Toward Making jPapaBench Fly: An Experience Report

Shaun Cosgrove, Yin Yan, Sai Tummala, Manish Jain,
Karthik Dantu, Steven Y. Ko, Lukasz Ziarek
SUNY Buffalo
{shaunger, yinyan, saisekha, manishja, kdantu, stevko, lziarek}@buffalo.edu

ABSTRACT

jPapaBench is one of the most popular benchmarks used in the evaluation of real-time Java virtual machines. It is based off of the open-source paparazzi project, a commonly used autopilot system that has seen much real-world use. In this paper we detail our progress in adapting jPapaBench to take flight. We begin by comparing jPapaBench's performance to its C-based inspiration, paparazzi. For an apples to apples comparison we port paparazzi to a real-time OS kernel and we integrate paprazzi's visualization system to work with jPapaBench to test ground station communication. We detail our current system status and our work in progress to utilize the NPS simulator and real-world flight dynamics models with jPapaBench.

1. INTRODUCTION

The real-time Java community has adopted jPapaBench [4] [1] as a popular benchmark for testing and validating Java virtual machines (JVMs). Based on paparazzi [2], an autopilot for unmanned aerial vehicles (UAVs), jPapaBench is a complex benchmark consisting of many tasks encompassing the autopilot module, the fly-by-wire module, and the 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) [2] and safety critical Java (SCJ) [3]. It is an especially good test 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.

In this paper we detail our efforts in expanding jPapaBench to work with the paparazzi framework with the ultimate goal of creating a Java based UAV system for benchmarking JVMs on real-world UAV deployments, including bare metal and RTOS C baselines for comparison purposes. We believe this effort will augment jPapaBench to benchmark a VM's and real-time Java specification's I/O infrastructure. To this end we present the current state of our system, including the integration of the ground station and visualization components of paparazzi with jPapaBench, the port of paparazzi to the RTEMS RTOS kernel [3], and the start of the integration of the NPS flight simulator leveraging the JSBSim flight dynamics model. We also present initial results that illustrate the difference between paparazzi and jPapaBench.

[1]`http://code.google.com/p/jpapabench/`

[2]`http://wiki.paparazziuav.org`

2. BACKGROUND

This section overviews Paparazzi as well as jPapaBench, a Java benchmark based on Paparazzi.

2.1 Paparazzi UAV

Paparazzi UAV is an open-source software system for autonomous aircraft. It has an advanced and proven autopilot that handles all aircraft tasks such as navigation, flight control, sensor management, data links, etc. It supports various types of aircraft, including fixed-wing and multi-rotor craft. The project provides a rich set of tools to achieve autonomous flight and with the appropriate aircraft and wireless communication hardware, represents a complete system for autonomous aircraft configuration, control, and real-time monitoring and visualization [1].

The Paparazzi UAV suite can be roughly divided into two components, the embedded autopilot with associated hardware and the PC-based software.

- **Autopilot:** Each aircraft must consist of a set of sensors and an embedded processor or processors to run the autopilot logic. The sensors are used for altitude and position information, and typically consist of a GPS unit, 6 or 9 degree of freedom (DOF) inertial measurement units (IMU), but can also use various other sensors such as IR sensors and barometric pressure sensors etc.. For weight and size reduction, all of these components can be assembled together onto a single embedded board.
- **PC-based software:** The PC software suite is a combination of four main components, which are utilized to create the embedded deployment profile and configure the system as a whole. The first component is a configuration suite whereby each autopilot deployment is configured and generated through various xml files. This allows the creation of a custom deployment that is tailored to a specific platform's needs, such as GPS unit types, IMU units, flight plans, servo configurations. The second component is a server, which is used for logging, preprocessing and data distribution. The

[3]`http://rtems.org`

third component is a ground control station (GCS) for live monitoring, planning, and control of flights. The last component is the extensive support for aircraft system simulation, including a 6 degree of freedom, physics based flight dynamics model JSBSim and the NPS simulator.

2.2 jPapaBench

jPapaBench is a Java version of PapaBench, a real-time embedded benchmark derived from Paparazzi. There are three major components in jPapaBench—the autopilot, the fly-by-wire, and the simulator. The autopilot controls UAV flight based on a flight plan. The fly-by-wire (FBW) receives commands from a ground station and relays the information to the autopilot. These two modules are supposed to run on two different computation units on real hardware; however, jPapaBench runs them on a single JVM. The simulator computes a physical model about the UAV that the autopilot controls. It generates sensor data based on the model, which is then used by the autopilot.

