applied sciences

MDPI

Article

On-the-Fly Repairing of Atomicity Violations in ARINC 653 Software

Eu-teum Choi [1], Tae-hyung Kim [1], Yong-Kee Jun [2,*], Seongjin Lee [3,*] and Mingyun Han [2]

[1] Department of Informatics, Gyeongsang National University, Jinju-daero 501, Jinjusi 52828, Korea; etchoi@gnu.ac.kr (E.-t.C.); miewcs2@gnu.ac.kr (T.-h.K.)
[2] Department of Aerospace and Software Engineering, Gyeongsang National University, Jinju-daero 501, Jinjusi 52828, Korea; wanye@gnu.ac.kr
[3] Department of AI Convergence Engineering, Gyeongsang National University, Jinju-daero 501, Jinjusi 52828, Korea
* Correspondence: jun@gnu.ac.kr (Y.-K.J.); insight@gnu.ac.kr (S.L.); Tel.: +82-055-772-1378 (S.L.)

Abstract: Airborne health management systems prevent functional failure caused by errors or faults in airborne software. The on-the-fly repairing of atomicity violations in ARINC 653 concurrent software is critical for guaranteeing the correctness of software execution. This paper introduces RAV (Repairing Atomicity Violation), which efficiently treats atomicity violations. RAV diagnoses an error on the fly by utilizing the training results of software and treats to control access to the shared variable of the thread where the error has occurred. The evaluation of RAV measured the time overhead by applying methods found in previous works and RAV to five synthesis programs containing an atomicity violation.

Keywords: airborne software; health management; on-the-fly repairing; atomicity violations

Citation: Choi, E.-t.; Kim, T.-H.; Jun, Y.-K.; Lee, S.; Han, M. On-the-Fly Repairing of Atomicity Violations in ARINC 653 Software. *Appl. Sci.* **2022**, *12*, 2014. https://doi.org/10.3390/app12042014

Academic Editors: Cheng-Wei Fei, Zhixin Zhan, Behrooz Keshtegar, Yunwen Feng

Received: 7 December 2021
Accepted: 11 February 2022
Published: 15 February 2022

1. Introduction

As the technology of avionics is advancing, the number of software components for an aircraft is also rapidly increasing. In 1960, there were only 8% of software components on F-4. In 2007, however, about 90% of the features on F-35 were enabled by software [1]. As the number of software components increases on an aircraft, the complexity of airborne software system increases and extends the scope of software. As a result, the probability of latent errors in airborne software is significantly increased.

The concurrency error is one of the well-known errors of software [2–8] and it is famous for its irreproducibility, which makes it very difficult to debug the error [4–8]. In a real-world application, about 70% of all concurrency errors are caused by atomicity violations [3]; it is a concurrent execution of a specific code region, i.e., critical section, that unexpectedly violates that region's atomicity. Not only it is a time-consuming trial-and-error process to find the atomicity violation of a program, but it is also almost impossible to test all the probable execution scenarios of a program. Thus, any concurrent code may violate atomicity at some point during the lifetime of software.

Health management systems (HMSs) handle the errors on airborne software [9–15]. The role of HMSs is to diagnose, isolate and treat the errors, so that the side effect of the error can be contained and airborne software can continue serving its purpose [14]. There are two notable works [16,17] reporting methods for the on-the-fly repairing of atomicity violations in airborne software. They both monitor access events in ARINC 653 software to diagnose the lock-usage violations of threads. Once an atomicity violation is detected, the error is treated either by inserting a lock [16] or stalling the execution of a critical section [17].

Ha et al. [16] and Tchamgoue et al. [17] exploited a diagnosis protocol [4] that compared real-time access information with access history created and maintained at every

access event to diagnose atomicity violations. However, the protocol they used has a time complexity of $O(T)$ on each test of atomicity violation, where T denotes the maximum number of threads that can be executed on a program. Because the complexity of airborne software is increasing, their approach is practically inefficient in real-time systems operating on an aircraft.

This paper introduces Repairing Atomicity Violation (RAV) that repairs atomicity violations on the fly based on the correct order of execution acquired from the pre-execution of the program. RAV monitors the share of the resources of each thread to diagnose the atomicity violations and treats the violation by exploiting the condition variable on code segments. RAV inserts `wait()` to delay the conflicting code segment and invokes `signal()` after the correct order of execution is restored. Section 2 describes the avionic health management system and atomicity violations as background. Section 3 introduces approaches for on-the-fly repairing of atomicity violations in general-purpose platforms and avionics. Section 4 explains the structure and the design of the proposed RAV. Section 5 describes and analyzes the experimental environment and the results from the evaluation. Section 7 concludes the paper.

2. Background

This section describes the avionic health management system and atomicity violations, a type of concurrency error. In addition, this section describes the on-the-fly repairing of atomicity violations.

2.1. Avionic Health Management System

Airborne software is embedded software that monitors, controls and manages the state of an airborne system. Since the early 1980s, the number of airborne software types to use software on aircrafts and in equipment used on aircrafts and their engines has been rapidly increasing. As the importance of airborne software grows in the development of aircraft functions, the complexity of software increases. As a side effect, the probability of potential errors also increases. Potential errors can lead to the failure of aircraft systems during the operation of the aircraft. As a representative example, the first test flight of the F-22 Raptor in 1992 caused a crash due to a control software error that failed to prevent pilot-induced oscillation [10]. There were two crashes, on a Boeing 737 MAX in October 2018 and March 2019, due to an error in which flight control software pushed the nose down with an incorrect value read from one sensor. The incident led to the death of a total of 345 people [18].

It is vital to prevent software errors in aircraft because accidents caused by airborne software cause loss of life and property. When developing safe airborne software, it is common practice to make software comply with RTCA's DO-178C, which is a standard of airborne software development [19]. The DO-178C strives to eliminate software faults by making about 60% of its objectives defined in the verification and validation (V&V) process to identify and minimize software faults. However, even if software strictly follows through a rigorous V&V process, potential faults can still exist in airborne software [20].

Software fault tolerance [20] is applied to prevent accidents from occurring due to software faults because no one can eliminate all faults during the development process due to the developer's mistakes or functional limitations of the error detection tools. Software fault tolerance makes it possible for software to partially or fully function normally even if errors or failures occur in some of the software components that make up the system. Representative techniques for software fault tolerance are redundancy and voting. It has to be noted that these techniques do not provide adequate coverage for problems such as common-mode faults and latent design bugs. Therefore, errors or failures must be resolved during aircraft operation to provide resilience for faults [9].

The purpose of avionic health management systems (HMSs) [9–15] is to prevent airborne software systems from failing. An HMS monitors, diagnoses and treats errors while the system operates. In the monitoring stage, it makes a log of any errors and related

events. In the diagnosis stage, it exploits the acquired information to categorize error types and predict probable errors in the system [14]. Finally, the treatment stage treats errors using online approaches to resolve faults raised during program execution or offline approaches that modify the source code by analyzing the log saved after correct execution.

HMSs are currently applied to ARINC 653 [14] and Future Airborne Capability Environment (FACE) [21]. ARINC 653 is an integrated modular avionic (IMA)-based real-time operating system (RTOS). It is an avionic system that integrates and operates an environment where multiple computer systems are operated in a distributed manner. Therefore, if an error occurs in an airborne system without an HMS, the entire system may fail due to one error. ARINC 653 provides health monitoring to detect and recover hardware and software errors at the process, partition and module levels. It isolates errors and prevents failures from propagating to other systems within the IMA. ARINC 653 provides a health monitor configuration table and error handler process for managing errors. In the health monitor configuration table, the level of error is defined as a process, partition and module and the recovery method for each error is specified to enable action appropriate to the situation. In the case of a process-level error, the HMS can call an error handler defined by a user.

The FACE [21] is an open architecture for the development of portable airborne software components targeting general-purpose, safety and security-purpose usages. The FACE consists of five local segments, such as operating system segment (OSS), input/out services segment (IOSS), platform-specific services segment (PSSS), transport services segment (TSS), portable components segment (PCS). The FACE provides OS-level health monitoring and fault management (HMFM) and system-level HMFM for HMSs. The The OS-level HMFM offers a standardized method for detecting, reporting and handling errors and failures within the scope of a single system or a platform. The OS-level HMFM detects and handles errors and faults during run-time at the process (i.e., thread), partition and module (i.e., platform) levels. The role of the system-level HMFM is to monitor faults and failures in systems and applications and report them. It can also support a repair option that allows the system-level HMFM to repair a failed or defective resource. The system-level HMFM can monitor or manage instantiation and termination of components. The system-level HMFM may generate alarm and notification to indicate internal state transitions of components.

2.2. Atomicity Violations

An atomicity violation is an error in parallel programming that fails to execute the atomic region atomically and runs in an unexpected order. It commonly occurs when two or more threads race against performing a write operation on the same shared variables without a proper synchronization mechanism protecting the variable [2,3,22]. Figure 1 shows the source code of an atomicity violation in the I2C (inter-integrated circuit) communication of Ardupilot. It shows the I2C device driver implemented in the hardware abstract layer to support the Pixhawk board in the Ardupilot. If two Pixhawk devices with the same name call *init*, the *init_ok* variable can be falsely assigned. Falsely assigned variables may cause the SMBus battery to fail to initialize. Even if the developer has sufficient knowledge and understanding of the development of a concurrency program, there may be potential atomicity violations in the code because one cannot consider all the program's execution paths. Additionally, even if a phenomenon of atomicity violations is discovered during an execution of a program, it is not easy to reproduce it again because there are numerous different types of interleaving [2].

thread1	thread2

`uint8_t PX4::PX4_I2C::instance;`
`...`
 	
`bool PX_I2C::do_transfer(...)`
`{`
`...`
`if(!init_done) {`
`...`
`if(init_ok) {`
`instance++;`
`}`
`...`
`}`
`...`
`}`
	`uint8_t PX4::PX4_I2C::instance;`
	`...`
	
	`bool PX_I2C::do_transfer(...)`
	`{`
	`...`
	`if(!init_done) {`
	`...`
	`if(init_ok) {`
	`instance++;`
	`}`
	`...`
	`}`
	`...`
	`}`

Figure 1. An example of atomicity violations (Issue #7129) in Ardupilot.

