Hard Real-Time General-Purpose Robotic Simulations of Autonomous Air Vehicles

Shawn Michael Walker *, Jinjun Shan †, and Robert S. Allison ‡
York University, Toronto, Ontario, Canada, M3J 1P3

High-fidelity *general-purpose robotic simulators* are a special class of simulator designed to simulate all the components of a real-world robotics system, including autonomous air vehicles and planetary exploration rovers, so that a real-world system can be tested and verified before/during deployment on the real-world hardware. General-purpose robotic simulators can simulate sensors, actuators, obstacles, terrains, environments, physics, lighting, fluids, and air particles, while also providing a means to verify the system's autonomous algorithms by using the simulated vehicle in place of the real-world one. General-purpose robotic simulators are typically coupled with an abstract robotic control interface so that autonomous systems evaluated on the simulated vehicles can be deployed, unchanged, on the corresponding real-world vehicles and vice versa. However, the problem with the current technology and research is that neither the robotic simulators nor the robotic control interfaces support *Hard Real-Time* capabilities, and cannot guarantee that Hard Real-Time constraints will be met. The lack of Hard Real-Time support has major implications on both the utility and the validity of the simulation results and the functioning of the real-world autonomous vehicle. As a solution, this paper will present Hard-RTSim, a novel hard real-time simulation framework that will: 1) Bring Hard Real-Time support to general-purpose robotic simulators; and 2) Bring Hard Real-Time support to abstract robotic control interfaces. Hard-RTSim guarantees that simulated events in the environment or modeled vehicle are produced and handled with finite (bounded) accuracy and precision. Furthermore it improves these temporal responses to ensure these bounds are representative of temporal requirements for a wide range of scenarios. The Hard-RTSim framework ensures that the simulator and the hard real-time processes will actually get to use the CPU when they request/need it, no matter how many other processes are loaded on the CPU. The experimental results of using the Hard-RTSim framework compared to not using it yield a huge improvement in responsiveness and reliability. There is an improvement of 35% when the CPU is minimally loaded and then as the CPU load is increased the improvement increases as well, all the way up to a 98% improvement when the CPU is loaded at its maximum. These substantial improvements in precision and reliability will help to further the state of space exploration, aerospace technology, and produce better and more reliable autonomous aerial vehicles and planetary exploration rovers.

I. Introduction

A. General-purpose robotic simulators

In its most general sense a *robot* is any kind of mechanical or virtual artificial agent controlled by a computer program; for example, humanoid robots, satellites, robotic arms, planetary exploration rovers, and autonomous air vehicles can all be considered robots. A *robotic simulator* allows for robotic hardware to be simulated on a computer and provides an easy, safe, and cost-effective platform for developing and verifying robotic systems without having to rely on the corresponding real-world hardware. Robotic simulators coupled

*PhD Candidate, Petrie 025, Department of Earth and Space Science, 4700 Keele Street, shawnmichaelwalker@hotmail.com. Student Member AIAA.

†Associate Professor, Petrie 255, Department of Earth and Space Science, 4700 Keele Street, jjshan@yorku.ca. Senior Member AIAA.

‡Associate Professor, CSB 3051, Department of Electrical Engineering & Computer Science, 4700 Keele Street, allison@cse.yorku.ca.

with an *abstract robotic control interface* give the ability to test autonomous algorithms using the simulated hardware in place of the real-world hardware by providing a uniform interface so that the autonomous system does not know whether the hardware is real or simulated; for example an autonomous control system can be developed and tested on a simulated unmanned air vehicle and then, when complete, it can be immediately deployed on the corresponding real-world version of the unmanned air vehicle. There are many types of robotic simulators for many different applications; some examples are robotic arm simulators, industrial automotive simulators, and general-purpose robotic simulators.

A *general-purpose robotic simulator* is designed to simulate everything inherent in a real-world robotics experiment, such as all types of hardware, sensor data, and environments. Examples of the hardware that can be simulated are motors, joints, actuators, moving and rotating parts, and sensors such as cameras, lidar, inertial measurement units, and global positioning units. In addition to simulating the hardware, the sensor data and physics associated with the hardware must be simulated realistically too; some examples of realistic sensor data are the imaging data coming from the simulated cameras, and the range data produced by the simulated lidar; while some examples of the associated physics are realistic movements by the actuators and moving parts, realistic error distributions, realistic interactions among objects, and realistic gravity and friction effects. General-purpose robotic simulators are also designed to simulate a varying range of environments and can simulate many types of terrain, objects, obstacles, textures, and even wind currents and liquids.

Another way to articulate the difference between a *general-purpose robotic simulator* and any other type of simulator is by a reverse-thinking approach. Imagine a real-world autonomous aerial vehicle such as a quad-copter that uses a camera for computer-vision based navigation and is running a specialized artificially intelligent control system to carry-out some task. Now imagine that you wanted to test out the very same specialized artificially intelligent control system (designed for real-world hardware) in a simulator without having to change any of the code or control system. This is where a general-purpose robotic simulator comes into play.

It is important to note that the simulator and the visualizations of the simulation are two different things. It is possible and maybe even typical for the visuals to look simple yet have an extremely powerful and precise simulator underneath, and vice-versa the visuals can look very realistic and have a lot of definition yet have an underlying simple and basic simulator. Nonetheless the visuals displayed to the user are very important because they allow human intuition and subconscious problem solving to be engaged. When humans run a real-world experiment they visually watch the experiment unfold because they can see things that might otherwise be unnoticed when analyzing the pure raw data afterwards, and similarly this concept applies when the experiment is attempted using a simulator; that is, it is important to display accurate visuals so that the researcher can see with their own two eyes what is happening.

