Reliable flight control system architecture for agile airborne platforms

an asymmetric multiprocessing approach

Majumder, Shibarchi; Nielsen, Jens Frederik Dalsgaard; Bak, Thomas; La Cour-Harbo, Anders

Published in:
The Aeronautical Journal

DOI (link to publication from Publisher):
10.1017/aer.2019.30

Publication date:
2019

Document Version
Accepted author manuscript, peer reviewed version

Reliable Flight Control System Architecture for Agile Airborne Platforms: An Asymmetric Multiprocessing Approach

Shibarchi Majumder, Jens Frederik Dalsgaard Nielsen, Thomas Bak, Anders la Cour-Harbo*

Department of Electronic Systems, Aalborg University
Aalborg
Denmark

ABSTRACT

System software subsystems in an unmanned aircraft system share hardware resources due to space, weight, and power constraints. Such subsystems have different criticality, requirements, and failure rates, and can cause undesired interference when sharing the same hardware. A component with high failure rate can reduce the reliability of the system unless a fault containment mechanism is adopted.

This work proposes an asymmetric multiprocessor architecture to establish isolation at the hardware level for distributed implementation of safety-critical subsystems along with user defined payload subsystems on the same hardware with minimally reduced reliability of the system. To achieve that, subsystems are strategically segregated in separate processors, connected to an on-chip protective interconnect for inter-processor communications. A custom watchdog and reset mechanism are implemented to reset a specific processor without affecting the entire system if required. The architecture is demonstrated on a FPGA chip. In addition, an example of an optimized distribution is provided for a specific flight control system with five subsystems.

Keywords:

Mixed-Criticality System; Reliable-multiprocessing; Embedded Flight Computer; Asymmetric-multiprocessing; Reliable Embedded System; Unmanned Aircraft System

* **This research is funded by Danish Independent Research Foundation under grant number 6111-00363B.** **Email: sm@es.aau.dk, jdn@es.aau.dk, tba@es.aau.dk, alc@es.aau.dk**

Received 12 12 2018; revised DD MM YYYY; accepted 12 03 2019.

NOMENCLATURE

Symbols

λ	Rate of failure per unit time
c_m	Criticality of subsystem m
S_m	Subsystem m
R_m	Reliability of subsystem m
R_{sys}	Reliability of overall system
S_{all}	Union of all subsystems

Abbreviations

ALU	Arithmetic and logic unit
AMP	Asymmetric multi-processor
FCS	Flight control system
FPGA	Field programmable gate array
HDL	Hardware description language
IMU	Inertial measurement unit
PID	Proportional-integral-derivative

1.0 INTRODUCTION

Reliability is the key aspect when considering operations of unmanned aircraft systems (UAS), or drones, in close vicinity to people. The reliability aspect of the embedded flight controller for UAS was neglected earlier, probably because faults were often encountered from other physical components. Recent research outcomes in control algorithms have made it possible to keep the craft airborne even after failure of major subsystems[1] that gives us the motivation to put effort towards a reliable embedded flight controller platform.

Conventional solutions to improve reliability in embedded platforms is achieved by extensive hardware redundancy, which is not feasible for agile aerial platforms due to space, weight, and power (SWaP) constraints. In a generic UAS embedded flight computer, the applications with different level of criticality share the same hardware due to tight SWaP budget. Operation or failure of one application can cause interference that might cause malfunction in other applications or even the entire system, compromising the overall reliability. Although the software implementation of conventional flight control algorithms are widely evaluated and considered reliable, the payload/mission specific applications, implemented by the end-user, can have higher complexity and can be hard to analyze, hence, may have imperfections.

An isolation mechanism is beneficial for fault containment and prevention of inter-application interference. A *hypervisor* and *embedded operating system*[6] can provide such an isolation in software by restricting the access of individual applications to shared resources up to a certain degree. However, implementing software-based safety mechanism is a complex task that becomes even more complex for mixed-criticality system and multicore implementation[9]. Additionally, the protective software (e.g. an operating system) consumes significant amount of computational resources. Distributed and redundant implementation in isolated partitions in embedded platforms

are done by considering the reliability of the subsystems of the system. For safety-critical systems, like UAS, analyzing the criticality of the subsystems is important for efficient distribution. In the aviation industry, such criticality levels are analyzed, and systems with different criticality levels are kept isolated, which is not feasible for low-cost platforms.

In this work, we present a distributed implementation of a conventional *flight control system* (FCS) software on a custom multi-processor architecture, that establishes isolation at hardware level. The proposed distribution method considers reliability as well as criticality of the subsystems for distributed implementation to minimize the probability of system failure; which we define to be a loss of control situation. For implementation on a resource limited platform, applications with different levels of criticality and reliability are allowed to share the same processor. Each processor is an independent system, with dedicated and isolated resources, connected to a protective inter-connect for inter-processor communication with message-passing technique. Establishing isolation at hardware level provides the benefits of a distributed and bare-metal implementation with ease of scheduling and execution time analysis. Furthermore, such a platform is free from the complexity of multi-threading and offers re-usability of applications developed for single processor systems.

The paper is structured as follows. Section 2 gives an overview of related works and presents regulations for unmanned aircraft and its subsystems. Section 3 provides an insight of a generic FCS architecture and functionality of different subsystems. Section 4 introduces the proposed system architecture and its components followed by implementation of a FCS in the architecture in Section 5. Section 6 presents an analysis of fault-tolerance under some given condition. Experimental results are presented in Section 7 and Section 8 concludes the paper.

2.0 Background and Related Work

Despite the fact that the dynamics and control of manned and unmanned aircraft systems are very similar, the software implementation in unmanned aircraft system is very different. Apart from the quality regulations and certification standards, the distributed and redundant control structure in manned aviation makes it more reliable where a single failure can hardly result in a catastrophe. Moreover, separate hardware and system isolation prevent interference.

The advancement in FPGA technology has made it possible to develop application-specific custom processor architectures, and asymmetric multiprocessor (AMP) architectures have gained the attention of researchers. Unlike synchronous multiprocessors where processors are connected to shared memory, in AMP architecture processors are distributed in clusters and resources are shared within a cluster only. Such architectures have been under consideration for safety-critical applications. In [7], researcher have investigated such a platform for avionics implementation, where a health monitoring system of a helicopter is implemented in a many-core architecture. The authors considered the regulations for airborne systems, and isolation was established between subsystems of different criticality level.

Established solutions for safety-critical applications largely depends on availability of resources for recovery. In [5], such a redundant implementation is discussed.

In resource-limited platforms, such availability of resources is not feasible. The researchers in [3] introduced an in-operation error detection and recovery technique and proposed novel methodologies for efficient resource utilization in redundant hardware. An adaptive approach with a dynamic scheduling solution is provided in [4]. For efficient resource utilization in multicore architectures load balancing techniques are considered where effort is given to distribute the computation tasks evenly on the available processors. Such practice of distributed implementation is beneficial for general purpose computation in terms of overall throughput and power, but for safety-critical applications considering reliability and criticality of the tasks is important. In [14] and [11] the reliability and dependability of software implementations for different application of different size and functionality is presented. The authors have described several approaches of estimating achievable software reliability under different conditions. In [15], the author has described software reliability modes, including system availability, safety, security, and criticality aspects of software.

In [13], field failure rate of an integrated software system is analyzed prior to field deployment, where a field failure rate prediction methodology is described by considering system test data and field data using software reliability growth models. Such a methodology gives an estimation of software reliability in a targeted platform before the actual deployment of the software. The authors in [12] propose a new software reliability model that takes into account the uncertainty of operating environments, that is helpful for estimating the reliability of the software implementation in a hardware environment that is different from the one, for which the software was developed and tested.

