A Framework for On-the-fly Race Healing in ARINC-653 Applications

Guy Martin Tchamgoue, Ok-Kyoon Ha, Kyong-Hoon Kim and Yong-Kee Jun
Department of Informatics, Gyeongsang National University,
Jinju 660-701, Republic of Korea
guymt@ymail.com, {jassmin,khkim,jun}@gnu.ac.kr

Abstract

The ARINC-653 standard architecture for flight software specifies an application executive (APEX) which furnishes an application programming interface of fifty-one routines. APEX enables the development of portable applications, providing a strict time and space partitioning for their execution along with intra- and inter-partition communication facilities. This architecture also defines a hierarchical health management framework for error detection and recovery. However, in every partition, asynchronously concurrent processes or threads may include concurrency bugs such as unintended data races which are common and difficult to remove by testing. To reinforce the capability of the ARINC-653 health management system and to increase the reliability of flight software, this article describes the development and the configuration of an on-the-fly race healing framework into a simulated ARINC-653 platform which provides real ARINC-653 programming interface. The experimental results allow us to argue that our race healing framework is practical enough to be configured under the ARINC-653 partitions.

Keywords: *ARINC-653, health management, partition, data races, race healing.*

1. Introduction

In highly critical real-time applications such as flight software, reliability is a very sensitive notion. Unfortunately, flight software applications are becoming more and more complex. Thus, new techniques have to be designed for the verification and mostly for the fault containment and recovery in such applications. To handle this need, the ARINC-653 [1,2,3,4] standard architecture for flight software has been introduced. This architecture specifies an application executive (APEX) which provides an application programming interface of fifty-one routines. APEX enables the development of portable applications on an Integrated Modular Avionics platform, supporting temporal and spatial partitioning along with communication between applications in different partitions through well-defined ports. The architecture also defines a health management framework which can be used to provide a hierarchical framework for error detection and recovery at the process, partition, and module levels, supporting fault tolerance and then operational availability to be increased.

In a partition of ARINC-653 architecture, processes or threads are executed concurrently according to a preemptive, priority-based scheduler. Accesses to shared variables or resources are coordinated by semaphores and events, such as waiting on empty or full buffers. However, misusing these facilities leads to concurrency bugs such as data races [5,6] which are common and difficult to remove by testing. A data race is a pair of unsynchronized instructions in concurrent processes or threads that access a shared variable with at least one write access. Data races threaten the reliability of shared-memory programs seriously and latently, because the races result in unintended nondeterministic executions of the programs. Traditional cyclical debugging with breakpoints is often not effective in the presence of races since breakpoints can change the execution timing causing the erroneous behavior to disappear.

In a recent work [7,8], we presented an on-the-fly race healing framework to be incorporated into the ARINC-653 health management architecture for healing data races that appear unexpectedly during the run-time of flight software. This framework instruments and monitors the target program for race detection using the Dinning and Schonberg protocol [9]. This protocol determines the logical concurrency between the current access and the previous conflicting accesses, and then maintains an access history for each shared variable for subsequent race detection. For the concurrency information to be used in the protocol, the framework generates a variation of vector timestamps [10,11] for every process in the execution using the Nest Region labeling [12]. In this framework, the detection protocol signals to the health monitor (HM) using the corresponding APEX system call, if a data race is detected. The health monitor then responds by invoking the race healing process that is assigned the highest possible priority. This special process uses an APEX call to identify and then heals the faulty process and the occurrence of race condition as an application error, one of seven error types defined by ARINC-653.

This article presents the implementation of this race healing framework in a simulated ARINC-653 environment. We describe the requirements and the design of the framework, and use the Simulated Integrated Modular Avionics (SIMA) [13] environment to provide real ARINC-653 programming interface. In order to achieve the reliability-related goal of the framework, we present details of the analysis to argue that the race healing framework is practical enough to be configured under the ARINC-653 partitions.

In the rest of the article, Section 2 introduces the ARINC-653 standard and presents the architecture of our race healing framework. Section 3 presents our implementation of the framework in SIMA. Section 4 is about the data race healing guarantee. In Section 5, details are provided on the analysis and the configurability of the framework in SIMA. Prior related work is presented in Section 6. The article is concluded in Section 7.

