2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
September 27 - October 1, 2021. Prague, Czech Republic

JCopter: Reliable UAV Software Through Managed Languages

Adam Czerniejewski[1], John Henry Burns[2], Farshad Ghanei[1],
Karthik Dantu[1], Yu David Liu[2], and Lukasz Ziarek[1]

[1]Department of Computer Science, SUNY Buffalo
[1]{*adamczer, farshadg, kdantu, lziarek*}*@buffalo.edu*
[2]Department of Computer Science, SUNY Binghamton
[2]{*jburns11, davidl*}*@binghamton.edu*

***Abstract*— UAVs are deployed in various applications including disaster search-and-rescue, precision agriculture, law enforcement and first response. As UAV software systems grow more complex, the drawbacks of developing them in low-level languages become more pronounced. For example, the lack of memory safety in C implies poor isolation between the UAV autopilot and other concurrent tasks. As a result, the most crucial aspect of UAV reliability–timely control of the flight–could be adversely impacted by other tasks such as perception or planning. We introduce JCopter, an autopilot framework for UAVs developed in a *managed* language, i.e., a high-level language with built-in safe memory and timing management. Through detailed simulation as well as flight testing, we demonstrate how JCopter retains the timeliness of C-based autopilots while also providing the reliability of managed languages.**

I. INTRODUCTION

UAVs are witnessing an explosive growth in commercial and public-interest applications, such as search-and-rescue, first response, merchandise and medicine delivery, and precision agriculture. Software reliability is critical on UAVs, and any failure in UAV autopilot software could have catastrophic real-world consequences. According to a recent empirical study on UAV failures [1], navigation-related failures have one of the shortest Mean Time Between Failures (MTBF) and one of the highest incidence (over 30%) for commercial UAVs. In a recent survey with commercial UAV operators in the UK [2], the top 4 forms of failures are communication failures, navigation failures, power failures, and firmware failures; together, they are shown to be significantly more common than hardware failures e.g., in blades and motors. All except power failures are directly related to the autopilot software. More broadly, software reliability is well known to be a crucial component in maintaining the reliability of robotic and autonomous systems [3], [4], [5].

A. Software Failures in UAVs

For UAV systems, software reliability is a challenging goal as onboard software becomes increasingly complex. In practice, most UAV systems run several tasks concurrently and collaboratively to achieve desired behavior. Examples of such tasks include perception (e.g., identifying objects in field of view as seen by an onboard camera), planning (e.g., planning the trajectory in 3-D avoiding previously detected objects) and control (e.g., closed-loop flight autopilot).

Take the most important dimension of UAV software reliability, timeliness, for example. An algorithmically correct robot will not function if components, either software or hardware, do not meet timing deadlines and run at a specific rate consistently. In modern UAV software frameworks, achieving timeliness is complicated by the complexity of co-running tasks. Some are time-critical: autopilot software may need to run at 100 Hz (i.e., every 10 ms) to determine a set of roll, pitch, yaw and thrust values to command the UAV, and a timing failure may result in a crash. Other tasks such as perception are not time-critical and can be executed on a best-effort basis - if one frame was skipped in the processing, the overall goal of object identification may not be affected. In existing frameworks, a non-critical task can easily corrupt the critical task, leading to a timing failure for both. In addition to timeliness, other dimensions of UAV reliability, such as fail-safe support, ease of debugging, and memory availability, all become challenging goals as software becomes increasingly complex.

A root cause behind the spectrum of reliability challenges in UAV software systems is the lower-level languages in which they are written, such as C or C++. First, applications written in low-level languages have the potential to corrupt the memory space of another task and as such provide weaker *memory safety*, which translates to *poor isolation* among UAV tasks. A simple buffer overflow within a secondary payload may immediately corrupt the autopilot's data with catastrophic consequences. Second, the lack of type safety and high-level language features such as checked exceptions makes programs written in low-level languages *error-prone*, and these errors are often challenging to debug. Third, manual memory management may lead to *memory leaks*. This is detrimental for memory availability, especially on UAV systems with limited resources. C++ does provide a means for automatically freeing no-longer needed resources through the use of smart pointers. The burden to use this functionality is placed on the developer to ensure correct use throughout the code base.

*This project is sponsored by NSF Awards CNS-1823260, CNS-1823230, CNS-1846320 and SHF-1749539.

B. JCopter

In this paper, we introduce JCopter, a demonstration of the UAV autopilot software system in a high-level language with a managed runtime. We are motivated by the hypothesis that higher reliability can be achieved by developing UAV software systems in high-level languages, such as Java, for several reasons. (a) high-level languages enjoy type safety and support robust exception handling, making individual UAV tasks more robust, with fail-safe support by design; (b) they are endowed with memory safety, making it less likely for errors to propagate from one task to another. (c) they support automatic memory management, so that memory availability is enhanced without manual developer efforts. (d) Furthermore, real-time variants of high-level languages such as Real-Time Specification of Java (RTSJ) [6] and Safety Critical Java (SCJ) [7] allow the use of Java in real-time applications providing rigorous timing support [8], [9], [10].