3.0 Flight Controller Architecture for UAS

The objective of an effective flight controller is to reach desired state at desired time instant by wisely manipulating the actuators, considering the dynamics of the vehicle. Although the implementation of a flight controller varies with the dynamics of the platform, the control architecture is very similar for conventional fixed wing and multirotor platforms.

A conventional flight controller consists of multiple cascaded closed-loop controllers, where the control command flows from the outer loop to the inner loop. The innermost loop is closely coupled with the physical actuators that establishes link between the plant (the UAS platform) and the controller. Required control loop frequency depends on the application, environmental factors, and dynamics of the platform.

A multirotor platform has faster dynamics compared to a fixed-wing platform of similar size and weight, and requires a faster control response for controlled maneuvers. For this reason, this work will present the proposed method as applied to a FCS architecture for a multirotor platform.

A multirotor is controlled by actuating the thrust from each rotor, by manipulating the angular velocity of the rotors. The inner-loop controller, typically an angular velocity controller, is a closed-loop controller that controls the angular velocity of the multirotor by directly controlling the rotors by taking desired angular velocities as a reference signal from the attitude controller and actual *roll, pitch* and *yaw* motions (from an on-board sensor/estimator) as feedback signal as shown in Figure-1. The

attitude controller gets references from a translational velocity controller, which in turn depends on a position controller and ultimately a trajectory planner. In a hover condition, where reference signals are zero, the angular velocity controller controls the rotors to hold the multirotor in a horizontal position.

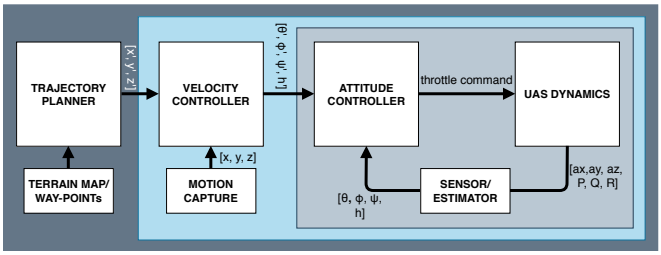

Figure 1. Block diagram of a generic FCS implementation, where inner to outer loops are represented in a right to left order. $[\theta', \phi', \psi', h']$ is the reference attitude and altitude. $[ax, ay, az, P, Q, R]$ are accelerations and angular velocities respectively.

The position controller is a *feedback controller* that takes a reference signal from a *trajectory planner*, feedback signal from a positioning device (such as GPS or motion capture system) and outputs desired attitude state variables that is fed to the attitude controller as an input. A trajectory planner could have very different approaches of generating output signal in the form of position (x', y', z') or position-velocity (x', y', z', u', v', w') combination that goes into the position/velocity controller(s). A vast range of different approaches like way-point navigation, vision-based navigation, vision-IMU odometry, genetic algorithm etc. are used for trajectory planning.

3.1 Sources of failure and effects

A system failure in this context can be defined as an uncontrolled, unrecoverable maneuver resulting in a crash of the platform. This work considers the failure related to the FCS only, and not failures related to mechanical, power or external peripheral (GPS/ motion capture device) failures. We have considered that the operation of the UAS is completely autonomous and there is no pilot input to avoid human error.

Failure in different control loops with different functionality has different effect on the system. The source of failure could be in the algorithm, in software implementation, in the dependencies, and even in the computing platform, i.e. embedded computer. It is also vital to consider the response time of the control loop, as a delayed response is also considered as a functional failure in control application.

The inner-loop is the most critical control loop in the system, as a failure at this level results in loss of control with a very high probability. However, a proportional-integral-differential (PID) controller, the widely used control algorithm in the inner-loop is fairly simple, deterministic, and easy to implement in software. The effectiveness of the inner-loop heavily rely on the attitude sensor/estimator output and redundant sensors are used to increase reliability. Assuming, the external devices are fault-free, the only possible source of failure in the inner-loop could be interference from other applications or a failure in the computational platform. The position controller is similar to the inner-loop where a PID controller is widely used to compute reference

attitude/thrust control commands to hold or attain a reference position in a three-dimensional space. The position controller itself is reliable, however, its dependency on an external positioning system (e.g. GPS) can reduce its reliability in terms of timely computation of control commands. A failure in the position-controller is not necessarily catastrophic, as a controlled descent is still possible with the inner-loop controller alone; however, an uncontrolled drift may result in a crash of the platform.

The trajectory controller may use a complex algorithms to compute desired trajectories and its software implementation can be hard to analyze for errors. Algorithms like way-point tracking is fairly simple, and a deterministic implementation is possible. However, most of the autonomy comes from the life-long trajectory planning where the trajectory is planned or updated on the go based on the updated information of the surroundings. Such iterative computations may result in memory overflow or infinite looping. Although a failure in the trajectory controller would not necessarily lead to a loss of control, it can significantly interfere with other applications when sharing the same computational platform.

4.0 System Architecture

In this section, we will discuss the architecture of the proposed system and the architectural benefits in the context of isolation. For designing the custom platform architecture, the following assumptions are considered.

1. The software is directly implemented without any operating system or any other software-based protection mechanism.
2. A malfunctioning software subsystem can adversely effect other subsystems sharing the same processor.
3. All subsystems can run on any of the processors.

The design goal is to establish isolation at the hardware level to prevent interference between processes such that no subsystem can interfere subsystems in other partitions or the entire system under all possible conditions to ensure a reliable computation framework for critical control applications. The idea behind this work is to consider a multicore architecture for an agile aerial platform with limited resources. Therefore, the focus is to reduce the footprint of the custom hardware implemented on FPGA thread for possible implementation on very small and low-cost platforms.

4.1 Processor

The system has four processors and each processor in this architecture is an independent system with its own I/O, cache, data memory, instruction memory, heap and stack and no memory is shared between two processors. The complete isolation prevents any kind of interference between the processes running on each processor.

The processor considered in this work is a 32-bit RISC processor implemented with separate 32-bit wide data and 16-bit wide instruction memories. The capacity of the data and instruction memory are configurable and depends upon application needs and available physical resources. The separate instruction and data memory allows it to fetch an instruction in every clock cycle. For simpler implementation of different

width of data and instruction, the processor does not support 'immediate' and all the operands are stored in the data-memory. Each processor has 8 32-bit general purpose registers that are used for computation. The ALU operates on the data stored in the general purpose registers or in the data memory and stores a result in the general purpose registers or in the data-memory. The supported data operations by the ALU are described in Table 1.

Table 1
Supported data operations by the ALU.

Category	Supported Operations by the ALU
Arithmetic	add, subs, mult, ++
Relational	==, !=, >=, >, <=, <
Logical	and, or, xor, neg
Shift	shift between 0 to 31 bit

Each processor has an internal 32-bit counter that can be directly accessed by the ALU with specific instruction. Furthermore, each processor is equipped with an internal watchdog with a 32-bit counter. On a start or reset condition, the program counter of the processor is set to the code entry point of the instruction memory associated to the processor, the general purpose registers in the processors are initialized to zero and the watchdog counter is set to zero.

