A Hierarchical Fault Detection Method for Aerospace Embedded Software

Cangzhou Yuan[1(✉)], Kangzhao Wu[1], Ran Peng[1], and Panpan Zhan[2]

[1] School of Software, Beihang University, Beijing 100191, China
{yuancz,smbody,ran987}@buaa.edu.cn

[2] Beijing Institute of Spacecraft System Engineering, Beijing 100094, China
panpan3210@qq.com

Abstract. Monolithic fault detection methods have low accuracy for comprehensive faults detection, because they can only detect specific type of faults. Therefore, it is necessary to study the combination of fault detection methods to improve the detection accuracy. In this paper, based on the fault propagation on component-based aerospace embedded software architecture, we analyze occur reasons, manifestations and effects of instruction-, component- and system-level faults whose root cause is single event upset (SEU) which is the main reason of aerospace embedded software, and propose a hierarchical fault model to specify characteristics of the three levels faults. And based on the hierarchical fault model, a hierarchical detection method is proposed to combine the three levels monolithic fault detection methods. The experimental results show that the hierarchical fault detection method has higher fault detection accuracy than the monolithic fault detection methods for comprehensive faults detection.

Keywords: Hierarchical fault model · Hierarchical fault detection · Single event upset

1 Introduction

Generally speaking, the execution of space missions has extremely high-reliability requirements for spacecraft hardware and software. However, due to the harsh conditions of strong solar radiation, low temperature, and ion radiation in the space environment, spacecraft hardware and software are likely to be damaged or malfunction, which has extreme risks for the successful execution of space missions [1].

To meet the reliability, safety, and many other requirements of spacecraft software operation, researchers have launched a large number of fault detecting and processing technology researches [1–3], including software fault mechanism, fault detecting, processing methods, and fault processing architecture.

There is strong cosmic radiation in the space environment, which is easy to generate single event upset (SEU) [4], that is, a single high-energy particle in the

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
Q. Wu et al. (Eds.): WiSATS 2020, LNICST 357, pp. 124–138, 2021.
https://doi.org/10.1007/978-3-030-69069-4_11

universe shoots into the sensitive area of semiconductor devices, which causes the logical state of devices to flip. Once SEU occurs, it may lead to a fault, including silent data corruption (SDC), detected unrecoverable error (DUE), prolong, exception or crash. In this paper, a fault is regarded as a situation where at least one attribute of the system doesn't conform to its expected behavior. When the impact of the fault manifests, the system generates an error [5]. In order to decrease complexity of analysis, we just consider SDC, prolong, exception and crash caused by SEU. It is reasonable to simplify because the hierarchical fault model and hierarchical fault method are adapt to other faults at the instruction-, component- and system-level that the faults we consider can be classified.

Under the harsh conditions, even the redundancy method can not assure 100% detection accuracy. The experimental results of Reis GA et al. [6] show that even if the three redundancy method is used, the SEU can not be detected to 100%. It can be seen that the fault domain caused by SEU is infinite and unpredictable, and the monolithic fault detection method still can be improved.

One way to improve the monolithic fault detection method is to use the comprehensive faults detection method, but there are two challenges: first, how to know the faults contained in the fault domain as fully as possible. It is very convenient for us to design fault detection methods if know what specific faults to be detected, for the specific methods can be used to detect specific faults. Second, how to determine the combination of fault detection methods. As a fault detection method only detects a specific fault domain, if the fault domain of two fault detection methods overlaps greatly, such combination is inefficient.

To solve the first challenge, a hierarchical fault model is established to analyze the fault types and characteristics in the single-particle inversion fault domain, which provides a clear fault target for the hierarchical fault detection method. To solve the second challenge, a hierarchical fault detection method based on the hierarchical fault model is proposed to improve the coverage of the fault domain. The result shows that the hierarchical fault detection method has a higher fault detection rate than the monolithic fault detection methods for comprehensive faults detection.

2 Related Work

Generally speaking, the hierarchical model, the fault propagation model and the fault detection method are three research aspects in fault detection. The hierarchical model focuses on analysing fault in different levels, generally component- and system-level. The fault propagation model focuses on analysing how faults propagate along with instruction dependency, component dependency or other types dependency. The fault detection method focuses on how to detect the target fault in run-time, aiming to high accuracy and cost-effective. In this paper, we try to consider the three aspects comprehensively.