One novel feature of the JCopter deployment is that the language we use is "Java with a twist": it syntactically and semantically conforms to Java language specification, but its managed runtime support is a low-overhead real-time variant of Java Virtual Machine (JVM) called Fiji [11]. Fiji provides the ability to execute multiple Java applications, providing strict time and space isolation between them [12], effectively allowing applications to execute in the same memory space but not be able to access each other's memory nor interfere with each other's timeliness. It compiles Java to C code with a minimal language runtime to produce fast executables. Most importantly, it provides a real-time garbage collector, allowing for automatic memory management with predictable timing [13]. The conformance to Java specification allows JCopter to achieve all the aforementioned benefits that come with a high-level language. The choice of a low-overhead variant of JVM offers JCopter with principled timing support and competitive performance. In a nutshell, this design integrates the best of both worlds.

To the best of our knowledge, JCopter is the first open-source autopilot for Java that flies on commodity UAVs. It is our reimplementation of the popular ArduPilot. In addition to the technical benefits of adopting a managed language, JCopter has the software engineering benefit of bringing Java developers into UAV software development, and bridging Java-based payload applications with UAV platforms. At the inception of UAVs decades ago, C perhaps was an appropriate choice considering the root of UAVs as resource-constrained embedded systems. However, the computing hardware support of this embedded system has gone a long way, and much progress has also been made on managed programming languages for embedded systems [12], [14], [15], [16], [17]. Our evaluation demonstrates that a Java-based autopilot can satisfy the performance and real-time requirements of UAVs in both simulation and real flights. Overall, JCopter is the first step toward a direction to build a Java ecosystem for UAVs, hitherto focusing on the most intricate part of UAV software, the autopilot.

Overall, this paper makes the following contributions:

- The implementation and demonstration of a Java-based autopilot for UAVs, with a novel combination of front-end Java syntactic and semantic support, and the back-end real-time runtime support.
- The evaluation of JCopter through systematic simulation and real-flight tests, demonstrating its feasibility and runtime characteristics.
- Our Java-based autopilot software is available for download [18], facilitating Java-based UAV software development and experimentation by others.

II. RELATED WORK

ArduPilot, Paparazzi UAV [19], [20] and PX4 [21] are the most popular autopilot frameworks for UAVs, but none is developed in a managed language. ROS [22] is widely used for robotic applications, written primarily in C++ with additional Python support. While ROS can work with UAVs, this middleware system in itself does not contain autopilot software modules, and still relies on a native autopilot.

From our survey, we believe ScanEagle [23] is the first autopilot written in Java that is RTSJ compliant. However its implementation is not publicly available. JAviator [24] includes multiple implementations for the flight control logic, including Java, but requires custom hardware. JUAV [25], [26] demonstrated the principles of Java-based autopilot support, but their proof-of-concept system does not compose with existing autopilot frameworks, and cannot be deployed to UAVs for flights.

Real-time Java has also been proven in other areas of robotics including industrial robot control [27]. IHMC investigated the use of real-time Java for humanoid robot control during the 2013 DARPA Robotics Challenge Trials [28], but found that the implementations available to them were too slow for their application.

More broadly, Java's use in embedded systems is ubiquitous with Oracle offering Java ME and Java SE, allowing developers to customize their JVM and deploy libraries to fit the footprint of their targeted embedded device. On the embedded and real-time side there are numerous real-time JVMs [14], [12] and two hardware-based JVMs, HVM and JOP. HVM [15], [16] provides a small footprint JVM capable of executing with only a few KB of RAM on bare metal. JOP [17] provides a custom processor able to execute Java bytecode with real-time extensions. None of these efforts is applicable to UAVs, but together, they offer evidence that Java can run efficiently on embedded systems.

Numerous empirical studies exist for failures in aviation or general robotic systems. As early as in 1996, NASA performed a comprehensive study in flight software complexity [29] and recommended complexity management including the use of higher-level languages to improve safety. A study of faults that occurred during RobotCup [3] shows a number of various errors including memory leaks, race

Autopilot	Issue #	Date	Type	Description
PX4	17908	7/13/2021	Multiple Memory Issues	Memory Leak, Write Out of Bounds, and Dereference Null Pointer
PX4	16140	9/6/2020	Process Isolation	Critical tasks being interrupted by non-critical tasks
ArduPilot	13917	03/28/20	Memory Leak	One-time memory leak for Lua scripting tool
ArduPilot	13820	03/16/20	Allocation Error	Memory reallocation method "realloc" was left undefined
ArduPilot	13792	03/12/20	Memory Leak	Memory leaks that cost too much flash memory to fix. Issue was left unresolved
ArduPilot	12146	08/26/19	Memory Leak	Memory leaks in the "heap_realloc" memory allocation method
PX4	12537	07/22/19	Memory Leak	Memory leak in uORB teardown
ArduPilot	11862	07/22/19	Write Out of Bounds	"readlink" method can return out-of-bounds memory region
ArduPilot	9137	08/8/18	Write Out of Bounds	User induced out of bounds write in RC channel
ArduPilot	8644	06/14/18	Memory Leak	Memory Leak in posix fprintf function
ArduPilot	8642	06/14/18	Memory Leak	Memory Leak in video bench marking

TABLE I: Relevant ArduPilot and PX4 GIT issues

conditions, and overflows. Sotiropoulos *et al.* [4] analyzed commits in libraries for navigation algorithms and ".. added a separate memory class to emphasize the high number of such bugs". More recently, Garcia *et al.* [5] present a comprehensive study of bugs found in autonomous vehicles. The authors classify the root cause for a given bug into one of 13 classes, including memory mismanagement, misuse of pointers, and concurrency-related problems.

