spikes. According to our runtime profiling during experiments, the spikes seen in Figure 7b are attributed to the pause from garbage collection (GC). We envision that these GC pauses will be significantly reduced once we utilize a real-time JVM. Figure 8 shows the comparison of time durations for stabilization logic. The purple lines on Figure 8a and Figure 8b are time durations of stabilization logic on Paparazzi UAV and jUAV respectively. The time duration on jUAV is 30% to 40% higher than the time duration on Paparazzi UAV as we only port the stabilization related computation in the Java-based jUAV. It still needs to interface with the data structures in the native implementation via JNI. There are 6 getter/setter functions that are called in the Java-based stabilization logic. To show a pure computational comparison, we plot a green line on both Figure 8a and Figure 8b. It presents the time durations of the Java-based stabilization computation on jUAV



(a) Paparazzi UAV (b) jUAV

Figure 7: Comparison of Time Duration for Autopilot Logic



(a) Paparazzi UAV (b) jUAV

Figure 8: Comparison of Time Duration for Stabilization Logic

that excludes the time cost of JNI (*jUAV without JNI*). As Figure 8a shows, the time duration on Paparazzi UAV and *jUAV without JNI* show a pattern. Similar to the time duration of the entire autopilot logic, due to the GC pauses, there are a few spikes in the plot of *jUAV without JNI*. These can potentially be eliminated by the use of a real-time JVM.

In conclusion, the time comparisons show that our Java-based jUAV has a reasonable performance in time perspective. Although it suffers from traditional Java application performance issues—JNI and garbage collection, we believe these problems can be resolved, once we move to a real-time Java virtual machine, like FijiVM [17].

5. RELATED WORK

There have been lots of emerging projects that build UAV platforms for industrial products or academic research, but only a few works focus on implementing an autopilot application in a managed language. To the best of our knowledge, ScanEagle [2] is the first autopilot application written in Java. It is based on a real-time JVM with RTSJ compliance. Due to the fact that the source code of the ScanEagle project is not publicly accessible, it is difficult if not impossible to fully understand the implementation. Another notable Java UAV platform JAviator [6] offers a means for UAVs to be flown while controlling from the ground station. The current implementation does not offer self-contained autopilot that allows for predefined routes to be flown by the airframe. Instead, the airframe must be in constant contact with the Ground Station so that it can request the latest navigation data[9]. JAviator however, implements a flight control system (airframe stabilization) that runs on the airframe with four alternative controller implementations: 1) JControl, a Java-thread-based implementation; 2) EControl, An Exotask-based controller written in Java; 3) CControl, a Linux-process-based controller written in C; 4) TControl, a micro-kernel-based controller written in a real-time micro-kernel called Tiptoe. Though these variations of the flight control system exist in JAviator, the only variation found in its released code base is the CControl implementation [10], making the currently publicly available system a hybrid of Java and C. One of design benefits to the JAvaitor platform, is that it targets a platform which was developed tandem to the software base meaning that it need only worry about developing the code to connect to the hardware once. Paparazzi on the other hand, chose to make their platform usable by multiple air frames and due to the widespread use of the Paparazzi UAV, many airframes are already supported by the system. This means that any system that can integrate directly with Paparazzi UAV can utilize much of the existing hardware specific connector code. Another point of note, is that only the stabilization code is run on the airframe. Routing is determined by the ground station,

and communicated back to the airframe meaning that the airframe may not currently be flown independent of the ground-station. The JAviator projects real-time Java component is not facilitated by the standard RTSJ. Unlike the JAviator project jUAV aims to provide a open-source platform built to the RTSJ standard thereby creating a means for the community to benchmark real-time JVMs.

There are various micro-benchmarks in RTSJ [5, 7] however as pointed out by Kalibera *et al.* [12] they only test very specific components independent to the rest of the system and external stimuli. CDx has provided a more complicated collision detection and avoidance benchmark, this system introduces artificial air traffic and synthetic noise to stress the JVMs capabilities. Though CDx is a vast improvement on micro-benchmarks the authors acknowledge that their benchmark is rather simplistic and there is a need for more realistic real-time task based benchmarks [13].