4.2 Memory

Memory is a critical element for establishing isolation. Erroneous applications can corrupt memory allocated to itself or memory regions of other applications. In this work, the memory device used are on-chip memory only and off-chip memory is out of the scope of this work. The memory is defined in HDL and the synthesis tool targets the on-chip memory blocks on the the FPGA for synthesizing memory on the hardware. The memory shares clock with the processor. The memory blocks allocated to each processor are physically isolated. Application running on a processor can only access the data memory assigned to that specific processor. Furthermore, the separate implementation of data memory and instruction memory helps to prevent corruption of the executable code. The memory writing mechanism of the instruction memory is only accessibly to a program downloading mechanism that is used for writing the executable code in the instruction memory by the user. The user can program and re-program the instruction memory and can read the instructions for verification as a RAM device. However, the processor can only read from the instruction memory, as a ROM device. It is the application developers responsibility to fit the application in the memory allocated to a processor to prevent overflow. The executable code needs to be uploaded separately to each isolated instruction memories, which can be different or same for each processor depending upon the user need.

The memory operations are single cycled, the processor requests data from a memory location and reads the data in the subsequent cycle.

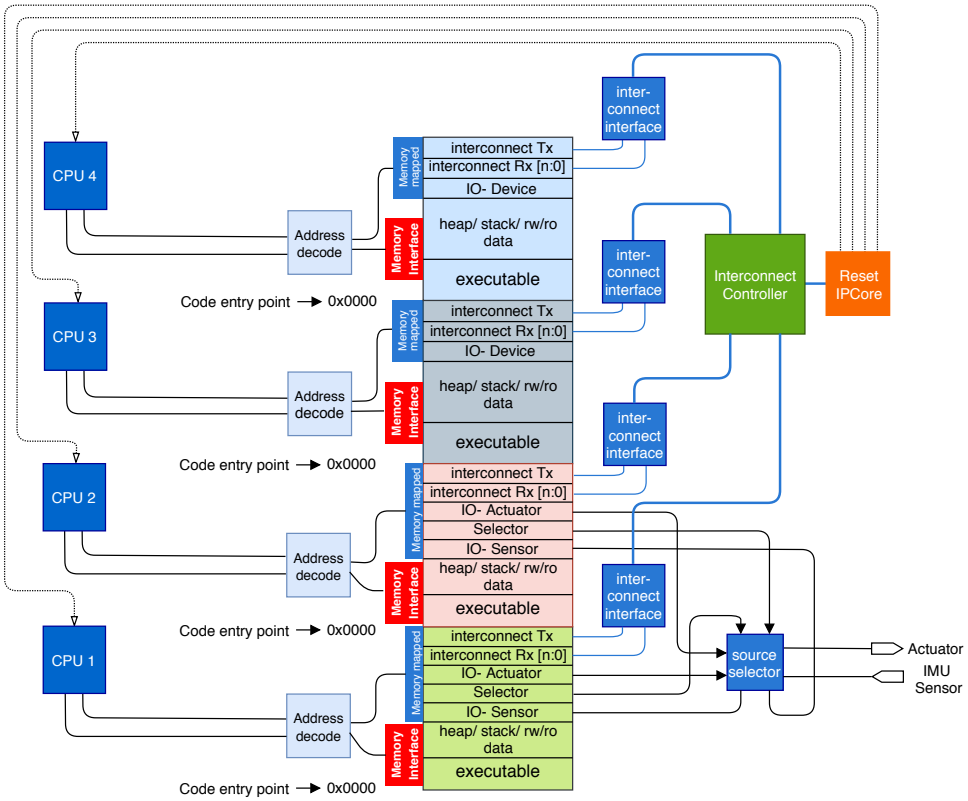

Figure 2. Block diagram of the proposed architecture in a four CPU configuration.

4.3 Peripherals

The access to external devices (sensors and actuators in this case) is restricted to the associated applications only to establish isolation, and a memory mapping approach is adopted to map the data-registers of the IO devices in the memory region allocated to the individual processors as shown in Figure 2. In this configuration, specific addresses from each partition is dedicated to memory mapped devices. The memory-mapped address share the same interface with the data-memory. The read-write request for the memory mapped addresses are bypassed to the IO-devices (IP Cores) by a memory-interface mechanism implemented in hardware by controlling the chip-select line. The physical memory location at the addresses of the memory mapped devices cannot be accessed and remains unused at the upper edge of each memory partition. A processor reads 0, if an IO-device is not present at a dedicated memory-mapped address.

4.4 Reset IPCore

A reset mechanism is necessary to reset the processors when a fault is encountered. In most platforms an universal reset is used to restart all processors and other components. The isolated implementation in this architecture allows implementing an

individual reset mechanism where any particular processor can be reset without affecting the other components in the system. The custom reset hardware is connected to the on-chip interconnect where any processor can send a reset request for itself or other processors. Once a reset request is received for a processor, the reset line of that processor is held active for one clock cycle. This hardware is beneficial for fault recovery as discussed in later sections. The reset IPcore can be a single point of failure for the system, however, it is defined in a simple and analyzable form with a small footprint, therefore, assumed to be reliable.

4.5 Source selection

In a system with multiple producers and consumers, a source control mechanism is essential. A polling based source selection mechanism is established practice for redundant architectures with N-redundant systems. In the proposed architecture with limited resources a polling based source selection mechanism is not effective. The sources selection IPcore controls the data flow between processors and peripherals where more than one source is present. The data flow from sensors to processors is simple, where the IPcore forks the sensor data to multiple processors without any control. The actuators can not handle simultaneous input from more than one source, moreover, simultaneous write from two separate functional sources can result in an unpredictable and erroneous data-write and a flow control mechanism is necessary. The IPCore has an internal counter and four 8-bit registers for each propulsion unit where each processor can write the control command (i.e. thrust command in percentage in this case). The source selection IPcore forwards the command from the default register to the propulsion controller i.e PWM generator in this case. If a fresh command is not received before a timeout period, the IPcore selects the next register as the default register.

4.6 On-Chip Interconnect

As there is no shared memory in this architecture, a message passing approach is taken for inter-processor communication to share data between applications running on separate processors, where application running on one processor can send data as message-packets to a destination processor connected with an interconnect. The interconnect has special mechanism for data-protection, isolation, and real-time end-to-end on-chip communication.

4.6.1 Micro-architecture

The interconnect has two components; an interface for each processor and a controller. The interface is an on-chip memory-mapped device where the transmitting and receiving buffers are independently mapped to the memory region of associated processor. The interface takes two address spaces for transmission, one for the data-packet and the other for destination address, and one address space for each receiving channel. The destination address is the destination processor id followed by the channel id.

Each interface has dedicated sampling buffers for each transmission and reception channel, where the data-packets are held before transmission and stored after reception

for the consumer application to read. The dedicated sampling buffers hold single data-packets from the associated channel unless a new-packet from the same channel over-writes the packet or it gets consumed by the consumer. Each interface has an independent transmission and reception line, a single bit request line, a single bit access line and a single bit active line to connect with the controller. The transmission and reception lines from each interface are connected to the controller with a cross-bar.

The controller controls the communication over the interconnect by providing transmission access to the interfaces. The controller has separate request, access, and active line connecting each interface. The controller is memory-less and the flow through the controller is atomic.

4.6.2 Operation

At the beginning of a transmission, a producer (sending application) writes the message and the destination address at the memory mapped transmission address of the interface. When a fresh data-packet is received at the interface, the interface raises a transmission request to the controller. A round-robin arbitration is implemented in the controller to provide transmission access to each interface by holding the associated access line high. Once access is gained, the interface keeps transmitting until all data-packets are transmitted or access is taken-back by the controller (access line is low). The controller checks for the destination processor address in the header of the data-packets and forwards the packet to the destination interface by controlling the x-bar. The destination interface reads the channel id in the header and holds the message in the associated channel buffer.