III. MOTIVATION

In this section, we summarize the benefits of high-level languages in enhancing the reliability of UAV systems. In general application domains, these benefits are well documented [30], [31]. The motivation of JCopter is to bring these benefits into the domain of UAV software. To set these benefits in perspective, Table I lists recent issues [32], [33] from popular C++-based UAV frameworks, ArduPilot [34] and PX4 [21]. Despite the large user base and many years of development, memory bugs still exist. If this was developed in a managed language such as Java, such errors could be eliminated by the guarantees provided by the language.

A. Memory Safety

A key benefit of a high-level language such as Java is that its program runtimes are guaranteed to stay within their allocated memory space. This becomes increasingly important when deploying multiple applications on the same hardware. For UAVs, this means that auxiliary payload written in Java will not be able to break the bounds of their approved memory space, ensuring critical system memory would remain safe from serious errors such as buffer overflows, found in lower-level languages like C/C++. Git issues 13820, 11862, and 9137 presented in Table I highlight issues maintaining memory safety with lower-level languages.

B. Type Safety

Java's strong typing allows many program errors to be found during compilation. Type safety further strengthens the ability of program reasoning, leading to powerful debugging tools with static and dynamic analysis. For weakly-typed languages such as C/C++, unsafe features such as pointers, casting, unions, and function pointers may lead to subtle bugs challenging for developers when software becomes complex.

C. Exception Handling

High-level languages such as Java provide comprehensive support for failure semantics. In Java for example, *checked exceptions* embody program errors, and the compiler will ensure they are explicitly handled by the program. Checked exceptions represent the majority of semantic errors in Java, such as null pointer dereferences, and array out of bounds. The exception handler offers principled support to gracefully recover from errors, e.g., disable a non-critical component, reinitialize resource, or provide suitable feedback to user. This could be useful in UAVs to mediate between critical tasks and non-critical tasks, allowing for correct recovery (e.g., skip the non-critical task).

In contrast, C does not provide exceptions, so errors may lead to undefined behavior. C++ provides exception handling, but all exceptions are unchecked, i.e., the compiler is not responsible for guaranteeing any exception to be handled. Some errors indicated in Table I occurred due to unhandled errors (e.g., null pointers, failure to open a resource).

D. Automatic Memory Management

Memory management in resource-constrained settings is challenging and requires programmers to be judicious in the memory usage, re-use resources (e.g. object pools), and for safety-critical applications, eschew dynamic allocation altogether (all memory is allocated up front prior to the mission phase). Automatic memory management such as garbage collection (GC) eliminates the need for the programmer to reason about memory, resulting in simpler application code. In C/C++, manual memory management may cause memory leaks (an availability threat) or dangling pointers (a correctness threat). The git issues 13917, 13792, 12146, 8644, and 8642 show the difficulties of memory management in a low-level language, as shown in Table I.

E. Timeliness and Prioritization

Real-time variants of Java provide a mechanism to execute Java applications both efficiently and predictably. This predictability is rooted in its underlying implementation of a real-time scheduler which facilitates GC to run at a lower priority, ensuring it does not induce unexpected pauses to the system that would be detrimental to overall execution. Prioritization is also accessible to developers, allowing for the creation of prioritized tasks where they can prioritize

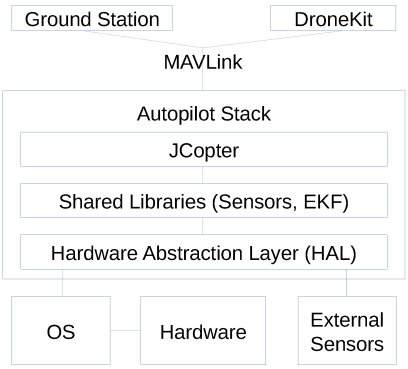


Fig. 1: The UAV Software Ecosystem

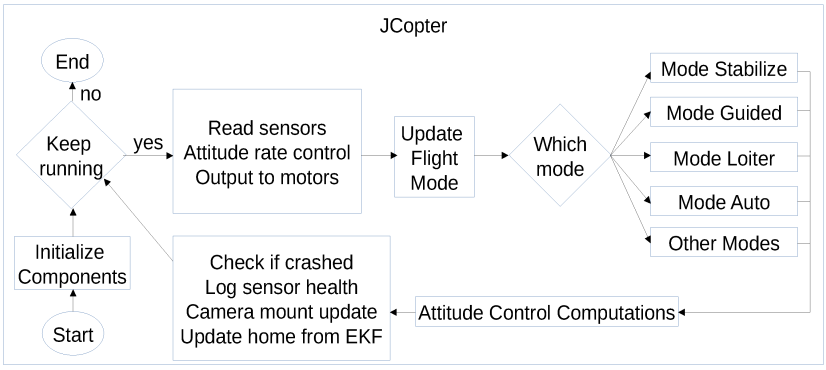


Fig. 2: A Flow Chart with JCopter Components

critical tasks such as flight control over less important tasks like route planning.