2.1 Hierarchical Model

Based on individual components and components dependency which corresponding to system-level, T. Pitakrat et al. [7] proposed an architecture-aware approach to improve predicition quality. Their result shows that HORA improves the overall area under ROC Curve (AUC) by 10.7% compared to a monolithic approach.

Gao Xiang et al. [8] divided system into serial levels functions to build a network diagram of signal propagation among functions for vulnerable analysis. They regarded system as combination of hierarchical functions.

Kalbarczyk et al. [9] modeled the impact of faults on software behavior by simulating fault propagation at different levels of hardware step by step, including physics-level, transistor-level, logic-level, chip-level and hardware-level.

Savor and Seviora [10] proposed a hierarchical supervisor that has the path-detection layer (PDL) and the base supervisor layer (BSupL) to detect software failures. BSupL receives execution path information from the PDL and checks detail behavior of software.

In our method, we consider instruction-level, component-level and system-level to model SEU propagation.

2.2 Fault Propagation Model

Most fault propagation models are based on fault injection with mathematical analysis and estimation [11]. Researchers often model fault propagation in a probabilistic way for quantitative analysis [12].

Abdelmoez et al. [13] analyzed fault propagation at system design stage based on system states and message. They defined the propagation probability between two components. Hiller et al. [14] introduced the error permeability to represent the propagation probability from a signal to another signal. This paper just analyzes fault propagation between different levels qualitatively, because we only consider how the fault propagates and what it manifests.

Avizienis et al. [15] described the process of system failure caused by component failure. If there is a defect in the code implementation of an internal active component when this part of the code is executed, the error will lead to an internal failure of the component; once the fault reaches the interface of the component, it will cause the component to fail. Based on this process, we think the system fault states caused by SEUes are the same as those caused by defect codes. So that SEU may lead to component or system failure.

Based on empirical observations of error propagations in programs, Guanpeng Li et al. [16] construct a three-level model to capture error propagation at the static data dependency, control flow, and memory levels. This model can predict the overall SDC probabilities and the SDC probabilities of instructions without fault injection, and the accuracy is close to fault injection, while the speed and scalability are better than fault injection. They only studied fault propagation in instruction-level by analyzing each bit's propagation probability. Our approach not only considers how the SEU propagates from instructions to components, but also how propagates from components to systems.

2.3 Fault Detection Method

In instruction-level, signatured instruction streams (SIS) [17] calculates CRC signature of each basic block of code segment, then recalculates signature and check it in run-time. Except for the code segment, the operands of instructions may generate SEU. Error detection by duplicated instructions (EDDI) [18] make another copy of instructions in one thread to cover the code and data segment. But the control flow isn't covered by EDDI. So Reis G et al. [19] proposed software implemented fault tolerance (SWIFT), using signature to cover control flow and redundancy to cover data flow.

In component-level, Huang et al. [20] proposed software rejuvenation, restarting the longest running component based some rules, to prevent software aging [21].

In system-level, Antonio Rodrigues et al. [22] proposed a platform to derive actionable insights from monitored metrics in distributed systems. They can filter out unimportant metrics and infer metrics dependencies between distributed components of the system. The result shows that they can reduce the number of metrics by at least an order of magnitude ($10 - 100$x) and improve existing monitoring infrastructures.

However, the common feature of the above methods is that they only studied fault detection for the faults in one level, without considering their propagation stage, manifestation, and severity. One-level methods cannot process all the faults, so we consider these three levels. The higher level can detect faults not detected by lower level.

3 Hierarchical Fault Model

As one of the main sources of spacecraft software faults, the soft errors caused by SEU will cause various types of software faults. Because of the differences in the propagation stage and manifestation of soft error, it is necessary to establish a hierarchical fault model for different stages and types of software faults caused by soft errors.

Nowadays the design trend of aerospace software architecture is component-based, which makes aerospace software rapidly developed and highly reusable. Because the aerospace software is real time required, it often consists of simple software structure. Sometimes its entire system has little subsystems and components. We do not consider subsystems because the number of subsystems is too little to build analysis in a aerospace software. Based on the architecture elements, aerospace software system can be divided into three levels: instruction, component and system. Instruction is the smallest code that can be executed by CPU. A set of instructions with specific logic rules forms a component. Similarly, a set of components with specific connection rules forms a system.