2. Background

Since ARINC-653 was introduced, its fault containment and health management capabilities have been attractive to the programmers of flight software. This section illustrates ARINC-653 and its functionality on health management, and introduces our healing framework architecture for ARINC-653 flight software.

2.1. ARINC-653

The ARINC-653 Specification [1,2,3,4] has been developed as a standardized interface definition of real-time operating system to simplify the development of Integrated Modular Avionics. This standard specifies an Application Executive (APEX) which provides services comprised of a set of fifty-one routines to enable the development of portable applications on an IMA platform. The main objective of the APEX is to provide a strict and robust time and space partitioning environment allowing a processing unit known as module to host multiple applications independently in each partition. A module is managed by an operating system called Module Operating System.

The temporal partitioning is a strict time slicing which guarantees that only one application at a time is accessing the system resources including the processor according to a periodic scheduler. The spatial partitioning provides strict memory management by guaranteeing that a memory area allocated to a partition and its processes cannot be corrupted by another partition and its processes. Each partition is governed by a Partition Operating System. A partition consists of a set of concurrently executing processes, sharing access to the system resources with the help of a preemptive, priority-based scheduler.

One of the most important features of ARINC-653 is indisputably its health monitor (HM) which has the responsibility to detect and provide recovery mechanisms for both hardware and software failures at the process, partition, and module levels. The HM manages three recovery tables for a precise error handling. The System HM Table is used to localize the level of the error which can be one of module, partition or process. The Module HM table is consulted for errors in module level. This table defines the action to be taken to handle the fault and optionally may provide a module-level user-defined call-back function. The Partition HM table similarly is consulted when the error is at partition level. Each partition has its own error recovery table which optionally provides a partition-level user-defined call-back function. For errors at the process level, the HM will invoke a user-defined aperiodic error handler with the highest possible priority to handle the error. Consequently, the error handler should be efficient and execute as fast as possible not to monopolize the system.

2.2. Race healing framework

The ARINC-653 Standard clearly defines a health monitoring function with the objective of keeping the entire system safe by providing mechanisms to handle software and hardware failures and errors. Data races are timing-dependent bugs which are hard to detect and may lead a program to non-deterministic executions. This behavior can be fatal especially for hard real-time applications such as ARINC-653 applications.

The architecture of the race healing framework [7,8] in ARINC-653 is presented in Figure 1. The race detector monitors the instrumented program execution, and once a data race is detected, the ARINC-653 Health Monitor is notified (see the second arrow in Figure 1) by the race detector using the application executive call RAISE_APPLICATION_ERROR. The Health Monitor will then contact the partition operating system (POS) of the partition in which the data race occurred. As a response, the race healer will be invoked by the concerned POS (see the third arrow in Figure 1). The race healer accesses the racing code and tries to heal the data race (see the fourth arrow in Figure 1). If the healer fails to do this, a notification is sent back to the Health Monitor (see the fifth arrow in Figure 1) which then initiates its native error recovery procedure. In fact, the race healer considers only asymmetric races [14] although the race detector can detect general race patterns. However, for this system to work well, the Health Monitor recovery tables have to be modified with new error codes to handle data races and to define emergency actions to be taken when the race cannot be healed.

When invoked, the race healer process obtains higher priority than the other processes in the system. This architecture fits the layering concept of ARINC-653 and permits to detach entirely the race healing function from the race detection. This mechanism can be implemented within a partition to handle intra-partition concurrency or within a module to check inter-partition communication.

