Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Fault Tolerant Development in Embedded Systems: Implementation of Fault Injection Methods and Fault Tolerant Policies on the Intel Myriad 2 VPU

Διπλωματικη Εργασια

του

Μπαμπίλη Γεώργιου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθήνα, Ιούνιος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Fault Tolerant Development in Embedded Systems: Implementation of Fault Injection Methods and Fault Tolerant Policies on the Intel Myriad 2 VPU

Διπλωματικη Εργασια

του

Μπαμπίλη Γεώργιου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Ιουνίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π

Αθήνα, Ιούνιος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

....
Γεωργιος Μπαμπιλης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
© 2022 – All rights reserved

Copyright © –All rights reserved Γεώργιος Μπαμπίλης, 2022.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Περίληψη

Τα τελευταία χρόνια υπάρχει έντονη αναφορά στην τέταρτη βιομηχανική επανάσταση. Η επανάσταση αυτή, χαρακτηρίζεται από την τρέχουσα τάση της αυτοματοποίησης της παραγωγής. Στη Βιομηχανία 4.0, ο ρόλος των ενσωματωμένων υπολογιστικών συστημάτων είναι καθοριστικός καθώς τα εργοστάσια απαιτούν συστήματα που να μπορούν να επεξεργαστούν σύνθετα δεδομένα. Για την επανάσταση αυτή εμφανίζεται μια νέα κατηγορία μικροεπεξεργαστών οι μονάδες επεξεργασίας όρασης(VPUs). Η ποίο χαρακτηριστική εκ των οποίων είναι οι μικροεπεξεργαστές **Myriad2** , στην οποία έγινε η παρούσα διπλωματική. Στη διπλωματική αυτή, αποφασίσαμε να μελετήσουμε τη συμπεριφορά διαφόρων error injection και fault tolerance τεχνικών. Οι τεχνικές αυτές εστιάζουν στη:

- Διόρθωση σφάλματος που εντοπίζεται στη μνήμη των συνεπεξεργαστών SHAVE.
- Διόρθωση σφάλματος που εντοπίζεται στα δεδομένα που μεταφέρονται από το LEON OS στους SHAVE.
- Διόρθωση σφάλματος που παρατηρείται σε μεταβλητές που είναι shared μεταξύ των SHAVE και του LEON OS.
- Διόρθωσή σφάλματος που παρατηρείται στα δεδομένα των SHAVE.
- Ανάπτυξη 3 Voting και 5 Voting συστημάτων.

Μελετώντας τα αποτελέσματα μας, παρατηρούμε ότι το υπολογιστικός κόστος που εισάγεται στο σύστημα με τις διάφορες τεχνικές που υλοποιήσαμε τις καθιστά ρεαλιστικές για ένα real time system. Οι τεχνικές διόρθωσης σφάλματος διορθώνουν το μεγαλύτερο ποσοστό των λαθών που εισαγάγαμε τεχνικά. Σε μετροπρογράμματα που το χρονικό overhead είναι συγκρίσιμο με τον χρόνο εκτέλεσης του ίδιου του benchmark παρατηρούμε ότι οι επιλογές που ελέγχουν την ορθότητα εκτέλεσης μέσω hashing είναι οι καλύτερες, ενώ στην περίπτωση που το μετροπρόγραμμα είναι γρήγορο τότε τα voting systems αποτελούν καλύτερες επιλογές αν μας ενδιαφέρει αποκλειστικά ο χρόνος εκτέλεσης. Συγκεκριμένα, στην περίπτωση της δισδιάστατης συνέλιξης το χρονικό πλεόνασμα που τοποθετήθηκε από τις μεθόδους που βασίζονταν στον κατακερματισμό δεν ξεπέρασε το τετραπλάσιο του αρχικού χρόνου εκτέλεσης. Στην περίπτωση του 2D binning υπήρξαν περιπτώσεις που το χρονικό overhead έφτασε μέχρι και 40 φορές το αρχικό. Αντιστοίχως στα συστήματα ψηφοφορίας στην περίπτωση της συνέλιξης αλλά και στο 2D binning η χρονική καθυστέρηση έφτασε μέχρι και 8 φορές τον αρχικό χρόνο εκτέλεσης, στην περίπτωση του 5 voting system.

Λέξεις Κλειδιά

Ετερογένεια, Ετερογενείς Αρχιτεκτονικές, Myriad2, Ενσωματωμένα Συστήματα, Εισαγωγή Σφάλματος, Ανοχή απέναντι σε σφάλματα, Διάστημα.

Abstract

In the past years, there has been a lot of discussion regarding the fourth industrial revolution. This revolution is characterized by the current trend of automation used by production lines. The contribution of embedded systems in industry 4.0 is crucial. Modern factories require high-performance embedded systems capable of processing data in real-time. Thus, a new category of microcomputers emerged called Vision Processing Units (VPUs). The most popular V.P.U is the **Myriad** microprocessor family developed by Intel, **Myriad 2**, used in this thesis, is a manycore, heterogeneous, powerful computing system. The main problem, which Industry 4.0 will have to solve is how embedded systems deployed in industrial environments will continue their intended operation in case of failure of one or more of their components. Erroneous operation under such circumstances could have disastrous consequences. In this thesis, we decided implement various error injection and fault tolerant techniques. The fault tolerant policies created can be seen below:

- Detect and correct error in SHAVE instruction memory.
- Detect and correct error in data transfered to SHAVEs.
- Detect and correct error in SHAVE data.
- Detect and correct variables shared with SHAVEs.
- Voting systems: with 3 and 5 voters.

In our results we observe that the computational overhead inflicted on the system by the fault-tolerant techniques renders them realistic choices for the needs of a system that should operate in real-time. Inspecting the experimental results, we notice that when the run-time of a benchmark is analogous to the time overhead introduced by data-hashing then the hashing methods are a good choice as a fault tolerance policy. When the time overhead is not analogous then hashing methods are not that suitable for fault tolerance. Specifically, in the case of 2d convolution the time overhead does not surpass the original run time by more than 4 times. In the case of 2D binning the overhead in some cases is 40 times more than the original. However, in the aforementioned methods the error percentage after these methods are applied is around 0%. In the case of the voting systems, error still remains especially when more than half of the total SHAVEs are corrupted. The time overhead introduced by the voting systems is from 4 to 8 times the original run time. Therefore, depending on the quality of service we want our system to provide we can choose whether to opt for a hashing method or for a voting system.

Keywords

Heterogeneity, Heterogeneous Architectures, Myriad2, Embedded Systems, Error Injection, Fault Tolerance, Space.

Ευχαριστίες

Ύστερα από πολλή κούραση και κατά τη διάρκεια μίας πανδημίας ένας σημαντικός αλλά και ωραίος κύκλος της ζωής μου έρχεται στο τέλος του. Συνεπώς, θα ήταν άδικο αν δεν απέδιδα τις απαραίτητες ευχαριστίες στα άτομα που συνετέλεσαν στην εκπόνηση της παρούσας εργασίας αλλά και που με βοήθησαν με τη στήριξη τους στα 6 αυτά χρόνια. Αρχικά, θα ήθελα να ευχαριστήσω τον κύριο Σούντρη που μου έδωσε την ευκαιρία να ασχοληθώ με το συγκεκριμένο θέμα καθώς και να βοηθήσω στο εργαστήριο συνολικά. Οι εμπειρίες που έλαβα μέσω αυτής της διαδικασίες είναι πάρα πολλές για να περιγραφούν σε μια σύντομη παράγραφο. Επιπλέον, θα ήθελα να ευχαριστήσω των υποψήφιο διδάκτορα Βασίλη Λεών καθώς και τον μεταδιδακτορικό ερευνητή Γεώργιο Λεντάρη για τη βοήθεια που μου προσέφεραν κατά τη διάρκεια πραγματοποίησης της συγκεκριμένης διπλωματικής εργασίας σε τεχνικό επίπεδο. Τα άτομα αυτά ήταν εκεί για να προσφέρουν τεχνική κατατόπιση όταν αυτή ήταν απαραίτητη και τα ευχαριστώ για αυτό. Έπειτα, θα ήθελα να ευχαριστήσω τους γονείς μου για τη στήριξη που μου προσέφεραν στα χρόνια αυτά. Τέλος, θέλω να ευχαριστήσω τους φίλους μου καθώς και το Study Room που φτιάξαμε που έκαναν τα χρόνια αυτά αξέχαστα. Επίσης, Βασίλη sorry που άργησα στο μπάρμπεκιου αλλά τέλειωσα τη διπλωματική.

Thesis Motivation

The usage of computationally hard algorithms and high data rate instrument in space application leads the space industry to investigate new solutions for on board data processing. Therefore, the usage of low power high performance SOCs are explored. However, apart from a great performance per watt embedded systems in space applications require radiation hard C.P.Us. In the past the most commonly used embedded platform was FPGAs due to their attractive performance per watt ratio. This need for new specialized SOCs led to the usage of alternative SOCs gaining more momentum in space applications. Thus, Vision Processing Units (VPUs) started gaining more traction in the space industry due to their excellence in digital signal processing and artificial intelligence task and their coding ease. Such a V.P.U is Intel Myriad 2 which is the platform used on this thesis. Furthermore, space application require real-time, highly reliable results. All of the above, organically lead to the need of development of fault-tolerant policies in order to amend errors that might occur due to different system parts. Therefore, in this thesis we developed multiple techniques that can be used to provide fault tolerance. Lastly, in this thesis apart from fault tolerance techniques we also developed error injection methods in order to be able to evaluate the performance of our developed policies.

Contents

Περίληψη **1**

Abstract **3**

Ευχαριστίες **5**

Contents **9**

List of Figures **11**

List of Tables **15**

Εκτεταμένη Περίληψη **17**

1 Introduction **55**

2 Development of Fault Injection Techniques on Myriad2 **61**

3 Development of Fault-Tolerant Architectures on Myriad2 **71**

4 Experimental Evaluation **101**

5 Conclusion and Future Work **125**

Bibliography **129**

List of Figures

1 Αξιολόγηση Fault-Tolerant συστήματος. 18
2 Αρχιτεκτονικές επεξεργαστών της οικογενείας Myriad 19
3 Fault Detection: Using CRC to detect data corruption. 23
4 Fault Tolerant Policy 1: Using CRC to detect data corruption LOS reschedules Shave data. 24
5 Fault Tolerant Policy 2: Left: Shaves Self Heal, Right: LOS schedules data to working shaves. 25
6 Fault Tolerant Policy 3: Left: Shaves Self Heal, Right: LOS schedules data to working shaves. 26
7 Fault Tolerant Policy 4: LOS schedules data to working shaves. 27
8 N-voting Systems: 4 Voter System. 29
9 N-voting Systems: 2 Voter System. 30
10 Fault Injection Method: LOS transfers corrupted data to Shaves. 31
11 Fault Injection Method: Shaves corrupt their own data. 32
12 Fault Injection Method: LOS corrupts shared variable. 32
13 Fault Injection Method: LOS corrupts shared memory space. 32
14 Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα για το 2D Convolution 39
15 Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα για το 2D Binning 39
16 LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα για το 2D Convolution 41
17 LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα για το 2D Binning 41
18 Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution 42
19 Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα για το 2D Binning 43
20 SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα για το 2D Convolution 44
21 SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα για το 2D Binning 45
22 LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution 46
23 LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Binning 47

24 SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution . . . 48
25 SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Binning . . . 49
26 3 Voting system: Συγκριτικά αποτελέσματα για το 2D Convolution . . . 50
27 3 Voting system: Συγκριτικά αποτελέσματα για το 2D Binning . . . 51
28 5 Voting system: Συγκριτικά αποτελέσματα για το 2D Convolution . . . 52
29 5 Voting system: Συγκριτικά αποτελέσματα για το 2D Binning . . . 53

1.1 Evaluation of a Fault Tolerant system. . . . 56
1.2 Myriad Architecture Myriad . . . 56

2.1 Fault Injection Method: LOS transfers corrupted data to Shave. . . . 62
2.2 Fault Injection Method: SHAVEs corrupt their own data. . . . 64
2.3 Fault Injection Method: LOS corrupts shared variable . . . 65
2.4 Fault Injection Method: LOS corrupts shared memory space . . . 68

3.1 Fault Detection: using CRC to detect data corruption . . . 73
3.2 Fault Tolerant Policy 1: Using C.R.C to detect data corruption LOS reschedules Shave data . . . 74
3.3 SHAVEs corrupt their data, SHAVEs amend error! . . . 81
3.4 SHAVEs corrupt their data, LOS amends error! . . . 82
3.5 LOS corrupts shared variables, SHAVEs amend error! . . . 84
3.6 LOS corrupts shared variables, LOS amend error! . . . 85
3.7 LOS correct error in instruction memory of SHAVEs by scheduling data to working shaves. . . . 87
3.8 N-voting Systems: 4 Voter System. . . . 89
3.9 N-voting Systems: 2 Voter System. . . . 94

4.1 Myriad 2[1] . . . 101
4.2 Leon OS corrects SHAVE instruction memory harm, results for 2D Convolution . . 109
4.3 Leon OS corrects SHAVE instruction memory harm, results for 2D Binning 109
4.4 LEON OS corrects corrupted data transferred, comparative results for 2D Convolution 111
4.5 LEON OS corrects corrupted data transferred, comparative results for 2D Binning 111
4.6 Leon OS corrects variables shared with SHAVEs, comparative results for 2D Convolution . . . 112
4.7 Leon OS corrects variables shared with SHAVEs, comparative results for 2D Binning 113
4.8 SHAVEs correct variables shared with LEON OS, comparative results for 2D Confolution . . . 114
4.9 SHAVEs correct variables shared with LEON OS, comparative results for 2D Binning 115
4.10 LEON OS corrects data destroyed by SHAVEs, comparative results for 2D Convolution 116
4.11 LEON OS corrects data destroyed by SHAVEs, comparative results for 2D Binning 117
4.12 SHAVEs correct data destroyed by SHAVEs, comparative results for 2D Convolution 118
4.13 SHAVEs correct data destroyed by SHAVEs, comparative results for 2D Binning . 119
4.14 3 Voting system, comparative results for 2D Convolution . . . 120
4.15 3 Voting system, comparative results for 2D Binning . . . 121
4.16 5 Voting system, comparative results for 2D Convolution . . . 122

4.17 5 Voting system, comparative results for 2D Binning 123

List of Tables

1 LEON OS sends corrupted data to SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution . . . 35
2 LEON OS sends corrupted data to SHAVEs: Συγκριτικά αποτελέσματα 2D Binning 35
3 LEON OS corrupts variable shared with SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution . . . 36
4 LEON OS corrupts variable shared with SHAVEs: Συγκριτικά αποτελέσματα 2D Binning . . . 36
5 SHAVEs corrupt their own data: Συγκριτικά αποτελέσματα 2D Convolution . . . 37
6 SHAVEs corrupt their own data: Συγκριτικά αποτελέσματα 2D Binning 37
7 Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα για το 2D Convolution . . . 38
8 Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα 2D Binning . . . 38
9 LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα για το 2D Convolution . . . 40
10 LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα 2D Binning 40
11 Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution . . . 42
12 Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα 2D Binning . . . 42
13 SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα 2D Convolution . . . 44
14 SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα 2D Binning . . . 44
15 LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution . . . 46
16 LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα 2D Binning 46
17 SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution . . . 48
18 SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα 2D Binning 48
19 3 Voting system: Συγκριτικά αποτελέσματα για το 2D Convolution 50
20 3 Voting system: Συγκριτικά αποτελέσματα 2D Binning 50
21 5 Voting system: Συγκριτικά αποτελέσματα για το 2D Convolution 52
22 5 Voting system: Συγκριτικά αποτελέσματα 2D Binning 52

4.1 LEON OS corrupts SHAVE instruction memory: Results of 2D Convolution 104
4.2 LEON OS corrupts SHAVE instruction memory: Results of 2D Binning 104

4.3 LEON OS sends corrupted data to SHAVEs: Results of 2D Convolution 105
4.4 LEON OS sends corrupted data to SHAVEs: Results of 2D Binning 105
4.5 LEON OS corrupts variable shared with SHAVEs: Results for 2D Convolution . . 106
4.6 LEON OS corrupts variable shared with SHAVEs: Results for 2D Binning 106
4.7 SHAVEs corrupt their own data: Results of 2D Convolution 107
4.8 SHAVEs corrupt their own data: Results of 2D Convolution 107
4.9 Leon OS corrects SHAVE instruction memory harm, results for 2D Convolution . . 108
4.10 Leon OS corrects SHAVE instruction memory harm, results for 2D Binning 108
4.11 LEON OS corrects corrupted data transferred, results for 2D Convolution 110
4.12 LEON OS corrects corrupted data transferred, results for 2D Binning 110
4.13 Leon OS corrects variables shared with SHAVEs, results for 2D Convolution 112
4.14 Leon OS corrects variables shared with SHAVEs, results for 2D Binning 112
4.15 SHAVEs correct variables shared with LEON OS, results for 2D Convolution . . . 114
4.16 SHAVEs correct variables shared with LEON OS, results for 2D Binning 114
4.17 LEON OS corrects data destroyed by SHAVEs, results for 2D Convolution 116
4.18 LEON OS corrects data destroyed by SHAVEs, results for 2D Binning 116
4.19 SHAVEs correct data destroyed by SHAVEs, results for 2D Convolution 118
4.20 SHAVEs correct data destroyed by SHAVEs, results for 2D Binning 118
4.21 3 Voting system, results for 2D Convolution 120
4.22 3 Voting system, results for 2D Binning 120
4.23 5 Voting system, results of 2D Convolution 122
4.24 5 Voting system, results of 2D Binning 122

Εκτεταμένη Περίληψη

Εισαγωγή

Στην ανάπτυξη εφαρμογών συχνά, ακολουθείται η παραδοχή ότι δεν υπάρχει η πιθανότητα να υπάρξει κάποιο λάθος είτε σε επίπεδο software είτε σε επίπεδο hardware. Ωστόσο, η παραδοχή αυτή συχνά καταρρίπτεται και κατά άμεση συνέπεια οδηγεί σε καταστροφικά αποτελέσματα. Υπό τις συνθήκες αυτές δημιουργείται μια νέα ανάγκη διασφάλισης υψηλής διαθεσιμότητας από τα υπολογιστικά μας συστήματα. Συνεπώς, προκύπτει ένας νέος τρόπος σχεδίασης υπολογιστικών συστημάτων με όνομα Fault-Tolerant Computing[2].

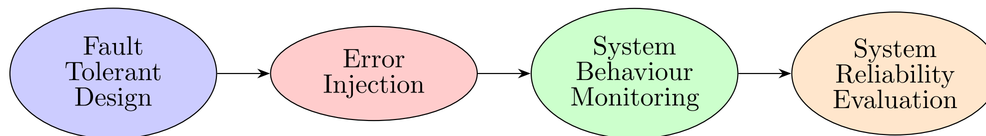
Ένας από τους πλέον διαδεδομένους κλάδους υπολογιστικών συστημάτων είναι τα ενσωματωμένα συστήματα. Τα συστήματα αυτά σε αντίθεση με τα υπολογιστικά συστήματα γενικού σκοπού χαρακτηρίζονται από διάφορους περιορισμούς. Αρχικά, διατελούν μία μόνο εξειδικευμένη λειτουργία αενάως. Επιπλέον, έχουν περιορισμούς ως προς το κόστος, το μέγεθος, την ενέργεια που καταναλώνουν αλλά και απόδοση τους. Επομένως, τα συστήματα αυτά πρέπει να χωράνε σε ένα μόνο chip και να μπορούν να επεξεργαστούν δεδομένα σε πραγματικό χρόνο με όσο το δυνατόν μικρότερη ενεργειακή κατανάλωση για να διατηρήσουν τη διάρκεια ζωής της μπαταρίας τους. Επιπλέον, πρέπει να αντιδρούν σε πραγματικό χρόνο. Δηλαδή, πρέπει να μπορούν να αντιδρούν στις αλλαγές του περιβάλλοντος με ακρίβεια ελαχίστων δευτερολέπτων και να υπολογίζουν διάφορα αποτελέσματα σε πραγματικό χρόνο χωρίς χρονική καθυστέρηση. Ας θεωρήσουμε για παράδειγμα τον χειριστή πορείας ενός αυτοκινήτου. Διαρκώς ελέγχει και αντιδρά στους σένσορες της ταχύτητας και των φρένων. Πρέπει επιπλέον, να υπολογίσει την επιτάχυνση και την επιβράδυνση σε πραγματικό χρόνο καθώς ένας αργοπορημένος έλεγχος μπορεί να οδηγήσει στην απώλεια του ελέγχου του αυτοκινήτου. Πέραν των αυτοκινήτων, τα ενσωματομένα συστήματα χρησιμοποιούνται σε διαστημικές εφαρμογές. Ωστόσο, στις διαστημικές εφαρμογές δεν χρειαζέται μόνο reliable υπολογισμούς αλλά και τα υποσυστήματα του συστήματος να είναι και rad-hard κάτι που καθυστά την πλατφόρμα της παρούσας διπλωματικής κατάλληλη για τέτοιες εφαρμογές.