Channel specific sampling buffers provide an isolation mechanism; unlike conventional FIFO buffers, overflow in one channel has no affect on the data in other channels. Additionally, the access is controlled by the on-chip controller, dysfunction in an application can not over-load the system, as the controller only forwards the data-packets from an interface, when transmission access is given to the interface.

5.0 Distributed Implementation

This section addresses the distributed implementation strategy of a conventional FCS in the proposed architecture. The design goal is to achieve higher reliability of the entire system to reduce the probability of a loss-of-control failure.

Distributed implementation does not necessarily provide improvements in reliability. In order to achieve improvements we have to provide a reasonably intelligent distribution of functionality on the available processors. In our setup, these functionalities are provided as function calls in a programming language and in the following will be called subsystems. Any functionality can be provided by more than one subsystem, but will then be programmed differently to achieve some degree of independency. Thus, the FCS consists of a number of subsystems, as shown in Figure 2.

5.1 Reliability and criticality

One method for determining the optimal distribution is to assign a reliability function to each functionality and simply pick the distribution that provides the better overall

reliability. In addition to this, we are also interested in optimizing for criticality, that is finding the distribution with the least probability of losing control of the aircraft. Criticality of a subsystem is determined by the possible impact on the system on the event of dysfunction in that particular subsystem. Each subsystem is assigned a criticality that expresses the importance of the subsystem being operational. Importance is to be understood in terms of aviation safety, where failure of a critical subsystem is more likely to lead to loss of the aircraft than failure of a less critical subsystem. For instance, maintaining attitude control of the aircraft is much more important than completing the mission of the aircraft. Subsystem criticality is quantified as a number between 0 and 1, with 0 being non-critical and 1 being very critical for maintaining control of the aircraft.

We impose the following assumptions on these subsystems.

1. Each subsystem is either operational or it has failed.
2. Failure of a subsystem results in failure of the processor it is running on, and thus of all other subsystems running on the same processor.
3. An operational subsystem produces correct results.
4. A known reliability function $R(t)$ is associated with each subsystem.
5. A known criticality index is associated with each subsystem.
6. The number of processors available on the platform are limited and known to the system designer.
7. Two subsystems can have the same functionality, but with different implementations. The subsystems are said to be redundant. It is assumed that redundant subsystems are independent. We will use the term 'functional subsystems' for subsystems with different functionalities.
8. A payload or foreign application software is not analyzed and might have imperfections.

The analysis only includes the algorithmic parts of the FCS. Thus, we assume sensors and actuators to be perfectly reliable.

5.2 Reliability analysis

The reliability $R(t)$ of a subsystem is the probability of non-occurrence of a subsystem failure up until time t. When two subsystem are interdependent, meaning that if one fails the systems composed of the two subsystems also fails, the joint reliability is

$$R_{\text{sys}}(t) = \prod_{k=1}^{n} R_k(t) \ . \tag{1}$$

We say that the subsystems are in series. If two subsystems are independent, meaning that the system composed of the two subsystems will continue to function correctly if either of the two subsystems fail, the joint reliability is

$$R_{\text{sys}}(t) = \left(1 - \prod_{k=1}^{n} (1 - R_k(t))\right) , \tag{2}$$

and we say that the subsystems are in parallel.

As stated in assumption 2 above, when subsystems run of the same core a failure of one subsystem is assumed to render the entire core inoperable. Therefore, any subsystems running on the same core will be in series. Redundant systems, as described in assumption 7, are in parallel. So, if subsystems S_1 and S_2 are redundant implementations of the same functionality, with reliability functions R_1 and R_2, respectively, the joint reliability $R_3(t)$ of the combined systems of S_1 and S_2 is

$$R_3(t) = 1 - \big(1 - R_1(t)\big)\big(1 - R_2(t)\big) \ .$$

For the entire system to operate all non-redundant subsystems must be functional, where a non-redundant subsystem is either a functionality that does not have redundant implementations, or the joint system of a number of redundant subsystems implementing the same functionality. That is, the non-redundant subsystems are all in series.

5.3 Criticality and optimization metric

So, if all subsystems were equally critical for the complete system to function, and no redundancy of subsystems were employed, the resulting reliability would be as given in (1). However, in many instances of subsystem failures an FCS is able to avoid loss of control. If, say, the trajectory planner fails the aircraft may not be able to complete its mission, but the FCS is still able to maintain sufficient control of the aircraft to conduct a safe landing (thus avoiding loss of control). Consequently, we assign a criticality somewhat less than one. However, if, say, the attitude controller fails the likelihood of loss of control is very high, since not controlling the attitude of the multirotor will most likely lead to a crash. Thus, we assign a criticality close to one.

Assume that we have a setup with N cores, and M subsystems with separate functionalities, that is, subsystems each implementing a specific functionality, such as trajectory planner or velocity controller. In addition, we have M_r subsystems that each provide redundancy to one of the M functional subsystems. Define the set of functional subsystems as $\Sigma = \{S_1, \ldots, S_M\}$ and the set of redundant subsystems as $\Sigma_\mathrm{r} = \{S_{M+1}, \ldots, S_{M+M_\mathrm{r}}\}$. The index set of subsystems running on each core is defined as

$$C_k = \{j \mid S_j \text{ runs on core } k\}, \ k = 1, \ldots, N \ .$$

The index set of subsystems providing the same functionality is defined as

$$\rho_m = m \cup \big\{M < j \leq M_\mathrm{r} \mid S_j \in \Sigma_\mathrm{r} \text{ same functionality as } S_m \in \Sigma\big\}, \ m = 1, \ldots, M \ . \tag{3}$$

Also we will need the set of cores on which a specific process runs, that is

$$\Gamma_m = \{k \mid n \in C_k\}, \ m = 1, \ldots, M + M_\mathrm{r} \ . \tag{4}$$

The optimization function will be composed by a

The reliability for a specific core C_j is given as the processes on the core running in series, so

$$R_j^{\text{core}} = \prod_{m \in C_j} R_m \ . \tag{5}$$

When a subsystem S_m is running on more than one core we can either assume that this provides redundancy if we assume that the exact same code running on two different cores is independent, or we can assume that if a subsystem fails on one core, it fails simultaneously on the other core because it is exactly the same code. Those two options are expressed as parallel and serial, respectively.

$$R_m^{\text{mcr}} = 1 - \prod_{i \in \Gamma_m} (1 - R_i^{\text{core}}),\ m = 1, \ldots, M + M_{\text{r}}\ , \tag{6}$$

$$R_m^{\text{mcd}} = \prod_{i \in \Gamma_m} R_i^{\text{core}},\ m = 1, \ldots, M + M_{\text{r}}\ , \tag{7}$$

where 'mcr' means multicore redundancy and 'mcd' means multicore dependency. As for the functional redundancy provided by the processes in Σ_{r}, these are by design parallel to their functional counterparts, and there we define

$$R_n^{\text{ar}} = 1 - \prod_{k \in \rho_n} (1 - R_k^{\text{mcx}}),\ n = 1, \ldots, M\ , \tag{8}$$

where 'mcx' is either 'mcr' or 'mcd'. Finally, the system reliability is given by all functional subsystems being operational simultaneously (possibly through their redundant counterparts). We also insert the criticality index for each functional subsystem as a power on the reliability. This gives the total system reliability as

$$R_{\text{sys}}(t) = \prod_{m=1}^{M} (R_m^{\text{ar}})^{c_m}\ . \tag{9}$$

where c_m is the criticality index of S_m. Note that all R functions in (5) through (9) are dependent on t, but this has been left out for clarity.