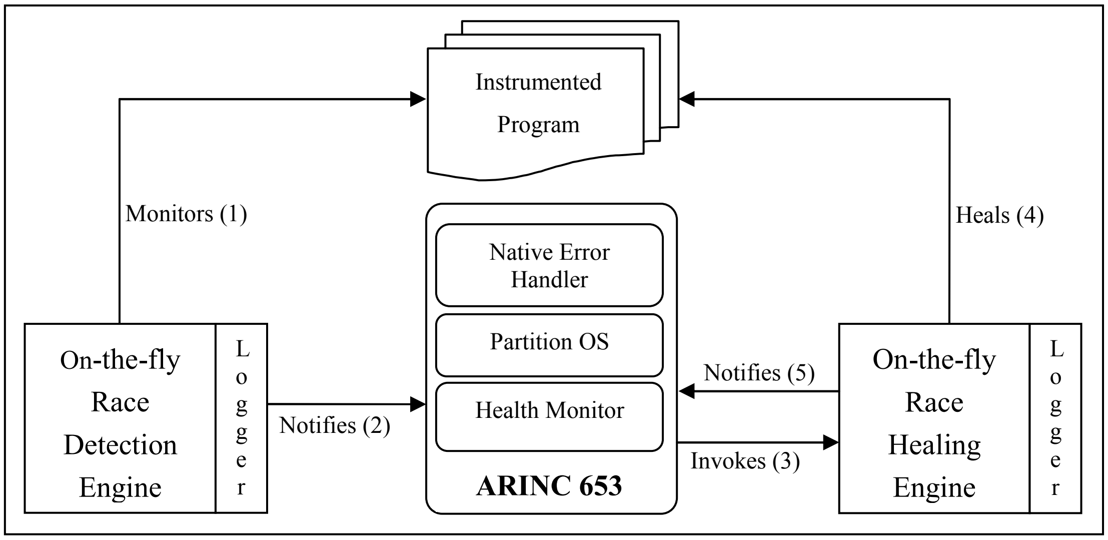


Figure 1. Overall architecture of race healing in ARINC-653

3. Race healing framework in SIMA

This section presents an implementation of the race healing framework and describes its incorporation into the SIMA platform.

3.1. The SIMA environment

The ARINC-653 services are provided by the Simulated Integrated Modular Avionics (SIMA) [13]. SIMA is an execution environment, providing the ARINC-653 application programming interface to operating systems like Linux that do not support such services. SIMA simulates ARINC-653 partitions, modules, and health monitor and provides mechanisms for inter-partition communication and logging. SIMA is designed to run on all Posix-compliant operating systems.

The SIMA environment simulates the two-layered architecture of most operating systems compatible with the integrated modular avionics specification. SIMA is developed as C libraries. The partition operating system is provided as a library called *pos.a* that is statically linked to the user application code. This library contains the ARINC-653 API. The SIMA module operating system is implemented by a program called *mos* that contains the partition scheduler, configured in the ARINC-653 XML configuration file. SIMA implements Part 1 [1] and most of the Extended Services of Part 2 [2] of the ARINC-653 specification. SIMA maps ARINC-653 partitions to POSIX processes and ARINC-653 processes to POSIX threads, allowing each SIMA application to be linked to its own POSIX program.

3.2. Race detection protocol

For on-the-fly race detection in our framework, we used the protocol presented by Dinning and Schonberg [9]. This protocol supports a program model with thread locking and guarantees to detect at least one race for each shared variable, if any exists. The protocol defines the structure and the management policy of an access history for programs with locking mechanism. The access history consists of four sets: *Reads, Writes, CS-Reads*, and *CS-Writes* sets, where CS stands for critical section. The *Reads* and *Writes* sets are for accesses outside critical sections, while the *CS-Reads* and *CS-Writes* sets are for accesses inside critical sections. Each entry is a pair <*label, locks*> for every monitored access. On access to a shared variable, according to which set the access belongs to, data races are reported according to the following policy:

- *CS-Read*: Data races are reported by comparing the concurrency information attached to the current access with those of prior accesses in *Writes* and *CS-Writes*.
- *CS-Write*: Data races are reported between the current access and previous accesses in *Writes* and *Reads* sets.
- *Read*: reports races with all previous accesses in *Writes* and *CS-Writes*.
- *Write*: Data races are reported between the current access and previous accesses in all other sets.

To generate logical concurrency information for threads, we used the Nest Region (NR) labeling [12] engine which generates a unique identifier for every thread during the monitored execution of a parallel program. NR labeling supports program model with nested parallelism and provides a constant-sized label for each entry of an access history. The labeling generates concurrency information using nest regions for nested threads, so that the complexity does not depend on the maximum parallelism of program. Because the labeling maintains an ordered list of parent threads for each thread, a binary search method can be used for the comparison of concurrency information. Thus, NR labeling generates small overhead for race detection.

The detection engine monitors accesses to shared variables and reports races by analyzing their labels and lock status. It consists of the following four modules and data structures for the maintenance of the access history: (1) *R-Checker* and *Read* access history for a read access without proper locking, (2) *W-Checker* and *Write* access history for a write access without proper locking, (3) *CSR-Checker* and *CS-Read* access history for a read access protected by a locking mechanism, (4) *CSW-Checker* and *CS-Write* access history for a write access protected by a locking mechanism. Upon race detection, the SIMA health monitor is notified by the race detector using the provided system call RAISE_APPLICATION_ERROR.