This paper uses the 2.5D Player/Stage general-purpose robotic simulator and abstract robotic control interface for all of its experiments because they are lightweight yet powerful, which make it well-suited to demonstrating our solutions to the underlying problems inherent in all robotic simulators and robotic control interfaces that are run on general-purpose computers. However it is important to note that the problems caused by the lack of hard real-time support are also found in other simulators and abstract robotic control interfaces such as ROS and Gazebo, and adaptations to these systems will be pursued later by the authors. The research and academic literature concerning the Player/Stage general-purpose robotic simulator is detailed in Ref. 1–8. Some of the main competitors to Player/Stage are Webots, Easy-Rob, Simbad, USARSim, Microsoft Robotics Studio, and MissionLab, which are detailed and compared in Ref. 9,10. The main attraction of Player/Stage over the other simulators is that Player/Stage is free, open-source, powerful yet simple to use, lightweight, and is still being actively developed.

B. Abstract robotic control interfaces

An abstract robotic control interface such as *Player* is a special middle layer that provides an interface between an artificially intelligent autonomous system and its sensor and actuator hardware such as motors, cameras, and other components. Having an abstract interface is very important when using a general-purpose robotics simulator because then the control system program does not need to be changed when transferring it to the real-world system. The only thing that must be changed is the source of the input data from the simulated device to the real-world device. In this type of scenario the only element of the system that is changing is the actual device that the Player device is associated with.

For example a camera may provide a control program with an image that has a pixel resolution of

800x600 where each pixel is represented by a 24-bit RGB value. Whether this image is coming from a simulated-camera within a simulated environment or a real-camera in a real-world environment is not a major point of contention for the operation of the underlying artificially intelligent autonomous program. The only difference between a real-world image and a simulated version of the same image is the accuracy of the simulated representation. The accuracy of the simulated data will depend on how accurate the robotic simulator actually is, and this is why it is important for continued research into increasing the fidelity and capabilities of general-purpose robotic simulators. The communication topology and interaction of Player with the real-world and simulated vehicles is shown in Fig. 1.

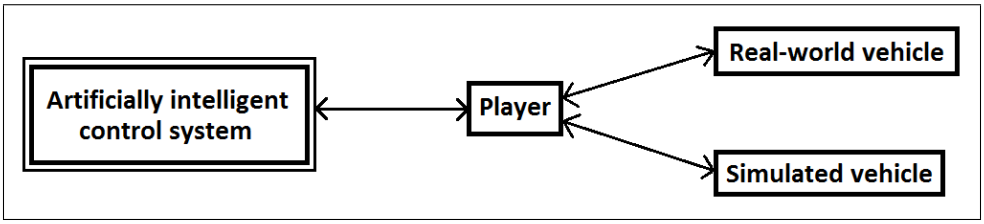

Figure 1: Overview of the communication topology of Player with real-world and simulated vehicles. An abstract robotic control interface such as Player allows a robotic system to be developed that can instantly be deployed on either a real-world system, a simulated system, or both simultaneously.

C. Hard real-time systems

A *hard real-time* system is one that has absolute timing deadlines that must be met and run-times must be deterministic and guaranteed, including the worst-case execution scenario. An example of a system with hard real-time constraints is a control loop for an unmanned aerial vehicle that reads in sensor data from cameras, performs some calculations, and then outputs new signals to the actuators, and perhaps it must do this precisely every 1.0 seconds, exactly on the second, and must complete within 0.2 seconds. Furthermore, if the system in this hypothetical example misses any of these deadlines then perhaps the vehicle will have an unrecoverable crash. One of the themes of hard real-time systems is that if such a system misses a deadline then the outcome could be catastrophic and non-recoverable, and so there must be a 100% guarantee that every single timing constraint will be met under every possible operating scenario. Hard real-time systems are well discussed in Ref. 13, which also discusses execution times, timing anomalies, and many other concepts.

D. Overview of problems to be solved

There are two distinct problems that this paper will solve using a single solution. The cause of the two problems are similar however it is best that they are addressed as separate issues.

1. Problem: Lack of hard real-time support in general-purpose robotic simulators

General-purpose robotic simulators need to run off a clock in order to simulate events that depend on time, such as an autonomous air vehicle moving with a given velocity. The simulator can either run off of a *simulated clock* or it can run off of a *real-world clock* such as the CPU's internal clock; although there is problems with both methods. A simulated clock allows the robotic simulator to take fixed time-step's of sequential time units at its own pace, whenever it is given the CPU. Thus the simulator keeps track of its own internal time that has no definite relation to real time. In contrast, if the robotic simulator uses a real-world clock then whenever it gets the CPU it calculates how much time has elapsed in real-world time since it last took a simulation time-step and then uses this value in its calculations and projections, so in this case the system maintains correspondence with real time but simulation step sizes vary. Using a simulated clock is problematic for systems destined for the real-world because such systems will ultimately have to use a real-world clock. If the robotic control system is only being used with the simulator and will never be used on a real-world system then using the simulated clock might be sufficient in some scenarios.