Fiji as a real-time JVM allows for prioritized predictable execution of applications implemented in Java. In the remainder of this paper, we discuss our implementation and demonstrate how the unpredictability of commodity Java JVMs can lead to failures of an application and how Fiji's predictability ensures that they do not occur.

IV. IMPLEMENTATION

Our JCopter framework is a UAV autopilot system written in Java. In the UAV software ecosystem, the autopilot represents the most safety-critical component where reliability matters the most. To bridge gracefully with other components of UAV software, we choose to develop JCopter as a module that interacts with ArduPilot [34], one of the most popular open-source autopilots currently available. Shown in Figure 1 is the interaction between JCopter and ArduPilot [35]. ArduPilot can be used to command several classes of UAVs, UGVs and AUVs. As shown in the diagram, the autopilot stack is divided into three main sections - (i) JCopter, which represents the actual control for the copter, (ii) Shared libraries, which are leveraged by JCopter to interface with the various sensors as well as the estimation, and (iii) a Hardware Abstraction Layer (HAL) which provides the means for interaction with the actual hardware available to the JCopter control code. The JCopter framework is a Java-based implementation which consists of 3590 LoC across 26 classes and makes use of Java dependencies managed in Maven Central. JCopter directly leverages the stack found in ArduPilot and only aims to support quad rotor UAVs. It focuses on bringing vehicle-specific flight code to Java (JCopter), using the hardware interactions and communication mechanisms provided by ArduPilot.

Specifically, JCopter implements the components that constitute the core autopilot computations - the attitude control for a number of flight modes. The overall execution flow for JCopter can be seen in Figure 2. The execution begins with the initialization of any hardware (sensors, ESCs, etc.) and internal data-structures. So long as the autopilot is not told to cease executing, it enters the main control loop of the autopilot. The first step is to read any updated input values, and send output values to the motors based on the attitude control computations of the previous iteration of the cycle. The next block ensures the flight mode is set appropriately based on the new command received. Then the set flight mode is executed. Each flight mode provides different functionalities than the others (Loiter causes the UAV to stabilize near a position based on GPS, Auto executes a series of steps configured in a flight plan, Guided allows for the dynamic setting of way-points, and Stabilize keeps the frame level based on attitude readings), but in the end they all determine a target attitude (Yaw, Pitch, and Roll) in the common Attitude Control Computations block. Lastly a series of checks are performed on the UAV and its position estimation. For non-safety-critical components we leverage the original ArduPilot code, communicating through Java Native Interface (JNI). Further, a number of modifications were made to the ArduPilot's code base to add required accessors and mutators. The result of any computations is maintained in the normal native locations ensuring both Java-based and C-based logic remain in sync throughout a given flight. The compilation of JCopter is modified based on the original ArduPilot's build process to ensure all dependencies (such as shared libraries) are correctly fulfilled.

With Fiji JVM, JCopter is capable of prioritizing threads in a real-time setting. In our implementation, we prioritize the autopilot's execution over any additional payloads that may be deployed. In a similar vein, the autopilot thread also has a higher priority than GC threads. As a result, JCopter can pause or delay the GC execution if the autopilot may risk missing a deadline.

V. EVALUATION

In this section we demonstrate the benefits of JCopter and its use of managed languages through three experiments.

- *Timing of Real-time Tasks*: We study the difference in execution speed on real-time tasks, the JCopter autopilot in this case, with the various implementations to study the ability of the underlying language to incorporate timing requirements.

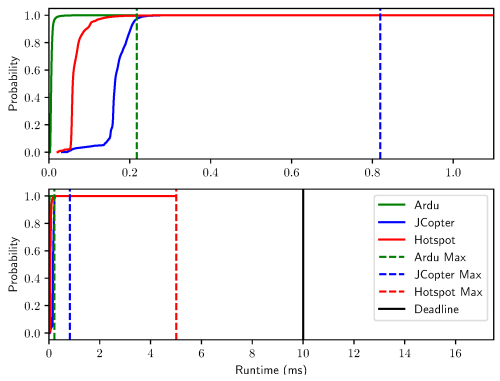


Fig. 3: CDF of Execution Time for Flight Mode Without Payload. The dotted lines indicate the maximum amount of time spent for completing a task. The lower graph shows the overall trend, and the upper one time from 0 to 1.1 ms.

- *Effects of Concurrency*: As described in section I, modern UAV software is complex with multiple concurrent tasks being run with varied timing requirements. To study the effect of the language on such concurrent tasks, we study the behavior of JCopter when run alongside other software payloads.
- *Real-world Deployment*: While the first two studies are conducted in simulation, we demonstrate the correspondence of those results on real-world hardware in flight that can be seen in video summarizing this work [36].