Since jPapaBench uses Java and is not intended to run on UAVs, certain design points are different from PapaBench. First, jPapaBench supports explicit synchronization available in Java. In contrast, PapaBench uses timed scheduling. Second, jPapaBench uses simulated hardware, thus the interface to various hardware components is simplified. For example, sensors are simulated by the simulator and a hardware bus between the autopilot and the fly-by-wire is replaced by a software buffer.

2.3 Project Design Goals

This project is a portion of a much larger body of work to create a fully working Java autopilot running on real world hardware, as such we would like to outline some of the higher level designs for this project. Our initial design is for the autopilot code to run on a resource constrained development board and have full interaction with the NPS simulator running on an external PC. This allows us to measure the correctness of our embedded implementation against the purely PC based simulation. This requires creating a communication path and specific modifications to the NPS simulator to support a remote target. The Java version should not require further modification to the NPS simulator. If the Java version can fly with the same simulation as the C version then that will give confidence in its correctness and a transition to a UAV can follow.

3. IMPLEMENTATION EXPERIENCE

For comparison to the Java version, we must first create a version of the Paparazzi autopilot running on the same hardware platform that is based on the original C version of the code. This version must have extended support for communication channels to allow the timely transfer of the simulation data on top of the standard volumes of downlink data, extra support for various simulator messages and compatibility with the NPS simulator. In the following sections, we discuss the default target for the Paparazzi UAV autopilot which highlights several challenges in implemenation on the LEON3 development board, our modifications to the communications channel, the simulator and the autopilot code which was required for functional operation.

3.1 Ground Station Implementation

Software in the Loop (SITL) autopilot operation on the LEON3 development board was achieved through minor modifications to the Paparazzi ground station code and utilizing the software Ivy Bus when possible. A new autopilot target was created for the simulation whose purpose is to be a bridge between the remote autopilot and the PC based simulator. This involved publishing specific simulation messages the Ivy Bus and subscribing to certain messages to take commands from the autopilot. Another program acted as a router to take messages from the Ivy Bus and send them to the autopilot and vice versa. This program can be configured to utilize one of two communication channels to interface with the remote autopilot, a high speed UART and a TCP/IP socket connection.

As this design focuses on isolation of all of the components and utilizes the common Ivy Bus for communication, the major hurdle was to ensure the messages were formed correctly and passed appropriately. Once this was complete, the interface with the ground station was operational and to extend it, the only major work required was generation and support of new message types. This design pattern is our fourth iteration of the communication and simulation architecture and revisions were necessary due to changing our simulator from JSBSim to NPS and other hardware communication limitations. The current solution has sufficient bandwidth to allow streaming of 120Hz simulation messages which are required for autopilot operation and various ground station control and configuration messages reacquired for system operation.

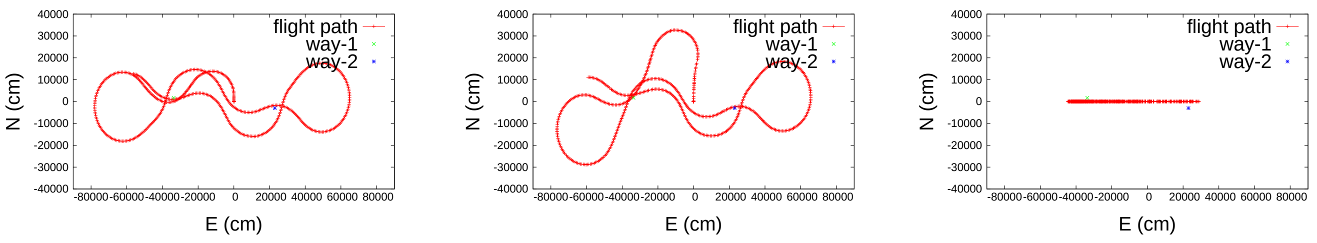
3.2 NPS Simulator

The Paparazzi NPS simulator is an advanced simulation platform that can simulate various sensors, environmental conditions, and also includes a FDM for flight physics. The various simulated data can be incorporated into a simulation to create a more realistic scenario of flight and autopilot operation. NPS can also be configured to send data directly into the autopilot from the simulator state which is a useful tool for debugging. The downside of the using the NPS simulator for our project is there is an increased volume of data required to perform the simulation which places an extra burden on the communications and message processing.