The robotic control system itself will also require a clock and its clock should be synced with the same clock as the robotic simulator, whether its using a real-world clock or simulated clock. As mentioned previously, the main advantage of utilizing general-purpose robotic simulators is so that a robotic control system can

be developed and tested using the simulator as an intermediary step and then deployed on the real-world system. Introducing actual control hardware into the simulated environment or using simulated controllers in the real world environment necessarily confronts us with the requirement to consider real world time. A real-world system cannot use a simulated clock because a real-world system is bounded by the phenomenon of time and has a temporal existence. This means that a real-world robotics system must use a real-world clock, which means that the simulated system must also use a real-world clock, otherwise the developed system cannot be seamlessly deployed on a real-world robot or UAV.

Having the general-purpose robotic simulator and the robotic control system synchronized on a real-world clock can cause anomalies due to the way the CPU schedules processes. If a robotic control system requires fresh data every 1.0 seconds, exactly on the second (0.0s, 1.0s, 2.0s, 3.0s, ...), then the simulator would itself need to be able to calculate a simulation step once every second as well, in order to match the requirements of the control system and mimic what would happen in the real-world. However when using a multi-tasked computer system that is not hard real-time there are no guarantees that the simulator will actually get the CPU when it needs it, and thus it cannot perform/calculate the next simulation step. This means that the simulator will be calculating its simulation steps at indeterminate points in time and the steps themselves will take an indeterminate amount of time to complete. So the data the robotic control system obtains from the simulator would not be *temporally* accurate and theoretically the timing error would be *unbounded*.

These indeterminacies can cause anomalies within the overall system. These occur because the control program is most likely expecting sequentially-ordered consistent data that is temporally accurate at least to some degree, especially if the control program is destined for deployment on a real-world system. For example, if the control program polls a real-world camera for an image every second, it should get the image at the expected interval[a] (assuming the camera hardware is functioning correctly), however if this exact same control program were polling a simulated camera instead (which is possible due to the use of abstract robotic control interfaces such as Player) then the actual image it receives from the robotic simulator could be dated due to the fact that the simulator may have never actually taken a time-step because it was not scheduled since the last update. This anomaly/error had nothing to do with the control program or the camera but everything to do with the simulator's indeterminate timing. When debugging the control system this type of error might be nearly impossible to identify because if you ran the same program a second time then the issue could disappear due to random chance, as will be shown in the next section. Even worse is that it may cause further complications as the control system developers attempt to rectify the problem by mistaking it as a problem with the controller rather than the CPU timing.

One goal of the research into general-purpose robotic simulation technology is for it to have a high-degree of accuracy and to mimic the real-world as closely as possible, including the timing of events and the temporal accuracy of data. The presence of timing anomalies in simulated systems that are not present in their corresponding real-world systems invalidates the results of the simulation and greatly reduces its reliability.

2. ***Problem: Lack of hard real-time support in abstract robotic control interfaces***

As previously discussed an *abstract robotic control interface* such as Player or ROS (although ROS is much more then just an abstract robotic control interface) is basically a middle layer between a control program and the target hardware devices (whether real-world or simulated). It allows a control program to interact with an abstracted version of the target hardware device so that the control program does not have to worry about lower-level details and device drivers.

To elaborate on the example given earlier, perhaps a control program acquires an image from a camera device and then computes its local position using computer vision techniques. Utilizing the abstract robotic control interface, the control system can first be tested using the robotic simulator's camera and vehicle, and then subsequently deployed and tested with the real-world camera and vehicle, and not have to change the control system at all in between. This is accomplished by having the control program interact with an abstracted *generalized camera* and then have the generalized camera interact with either the real-world camera or a simulated camera using Player, as depicted in Fig. 2. In this type of scenario the parameters of the obtained image would need to be the same (such as resolution, file-type, etc).

The problem is that the abstract robotic control interface process is indeterminately scheduled by the CPU. So just because a control program requests or sends a particular piece of data does not mean that it

[a]Although it depends on the actual interfacing system being used; see the following section Lack of Hard Real-Time Support in Abstract Robotic Control Interfaces.

Figure 2: An example showing the Player abstraction of a simulated and real-world camera device as well as the associated communication topology.

will *actually* get or send the data when it wants/needs to. It will most likely get the data at some point, but the interval of time that it takes to get the data will be different every time, and as mentioned in the previous section the error is theoretically unbounded. Sometimes the only real-world solution is to compute average response times and use these values, however a mission-critical system with strict timing constraints cannot rely on this method, especially if a missed deadline results in an unrecoverable and catastrophic failure. Being able to rely 100% on guaranteed deterministic timing deadlines is integral for such a system and this is why hard real-time capabilities are needed for abstract robotic control interfaces as well as the simulators themselves.

E. Configuration of experimental system used throughout this paper

To keep things consistent, this paper uses the exact same experimental setup throughout every test and experiment that is discussed. The focus of this paper is in relation to what is known as a general-purpose operating system (such as Windows, Ubuntu Linux, OS X) on a general-purpose computer (the average mass-produced personal computer used worldwide); this is in contrast to specialized computers used in embedded systems or custom designed expensive computer hardware used for very specific applications. General-purpose computers and operating systems such as Ubuntu Linux give a huge advantage to the researcher due to the massive wealth of open-source, closed-source, and commercially produced software and libraries available, as well as many other software resources available online, most of which are designed for the typical cost-effective average general-purpose computer running a general-purpose operating system. As previously discussed, one of the drawbacks with the typical average computer is the lack of hard real-time support, as well as the lack of hard real-time support in particular projects and tools, such as robotic simulators and abstract robotic control interfaces.