The simulation-based experiments (the first two above) allow us to study the design space comprehensively and identify configurations that might be unsafe to fly. Timely execution in more resource-constrained environment (the third above) demonstrates the real-world utility of JCopter. All simulations were performed on a 4-core Intel i7 with 16GB of RAM running Ubuntu 16.04. The flight testing was done on the Erle-Copter [37] quad copter with Erle-Brain 3 as the computing module. Every experiment involves running three versions of the autopilot.

- *ArduPilot (Ardu)*: The unmodified C++-based open-source autopilot.
- *JCopter-HotSpot Hybrid (HotSpot)*: The JCopter autopilot running on Hotspot JVM.
- *JCopter*: The JCopter autopilot running on the Fiji real-time JVM.

Among the 3 versions, *Ardu* serves as the baseline with an unmanaged language implementation, representing the state of the art. *Hotspot* forms another baseline to demonstrate "what would have been" if a developer were on board with a managed language, but naively chose a commodity JVM. Together, these two baselines are selected for a comprehensive comparative study of the benefits of JCopter.

In UAV systems, *timing* is critical to the operational safety of the system, and thus fundamental to the reliability of the underlying software. Unlike conventional computer systems where the most effective design comes with the shortest execution time, the key metric for UAVs is to ensure safety-critical tasks are processed in a timely manner, i.e., always meeting their deadlines. For the ArduPilot attitude control the deadline is 10 ms as each of the flight modes is expected to run at a rate of 100 Hz. All experiments are conducted over a single flight and measure the absolute execution time for every execution for the current flight mode through the attitude control computations as seen in Figure 2.

A. Timing of Real-Time Tasks

We start with a simple experiment where the UAV autopilot is running without any additional payload. This serves as a sanity check on the UAV "flying but idle" behavior. Figure 3 shows cumulative distribution function (CDF) for the execution time of the flight mode being used in simulation. As seen here, the Ardu maximum (0.218 ms) is significantly lower than the Hotspot and JCopter maximums (5.005 and 0.819 ms respectively). This is not surprising considering the additional CPU cycles required for maintaining bookkeeping services of the JVM. However, note that both Java versions are able to finish their execution without missing *any* deadline. Indeed, the maximum execution time of JCopter is more than one order of magnitude lower than the deadline. Therefore, both Java-based solutions remain effective for the autopilot task in this scenario.

From the zoomed-in figure, it may be noted that *on average*, *Hotspot* appears to have a shorter execution time than JCopter. This trend will recur in the rest of our evaluation. UAV reliability however, is not about the *average* execution time, but about whether the *worse case* execution time exceeds the deadline. As we shall see, *Hotspot* is prone to "wild swings" in execution time. In JCopter's JVM choice Fiji, additional support is added to improve the predictability of scheduling and garbage collection behavior, resulting in a mild overhead. As long as the deadline is met however, it remains an effective solution.

B. Autopilot with Concurrent Payload

We next consider the more realistic scenario where the autopilot is running concurrently with some payload application. Our hypothesis is that, if the autopilot and the payload application run on the same processor, not only do the memory and computational requirements of the payload affect the autopilot timing, but they may have the ability to disrupt execution through improperly implemented code. In order to test this hypothesis, we have created three payloads:

- *CPU-Intensive Payload*: a navigation application which is typically run alongside the autopilot for high level path planning.
- *Memory-Intensive Payload*: an application that stresses the memory allocation and garbage collection.
- *Memory-Corruption Payload*: an application that triggers a buffer overflow which may corrupt the memory area of other applications.

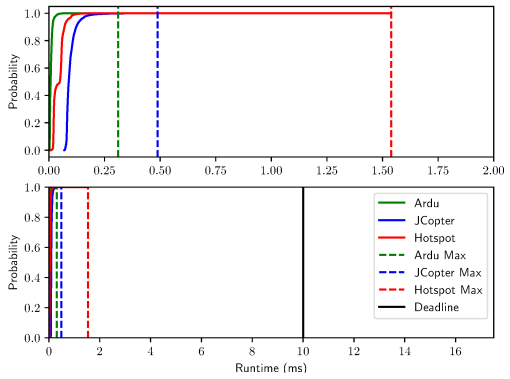


Fig. 4: CDF of Execution Time for Flight Mode with Concurrent CPU-Intensive Payload. The lower graph shows the overall trend, and the upper one time 0 to 2 ms.

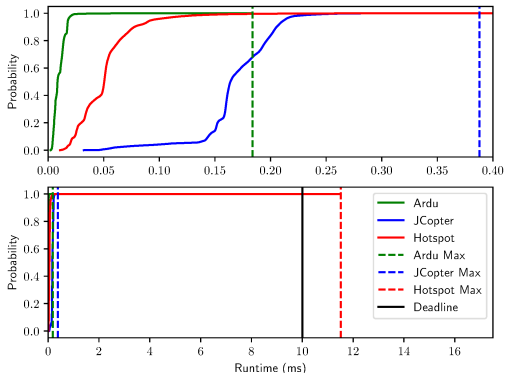


Fig. 5: CDF of Execution Time for Flight Mode with Concurrent Memory-Intensive Payload. The lower graph shows the overall trend, and the upper one time 0 to 0.4 ms.