Because of the uncertainty of SEU, the fault may occur at anywhere, and the lower level fault will propagate along the data or control flow to the higher level and cause higher level fault. Generally, high-level and low-level fault have different manifestations, and higher level the fault occurs on, more serious the harm is. In this paper, a hierarchical fault model is established based on the following two reasons. One is that if the fault is detected at the source when it occurs, the propagation of the fault can be prevented, so as to minimize the harm of the fault. It is easier to distinguish the characteristics and detect of each level fault by using hierarchical fault model. Another is that the fault detection system can also organize hierarchical fault detection methods according to the hierarchical model. The fault detection system architecture is compatible with the aerospace software system architecture, which is instruction, component and system. And it is easy to deploy fault detection system into aerospace software system.

3.1 Instruction-Level Analysis

The first stage is instruction-level, where generates soft errors. There are a large number of high-energy particles in the space environment. These particles come from all kinds of radiation in the universe. When these particles pass through semiconductor devices, they will produce electron-hole pairs. When accumulated to a certain amount, they can reverse the state of logic devices [5]. These logical devices can be registers, cache, memory. The flip of the logical device state is called SEU, and the most direct effect is to cause a bit inversion of the data segment (DS) or code segment (CS).

According to whether it is detected by the system, SEU can be divided into two types, SDC and DUE. SDC refers to the soft errors that are not detected by the system but will affect the system behavior. DUE refers to the soft errors that are detected by the system and may affect the system behavior. The impact of SDC on the system is worse than DUE because it can not be detected. Because instruction-level is the generation stage of soft error, a soft error is in the state of just generated but not started to propagate. If it is detected and recovered when generated, its propagation can be avoided. Since the influence is within the instruction, bit inversion belongs to the instruction-level.

3.2 Component-Level Analysis

When soft errors are not successfully detected or recovered in the generation stage, they may cause exception, prolong, or crash after further propagation along with the data flow or control flow.

Program with soft errors usually enters the wrong control or data flow and then generates exceptions. After the bit of data in control or data, flow is reversed, the program receives the wrong input or produces the wrong output, thus triggering some pre- or post-conditions set by the software developer and resulting in exceptions. For example, if the sign bit of an integer is reversed,

the program detects that the data is out of the input range, resulting in data exception.

The cause of prolonging is similar to exception, but just without the data flow. For example, the data controlling the instruction stream increases because of bit inversion, which will increase the time to complete the task. The specific manifestation of prolonging is that the running time of software doesn't meet the expectation, such as no response for a long time, increasing response time, etc.

Exception and prolong can only be detected by monitoring the status of the components. Once they occur, their influence and impact are more serious than SDCs. Because these two types of faults have prevented the components from completing tasks normally or in time, they are component-level faults.

3.3 System-Level Analysis

There are two main reasons for a soft error to cause a crash. First, the soft error changes the value of pointers, resulting in invalid address and causing a crash. Second, the soft error mentioned above occurs in the code segment, resulting in an illegal instruction, then a crash occurs. In Xin Xu and Man-Lap Li's research [23], they injected address fault and found 88% of faults caused crash. Gu et al. [24] studied Linux behavior under soft errors. They found that 95% crash are caused by null pointer, invalid instruction and page fault. The page fault means the kernel tries to access the bad page so it can be classified in invalid address.

The crash causes the components unable to provide function, which may eventually make the whole system crash. Therefore, when the crash occurs, the influence and impact are the largest, which is a system-level fault. The specific manifestation of crash is that the crashed system is stop, means no interaction with environment. So system crash can be detected by monitoring interactions between system and environment.

3.4 Hierarchical Fault Model Proposed

According to the analysis, a model can be built with fault type, effect level and source-critical. Figure 1 shows the model.

We classify type of faults by its generating level and consequence. First at instruction-level, bit inversion on data segment may have no effect, meaning benign fault, and also may cause crash. But bit inversion on code segment will cause crash in most cases, so we put bit inversion (DS) under bit inversion (CS). Second at component-level, an exception means that the component does not function properly but it can be recovered by retrying. And a prolong means that a component can only be recovered by rebooting. That is why we put exception under prolong. Third at system-level, when system crashed, the effect is system level and system can be only recovered by restarting it, so we put crash on the highest position.