3.3. Race healing protocol

Generally, it is hard to heal data races on-the-fly, because this requires detecting data races and its localization, identifying error patterns, and choosing a proper healing method. However, it may be simple to heal the racing code if the pattern is well specified such as asymmetric races [14]. This section presents our algorithm for on-the-fly healing of asymmetric races implemented in SIMA.

To heal data races, our technique inserts delay operations before an access which is about to be involved in a race. Figure 2 shows the healing technique implemented in our framework. In this figure, *r* and *w* in the circle represent the read and write accesses respectively. The numbers attached to the accesses represent an observed order. Figure 2(a) illustrates an example of data race. Data races are detected early since instructions accessing a shared variable are monitored before their execution. The healing function is registered in each monitored program as its error handler. This is done using the SIMA system call CREATE_ERROR_HANDLER.

Let's assume in Figure 2(a) that *Thread B* preempts *Thread A* right when *Thread A* is about to perform its write operation. The race detector will detect a potential data race between the *read* operation in *Thread B* and the *write* in *Thread A*. The error handler that corresponds to the healing function is then invoked. In the SIMA environment, the error handler once invoked, accesses the racing code and inserts at the race detection point the SIMA system call *WAIT_EVENT*. Thus, this system call becomes the next instruction be executed by *Thread B*. This *WAIT_EVENT* system call forces the racing process to wait for a notification. This mechanism is shown in Figure 2(a) and (b). On lock release, a racing process executes the SIMA system call *SET_EVENT* to signal other waiting processes. Processes blocked because of a race can then resume and continue their execution. As shown in Figure 2(b), *Thread B* continues its execution only when it has received a notification from *Thread A* after the critical section. This mechanism allows Threads *A* and *B* to execute *safely* as if they were all enforcing a strict locking discipline as presented in Figure 2(c).

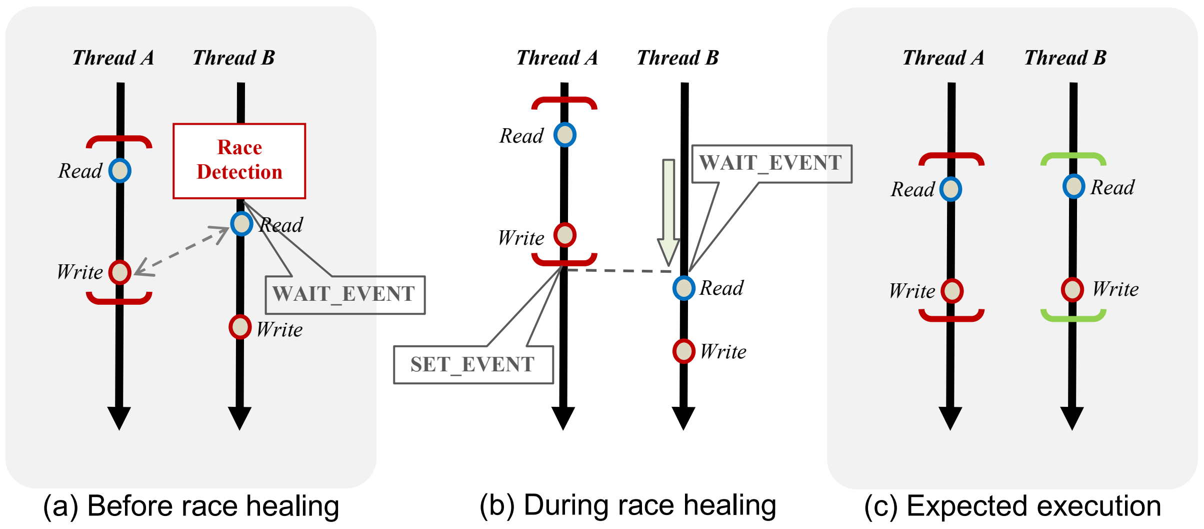
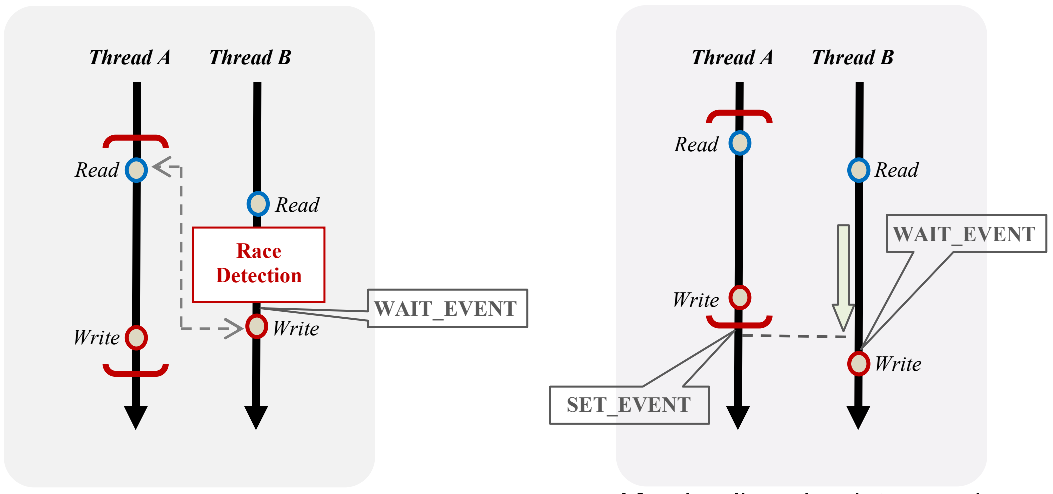