Following, we elaborate on each of the payload applications and our results.

1) CPU-Intensive Payload: Our navigation payload application is based on the A* algorithm [38], and operates on a NxN grid. For simulation we use a larger 900x900 grid to stress the system highlighting potential deficiencies prior to a real-world deployment. Like most real-world payload applications, the execution of this algorithm is interruptible. It operates on a 1000 ms period, with a 50% duty cycle where it executes for 500 ms and is halted for the remaining 500 ms. This downtime gives the Fiji and Java JVMs some time for their own bookkeeping (such as garbage collection). When testing our A* payload application in simulation, we found that all three autopilots were able to meet the deadline of 10 ms. The results are shown in Figure 4. As expected, Ardu had the lowest maximum execution time of 0.312 ms. Hotspot and JCopter ended in 1.540 ms and 0.489 ms, respectively, both significantly ahead of the deadline. Comparing these results we notice that the longest executions for both Hotspot and JCopter are shorter with the CPU intensive payload than when run without a secondary payload. Increased contention for compute time decreases memory usage between garbage collections, allowing for only minor collections to be needed by Hotspot, and shorter runs of the Fiji's garbage collector used by JCopter.

2) Memory-Intensive Payload: Our memory consuming payload application repeatedly allocates 1.6 Megabytes of memory and halts (as to not consume processor time). This places a stress test to the garbage collector in managed languages. Figure 5 shows the CDF of execution times in the presence of the memory payload. In this execution, Ardu finishes all its runs within the deadline with the maximum run finishing in 0.184 ms. This is not surprising given Ardu has manual memory management and no garbage collection. Hotspot missed one of its deadlines and had a maximum execution time of 11.519 ms. JCopter finished its longest execution in 0.388 ms, well below the specified deadline. This experiment is interesting as it shows the importance of Fiji, a JVM with efficient memory management and a small memory footprint, as opposed to Hotspot. It is a conscious reminder that UAVs have distinct requirements in reliability and resource management: Java can work well, but a careful design in JVM is required.

3) Memory-Corruption Payload: Java provides runtime array bounds checking, a feature C/C++ does not support. In this experiment, we inject a buffer overflow error to our navigation payload program, by writing to an array until it exceeds its bound of 500 elements. In *Ardu*, the application payload continued to write beyond the bound of the array until crashing due to a segmentation fault, leading to crashing the autopilot as well. When tested in three consecutive runs, *Ardu* crashed at the 464th, 461th, and 459th iterations after exceeding the array bound, causing corruption in core autopilot data structures. In both *Hotspot* and JCopter however, the payload application was able to exit through exception handling upon the first write that exceeded the array bound. As a result, the autopilot continues to run. This experiment demonstrates the benefit of Java's support for memory safety in isolating failures. In a real-world application, the autopilot program should by no means crash if the navigation payload is faced with an error. With JCopter, the payload application can be terminated gracefully without interfering with other concurrent tasks such as the safety-critical autopilot.

C. Real-world Deployment

While the simulation provides a controlled environment for the relative comparison of the three autopilots, hardware variations may still play a role in program behavior when the autopilot is deployed in a real-world setting. For instance, the Raspberry Pi 3 has the same number of physical cores but can compute fewer floating point operations per second and has only 1GB of RAM compared to the 16GB in our simulation environment. In this section, we repeat our earlier experiments on a real-world UAV hardware.

1) JCopter Experiments without Payload: Figure 6 shows the CDF of execution times for the three autopilots while flying the Erle-Copter with no additional payload. All three of our autopilots missed no deadlines. Ardu finished its

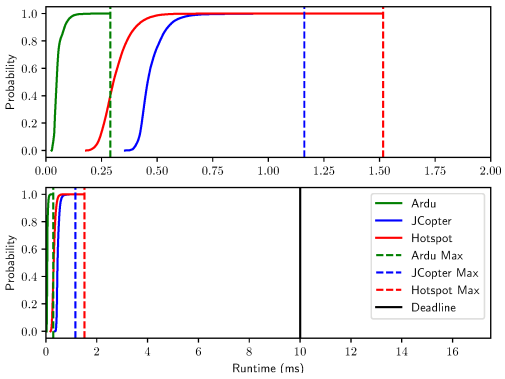


Fig. 6: CDF of Execution Time for Flight Mode Without Payload While In-Flight. The lower graph shows the overall trend, and the upper one time 0 to 2 ms.