Με βάση τα προηγούμενα συμπεραίνουμε ότι, τα ενσωματωμένα συστήματα απαντώνται σε safety-critical συστήματα [3] όπου λανθασμένη λειτουργία του software ή του hardware θα μπορούσε να οδηγήσει σε καταστροφικές συνέπειες. Συνεπώς, η ανάγκη για υψηλή διαθεσιμότητα αλλά και εμπιστοσύνη στην ορθή λειτουργία των ενσωματωμένων συστημάτων είναι ακρογωνιαία. Επομένως, και οι εφαρμογές οι οποίες σχεδιάζονται για τα συστήματα αυτά οφείλουν να βασίζονται στην Fault-Tolerant παραδοχή που αναφέραμε προηγουμένως.

Ωστόσο, ένα ερώτημα εγείρεται από τα παραπάνω, πως μπορεί μια σχεδιαστική ομάδα να ελέγξει εάν το Fault-Tolerant design της θα λειτουργήσει σύμφωνα με τις καθορισμένες προδιαγραφές σε περίπτωση σφάλματος Δηλαδή, πως μπορεί πρακτικά να ελεγχθεί η διαθεσιμότητα του συστήματος Για να απαντηθούν τα παραπάνω ερωτήματα μια νέα μέθοδος testing έχει αναπτυχθεί με όνομα Fault Injection[4]. Σύμφωνα με τη μέθοδο αυτή, οι σχεδιαστές εισάγουν εσκεμμένα σφάλμα στο σύστη-

μα τους προκειμένου να μπορέσουν να ελέγξουν τη συμπεριφορά του υπό εσφαλμένες συνθήκες. Η εισαγωγή σφάλματος μπορεί να γίνει είτε σε επίπεδο υλικού(π.χ εσκεμμένη έκθεση συστήματος σε ηλεκτρομαγνητική ακτινοβολία προκείμενου να πραγματοποιηθεί αλλαγή των περιεχομένων των καταχωρητών του[5]) είτε σε επίπεδο λογισμικού(π.χ εισαγωγή τυχαίων δεδομένων στο σύστημα[6]).

Παρακάτω βλέπουμε μια οπτική αναπαράστασή της διαδικασίας αξιολόγησης ενός Fault-Tolerant συστήματος:



Σχήμα 1: Αξιολόγηση Fault-Tolerant συστήματος.

Σύμφωνα με τις παραπάνω ανάγκες δημιουργήθηκε και η παρούσα διπλωματική η οποία αξιολογεί τη συμπεριφορά διαφόρων τεχνικών Fault-Tolerance σε εξαιρετικά ετερογενείς αρχιτεκτονικές όπως οι VPUs. Επιπλέον, στα πλαίσιά της προσπάθειας αυτής υλοποιήθηκαν διάφορες τεχνικές Fault-Injection προκειμένου να αξιολογηθούν η τελική διαθεσιμότητα του συστήματος.

Υπόβαθρο

Vision Processing Units

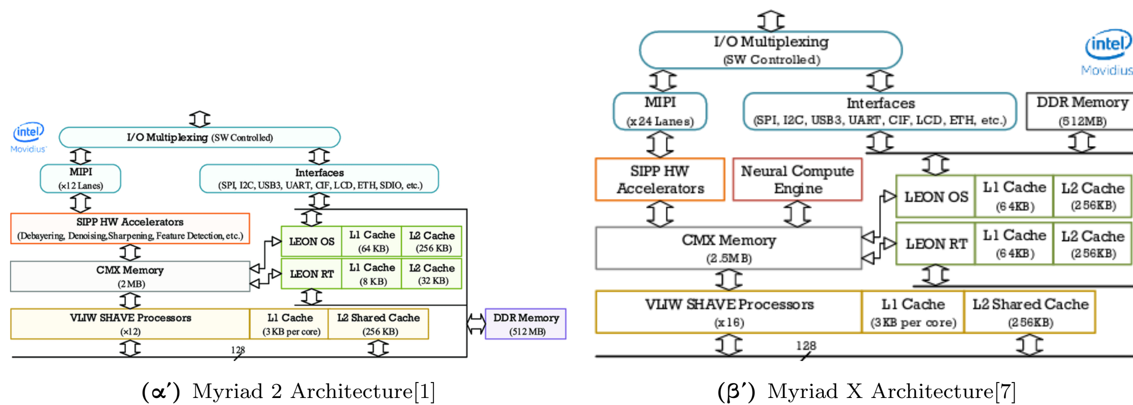
Τα Vision Processing Units αποτελούν μια καινούργια κατηγορία μικροεπεξεργαστών στόχος των οποίων είναι η παροχή υψηλής απόδοσης με πολύ μικρή κατανάλωση ισχύος. Επιπλέον οι μικροεπεξεργαστές αυτοί είναι κατάλληλα βελτιστοποιημένοι για να τρέχουν προ-εκπαιδευμένα μοντέλα συνελικτικών νευρωνικών δικτύων. Συνεπώς, είναι ιδανικοί για να τρέχουν Ai at the Edge εφαρμογές.

Σε σχέση με άλλους hardware accelerators όπως τα F.P.G.A και τα G.P.U έχουν χειρότερη επίδοση. Ωστόσο, το γεγονός που τα καθιστά ιδανικά για τα ενσωματωμένα συστήματα είναι η πόλη υψηλή απόδοση ανά καταναλωμένου Watt. Επιπλέον, τα V.P.Us χαρακτηρίζονται από μεγάλη προγραμματιστική δυσκολία. Συνεπώς, είναι μια πλατφόρμα με διάφορα tradeoffs που ήρθε, ενδεχομένως, για να επαναπροσδιορίσει το Edge Computing.

Intel Movidius Myriad Family of V.P.Us

Η ποιο διαδεδομένη σειρά V.P.U είναι η Myriad που αναπτύχθηκε από τη Movidius. Η Movidius αποτελεί μια θυγατρική της Intel που εξαγοράστηκε το 2016 έναντι 400 εκατομμυρίων δολαρίων[;].

Τα μοντέλα της οικογένειας Myriad είναι η Myriad 2 και η Myriad X. Παρακάτω βλέπουμε την αρχιτεκτονική των προαναφερθέντων μικροεπεξεργαστών: Όπως γίνεται εμφανές από τα παραπάνω σχήματα τόσο η Myriad 2 όσο και η Myriad X έχουν δύο 32 bit LEON4 επεξεργαστές γενικού σκοπού RISC SPARCv8 αρχιτεκτονικής (LEON OS, LEON RT). Ο LEON OS υποστηρίζει ένα Real Time Operating System με όνομα RTEMS ενώ ο LEON RT είναι υπεύθυνος για τη διαχείριση των περιφερειακών. Υπεύθυνοι για την υψηλή ρυθμαπόδοση του συστήματος είναι οι Streaming Hybrid Architecture Vector Engines (SHAVEs) πυρήνες αρχιτεκτονικής 128 bit, VLIW και SIMD. Την ενορχήστρωση της λειτουργίας των SHAVEs αναλαμβάνουν οι προαναφερθέντες επεξεργαστές γενικού σκοπού. Τέλος, τόσο η Myriad 2 όσο και η Myriad X έχουν μια σειρά από hardware φίλτρα που ονομάζονται Streaming Image Processing Pipeline (SIPP).



(α′) Myriad 2 Architecture[1] (β′) Myriad X Architecture[7]

Σχήμα 2: Αρχιτεκτονικές επεξεργαστών της οικογενείας Myriad

Η οργάνωση της μνήμης των δύο συστημάτων είναι παρόμοια καθώς και τα δύο διαθέτουν τόσο DDR αλλά και DRAM. Επιπλέον, ειδική μνεία πρέπει να δοθεί στην Connection Matrix Memory (CMX) η οποία χρησιμοποιείται ως NUMA ScratchPad μνήμη και συνεισφέρει σημαντικά στην υψηλή απόδοση του συστήματος. Η μεταφορά των δεδομένων από τη DDR στη CMX, και το αντίστροφο, μέσω ενός Direct Memory Access engine. Επιπλέον, ο LEON OS καθώς και ο LEON RT διαθέτουν L1 cache για δεδομένα αλλά και για εντολές καθώς και L2 cache. Τέλος, οι SHAVEs διαθέτουν μια ξεχωριστή L1 cache ο καθένας και μια κοινή L2 cache.

Οι βασικές διαφορές των δύο μοντέλων είναι το γεγονός ότι η Myriad X διαθέτει έναν εξειδικευμένο επεξεργαστή για την εκτέλεση Convolutional Neural Networks το Neural Compute Engine. Επιπλέον, η τελευταία γενιά της Myriad διαθέτει 16 SHAVEs σε αντίθεση με την προγενέστερη που διαθέτει μόλις 12. Επιπρόσθετα, το μέγεθος τόσο της CMX καθώς και οι L1, L2 caches του LEON RT έχουν αυξηθεί. Τέλος, η Myriad X σύμφωνα με τη Movidius δύναται να φτάσει επιδόσεις έως 1 τρισεκατομμύριο εντολές το δευτερόλεπτο[8].

Fault Tolerance

Η ανοχή στο σφάλμα (Fault Tolerance) ουσιαστικά σημαίνει τη δυνατότητά ενός συστήματος να συνεχίσει τη λειτουργία του παρόλο που ένα η περισσότερα από τα υποσυστήματα του σταματήσει να λειτουργεί. Δηλαδή, η ανοχή στο σφάλμα είναι πως ένα σύστημα ανταποκρίνεται σε δυσλειτουργίες του hardware η του software.

Fault Tolerant Computing

Ο στόχος ενός Fault Tolerant συστήματος είναι να διασφαλίσει τη συνεχόμενη λειτουργία και υψηλή διαθεσιμότητα του συστήματος αποτρέποντας διακοπές που θα προέκυπταν από κάποια αποτυχία.

Το Fault Tolerant Computing μπορεί να έχει διάφορα επίπεδα ανοχής στο σφάλμα:

- Στο χαμηλότερο επίπεδο το σύστημα μπορεί να απαντάει σε διακοπές ρεύματος με εναλλακτικές πηγές τροφοδοσίας.
- Σε ένα υψηλότερο επίπεδο τη δυνατότητα χρήσης ενός εφεδρικού συστήματος με πολλή μικρή χρονική καθυστέρηση.
- Σε περίπτωση που σημειωθεί σφάλμα σε κάποιο δίσκο να υπάρχει κάποιος εφεδρικός δίσκος που περιέχει όλα τα δεδομένα και να μπορεί να αντικαταστήσει τον βασικό δίσκο κατευθείαν. Με

τον τρόπο αυτόν η λειτουργία του συστήματος συνεχίζει παρά τη μερική βλάβη αντί να υπάρχει άμεση καταστροφή λόγο της απώλειας λειτουργίας.

- Υψηλού επιπέδου Fault Tolerant συστήματα αποτελούνται από πολλές επεξεργαστικές μονάδες οι οποίες ανιχνεύουν το σφάλμα και προσπαθούν αμέσως να το διορθώσουν.

Συνεπώς, τα Fault-Tolerant συστήματα διασφαλίζουν ότι δε θα υπάρξει διακοπή στη λειτουργία τους έχοντας εφεδρικά υποσυστήματα που θα λάβουν τη θέση των αρχικών σε περίπτωση σφάλματος. Ορισμένα από αυτά μπορεί να είναι:

- Hardware συστήματα με ίδια η εφεδρικά λειτουργικά συστήματα. Συνεπώς, για να υπάρχει πραγματικά αδιάλυπτη λειτουργία του συστήματος σε ένα τέτοιο σύστημα θα πρέπει στο εφεδρικό σύστημα να υπάρχουν όλες οι πληροφορίες του αρχικού συστήματος, π.χ. το scheduling των διαφόρων εφαρμογών, να είναι `1 προς 1` αντίγραφο των αντίστοιχων δεδομένων του αρχικού συστήματος.
- Software συστήματα που διαθέτουν άλλα Software συστήματα σε προηγούμενη κατάσταση ως εφεδρικά.
- Εφεδρικά τροφοδοτικά που μπορούν να βοηθήσουν στην αποφυγή δυσλειτουργίας του συστήματος σε περίπτωση που υπάρξει πρόβλημα στο τροφοδοτικό. Με τον τρόπο αυτό διασφαλίζεται ότι δε θα υπάρξει πτώση του συστήματος.

Fault Injection

Το Fault Injection είναι μια ιδιαίτερα διαδεδομένη τεχνική testing. Μέσω της συγκεκριμένης τεχνικής εισάγεται σφάλμα ή πίεση σε ένα σύστημα προκειμένου να παρατηρηθεί η συμπεριφορά του συστήματος υπό αυτές τις συνθήκες.

Σύντομη ιστορική αναδρομή του Fault Injection

Η εισαγωγή σφάλματος ξεκίνησε ως μια τεχνική προσομοίωσής σφάλματος σε επίπεδο hardware. Οι μηχανικοί εξέθεταν τις συσκευές σε διάφορες επιβλαβές συνθήκες και παρατηρούσαν πόσο καλά θα συνέχιζαν τη λειτουργία τους. Μερικά από τα test που έκαναν ήταν να βραχυκυκλώνουν pin των συσκευών, να εισάγουν ηλεκτρομαγνητικές παρεμβολές, να διαταράσσουν την τροφοδοσία του συστήματος και ακόμη να βομβαρδίζουν τα κυκλώματα του συστήματος με ακτινοβολία. Ο στόχος ήταν να παρατηρήσουν πως οι παραπάνω συνθήκες επηρέαζαν τη λειτουργία του συστήματος και αν η συσκευή σταματούσε τη λειτουργία της να την επανασχεδιάσουν.

Με την πάροδο του χρόνου οι μηχανικοί σχεδίασαν εργαλεία που επέτρεπαν την εισαγωγή σφάλματος με διαφορετικές μεθόδους. Συνεπώς, οι συσκευές άρχισαν να διαθέτουν διάφορα debugging ports όπως το JTAG που επέτρεπε την εισαγωγή ελεγχόμενου σφάλματος κατευθείαν στα κυκλώματα του συστήματος. Έπειτα, σχεδιαστήκαν μέθοδοι για την εισαγωγή σφάλματος σε επίπεδο λογισμικού προκειμένου να προσομοιώσουν σφάλμα στις εφαρμογές τους και να ελέγξουν τις διάφορες συναρτήσεις που διαχειρίζονται τα σφάλματα και τις εξαιρέσεις του προγράμματος. Προκειμένου να επιτευχθούν τα παραπάνω οι μηχανικοί είτε άλλαζαν τον πηγαίο κώδικα για να εισάγουν προσομοιωμένα λάθη(ςομπιλετιμε ερρορ) και να πυροδοτήσουν λάθη σε συστήματα που ήδη τρέχουν (runtime error).

Η χρησιμότητα του Fault Injection

Η εισαγωγή σφάλματος είναι μια από της ποιο βασικές μορφές testing καθώς βοηθάει την ομάδα μηχανικών στο στάδιο σχεδίασης να κατανοήσουν τη συμπεριφορά του συστήματος τους υπό συνθήκες

πίεσης ή ακόμη και καθολικής δυσλειτουργίας. Επιπλέον, η προσομοίωση της συμπεριφοράς αυτής επιτρέπει στους μηχανικούς να μπορούν να διασφαλίσουν ένα συγκεκριμένο επίπεδο quality of service στους καταναλωτές. Τα θετικά χαρακτηριστικά της εισαγωγής σφάλματος επομένως είναι ποικίλα. Τα ποιο βασικά από αυτά παρατίθενται αναλυτικά παρακάτω:

- **Οι μηχανικοί μπορούν να ελέγξουν εκτενώς τις εφαρμογές τους και τα συστήματα τους**. Παραδοσιακά, το software testing εστιάζει στο happy path testing[9]. Δηλαδή, ελέγχει τα μονοπάτια των συστημάτων που προσδοκούν οι μηχανικοί ότι το θα λάβουν. Αυτό όμως δεν ελέγχει τους τρόπους με τους οποίους τα συστήματα μπορούν να αποκλίνουν από τις προσδοκίες της σχεδιαστικής ομάδας λόγο μην προσδοκώμενης συμπεριφοράς, αλλαγής των συνθηκών λειτουργίας, λαθών στις εξαρτήσεις η οποιαδήποτε άλλη συνθήκη. Συνεπώς, θέλουμε να βεβαιωθούμε ότι οι ανθεκτικοί μηχανισμοί που έχουν τοποθετηθεί στα συστήματά θα δουλέψουν όπως είναι αναμενόμενο και σε αυτό ακριβώς βοηθάει η εισαγωγή σφάλματος.
- **Δίνει τη δυνατότητά συστηματικού εντοπισμού της φύσης και της αιτίας διαφόρων σφαλμάτων**. Όταν ένα σύστημα αποτυγχάνει, η τεχνική ομάδα μπαίνει σε λειτουργία αντίδρασης. Ο βασικός της στόχος είναι να σταματήσει το πρόβλημα και μετά από αυτό να επανέλθει η κανονική λειτουργία του συστήματος όσο το δυνατών συντομότερα. Ανάλογα με τη σοβαρότητα της κατάστασης μπορεί να πάρει ημέρες ή ακόμη και εβδομάδες μέχρι να υπάρχει κάποια ξεκάθαρη απάντηση. Η εισαγωγή σφάλματος δίνει επομένως στου μηχανικούς απόλυτο έλεγχο στο πόσο σφάλμα αλλά και πότε το σφάλμα αυτό εισάγεται στο σύστημα. Τέλος, δίνει στους μηχανικούς τη δυνατότητα να αναπαράγουν τα διάφορα προβλήματα που δημιουργούνται επιτρέποντας έτσι την ουσιαστική επιδιόρθωση τους.
- **Επιτρέπει στους μηχανικούς να προετοιμαστούν για το αναπάντεχο**. Τα πράγματα που μπορούν να πάνε λάθος στα πλαίσια της παραγωγής είναι πάρα πολλά. Επιπλέον, ακόμη και μικρά λάθη μπορούν να προκαλέσουν μεγάλες απώλειες. Συνεπώς, η εισαγωγή σφάλματος επιτρέπει να γίνει έλεγχος στη συμπεριφορά τους συστήματος σε συνθήκες που κανονικά δε θα ήταν αναμενόμενες όπως σφάλματα στη μνήμη, απότομες αυξήσεις στη χρήση της κεντρικής επεξεργαστικής μονάδας κ.ο.κ. Η τεχνική αυτή επιτρέπει επομένως στη σχεδιαστική ομάδα να προετοιμαστεί για το αναπάντεχο προσθέτοντας μηχανισμούς που θα συντελέσουν στην αποδοτική αντιμετώπιση του προβλήματος που προέκυψε.

Fault Injection μέθοδοι και Fault Tolerance τεχνικές

Η συγκεκριμένη διπλωματική κινείται σε δύο βασικούς άξονες. Αρχικά, ο πρώτος βασικός άξονας της είναι η ανάπτυξη πολιτικών Fault Tolerance προκειμένου η πλατφόρμα Myriad 2 να μπορέσει να συνεχίσει την αδιάλυπτη λειτουργία της σε περίπτωσή σφάλματος. Ο δεύτερος βασικός άξονας είναι ο σχεδιασμός διάφορων τεχνικών Fault Injection [10]προκειμένου να μπορέσουμε να επιβεβαιώσουμε την ορθή λειτουργία των Fault-Tolerant πολιτικών που αναπτύξαμε. Οι παραπάνω πολιτικές και μέθοδοι επεξηγούνται αναλυτικότερα στη συνέχεια.