It is necessary to consider source-critical. If the source of the fault is not critical, good fault detection methods, which are usually expensive, needn't be

implemented, and we can focus on the critical part of the system. The number of source-critical is bigger, the source is more critical. The specific meaning of numbers is depended on the definition of the system. In the implement section, 1 means application that will not affect other applications after failure, 2 means fault detection, isolation, and recovery system which will place the system into an unsafe state after failure and 3 means system critical component which will make system failure when it appears failure.

The effect level is easy to understand. The effect level of bit inversion (DS and CS) can be instruction, component or system because it can just affect one instruction, make an exception of component or crash whole system. The effect level of exception and prolong can be component or system because they can not affect instructions but components and cause components or system failure. The effect level of crash is system. Whenever the crash occurs, the system will fall into stopped state immediately.

If a fault can be modeled with the three dimensions, a more appropriate fault detection method can be used.

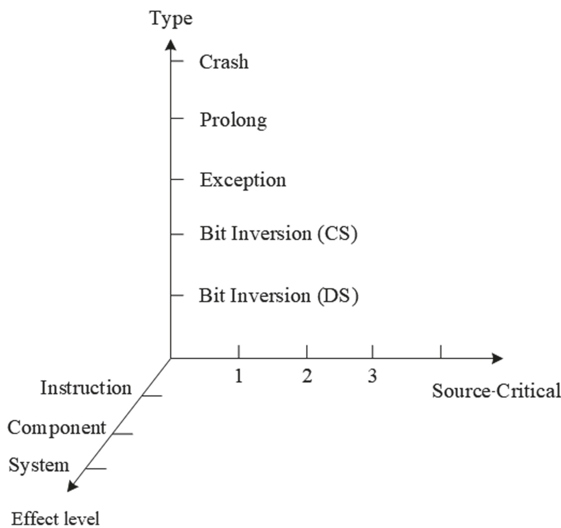

Fig. 1. Hierarchical fault model

4 Hierarchical Fault Detection Method

According to the hierarchical fault model, faults may propagate. If the initial fault can be detected at source and recovery actions are taken, a greater loss can be prevented from the fault propagation, but fault cannot be completely detected by a monolithic fault detection method. Therefore, for the faults caused

by bit inversion, we propose the hierarchical fault detection method based on the hierarchical fault model, aiming to detect different types of faults in different propagation stages to improve the accuracy of fault detection.

4.1 Instruction-Level Method

According to the hierarchical fault model, there are two types of bit inversion in the components. One is that the operands of the instruction are reversed, causing the execution path of the program and the function call sequence changed, which may lead to functional failure. The other is that the code segment is reversed and the instruction is changed, resulting in invalid instruction.

For the bit inversion in operands, triple redundancy is used to detect and recover. Two copies of the parameters passed in and condition expressions in the function are created. When the parameter is used, values of the parameter and its two copies are voted, then select the most value. When a branch is encountered, the outputs of the condition expression and its two copies are calculated, then select the most output. In this way, the bit reversal fault can be prevented from propagating. But how to detect it? When voting values of parameters and outputs of condition expressions, it can be checked whether all values or outputs are the same. If not, the bit inversion is detected, then the value or output with the most votes is used to recover the wrong one.

There are remain two questions: why not use double or more redundancy and how to select value and output if all three redundancies are different? First, though the bit inversion can be detected by double redundancy, it can not be recovered. It is cost- and loss-effective if recover the fault when it just occurred. But now, adding just one more redundancy can achieve this goal. Second, it can assume that the probability of bit inversion of one redundancy is P and that when all redundancies occur bit inversion, they must be in different positions each other to simplify our calculation, that is assuming the probability P_d of all redundancies occur in different positions equal to the probability of all redundancies occur bit inversion because that the probability of former is just a little smaller than latter.

According to Aiguo Li et al. [25], the monitoring device experienced 65 times bit reversals in 19 days on orbit. P is calculated about 4e10^{-5} and if there are N-group redundancies, $P_d = P^N$. If $N = 3$, $P_d = 6.4\text{e}10^{-14}$ and $N = 4$, $P_d = 2.56\text{e}10^{-18}$ and so on. If it assume that adding one redundancy will add cost C, then the performance

$$\mathrm{E} = \left(1 - \mathrm{P^N}\right)/\mathrm{NC} \tag{1}$$

which is a minus function. According to the first reason, double redundancy should not be used so we choose triple redundancy which has the best performance. Third, it can be seen that $P_d = 6.4\text{e}10^{-14}$ when $N = 3$, which is a very small number, thus it can assume that bit inversion in all three redundancies will not occur. For the above reasons, we finally choose triple redundancy.