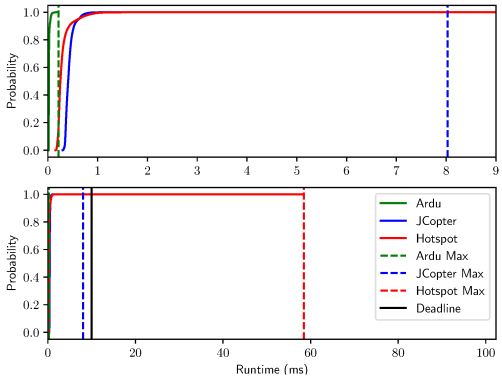
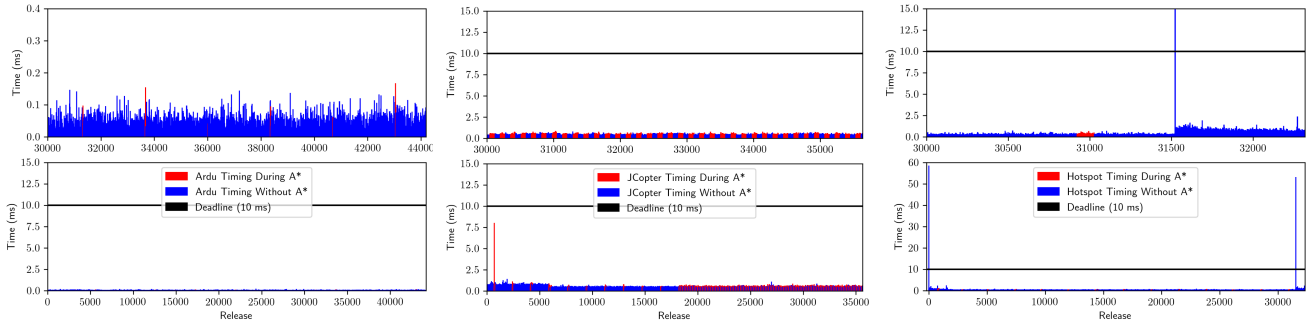


Fig. 7: CDF of Execution Time for Flight Mode with A* Payload While In-Flight. The lower graph shows the overall trend, and the upper one time 0 to 9 ms.



(a) Ardu (b) JCopter (c) Hotspot

Fig. 8: Execution Time for Flight Mode with A* payload While In-Flight (Each point in the X-Axis indicates a single execution and release of the Flight Mode, and the Y-Axis indicates its execution time. The lower graph shows all releases, and the upper one shows release 30000 and onward in more detail.)

maximum execution time in 0.290 ms. Hotspot finished in 1.516 ms. However, using Fiji's real-time abstractions, JCopter reduced the maximum execution time to 1.162 ms.

2) JCopter Experiments with Payload: Figure 7 shows the result of the flight mode for the Ardu, Hotspot, and JCopter autopilots running concurrently with the A* payload, on a grid of 128x128. For safety reasons we executed the autopilot along with the navigation application with the propellers removed. The copter was manually kept in the air so it did not auto-disarm while collecting the timing measurements. As shown in the figure, Hotspot may miss a deadline whereas both Ardu and JCopter do not. To gain an in-depth understanding of the in-flight behavior, Figure 8 shows the absolute execution time *per release*. The red bars indicate the absolute time for each execution of the flight mode through the attitude control computations that occurred during the active time of the A* payload, while the executions indicated by the blue bars occurred during the payload's downtime. Ardu and JCopter, in Figure 8 (a) and Figure 8 (b) respectively, missed no deadlines while running the A* payload. However, Figure 8 (c) shows that Hotspot missed two deadlines while the payload was halted. The absolute execution time for these iterations were 58.470 ms. and 53.226 ms. A missed deadline may result from the competition either from the navigation payload or the garbage collector thread. The observed misses occurred due to GC because as seen in the figure, our A* payload was halted when they occurred. In JCopter, the Fiji support places the autopilot at a higher priority with real-time deadline support, and no deadline is missed as a result. As can be seen in Figure 7 these features allow JCopter to decrease its maximum execution time from Hotspot's 58.470 ms, to 8.027 ms. Ardu had a maximum execution time of 0.213 ms, thus both JCopter and Ardu met all their deadlines.

VI. CONCLUSIONS

With the utility of UAVs, it is not surprising their use-cases are increasing. Compound that with the proliferation of UAV hardware available to both industry and consumers and it is understandable that new applications to further enhance them will be created. When these new functionalities are co-located with the flight controller, reliability is a key concern: the payload applications should not negatively affect the reliability of critical tasks. This is a non-trivial problem when programs implemented in low-level languages may cause the UAVs to fail due to software bugs (e.g., buffer overflow), unhandled errors, and resource contention.

In this paper we presented JCopter, a Java-based autopilot which makes use of managed languages, reducing the possibility of run-time errors by providing (1) memory safety,

(2) type safety, (3) exception handling, and (4) automatic memory management. These benefits ensure the reliability of applications and decreased development time. Our Fiji-based JVM further ensures predictable execution of applications in real-time deployments. Together, we show that JCopter offers a practical solution to improve the reliability of UAVs. We believe that JCopter provides a means for the community to assess and gain confidence in the applicability of managed languages in time-sensitive robotic platforms, with current real-time Java implementations.