Fault Tolerance πολιτικές

Όπως αναλύσαμε στις προηγούμενες υποενότητες είναι απαραίτητο τα ενσωματωμένα συστήματα να λειτουργούν δίχως διακοπές καθώς συνήθως τοποθετούνται σε time-critical συστήματα. Στα πλαίσια της συγκεκριμένης διπλωματικής υλοποιήθηκαν και αξιολογήθηκε η απόδοση διάφορων πολιτικών Fault Tolerance που θα αναλυθούν παρακάτω.

Fault Detection

Βασικό στάδιο για την αντιμετώπιση και επιδιόρθωση ενός σφάλματος είναι η ανίχνευση του σφάλματος στο σύστημα. Η ανίχνευση αυτή πρέπει να συμβαίνει με απόλυτη ακρίβεια καθώς είναι το ποίο βασικό όπλο για τον σχεδιασμό ενός Fault Tolerant συστήματος. Στην παρούσα εργασία χρησιμοποιήθηκε μια αρκετά διαδεδομένη τεχνική για την ανίχνευση του σφάλματος που ονομάζεται Cyclic Reduncancy Check[11].

Ανίχνευση σφάλματος με Cyclic Reduncancy Check

Το Cyclic Redundancy Check (CRC) είναι ένας κώδικας ανίχνευσης σφάλματος που χρησιμοποιείται συχνά σε ψηφιακά δίκτυα και σε συσκευές αποθήκευσης προκειμένου να ανιχνεύονται τυχαίες αλλαγές στα δεδομένα. Τα μπλοκ των δεδομένων που μπαίνουν στα συστήματα έχουν μια τιμή ελέγχου που βασίζεται στη διαίρεση πολυωνύμων των περιεχομένων τους. Κατά την ανάκτηση ο υπολογισμός επαναλαμβάνεται και αν η τιμή ελέγχουν έχει αλλάξει τότε υπάρχει κάποια αλλαγή στα αποθηκευμένα δεδομένα. Ο λόγος που τα CRC είναι τόσο διαδεδομένα είναι το γεγονός ότι είναι εύκολο να υλοποιηθούν σε διάδικο υλικό, εύκολα να αναλυθούν μαθηματικά και εύκολα να ανιχνεύουν συνήθη λάθη που οφείλονται σε θόρυβο. Καθώς το αποτέλεσμα τους είναι συγκεκριμένου μήκους συχνά τα CRC χρησιμοποιούνται ως συναρτήσεις κατακερματισμού[12].

Όπως αναφέραμε προηγουμένως ο βασικός λόγος της υψηλής απόδοσης της Myriad 2 οφείλεται στους γρήγορους VLIW Shave συνεπεξεργαστές. Η γενική μέθοδος προγραμματισμού της συγκεκριμένης πλατφόρμας βασίζεται στα παρακάτω βήματα:

1. Ο LEON OS λαμβάνει τα δεδομένα εισόδου.

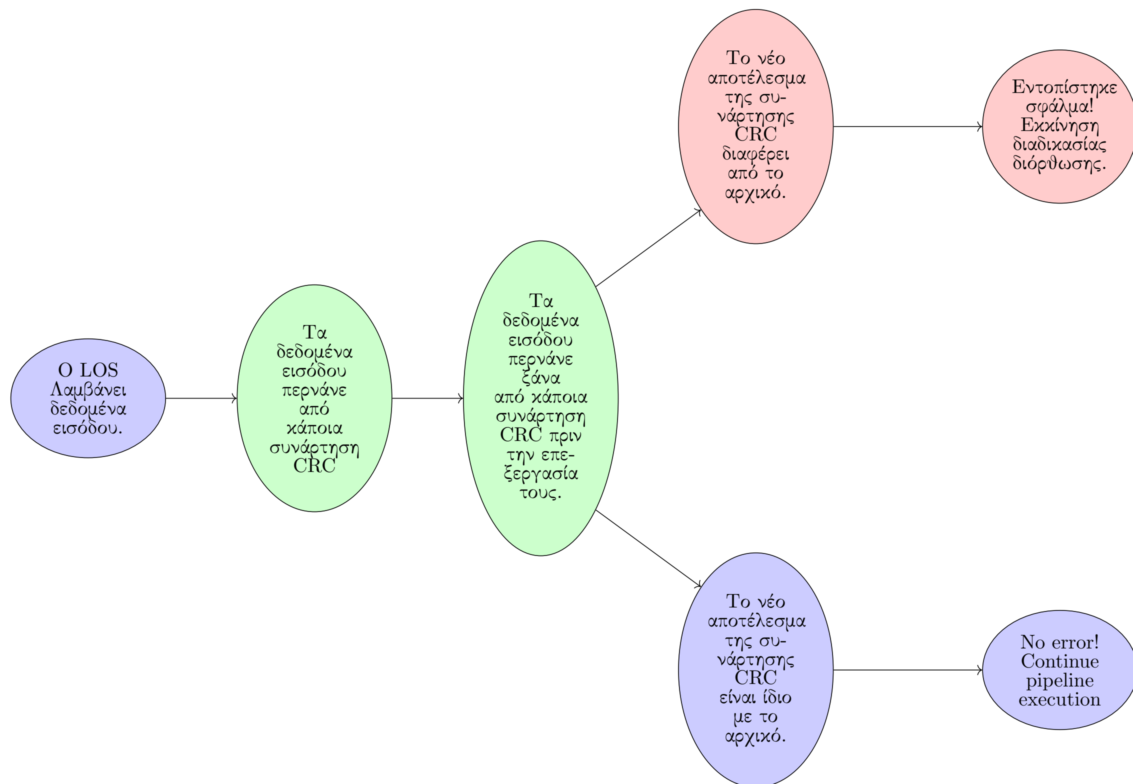
2. Μεταφορά δεδομένων από τον LOS στους Shaves.

3. Επεξεργασία των δεδομένων στους Shaves.

4. Επιστροφή δεδομένων πίσω στον LOS μετά την ολοκλήρωση της επεξεργασίας τους.

Τα παραπάνω βήματα επαναλαμβάνονται μέχρι να ολοκληρωθεί η επεξεργασία των δεδομένων εισόδου.

Συνεπώς, για να ανιχνεύονται μεταβολές στα δεδομένα οφείλουμε να περνάμε τα δεδομένα εισόδου από μια συνάρτηση CRC και σε τακτά χρονικά διαστήματα, η τουλάχιστον πριν την εξεργασία τους, να τα περνάμε ξανά για ελέγξουμε αν έχουν υποστεί κάποια αλλοίωση. Στη συνέχεια φαίνεται η διαδικασία εντοπισμού σφάλματος με τη χρήση του Cyclic Redundancy Check.

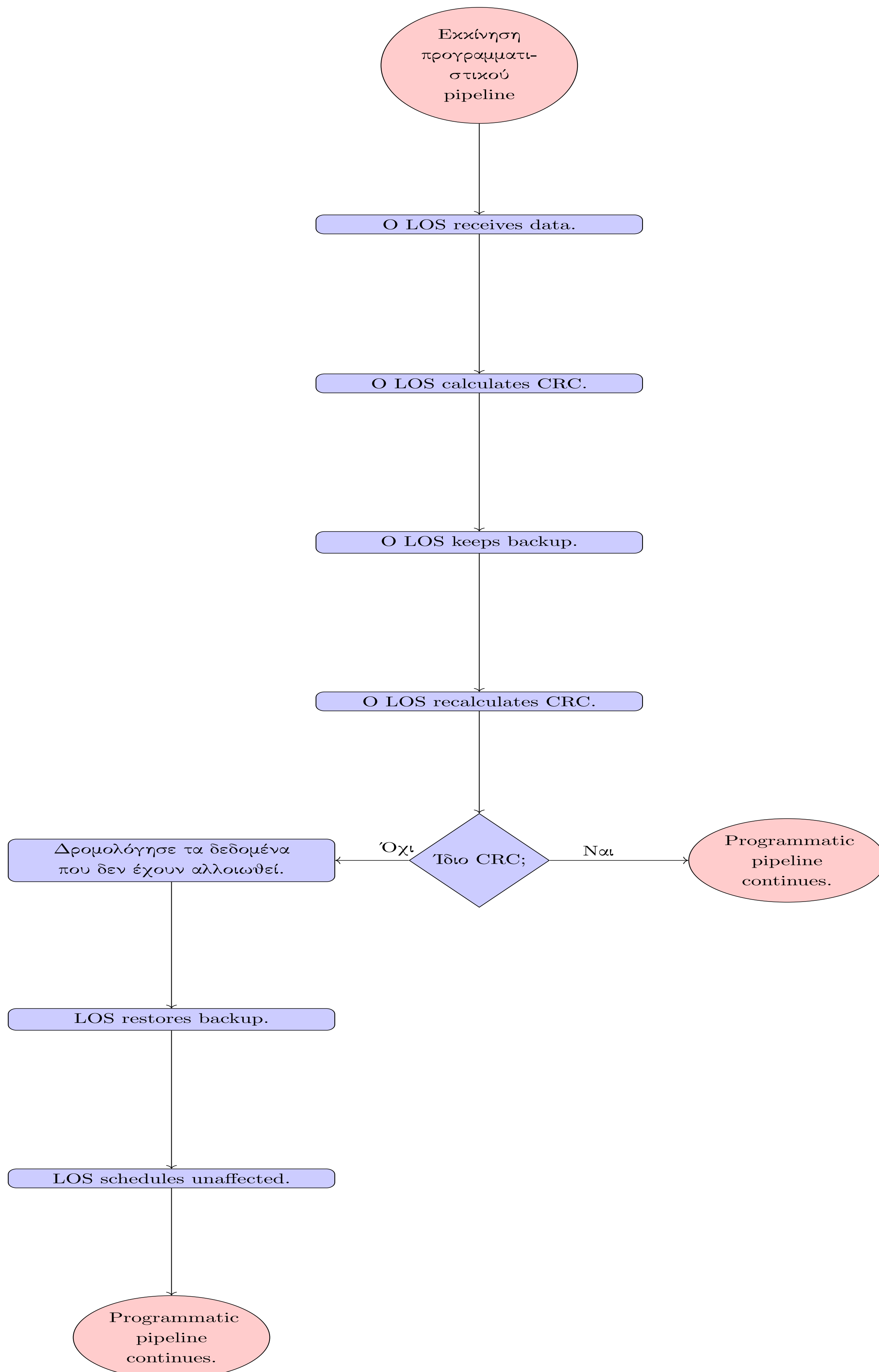


Σχήμα 3: Fault Detection: Using CRC to detect data corruption.

Επομένως, οι Fault Tolerant πολιτικές που θα περιγραφούν στη συνέχεια θα βασίζονται σε αυτή την τεχνική ανίχνευσης σφάλματος.

Fault Tolerant Πολιτική: Βλάβη στον LEONOS

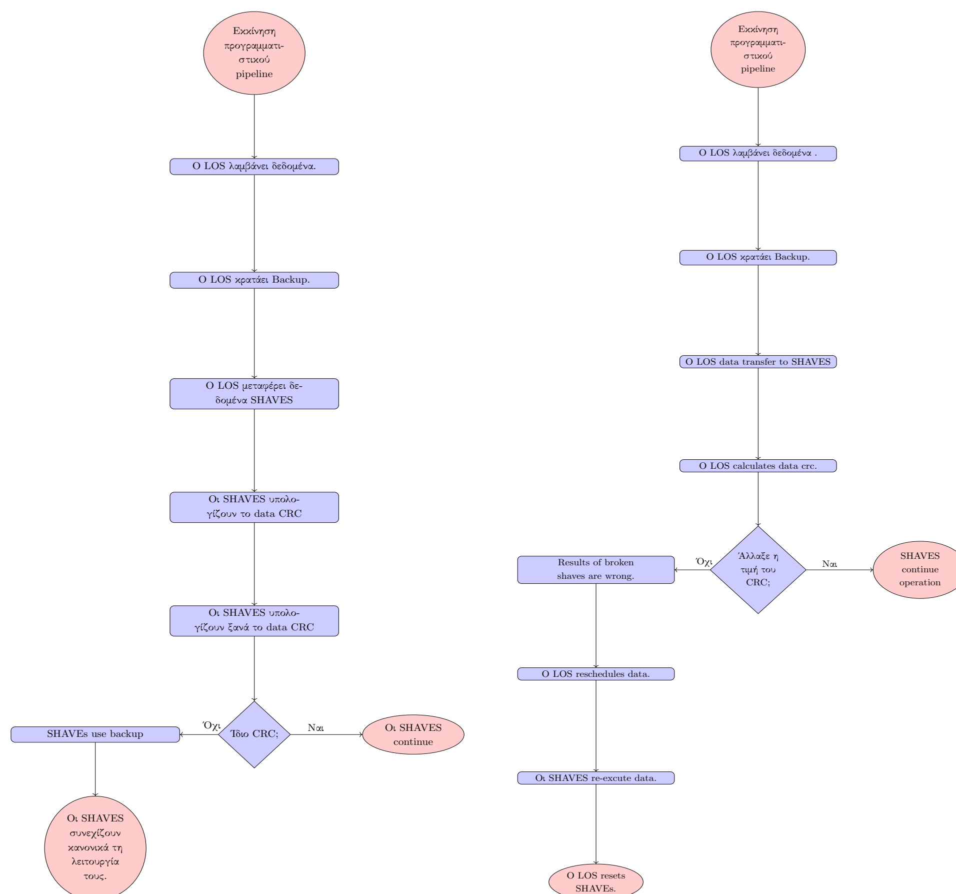
Σε προηγούμενη ενότητα αναφέραμε ότι ο LEONOS είναι ο βασικός ενορχηστρωτής της λειτουργίας της Myriad καθώς είναι υπεύθυνος για τη μεταφορά δεδομένων στους Shaves στα διάφορα στάδια του προγραμματιστικού pipeline. Συνεπώς, βγάζει ιδιαίτερο νόημα να πραγματοποιεί διαφόρους ελέγχους στα δεδομένα καθώς τα μεταφέρει στους διάφορους συνεπεξεργαστές. Στη συγκεκριμένη πολιτική ο LOS περνάει από μια συνάρτηση CRC τα δεδομένα εισόδου μόλις τα παραλάβει από κάποιο περιφερειακό. Επίσης, αφού τα περάσει τα δεδομένα αυτά από τη CRC συνάρτηση δημιουργεί ένα προσωρινό backup τους. Στη συνέχεια, προτού τα στείλει στους Shaves τα περνάει ξανά από μια CRC συνάρτηση και ελέγχει αν η τιμή επιστροφής της συνάρτησης θα είναι διαφορετική. Σε περίπτωση που η τιμή ταυτίζεται με την προηγούμενη τότε μεταβιβάζει τα δεδομένα στους Shaves κανονικά. Εάν, η τιμή αυτή είναι διαφορετική τότε, ο LOS έπαθε κάποια βλάβη και δεν μπορούμε να εμπιστευτούμε τα δεδομένα που προσπαθεί να μεταφέρει στους SHAVES. Επομένως, δρομολογεί τα δεδομένα που γνωρίζει ότι είναι σωστά στους Shaves και στη συνέχεια επαναφέρει τα corrupted δεδομένα στην αρχική τους κατάσταση μέσω του backup που είχε δημιουργήσει μόλις τα παρέλαβε και δρομολογεί τα δεδομένα αυτά ξανά στους Shaves. Παρακάτω παρατίθεται ένα διάγραμμα ροής της παραπάνω διαδικασίας προκειμένου να γίνει πλήρως κατανοητή:



Σχήμα 4: Fault Tolerant Policy 1: Using CRC to detect data corruption LOS reschedules Shave data.

Fault Tolerant Πολιτική: Επαναφορά του συστήματος από βλάβη στους SHAVES

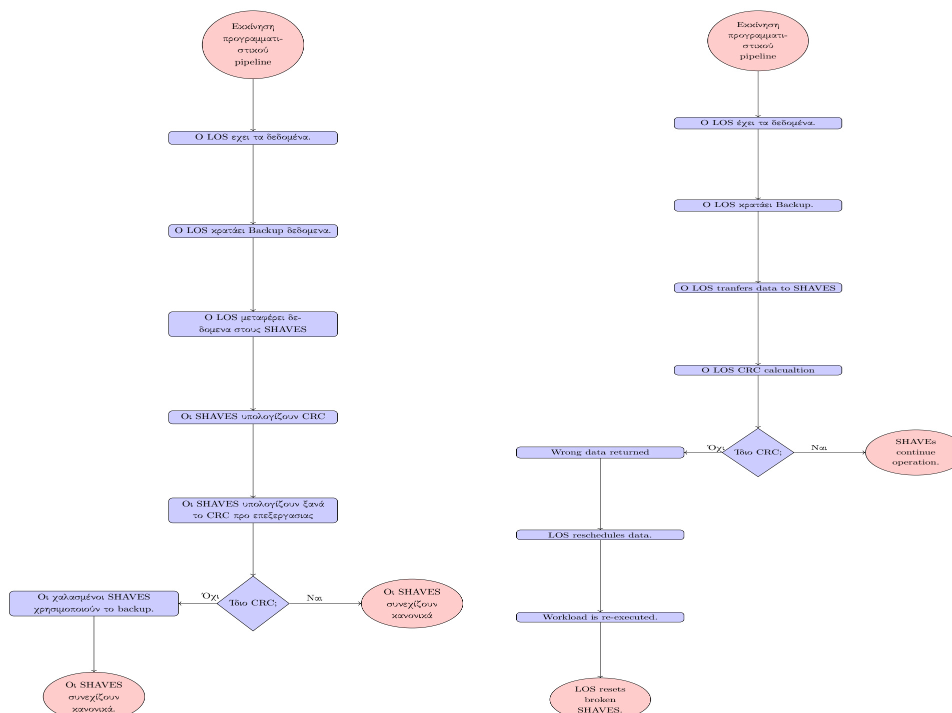
Στα πλαίσια της παρούσας πολιτικής οι Shaves αφού παραλάβουν τα δεδομένα τους από τον LOS τα περνάνε από μια CRC συνάρτηση. Στη συνέχεια αποθηκεύουν την τιμή που τους επέστρεψε η συνάρτηση αυτή προκείμενου μελλοντικά να μπορούν να κατανοήσουν αν τα δεδομένα έχουν αλλοιωθεί. Πριν την επεξεργασία των δεδομένων τα ξαναπερνάμε από τη συνάρτηση αυτή για να μην υπάρχει αμφιβολία για τον αν έχουν μεταβληθεί. Στην περίπτωση που εντοπισθεί κάποιο σφάλμα τότε το σύστημα μπαίνει σε λειτουργία ανάνηψης καθώς αυτό σημαίνει ότι υπάρχει κάποια βλάβη στους SHAVES. Ποιο συγκεκριμένα, διακρίνουμε δύο περιπτώσεις. Στην πρώτη περίπτωση ο κάθε χαλασμένος SHAVE αναλαμβάνει την αυτοΐαση του. Συνεπώς, επαναφέρει τα δεδομένα του από το διαθέσιμο backup που δημιούργησε ο LOS όταν παρέλαβε τα δεδομένα από το αντίστοιχο περιφερειακό. Εν συνεχεία, αφού επαναφέρει τα ορθά δεδομένα τρέχει τον κώδικα προκειμένου να υπολογιστούν τα ορθά δεδομένα. Η άλλη περίπτωση που διακρίνουμε είναι η εξής: O LOS ελέγχει αν τα δεδομένα έχουν αλλοιωθεί. Σε περίπτωση αλλοίωσης επαναφέρει τα δεδομένα τους στην αρχική τους, μη χαλασμένη, κατάσταση και τα δρομολογεί ξανά σε SHAVES που γνωρίζει ότι λειτουργούν, προκειμένου να βεβαιωθεί για την ορθή επεξεργασία των δεδομένων εισόδου. Έπειτα, κάνει reset τους χαλασμένους SHAVES προκειμένου να μπορεί να τους χρησιμοποιήσει μελλοντικά. Παρακάτω παρατίθενται δύο διαγράμματα ροής για κάθε μια από τις πολιτικές που αναφέραμε μόλις.



Σχήμα 5: Fault Tolerant Policy 2: Left: Shaves Self Heal, Right: LOS schedules data to working shaves.

Fault Tolerant Πολιτική: Ανάνηψη από σφάλμα σε κοινές μεταβλητές λόγο βλάβης κάποιου υποσυστήματος.