By implementing the criticality as a power on the reliability function, criticality is in effect increasing the reliability. A low criticality is thus expressed as a significantly increased reliability, while a high criticality does not increase the reliability by much.

The aim is to find the distribution of subsystem on the cores that maximizes the system reliability. From a aviation safety point-of-view, there is no particular interest in the time variation of the failure rate, only in a overall low rate. This in turn means that the system reliability function should decay as slowly as possible, and given that this function is monotonically decreasing, we will use the integral $\int_0^{t_\infty} R_{\text{sys}}(t)dt$ over a relatively long time period t_∞ (long enough that $R_{\text{sys}}(t)$ has decayed below 0.01) as the optimization metric.

5.4 Implementation

An FCS for an unmanned multirotor consists of a number of subsystems, which may vary slightly from platform to platform. For the purpose of demonstrating the method proposed above, we employ a fairly simply FCS with 5 functional subsystems, listed in Table 2. This table also lists estimated values for constant failure rates, for variable failure rates, and for estimated criticality. Those values are discussed in Section **??**.

To demonstrate the optimal distribution for a given system, assume first that all subsystems have constant failure rates λ_k, that criticality is 1 for all subsystems, and

that there are no redundant subsystems. The reliability function for each subsystem is then $R_k(t) = \exp(-\lambda_k t)$, and the optimal distribution according to $R_{\text{sys}}(t)$ in (9) will then be assigning the $N-1$ subsystem with the $N-1$ biggest λ to the first $N-1$ cores, and all remaining subsystems to the remaining core.

When criticality is no longer all 1, and when the possibility of redundancy of subsystems in introduced, the result is no longer so obvious. Simulation results are presented in Section 7.

Given that the subsystems are in fact software routines it is likely that the failure rate will not be constant. For relatively simple functions such as a complementary filter and attitude controller it is reasonable to assume that if they have been operational for some time there is very little probability of failure. Whereas highly complex functions such as a trajectory planner may still fail well into their life span given the variety of inputs it may receive from different scenarios and different users. Finding true parameters for the failures rate of algorithms implemented in e.g. C code (or similar) is outside the scope of this work. A number of common sources for programs stopping execution is described in Section 6, and while these do regularly occur even in tested programs it is difficult to make a good estimate of how often they occur in a specific code. A substantial amount of literature on the reliability of software (see Section 2). Based on the size of executable code and complexity of operation, we have estimated failure rates for different subsystems as presented in Table 2. However, we don't argue for the correctness of the considered failure rates, instead, the proposed method remains unchanged as long as some estimation of failure rates can be made for the subsystems in use.

Table 2
Subsystem parameters

Subsystem		**Failure rate** λ [1/hour]	**Criticality index** c_m
S_1	Trajectory planner	1/200	0.01
S_2	Full state Kalman filter	1/750	0.80
S_3	Velocity controller	1/6000	0.80
S_4	Complementary filter	1/20000	0.90
S_5	Attitude controller	1/25000	0.99

6.0 Fault Tolerance Analysis

The concept of reliability is deeply connected with the fault tolerance capabilities of the platform. In this section, we discuss the behaviour of the proposed architecture under faulty conditions, and its fault containment capabilities. We have selected three different sources of fault due to poor memory handling, algorithm development, and software implementation where the proposed architecture can be beneficial.

6.1 Fault Injection

To evaluate the response under faulty conditions, common software faults in embedded control applications are asserted in the source code for bare-metal execution. In a bare-metal implementation, where no assistance is available from the operating system, error handling is more challenging. Some frequently encountered faults are memory leak, infinite looping, and communication overloading that we have investigated in this work.

6.1.1 Memory leak and overflow

C and C++ offers dynamic memory allocation where memory can be allocated and re-allocated in run-time, which might cause unpredictable execution, performance degradation, or crash. A memory leak occurs when a memory is allocated but not freed after use and the location cannot be reused. An overflow occurs when the application try to dynamically allocate more memory than the physically available heap size that might or might not be detectable during compilation. In a bare metal system, the application needs to be robust enough to prevent and handle such possible issues. An array with a variable size can be used to enforce a memory leak and overflow condition, with dynamic memory allocation as shown below.

define pointer my_array;
define variable my_array_size;
while *execution loop* **do**
  array_size = user_function();
  my_array = **m_allocate**(my_array_size * **datatype**)
end

6.1.2 Infinite wait and loop

Infinite wait and infinite looping are results of poor algorithm development or poor software implementation that occur when a time-out functionality is not properly implemented and when a break point in a loop is not achieved in an iterative computation. Solver or iterative applications like trajectory planner can be subjected to such faults when an application infinitely computes to achieve accuracy or computes an infinite series, e.g. Taylor series.

6.1.3 Communication overloading

A communication overloading occurs when a producer violates the transmission agreement and transmits more data-packets causing a congestion in the communication system. Such a congestion in the network may result in untimely delivery or dropping of critical data-packets resulting in a system failure. This is reproduced by excessive transmission over the interconnect from a single channel.

6.2 Failure Analysis

The asserted faults in the system has different implication causing dysfunction in the system. Memory leak or memory overflow caused by one application interferes all the applications sharing the same processor; moreover, the architecture does not provide any special feature or technique to detect or handle such malfunction. However, the isolated implementation of the memory with individual processors restricts the faulty application to not corrupt other memory partitions and applications implemented on other processors are unaffected.

Once an application is stuck in an infinite loop or wait, the timely delivery of the result is missed. Once a time-out is reached before the expected message is received, the consumer processor triggers the reset signal of the processor running high level application, which restarts the applications on that processor from the initial condition and executes unless the outlier condition (infinite loop/ wait) is hit again.

Dedicated sampling buffer for each communication channel in the interconnect interface prevents corruption of data packets. On violation of transmission agreement by a channel affects the data packet from that channel only. Moreover, the arbitration mechanism guarantees timely access to all the communicating channel with no stalling or packet-drop when producer follows transmission agreement, although, packet-drop may occur in the violating channel(s).

7.0 Experiment and Results

The experiments in this work can be categorized in two sub-sections. The first is to find an effective distribution of the subsystems in separate processors and the second is to implement the executable in the physical platform for analysis. We consider a platform with four processors contemplating the feasibility of practical implementation on an agile airborne platform.

7.1 Finding Optimal Distribution

The goal of the distribution is to strategically segregate the subsystems described in Table 2 in the four isolated processors to minimize the probability of loss of control. At present, there is no numerical method available to directly calculate the optimal distribution in the context of this work, hence, a graph-search approach is adopted to form and evaluate all possible distributions.

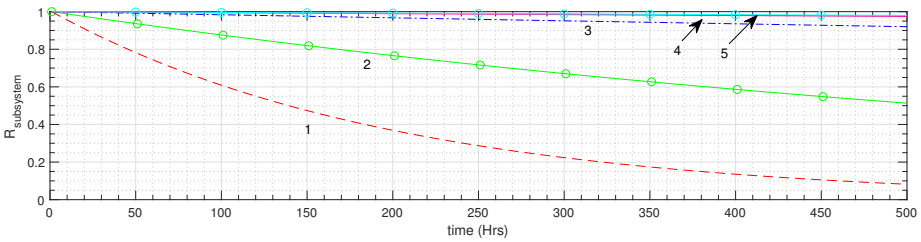

Figure 3. Reliability of individual subsystems as listed in Table 2.

Redundant implementation results in increased reliability, however, full redundancy is not feasible on limited resources. We have considered distribution for two scenarios, one without and one with redundancy. Furthermore, system reliability of distributions based on only the reliability aspect of the subsystem disregarding criticality are compared with distributions considering both aspects in Table 3. A search-tree method is implemented to compute all possible distribution where multiple occurrence of an element is permitted for redundant implementation and restricted otherwise.