One may carefully write a code to prevent all the errors one might think of; however, even in such a case, there is a high probability of unexpected errors in the code. For example, both the first shuttle flight and the 44th flight of NASA's Advanced Fighter Technology Integration (AFTI) F-16 software exhibited issues associated with redundancy management [23]. The first shuttle flight was stopped 20 min before the scheduled launch because of a race condition between the two software versions. Open-source programs such as Apache, which has about 2500 committers and authors [24] in total, working on the project monthly, still have unresolved atomicity violations in the code [3].

Even experienced programmers with sufficient background in parallel and concurrent programming cannot write a code considering all the facets of exceptions. Thus, there is an atomicity violation in the code with a very high probability. There are two approaches in the field of atomicity violations. The first focuses on the use of a detection tool to identify the error and its cause before the system is deployed. The second focuses on repairing the software error on the fly to prevent the system from failing.

Detection tools can be categorized into static and dynamic analysis tools. Static analysis tools [25–27] inspect for all the possible thread interleaving types within the source code of software to identify the atomicity violation. Since static analyses do not have the knowledge of how software would operate, it counts improbable interleaving. As a result, static analysis tools have high false alarm rates and have low reliability. Dynamic analysis tools [4–7,28–31] detect atomicity violations by tracking, replaying and watching software; however, their overhead is large and false alarm is also present. It is not possible to identify and fix all the atomicity violations in software. There are some debugging tools to detect and debug atomicity violations and concurrency errors; however, there are few to choose from. The high learning curve of the tools makes it even more challenging to adopt the program to detect the bugs in the code. Lu et al. [3] claimed that debugging concurrency errors on a multi-threaded program is about 17% more time-consuming than debugging errors such as memory leaks in a sequential program. They also showed that 46% more files were related to concurrency errors than a sequential program and 72% more patches were generated to fix the bug. However, after going through such tedious endeavors, about 39% of the patches to fix the concurrency errors were incorrect.

It is most crucial to prevent software from experiencing atomicity-violation-related failures that have survived the inspection of detection tools and manifest themselves in the operation. There are two types of repairing approaches, backward recovery and forward recovery [15]. Backward recovery makes use of checkpoints to recover to the last known correct state. Although it can fix all unexpected errors, it suffers from large time and space overheads. There are many different ways to achieve forward recovery of failures. In terms of time and space overheads, forward recovery is efficient. Since backward recovery may not meet the real-time constraints because of its protocol, it is necessary to apply forward

recovery in mission-critical real-time applications such as airborne software. However, the forward recovery approach depends on the execution scenario. Thus, it has to be tailored to each program. Even in such scenarios, it can handle only the predictable errors in the system. Making use of software version redundancy and voting can also be a solution to the problem; however, it also increases the space complexity of the software and it is hard to identify the root cause of the failure. Moreover, if the system fails due to the violation, these approaches fail together. There are methods that exploit the synchronization mechanism or logging to find the root cause of the failure. Note that adopting them may increase the time complexity of the repairing tool.

Table 1 summarizes the research studies on atomicity violations in airborne software which have been initiated since 2010. Research on types of analyses has mainly focused on design considerations and identification of the errors in software [32,33]. There are no static analyses specific to airborne software because it inspects the source code. There are a few research studies on dynamic and mixed analyses targeted to airborne software [31,34,35]. In theory, there are backward and forward recovery schemes. However, research studies have been focused on forward recovery only [16,17] and the most recent work dates back to 2011. We can see that the research community has focused on the detection of the errors but not on the repairing of software. It is a fact that not all errors can be fixed in software. Thus, the repairing of software is as important as the detection of errors to withstand the failures that manifest during operation.

Table 1. Research studies on atomicity violations in airborne software.

Years	Analysis	Detection			Repairing	
		Static	Dynamic	Mixed	Forward	Backward
2011∼2020	Domingues et al. [32], Kim et al. [33]	-	Cheptsov et al. [31]	Singh et al. [34]	Tchamgoue et al. [17], Ha et al. [16]	-
2021∼	-	-	Singh et al. [35]	-	-	-

2.3. On-the-Fly Repairing of Atomicity Violations

Researchers have proposed on-the-fly repairing of atomicity violation techniques for real-world software to prevent system failure [36,37]. In general, these techniques can be classified in diagnosis phase and treatment phase. In the diagnosis phase, the faults or anomalies of the system are detected with error diagnosis approaches, such as detection protocols or correct interleaving information [36–38]. The detection protocol monitors the information of shared variables executed in each thread during execution and analyzes correlations to diagnose the occurrence of atomicity violations. The algorithm for diagnosing atomicity violations checks the locking discipline or compares the order relationships among shared variables. The detection protocol uses access history to store the information required by the detection algorithm. The access history is a data structure to store a set of threads, synchronizations and shared variables. Some detection algorithms include time information in this set. However, the access history causes high time and space overheads to maintain information during executions.

The diagnosis phase compares the correct interleaving information acquired during correct interleaving provided at the development stage with data obtained in actual interleaving during the execution to diagnose the occurrence of atomicity violations. This technique is suitable for application in modern systems, since it has lower time and space overheads than detection-protocol-based techniques and provides higher accuracy in diagnosing atomicity violations.

The correct interleaving information can be provided by developers or through testing. The developer can directly specify additional execution information as an assertion or comment near the shared variable in the source code during the development stage. To diagnose an atomicity violation, we compare the programmer-specified information in the

source code with the actual execution information. This method has disadvantages. An inexperienced developer may omit the defects or gather incorrect execution information.

Once the software development process is finished, the posthumous methods for testing collects the correct interleaving through numerous repeats of the developed software. Then, following the suit of the previous way, the atomicity violations are diagnosed by comparing the correct interleaving with the actual execution information. This method has the advantage of collecting relatively accurate correct interleaving. However, there may be cases where an incorrect interleaving is collected during testing.

When an error is detected in the diagnosis phase, the treatment phase handles the error based on the type of error. There are two types of approaches for treatment, forward and backward recovery. Forward recovery avoids the faults by inserting synchronization or changing the priority of thread scheduling. Backward recovery re-executes the program by returning to the line of the code immediately before the error occurred, that is, to the line of the code where the last operating state shows no system faults.

Due to its relatively low overhead, forward recovery is performed for repairing access faults. Thus, it employs various methods to implement synchronization, such as inserting condition variables, inserting time-out and inserting locks. The method of inserting condition variables, which changes thread scheduling, puts the thread in the waiting state before the access event of the shared variable in which the error occurred. Then, it resumes the execution of the waiting thread after processing with the shared variable access of other thread capable of correct interleaving. The time-out method allows the execution timing of the threads to be changed by using delaying techniques (e.g., using sleep() or yield()) and leads to fixed-time overheads as delayed times. However, it is crucial to use appropriate latency to handle access faults because the errors may not be fixed by applying low latencies. Conversely, applying high latencies may lead to high-cost overheads. The method of inserting locks is more suitable for repairing atomicity violations.

3. Related Work

This section describes approaches for the on-the-fly repairing of atomicity violations in general-purpose platforms and avionics. The repair of atomicity violations has mainly been researched in old, high-user and complex real-world applications, such as MySQL, Apache and Chrome [3], to capture various types of error patterns. It has not been actively conducted in airborne software because it is closed for security reasons.

3.1. General-Purpose Platform

Atomicity violations have been around for a long time in general-purpose platforms that we often use. The occurrence information of atomicity violations is shared among developers through each software's bug report. This information provides valuable patterns (or types) for research on on-the-fly repair approaches [36–38] to prevent system failure during operation. Since there is a wealth of information on atomicity violations in the bug report, most of the on-the-fly repairing research studies have been conducted on general-purpose platforms. It has to be noted that airborne software is hard real-time software. However, the software of a general-purpose platform does not require hard real-time.

Considering the accuracy and overhead of existing research studies performed on a general-purpose platform, the method by J. Yu et al. [36] and AI (anticipate invariant) [37] are suitable for use in real-time systems. These two research studies are similar in terms of the methods for diagnosing errors using the pre-test information and treating errors by stalling the thread where the error is diagnosed.

J. Yu et al. [36] defined the order of each static memory operation in a data structure as *PSet* (predecessor set). The collection conditions of *PSet* are as follows: (1) When there are two static memory operations, P and M, at least one between P and M is a write access; (2) P and M are executed in two different threads; (3) M is executed immediately after P is executed. AI [37] defines the order for each static instruction as *BSet* (belonging set). The *BSet* collects static instructions (S_x) that satisfy the following conditions based on a

dynamic instruction (D_x): (1) It accesses the same address as D_x; (2) it is executed in another thread, to which D_x does not belong; (3) S_x, accessed just before D_x, is executed and stored. The methods of these two research studies are similar in that they store access information from different threads. However, J. Yu et al. [36]'s method does not store access information in *PSet* if the previous access was executed in the same thread. On the other hand, AI stores the access information executed in a different thread even if the previous access was executed in the same thread. This difference makes the coverage of errors that AI can repair larger than that of J. Yu et al.'s method.