The experimental system used throughout this paper is a 64-bit HP Pavilion dv3510nr laptop, with a 2.0 GHz Intel Core 2 Duo P7350 CPU, 2GB DDR2 SDRAM, 512MB Nvidia GeForce 9300M GS graphics card, and running Ubuntu Linux 11.04 Natty Narwhal (Linux kernel version 2.6.38), with Player 3.0.2 and Stage 4.1.0.

II. Motivating Example

A. Indeterminate Results Using the General-Purpose Stage Simulator

This example shows a simple autonomous aerial vehicle simulation using Player/Stage in Ubuntu Linux. This simulation does not have hard real-time support and because of this there are timing anomalies which affect the quality of the results. These failures highlight the fact that hard real-time support is an important consideration for general-purpose robotic simulation technologies, especially those used for mission-critical systems that cannot tolerate any timing failures.

In this example there is an autonomous aerial vehicle with its center of mass beginning at position (0,0) and a 1-unit by 1-unit obstacle centered at position (2,0). The autonomous aerial vehicle is set to move forward with a speed of 1.0 units/second and it is trying to avoid crashing into the object by stopping as soon as it detects the object within a certain threshold distance. It does this by constantly polling its sensor in a cyclic fashion to see if it detects an object within the threshold distance. The initial setup is shown in Fig. 3 and the pseudo-code for the autonomous aerial vehicle is displayed in Fig. 4.

For this demonstration the CPU is not loaded with any additional processes aside from the bare essentials required by the operating system. The simulation is run a number of times, without changing anything in

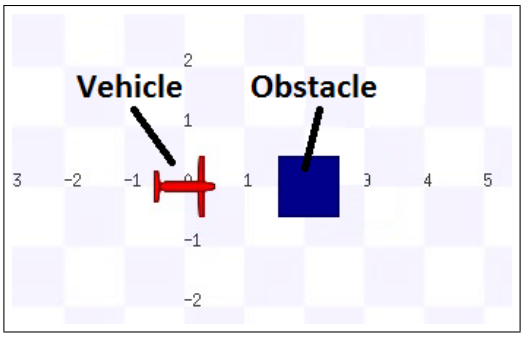

Figure 3: Initial setup of the simulated autonomous air vehicle and the object in its direct path.

```
Mission-critical control loop of UAV

set vehicle velocity to 1 unit/second

LOOP-forever

    sleep for 1 second
    read sensor data

    if object detected
        then stop vehicle
    else
        do nothing

END-LOOP
```

Figure 4: Pseudo-code showing the mission-critical control-loop code running on the unmanned aerial vehicle. The UAV is set to fly straight while continually checking its range-detector sensor to see if there is an object it can detect.

between. Sometimes the vehicle was able to detect the object and stop in time and sometimes it crashed into it. The results from Trial #1 when the autonomous vehicle crashed into the object are shown in Fig. 5 and the results from Trial #2 when the autonomous vehicle was able to stop in time are showed in Fig. 6. The reason for this random result is due to the way a typical general-purpose operating system schedules processes in that there is no guarantee that the control-loop process will get to use the CPU when it requests it. On some trials the simulator got the CPU near the expected time and sometimes it did not. There is no way to predict or determine which situation will occur as it is completely indeterminate due to the nature of how computer processes are scheduled when there is no hard real-time support.

Looking at the results more closely (the right portions of Fig. 5 and Fig. 6) it becomes evident why sometimes the autonomous vehicle crashes and sometimes it can detect the object in time. The amount of time that the cyclic step takes varies by a significant amount every trial, and this ultimately affects the distance traveled and whether or not the autonomous vehicle will detect the object in time. For some seemingly unknown reason the control-loop time for Trial #1 took 9.32% longer then Trial #2. This extra delay is what caused the unrecoverable and fatal error that resulted in the autonomous vehicles crash. This delay is due to the indeterminate nature of the CPU process scheduler, and a solution will be provided.

Additionally it would not be a reliable solution to simply slow the speed of the vehicle, or to increase the frequency of the control-loop cycle because the whole problem is that there is always the chance that the control-loop cycle will take too long. It is impossible to put a bound on the control-loop cycle with a 100% guarantee that it will not be violated, and this is because the process scheduler is indeterminate.

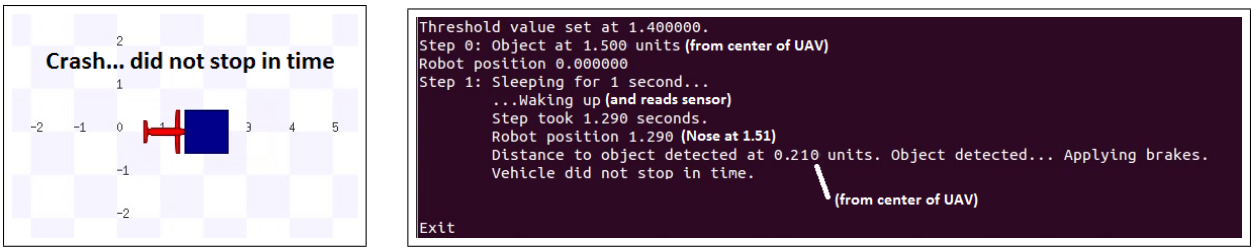

Figure 5: Trial #1: Autonomous vehicle does not detect the object in time to stop and so it crashes into it. In the figures, *Position* is defined as the center of the UAV, the UAV is moving at a speed of 1 unit per second, and the nose of the UAV extends 0.22 units from the center.