Table 3
The distribution with different considerations (redundancy, criticality and reliability) and system reliability, R_{sys}, in each distribution for $t = 0 : 100$ hrs

Consideration			Remark	Distribution				R_{sys}
Red	Cri	Rel		Core1	Core2	Core3	Core4	
no	no	yes	min	s_1	s_2	s_3, s_4	s_5	0.7326
		yes	max	s_1	s_4	s_2, s_3	s_5	0.7326
	yes	yes	min	s_2	s_3	s_4	s_1, s_5	0.7443
		yes	max	s_1	s_2	s_3	s_4, s_5	0.9359
yes	no	yes	min	s_1, s_2	s_1, s_3	s_1, s_5	s_1, s_4	0.4098
		yes	max	s_1, s_5	s_3, s_4, s_5	s_2, s_3, s_4	s_1, s_2	0.9059
	yes	yes	min	s_1, s_2	s_1, s_3	s_1, s_5	s_1, s_4	0.4530
		yes	max	s_2, s_3, s_5	s_3, s_4, s_5	s_1	s_1	0.9960
Single core			n/a	s_{all}	n/a	n/a	n/a	0.7347
Quad-redundant			n/a	s_{all}	s_{all}	s_{all}	s_{all}	0.9865

The exponential correlation of reliability with failure-rate-over-time results in a decaying measure of reliability with time as shown in Figure 3. Furthermore, time-variant failure-rate adds the requirement of analysis over a period, where the time-period depends on the operational lifetime of the system. Hence, to compare the reliability in two different distribution, a single point measurement is not enough. Thus, the integral framework is considered to analyze the reliability of the over-all system consisting subsystems with time-variant reliability.The proposed theorem, described in (9), assigns a score to each distribution based on the relative reliability and criticality of subsystems running on each processor, such that, a higher score represents higher degree of reliability against loosing control of the craft. To evaluate distributions based on reliability alone, the c_m term in (9) is set to 1, that implies that all the subsystems have highest contribution to the overall system, i.e. the system fails if any of the subsystem fails.

Figure 4.*A* and 4.*B* shows system reliability, R_{sys}, with and without redundancy when criticality of subsystems are overlooked. Figure 4.*A.1* and 4.*B.1* represent the highest R_{sys} and Figure 4.*A.3* and 4.*B.3* represents the lowest R_{sys} in the respective distributions. The distributions are listed in Table 3. Figure(s) 4.*x.2* shows R_{sys} in a single core implementation.

The distribution based on both reliability and criticality are evaluated based on the associated reliability and criticality index of each subsystem, as described in (9). The results are plotted in Figure 5.*A* and 5.*B* for distributions with and without re-

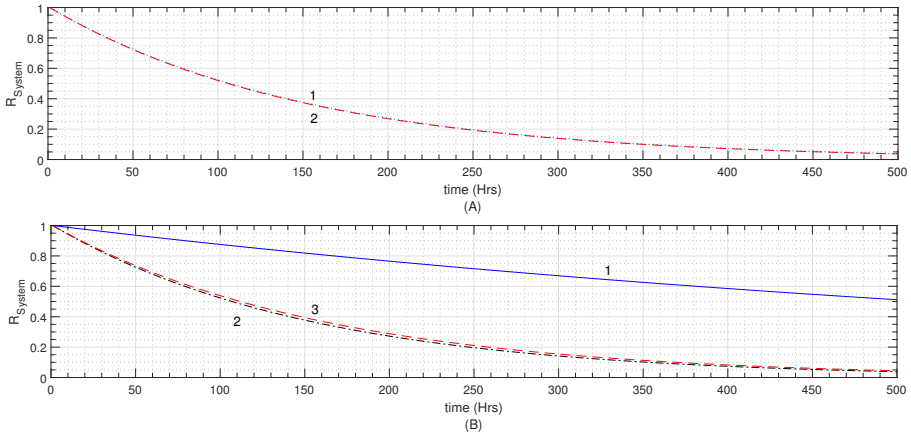

Figure 4. Maximum and minimum system reliability, R_{System}, with time in distribution without redundancy: (A) Distribution without considering criticality (B) Distribution considering criticality.

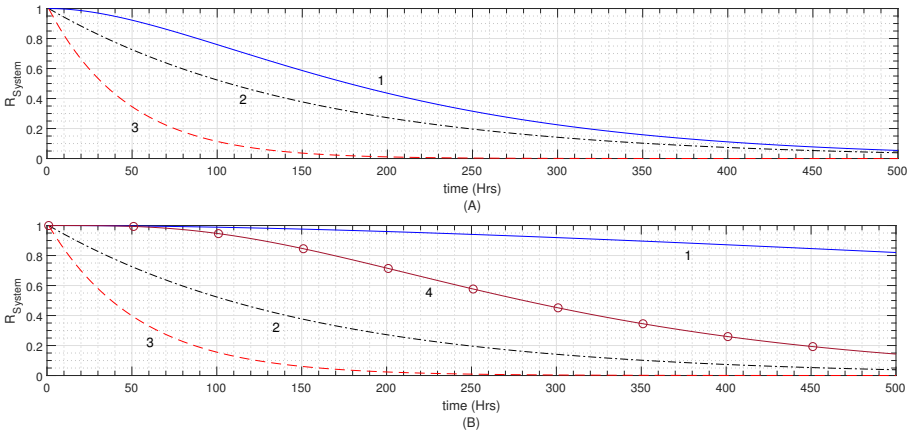

Figure 5. Maximum and minimum system reliability, R_{Sys}, with time in distribution with redundancy: (A) Distribution without considering criticality (B) Distribution considering criticality.

dundancy. Figures 5.A.1 and 5.B.1 represent the highest R_{sys} and 5.A.3 and 5.B.3 represents the lowest R_{sys} in the respective distributions. Figure(s) 5.x.2 shows R_{sys} in a single core implementation and 5.2.4 shows R_{sys} in a quad-redundant implementation.

Table 3 shows the system reliability, R_{sys}, for distributions under different criteria-redundancy, reliability and criticality. Although, reliability is our primary concern and considered in every distribution. The *best* and the *worst* distributions under the considered criteria(s) are marked, and the subsystem distributions are listed. The average R_{sys} for all cases are listed for comparison.

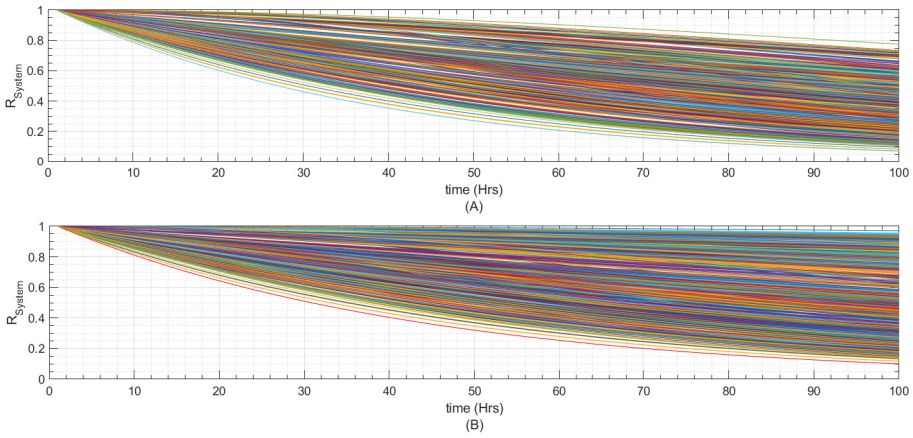

Figure 6. System reliability, R_{Sys}, in all possible distribution with redundancy. Each line represent R_{Sys} in a specific distribution. (A) Distribution without considering criticality (B) Distribution considering criticality.