Figure 2 shows two types of interleaving that can occur in Figure 1. In Figure 1, the developer would have intended $instance++$ to be executed atomically, as shown in Figure 2a. If this program is correctly executed in the testing phase, as shown in Figure 2a, since only read access $R3$ of $thread2$ satisfies the collection condition of *PSet*, the *PSet* corresponding to this interleaving is stored as a correct interleaving in Figure 2c. This interleaving information is used when diagnosing the occurrence of atomicity violations in comparison with actual interleaving in the operation phase. Figure 2b is the interleaving of atomicity violations that may occur in this program. If this program is incorrectly executed in the operation phase, as shown in Figure 2b, *PSet* is collected as shown in the atomicity violation of Figure 2c. If the actual execution is as shown in Figure 2b, J. Yu et al.'s method repairs the atomicity violation using the following steps: First, when read access $R1$ of $thread1$ executes, it is compared with the $PSet(R1)$ of the correct interleaving. It is not diagnosed as an error, since the $PSet(R1)$ of the correct interleaving and the $PSet(R1)$ of the actual interleaving are the same. Next, when the read access $R3$ of $thread2$ executes, it is compared with the $PSet(R3)$ of the correct interleaving. It is diagnosed as an error, since the $PSet(R3)$ of the correct interleaving and the $PSet(R3)$ of the actual interleaving are different. At this time, J. Yu et al.'s method performs stalling before the read access $R3$ of $thread2$ is executed to delay the read access $R3$ until the write access $W2$ of $thread1$ comes to repair the atomicity violation.

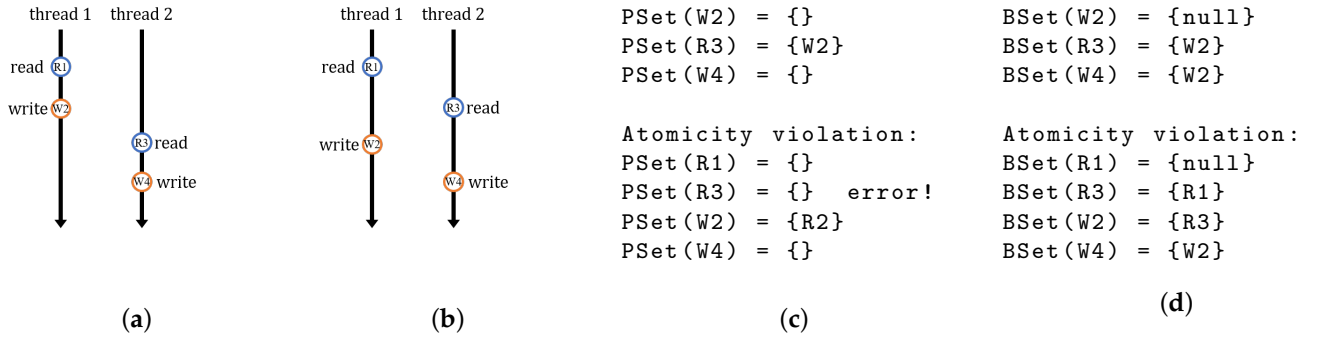

Figure 2. The result of PSet and BSet of interleaving. (**a**) Correct interleaving. (**b**) Atomicity violation. (**c**) *PSet* [36]. (**d**) *BSet* (AI) [37].

Similarly, if this program is correctly executed in the testing phase as shown in Figure 2a, according to the *BSet* collection conditions, $BSet(R1)$ and $BSet(W2)$ are stored as *null* and $BSet(R3)$ and $BSet(W4)$ are stored as $W2$. This *BSet* is used to diagnose atomicity violations in comparison with the actual interleaving in the operation phase. If this program is incorrectly executed in the operation phase, as shown in Figure 2b, *BSet* is collected as shown in the atomicity violation of Figure 2d. The actual execution is shown in Figure 2b. AI repairs the atomicity violation using the following steps: First, when read access $R1$ of $thread1$ is issued, it is compared with the $BSet(R1)$ from the correct interleaving. We find that it is not diagnosed as an error, because the $BSet(R1)$ from the correct interleaving and the $BSet(R1)$ from the actual interleaving are equal to *null*. Next, when the read access $R3$ of $thread2$ is issued, it is compared with the $BSet(R3)$ from the correct interleaving. Since

the $BSet(R3)$ from the correct interleaving and the $BSet(R3)$ from the actual interleaving are different from *null* and $R1$, it is diagnosed as an error. To treat the diagnosed error, AI stalls the execution read access $R3$ of *thread*2 via `sleep()` and lets the program continue with *thread*1 and issue write access $W2$. Since the execution sequence is reordered to the correct sequence, the atomicity violation is repaired.

The on-the-fly repairing of atomicity violations is essential in two aspects, time overhead and coverage. These approaches have the advantage of low overhead because atomicity violations are diagnosed based on the correct interleaving collected in the test phase and treated with stalling. The low overhead of these approaches shows their applicability to real-time systems. The coverage is different because the methods of collecting the correct interleaving of the two approaches are different. J. Yu et al. [36]'s method collects only one correct interleaving and AI [37] collects as many correct interleaving types as possible. This difference allows AI to diagnose the correct interleaving so that the accuracy of the diagnosis is higher than that of J. Yu et al.'s method [36].

3.2. Avionics

There are reports and research studies of concurrency errors occurred in airborne software. According to the report in the DoD JSSSEH [23], a race condition, which is one of the concurrency errors, occurred in the first shuttle flight and the 44th flight of NASA's Advanced Fighter Technology Integration (AFTI) F16. All airborne software must comply with ARINC 653, the standard for airborne software. To prevent airborne software from crashing, ARINC 653 introduces health management systems (HMSs) for IMA (integrated modular avionics). The job of HMSs is to detect and repair faults in software. Regardless of this preventive structure, there are reports and research studies documenting concurrency errors in airborne software. However, it is not surprising, because all multi-threaded programs are inherent with concurrency errors. There are research studies focusing on repairing atomicity violations raised in ARINC 653-based airborne software [6,16]. Ha et al. [16] and Tchamgoue et al. [6] experimentally proved that the atomicity-violation issue exists in ARINC 653-based software. The approach these works used to treat the faults are similar. They store the shared variable access information of each thread in a data structure called `access history` [5]. The atomicity violation is diagnosed by examining the `access history` to check if the shared variables of each thread can be parallelly executed and whether they are protected by a lock [4,28]. To treat atomicity violations, Ha et al.'s method [16] inserts a lock around a shared variable that is not protected by a lock. On the other hand, Tchamgoue et al.'s method [6] delays the thread initiating an access to the shared variable.

The `access history` used to check the parallelism is a set of data structures generally called `label`. The `label` stores parallelism and order relationship information for each access. The `label` can be expressed as `[`α`, `β`]`. α and β are any integers defined by the programmer to denote the start and the end of a thread, respectively. α must always be less than β ($\alpha < \beta$). Here, we give an example to better understand how labeling works to identify concurrent access to shared variables. Let us assume that there are two threads, T_i and T_j, branching from the main thread. The `label` of the main thread at the program startup time is `[1, 100]`. When the program creates two threads on the main thread, the `label` is split in half. Then, the `label` of *thread*1 is set to `[1, 50]` and the `label` of *thread*2 is set to `[51, 100]`. Next, we compare $[\alpha_i, \beta_i]$ of T_i and $[\alpha_j, \beta_j]$ of T_j to check if the two regions collide. The following rules identify the concurrent relationship between threads:

- The two threads T_i and T_j are in a concurrent relationship if the regions are $\alpha_i < \beta_j$ and $\alpha_j > \beta_i$.
- Otherwise, T_i and T_j are in a sequential relationship.

The atomicity violation detection protocol guarantees to find at least one error if there are concurrency violations in a code. Every access event `[label, Locks]` pair is logged in the history and categorizes the access event. There are four types of access events, Read, Write, Critical Section (CS)-Read and CS-Write. Based on the diagnosis, we take actions

based on a policy. When the read operation is detected, it checks for the write and CS-write operations in the history and whether they collide. We search for the Read, CS-read and CS-write operations to identify the atomicity violations of write operations.

The labeling scheme needs to log every access event in the history. The time and space overheads of existing atomicity violation diagnosing schemes depend on the maximum parallelism T and the time and space complexity are $O(T)$. Recent real-world programs are known to have millions of threads running at the same time [39–41] and navigation software of an aircraft has about a billion lines of code [42]. An autonomous repair-based HMS cannot use $O(T)$ software because it is inefficient and unreliable for airborne software.

4. RAV: Repairing Atomicity Violations

This section describes the details of repairing AV (RAV; Repairing Atomicity Violation), an on-the-fly repairing tool for atomicity violations in ARINC 653-based airborne software. First, we describe the procedure of the repairing operation of RAV in ARINC 653. Then, we describe the improved version of the AI (anticipate invariant) algorithm, which is called embedded AI (eAI). It is designed to be applicable to embedded systems. Then, we describe how RAV fits into the ARINC 653 health monitor system to repair atomicity violations in ARINC 653 application.

4.1. Procedure

The repairing AV (RAV) is an on-the-fly tool to repair atomicity violations in ARINC 653-based software. The architecture of RAV is illustrated in Figure 3. RAV is comprised of a diagnosis engine (AV-DE) and a treatment engine (AV-TE). The AV-DE watches the shared variables of each thread for diagnosis. The AV-TE stalls instructions that violate the atomicity. RAV has three execution paths, described below.

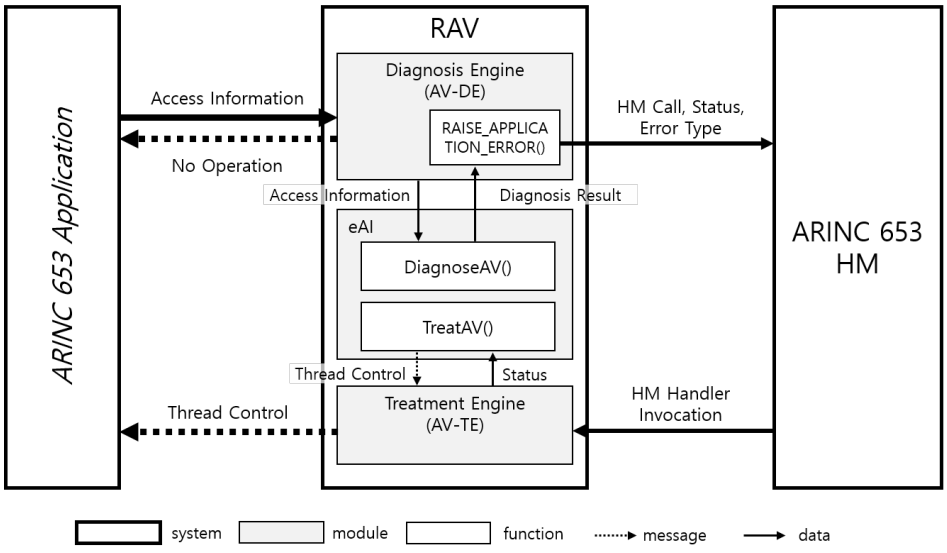

Figure 3. The architecture of RAV.