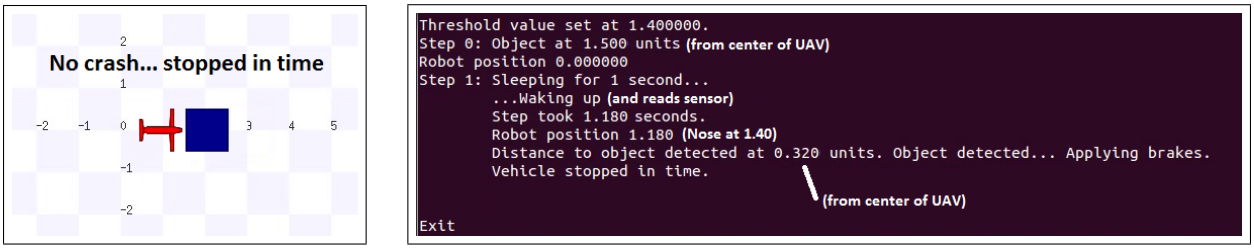

Figure 6: Trial #2: Autonomous vehicle successfully detects the object in time and is able to stop before hitting it. In the figures, *Position* is defined as the center of the UAV, the UAV is moving at a speed of 1 unit per second, and the nose of the UAV extends 0.22 units from the center.

B. A closer look at the motivating examples timing anomaly

As discussed in Ref. 13, it is nearly impossible to determine the actual execution time of any segment of computer code, especially from just the source code alone. The best bet is an experimental approach by taking an extremely large number of runtime measurements so as to have a somewhat statistically significant result. Since Linux kernel version 2.6.23 the CPU scheduler is the Completely Fair Scheduler (CFS), which is relatively fair to all running processes although there is no guarantee that a particular process will get the CPU when it requests it.

Similar to the experiment already shown in the motivating example, another experiment is run again except this time there is a certain number of low-priority processes running (in addition to the basic operating system processes) so as to increase the load on the CPU. Fig. 7 shows the pseudo-code of the low-priority process, which has a lower priority then the UAV's mission-critical control loop process.

The UAV mission-critical control-loop, as shown in Fig. 4, is run again multiple times with an increasing number of low-priority processes run in parallel so as to increase the load on the CPU. It is important to note that nothing else has changed in the experimental system; everything is the exact same aside from the number of low-priority processes being run at the same time. Each time the CPU load is increased the mission-critical control-loop from Fig. 4 is allowed to execute 10,000 times. The results of these many experiments are summarized in Fig. 8.

Each of the 10,000 executions of the mission-critical control-loop are run with a constant unchanging number of low-priority processes so as to yield a percent-utilization of the CPU. For each of these 10,000 runs there are three metrics obtained, the time of longest control-loop cycle, the time of the shortest control-loop cycle, and the overall average time of the 10,000 control-loop cycles. The smallest time range between the longest and shortest control-loop cycle was when the CPU was loaded at nearly 0% with a value of 0.040 seconds (longest - shortest: 1.219 seconds - 1.179 seconds), while the biggest time range being when the CPU was loaded at 100% with a value of 1.443 seconds (longest - shortest: 2.541 seconds - 1.098 seconds).

As can be seen in Fig. 8, as the percent utilization of the load on the CPU increases, the longest control-loop cycle increases, the overall average control-loop cycle increases, and very interestingly, the shortest control-loop cycle decreases. It is also very important to note that no matter what percent the CPU is loaded with all three of these metrics are actually unbounded theoretically, and this is due to the indeterminate nature of the CFS scheduler. These results imply that it would not be a good idea to use this type of system

```
Low-priority process to load CPU

LOOP-forever

    double x = 500
    double y = 3.33
    double z = x/y

    print x
    print y
    print z

END-LOOP
```

Figure 7: Pseudo-code showing the low-priority process used to increase the load on the CPU.