Figure 2. An example of race healing

4. Race healing guarantee

On-the-fly healing of data races is a promising way to help providing *reliable* multithreaded applications. Despite all the progress in the testing and verification technology, it still impossible to detect all data races in a given program. However, healing data races on-the-fly as presented in this article can introduce new bugs in the program. Moreover, there is no real guarantee that all detected data races will be healed during an execution. As an example, let us consider again the case depicted by Figure 2(a) and let us assume that there are other instructions in between the *read* and the *write* operations. Now, we assume that *Thread B* preempts *Thread A* somewhere between the *read* and the *write* operations and before the *write* is monitored by the race detector as shown in Figure 3(a). The race detector will only report a race between the *write* in *Thread B* and the *read* in *Thread A*. Consequently, as shown in Figure 3(b), the healing mechanism will not produce the prospective result. Another problem with this technique is that the notification event can be lost making other threads to wait forever.

As proposed in [15], we believe that the use of suitable *formal verification* techniques like *model checking* or *static analysis* [16,17] can help ensuring that a new bug will not be introduced in the program by actions of the healing protocol. However, we should also consider the fact that these verification techniques cannot check the complete state space of a given program due to the exponential number of possible paths to be analyzed.



(a) Before race healing (b) After healing: the data race is not removed

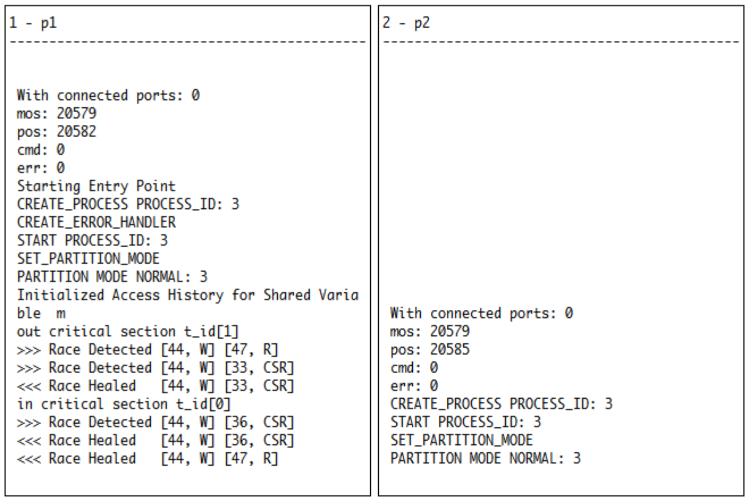
Figure 3. Example of an unhealed data race

5. Efficiency and configurability

We evaluated the efficiency of the framework on a set of synthetic OpenMP programs. Each program contains 100 threads and presents at least one possible asymmetric race on a shared variable. Programs are divided into three sets, each with five programs based on the following criteria: there is at least one thread in which (A) all accesses to the shared variable use no lock, (B) at least one access is unprotected before a critical section, and (C) at least one access is unprotected after a critical section. We implemented five programs for each category. To exhibit the non-determinism, each thread in each program has at least one write access to the shared variable. Indeed, these three groups of programs reflect the common cases of asymmetric race patterns in real-world projects.