3.3 jPapaBench Integration

Getting the Java autopilot to run the LEON3 was a straightforward process. As the autopilot is a benchmark with a working autopilot computation, it does not generate any message that are compatible with the ground-station and its internal representation of its' data differs from our C based autopilot. So, all of the messages to be sent to the GCS had to be generated and the specific data values needed to be pulled from the autopilot and converted to the format and types expected by the ground station. However, our first challenge was to enable Ethernet transport through RTEMS for the Java code. After this was completed and we had a working communication channel from the Java autopilot to the pc based simulator we integrated a subset of the possible messages the autopilot can generate and began sending them to the ground station. This very quickly allowed the visualization of the Java autopilot, which can be seen in the graphs below, and confirmed that the simulation it is performing is not fully featured.

For transmission of our status and general messages, we created a JUAVReportingTaskHandler which has a period of 100ms and a priority of 19. Each time the task is handled, it updates a statemachine which returns the relevant messages that are ready for transmission. To transmit our command messages, which will be fed into the NPS simulator, we created a JUAVRadioControlTaskHandler which has a period of 25ms and a priority of 33. Each handler sends the messages over the TCP/IP communication channel. For reception of messages a JUAVSimulatorFlightModelTaskHandler



(a) C Simulator Flight Path (b) C UAV Flight Path (c) Java UAV Flight Path

Figure 1: UAV flight path on an X,Y GPS coordinate system.

was created. This was chosen as an ideal place to receive messages from the NPS simulator and insert them into the system. Its period is 25ms and its priority is 26. [4]

3.4 Creating C Baselines

Bare Metal. Paparazzi UAV targets several compact integrated embedded processor boards. The processors on these boards are typically ARM based processors and have limitations in their available system resources, such as MIPS, flash and system RAM. Due to these types of embedded targets, the code base makes has many optimizations for performance and reduction in program size that includes multiple layering of #define MACROS and transfer of certain messages through raw bytes as opposed to strings etc... These macros are systemic in the code and serve two main purposes, the first as mentioned above is performance related, the second is isolation and layering of various system components. For instance, all messages are generated in macros and are then passed through a transport layer to handle the specific message construction which then passes the message to an actual hardware communication channel. Each layer can be redirected by changing compile time definitions and clean isolation between each layer is possible.

Much of the code is generated in the preprocessor and the macros are resolved at compile time, everything is statically linked and a single binary with one thread of operation, which is typical for this class of embedded hardware, is created. This is then deployed on the target board and operates with a cyclic processing schedule.

Several challenges occur for our implementation as a result of the default target for the system. Compatibility with the our chosen target architecture is the first challenge as the arm based targets are not compatible with our development platform. Creating a bare metal implementation that runs natively on the sparc-instruction set and LEON3 CPU is a large and involved task. Instead, we decided to migrate the Paparazzi UAV autopilot to RTEMS.

RTEMS on LEON3. This version is based on the accumulated sources for a *Microjet LisaM* configuration with a custom flight plan. Once compiled for the simulator target, all of the automatically generated files and the other required source code were taken and combined into a standalone project. Our goal was to have fully working autopilot that existed as an application on top of RTEMS. The autopilot application, where appropriate was linked into the underlying hardware through our defined layering which was designed to be integrated neatly into the Paparazzi autopilot code. Substantial effort was required to get an initial prototype implementation running. Most notably, much work was placed into the communication channels, ensuring appropriate processing of messages while maintaining the scheduling of the autopilot tasks, transitioning the simulator from the JSBSim to NPS simulator, and finally a fully integrated TCP/IP communication channel to allow for the streaming of large volumes of communication data while still enabling the autopilot's cyclic scheduling to operate effectively.