6. CONCLUSION

The real-time Java community has identified a necessity for benchmarks that recreate the complexity seen in real world deployments. The current benchmarks used in testing RTSJ and SCJ, like jPapaBench, do represent a complex system and give a good sense of the performance of a given JVM. However, the workloads on these systems are still lacking the complexity seen in the physical world. The jUAV project seeks to re-create the complexity found in real deployments by introducing real world physics simulation on the aerial vehicle requiring the autopilot to respond as would be required on an actual UAV. This integration with full physics simulation will allow the real-time Java community to validate their design prior to testing on actual systems, where failure could potentially mean the loss of life or expensive equipment. In this work, we have outlined the development of jUAV in three phases. We have also shown results from the first phase of implementation which included portions of the autopilot. Results show that the simulations are similar to Paparazzi UAV, a well known autopilot and simulation framework that has been previously deployed on aircrafts. This gives us the confidence that we will be successful in creating a real-world benchmark that meets our goals of timeliness while being able to control a real-world UAV. Our code is open-source and available for everyone to use, detailed instructions needed to setup and run jUAV can be found in Appendix A.

References

[1] New paparazzi simulator. `http://wiki.paparazziuav.org/wiki/NPS`.

[2] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(1):5, 2007.

[3] P. Brisset, A. Drouin, M. Gorraz, P. Huard, and J. Tyler. The paparazzi solution. rapport technique, 2006.

[4] G. Cai, B. M. Chen, and T. H. Lee. Coordinate systems and transformations. In *Unmanned Rotorcraft Systems*, pages 23–34. Springer, 2011.

[5] A. Corsaro and D. C. Schmidt. Evaluating real-time java features and performance for real-time embedded systems. In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pages 90–100. IEEE, 2002.

[6] S. Craciunas, C. Kirsch, H. Röck, and R. Trummer. The javiator: A high-payload quadrotor uav with high-level programming capabilities. *Proc. GNC*, 2008.

[7] B. P. Doherty. A real-time benchmark for java™. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 35–46. ACM, 2007.

[8] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[9] `http://javiator.cs.uni-salzburg.at/system/javiator_software_system/ground_control_system.html`.

[10] javiator-github. `https://github.com/cksystemsgroup/JAviator`.

[11] JSR 302. Safety Critical Java Technology, 2007.

[12] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. Titzer, and J. Vitek. A family of real-time java benchmarks. *Concurrency and Computation: Practice and Experience*, 23(14):1679–1700, 2011.

[13] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. Cd x: a family of real-time java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 41–50. ACM, 2009.

[14] T. Kalibera, P. Parizek, M. Malohlava, and M. Schoeberl. Exhaustive testing of safety critical Java. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, 2010.

[15] T. Luukkonen. Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*, 2011.

[16] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. Papabench: a free real-time benchmark. In *OASIcs-OpenAccess Series in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[17] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

Appendices

A. USING jUAV

The current prototype of the jUAV code base consists of two repositories that need to be cloned:

- Modified Paparazzi UAV (https://github.com/adamczer/paparazzi-native-jni)
- jUAV (https://github.com/adamczer/pappa)

A.1 Building Paparazzi UAV and jUAV

The Modified Paparazzi UAV repository is a fork of Paparazzi UAV v5.8 stable. Directions on how to build this code can be found on the Paparazzi UAV wiki at http://wiki.paparazziuav.org/wiki/Installation.

Once the modified version of Paparazzi UAV has been built one can now create the required shared libraries for the jUAV JNI calls:
1) Execute the paparazzi.sh located in `PAPARAZZI_HOME`.
2) Select the `Quad_Lisa_M2` for the airframe and the NPS as the simulator.
3) Click the build button initiating the build process for the airframe.
4) Copy the created shared library from `PAPARAZZI_HOME/var/aircrafts/Quad_LisaM_2/nps/libpapa.so` to the jUAV sources `paparazzi-jni/libs/`.

With the shared library where it is required, one can now build jUAV using Apache Maven. Assuming Maven is installed one need only execute ”mvn install” in the root of the jUAV checkout.

A.2 Running jUAV

With the code now built, one can return to the Paparazzi UAV UI doing the following:
1) Select `Quad_LisaM_2` and NPS if not already selected.
2) Click Execute.
3) Click Stop on the ”Simulator”.
4) Change the ”Simulator” command by appending ”–java yes”
5) Click Execute

The Simulation will now be using the jUAV code base. The above instructions are subject to change, however if they do change the updated instructions will be found on the jUAV GitHub.