We configured SIMA to run two partitions called P1 and P2. Each partition runs the synthetic programs described above. Only Partition P1 embeds our healing framework. Upon race detection, the SIMA health monitor is notified by the race detector using the provided system call RAISE_APPLICATION_ERROR. This system call provokes the execution of the registered error handler that is the healing function. Figure 4 shows an output of the two partitions P1 and P2 in the SIMA environment. The output of partition P1 reveals that three data races [44,W][47,R], [44,W][33,CSR], [44,W][36,CSR] are reported and healed. Each reported race is a pair of line number and access event between two threads. For example, [44,W][47,R] represents a data race that happened between a write access at line 44 in the first thread and a read access at line 47 in the second thread. Moreover, CSR stands for a read access inside a critical section.

We measured and compared the total execution time of our synthetic programs as shown in Figure 5. The dark bars represent for each group of programs, the execution time measured in partition P2. On the other hand, the gray bars show the execution time when the programs are monitored by our framework in partition P1. The result of Figure 4 shows that our technique slows down in average about 2 times the original program execution.



```
1 - p1
--------------------------------------------


 With connected ports: 0
 mos: 20579
 pos: 20582
 cmd: 0
 err: 0
 Starting Entry Point
 CREATE_PROCESS PROCESS_ID: 3
 CREATE_ERROR_HANDLER
 START PROCESS_ID: 3
 SET_PARTITION_MODE
 PARTITION MODE NORMAL: 3
 Initialized Access History for Shared Varia
 ble  m
 out critical section t_id[1]
 >>> Race Detected [44, W] [47, R]
 >>> Race Detected [44, W] [33, CSR]
 <<< Race Healed   [44, W] [33, CSR]
 in critical section t_id[0]
 >>> Race Detected [44, W] [36, CSR]
 <<< Race Healed   [44, W] [36, CSR]
 <<< Race Healed   [44, W] [47, R]
```

```
2 - p2
--------------------------------------------

 With connected ports: 0
 mos: 20579
 pos: 20585
 cmd: 0
 err: 0
 CREATE_PROCESS PROCESS_ID: 3
 START PROCESS_ID: 3
 SET_PARTITION_MODE
 PARTITION MODE NORMAL: 3
```

Figure 4. Output of SIMA partitions

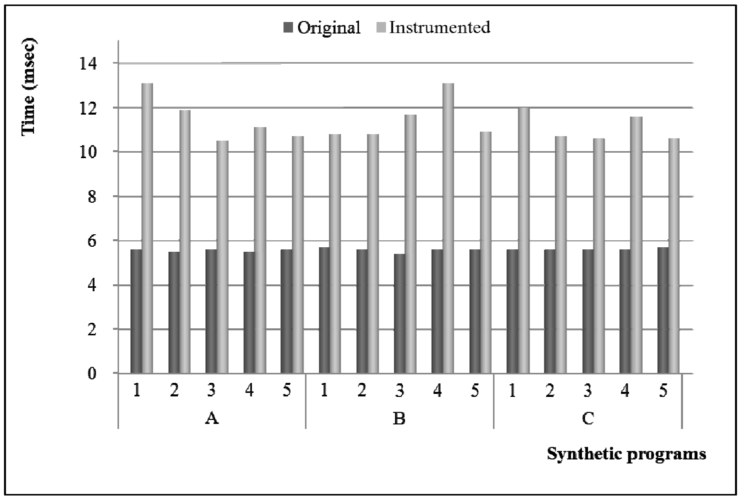


Figure 5. Race detection overhead

The results show that the proposed framework is configurable under an ARINC-653 platform. Indeed, the framework slows down the monitored application by about 2 times. ARINC-653 uses a XML formatted file to configure each partition. This file defines the scheduling time and the scheduler for each partition and its processes. This file is configured by the system designer for a better integrity of the whole system.