REFERENCES

[1] E. Petritoli, F. Leccese, and L. Ciani, "Reliability and maintenance analysis of unmanned aerial vehicles," *Sensors*, vol. 18, no. 9, p. 3171, 2018.

[2] M. Osborne, J. Lantair, Z. Shafiq, X. Zhao, V. Robu, D. Flynn, and J. Perry, "Uas operators safety and reliability survey: Emerging technologies towards the certification of autonomous uas," in *2019 4th International Conference on System Reliability and Safety (ICSRS)*. IEEE, 2019, pp. 203–212.

[3] G. Steinbauer, "A survey about faults of robots used in robocup," in *Robot Soccer World Cup*. Springer, 2012, pp. 344–355.

[4] T. Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand, "Can robot navigation bugs be found in simulation? an exploratory study," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 150–159.

[5] Garcia, Joshua and Feng, Yang and Shen, Junjie and Almanee, Sumaya and Xia, Yuan and Chen, and Qi Alfred, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 385–396.

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

[7] JSR 302, "Safety Critical Java Technology," 2007.

[8] H. Park, A. Malik, M. Nadeem, and Z. Salcic, "The cardiac pacemaker: Systemj versus safety critical java," in *Proceedings of the 12th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 37–46. [Online]. Available: https://doi.org/10.1145/2661020.2661030

[9] A. Cavalcanti, A. Wellings, J. Woodcock, K. Wei, and F. Zeyda, "Safety-critical java in circus," in *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 20–29. [Online]. Available: https://doi.org/10.1145/2043910.2043915

[10] A. Wellings, M. Luckcuck, and A. Cavalcanti, "Safety-critical java level 2: Motivations, example applications and issues," in *Proceedings of the 11th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 48–57. [Online]. Available: https://doi.org/10.1145/2512989.2512991

[11] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, "High-level programming of embedded hard real-time devices," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 69–82. [Online]. Available: https://doi.org/10.1145/1755913.1755922

[12] L. Ziarek and E. Blanton, "The fiji multivm architecture," in *Proceedings of the 13th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2822304.2822312

[13] 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 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 146–159. [Online]. Available: https://doi.org/10.1145/1806596.1806615

[14] F. Siebert, "Hard real-time garbage-collection in the jamaica virtual machine," in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, ser. RTCSA '99. USA: IEEE Computer Society, 1999, p. 96.

[15] S. E. Korsholm, H. SÃžndergaard, and A. P. Ravn, "A real-time java tool chain for resource constrained platforms," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 14, pp. 2407–2431, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3164

[16] K. S. Luckow, B. Thomsen, and S. E. Korsholm, "Hvmtp: A time predictable and portable java virtual machine for hard real-time embedded systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 22, p. e3828, 2017, e3828 cpe.3828. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3828

[17] M. Schoeberl, "Jop: A java optimized processor," in *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, R. Meersman and Z. Tari, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 346–359.

[18] "Jcopter-repository," https://github.com/adamczer/JUAV-ardupilot.

[19] "Paparazzi uav," https://wiki.paparazziuav.org/wiki/Main_Page.

[20] "paparazzi-github," https://github.com/paparazzi/paparazzi.

[21] "Px4 github," https://github.com/PX4/PX4-Autopilot.

[22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[23] 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)*, vol. 7, no. 1, p. 5, 2007.

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

[25] A. Czerniejewski, S. Cosgrove, Y. Yan, K. Dantu, S. Y. Ko, and L. Ziarek, "juav: A java based system for unmanned aerial vehicles," in *Proceedings of the 14th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2016, pp. 1–10.

[26] A. Czerniejewski, K. Dantu, and L. Ziarek, "juav: A real-time java uav autopilot," in *2018 Second IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2018, pp. 258–261.

[27] S. G. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, "Using real-time java for industrial robot control," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, 2007, pp. 104–110.

[28] M. Johnson, B. Shrewsbury, S. Bertrand, T. Wu, D. Duran, M. Floyd, P. Abeles, D. Stephen, N. Mertins, A. Lesman, *et al.*, "Team ihmc's lessons learned from the darpa robotics challenge trials," *Journal of Field Robotics*, vol. 32, no. 2, pp. 192–208, 2015.

[29] D. Dvorak, "Nasa study on flight software complexity," in *AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference*, 2009, p. 1882.

[30] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*. Addison-Wesley Professional, 2000.

[31] M. L. Scott, *Programming language pragmatics*. Morgan Kaufmann, 2000.

[32] "Sample of ardupilot memory related git issues," https://github.com/ArduPilot/ardupilot/issues?q=13917+13820+13792+12146+11862+9137+8644+8642.

[33] "Sample of px4 git issues," https://github.com/PX4/PX4-Autopilot/issues?q=17908+16140+12537.

[34] "Ardupilot," https://ardupilot.org/.

[35] "Ardupilot architecture," https://ardupilot.org/dev/docs/apmcopter-code-overview.html.

[36] "Jcopter-iros-2021-video," https://vimeo.com/580019688.

[37] "Erle-copter," https://erlerobotics.gitbooks.io/erle-robotics-erle-copter/content/en/.

[38] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.