1. Halt: ARINC 653 Application → AV-DE → HM of ARINC 653 → AV-TE → Thread control;
2. Resume: ARINC 653 Application → AV-DE → HM of ARINC 653 → AV-TE → Thread control;
3. No Operation.

RAV halts a program in the halt execution path when an ARINC 653 program is diagnosed with an error. RAV executes halted thread again in the resume path if RAV determines that the thread does not violate atomicity. The last execution path of RAV, i.e., no operation, continues to diagnose the program and does not affect the execution of an ARINC 653 program.

RAV was designed based on AI [37] of the general-purpose platform. It was performed in the test phase to create correct interleaving. In the test phase, RAV executed the program ten times to acquire interleaving information. From the data, it generated an interleaving set. Since the interleaving set may have contained incorrect interleaving information, we manually inspected the correctness of the set to finalize the interleaving set. In the operation phase, RAV treats atomicity violations raised during execution. While testing embedded software that performs repetitive tasks, AI collects incorrect interleaving information, such as tracking local variables. We also observed a performance issue in performing the treatment in the operation phase caused by a `sleep()` time. Figure 4 shows the repairing time overhead of each treatment method. As shown in the figure, the greater the sleep time, the more the total repairing time increases in proportion.

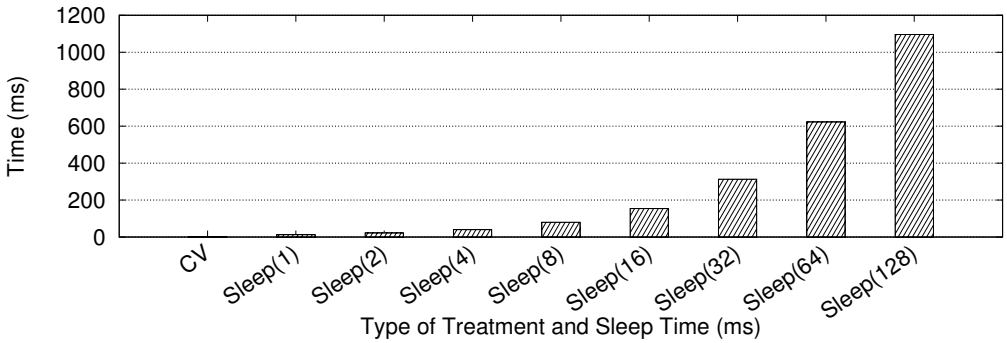

Figure 4. The repairing time of synthetic programs with 1000 iterations.

4.2. Embedded AI (eAI)

RAV modifies the algorithm of AI to collect correct interleaving in the test phase and to treat atomicity violations in the operation phase by reflecting the problem as described above. We named eAI (embedded AI) the modified version of AI. When a loop is executed in embedded software, eAI collects the static instruction S_x that satisfies the following conditions for the current dynamic instruction D_x:

- A variable in the data area of low memory is allocated;
- The same memory address is assessed;
- The instruction comes from another thread;
- The instruction immediately before D_x is assessed.

The training algorithm of AI keeps track of all variables and then identifies variables that have accessed the same memory address in different threads as shared variables. Local variables declared in a loop can be repeatedly allocated and deallocated. Then, the variable may be allocated in the same address as the previous allocation. Then, AI faces the problem of identifying the local variables as shared variables. The training algorithm of eAI does not track all variables but only the variables allocated to the data area of memory. The overall correct interleaving collection process is as follows: (1) Instrument `TraceMemoryAccesses()` in the target program. (2) Execute the target program to trace information of shared variables. (3) Once the target program's execution is completed, the *BSet* is executed to create a program with trace information. (4) Obtain *BSet*, which is a correct interleaving set. This process is the same as AI, but the algorithm for instrumenting the memory tracking function into the target program was modified into Algorithm 1.

Algorithm 1: Training of eAI.

```
1 Function InstrumentLoadOrStore(Instruction *I):
2     IRBuilder<> IRB(I)
3     MDNode *Node = I->getMetadata("eAIMemoryAccessID")
4     Value *ID = Node->getOperand(0)
5     ConstantInt *CI = dyn_cast<ConstantInt>(ID)
6     Value *Addr = IsWrite ? cast<StoreInst>(I)->getPointerOperand() :
        cast<LoadInst>(I)->getPointerOperand()
7     if const GlobalValue* G = dyn_cast<GlobalVAlue>(Addr) then
8         IRB.CreateCall2(TraceAccesses, CI, IRB.CreatePointerCast(Addr,
            IRB.getInt8PtrTy()))
```

When atomicity violations are diagnosed, the `treatAV()` of AI stalls the thread using `sleep()`. Since stalling requires a fixed delay time, there is an issue of time overhead in the treatment process. If the overhead is high, the program may not meet the real-time performance requirements; thus, we should appropriately manage stalling time. As shown in Figure 4, the overhead of stalling in repair time increases rapidly according to the size of the delay time set for treatment. On the contrary, when we use the condition variable, the time spent repairing can be shorter than that when using the stalling. This paper used the condition variable, so that we could apply it in a real-time system. The stalling, which uses `sleep()`, can be easily applied because it only needs to delay the execution of the thread in which an atomicity violation has occurred. However, to use the condition variable, we must know the current repairing status of the program. Therefore, `treatAV() of eAI` uses a state machine to determine the current repairing status. The finite-state machine is illustrated in Figure 5. The state when the program is executed is *Normal*. When an atomicity violation is diagnosed during execution and `treatAV()` executes `wait()`, a transition is made from the "Normal" state to the "Wait" state. Since the diagnosis of atomic violation after switching to the *Wait* state occurs because the repairing has not been completed yet, `treatAV()` of eAI is not performed. Upon diagnosing that there are no atomic violations (it has been repaired), `treatAV()` transitions from the *Wait* state to the *Normal* state, resuming the stopped thread via `signal()`. The treatment algorithm of eAI is described in Algorithm 2.

Algorithm 2: Algorithm of eAI treatment.

```
1 Function TreatAV(Repairing Status status):
2     if status == Normal then
3         Wait()
4         status = "Wait"
5     else if status == Wait then
6         Signal()
7         status = "Normal"
```

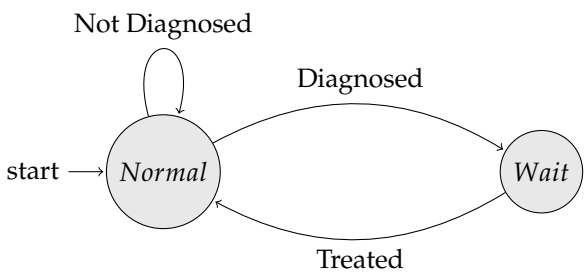

Figure 5. State diagram of eAI.

4.3. Application to ARINC 653

The operation of RAV between target software and the ARINC 653 HM is illustrated in Figure 3. The diagnosis engine of the RAV receives access information from the target application, diagnoses an atomicity violation and reports the result to the ARINC 653 HM. The treatment engine is invoked by the ARINC 653 HM to control a thread of the target application. Algorithm 3 represents a procedure that describes the process depicted in Figure 3. The diagnosis engine (AV-DE) is instrumented before and after the shared variables of ARINC 653 application. The instruction and memory address required for diagnosis are passed to `DiagnoseAV()` of eAI and the result is returned to the ARINC 653 HM. If an error occurs, the AV-DE reports the error to the ARINC 653 Health monitor (ARINC 653 HM) through `RAISE_APPLICATION_ERROR()`. Then, the ARINC 653 HM executes the registered error handler, the treatment engine (AV-TE), and takes actions to treat atomicity violations. The AV-DE can take on different roles depending on where it is instrumented. In the case of AV-DE being instrumented before shared variable access, the thread waits immediately when an atomicity violation occurs. In the case of AV-DE being instrumented after shared variable access, as soon as it is diagnosed that there are no atomicity violations, *signal* by `treatAV()` is executed so that the waiting thread is normally executed. All the diagnostic results from the AV-DE are reported to the ARINC 653 HM and the ARINC 653 HM treats the violations.

Algorithm 3: Algorithm of eAI with ARINC 653 HM.

```
1 Function AV-DE(Dynamic Instruction ins, Memory Adress addr):
2     bool isAV = DiagnoseAV(ins, addr)
3     int error_type = "atomiciy_violations"
4     int rc = 0
5     if isAV == true then
6         RAISE_APPLICATION_ERROR(error_type, status, sizeof(int), rc)
7 Function AV-TE(Repairing Status status):
8     TreatAV(status)
```

5. Experiments

This section describes the test environment, synthetic programs used for the experiments and results. Since airplanes are safety critical systems composed of hardware and software components, the system undergoes strict software development and testing processes. Typically, airborne software follows the V&V model and it is developed in the order item, system and aircraft [43]. As each component is integrated with other components, it is tested for correctness as well as reliability. The proposed method can be used in item verification and can also be deployed in health management systems for the live diagnosis and treatment of faults. In this paper, we show the results of experiments performed on an ARINC 653 environment called SIMA. The synthetic program we used was implemented as ARINC 653 software. The purpose of the program is to verify functionality and measure the performance of RAV.