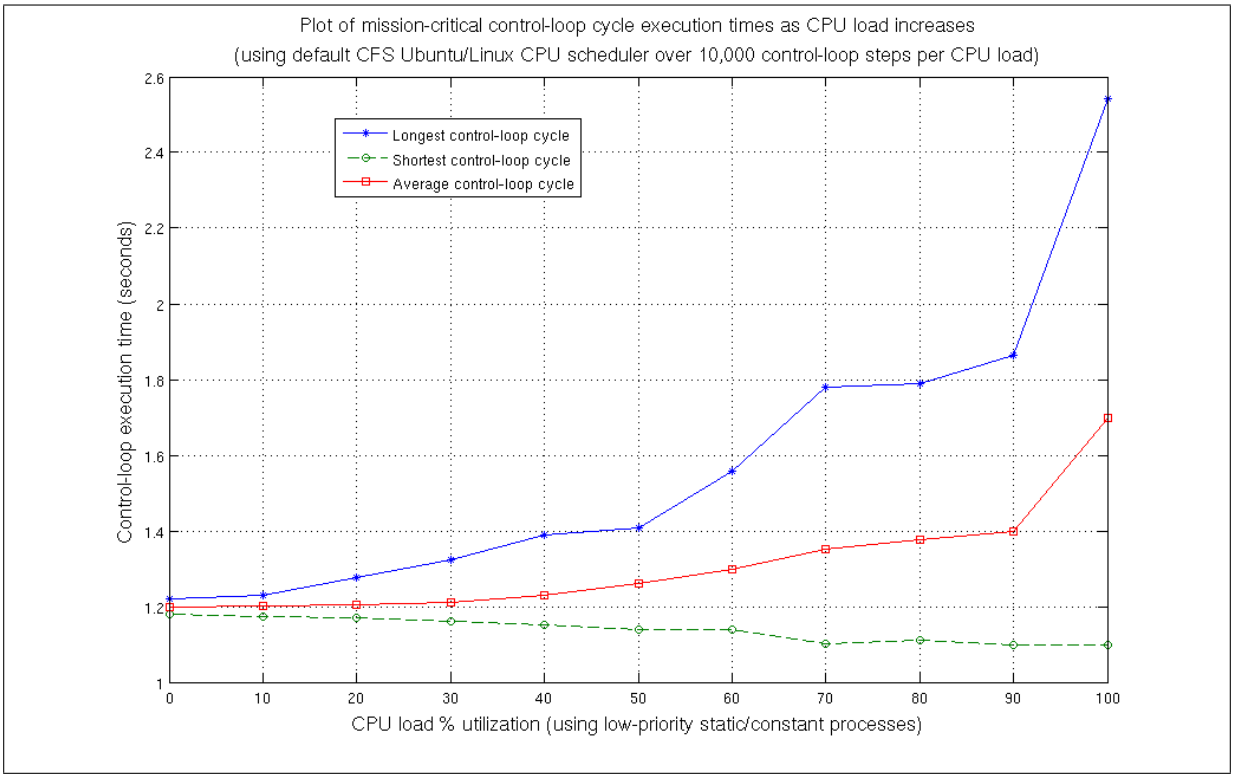

Figure 8: Plot showing the execution time of the mission-critical control loop against percent utilization of CPU using the Linux kernel's CFS process scheduler. The CPU load is achieved using multiples of the low-priority processes detailed in Fig. 7.

for a critical mission that absolutely cannot miss its timing deadline, and a solution will be presented in the next section.

III. Hard-RTSim: A Hard Real-Time Framework for Player/Stage

Most hard real-time software systems are realized using embedded computer systems with specialized hardware, specialized operating systems, and specialized programming languages. These systems are often

very basic and simple in design, and are extremely reliable for hard real-time systems. However one thing they lack is the flexibility and power that comes from traditional normal everyday computer systems that use normal general-purpose operating systems, such as Ubuntu Linux. Traditional general-purpose computer systems have the ability to use any programming language and interface with any kind of hardware, while taking advantage of all the open-source software libraries already available throughout the academic literature and the internet as well as many closed-source and proprietary software packages and programs. The ultimate desire then is to combine the two concepts to realize a hard real-time system using a traditional general-purpose computer system, and not having to use any kind of specialized hardware. It would then be possible to design powerful hard real-time autonomous systems in simulation as well as in the real-world.

This section will present the open-source Hard-RTSim framework as a preliminary solution to bring hard real-time support to general-purpose robotic simulators and abstract robotic control interfaces for real-world and simulated autonomous aerial vehicles. Hard-RTSim and an accompanying tutorial are all available for free download at Ref. 14.

A. Hard Real-Time Operating System Kernel

The first step in the solution presented by this paper to achieve hard real-time support within a traditional general-purpose computer system is to bring full preemption to the operating system kernel. This paper will use Linux as a case study, although the underlying theory can be used with any operating system. As discussed in Ref. 15, the standard Linux kernel has no guarantees for hard timing deadlines. By compiling the *RT-Preempt Patch* into the kernel it will gain hard real-time capabilities and convert Linux into a fully preemptible operating system. This means that every process can be assigned a priority and whichever process is ready and has the highest priority gets to use the CPU no matter what, even if another process is still using it. This contrasts with traditional computer systems where the processes are usually scheduled based on maximizing the efficiency and throughput. On these traditional systems a process that needs the CPU may have to wait a tiny bit if it means that it will be more optimal in the long run for the entire system. However the situation is vastly different for a system with hard real-time constraints because if a single process misses a single timing deadline then the result would be catastrophic and the original system would not longer exist. As detailed in Ref. 15, the RT-Preempt Patch makes in-kernel locking-primitives preemptive, makes critical sections protected by spinlock_t and rwlock_t preemptible, implements priority inheritance for in-kernel mutexes, spinlocks, and rw_semaphores, converts interrupt handlers into preemptible kernel threads, and converts the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts which leads to userspace POSIX timers with high resolution. In addition to the RT-Preempt Patch in Ref. 15, other options for hard real-time Linux are RTAI as detailed in Ref. 16, and Xenomai detailed in Ref. 17. More details about how to actually realize hard real-time Linux and get it working can be found in Ref. 14.

B. Dividing the System

This step requires the precondition that the target system must have more then one CPU so that the operating systems processes can be allowed to have at least one CPU to themselves and that the hard real-time processes also have at least one CPU to themselves as well. The reason for this separation is because intermixing the standard Linux processes with the hard real-time processes can cause the system to become unstable and crash, such as when a hard real-time process interferes or completely blocks standard processes like the mouse movement, interactions with the user, and other processes related to the graphics. This also makes it impossible for any of the other processes to interfere with the CPU being used by the hard real-time processes. Overall it makes the system clearer, simpler, less error prone, and most importantly it keeps the operating system stable which keeps the entire system stable. It is important to note that if the operating system crashes or freezes then this will affect the hard real-time processes. Please see Ref. 14 for detailed information about how to realize this on a Linux system.