For the bit inversion in the code segment, since the storage area will not change when the code segment is not in a fault state, the check code of code

segment can be calculated to detect the change of code segment. We detect the fault by periodically calculating the cyclic redundancy check (specifically CRC-16) code of the code segment. First, starting a thread to calculate the check code regularly, which takes the first value as the standard check value. Then, the thread will calculate the check value of the code segment in cycle T and compare it with the standard check value. If they are the same, no bit reverse occurs; otherwise, a fault is detected.

4.2 Component-Level Method

Soft errors that are not detected within a function can further propagate to cause an exception and prolong. The method of fault detection at the instruction-level can not guarantee that there is no fault at all. At the same time, there are other faults of instruction-level, which lead to the generation of exception and prolong. Therefore, to detect faults which instruction-level can not detect and may lead to exception and prolong at the instruction-level, methods are implemented to detect exception and prolong at component-level.

When an exception occurs, the pre- or post-condition set by developers will be violated, so we can input the checked value and check whether there is any violation according to the pre- or post-condition. If so, an exception occurs. Because there are many kinds of exceptions, the pre- or post-conditions should be defined by developers themselves, to detect any exceptions they want to detect, which caused by the missed soft errors at instruction-level, and to prevent its further propagation.

When the prolong occurs, the specific manifestation is that the program continuously executes the instruction in the loop, while the instruction outside will not be executed and the data beyond the scope of the loop instruction will not be modified. Therefore, the heartbeat mechanism can be used to detect prolong. Each component will send a heartbeat report to the heartbeat detection component every cycle of execution. If the interval between two adjacent heartbeats exceeds the threshold, a prolonged is detected. Due to different space, embedded software have different environments, the timeout time needs to be decided by the developers themselves.

4.3 System-Level Method

Similar to the component-level, soft errors not detected at the component-level may cause a crash. The method of fault detection at component-level can not guarantee that there is no fault at all. At the same time, there are other faults of component-level, which lead to the generation of the crash. Therefore, to prevent the system from permanently stopping due to various possible faults, the method is implemented to detect crash at system-level.

When the crash occurs, the software can not run and can not use a method to detect a crash. So the hardware watchdog is used to detect the crash of the software system. Because hardware watchdog doesn't depend on the services provided by the spacecraft software, when the software crash occurs, it doesn't

affect the watchdog. A thread is created to reset the timer of hardware watchdog every cycle of execution, if there is a crash, then the thread will stop and can not reset the timer, which will be zero in a short time. When the timer is zero, the hardware watchdog will be triggered and send a signal, which means that a crash is detected. Others can use this signal to implement some mechanisms of fault recovery. For example, this signal can be used to restart the system automatically or send a crash report to the ground.

5 Implementation

In this section, we implement the hierarchical fault detection method and use fault injection to evaluate its accuracy for comprehensive faults detection. To evaluate accuracy, we use the hierarchical fault detection method to detect comprehensive faults generated by fault injection and compare accuracy with monolithic fault detection methods.

5.1 Target Platform

The hierarchical fault detection method need a target platform to implement it so that it can be sure what the manifestations are of its faults and how to implement the hierarchical detection method on it. The target platform is a spacecraft software middleware, whose architecture is component-based. Every component running on this middleware is a thread, and the middleware supports dynamic loading and unloading components when running. All components follow the publish/subscribe pattern to communicate with each other. There are three reasons why choose the component-based middleware. First, it has no subsystem so that it can be divided into instruction-, component- and system-level, which adapt to the hierarchical fault detection method. Second, it has a well defined and implemented communication mechanism that will help to record the detection result. Third, as a middleware, it shields details of operating system and hardware that are not our concern. The entire middleware will run with Linux on the SPARC hardware.