5.1. Test Environment

We implemented and experimented on a system with Intel Xeon E5-2650 2.3 GHz CPU with 64 GB memory running on Ubuntu 14.04 LTS-64 bit. This system was installed in real-time kernel version 4.14.139-rt66 and the ARINC 653 simulator SIMA (Simulated Integrated Modular Avionics) v1.3.1. We compiled the code using GCC 5.4.0 and instrumented the code using LLVM v6.0.0.

5.2. Synthetic Programs

There are four reasons for using a synthetic program. First, even though a program is tested with static and dynamic debugging tools to find and fix the bugs, there are latent

bugs which are not discovered. We developed ARINC 653-compliant synthetic software that runs in an ARINC 653 environment. This is common practice because the effect of these bugs is very difficult to reproduce. Second, as the code size increases, the probability of atomicity violations decreases. It is because there is a high probability that critical sections avoid exhibiting atomicity violations. Thus, to reproduce and accelerate the debugging process, we used a synthetic program that models the critical sections. Third, atomicity violation is prevalent both in airborne software and general-purpose software. We analyzed atomicity violation patterns in different software types. The final and most critical reason why we used a synthetic program is because airborne software codes are proprietary and are not open to the public.

To test the proposed RAV, we modeled five atomicity violation patterns prevalent in real-world software and developed five synthetic programs for the ARINC 653 simulator. These synthetic programs were implemented as ARINC 653-compliant software that runs in an ARINC 653 environment. In the ARINC 653 operating system, these synthetic programs were loaded and executed in one partition. We did not consider the case of multiple partitions because the scope of this work is to repair atomicity violations in an intra-partition. It has to be noted that 97% of concurrency errors in real-world software are observed while two threads are executing [3]. We designed a synthetic program that ran two threads to create atomicity violations. We only included access events on a single shared variable without nested parallelism nor locks.

The five types of atomicity violations are described in Table 2 using bug reports on MySQL [44], Mozilla [45], Apache [46] and Ardupilot [47]. In the pattern, "R" and "W" denote read and write events, respectively. "[" and "]" denote the beginning and the end of a region, respectively, that must be protected by a lock. We used "||" to express the concurrent execution of threads. In the code shown in Figure 6a, `thread1()` acquires `lock` and performs write and read operations, which is denoted as $T_i^{[W-R]}$; `thread2()` runs without any protection and can be denoted as T_j^W. Since the two threads can be concurrently executed, we use $T_i^{[W-R]} \parallel T_j^W$ to represent this behavior.

Table 2. Patterns of real-world applications.

Case	Pattern	Report of Real-World Applications			
		MySQL [44]	**Mozilla** [45]	**Apache** [46]	**Ardupilot** [47]
Case 1	$T_i^{[R-R]} \parallel T_j^W$	#644, #3596, #12228	#341323, #224911	N/A	N/A
Case 2	$T_i^{[W-W]} \parallel T_j^R$	#791, #12848, #19938	#52111, #73761, #62269	N/A	N/A
Case 3	$T_i^{[W-R]} \parallel T_j^W$	#128486	N/A	N/A	N/A
Case 4	$T_i^{[R-W]} \parallel T_j^{R-W}$	#56324, #59464	#342577, #270689	#48735, #21287	#7129
Case 5	$T_i^{[R-R-W]} \parallel T_j^{R-R-W}$	N/A	N/A	#225525	N/A

Figure 6 describes the source code and the result of the code that executes $T_i^{[W-R]} \parallel T_j^W$. The initial value for the shared variable is 0. `thread 1` updates the shared variable (`sv = 1`), then checks the shared variable (`sv = 0`) to update the local variable. `thread 2` updates the shared variable (`sv = 0`). The program should not execute the conditional statement (line 11) because line 8 (`sv = 1`) of `thread 1`. However, as can be seen from the result shown in Figure 6b, the conditional statement in line 11 is executed and the text `lv: 1` is printed. In this case, the atomicity is violated where `sv = 0` (line 19) of `thread 2` executes after `sv = 1` (line 8) of `thread 1`. The result shows that there were five atomicity violations during the life of this short example.

```c
#define ITERATION 10000
int sv = 0; //shared variable

pthread_mutex_t mlock;

void *thread1(void *v) {

    sv = 1; //write

    int lv = 0; //local variable
    if(sv == 0) //read
    {
        lv++;
        printf("lv: %d\n", lv); fflush(stdout);
    }
}

void *thread2(void *v) {
    sv = 0; //write
}

void repairing_server()
{
    for(int i = 0; i < ITERATION; i++)
    {
        pthread_t t1id, t2id;
        pthread_create(&t1id, NULL, thread1, NULL);
        pthread_create(&t2id, NULL, thread2, NULL);

        pthread_join(t1id, NULL);
        pthread_join(t2id, NULL);
    }
}
```

(**a**)

```
<10 - WR|W Synthetic Program      <
<---------------------------------<
< With connected ports: 0         <
< mos: 24921                      <
< pos: 24924                      <
< cmd: 0                          <
< err: 0                          <
< The process for repairing_serve <
< r is created: 0000000472800000  <
< The process for repairing_serve <
< r is started: 0000000472800000  <
< ctl_demo_init end point: 000000 <
< 0472800000                      <
< Error handler is created: 00000 <
< 00472800000                     <
< ctl_err_init end point: 0000000 <
< 472900000                       <
< start W-R|W synthetic program!! <
<                                 <
< lv : 1                          <
<  lv : 1                         <
<  lv : 1                         <
<  lv : 1                         <
<  lv : 1                         <
<                                 <
< value of shared variable: 0     <
< memory usage: 20.404000         <
< execution time: 2729.641000     <
<                                 <
```

(**b**)

Figure 6. An example of execution order of $T_i^{[W-R]} \parallel T_j^W$ in the synthetic program. (**a**) Source code. (**b**) Result.

5.3. Functional Evaluation

This section evaluates the function of RAV when it is operating normally. The functions to be evaluated in RAV are instrumentation, collection of correct interleaving, diagnosis and treatment. Instrumentation examines the binary code to ensure that the algorithm's functions are properly inserted before and after the lines of the shared variable. RAV uses `_ai_trace_memory_access()` to trace memory access in the test phase. `_ai_pre_diagnosis_atomocity_violations()` and `_ai_post_diagnosis_atomocity_violations()` are instrumented for diagnosis and treatment in the operation phase. `_ai_pre_diagnosis_atomocity_violations()` is a function for `wait` when an error is reported after diagnosis and `_ai_post_ diagnosis_atomocity_violations()` is a function for `signal` when no errors are reported after diagnosis. Additionally, RAV instruments functions for initialization, destroy and experiment results. If these functions are not instrumented in the intended position of the code, the functions of RAV do not work properly. Figure 7 is the intermediate code showing the instrumented function of the code. As shown in the figure, the code was correctly instrumented before and after the shared variable `sv`.

Since correct interleaving is used for diagnosis in the operation phase, incorrect interleaving is not collected in the test phase. We examined the collected interleaving by performing training 100 times. Table 3 represents the training results. There were seven incorrect interleaving types for $T_1^{[R-R]} \parallel T_2^W$, none for $T_1^{[W-W]} \parallel T_2^R$, one for $T_1^{[W-R]} \parallel T_2^W$, fifteen for $T_1^{[R-W]} \parallel T_2^{R-W}$ and fourteen for $T_1^{[R-R-W]} \parallel T_2^{R-R-W}$. It was confirmed that the more complex the interleaving of the synthetic program was, such as $T_1^{[R-W]} \parallel T_2^{R-W}$ and $T_1^{[R-R-W]} \parallel T_2^{R-R-W}$, the more incorrect interleaving was collected. In the case of complex interleaving, incorrect interleaving occurred at about 15%, so correct interleaving could be obtained by performing sufficient training at least ten times. However, since the single execution of the training process may take a long time depending on the program's size, the

number of training was reduced with respect to the complexity of interleaving of shared variables.

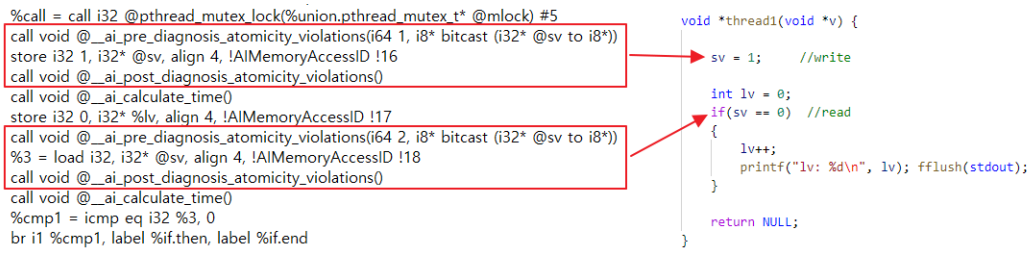

Figure 7. Result of instrumentation.

Table 3. Results of training.