To meet the real-time requirements of each partition and its applications, the XML configuration file must take into consideration our results. Thus, the execution time for a partition running our framework should be at least 2 times the original execution time. SIMA supports and provides the ARINC-653 configuration file.

6. Related work

Data races and other timing-dependent bugs like deadlocks are harmful for the program since they are notoriously hard to reproduce, debug and remove. Thus, healing data races becomes an important research issue especially for highly critical software. Algorithms are proposed in [15, 18], to detect and heal data races and atomicity violation bugs in Java programs by influencing the Java Virtual Machine or by injecting new locks into the application. Ratanaworabhan et al. [14] presented ToleRace, a tool for detecting and tolerating asymmetric races in multithreaded programs by influencing the scheduler. ToleRace instruments critical sections to maintain local copies of shared variables. Data race detection is performed on accessed shared variables at the end of each critical section. If a detected data race is a *read-write* race, ToleRace can heal it by replacing the content of the corrupted variable with the expected value based on the local copies of shared variables. However, ToleRace only reports *write-write* data races since it cannot heal them.

Goldberg and Horvath [19] present an enhancement of the ARINC-653 Health Monitor with a new facility of software fault protection. This work proposes to monitor and analyze the behavior of instrumented software against a predefined model at runtime and uses the ARINC-653 health monitoring capability to provide responses to detected faults. The model consists of behavioral properties, failure modes, and responses. Authors also show how models can be defined and how the overall architecture can be integrated into the ARINC-653 Health Monitor by extending the content of the configuration file. However, this enhancement of the ARINC-653 health monitor does not consider detecting and healing data races. Therefore, our framework can be regarded as a complement to their work.

7. Conclusion

The ARINC-653 standard architecture for flight software specifies an application executive which furnishes an application programming interface and defines a hierarchical health management framework for error detection and recovery. This application executive provides buffers, blackboards, semaphores and events for intra- and inter-partition communication and synchronization. Unfortunately, misusing these facilities may produce concurrency bugs such as deadlocks and data races that can be fatal for highly sensitive applications like flight software. In this article, we presented an implementation and configuration under a simulated ARINC-653 platform, of a race healing framework. The efficiency of the framework shows that it is possible to reinforce the ARINC-653 health management function with a race healing mechanism in order to increase the reliability of flight software.

Acknowledgements

This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency), NIPA-2010-(C1090-1031-0007).

References

[1] Airlines Electronic Engineering Committee (AEEC), “Avionics Application Software Standard Interface - ARINC Specification 653 - Part 1. (Supplement 2 - Required Services)”, ARINC, 2006.

[2] Airlines Electronic Engineering Committee (AEEC), “Avionics Application Software Standard Interface - ARINC Specification 653 - Part 2. Extended Services”, ARINC, 2008.

[3] P. J. Prisaznuk, “ARINC-653 Role in Integrated Modular Avionics (IMA)”, Proceedings of the 27th Digital Avionics Conference, IEEE, Minnesota, 2008, pp.1E2.1-7.

[4] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, “A Portable ARINC-653 Standard Interface”, Proceedings of the 27th Digital Avionics Conference, IEEE, Minnesota, 2008, pp.1E2.1-7.

[5] A. Jannesari, and W. F. Tichy, “On-the-fly Race Detection in Multi-threaded Programs”, Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, ACM, Seattle, July 2008, pp.1-10.

[6] R. H. B. Netzer, and B. P. Miller, “What Are Race Conditions? Some Issues and Formalizations”, Proceedings of the ACM Letters on Programming Languages and Systems, ACM, March 1992, pp. 1(1),74-88.

[7] O.-K. Ha, G. M. Tchamgoue, J.-B. Suh, and Y.-K. Jun, “On-the-fly Healing of Race Conditions in ARINC-653 Flight Software”, Proceedings of the 29th Digital Avionics Conference, IEEE, Salt Lake City, 2010, pp.(5.A.6)1-11.

[8] G. M. Tchamgoue, I.-B. Kuh, O.-K. Ha, K.-H. Kim, and Y.-K. Jun, "A Race Healing Framework in Simulated ARINC-653", Proceedings of the 2010 International Conference on Future Generation Communication and Networking (FGCN'10), Springer-Verlag, December 2010, CCIS 120, Part II, pp.238-246.

[9] A. Dinning, and E. Schonberg, “Detecting Access Anomalies in Programs with Critical Sections”, Proceedings of the ACM/ONR workshop on Parallel and Distributed Debugging, ACM, New York 1991, pp.85-96.