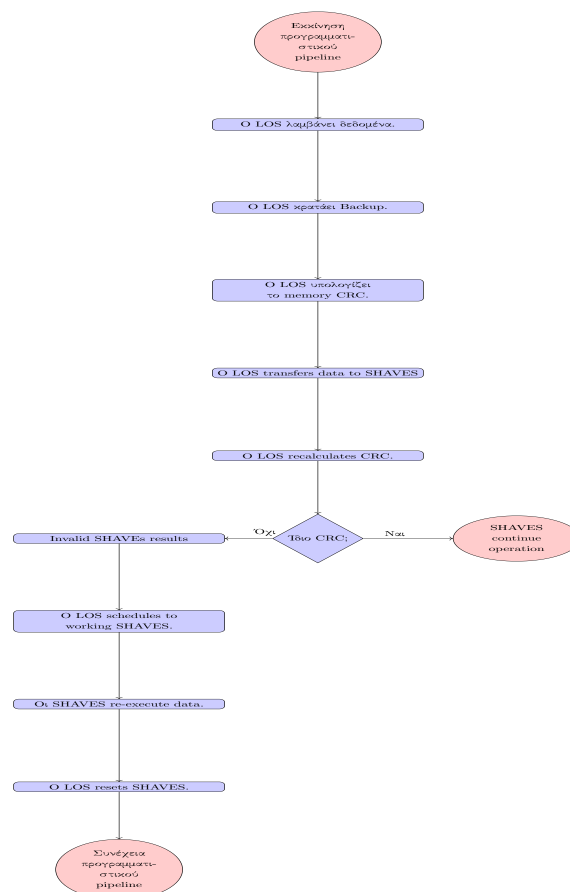
Στη συγκεκριμένη πολιτική το σύστημα δουλεύει ορθά, για κάποιο χρονικό διάστημα. Αλλά, οι SHAVES και ο LEONOS χρησιμοποιούν κοινές μεταβλητές. Μέτα από τη βλάβη όμως οι κοινές μεταβλητές των δύο προαναφερθέντων υποσυστημάτων αλλάζουν. Το συμβάν αυτό οδηγεί με τη σειρά του οδηγεί σε αλλοίωσή των δεδομένων που επεξεργάζεται το σύστημα και κατά συνέπεια, αν δε διορθωθεί, σε λανθασμένο αποτέλεσμα. Προκειμένου να αποτραπεί η λανθασμένη λειτουργία του συστήματος οι SHAVES περνάνε τα κοινά δεδομένα από μια συνάρτηση που υπολογίζει το CRC τους. Συνεπώς, πριν την επεξεργασία των δεδομένων αυτών ελέγχουν αν έχουν αλλοιωθεί και κατά συνέπεια διακρίνονται δύο περιπτώσεις. Η πρώτη περίπτωση είναι να μην υπάρχει καμία αλλοίωσή των δεδομένων και κατά συνέπεια δεν πρέπει να ληφθεί καμία δράση. Η δεύτερη, είναι να αλλάξει η τιμή του CRC και κατά συνέπεια οι SHAVES να αντιληφθούν ότι κάτι δεν πάει σωστά. Αφού, καταλάβουν ότι υπάρχει σφάλμα τότε εφαρμόζονται δύο πολιτικές επαναφοράς των δεδομένων. Στην πρώτη πολιτική οι ¨χαλασμένοι' SHAVES χρησιμοποιούν ένα τοπικό backup των δεδομένων που έχουν αποθηκεύσει. Με τον τρόπο αυτό διασφαλίζεται η ορθή λειτουργία. Στη δεύτερη πολιτική, αντί να αντιλαμβάνονται οι SHAVES το πρόβλημα στα δεδομένα, το αντιλαμβάνεται ο LEONOS. Ό τρόπος με τον οποίο εξάγει το συμπέρασμα αυτά είναι να περάσει τα δεδομένα που στέλνει στους SHAVES από μια CRC συνάρτηση. Με τον τρόπο αυτό καταλαβαίνει ότι τα κοινά δεδομένα έχουν αλλοιωθεί. Έπειτα δρομολογεί ξανά τα δεδομένα αυτά στους SHAVES που γνωρίζει ότι λειτουργούν ορθά προκειμένου να τα επεξεργαστούν ξανά και να διασφαλιστεί ένα ορθό αποτέλεσμα. Τα παρακάτω διαγράμματά ροής δείχνουν το τρόπο εκτέλεσης των παραπάνω πολιτικών.



Σχήμα 6: Fault Tolerant Policy 3: Left: Shaves Self Heal, Right: LOS schedules data to working shaves.

Fault Tolerant Πολιτική: Ανάνηψη από σφάλμα στη μνήμη που αποθηκεύεται ο κώδικας των SHAVES.

Ένα βασικό χαρακτηριστικό της Myriad 2 είναι το γεγονός ότι έχει ενιαίο Memory Space. Συνεπώς, μπορούν να γίνουν έλεγχοι για το αν έχει γίνει κάποια αλλοίωσή σε όλο το memory space. Επίσης, πρέπει να αναφέρουμε ότι το μέρος στη μνήμη στο οποίο θα αποθηκευτούν οι διάφορες εντολές των SHAVES ορίζεται χειροκίνητα σε ένα ςονφιγ αρχείο. Επομένως, δύναται να γίνουν αλλαγές στα περιεχόμενα όχι μόνο της μνήμης αλλά και στις εντολές που εκτελούνται από τους SHAVES. Στη συγκεκριμένη πολιτική, ο LEONOS παίρνει τις CRC τιμές όλου του memory-space που καταλαμβάνουν οι εντολές των SHAVES. Κατ΄ επέκταση ο LEONOS ελέγχει εάν περιεχόμενα της instruction-memory των SHAVES έχουν μεταβληθεί, μέσω του επαναλαμβανόμενου υπολογισμού του CRC αυτών των θέσεων μνήμης. Δεδομένου ότι ο κώδικας των SHAVES δεν αλλάζει στο run-time υπό κανονικές συνθήκες οποιαδήποτε μεταβολή αυτών των θέσεων μνήμης οφείλονται σε κάποια δυσλειτουργία του υλικού. Άρα σε μια τέτοια περίπτωση ο LEONOS αντιλαμβάνεται τη δεδομενική αλλοίωσή και θεωρεί invalid τα αποτελέσματα των SHAVES που έχει αλλοιωθεί η μνήμη που αποθηκεύονται οι εντολές τους. Έτσι η επόμενη δράση του είναι να δρομολογήσει τα input data που οδήγησαν σε λανθασμένα αποτελέσματα στους λειτουργικούς SHAVES. Τέλος, αφού η εκτέλεση του workload ολοκληρωθεί ο LEONOS κάνει reset τους ελαττωματικούς shaves. Ακολουθεί το σχετικό διάγραμμα ροής:



Σχήμα 7: Fault Tolerant Policy 4: LOS schedules data to working shaves.

N-voting Systems

Μια βασική προϋπόθεση για να διασφαλιστεί το system reliability ενός ενσωματωμένου συστήματος είναι η ύπαρξη πλεονασματικών πόρων. Ποιο συγκεκριμένα πρέπει πάντα να υπάρχει ένα εναλλακτικό μονοπάτι για ενδεχόμενο βλάβης. Το πλεόνασμα αυτό δύναται να βρίσκεται είτε σε μνήμη, π.χ εκχώρηση δεδομένων περισσότερες από μία φορές, είτε σε υπολογιστικές μονάδες, δηλαδή ύπαρξη περισσότερων της μιας υπολογιστικής μονάδας προκειμένου σε περίπτωση σφάλματος τα δεδομένα να μπορούν να δρομολογηθούν στην εφεδρική. Συνεπώς, στην περίπτωση της Myriad 2 που διαθέτει πόλους πυρήνες επεξεργασίας λόγο των SHAVES δύναται ορισμένοι από αυτούς να χρησιμοποιηθούν προκειμένου να εξασφαλίσουν την ανθεκτικότητα του συστήματος στα σφάλματα.

Η ποιο γνωστή τεχνική ελέγχου της ορθής λειτουργίας των διάφορων υπολογιστικών μονάδων που χρησιμοποιείται από τα ενσωματωμένα μέχρι το υπολογιστικό νέφος είναι τα N-voting συστήματα. Τα συστήματα αυτά διαμοιράζουν τους διαθέσιμους πόρους σε κάλπες (ballots) και κάθε υπολογιστική μονάδα αντιστοιχίζεται σε μια κάλπη. Οι υπολογιστικές μονάδες που έχουν αντιστοιχιστεί στην ίδια κάλπη έχουν τα ίδια δεδομένα εισόδου αλλά και περνάνε τα δεδομένα από την ίδια επεξεργασία. Εν συνεχεία, κάθε υπολογιστική μονάδα τοποθετεί το αποτέλεσμα των υπολογισμών της μέσα στην κάλπη. Αφού κάθε υπολογιστική μονάδα τελειώσει τα αποτελέσματα συγκρίνονται. Μετά τη σύγκριση σωστό θεωρείται το αποτέλεσμα που προέκυψε από την πλειοψηφία των SHAVES. Από τα παραπάνω καταλήγουμε στα παρακάτω συμπεράσματα:

- Το πλήθος των επεξεργαστικών μονάδων ανά κάλπη πρέπει να είναι περιττός αριθμός. Σε περίπτωση που ήταν άρτιος τότε ενδέχεται να προέκυπταν ισοψηφίες στα αποτελέσματα και να μην μπορούμε να εξάγουμε ποιο είναι το σωστό αποτέλεσμα.

- Το γεγονός ότι κάποιο αποτέλεσμα έχει ψηφιστεί από την πλειοψηφία ως σωστό δε σημαίνει ότι είναι και σωστό. Ποιο συγκεκριμένα ας θεωρήσουμε το σφάλμα ως μια συνάρτηση f που έχει ως σύνολο τιμών της συνάρτησης του σφάλματος είναι το σύνολο των δεδομένων που θα επηρεαστούν από τη συνάρτηση του σφάλματος. Το αποτέλεσμα της συνάρτησης αυτής είναι τα λανθασμένα δεδομένα εισόδου a. Συνεπώς, όλα τα προηγούμενα μπορούν να συνοψιστούν στην έκφραση $f(x) = a$. Συνεπώς, το μόνο που χρειάζεται προκειμένου δύο επεξεργαστικές μονάδες να βρουν το ίδιο αποτέλεσμα είναι η συνάρτηση f αλλά και τα δεδομένα x να είναι ίδια. Κατά συνέπεια, η πλειοψηφία ενδέχεται να καταλήξει σε λανθασμένα αποτελέσματα.

Στα πλαίσια της παρούσας εργασίας έχουν υλοποιηθεί δύο διαφορετικά συστήματα ν-ψηφοφόρων. Τα δύο συστήματα αυτά είναι:

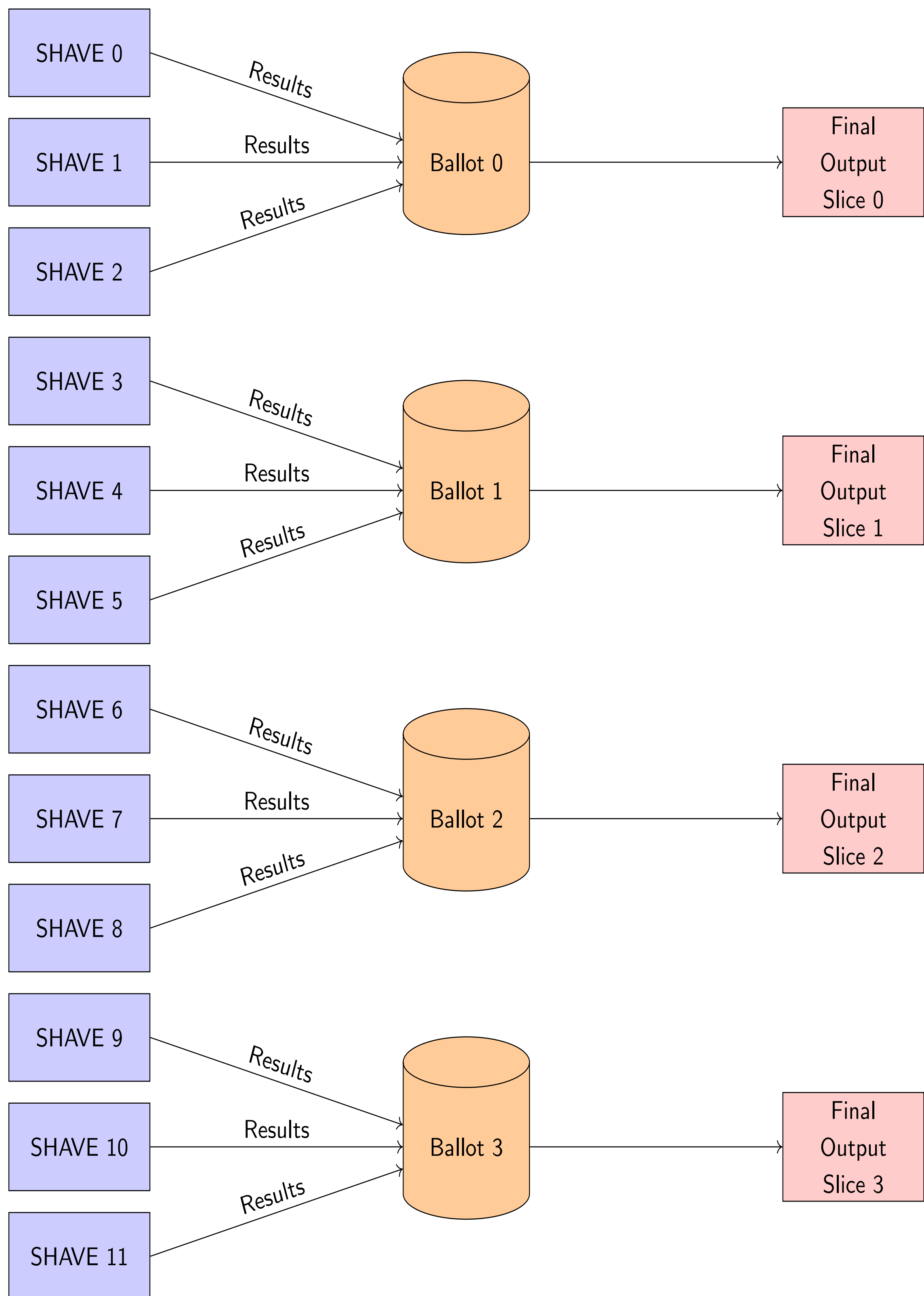
- **3-Voting System**

  Στο σύστημα αυτό οι SHAVES ανά τρεις αντιστοιχίζονται σε μια κάλπη. Στην περίπτωση αυτή κάθε ένας από τους SHAVES επεξεργάζονται με τον ίδιο τρόπο τα ίδια δεδομένα. Τέλος, τοποθετούν τα αποτελέσματα τους στην κάλπη που τους αντιστοιχίζεται και βρίσκουμε το αποτέλεσμα με το οποίο η πλειοψηφία συμφωνεί.

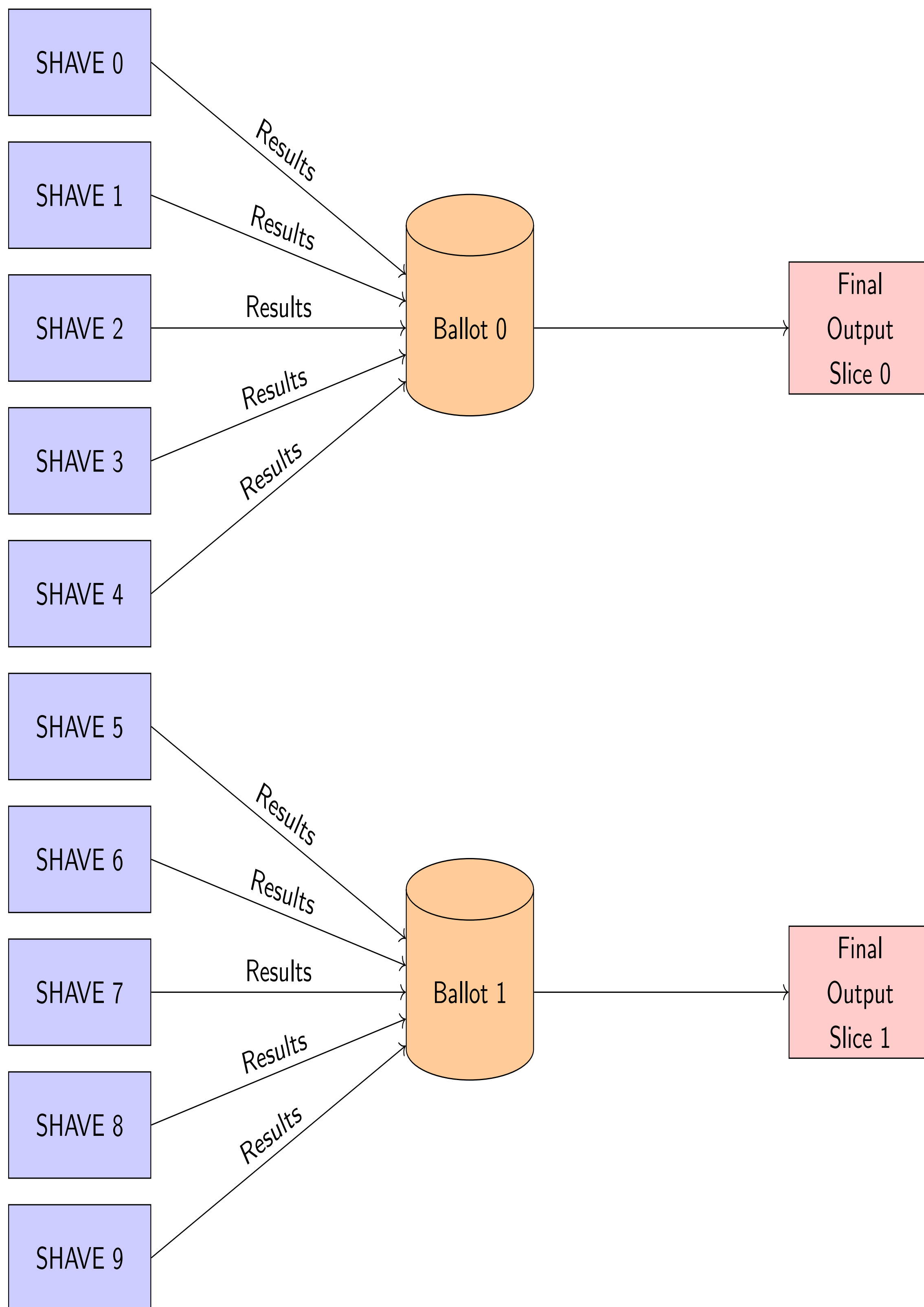
- **5-Voting Sytem**

  Στην περίπτωση αυτή έχουμε δύο διαφορετικές κάλπες και οι SHAVES αντιστοιχίζονται ανά πέντε σε μια από αυτές. Ο λόγος για τον οποίο οι SHAVES που αντιστοιχίζονται ανά κάλπη είναι πέντε είναι για να αποφευχθούν ισοψηφίες στις ψήφους.

Παρακάτω φαίνεται και η σχηματική αναπαράσταση των παραπάνω συστημάτων:



Σχήμα 8: N-voting Systems: 4 Voter System.



Σχήμα 9: N-voting Systems: 2 Voter System.

Μέθοδοι Fault Injection

Προκειμένου να μπορέσουμε να ελέγξουμε την απόδοση των προαναφερθέντων πολιτικών Fault Tolerance αναπτύξαμε διάφορες μεθόδους εσκεμμένης εισαγωγής σφάλματος.

Οι τεχνικές αυτές θα αναλυθούν στις επακόλουθες υποενότητες.

Μεταφορά καταστραμμένων δεδομένων από τον επεξεργαστή LEON OS

Παρακάτω επαναλαμβάνεται η γενική μέθοδος προγραμματισμού της συγκεκριμένης πλατφόρμας για διευκόλυνση του αναγνώστη:

1. Ο LEON OS λαμβάνει τα δεδομένα εισόδου.