7.2 Distributed Implementation

Only the distribution with redundancy and criticality is considered for hardware implementation from Table 3. The distribution of subsystems in separate cores and the inter-core message passing for inter-subsystem data sharing is presented in Figure 7 and pseudo code is described in Appendix A.

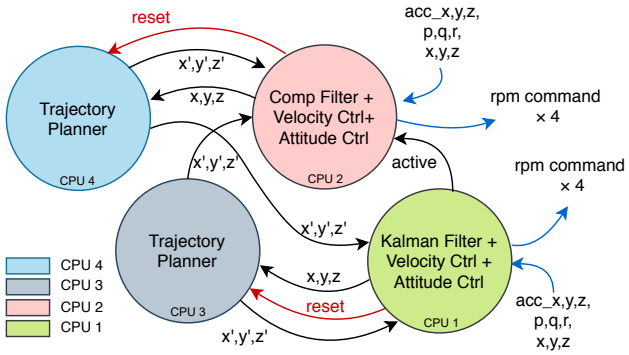

Figure 7. A flow diagram showing the distribution of subsystems in four processors and inter-subsystem communication.

In a conventional N-redundant systems, polling based data-selection is a standard practice. However, in this case, the degree of redundancy is not the same for all of the subsystems (refer Table 3), and for a lower degree of redundancy a polling based mechanism may not be efficient. Instead a sequential data selection method is implemented such that data from a redundant subsystem running on a processor with higher reliability gets preference over the subsystems running on processor with lower

reliability at the consumer end. Note that the processor reliability is subjected to the reliability of all the subsystems running on it. The data from secondary source(s) is considered only if the data is not received before a time-out period is reached or received data is not valid from the preferred source.

In this work, we have assumed that an operational subsystem produces correct results and a malfunctioning subsystem can adversely effect other subsystems sharing the same core. However, in a practical implementation, the assumptions may not be true. A subsystem sharing a core with a faulty subsystem may produce incorrect results. We have considered that a data validation mechanism is implemented at the receiving end that has the capability of detecting incorrect data. Once the consumer subsystem receives a erroneous data, it treats the transmitting core as not operational. The custom reset mechanism explained in Section 4.4 is used for a smart watchdog implementation where a consumer subsystem of higher criticality can reset a malfunctioning producer with lower criticality as shown in Figure 7.

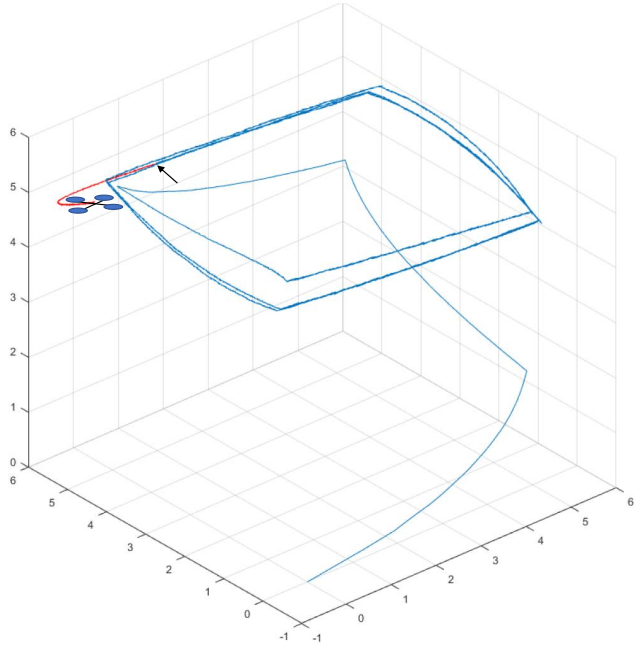

Figure 8. Simulation results shows controlled flight after failure of core 1,3 and 4 at t = 300 s. The time of failure is pointed with an arrow.

The physical implementation is done on an Intel Cyclone V FPGA platform with a 50 MHz oscillator. The platform architecture is set up with four processors with isolated instruction and data memory as presented in Figure 2. All the components are written in Verilog HDL and synthesized with Intel Quartus-Lite tool. For hardware-in-loop experiments the hardware platform is interfaced to a MathWorks Simulink simulation platform, that simulates a quadrotor dynamics, through UART connectors.

The simulation framework transmits simulated sensor measurements to the Intel Cyclone board and the controller running on the boards computes control commands

based on the sensor measurements and sends back to the simulation platform. The simulator only simulates the plant (quadrotor) dynamics based on the control inputs and has no contribution in the controlling of the plant.

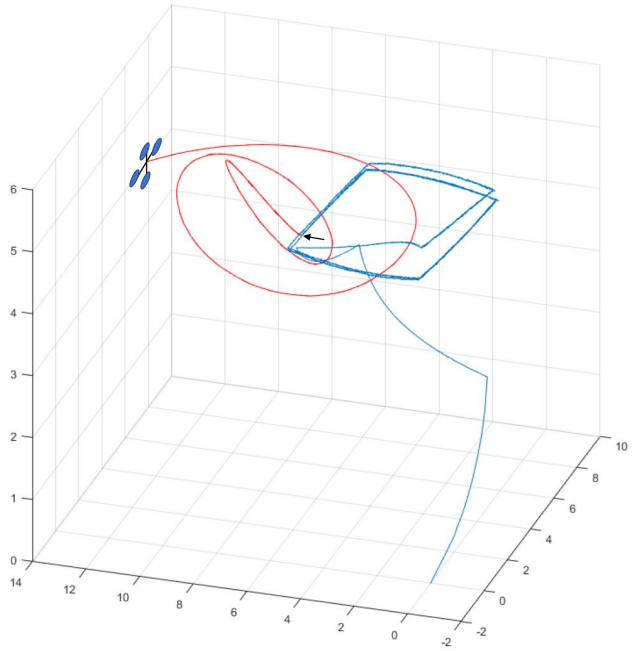

Figure 9. Simulation results shows loss of control after failure of core 1, 2, 3 and 4 at t = 300 s. The time of failure is pointed with an arrow.

Four switches on the Intel Cyclone V board are used to individually turn off the individual cores to analyze the impact on the operation of the plant. For a broader range of experiment, a script is used to turn off cores based on the reliability index of the core, i.e. the product of reliability of the subsystem sharing the core. Two simulation results are shown in Figure 8 and Figure 9; Figure 8 shows the flight trajectories after failure of core 1, 3, and 4 and Figure 9 shows flight trajectories after failure of all four cores. Note that the plant is controllable even after failure of 3 cores as shown in Figure 8. In both cases, fault is injected at $t = 300$ seconds and response is monitored for next 300 seconds, which is a reasonable period for safe landing.

7.3 Discussion

The results presented in Table 3 shows the evaluation of different distribution under different criteria. Ignoring the criticality index implies that each subsystem has equal impact on the overall system, hence, lowers the overall reliability factor. Consideration of criticality index gives a more realistic distribution analysis, considering different impact on overall system from different subsystem.

From the results presented, it is evident that a poor distribution can significantly degrade the reliability of the system, even resulting in worse reliability than a single

core implementation.