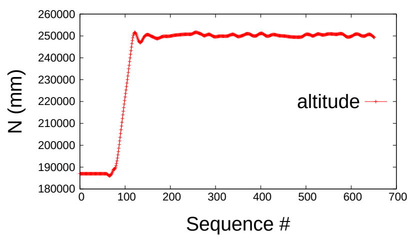
The main challenges for the C implementation related to the single thread cyclic nature of the scheduling and the nature of the communication handling which is designed and optimized for data reception at low rates. Increasing the speed of the UART beyond a certain point, with high volumes of incoming data, causes blocking in the main loop. Transitioning to the Ethernet communication broke the expected way that data was introduced to the autopilot, single bytes versus blocks of 100s of bytes of data from Ethernet packets, and required careful integration before a correct balance between processing of incoming data and the associated overhead was obtained. Our resulting C application is capable of handling large volumes of streaming simulation data and both the Ethernet and UART versions fly with the NPS simulator.

Due to the barriers to development mentioned above, it was not feasible to generate a threaded RTEMS version of the autopilot logic, also it was not possible to port only the application layer as it is strongly coupled to transport layer. We chose instead to port the entire Paparazzi code while trying to maintain as much of the original code and functionality as possible, including the single thread of operation and cyclic scheduling. This was done by creating the interfaces to the hardware specific components where required, such as the system timer functionality and the hardware communication channel. Paparazzi operates as if it were running on an embedded processor while in fact it is actually running as a task on a hard real-time OS.

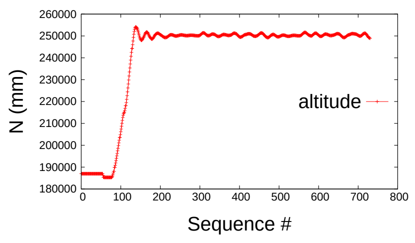
4. EVALUATION AND DISCUSSION

To evaluate the flightworthiness of jPapaBench we have conducted a series of experiments comparing the Java-based UAV system to the C-based UAV system. Since UAVs typically have a small ca-

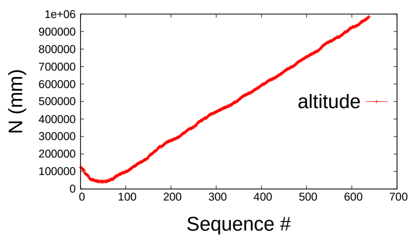
[4]We have bi-directional Ethernet communication in operation but its effect is not evaluated in this paper



(a) C Simulator Altitude



(b) C UAV Altitude



(c) Java UAV Altitude

Figure 2: Altitude of the UAV over time.

pacity in terms of a physical load they can carry, we use a resource constrained embedded board for the experiments. We use a GR-XC6S-LX75 LEON3 development board running RTEMS version 4.9.6. The board's Xilinx Spartan 6 Family FPGA is flashed with a modified LEON3 configuration running at 50Mhz. The development board has an 8MB flash PROM and 128MB of PC133 SDRAM. To execute the simulator we use an AMD A10-5800k APU 8 GB RAM running Ubuntu 14.04 LTS, which is based on Linux 3.13.0-24-generic x84_64 kernel. We use the Fiji VM [5] and the RTSJ version of jPapaBench in our experiments.

On these two platforms, we use three system level configurations: (1) *C Simulator*: a software simulator of the C autopilot leveraging the NPS simulator executing on a PC, (2) *C UAV*: the C autopilot executing on the LEON3 with simulated sensors and actuators, and (3) *Java UAV*: executing on the LEON3 with simulated sensors and actuators. For the Java UAV configuration, the data generated by the NPS simulator is sent and parsed. However, the precomputed values in the benchmark are used instead of the dynamic generated values from the NPS simulator due to the limitations described earlier.

Fig. 1 shows the comparison of the flight path for all system level configurations. The flight plan is configured to travel between two way-points and the UAV is to first launch from its start position, climb in altitude and then navigate to the first way-point and then to the second. The UAV starts on the ground, launches at a set velocity, proceeds to reach its flight altitude, while flying toward the first way-point. The change in altitude over time is given in Fig. 2. The coordinates and altitude are derived from the messages the UAV sends to the paparazzi base station during flight. Each flight path graph is offset so 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 choose to represent in the graph but for simplicity, they can be viewed as distance from the initial position in X and Y coordinates. Both the C Simulator and C UAV system configurations make dynamic adjustments in their flight path until they precisely reach the way-point. The flowing nature of the graphs represent the interaction of the simulated physical plane, the physics of the planes motion in atmosphere and the autopilot trying to meet its targets.

References

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

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

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

[4] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin 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.

[5] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 69–82, New York, NY, USA, 2010. ACM.