C. Design of the Hard-RTSim framework

Once the underlying system is properly configured with preemption and divided the next step is to design an appropriate CPU schedule for the hard real-time processes, and then implement this schedule using a programming language. There has to be a delicate balance between all the involved hard real-time processes

and there are special novel programming designs in order to achieve a system that will actually work. The frameworks source-code and design details can be found at Ref. 14.

It is necessary to experimentally determine the worst-case execution time of the mission-critical control-loop process as well as the simulator execution time to calculate one full simulation step. If the mission-critical control-loop is being deployed on a real-world system then the worst-case execution time of the sensor reads and actuator commands must be determined as well. For example if the control-loop is grabbing an image from a camera that is not hard real-time then this error must be factored in.

D. Problems with the Hard-RTSim framework

There are still some minor problems with making a general-purpose operating system fully preemptible that the research community is currently working on. The problem is that there still may be a few small worker kernel threads running on the CPU designated for the hard real-time processes depending on which operating system is being used and on how it is made to be fully preemptible.

IV. Results of using the Hard-RTSim framework

Using the Hard-RTSim framework and re-running the same mission-critical control-loop process on the simulated UAV along with the low-priority processes to load the CPU, as previously demonstrated in the motivating example and outlined in Fig. 3, Fig. 4, Fig. 5, Fig. 6, Fig. 7, and Fig. 8, yields the results in Fig. 9.

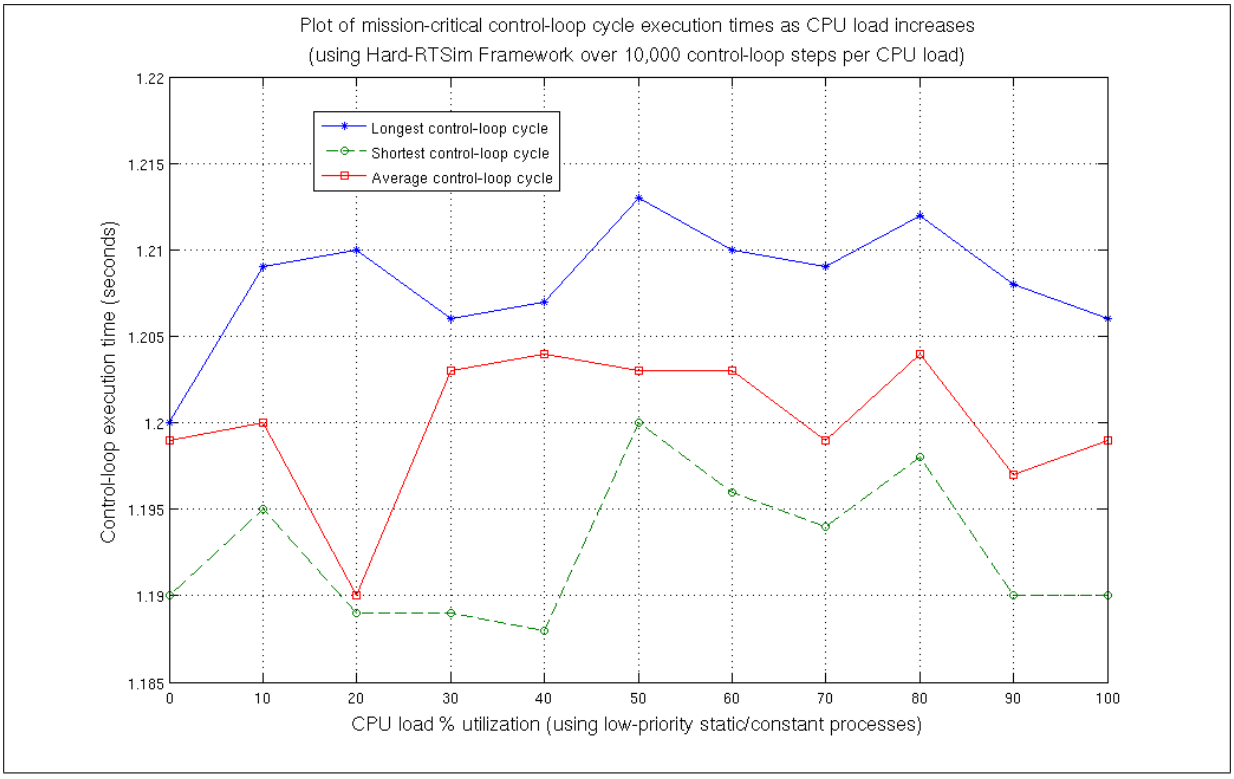

Figure 9: Plot showing the execution time of the mission-critical control loop against percent utilization of CPU using the Hard-RTSim framework solution presented in this paper. The CPU is load is achieved using multiples of the low-priority processes detailed in Fig. 7.