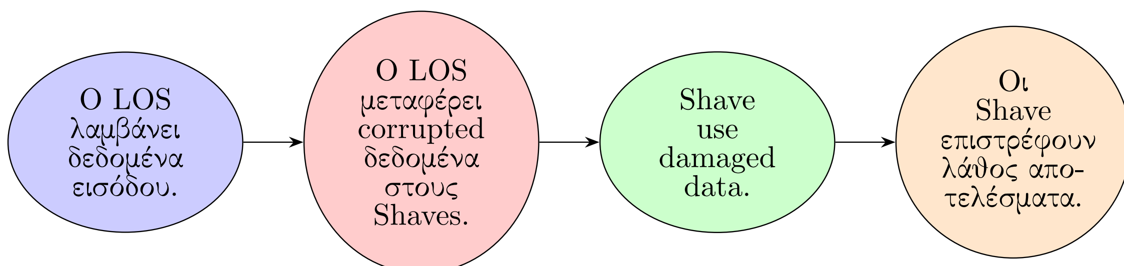
2. Μεταφορά δεδομένων από τον LOS στους Shaves.

3. Επεξεργασία των δεδομένων στους Shaves.

4. Επιστροφή δεδομένων πίσω στον LOS μετά την ολοκλήρωση της επεξεργασίας τους.

Τα παραπάνω βήματα επαναλαμβάνονται μέχρι να ολοκληρωθεί η επεξεργασία των δεδομένων εισόδου.

Συνεπώς, στα παραπάνω βήματα βλέπουμε και μια από τις μεθόδους εισαγωγής σφάλματος που υλοποιήσαμε στην παρούσα ενότητα. Ποιο συγκεκριμένα στα πλαίσια της μεταφοράς κατεστραμμένων δεδομένων εισόδου από τον LOS corrupted δεδομένα μεταφέρονται από στους Shaves προς επεξεργασία. Επομένως, οι Shaves επεξεργάζονται λανθασμένα δεδομένα και όπως είναι αναμενόμενο επιστρέφουν λανθασμένα αποτελέσματα. Η παραπάνω διαδικασία για την εισαγωγή σφάλματος περιγράφεται σχηματικά στο παρακάτω διάγραμμα:



Σχήμα 10: Fault Injection Method: LOS transfers corrupted data to Shaves.

Καταστροφή δεδομένων εισόδου από τους Shaves

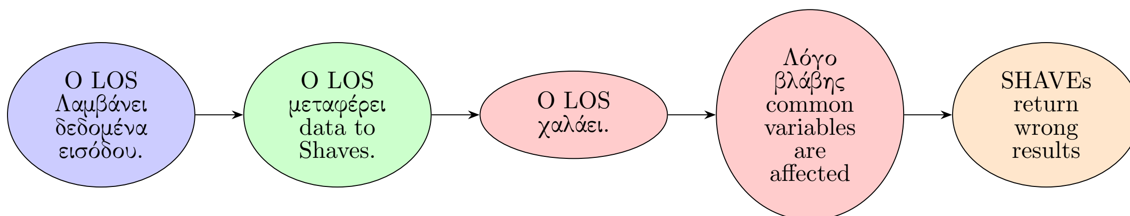
Ακολουθώντας τη βασική προγραμματιστική παραδοχή που αναφέραμε στην προηγούμενη ενότητα, τη μεταφορά δηλαδή δεδομένων εισόδου από τα διάφορα περιφερειακά του συστήματος από τον LEON OS στους Shaves, παρατηρούμε μια νέα περίπτωση εισαγωγή σφάλματος. Τι θα συμβεί εάν τα δεδομένα εισόδου μεταφερθούν σωστά στους Shaves από τον LOS αλλά κάποιοι εκ των Shave χαλάσουν και αρχίσουν να αλλοιώνουν τα δεδομένα εισόδου Τη συμπεριφορά αυτή προσομοιώνουμε με την παρούσα τεχνική εισαγωγής σφάλματος. Παρακάτω φαίνεται και παραστατικά η τεχνική αυτή:

Σχήμα 11: Fault Injection Method: Shaves corrupt their own data.

Καταστροφή δεδομένων κοινών για τους Shaves και τον LEON OS από τον LOS.

Στον προγραμματισμό πολυπύρηνων συστημάτων είναι αρκετά σύνηθες να υπάρχουν κοινές μεταβλητές μεταξύ των διάφορων υπολογιστικών πυρήνων. Αντίστοιχα στην πλατφόρμα Myriad 2 υπάρχει δυνατότητα να υπάρχουν κοινές μεταβλητές μεταξύ των Shaves και του LEON OS. Συνεπώς, υπάρχει ενδεχόμενο σε μια τέτοια προγραμματιστική παραδοχή βλάβη στον LOS να οδηγήσει σε σφάλματα στους Shaves. Επομένως, στη συγκεκριμένη τεχνική έχουμε διαμορφώσει κατάλληλα τον κώδικα έτσι ώστε η αλλαγή που οφείλεται σε προσομοίωση της βλάβης στον LOS να προκαλέσει στους πρόβλημα λειτουργίας στους Shaves σε πραγματικό χρόνο .

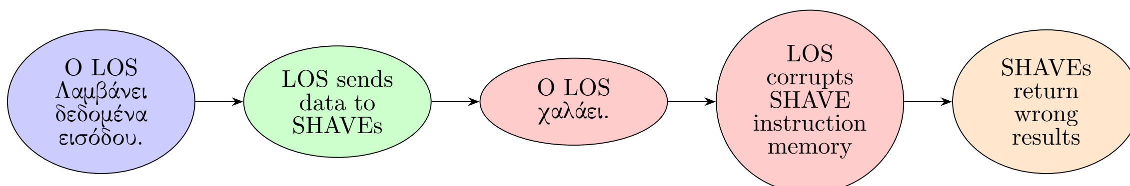
Παρακάτω φαίνεται η προηγούμενη τεχνική σε βήματα:



Σχήμα 12: Fault Injection Method: LOS corrupts shared variable.

Καταστροφή περιεχομένων μνήμης από τον LEON OS

Ένα από τα βασικά χαρακτηριστικά της πλατφόρμας Myriad 2 είναι το γεγονός ότι διαθέτει ενιαίο χώρο μνήμης. Αυτό σημαίνει ότι περιοχές της DDR αλλά και της CMX είναι προσβάσιμες από όλες τις υπολογιστικές μονάδες του συστήματος. Πρέπει όμως, να σημειωθεί ότι η πρόσβαση στη μνήμη δεν είναι ομοιόμορφη κάτι που ενδέχεται να προκαλέσει race conditions. Λόγο τον παραπάνω ιδιοτήτων της Myriad δύναται κάποιο δυσλειτουργικό κομμάτι να καταστρέψει δεδομένα άλλων υποσυστημάτων. Τέλος, ο κώδικας τον Shaves αποθηκεύεται σε συγκεκριμένες θέσεις μνήμης, που καθορίζονται από τον προγραμματιστή, και κατά συνέπεια μπορεί να αλλοιωθεί μέχρι και ο κώδικας που τρέχει στους Shaves.



Σχήμα 13: Fault Injection Method: LOS corrupts shared memory space.

Benchmarks

Προκειμένου να μπορέσουμε να ποσοτικοποιήσουμε την απόδοση των προτεινομένων μοντέλων χρειάστηκε να αναπτύξουμε και να χρησιμοποιήσουμε προγράμματα που αντιστοιχούν σε ένα ρεαλιστικό υπολογιστικό φόρτο για την πλατφόρμα Myriad 2. Δεδομένου ότι η συγκεκριμένη μονάδα χρησιμοποιείται σε εφαρμογές μηχανικής όρασης λογικό είναι τα benchmarks που αναπτύχθηκαν να αντιστοιχούν σε υπολογιστικά δύσκολες πράξεις πάνω σε στοιχεία πινάκων. Στα προγράμματα αυτά μετρήσαμε την απόδοση τους σε συνθήκες κανονικής λειτουργίας αλλά και υπό συνθήκες stress προκειμένου να μπορέσουμε να μετρήσουμε την αποδοτικότητα τόσο των Fault-Tolerant πολιτικών αλλά και των τεχνικών εισαγωγής σφάλματος. Παρακάτω ακολουθεί αναλυτικότερη παρουσίαση των μετροπρογραμμάτων που χρησιμοποιήθηκαν.

2D Convolution

Μια από τις ποιο κλασικές πράξεις ψηφιακής επεξεργασίας σημάτων είναι η συνέλιξη. Η πράξη της συνελιξέως ορίζεται μεταξύ δύο σημάτων τουλάχιστον μιας διάστασης. Στη μονοδιάστατη συνέλιξη οι στιγμιαίες τιμές των σημάτων που κάνουν overlapp αθροίζονται και πολλαπλασιάζονται μεταξύ τους. Επιπλέον, πρέπει να αναφέρουμε ότι το ένα εκ των δύο σημάτων είναι ανεστραμμένο. Η μαθηματική αναπαράσταση της συνέλιξης για διακριτά σήματα είναι:

$$(f * g)(t) = \sum_{-\infty}^{+\infty} f(\tau)g(t-\tau)$$

Ωστόσο, πέραν από διακριτά σήματα υπάρχουν και χρονικά συνεχή. Στην περίπτωση αυτή η συνέλιξη ορίζεται ως:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau$$

Ο ορισμός αυτός επεκτείνεται για περισσότερες από μια διαστάσεις. Η βασική διαφορά μεταξύ της συνέλιξης σε μια και σε δυο διαστάσεις είναι το γεγονός ότι το ένα εκ των δυο σημάτων αναστρέφεται δύο φορές. Η προαναφερθείσα διαδικασία χρησιμοποιείται εκτεταμένα σε διάφορες διοχετεύσεις μηχανικής όρασης που χρησιμοποιούνται ευρέως όπως η ανίχνευση ακμών σε μια εικόνα. Στην περίπτωση των εικόνων διακρίνονται δύο πίνακες ως τα συνελισόμενα σήματα ορίζονται ένας μεγάλος πίνακας δύο διαστάσεων και ένας μικρότερος. Ο μικρότερος πίνακας ονομάζεται πυρήνας. Οι προαναφερθείσες ιδιότητες της πράξης αυτής καθιστούν τη συνέλιξη ως κατάλληλο μετροπρόγραμμα για να μετρηθεί η απόδοση των πολιτικών που σχεδιάστηκαν και υλοποιήθηκαν τόσο για την ανοχή στο σφάλμα αλλά και για την εισαγωγή λάθους.

2D Binning

Το Binning είναι μια διαδικασία κατά την οποία τα περιεχόμενα των pixel γειτονικών περιοχών συνδυάζονται σε ένα super-pixel και είναι μια συνήθης διεργασία για να μειωθεί ο θόρυβος της εικόνας αυξάνοντας τον λόγο σήματος προς θόρυβο της εικόνας. Συνήθως, το Binning συμβαίνει σε ομάδες των τεσσάρων pixel για τον σχηματισμό μιας τετράδας (περιοχή $2x2$). Ωστόσο, υπάρχουν περιπτώσεις στις οποίες δεκαεξάδες pixels (περιοχή $4x4$). Ο λόγος χρήσης περιοχών $4x4$ είναι το γεγονός ότι αυξάνουν τον λόγο σήματος προς θόρυβο τέσσερις φορές αλλά υποτετραπλασιάζουν την ανάλυση. Τέλος, γίνεται και 2D Binning σε περιοχές $2x1$ και $1x2$. Ωστόσο, οι χρήση αυτή δεν είναι διαδεδομένη σε εμπορικές εφαρμογές. Ο βασικός λόγος χρήσης του Binning είναι το γεγονός ότι αυξάνει το λόγο σήματος προς θόρυβο που αποτελεί μια από τις βασικές μετρικές στην ποιότητα των εικόνων.

Επιπλέον, υπάρχουν δύο διαφορετικές κατηγορίες binning το αθροιστικό και το binning μέσου όρου. Το ποιο εκ των δύο ωστόσο επιφέρει το βέλτιστο αποτέλεσμα εξαρτάται ιδιαίτερα από την εφαρμογή. Με λιγότερο θόρυβο τα δεδομένα μπορούν να υποβληθούν σε μεγαλύτερα επίπεδα ενίσχυσης κατά τη διαδικασία της προ επεξεργασίας. Συνεπώς, το 2D Binning είναι ένα ιδιαίτερα χρήσιμο benchmark για να ποσοτικοποιήσουμε την απόδοση των τεχνικών της διπλωματικής.

Ανάλυση Αποτελεσμάτων

Στις επόμενες υποενότητες παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων που πραγματοποιήθηκαν στα πλαίσια της διπλωματικής.

Fault Injection method evaluation

Fault injection method: LEON OS αλλιώνει τα περιεχόμενα μνήμης των SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων αναφορικά με την αλλίωση των περιεχομένων μνήμης των SHAVE από τον LEON OS και για τα δύο μετροπρογράμματα:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	13.5	24.2	31.5	44
Mean absolute error	235.6	137.7	170	172
Mean relative error	15	15	15	15
Max error	255	255	255	255
Max relative error	254	254	254	254
PSNR error (db)	9	9	7	6

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	8.5	12	16	24
Mean absolute error	33.6	15.2	26	18
Mean relative error	2	0.9	1.5	1
Max error	254	176	254	176
Max relative error	241	80	241	166
PSNR error (db)	35	42	34	37

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Το ποσοστό του σφάλματος αυξάνεται καθώς το πλήθος των χαλασμένων SHAVE αυξάνεται.
- Τόσο στην συνέλιξη όσο και στο binning το τελικό σφάλμα δεν ξεπερνάει το 50%. Αυτό οφείλεται στο πλήθως των byte της μνήμης που αλλιώνονται.
- Όλες οι μετρικές πλήν του μέσου απόλυτου σφάλματος παραμένουν σε γενικές γραμμές σταθερό στην περίπτωση της συνέλιξης ενώ στο Binning αλλάζει.

- Το πολύ υψηλό max error οφείλεται στο ότι κατά την αλλιώση διάφορα στοιχεία της μνήμης παίρνουν τιμή 255 από 0 που ήταν η αρχική τους τιμή.

Fault injection method: LEON OS στέλνει λάθος δεδομένα στους SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων σε αναφορά με την αλλίωση των δεδομένων εισόδου των SHAVE από τον LEON OS και για τα δύο μετροπρογράμματα:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	25	50	74.1	99.3
Mean absolute error	8.6	8.6	8.6	7.9
Mean relative error	0.1	0.1	0.1	0.1
Max error	155	160	160	160
Max relative error	6	6	7	7
PSNR error (db)	33	31	29	28

Table 1: LEON OS sends corrupted data to SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	25	48	74	97
Mean absolute error	8.7	8.7	8.7	8.7
Mean relative error	0.3	0.3	0.3	0.3
Max error	72	72	72	72
Max relative error	7	7	7	7
PSNR error (db)	36	36	36	36

Table 2: LEON OS sends corrupted data to SHAVEs: Συγκριτικά αποτελέσματα 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Το ποσοστό του σφάλματος αυξάνεται καθώς το πλήθος των χαλασμένων SHAVE αυξάνεται.
- Τόσο στην συνέλιξη όσο και στο binning το τελικό σφάλμα οριακά φτάνει το 100%.
- Όλες οι μετρικές παραμένουν σε γενικές γραμμές σταθερές τόσο στην περίπτωση της συνέλιξης αλλά και στο Binning.
- Οι τιμές των διαφόρων μετρικών εξαρτόνται από το σφάλμα που έχει εισαχθεί. Συνεπώς, οι τιμές τους παραμένουν σε γενικές γραμμές σταθερές καθώς το σφάλμα είναι το ίδιο για κάθε SHAVE.

Fault injection method: LEON OS αλλιώνει μεταβλητές κοινές με τους SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων αναφορικά με την αλλίωση μεταβλητών κοινών μεταξύ των SHAVE από του LEON OS για τα δύο μετροπρογράμματα:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	24	500	72.7	97.8
Mean absolute error	9.5	10	9.5	10
Mean relative error	0.5	0.4	0.4	0.3
Max error	79	155	160	160
Max relative error	64	73	73	73
PSNR error (db)	32	28	27	25

Table 3: LEON OS corrupts variable shared with SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	23.5	48.2	73.4	99
Mean absolute error	14	14.8	14.5	14.6
Mean relative error	0.9	0.7	0.7	0.8
Max error	178	178	178	178
Max relative error	176	176	173	176
PSNR error (db)	34	31	29	28

Table 4: LEON OS corrupts variable shared with SHAVEs: Συγκριτικά αποτελέσματα 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Το ποσοστό του σφάλματος αυξάνεται καθώς το πλήθος των χαλασμένων SHAVE αυξάνεται.
- Τόσο στην συνέλιξη όσο και στο binning το τελικό σφάλμα οριακά φτάνει το 100%.
- Όλες οι μετρικές παραμένουν σε γενικές γραμμές σταθερές τόσο στην περίπτωση της συνέλιξης αλλά και στο Binning.
- Οι τιμές των διαφόρων μετρικών εξαρτόνται από το σφάλμα που έχει εισαχθεί. Συνεπώς, οι τιμές τους παραμένουν σε γενικές γραμμές σταθερές καθώς το σφάλμα είναι το ίδιο για κάθε SHAVE.

Fault injection method: SHAVE καταστρέφουν τα ίδια τους τα δεδομένα

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων σε αναφορά με την καταστροφή των SHAVE data από τους ίδιους τους SHAVE και για τα δύο μετροπρογράμματα:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	24	500	74.7	99.3
Mean absolute error	8	7.9	8	8
Mean relative error	0.1	0.1	0.1	0.1
Max error	121	155	160	160
Max relative error	7	7	7	77
PSNR error (db)	34	31	29	28

Table 5: SHAVEs corrupt their own data: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	24	47.5	74.6	99
Mean absolute error	9.3	9.3	9.6	9.4
Mean relative error	0.2	0.2	0.2	0.2
Max error	165	165	165	165
Max relative error	12	8	14	14
PSNR error (db)	39	36	34	33

Table 6: SHAVEs corrupt their own data: Συγκριτικά αποτελέσματα 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Το ποσοστό του σφάλματος αυξάνεται καθώς το πλήθος των χαλασμένων SHAVE αυξάνεται.
- Τόσο στην συνέλιξη όσο και στο binning το τελικό σφάλμα οριακά φτάνει το 100%.
- Όλες οι μετρικές παραμένουν σε γενικές γραμμές σταθερές τόσο στην περίπτωση της συνέλιξης αλλά και στο Binning.
- Οι τιμές των διαφόρων μετρικών εξαρτόνται από το σφάλμα που έχει εισαχθεί. Συνεπώς, οι τιμές τους παραμένουν σε γενικές γραμμές σταθερές καθώς το σφάλμα είναι το ίδιο για κάθε SHAVE.

Fault Tolerance policy evaluation

Fault tolerant policy: LEON OS διορθώνει το σφάλμα στη μνήμη του κώδικα των SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης από σφάλμα στην περίπτωση σφάλματος στη μνήμη του κώδικα των SHAVE συνεπεξεργαστών για τα δύο μετροπρογράμματα.

2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	81 ms	81 ms	65 ms	65 ms
2. Hashing memory / SHAVE	4.5 ms	4.5 ms	4.5 ms	4.5 ms
3. Rescheduling run time	0.003 ms	0.004 ms	0.006 ms	0 ms
4. SHAVE rerun time	81 ms	81 ms	87 ms	114 ms
5. Total run time	216 ms	217 ms	206 ms	233 ms
8. Error percentage	0%	0%	0%	0%

Table 7: Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	2 ms	2 ms	2 ms	2 ms
2. Hashing memory / SHAVE	3 ms	3 ms	3 ms	3 ms
3. Rescheduling run time	0.002 ms	0.002 ms	0.003 ms	0 ms
4. SHAVE rerun time	2 ms	2 ms	8 ms	38 ms
5. Total run time	49 ms	49 ms	55 ms	85 ms
8. Error percentage	0.2%	0.4%	0.5%	0%

Table 8: Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα 2D Binning

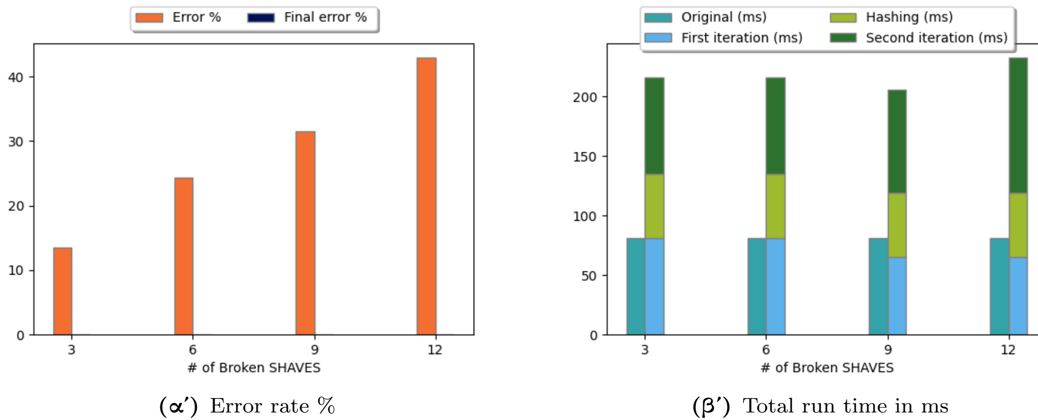
Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Ο αρχικός χρόνος εκτέλεσης των δεδομένων από τους SHAVEs είναι σταθερός ανεξάρτητα από τους SHAVE που είναι χαλασμένοι.
- Αισθητό overhead στον χρόνο εκτέλεσης τοποθετείται από το hashing των δεδομένων.
- Ο χρόνος επαναδρομολόγησης είναι αισθητά μικρός και στην περίπτωση που όλοι οι SHAVE είναι χαλασμένοι μηδενικός καθώς στην περίπτωση αυτή όλοι οι SHAVE γίνονται reset.
- Το τελικό error που παραμένει είναι πολύ μικρό.