8.0 Conclusion

In this work we have demonstrated a distributed implementation of a safety-critical system in an asymmetric multi-core platform with limited resources. The outcome of the experiment shows that consideration of criticality along with reliability of the individual subsystems is beneficial to minimizing the probability of system failure (loss of control). Additionally, the results show that a strategic implementation of subsystems with different level of criticality on a shared hardware can improve the overall system reliability, where system level redundancy is not feasible due to the limitation of resources. Furthermore, such a distributed implementation is free from the complexities of multi-threading, while taking advantages of a multi-core architecture.

The proposed method is scalable to more cores and more subsystems, though the search time for the optimal distribution using a graph search grows rapidly. The work can also be extended to subsystem dependencies where dependent subsystems can be implemented together to reduce inter-core communications.

9.0 Acknowledgement

The authors would like to thank Henrik Schiøler from department of electronic systems, Aalborg University for his insightful comments and helpful discussion.

REFERENCES

1. M. W. MUELLER and R. D'ANDREA Stability and control of a quadrocopter despite the complete loss of one, two, or three propellers, *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, **doi: 10.1109/ICRA.2014.6906588**, ISSN 1050-4729, pp 45-52.
2. ULBRICH, PETER and HOFFMANN, MARTIN and KAPITZA, R and LOHMANN, D and PREIKSCHAT, W and SCHMID, R Eliminating single points of failure in software-based redundancy, *Proceedings - 9th European Dependable Computing Conference, EDCC 2012*, 2012, **doi: 10.1109/EDCC.2012.21**, ISSN 1050-4729, pp 45-52.
3. D. GIZOPOULOS and M. PSARAKIS and ADVE, S and RAMCHANDRAN, P and HARI, S. K. and SORIN, D and MEIXNER, A and BISWAS, A and VERA, X Architectures for online error detection and recovery in multicore processors, *Design, Automation & Test in Europe Conference & Exhibition*, 2011, **doi: 10.1109/DATE.2011.5763096**, ISBN 978-3-9810801-7-9.
4. BOLCHINI, C and MIELE, A and SCIUTO, D An adaptive approach for online fault management in many-core architectures, *2012 Design, Automation & Test in Europe Conference & Exhibition*, 2012, **doi: 10.1109/DATE.2012.6176589**, ISBN 978-1-4577-2145-8, pp 1429–1432.
5. AGGARWAL, NIDHI and RANGANATHAN, PARTHASARATHY and JOUPPI, N and SMITH, J. E Configurable isolation: building high availability systems with com-

modity multi-core processors, *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, **doi: 10.1145/1273440.1250720**, ISBN 978-1-59593-706-3, pp 470.

6. Döbel, Björn and Härtig, Hermann and Engel, M Operating system support for redundant multithreading, *Proceedings of the tenth ACM international conference on Embedded software - EMSOFT '12*, 2012, **doi: 10.1145/2380356.2380375**, ISBN 9789066052079, pp 83.
7. Lo, Moustapha and Valot, Nicolas and Maraninchi, F and Raymond, P IMPLEMENTING A REAL-TIME AVIONIC APPLICATION ON A MANY-CORE PROCESSOR, *42nd European Rotorcraft Forum (ERF), Sep 2016, Lille, France. ffhal-01718139f*, 2016.
8. Henkel, Jörg and Bauer, Lars and Dutt, N and Gupta, P and Sahi, N and Muhammad, S and Mehdi, T and Wenn, N Reliable on-chip systems in the nano-era, *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*, 2013, **doi: 10.1145/2463209.2488857**, ISBN 9781450320719, pp 1.
9. Alhakeem, M S and Munk, P A Framework for Adaptive Software-Based Reliability in COTS Many-Core Processors, *ARCS 2015 - The 28th International Conference on Architecture of Computing Systems. Proceedings*, 2015, ISBN 9783800736577, pp 1-4.
10. Huang, Lin and Xu, Qiang Characterizing the lifetime reliability of many-core processors with core-level redundancy, *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2010, **doi: 10.1109/ICCAD.2010.5654250**, ISBN 9781424481927, pp 680–685.
11. Finkelstein, Maxim. Failure Rate Modelling for Reliability and Risk chapter 9, 2008, Springer London.
12. Song, Kwang and Chang, In and Hoang, P A Software Reliability Model with a Weibull Fault Detection Rate Function Subject to Operating Environments, *ARCS 2015 - The 28th International Conference on Architecture of Computing Systems. Proceedings*, 2017, **doi: 10.3390/app7100983**, Publisher: Springer London, pp 983.
13. Zhang, Xuemei. and Pham, Hoang Software field failure rate prediction before software deployment, *Journal of Systems and Software*, 2006, **doi: 10.1016/j.jss.2005.05.015**, ISSN 01641212, pp 291–300.
14. Littlewood, Bev and Strigini, Lorenzo Software reliability and dependability, *Proceedings of the conference on The future of Software engineering - ICSE '00*, 2000, **doi: 10.1145/336512.336551**, ISBN 1581132530, pp 175–188.
15. O'Regan, Gerard Software Reliability and Dependability, *Software failures*, 2017, **doi: doi.org/10.1007/978-3-319-64021-1**$_2$, $ISSN$01641212, pp983.

10.0 Appendix A

The appendix describes the software implementation of the subsystems in 4 separate processors as shown in Figure 7. Each subsystem is presented as a function call with

the required arguments. The *read* and *write* functions in the pseudo code are methods to read and write memory mapped address spaces, denoted with parameters starting with '*'. Such memory mapped addresses are interfaced to external devices and on-chip communication network and can be accessed from the processor by reading or writing at the associated address.

Pseudo code for **Processor 1**

```
while continuous loop do
    IMU_data = read(*IMU_IPCore);
    actual_position = read(*MotionCapture_IPCore);
    reference_position = read(*data_from_cpu_3);
    if reference_position_received then
        reset(timer);
    else
        reference_position = read(*data_from_cpu_4);
        if timer ≥ timeout then
            reset(cpu3);
        else
            timer++;
        end
    end
    estimated_attitude = Kalman_filter(IMU_data);
    reference_attitude = velocity_controller(reference_position, actual_position);
    thrust_command = attitude_controller(reference_attitude,
     estimated_attitude);
    write (cpu_1_status_ok to *cpu_2);
    write (actual_position to *cpu_3);
    write(thrust_command to *motors);
end
```

```
Pseudo code for Processor 2
while continuous loop do
    IMU_data = read(*IMU_IPCore);
    actual_position = read(*MotionCapture_IPCore);
    reference_position = read(*data_from_cpu_4);
    if reference_position_received then
        reset(timer);
    else
        reference_position = read(*data_from_cpu_3);
        if timer ≥ timeout then
            reset(cpu4);
        else
            timer++;
        end
    end
    estimated_attitude = complimentary_filter(IMU_data);
    reference_attitude = velocity_controller(reference_position, actual_position);
    thrust_command = attitude_controller(reference_attitude,
     estimated_attitude);
    cpu_1_status = read(*data_from_cpu_1);;
    if cpu_1_status_ok then
        reset(timer);
    else
        if timer ≥ timeout then
            write(thrust_command to *motors);
        else
            timer++;
        end
    end
    write (actual_position to *cpu_4);
end
```

Pseudo code for **Processor 3**
while *continuous loop* **do**
    actual_position = **read**(*data_from_core1);
    reference_position = **trajectory_planner**(actual_position);
    write (reference_position **to** *cpu_2);
    write (reference_position **to** *cpu_1);
end
Pseudo code for **Processor 4**
while *continuous loop* **do**
    actual_position = **read**(*data_from_core2);
    reference_position = **trajectory_planner**(actual_position);
    write (reference_position **to** *cpu_1);
    write (reference_position **to** *cpu_2);
end