5.2 Way to Implement the Hierarchical Method

Implement for Bit Inversion on Data Segment. Because this method is instruction-level, it is platform independent. We implement it by insert redundancy code that also adapt to other softwares. For two reasons, we just choose three functions of one of its core components, which means it is very critical to implement this method. First, too much implementation of triple redundancy will affect performance of the middleware and it is time-consuming to implement. It can perform well just implemented on core components to prevent SEU propagating and cause serious effect. Second, it make against for the fault injection if all components implement the method. Because we just inject SEU to

cause other faults and triple redundancy can prevent most SEU propagating to cause other faults, it will be inconvenient to evaluate accuracy of other methods.

Implement for Bit Inversion on Code Segment. Because we use CRC16 to calculate the check code, it is necessary to know start address and length of each component and have access to read. Luckily, this middleware provides memory management for its components. Based on the memory management, a component is implemented to calculate the check code of each component with CRC16.

Implement for Exception. For exception detection, we need to know when and where the exception occurs and what exception it is. Components of the middleware will send an exception event when occurs an exception, so we can know the information of this exception. Based on the middleware's event management, a component is implemented to detect exception events.

Implement for Prolong. For prolong detection, we need to know the states of components. Component of the middleware will increase its loop counter by one every loop, which can seen as a heart beat. Based on this rule, a component is implemented to monitor whether the loop counter of each component is increased in a fixed time interval. The interval is set to 10 loops of the heart beat monitor component so that it is neither sensitive too much nor insensitive too much.

Implement for Crash. Crash is a little different. When crash occurs, it is hardware watchdog who detect the crash, not components of the middleware. So we record the detection result by manual according to whether the middleware restarts after crash. To active hardware watchdog, a component is implemented to reset the hardware watchdog timer based on the hardware management of the middleware.

5.3 Fault Injection Method

Fault injection is an effective and convenient method to introduce fault into the middleware. To simulate the SEU propagation process causing bit inversion on data segment and code segment, exception, prolong and crash, the base rule to inject fault is that we just inverse one bit of a random byte. To know what specific fault one injection causes, the random injection is not so 'random'. We design some injections that will cause specific faults known by us, and we will inject faults from the designed injections randomly so that we know what faults we inject.

6 Results

We did a total of five groups of fault injection. In each group, we injected bit inversion (DS), bit inversion (CS), exception event, prolong and crash random, and we recorded the detected numbers of each fault. Table 1 shows the number of fault injected and Table 2 shows the number of fault detected.

Table 1. Number of fault injected in each group

Fault Type	Group 1	Group 2	Group 3	Group 4	Group 5
Bit Inversion (DS)	132	324	238	319	317
Bit Inversion (CS)	58	130	107	113	120
Exception	157	189	114	237	275
Prolong	90	89	80	81	73
Crash	45	70	32	55	38
Total	482	802	571	805	823

Table 2. Number of fault detected each group

Fault Type	Group 1	Group 2	Group 3	Group 4	Group 5
Bit Inversion (DS)	126	309	228	306	302
Bit Inversion (CS)	55	126	101	110	117
Exception	157	189	114	236	275
Prolong	90	89	80	81	73
Crash	45	70	32	55	38
Total	473	783	555	788	805

We calculate the detection accuracy of each fault and the total accuracy of each fault detection method. Figure 2 and Fig. 3 shows the result. We find that average detection accuracy of bit inversion(DS) is about 95.6%. Average detection accuracy of bit inversion(CS) is about 96.2%. Respective average detection accuracy of exception, prolong and crash nearly is or just is 100%. Figure 2 also shows detection accuracy of each monolithic method for their specific fault because in our fault injection design, a fault can only be detected by one specific detection method so that other methods will not introduce errors into detection accuracy of this fault.

Figure 3 shows low accuracy (less 40%) of each monolithic method for comprehensive faults detection. Compared Fig. 2 with Fig. 3, we find that each monolithic fault detection method performs well for its fault but poor for comprehensive faults. The hierarchical fault detection method has about 98% accuracy, much bigger than other monolithic methods. It proves that the hierarchical method can combine monolithic methods at different levels to improve detection accuracy for comprehensive faults detection. And we can reasonably derive that if comprehensive faults only contain one fault, the accuracy of hierarchical method is equal to monolithic one.

7 Discussion

In this section, we want to discuss the inaccuracy of bit inversion detection and accuracy of exception, prolong and crash.