Synthetic Program	Interleaving	Correctness	Number
$T_1^{[R-R]} \parallel T_2^W$	$T_1^R \rightarrow T_1^R \rightarrow T_2^W$	Yes	51
	$T_2^W \rightarrow T_1^R \rightarrow T_1^R$	Yes	42
	$T_1^R \rightarrow T_2^W \rightarrow T_1^R$	No	7
$T_1^{[W-W]} \parallel T_2^R$	$T_1^W \rightarrow T_1^W \rightarrow T_2^R$	Yes	48
	$T_2^R \rightarrow T_1^W \rightarrow T_1^W$	Yes	52
	$T_1^W \rightarrow T_2^R \rightarrow T_1^W$	No	0
$T_1^{[W-R]} \parallel T_2^W$	$T_1^W \rightarrow T_1^R \rightarrow T_2^W$	Yes	86
	$T_2^W \rightarrow T_1^W \rightarrow T_1^R$	Yes	13
	$T_1^W \rightarrow T_2^W \rightarrow T_1^R$	No	1
$T_1^{[R-W]} \parallel T_2^{R-W}$	$T_1^R \rightarrow T_1^W \rightarrow T_2^R \rightarrow T_2^W$	Yes	57
	$T_2^R \rightarrow T_2^W \rightarrow T_1^R \rightarrow T_1^W$	Yes	28
	$T_1^R \rightarrow T_2^R \rightarrow T_1^W \rightarrow T_2^W$	No	5
	$T_1^R \rightarrow T_2^R \rightarrow T_2^W \rightarrow T_1^W$	No	7
	$T_2^R \rightarrow T_1^R \rightarrow T_2^W \rightarrow T_1^W$	No	1
	$T_2^R \rightarrow T_1^R \rightarrow T_1^W \rightarrow T_2^W$	No	2
$T_1^{[R-R-W]} \parallel T_2^{R-R-W}$	$T_1^R \rightarrow T_1^R \rightarrow T_1^W \rightarrow T_2^R \rightarrow T_2^R \rightarrow T_2^W$	Yes	14
	$T_2^R \rightarrow T_2^R \rightarrow T_2^W \rightarrow T_1^R \rightarrow T_1^R \rightarrow T_1^W$	Yes	72
	$T_2^R \rightarrow T_1^R \rightarrow T_1^R \rightarrow T_2^R \rightarrow T_2^W \rightarrow T_1^W$	No	1
	$T_2^R \rightarrow T_1^R \rightarrow T_1^R \rightarrow T_1^W \rightarrow T_2^R \rightarrow T_2^W$	No	2
	$T_1^R \rightarrow T_2^R \rightarrow T_2^R \rightarrow T_1^R \rightarrow T_2^W \rightarrow T_1^W$	No	1
	$T_2^R \rightarrow T_2^R \rightarrow T_1^R \rightarrow T_1^R \rightarrow T_2^W \rightarrow T_1^W$	No	1
	$T_1^R \rightarrow T_1^R \rightarrow T_2^R \rightarrow T_2^R \rightarrow T_1^W \rightarrow T_1^W$	No	4
	$T_2^R \rightarrow T_2^R \rightarrow T_1^R \rightarrow T_1^R \rightarrow T_1^W \rightarrow T_2^W$	No	1
	$T_1^R \rightarrow T_2^R \rightarrow T_2^R \rightarrow T_2^W \rightarrow T_1^R \rightarrow T_1^W$	No	2
	$T_1^R \rightarrow T_1^R \rightarrow T_2^R \rightarrow T_2^R \rightarrow T_1^W \rightarrow T_2^W$	No	1
	$T_1^R \rightarrow T_1^R \rightarrow T_2^R \rightarrow T_1^W \rightarrow T_2^R \rightarrow T_2^W$	No	1

The state machine used for diagnosis and the condition variable used for treatment must work correctly for accurate repair. We performed a functional evaluation of diagnosis

and treatment and the results are shown in Figure 8, which is a part of the log of the repairing process of $T_1^W \parallel T_2^{[W-R]}$. `thread 1` on the left and `thread 2` on the right are executed in order. `thread 1` repeatedly executes the write operation. `thread 2` executes the read and write operations atomically. After the read operation in `thread 2` (id: 6448), the write operation is not executed and the write operation of `thread 1` (id: 6449) tries to run. At this point, the diagnosis engine reports an atomicity violation. Moreover, before accessing write, the treatment engine executes `wait` to stop `thread 1`. Then, the write operation in `thread 2` (id: 6448) executes normally and the treatment engine executes `signal` and resumes the halted `thread 1` to repair atomicity violations. Post-repair runs appeared to execute with normal interleaving.

Figure 8. Log of repairing.

We created a synthetic program with a loop which repeated for a hundred times. Table 4 shows the result of running RAV on the created program. The first column shows the type of program and the second column shows the interleaving of threads observed during the execution of the program. We analyzed the interleaving types and identified the ones with atomicity violations; we here show them in the third column. The diagnosis column shows the count of different interleaving observed in the program. In the diagnosis phase, every interleaving is inspected whether it is correct interleaving or interleaving with atomicity violation. Since a tread execution is nondeterministic, the number shown in the table is a sample observation. Once an interleaving is diagnosed as a violation of atomicity, the interleaving is treated. The fifth column shows the number of treated interleaving types. Since there cannot be a false negative or false positive in treatment, the accuracy of the treatment is 100%.

Table 4. Result of functional evaluation of RAV.

Synthetic Program	Interleaving	Atomicity Violation	Diagnosis	Treatment	Accuracy
$T_i^{[R-R]} \parallel T_j^W$	$T_1^R \to T_2^W \to T_1^R$	Yes	27	27	100%
	$T_1^R \to T_1^R \to T_2^W$	No	24	-	
	$T_2^W \to T_1^R \to T_1^R$	No	49	-	
$T_i^{[W-W]} \parallel T_j^R$	$T_1^W \to T_2^R \to T_1^W$	Yes	24	24	100%
	$T_1^W \to T_1^W \to T_2^R$	No	24	-	
	$T_2^R \to T_1^W \to T_1^W$	No	52	-	
$T_i^{[W-R]} \parallel T_j^W$	$T_1^W \to T_2^W \to T_1^W$	Yes	29	29	100%
	$T_1^W \to T_1^R \to T_2^R$	No	34	-	
	$T_2^R \to T_1^W \to T_1^R$	No	37	-	
$T_i^{[R-W]} \parallel T_j^{R-W}$	$T_1^R \to T_2^R \to T_1^W \to T_2^W$	Yes	28	28	100%
	$T_2^R \to T_1^R \to T_2^W \to T_1^W$	Yes	39	39	
	$T_1^R \to T_1^W \to T_2^R \to T_2^W$	No	27	-	
	$T_2^R \to T_2^W \to T_1^R \to T_1^W$	No	6	-	
$T_i^{[R-R-W]} \parallel T_j^{R-R-W}$	$T_1^R \to T_2^R \to T_1^R \to T_1^W \to T_2^R \to T_2^W$	Yes	18	18	100%
	$T_2^R \to T_1^R \to T_2^R \to T_2^W \to T_1^R \to T_1^W$	Yes	50	50	
	$T_1^R \to T_1^R \to T_1^W \to T_2^R \to T_2^R \to T_2^W$	No	24	-	
	$T_2^R \to T_2^R \to T_2^W \to T_1^R \to T_1^R \to T_1^W$	No	8	-	

5.4. Performance Evaluation

This section evaluates the performance of RAV. We measured the time and space overheads with native and Tchamgoue et al.'s [17] algorithm to verify the performance of RAV. Time overhead measured program execution time, diagnosis time and treatment time. Space overhead measured total memory usage used during program execution. We measured the overhead in the program execution time by increasing the number of repetitions of the synthetic program from 100,000 to 1,280,000 while increasing the repetition by multiples of two.

Figure 9 shows the time overhead of five synthetic programs. The total program execution time increased approximately twice in proportion to the number of program iterations. The total execution time of each approach was also similar. The time overhead of Tchamgoue et al.'s [17] algorithm took an average of 1.19×, a minimum of 1.06× and a maximum of 1.29×, compared to the native algorithm. The time overhead of RAV took an average of 1.13×, a minimum of 1.06× and a maximum of 1.20×, compared to the native algorithm. RAV showed slightly lower time overhead than Tchamgoue et al.'s [17] algorithm, but the difference was very small. The cause of the time overhead is rooted in the time spent on diagnosis and treatment in the repairing algorithm. When the number of branches in the program is small, Tchamgoue et al.'s [17] algorithm does not require a significant overhead because the time spent comparing access history is small. However, if the number of threads is large, the overhead increases considerably. Since RAV only checks the collected correct interleaving, it can be diagnosed with lower overhead than Tchamgoue et al.'s [17] algorithm. The experimental results show that it is repaired with more overhead than the native program, but less than Tchamgoue et al.'s [17] algorithm. As a result, the difference in the time overheads of the two approaches was a minimum of 0% and both techniques are adequate for a real-time system if we can tolerate a time overhead of about 5–30%.

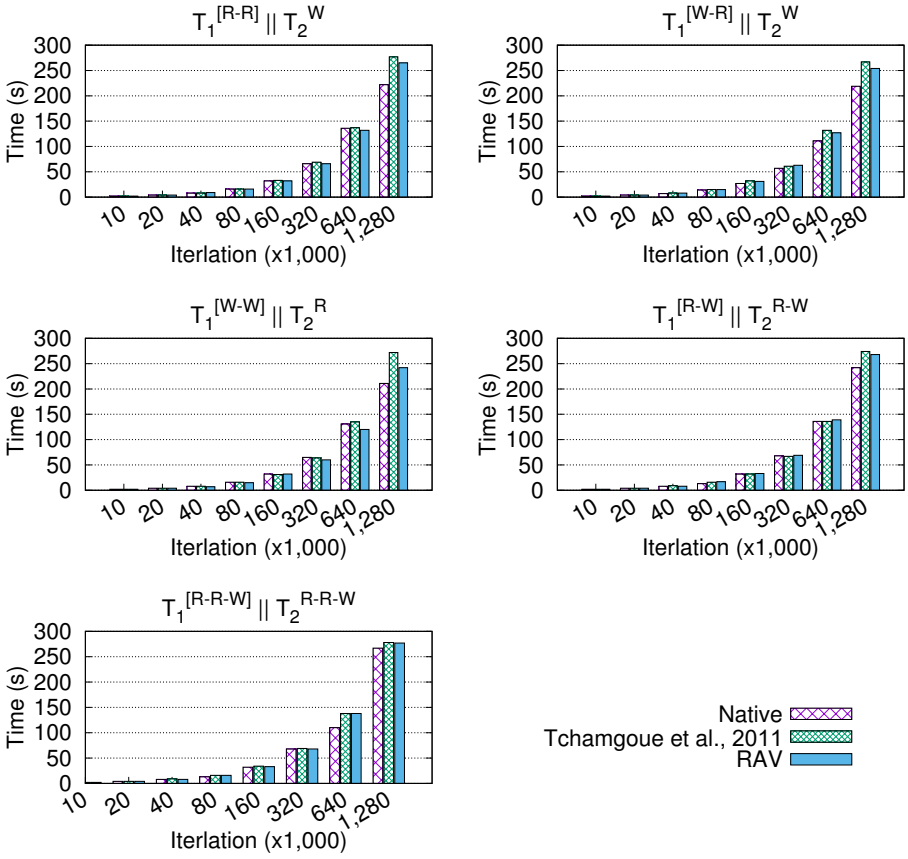

Figure 9. Execution time of synthetic programs [16].