Fig. 9 demonstrates that there exists a bound on the execution time of the mission-critical control-loop despite the increasing load on the CPU. Of course there also appears to be some level of error associated with the results since the execution time is not constant, although this can be explained by the fact that there are still *some* kernel worker processes running on the same CPU as the mission-critical control-loop process and the low-priority processes that could interfere (as discussed in Section 3D), or it could be some

other cause such as CPU jitter. It is also important to note that these are experimental results over 10,000 data runs and do not constitute absolute proof that a deadline will not be missed or that the numbers could change if left to run for an infinite amount of time. However it can also be said that after more then 10,000,000 data runs the mission-critical control-loop cycle never went above 1.215 seconds and never went below 1.185 seconds no matter what percentage the CPU was loaded with. So in the author's opinion this result is still significant if at least for a preliminary result.

The maximum time range of the control-loop execution cycle over all CPU loads while using the Hard-RTSim framework was 0.026 seconds (1.213 seconds - 1.187 seconds). Comparing this value with the values obtained using the CFS scheduler in Section 2 (0.040 seconds at 0% CPU load and 1.443 seconds at 100% CPU load), as shown in Fig. 8, yields an improvement of 35% when the CPU is at 0% load ([[0.026 - 0.040] / 0.040] * 100) and an improvement of 98% when the CPU is at 100% load ([[0.026 - 1.443] / 1.443] * 100), with a steady increase in between. It is still important to note however that when the CPU is loaded at 100% the low-priority processes begin to starve. However this is unavoidable as there are a finite number of CPU cycles available per second. In relation to the motivating example, the UAV will not crash when the boundaries identified in Fig. 9 are respected. This was verified through a very large number of trials using the Hard-RTSim framework.

V. Conclusions

This paper has shown the problems that arise with general-purpose robotic simulations of autonomous aerial vehicles that are carried out on systems without hard real-time support as well as the similar problems with abstract robotic control interfaces that lack hard real-time support. The motivating example shows that timing anomalies can have catastrophic and unrecoverable affects on vehicles due to the way the process scheduler handles processes. The Hard-RTSim framework is presented as a solution to the identified problems and the experimental results shows an improvement of 35% when the CPU is minimally loaded and steadily increasing to an improvement of 98% when the CPU is loaded at its maximum, while also realizing a statistically significant finite bound on the CPU scheduling of the involved hard real-time processes. These results demonstrate that a general-purpose computer running a general-purpose operating system can realize hard real-time support for robotic simulators and abstract robotic control interfaces to yield very reliable simulations and real-world systems, and this will help to improve the state of real-world autonomous aerial vehicles and their associated algorithms.

References

[1]Vaughan, R. T. and Gerkey, B. P., "Reusable Robot Software and the Player/Stage Project," *Software Engineering for Experimental Roboticss*, Vol. 30, Springer Berlin Heidelberg, 2007, pp. 267–289.

[2]Gerkey, B. P., Vaughan, R. T., and Howard, A., "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," *In Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, 2003, pp. 317–323.

[3]Collett, T. H. J., Macdonald, B. A., and Gerkey, B., "Player 2.0: Toward a Practical Robot Programming Framework," *In Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, 2005.

[4]Vaughan, R. T., Gerkey, B. P., and Howard, A., "On device abstractions for portable, reusable robot code," *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, 2003, pp. 2421–2427.

[5]Gerkey, B. P., Vaughan, R. T., Stoy, K., Howard, A., Sukhatme, G. S., and Mataric, M. J., "Most Valuable Player: A Robot Device Server for Distributed Control," *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*, 2001, pp. 1226–1231.

[6]Wong, N., Hsu, J.-C. P., Collett, T. H. J., and MacDonald, B. A., "Improving The 2.5D STAGE Robotic simulator," *In Proceedings of the 2008 Australasian Conference on Robotics & Automation (ACRA 2008)*, 2008.

[7]Vaughan, R., "Massively multi-robot simulation in stage," Vol. 2, Springer Berlin Heidelberg, 2008, pp. 189–208.

[8]Biggs, G., Rusu, R., Collett, T., Gerkey, B., and Vaughan, R., "All the Robots Merely Players: History of Player and Stage Software," *IEEE Robotics & Automation Magazine*, Vol. 20, No. 3, 2013, pp. 82–90.

[9]Craighead, J., Murphy, R., Burke, J., and Goldiez, B., "A Survey of Commercial & Open Source Unmanned Vehicle Simulators," *In Proceedings of the IEEE International Conference on Robotics and Automation*, 2007, pp. 852–857.

[10]Staranowicz, A. and Mariottini, G., "A survey and comparison of commercial and open-source robotic simulator software," *In Proceedings of the International Conference on PErvasive Technologies Related to Assistive Environments (PETRA)*, 2011, pp. 56:1–56:8.

[11]Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y., "ROS: an open-source Robot Operating System," *ICRA workshop on open source software*, Vol. 3, 2009.

[12]Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., , and Ng, A., "ROS: an

open-source Robot Operating System," *In Proceedings of the IEEE International Conference on Robotics and Automation Workshop on Open Source Software in Robotics (ICRA 2009)*, 2009, pp. 1–6.

[13]Wilhelm, R. and Grund, D., "Computation Takes Time, But How Much?" Vol. 57, Association for Computing Machinery, 2014, pp. 94–103.

[14]Walker, S. M., "Hard-RTSim Source-Code and Tutorial," `"http://shawnmichaelwalker.com/research/Hard-RTSim"`.

[15] "Real-Time Linux Kernel Project," 2014, https://rt.wiki.kernel.org/index.php/Main_Page.

[16] "RealTime Application Interface for Linux," 2014, https://www.rtai.org/.

[17] "Real-time framework for Linux," 2015, https://xenomai.org/.