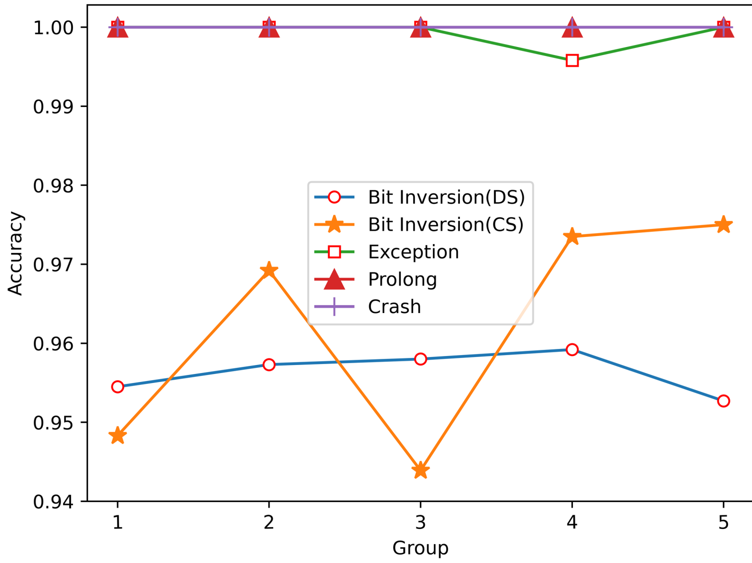

Fig. 2. Each fault detection accuracy of hierarchical detection method

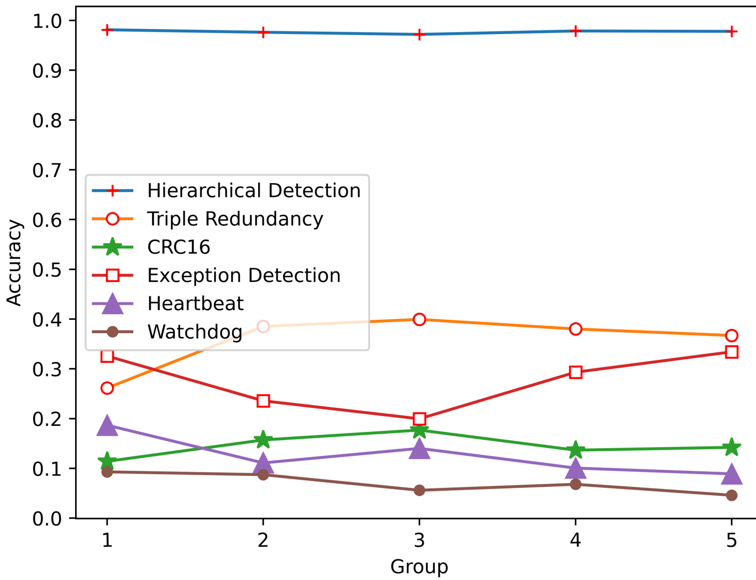

Fig. 3. Comprehensive fault detection accuracy of each detection method

Why Accuracy of Bit Inversion Detection is Not 100%? Because fault injection may occur in any position, some fault may be injected after voting and thus can not be detected by triple redundancy. That is the reason why the detection accuracy of bit inversion(DS) isn't 100%. For the bit inversion(CS), because the CRC check is executed periodically, some fault may be injected after checking and can not be detected.

Why Each Accuracy of Exception, Prolong and Crash Detection is or Nearly is 100%? There are two reasons. One is that exception, prolong and crash are at relatively high levels, where faults are much more easily detected than instruction-level. Another is that the fault injection method can just simulate part of space environment effect and the injection number is maybe little so that detection accuracy of some faults is 100%.

8 Conclusion

We think the reason why a monolithic fault detection method has low accuracy for detection multi-level faults is that the detection coverage of the monolithic fault detection method is too small for the fault domain of multi-level faults. Thus, for the bit inversion fault domain, the hierarchical fault model is proposed to identify what faults the domain has and the hierarchical fault detection method is proposed to increase the detection coverage which leads to increased accuracy. We propose an idea about fault detection that first to build a hierarchical fault to identify faults in a specific fault domain because of fault propagation and second to design hierarchical fault detection method for each fault identified to increase the accuracy.