Figure 10 shows the result of the space overhead of the five synthetic programs. In all synthetic programs, the native algorithm showed the same space overhead of 20 KB, regardless of iterations. Tchamgoue et al.'s [17] algorithm showed an exponential increase in space usage. From 10,000 to 80,000 times, the overhead was lower than that of RAV, but, after 80,000, the overhead was higher than that of RAV. RAV showed the same overhead of 219 KB from program start to end. Tchamgoue et al.'s [17] algorithm creates the access history every time a thread occurs and uses a lot of memory because it needs to update the access history every time the shared memory is accessed. RAV shows the same memory usage from beginning to end because correct interleaving is inserted into the memory at the same time as program execution starts. As a result, Tchamgoue et al.'s [17] algorithm has a higher memory usage, so, when it is applied in airborne software, it is necessary to secure the corresponding memory usage. Since RAV has the same or lower overhead, it can be applied to airborne software.

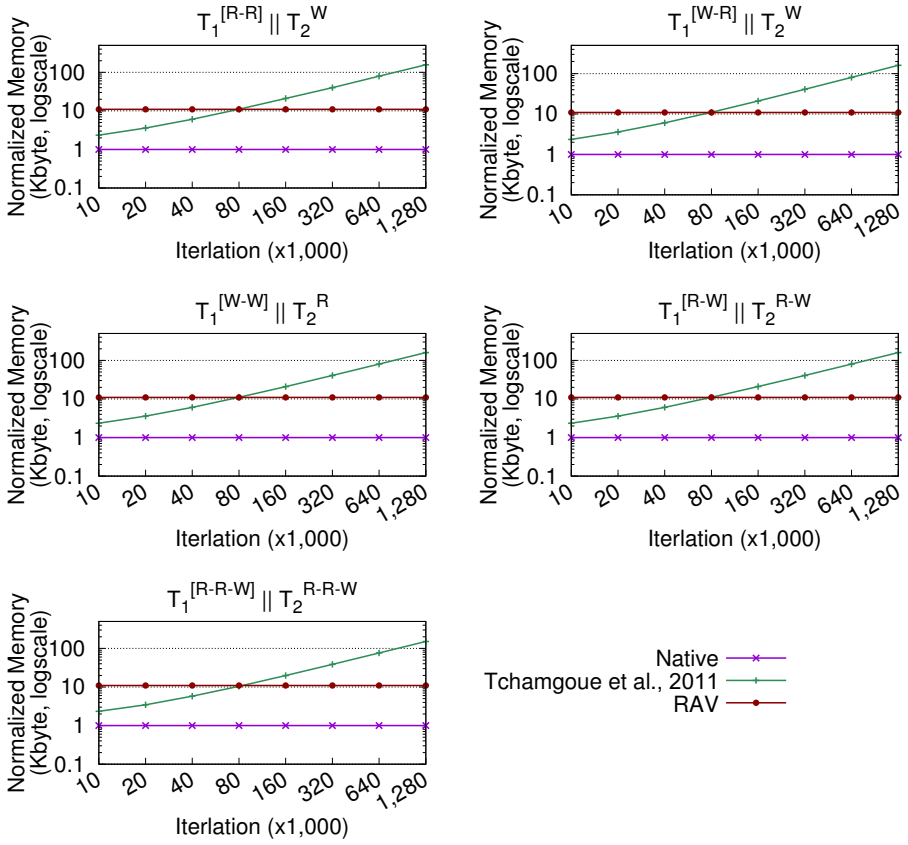

Figure 10. Memory usage of synthetic programs [16].

Figure 11 shows the repairing time spent while executing a synthetic program with 1,280,000 iterations. The repairing time of Tchamgoue et al.'s [17] algorithm was 3.1 s on average, a maximum of 4 s for $T_1^{[W-R]} \parallel T_2^W$ and a minimum 2 of seconds for $T_1^{[W-W]} \parallel T_2^R$. The repairing time of RAV was, on average, 1.8 s, a maximum of 2.0 for $T_1^{[W-R]} \parallel T_2^W$ and a minimum of 1.6 for $T_1^{[R-R]} \parallel T_2^W$. The repairing time of Tchamgoue et al.'s [17] algorithm showed an average of 1.7×, a maximum of 1.99× and a minimum of 1.4× longer than RAV. Tchamgoue et al.'s [17] program showed that the diagnosis time was lower than that of RAV, but the total repairing time was higher than that of RAV because of the high treatment time. The cause of the overhead is that, after diagnosing an atomic violation by examining the locking discipline, it waits until the execution of the lock is finished before performing treatment. Unnecessary treats can also cause overhead to increase due to many false positives. RAV has a higher diagnostic time than Tchamgoue et al.'s [17] algorithm but has lower overhead because it resumes the stopped thread as soon as the variable executes

correctly. As a result, the proportion of treatment time in the total repairing time shows an average of 0.13%, a maximum of 0.18% for $T_1^{[R-R]} \parallel T_2^W$ and a minimum of 0.05% for $T_1^{[W-R]} \parallel T_2^W$.

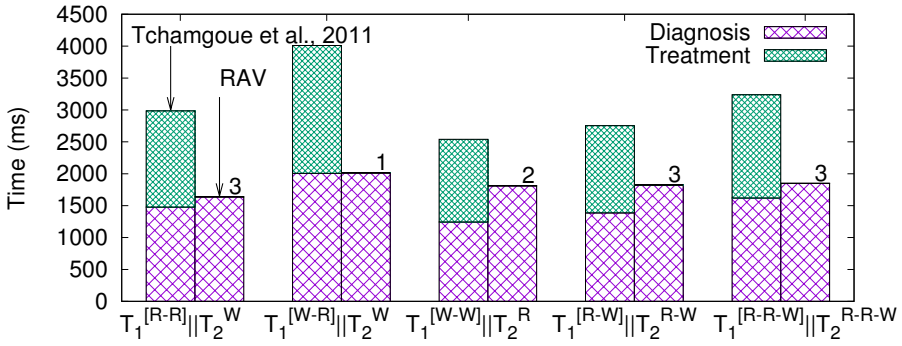

Figure 11. Repairing time of synthetic programs with 1,280,000 iterations (note that the treatment time for RAV is described above the bar) [16].

Summarizing the results, RAV collected incorrect interleaving with a low probability when training a program in which the interleaving of shared variables was complex. Therefore, it is necessary to perform sufficient training and check the correctness of the collected interleaving. RAV repaired all five types of atomicity violations in real-world software. It showed an average of 1.13× time overhead and a constant space overhead of 219 KB. Most of the repair time was spent on diagnosis and the time required for action was very low. Therefore, RAV can be applied to airborne software to repair atomicity violations with low overhead. Succeeding researchers can focus on automatic verification or collection methods to increase the accuracy of correct interleaving collection in training. In addition, we plan to improve the algorithm to be repaired in RAV by analyzing multi-variable atomicity violations that manifest in real-world software.

6. Discussion

In Section 5, we show that the proposed tool treated the atomicity violations in the synthetic program with low overhead. The synthetic program was developed in an ARINC 653 environment called Simulated Integrated Modular Avionics (SIMA), which is common practice in the industry. We used synthetic programs because airborne software is proprietary and the industry does not disclose them for security reasons. Synthetic or not, airborne software is vulnerable to concurrency errors because the ARINC 653 operating system supports Pthread library. Moreover, it is impossible to guarantee that the treatment of atomicity violations is correct. Additionally, it is difficult to guarantee that a piece of code does not have side effects on other software components unless it is tested in a control environment.

UAVs and drones are attracting many researchers in the field, because, unlike other airborne software, many of their source codes are available as opensource projects. These software types must be inspected for any latent errors because these projects do not always strictly follow the standards.

In RAV, we manually inspected the accuracy of the interleaving set acquired in the test phase. Since it is manually inspected, this might increase the number of false positive and false negative cases when the code is complex or when the developer makes mistakes. For future work, we need to develop an automated tool that guarantees the integrity of the correct interleaving set.

7. Conclusions

It is essential for the aircraft health management system in ARINC 653-based concurrent programs to repair atomicity violations on the fly to ensure the program's normal

execution. In previous works, atomicity violations have been diagnosed by comparing the `access history` generated and maintained whenever an access operation is executed with the access information during execution. There is time and space overhead for repairing whenever an access operation executes. This paper proposes RAV (Repairing Atomicity Violation). It diagnoses atomicity violations by using correct interleaving collected in the test phase.

RAV showed a time overhead of 1.6× 1.20× regardless of the number of accesses to shared variables. It also showed the space overhead of 219 KB. The aircraft is a hard real-time system, so the program must have a constant time and space overhead during operation. If the design considers the time and space overhead required for repairs, RAV is well suited for use in airborne software.