[10] R. Baldoni, and M. Raynal, “Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems”, IEEE Distributed Systems Online, IEEE, 2002, pp.1-25.

[11] C. J. Fidge, “Logical Time in Distributed Computing Systems”, Computer, ACM, 1991, pp.28-33.

[12] Y. Jun, and K. Koh, “On-The-Fly Detection of Access Anomalies in Nested Parallel Loops”, Proceedings of the ACM/ONR workshop on Parallel and Distributed Debugging, ACM, California, 1993, pp.107-117.

[13] T. Schoofs, S. Santos, C. Tatibana, J. Anjos, “An Integrated Modular Avionics Development Environment”, Proceedings of the 28th Digital Avionics Systems Conference, IEEE, Orlando, October 2009, pp.(1.A.1)1-9.

[14] P. Ratanaworabhan, M. Burstscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman, “Detecting and Tolerating Asymmetric Races”, Proceedings of the Principles and Practices of Parallel Programming, ACM, 2009, pp.173-184.

[15] B. Krena, Z. Letko, R. Tzoref, S. Ur and T. Vojnar, “Healing Data Races On-the-fly”, Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging, ACM, London, 2007, pp.54-64.

[16] S. Qadeer and D. Wu, “KISS: Keep It Simple and Sequential”, Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation, ACM, June 2004, pp.14-24

[17] T. Ball and S. K. Rajamani, “The SLAM Project: Debugging System Software via Static Analysis”, Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’02), ACM, January 2002, pp.1-3.
[18] Z. Letko, T. Vojnar and B. Krena, “AtomRace: Data Race and Atomicity Violation Detector and Healer”, Parallel and Distributed Systems: Testing and Debugging, ACM, Washington, 2008.
[19] Goldberg and G. Horvath, March 2007, “Software Fault Protection with ARINC 653”, Aerospace Conferences, IEEE, Montana, 2007, pp.1-11.

Authors

Guy Martin Tchamgoue is currently enrolled as a PhD candidate with the Department of Informatics in Gyeongsang National University, South Korea. He received his Bachelor of Science, Master of Science and Master of Advanced Studies in Computer Science from the University of Yaoundé I in Cameroon in 2004, 2005 and 2006 respectively. He worked as IT manager for several years in a Cameroonian enterprise. His research interests include event-driven programming and debugging, parallel and distributed computing, operating systems and real-time systems.

Ok-Kyoon Ha received his BS degree in Computer Science under the Bachelor’s Degree Examination Law for Self-Education from National Institute for Lifelong Education and the Master of Science from Gyeongsang National Unversity, South Korea. He is currently enrolled as a PhD candidate with the Department of Informatics in GNU. He had worked as the manager of IT department in Korea industry for several years. His research interests include parallel/distributed programming and debugging, embedded system programming and operating systems. Mr. Ha is a member of the Korean Institute of Information Technology (KIIT) and the Korean Institute of Information Scientists and Engineers (KIISE).

Kyong-Hoon Kim received his B.S., M.S., and Ph.D. degrees in Computer Science and Engineering from POSTECH, Korea, in 1998, 2000, 2005, respectively. Since 2007, he has been an assistant professor at the Department of Informatics, Gyeongsang National University, Jinju, Korea. From 2005 to 2007, he was a post-doctoral research fellow at CLOUDS lab in the Department of Computer Science and Software Engineering, the University of Melbourne, Australia. His research interests include real-time systems, Grid and Cloud computing, and security.

Yong-Kee Jun received his BE degree in Computer Engineering from Kyungpook National University in 1980. He obtained his Master and PhD degrees in Computer Science from the Seoul National University in 1982 and 1993 respectively. He is now a Full Professor in the Department of Informatics, Gyeongsang National University (GNU), where he had served as the first director of the GNU Research Institute of Computer and Information Communication (RICIC), and as the first operating director of the GNU Virtual College. He is also the head the Embedded Software Center for Avionics (GESCA), a national IT research Center (ITRC) in South Korea. As a scholar, he has produced both domestic and international publications developed by some professional interests including parallel/distributed computing, embedded systems, and systems software. Professor Jun is a member of the Association for Computing Machinery (ACM) and the IEEE Computer Society.