Acknowledgements. This paper is partly supported by the Pre-research of Civil Spacecraft Technology (No. B0204).

References

1. Tipaldi, M., Bruenjes, B.: Survey on fault detection, isolation, and recovery strategies in the space domain. J. Aerosp. Inf. Syst. **12**, 235–256 (2015)
2. Tang, M., Ning, H., Li, T.: Design and research of FDIR framework for integrated electronic system on satellite (2010)
3. Jiang, L., Li, H., Yang, G.: Research progress of spacecraft autonomous fault diagnosis technology. J. Astronaut. **30**, 13–17 (2009)
4. Li, A., Hong, B., Wang, S.: Research on software vulnerability identification method based on error propagation analysis (2007)
5. Mukherjee, S.: Architecture Design For Soft Errors. Morgan Kaufmann, Burlington (2011)
6. Reis, G.A., Chang, J., August, D.I.: Automatic instruction-level software-only recovery. IEEE Micro **27**, 36–47 (2007)
7. Pitakrat, T., Okanovic, D., Van Hoorn, A., Grunske, L.: An architecture-aware approach to hierarchical online failure prediction. In: International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA) (2016)

8. Xiang, G., Guochang, Z., Xiaoling, L., et al.: Soft-error hierarchical evaluation algorithm of fault vulnerabilities based on sensitive attributes of single event effect. In: IEEE International Conference on Electronic Measurement and Instruments (2015)
9. Kalbarczyk, Z., Ries, G., Lee, M.S., et al.: Hierarchical approach to accurate fault modeling for system evaluation. In: IEEE International Computer Performance and Dependability Symposium (1998)
10. Savor, T., Seviora, E.: Hierarchical supervisors for automatic detection of software failures. In: International Symposium on Software Reliability Engineering (1997)
11. Morozov, A., Janschek, K.: Probabilistic error propagation model for mechatronic systems. Mechatronics **24**, 1189–1202 (2014)
12. Sarshar, S., Simensen, J.E., Winther, R., et al.: Analysis of error propagation mechanisms between software processes. In: Safety and Reliability Conference (2007)
13. Abdelmoez, W.M., Nassar, D., Shereshevsky, M., et al.: Error propagation in software architectures. In: IEEE Computer Society, IEEE International Symposium on Software Metrics (2004)
14. Hiller, M., Jhumka, A., Suri, N.: An approach for analysing the propagation of data errors in software. In: Proceedings of the 2001 International Conference on Dependable Systems and Networks (2001)
15. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**, 11–33 (2004)
16. Li, G., Pattabiraman, K., Hari, S.K., Sullivan, M., Tsai, T.: Modeling soft error propagation in programs. In: IEEE/IFIP(2018)
17. Schuette, M.A., Shen, J.P.: Processor control flow monitoring using signatured instruction streams. IEEE Trans. Comput. **36**, 264–276 (1987)
18. Oh, N., Shirvani, P.P., McCluskey, E.J.: Error detection by duplicated instructions in super-scalar processors. IEEE Trans. Reliab. **51**, 63–75 (2002)
19. Reis, G.A., Chang, J., Vachharajani, N., et al.: SWIFT: software implemented fault tolerance. In: International Symposium on Code Generation and Optimization (2005)
20. Huang, Y., Kintala, C., Kolettis, N., et al.: Software rejuvenation: analysis, module and applications. In: International Symposium on Fault-Tolerant Computing (1995)
21. Cotroneo, D., Natella, R., Pietrantuono, R., et al.: A survey of software aging and rejuvenation studies. ACM J. Emerg Technol. Comput. Syst. (JETC) **10**, 1–34 (2014)
22. Thalheim, J., Rodrigues, A., Akkus, I.E., et al.: Sieve: actionable insights from monitored metrics in distributed systems. In: The 18th ACM/IFIP/USENIX Middleware Conference (2017)
23. Xu, X., Li, M.L.: Understanding soft error propagation using efficient vulnerability-driven fault injection (2012)
24. Gu, W., Kalbarczyk, Z., Iyer, R.K., Yang, Z.: Characterization of linux kernel behavior under errors. In: International Conference on Dependable Systems and Networks (2003)
25. Li, A., Hong, B., Wang, S.: A soft fault correction algorithm for data stream of onboard computer. J. Astronaut. **28**, 1044–1048 (2007)