Author Contributions: Conceptualization, T.-h.K. and Y.-K.J.; methodology, E.-t.C.; software, E.-t.C. and T.-h.K.; validation, E.-t.C., and M.H.; investigation, E.-t.C. and T.-h.K.; resources, E.-t.C.; writing—original draft preparation, E.-t.C. and T.-h.K.; writing—review and editing, S.L.; visualization, E.-t.C.; supervision, Y.-K.J. and S.L.; project administration, Y.-K.J.; funding acquisition, Y.-K.J. and S.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research study was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF2018R1D1A3B07041838) and was supported by the Technology Innovation Program (or Industrial Strategic Technology Development Program, 20005378, Open Avionics System Architecture and Software Development for Small to Medium Aircraft Class) funded by the Ministry of Trade, Industry and Energy (MOTIE, Korea). It was also supported by the Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (21ZS1300; Research on High Performance Computing Technology to overcome limitations of AI processing).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Firesmith, D.G.; Capell, P.; Falkenthal, D.; Hammons, C.B.; Latimer IV, D.T.; Merendino, T. *The Method Framework for Engineering System Architectures*; CRC Press: Boca Raton, FL, USA, 2008.
2. Netzer, R.H.; Miller, B.P. What are race conditions? Some issues and formalizations. *ACM Lett. Program. Lang. Syst.* **1992**, *1*, 74–88. [CrossRef]
3. Lu, S.; Park, S.; Seo, E.; Zhou, Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGOPS Oper. Syst. Rev.* **2008**, *42*, 329–339. [CrossRef]
4. Dinning, A.; Schonberg, E. Detecting access anomalies in programs with critical sections. In Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, USA, 20–21 May 1991; pp. 85–96.
5. Jun, Y.K.; Koh, K. On-the-fly detection of access anomalies in nested parallel loops. *ACM SIGPLAN Not.* **1993**, *28*, 107–117. [CrossRef]
6. Ha, O.K.; Kuh, I.B.; Tchamgoue, G.M.; Jun, Y.K. On-the-Fly Detection of Data Races in OpenMP Programs. In Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, Minneapolis, MN, USA, 16 July 2012; pp. 1–10. [CrossRef]
7. Ratanaworabhan, P.; Burtscher, M.; Kirovski, D.; Zorn, B.; Nagpal, R.; Pattabiraman, K. Detecting and Tolerating Asymmetric Races. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, 15–19 February 2009; pp. 173–184. [CrossRef]
8. Lucia, B.; Ceze, L. Cooperative Empirical Failure Avoidance for Multithreaded Programs. *SIGPLAN Not.* **2013**, *48*, 39–50. [CrossRef]
9. Mahadevan, N.; Dubey, A.; Karsai, G. Application of Software Health Management Techniques. In Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, New York, NY, USA, 23–24 May 2011; pp. 1–10. [CrossRef]
10. Srivastava, A.N.; Schumann, J. The Case for Software Health Management. In Proceedings of the 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, Palo Alto, CA, USA, 2–4 August 2011; pp. 3–9. [CrossRef]

11. Goldberg, A.; Horvath, G. Software Fault Protection with ARINC 653. In Proceedings of the 2007 IEEE Aerospace Conference, Big Sky, Montana, 3–10 March 2007; pp. 1–11. [CrossRef]
12. Ofsthun, S. Integrated vehicle health management for aerospace platforms. *IEEE Instrum. Meas. Mag.* **2002**, *5*, 21–24. [CrossRef]
13. Spitzer, C.; Ferrell, U.; Ferrell, T. *Digital Avionics Handbook*; CRC Press: Boca Raton, FL, USA, 2014.
14. Committee, A.E.E. *Avionics Application Software Standard Interface: ARINC Specification 653 Part 1 (Supplement 4—Required Services)*; Aeronautical Radio: Annapolis, MD, USA, 2015.
15. Pullum, L.L. *Software Fault Tolerance Techniques and Implementation*; Artech House, Inc.: Norwood, MA, USA, 2001.
16. Ha, O.; Tchamgoue, G.M.; Suh, J.; Jun, Y. On-the-fly healing of race conditions in ARINC-653 flight software. In Proceedings of the 29th Digital Avionics Systems Conference, Salt Lake City, UT, USA, 3–7 October 2010; pp. 5.A.6-1–5.A.6-11. [CrossRef]
17. Tchamgoue, G.M.; Ha, O.K.; Kim, K.H.; Jun, Y.K. A framework for on-the-fly race healing in ARINC-653 applications. *Int. J. Hybrid Inf. Technol.* **2011**, *4*, 1–12.
18. New Software Glitch Found in Boeing's Troubled 737 Max Jet. Available online: https://apnews.com/article/f192296ce28843c3aa5b7cd599e0a69f. (accessed on 26 November 2020).
19. RTCA, I. *Software Considerations in Airborne Systems and Equipment Certification*; RTCA, Inc.: Washington, DC, USA, 2011.
20. Wilfredo, T.P. *Software Fault Tolerance: A Tutorial*; NASA: Washington, DC, USA, 2000.
21. The Open Group. *FACE (Future Airborne Capability Environment) Technical Standard, Edition 3.0*; The Open Group: San Francisco, CA, USA, 2017.
22. Wang, J.; Dou, W.; Gao, Y.; Gao, C.; Qin, F.; Yin, K.; Wei, J. A Comprehensive Study on Real World Concurrency Bugs in Node.Js. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Urbana, IL, USA, 30 October–3 November 2017; pp. 520–531.
23. Department of Defense. *Joint Software Systems Safety Engineering Handbook*; Department of Defense: Defense, VI, USA, 2010; pp. E-15–E-18.
24. Apache Project Statistics. Available online: https://projects.apache.org/statistics.html (accessed on 9 April 2020).
25. Giebas, D.; Wojszczyk, R. Detection of Concurrency Errors in Multithreaded Applications Based on Static Source Code Analysis. *IEEE Access* **2021**, *9*, 61298–61323. [CrossRef]
26. Singh, A.; Pai, R.; D'Souza, D.; D'Souza, M. Static Analysis for Detecting High-Level Races in RTOS Kernels. In *Formal Methods—The Next 30 Years*; ter Beek, M.H., McIver, A., Oliveira, J.N., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 337–353.
27. Giebas, D.; Wojszczyk, R. Atomicity Violation in Multithreaded Applications and Its Detection in Static Code Analysis Process. *Appl. Sci.* **2020**, *10*, 8005. [CrossRef]
28. Savage, S.; Burrows, M.; Nelson, G.; Sobalvarro, P.; Anderson, T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* **1997**, *15*, 391–411. [CrossRef]
29. Eslamimehr, M.; Lesani, M.; Edwards, G. Efficient detection and validation of atomicity violations in concurrent programs. *J. Syst. Softw.* **2018**, *137*, 618–635. [CrossRef]
30. Ma, X.; Wu, S.; Pobee, E.; Mei, X.; Zhang, H.; Jiang, B.; Chan, W.K. RegionTrack: A Trace-Based Sound and Complete Checker to Debug Transactional Atomicity Violations and Non-Serializable Traces. *ACM Trans. Softw. Eng. Methodol.* **2021**, *30*, 1–49. [CrossRef]
31. Cheptsov, V.; Khoroshilov, A. Dynamic Analysis of ARINC 653 RTOS with LLVM. In Proceedings of the 2018 Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russia, 22–23 November 2018; pp. 9–15. [CrossRef]
32. Domingues, R.P.; De Melo Bezerra, J.; Hirata, C.M. Design recommendations to mitigate memory and cache non-determinisms in multi-core based IMA platform of airborne systems. In Proceedings of the 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC), Prague, Czech Republic, 13–17 September 2015; pp. 7A1-1–7A1-9. [CrossRef]
33. Kim, H.J.; Ha, O.K.; Jun, Y.K.; Park, H.D. Message Races in Data Distribution Service Programs. In Proceedings of the 2015 8th International Conference on Database Theory and Application (DTA), Jeju Island, Korea, 25–28 November 2015; pp. 33–36. [CrossRef]
34. Singh, A.; D'Souza, M.; Ebrahim, A. Formal Verification of Datarace in Safety Critical ARINC653 compliant RTOS. In Proceedings of the 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Bangalore, India, 19–22 September 2018; pp. 1273–1279. [CrossRef]
35. Singh, A.; D'Souza, M.; Ebrahim, A. Conformance Testing of ARINC 653 Compliance for a Safety Critical RTOS Using UPPAAL Model Checker. In Proceedings of the 36th Annual ACM Symposium on Applied Computing, New York, NY, USA, 22–26 March 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 1807–1814.
36. Yu, J.; Narayanasamy, S. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. *SIGARCH Comput. Archit. News* **2009**, *37*, 325–336. [CrossRef]
37. Zhang, M.; Wu, Y.; Lu, S.; Qi, S.; Ren, J.; Zheng, W. A Lightweight System for Detecting and Tolerating Concurrency Bugs. *IEEE Trans. Softw. Eng.* **2016**, *42*, 899–917. [CrossRef]
38. Krena, B.; Letko, Z.; Tzoref, R.; Ur, S.; Vojnar, T. Healing Data Races On-the-Fly. In Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging, London, UK, 9 July 2007; pp. 54–64. [CrossRef]
39. Dang, H.V.; Snir, M.; Gropp, W. Towards Millions of Communicating Threads. In Proceedings of the 23rd European MPI Users' Group Meeting, Edinburgh, UK, 25–28 September 2016; pp. 1–14. [CrossRef]

40. Ha, O.K.; Jun, Y.K. An efficient algorithm for on-the-fly data race detection using an epoch-based technique. *Sci. Program.* **2015**, *2015*, 1–14. [CrossRef]
41. Sridharan, S.; Gupta, G.; Sohi, G.S. Adaptive, Efficient, Parallel Execution of Parallel Programs. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK, 9–11 June 2014; pp. 169–180. [CrossRef]
42. Ebert, C.; Jones, C. Embedded Software: Facts, Figures, and Future. *Computer* **2009**, *42*, 42–52. [CrossRef]
43. International, S. *Guidelines for Development of Civil Aircraft and Systems*; SAE International.: Warrendale, PA, USA, 2010.
44. MySQL Bugs. Available online: http://bugs.mysql.com/ (accessed on 1 November 2020).
45. Mozilla Bugs. Available online: https://bugzilla.mozilla.org (accessed on 1 November 2020).
46. Apache Bugs. Available online: https://httpd.apache.org/bug_report.html (accessed on 1 November 2020).
47. Ardupilot Bugs. Available online: https://github.com/ArduPilot/ardupilot/issues (accessed on 1 November 2020).