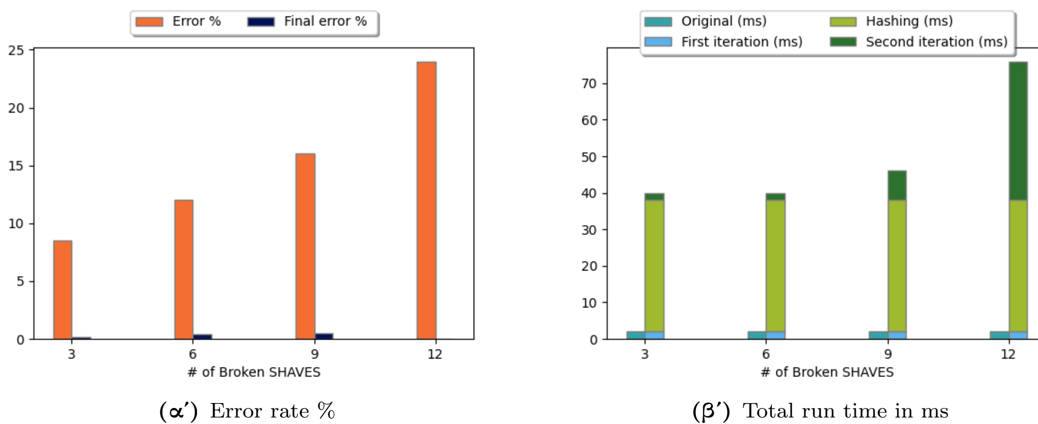
Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι πολύ μικρό σε σχέση με το αρχικό σφάλμα που εισάχθηκε.



(α′) Error rate % (β′) Total run time in ms

Σχήμα 14: Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate % (β′) Total run time in ms

Σχήμα 15: Leon OS corrects SHAVE instruction memory harm: Συγκριτικά αποτελέσματα για το 2D Binning

- Στην περίπτωση του χρόνου εκτέλεσης παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται καθώς ο συνολικός αριθμός των SHAVE που είναι χαλασμένοι χαλάνε. Κάτι το λογικό καθώς όσο το πλήθος τους αυξάνεται τόσο περισσότεροι SHAVE χρειάζεται να ξανά εκτελέσουν το workload τους.
- Το χρονικό time overhead που τοποθετείται από τον χρόνο που χρειάζεται από το data hashing είναι αισθητός.
- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλίξης μετά την εφαρμογή της πολιτικής διόρθωσης σφάλματος είναι περίπου στις 3 φορές τον αρχικό χρόνο εκτέλεσης. Κάτι το λογικό εφόσον απαιτούνται έλεγχοι ορθότητας αλλά και να ξανατρέξουν τα δεδομένα δύο φορές.
- Ο χρόνος εκτέλεσης στην περίπτωση του 2D binning είναι γύρω στις 25 φορές μεγαλύτερος από τον αρχικό. Η αύξηση αυτή οφείλεται στο γεγονός ότι το 2D binning είναι αρκετά ποιο γρήγορο από τη συνέλιξη.

Fault tolerant policy: LEON OS διορθώνει το σφάλμα στη μεταφορά δεδομένων στους SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης από σφάλμα στα δεδομένα που μεταφέρει ο LEONOS στους SHAVE συνεπεξεργαστές για τα δύο μετροπρογράμματα.

2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	81 ms	81 ms	81 ms	81 ms
2. Hashing data / SHAVE	7.4 ms	7.4 ms	7.4 ms	7.4 ms
3. Rescheduling run time	0.002 ms	0.003 ms	0.004 ms	0 ms
4. SHAVE rerun time	81 ms	86 ms	87 ms	81 ms
5. Total run time	252 ms	257 ms	258 ms	252 ms
8. Error percentage	0%	0.2%	0.2%	0%

Table 9: LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	2 ms	2 ms	2 ms	2 ms
2. Hashing data / SHAVE	3.4 ms	3.4 ms	3.4 ms	3.4 ms
3. Rescheduling run time	0.002 ms	0.002 ms	0.003 ms	0 ms
4. SHAVE rerun time	8.3 ms	15.9 ms	7.7 ms	2 ms
5. Total run time	52 ms	59 ms	50 ms	44 ms
8. Error percentage	1.5%	2.6%	0.4%	0%

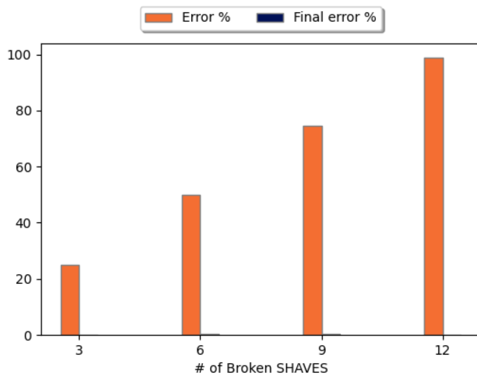
Table 10: LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα 2D Binning

Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:
Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

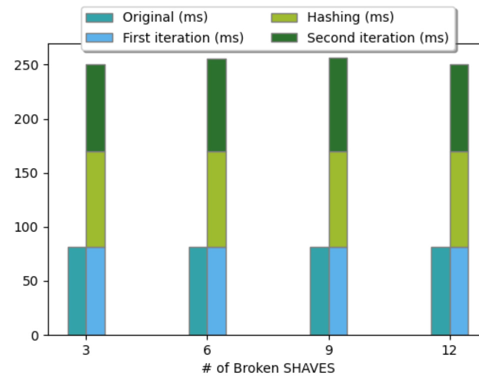
- Ο αρχικός χρόνος εκτέλεσης των δεδομένων από τους SHAVEs είναι σταθερός ανεξάρτητα από τους SHAVE που είναι χαλασμένοι.
- Αισθητό overhead στον χρόνο εκτέλεσης τοποθετείται από το hashing των δεδομένων.
- Ο χρόνος επαναδρομολόγησης είναι αισθητά μικρός και στην περίπτωση που όλοι οι SHAVE είναι χαλασμένοι μηδενικός καθώς στην περίπτωση αυτή όλοι οι SHAVE γίνονται reset.
- Το τελικό error που παραμένει είναι πολύ μικρό.

Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι πολύ μικρό σε σχέση με το αρχικό σφάλμα που εισάχθηκε.
- Στην περίπτωση του χρόνου εκτέλεσης παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται καθώς ο συνολικός αριθμός των SHAVE που είναι χαλασμένοι χαλάνε. Κάτι το λογικό καθώς όσο το

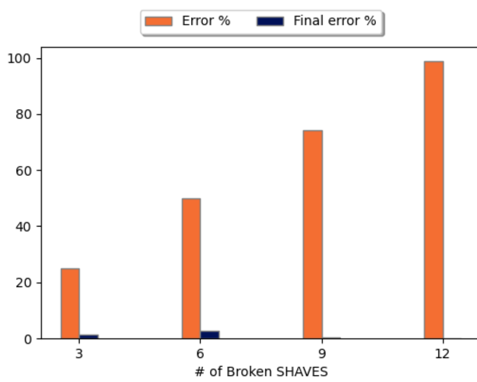


(α′) Error rate %

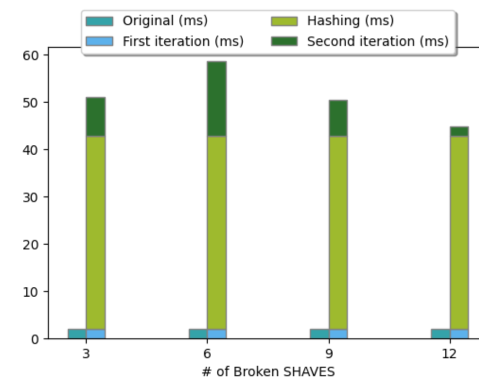


(β′) Total run time in ms

Σχήμα 16: LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate %



(β′) Total run time in ms

Σχήμα 17: LEON OS corrects corrupted data transferred: Συγκριτικά αποτελέσματα για το 2D Binning

πλήθος τους αυξάνεται τόσο περισσότεροι SHAVE χρειάζεται να ξανά εκτελέσουν το workload τους.

- Το χρονικό time overhead που τοποθετείται από τον χρόνο που χρειάζεται από το data hashing είναι αισθητός.
- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλίξης μετά την εφαρμογή της πολιτικής διόρθωσης σφάλματος είναι περίπου στις 3 φορές τον αρχικό χρόνο εκτέλεσης. Κάτι το λογικό εφόσον απαιτούνται έλεγχοι ορθότητας αλλά και να ξανατρέξουν τα δεδομένα δύο φορές.
- Ο χρόνος εκτέλεσης στην περίπτωση του 2D binning είναι γύρω στις 25 φορές μεγαλύτερος από τον αρχικό. Η αύξηση αυτή οφείλεται στο γεγονός ότι το 2D binning είναι αρκετά ποιο γρήγορο από τη συνέλιξη.
- Αύξηση στον χρόνο επανεκτέλεσης καθώς το πλήθος των SHAVE που είναι χαλασμένοι αλλάζει οφείλεται στο γεγονός ότι όσο περισσότεροι συνεπεξεργαστές είναι χαλασμένοι το workload τους εκτελείται κατά ριπάς. Δηλαδή, άμα 9 SHAVE είναι χαλασμένοι τότε οι λειτουργικοί SHAVE που υπάρχουν για να εκτελέσουν το workload τους είναι 3. Συνεπώς, τα δεδομένα των χαλασμένων θα πρέπει να εκτελεστούν ανά τρία. Τέλος, λόγο της αναμονής αυτής εισάγεται επιπλέον καθυστέρηση.

Fault tolerant policy: LEON OS διορθώνει το σφάλμα σε μεταβλητή κοινή με τους SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης από σφάλμα σε μεταβλητή κοινή ανάμεσα στον LEONOS και στους SHAVE συνεπεξεργαστές για τα δύο μετροπρογράμματα.

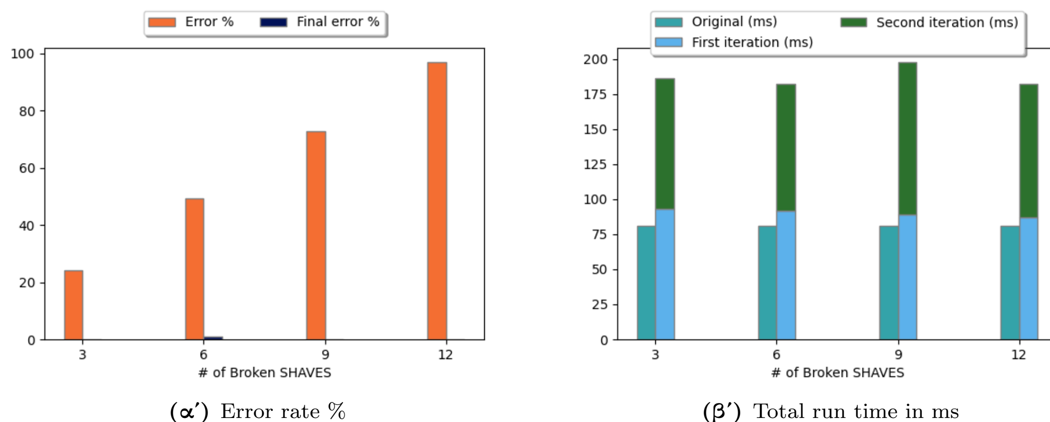
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	93 ms	92 ms	89 ms	87 ms
2. Rescheduling run time	0.002 ms	0.003 ms	0.004 ms	0 ms
3. SHAVE rerun time	93 ms	90 ms	109 ms	95 ms
4. Total run time	186 ms	182 ms	199 ms	183 ms
5. Error percentage	0%	1%	0%	0%

Table 11: Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	31 ms	33 ms	24 ms	21 ms
2. Rescheduling run time	0.001 ms	0.001 ms	0.003 ms	0 ms
3. SHAVE rerun time	27 ms	22 ms	62 ms	32 ms
4. Total run time	58 ms	55 ms	89 ms	53 ms
5. Error percentage	0%	0%	0%	0%

Table 12: Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα 2D Binning

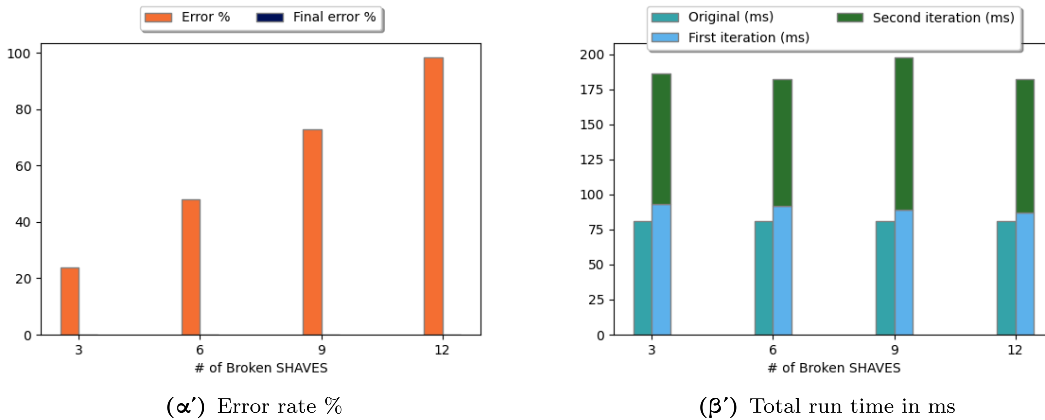
Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 18: Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 19: Leon OS corrects variables shared with SHAVEs: Συγκριτικά αποτελέσματα για το 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Ο αρχικός χρόνος εκτέλεσης των δεδομένων από τους SHAVEs είναι σταθερά μειούμενος α-νάλογα με τους SHAVE που είναι χαλασμένοι.
- Πλέον το hashing των δεδομένων παραλληλοποιείται και κατά συνέπεια δεν παίρνει τόσο χρόνο όσο οι προηγούμενες περιπτώσεις.
- Ο χρόνος επαναδρομολόγησης είναι αισθητά μικρός και στην περίπτωση που όλοι οι SHAVE είναι χαλασμένοι μηδενικός καθώς στην περίπτωση αυτή όλοι οι SHAVE γίνονται reset.
- Το τελικό error που παραμένει είναι οριακά 0%.

Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι ελάχιστο σε σύγκριση με το αρχικό σφάλμα που εισάχθηκε.
- Στην περίπτωση του χρόνου εκτέλεσης παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται καθώς ο συνολικός αριθμός των SHAVE που είναι χαλασμένοι χαλάνε. Κάτι το λογικό καθώς όσο το πλήθος τους αυξάνεται τόσο περισσότεροι SHAVE χρειάζεται να ξανά εκτελέσουν το workload τους.
- Το χρονικό time overhead που τοποθετείται από τον χρόνο που χρειάζεται από το data hashing είναι αισθητός.
- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλίξης μετά την εφαρμογή της πο-λιτικής διόρθωσης σφάλματος είναι περίπου στις 2.5 φορές ο αρχικός χρόνος εκτέλεσης. Κάτι το λογικό εφόσον απαιτούνται έλεγχοι ορθότητας αλλά και να ξανατρέξουν τα δεδομένα δύο φορές.
- Ο χρόνος εκτέλεσης στην περίπτωση του 2D binning είναι γύρω στις 25-30 φορές μεγαλύτερος από τον αρχικό. Η αύξηση αυτή οφείλεται στο γεγονός ότι το 2D binning είναι αρκετά ποιο γρήγορο από τη συνέλιξη.
- Αύξηση στον χρόνο επανεκτέλεσης καθώς το πλήθος των SHAVE που είναι χαλασμένοι αλλάζει οφείλεται στο γεγονός ότι όσο περισσότεροι συνεπεξεργαστές είναι χαλασμένοι το workload τους εκτελείται κατά ρυπάς. Δηλαδή, άμα 9 SHAVE είναι χαλασμένοι τότε οι λειτουργικοί SHAVE που υπάρχουν για να εκτελέσουν το workload τους είναι 3. Συνεπώς, τα δεδομένα των

χαλασμένων θα πρέπει να εκτελεστούν ανά τρία. Τέλος, λόγο της αναμονής αυτής εισάγεται επιπλέον καθυστέρηση. Εξαίρεση, αποτελεί η περίπτωση που όλοι οι SHAVE είναι χαλασμένοι καθώς στην περίπτωση αυτή όλοι οι SHAVE χρειάζονται επανεκκίνησή.

Fault tolerant policy: SHAVEs διορθώνουν το σφάλμα σε μεταβλητή κοινή με τον LEON OS

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης από σφάλμα σε μεταβλητή κοινή ανάμεσα στον LEONOS και στους SHAVE συνεπεξεργαστές για τα δύο μετροπρογράμματα. Στη συγκεκριμένη περίπτωση, οι SHAVE αυτό διορθώνουν το σφάλμα και δεν κάνουν notify τον LEON OS.

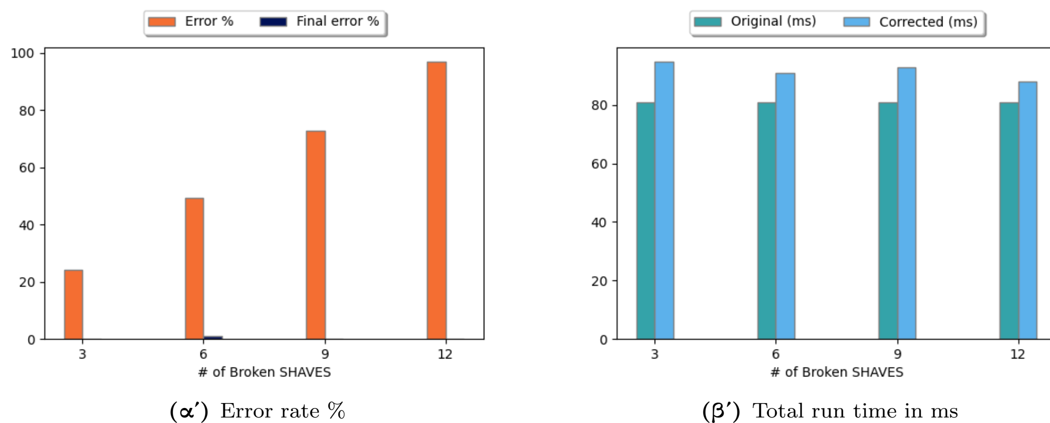
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	95 ms	91 ms	93 ms	88 ms
2. Error percentage	0%	1%	0%	0%

Table 13: SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	31 ms	33 ms	24 ms	21 ms
2. Error percentage	0%	0%	0%	0%

Table 14: SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα 2D Binning

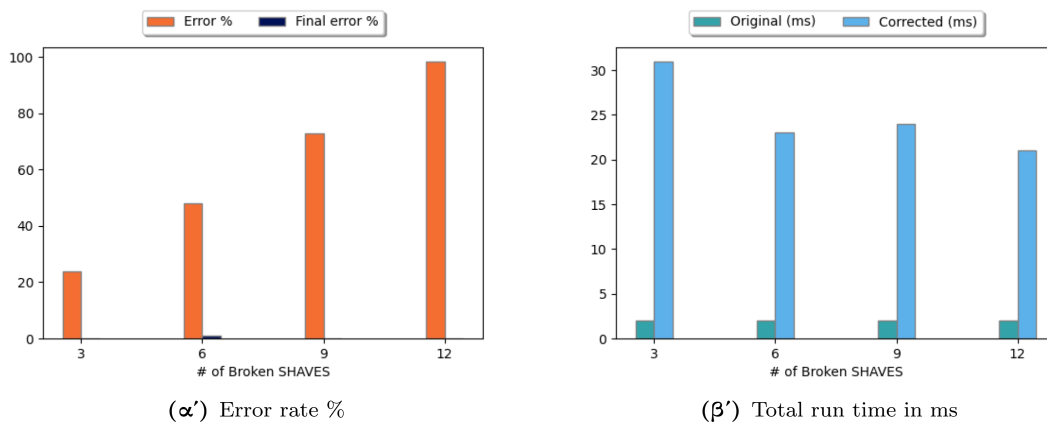
Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 20: SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 21: SHAVEs correct variables shared with LEON OS: Συγκριτικά αποτελέσματα για το 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Ο αρχικός χρόνος εκτέλεσης των δεδομένων από τους SHAVEs είναι σταθερά μειούμενος ανάλογα με τους SHAVE που είναι χαλασμένοι.
- Πλέον το hashing των δεδομένων παραλληλοποιείται και κατά συνέπεια δεν παίρνει τόσο χρόνο όσο οι προηγούμενες περιπτώσεις.
- Ο χρόνος επαναδρομολόγησης πλέον είναι ανύπαρκτος καθώς οι SHAVE αντιλαμβάνονται το σφάλμα τους και το διορθώνουν μόνοι τους με τη χρήση ενός backup frame που διαθέτουν.
- Ο χρόνος εκτέλεσης μέχρι τώρα είναι ο μικρότερος παρατηρημένος καθώς δεν απαιτείται κάποια επανεκτέλεση.
- Το τελικό error που παραμένει είναι οριακά 0%.

Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι ελάχιστο σε σύγκριση με το αρχικό σφάλμα που εισάχθηκε.
- Το χρονικό time overhead που τοποθετείται από τον χρόνο που χρειάζεται από το data hashing είναι αισθητός αλλά μικρότερο από το αρχικό λόγο παραλληλίας.
- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλιξης μετά την εφαρμογή της πολιτικής διόρθωσης σφάλματος είναι περίπου στις 1.2 φορές ο αρχικός χρόνος εκτέλεσης. Κάτι το λογικό εφόσον απαιτούνται έλεγχοι ορθότητας και πλέον δε χρειάζεται να τρέξει δύο φορές κάθε SHAVE το workload του.
- Ο χρόνος εκτέλεσης στην περίπτωση του 2D binning είναι γύρω στις 15 φορές μεγαλύτερος από τον αρχικό. Η αύξηση αυτή οφείλεται στο γεγονός ότι το 2D binning είναι αρκετά ποιο γρήγορο από τη συνέλιξη και το overhead που τοποθετείται λόγο το data hashing είναι πολύ μεγαλύτερο από τον χρόνο εκτέλεσης του ίδιο του μετροπρογράμματος.

Fault tolerant policy: LEON OS διορθώνει το σφάλμα που προκάλεσαν οι SHAVE στον εαυτό τους

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης από σφάλμα που προκαλούν στον εαυτό τους. Στην παρούσα περίπτωση ο LEON OS είναι υπεύθυνος για να διορθώσει το σφάλμα αφού ενημερωθεί για την ύπαρξη του από τους προβληματικούς συνεπεξεργαστές.

Παρακάτω παρατίθενται τα αποτελέσματα και για τα δύο μετροπρογράμματα.

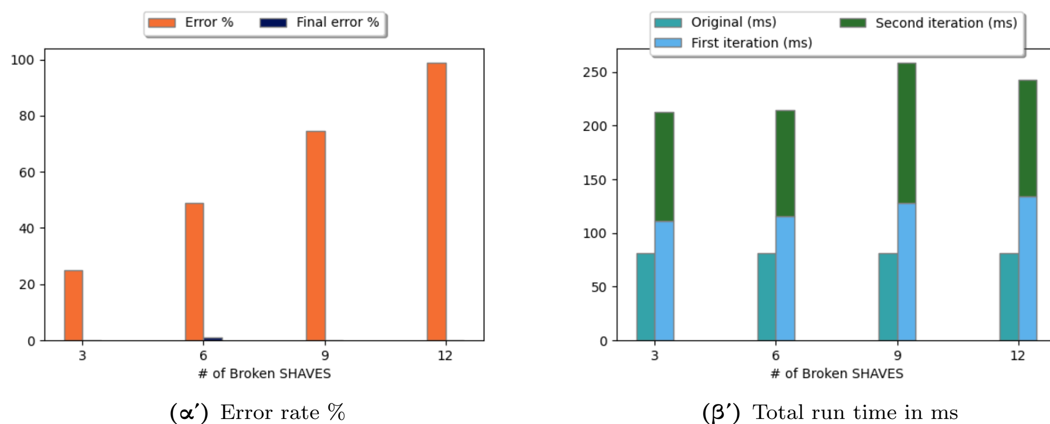
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	111 ms	116 ms	128 ms	134 ms
2. Rescheduling run time	0.002 ms	0.002 ms	0.004 ms	0 ms
3. SHAVE rerun time	102 ms	99 ms	131 ms	109 ms
4. Total run time	213 ms	215 ms	259 ms	243 ms
5. Error percentage	0%	1%	0%	0%

Table 15: LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	81 ms	98 ms	126 ms	142 ms
2. Rescheduling run time	0.001 ms	0.002 ms	0.003 ms	0 ms
3. SHAVE rerun time	52 ms	47 ms	122 ms	62 ms
4. Total run time	134 ms	146 ms	249 ms	204 ms
5. Error percentage	0%	1%	0%	0%

Table 16: LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα 2D Binning

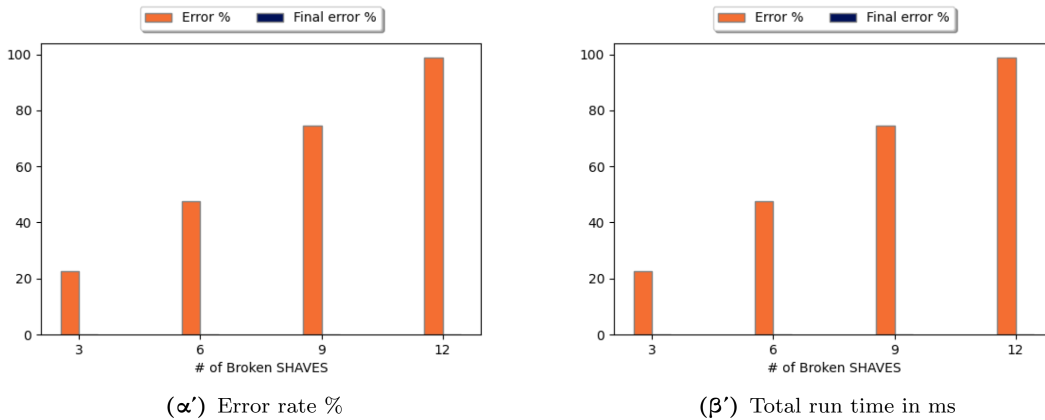
Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:



(α') Error rate %

(β') Total run time in ms

Σχήμα 22: LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 23: LEON OS corrects data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Ο αρχικός χρόνος εκτέλεσης των δεδομένων από τους SHAVEs είναι σταθερά αυξανόμενος ανάλογα με τους SHAVE που είναι χαλασμένοι. Αυτό οφείλεται στη διαδικασία εισαγωγής σφάλματος.
- Πλέον το hashing των δεδομένων παραλληλοποιείται και κατά συνέπεια δεν παίρνει τόσο χρόνο όσο οι προηγούμενες περιπτώσεις.
- Ο χρόνος επαναδρομολόγησης είναι αισθητά μικρός και στην περίπτωση που όλοι οι SHAVE είναι χαλασμένοι μηδενικός, καθώς στην περίπτωση αυτή όλοι οι SHAVE γίνονται reset.
- Το τελικό error που παραμένει είναι οριακά 0%.

Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι ελάχιστο σε σύγκριση με το αρχικό σφάλμα που εισάχθηκε.
- Στην περίπτωση του χρόνου εκτέλεσης παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται καθώς ο συνολικός αριθμός των SHAVE που είναι χαλασμένοι χαλάνε. Κάτι το λογικό καθώς όσο το πλήθος τους αυξάνεται τόσο περισσότεροι SHAVE χρειάζεται να ξανά εκτελέσουν το workload τους.
- Το χρονικό time overhead που τοποθετείται από τον χρόνο που χρειάζεται από το data hashing είναι αισθητός.
- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλίξης μετά την εφαρμογή της πολιτικής διόρθωσης σφάλματος είναι περίπου στις 2.5 φορές ο αρχικός χρόνος εκτέλεσης. Κάτι το λογικό εφόσον απαιτούνται έλεγχοι ορθότητας αλλά και να ξανατρέξουν τα δεδομένα δύο φορές.
- Ο χρόνος εκτέλεσης στην περίπτωση του 2D binning είναι πάνω από 30 φορές μεγαλύτερος από τον αρχικό. Η αύξηση αυτή οφείλεται στο γεγονός ότι το 2D binning είναι αρκετά ποιο γρήγορο από τη συνέλιξη.
- Αύξηση στον χρόνο επανεκτέλεσης καθώς το πλήθος των SHAVE που είναι χαλασμένοι αλλάζει οφείλεται στο γεγονός ότι όσο περισσότεροι συνεπεξεργαστές είναι χαλασμένοι το workload τους εκτελείται κατά ρυπάς. Δηλαδή, άμα 9 SHAVE είναι χαλασμένοι τότε οι λειτουργικοί

SHAVE που υπάρχουν για να εκτελέσουν το workload τους είναι 3. Συνεπώς, τα δεδομένα των χαλασμένων θα πρέπει να εκτελεστούν ανά τρία. Τέλος, λόγο της αναμονής αυτής εισάγεται επιπλέον καθυστέρηση. Εξαίρεση, αποτελεί η περίπτωση που όλοι οι SHAVE είναι χαλασμένοι καθώς στην περίπτωση αυτή όλοι οι SHAVE χρειάζονται επανεκκίνησή.

Fault tolerant policy: SHAVEs διορθώνουν το σφάλμα που προκάλεσαν στον εαυτό τους

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης από σφάλμα που προκαλούν οι SHAVE στα ίδια τους τα δεδομένα για τα δύο μετροπρογράμματα. Στη συγκεκριμένη περίπτωση, οι SHAVE αυτό διορθώνουν το σφάλμα και δεν κάνουν notify τον LEON OS.

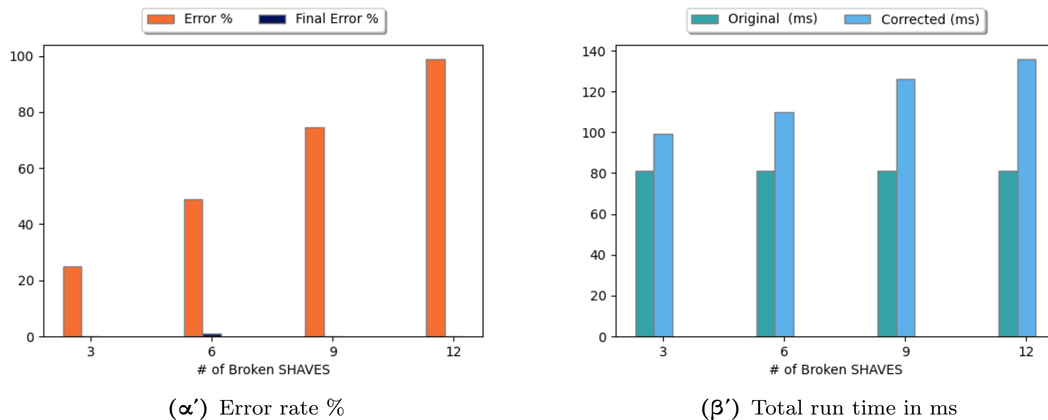
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	99 ms	110 ms	126 ms	136 ms
2. Error percentage	0%	1%	0%	0%

Table 17: SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	52 ms	62 ms	76 ms	85 ms
2. Error percentage	0%	0%	0%	0%

Table 18: SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα 2D Binning

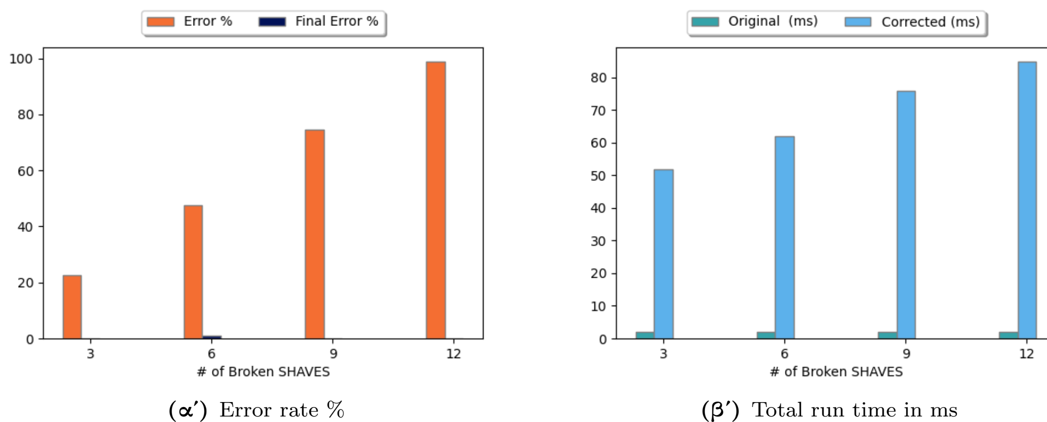
Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 24: SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 25: SHAVEs correct data destroyed by SHAVEs: Συγκριτικά αποτελέσματα για το 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Ο αρχικός χρόνος εκτέλεσης των δεδομένων από τους SHAVEs είναι σταθερά αυξανόμενος ανάλογα με τους SHAVE που είναι χαλασμένοι. Η αύξηση αυτή οφείλεται στην εισαγωγή του σφάλματος.

- Πλέον το hashing των δεδομένων παραλληλοποιείται και κατά συνέπεια δεν παίρνει τόσο χρόνο όσο οι προηγούμενες περιπτώσεις.

- Ο χρόνος επαναδρομολόγησης πλέον είναι ανύπαρκτος καθώς οι SHAVE αντιλαμβάνονται το σφάλμα τους και το διορθώνουν μόνοι τους με τη χρήση ενός backup frame που διαθέτουν.

- Ο χρόνος εκτέλεσης μέχρι τώρα είναι ο δεύτερος μικρότερος παρατηρημένος καθώς δεν απαιτείται κάποια επανεκτέλεση.

- Το τελικό error που παραμένει είναι οριακά 0%.

Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι ελάχιστο σε σύγκριση με το αρχικό σφάλμα που εισάχθηκε.

- Το χρονικό time overhead που τοποθετείται από τον χρόνο που χρειάζεται από το data hashing είναι αισθητός αλλά μικρότερο από το αρχικό λόγο παραλληλίας.

- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλιξης μετά την εφαρμογή της πολιτικής διόρθωσης σφάλματος είναι περίπου στις 1.5 φορές ο αρχικός χρόνος εκτέλεσης. Κάτι το λογικό εφόσον απαιτούνται έλεγχοι ορθότητας και πλέον δε χρειάζεται να τρέξει δύο φορές κάθε SHAVE το workload του.

- Ο χρόνος εκτέλεσης στην περίπτωση του 2D binning είναι πάνω από 30 φορές μεγαλύτερος από τον αρχικό. Η αύξηση αυτή οφείλεται στο γεγονός ότι το 2D binning είναι αρκετά ποιο γρήγορο από τη συνέλιξη και το overhead που τοποθετείται λόγο το data hashing είναι πολύ μεγαλύτερο από τον χρόνο εκτέλεσης του ίδιο του μετροπρογράμματος. Επιπλέον, υπάρχει ένα χρονικό overhead για την εισαγωγή του σφάλματος που οδηγεί σε αυξημένο run time.

Fault tolerant policy: 3 Voting System διορθώνει τα εσφάλμένα δεδομένα που μετέφερε ο LEON OS στους SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης απο εσφαλμένη μεταφορά δεδομένων από τον LEON OS στους SHAVEs και για τα δύο μετροπρογράμματα. Στη συγκεκριμένη περίπτωση το σύστημα που χρησιμοποιείται για την ανάνηψή είναι ένα σύστημα τριών ψηφοφόρων.

Παρακάτω βρίσκονται οι σχετικοί πίνακες των αποτελεσμάτων:

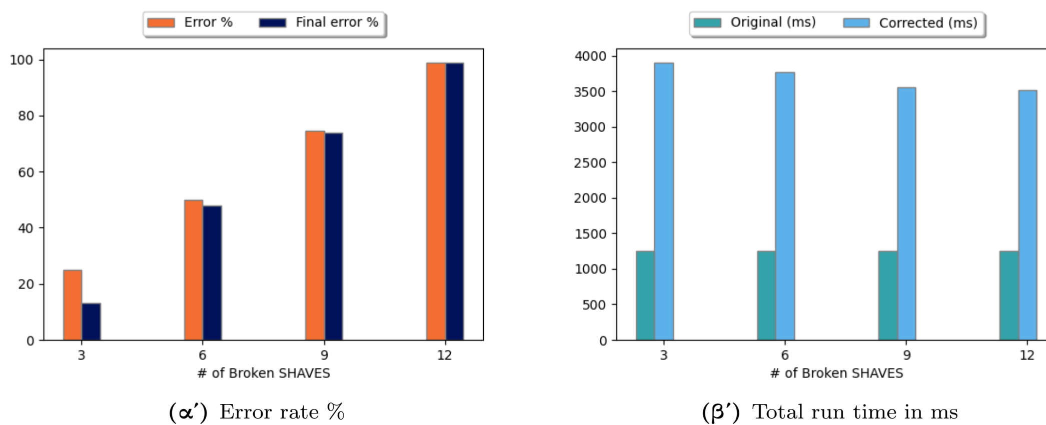
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	3908 ms	3765 ms	3559 ms	3518 ms
2. Error percentage	13%	48%	74%	99%

Table 19: 3 Voting system: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	15 ms	14 ms	14 ms	14 ms
2. Error percentage	11%	48%	69%	99%

Table 20: 3 Voting system: Συγκριτικά αποτελέσματα 2D Binning

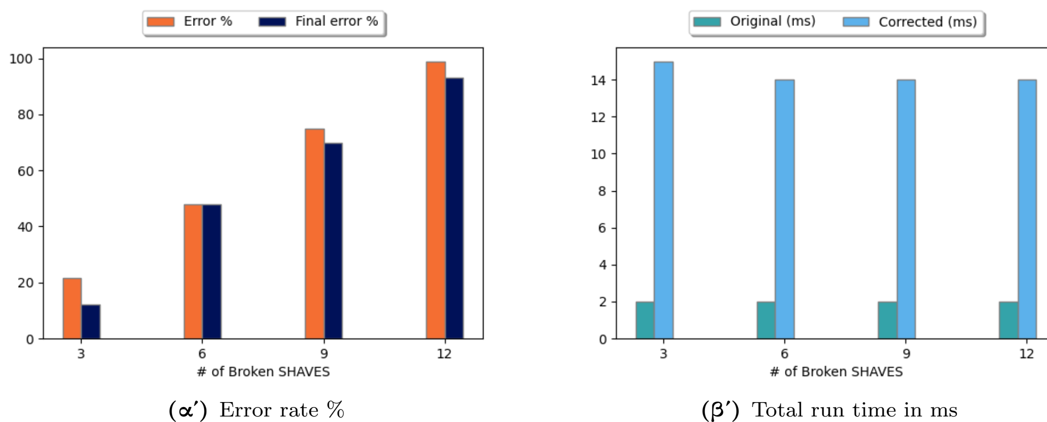
Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 26: 3 Voting system: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 27: 3 Voting system: Συγκριτικά αποτελέσματα για το 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Ο χρόνος εκτέλεσης είναι σχετικά σταθερός.
- Πλέον δεν πραγματοποιείται hashing.
- Ο χρόνος επαναδρομολόγησης πλέον είναι ανύπαρκτος καθώς δεν υπάρχει επανεκτέλεση του workload από τους SHAVE. Στη συγκεκριμένη περίπτωση απλά εισάγεται redundancy προκειμένου αποτελέσματα να συγκριθούν και να επανυπολογίθσουν.
- Η αισθητή αύξηση του χρόνου οφείλεται στο γεγονός ότι δεν έχουμε ενεργοποιήσει το DMA για μεταφορά δεδομένων από και προς τη CMX.
- Το τελικό error είναι αισθητό.
- Το error οφείλεται στο γεγονός ότι αν έχουν χαλάσει πάνω από τους μισούς SHAVE τότε τουλάχιστον τις μισές κάλπες θα επιστρέψουν λάθος αποτελέσματα.

Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι συγκρίσιμο με το αρχικό σφάλμα που εισάχθηκε.
- Το χρονικό overhead οφείλεται στο resource underutilization των SHAVE.
- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλίξης και του 2D binningμετά την εφαρμογή της πολιτικής διόρθωσης σφάλματος είναι περίπου στις 4 φορές ο αρχικός χρόνος εκτέλεσης. Κάτι το λογικό εφόσον οι SHAVE ανά τρεις εκτελούν το ίδιο φορτίο.
- Στην περίπτωση του 2D binning ο νέος χρόνος εκτέλεσης είναι καλύτερος από τον χρόνο των μεθόδων που απαιτούσαν data hashing καθώς ο χρόνος που απαιτείται για το data hashing ήταν αισθητά μεγαλύτερος από τον χρόνο εκτέλεσης του μετροπρογραμμάτος υποχρησιμοποιόντας τους διαθέσιμους πόρους. Αντίθετα, στην περίπτωση της συνέλιξης που ο χρόνος του data hashing είναι συγκρίσιμος ο χρόνος εκτέλεσης είναι αισθητά χειρότερος σε σχέση με τις προηγούμενες μεθόδους.

Fault tolerant policy: 5 Voting System διορθώνει τα εσφάλμένα δεδομένα που μετέφερε ο LEON OS στους SHAVE

Στην παρούσα υποενότητα παρατίθενται τα αποτελέσματα των πειραματικών μετρήσεων για την πολιτική ανάνηψης απο εσφαλμένη μεταφορά δεδομένων από τον LEON OS στους SHAVEs και για τα δύο μετροπρογράμματα. Στη συγκεκριμένη περίπτωση το σύστημα που χρησιμοποιείται για την ανάνηψή είναι ένα σύστημα πέντε ψηφοφόρων.

Παρακάτω βρίσκονται οι σχετικοί πίνακες των αποτελεσμάτων:

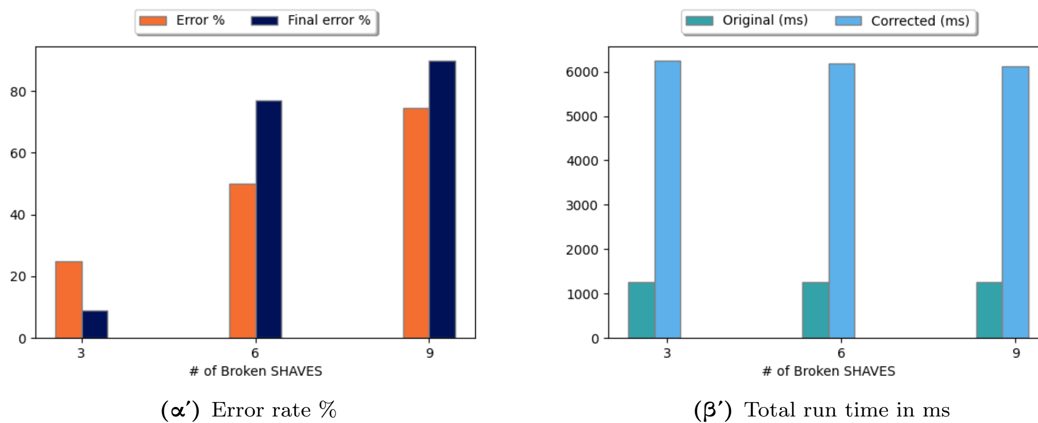
2D Convolution			
# of Broken SHAVES	3	6	9
1. Initial SHAVE run time	6257 ms	6185 ms	6130 ms
2. Error percentage	9%	77%	99%

Table 21: 5 Voting system: Συγκριτικά αποτελέσματα 2D Convolution

2D Binning			
# of Broken SHAVES	3	6	9
1. Initial SHAVE run time	28 ms	28 ms	28 ms
2. Error percentage	10%	54%	88%

Table 22: 5 Voting system: Συγκριτικά αποτελέσματα 2D Binning

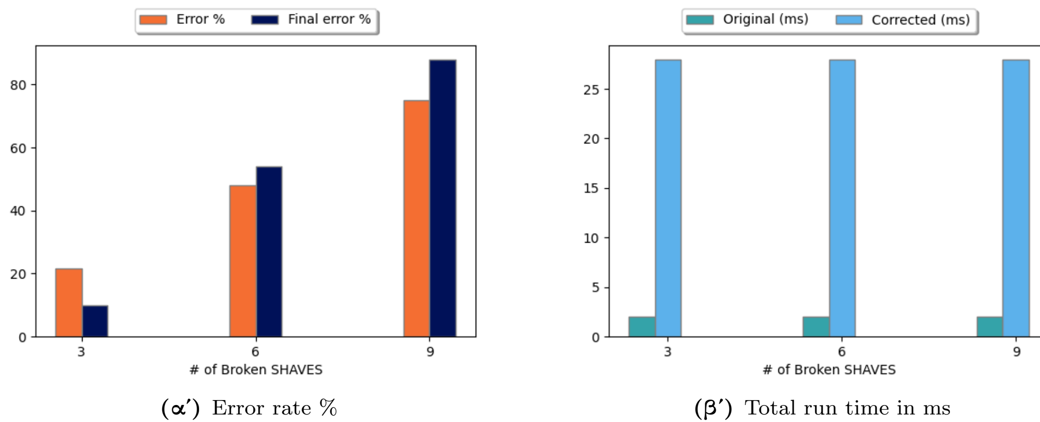
Επιπλέον, παρακάτω παρατίθενται και διαγραμματικά τα αποτελέσματα σε σχέση τόσο με τον αρχικό χρόνο εκτέλεσης αλλά και με το αρχικό error rate:



(α΄) Error rate %

(β΄) Total run time in ms

Σχήμα 28: 5 Voting system: Συγκριτικά αποτελέσματα για το 2D Convolution



(α′) Error rate %

(β′) Total run time in ms

Σχήμα 29: 5 Voting system: Συγκριτικά αποτελέσματα για το 2D Binning

Αναλύοντάς τους παραπάνω πίνακες παρατηρούμε τα παρακάτω και για τα δύο benchmarks:

- Ο χρόνος εκτέλεσης είναι σχετικά σταθερός.
- Πλέον δεν πραγματοποιείται hashing.
- Ο χρόνος επαναδρομολόγησης πλέον είναι ανύπαρκτος καθώς δεν υπάρχει επανεκτέλεση του workload από τους SHAVE. Στη συγκεκριμένη περίπτωση απλά εισάγεται redundancy προκειμένου αποτελέσματα να συγκριθούν και να επανυπολογίθσουν.
- Η αισθητή αύξηση του χρόνου οφείλεται στο γεγονός ότι δεν έχουμε ενεργοποιήσει το DMA για μεταφορά δεδομένων από και προς τη CMX.
- Το τελικό error είναι αισθητό.
- Το error οφείλεται στο γεγονός ότι αν έχουν χαλάσει πάνω από τους μισούς SHAVE τότε τουλάχιστον οι μισές κάλπες θα επιστρέψουν λάθος αποτελέσματα.
- Δύο SHAVE δε χρησιμοποιούνται καθόλου.

Παρατηρώντας τα παραπάνω διαγράμματα καταλήγουμε στις παρακάτω παρατηρήσεις:

- Το ποσοστό σφάλματος είναι συγκρίσιμο με το αρχικό σφάλμα που εισάχθηκε.
- Το χρονικό overhead οφείλεται στο resource underutilization των SHAVE.
- Ο χρόνος εκτέλεσης στην περίπτωση της δισδιάστατης συνέλίξης και του 2D binningμετά την εφαρμογή της πολιτικής διόρθωσης σφάλματος είναι περίπου στις 8 φορές ο αρχικός χρόνος εκτέλεσης. Κάτι το λογικό εφόσον οι SHAVE ανά 5 εκτελούν το ίδιο φορτίο.
- Στην περίπτωση του 2D binning ο νέος χρόνος εκτέλεσης είναι καλύτερος από τον χρόνο των μεθόδων που απαιτούσαν data hashing καθώς ο χρόνος που απαιτείται για το data hashing ήταν αισθητά μεγαλύτερος από τον χρόνο εκτέλεσης του μετροπρογραμμάτος υποχρησιμοποιόντας τους διαθέσιμους πόρους. Αντίθετα, στην περίπτωση της συνέλιξης που ο χρόνος του data hashing είναι συγκρίσιμος ο χρόνος εκτέλεσης είναι αισθητά χειρότερος σε σχέση με τις προηγούμενες μεθόδους.

Chapter 1

Introduction

In application development, the assumption is often made that there is no possibility of an error, either at the software or hardware level. However, this assumption is often disproved and as a direct consequence leads to disastrous results. Under these circumstances, a new need arises to ensure high availability from our computing systems. Consequently, a new way of designing computing systems arises, called Fault-Tolerant Computing [2].

One of the most widespread branches of computing systems is embedded systems. These systems, unlike general-purpose computing systems, are characterized by several limitations. First, they serve only one specialized function. In addition, they have limitations in terms of cost, size, energy consumption and performance. Therefore, these systems must fit on a single chip and be able to process data in real time with as little energy consumption as possible to maintain their battery life. In addition, they must react in real time. That is, they must be able to react to changes in the environment with an accuracy of a few seconds and calculate various results in real time without time delay. Let us consider the autopilot of a car, for example. It constantly checks and reacts to the speed and brake sensors. It must, in addition, compute acceleration and deceleration in real time, as a late control can lead to loss of control of the car.

Based on the foregoing, we conclude that, embedded systems in safety-critical systems [3] where incorrect operation of the software or hardware could lead to catastrophic consequences. Therefore, the need for both high availability and confidence in the correct system functioning. Therefore, the applications which are designed for these systems must also be based on the Fault-Tolerant assumption mentioned earlier.

However, a question is raised by the above, how can a design team check whether its Fault-Tolerant design will work according to the specified specifications in case of a fault; that is, how can the availability of the system be practically checked. To answer the above questions, a new testing method has been developed called Fault Injection [4]. According to this method, designers intentionally inject a fault into their system in order to be able to test its behavior under incorrect conditions. The introduction of error can be done either at the hardware level (e.g., deliberate exposure of a system to electromagnetic radiation in order to change the contents of its registers[5]) or at the software level (e.g., introduction of random data into the system[6]).

Below is a visual representation of the evaluation process of a Fault-Tolerant system:

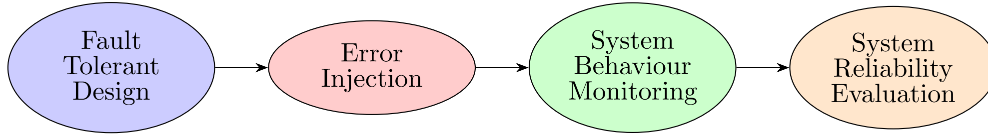


Figure 1.1: Evaluation of a Fault Tolerant system.

According to the above needs, this thesis was created to evaluate the behavior of various Fault-Tolerance techniques on highly heterogeneous architectures such as VPUs. Furthermore, in the context of this effort, various Fault-Injection techniques were implemented in order to to evaluate the final availability of the system.

Background

Vision Processing Units

Vision Processing Units are a new class of microprocessors whose goal is to provide high performance with very low power consumption. In addition, these microprocessors are suitably optimized to run pre-trained convolutional neural network models. Therefore, they are ideal for running Ai at the Edge applications.

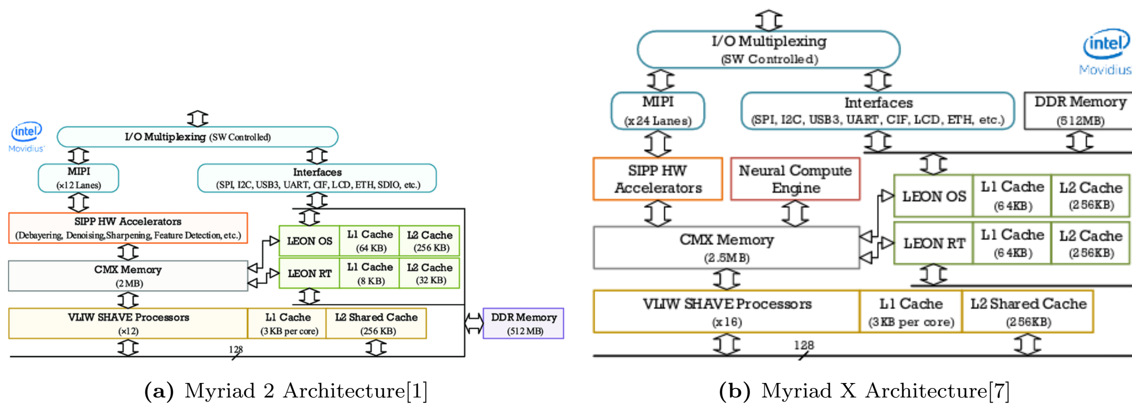
Compared to other hardware accelerators such as F.P.G.A and G.P.U, they have worse performance. However, what makes them ideal for embedded systems is the high efficiency per Watt consumed. In addition, V.P.Us are characterized by high programming difficulty. Therefore, it is a platform with several tradeoffs that came, possibly, to redefine Edge Computing.

Intel Movidius Myriad Family of V.P.Us

The most widely used V.P.U series is the Myriad series developed by Movidius. Movidius is a subsidiary of Intel that was acquired in 2016 for 400*million* [**?**].

The models in the Myriad family are Myriad 2 and Myriad X.

Below, we see the architecture of the aforementioned microprocessors:



(a) Myriad 2 Architecture[1] **(b)** Myriad X Architecture[7]

Figure 1.2: Myriad Architecture Myriad

As can be seen from the above figures, both Myriad 2 and Myriad X have two 32 bit LEON4 general purpose RISC SPARCv8 processors of (LEON OS, LEON RT) architecture. The LEON OS supports a Real Time Operating System called, RTEMS while the LEON RT is responsible for peripheral management. Responsible for the high throughput of the system are the Streaming Hybrid Architecture Vector Engines (SHAVEs) architecture kernels 128 bit, VLIW and SIMD. The orchestration of the operation of the SHAVEs is performed by the aforementioned general purpose processors. Finally, both Myriad 2 and Myriad X have a set of hardware filters called Streaming Image Processing Pipeline (SIPP)

The memory organization of the two systems is similar, as both have both DDR and DRAM. In addition, special mention should be made of the Connection Matrix Memory (CMX) which is used as NUMA ScratchPad memory and contributes significantly to the high performance of the system. The transfer of data from DDR to CMX, and vice versa, is done via a Direct Memory Access engine. In addition, LEON OS as well as LEON RT have L1 cache for both data and instructions as well as L2 cache. Finally, the SHAVEs each have a separate L1 cache and a shared L2 cache.

The main differences between the two models is the fact that Myriad X has a specialized processor to run Convolutional Neural Networks the Neural Compute Engine. In addition, the latest generation of Myriad has 16 SHAVEs as opposed to the earlier one which has only 12. Additionally, the size of both the CMX and the L1, L2 caches of LEON RT have been increased. Finally, the Myriad X according to Movidius can reach a performance of up to 1 trillion instructions per second[8].

Fault Tolerance

Fault Tolerance essentially means the ability of a system to continue operating even though one or more of its subsystems stop working. That is, fault tolerance, is how a system responds to malfunctions of hardware or software.

Fault Tolerant Computing

The goal of a Fault Tolerant system is to ensure continuous operation and high availability of the system by preventing interruptions that would result from a failure.

Fault Tolerant Computing may have various levels of fault tolerance:

- At the lowest level, the system can respond to power outages with alternative power sources.
- At a higher level, the ability to use a backup system with very short time delay.
- In the event of a disk failure, there is a backup disk containing all the data that can replace the main disk directly. In this way the system operation continues despite the partial failure instead of immediate destruction due to the loss of operation. -level Fault Tolerant systems consist of many processing units which detect the fault and immediately try to correct it.

Therefore, Fault-Tolerant systems ensure that there will be no interruption in their operation by having redundant subsystems to take the place of the original ones in case of a fault. Some of these may be:

- Hardware systems with their own or backup operating systems. Therefore, for there to be truly seamless system operation on such a system, all the information of the original system,

e.g., the scheduling of the various applications, must be present in the backup system and have a '1 to 1' copy of the corresponding data of the original system.The scheduling of the various applications must also be a '1 to 1' copy of the corresponding data of the original system.

- Software systems that have other Software systems in a previous state as backups.
- Backup power supplies that can help prevent system malfunction in the event of a power supply problem. This ensures that there will be no system crash.

Fault Injection

Fault Injection is a particularly widespread testing technique. Through this technique, a fault or pressure is injected into a system in order to observe the behavior of the system under these conditions.

Fault injection started as a fault simulation technique at the hardware level. Engineers exposed devices to various harmful conditions and observed how well they would continue to function. Some of the tests they did were to short-circuit pin devices, introduce electromagnetic interference, disrupt the system power supply, and even bombard the system circuits with radiation. The goal was to observe how these conditions affected the operation of the system and if the device stopped working, to redesign it.

Over time, engineers designed tools that allowed for error insertion by different methods. Consequently, devices began to have various, debugging ports such as JTAG that allowed the introduction of controlled error directly into the system's circuits. Then, methods for software-level error injection were designed in order to simulate error in their applications and control the various functions that handle errors and program exceptions. In order to achieve the above, engineers either changed the source code to introduce simulated errors (compile-time error) and trigger errors in systems already running (runtime error).

The usefulness of Fault Injection

Fault injection is one of the most basic forms of, testing as it helps the engineering team at the design stage to understand the behavior of their system under stressed or even global malfunction conditions. Furthermore, simulating this behavior allows engineers to be able to ensure a certain level of quality of service to consumers. The positive features of fault insertion are therefore diverse. The most basic ones are detailed below:

- **Engineers can extensively test their applications and systems**. Traditionally, software testing focuses on happy path testing. That is, it tests the paths of systems that engineers expect to receive. But it does not test the ways in which systems may deviate from the design team's expectations due to unexpected behavior, changes in operating conditions, errors in dependencies, or any other condition. Therefore, we want to make sure that the robust mechanisms placed in the systems will work as expected, and this is exactly what fault injection helps to do.
- **It gives the ability to systematically identify the nature and cause of various errors**. When a system fails, the technical team goes into reaction mode. Its main objective is to stop the problem and then to return the system to normal operation as soon as possible. Depending on the severity of the situation, it may take days or even weeks before there is a

clear response. Fault insertion therefore gives engineers total control over how much fault and when the fault is inserted into the system. Finally, it gives engineers the ability to reproduce the various problems that are created, thus allowing them to be effectively fixed.

- **Allows engineers to prepare for the unexpected**. The things that can go wrong in the context of production are too many. Moreover, even small mistakes can cause big losses. Therefore, error insertion allows controlling the behavior of the system in conditions that would not normally be expected such as memory errors, sudden increases in CPU usage and so on.This technique therefore allows the design team to prepare for the unexpected by adding mechanisms that will contribute to efficiently dealing with the problem that has arisen.

Related Work

Fault Tolerance in embedded systems

As soon as computing became mainstream, embedded systems have been commonly used. These embedded systems are essential in the proper operation of most of our day to day automated procedures, from transportation to stock trading. In recent years, with the appearance of edge computing and the discussions regarding industry 4.0 there is a new need to quickly process data in order to automate production. However, we do not only need to process said data as fast as possible we also need the to have high up-time and reliability. When referring to reliability we are referencing the ability to trust the correctness of the output of our systems. System reliability is essential in real time scenarios where embedded systems are deployed since a wrong output could result in catastrophic consequences, from huge sums of money being lost to death. Therefore, as modern computing evolves new programming paradigms need to be designed to assure system fault tolerance. However, fault tolerant design is not limited to embedded systems but is a general software design methodology commonly used in distributed systems as well. To provide an example of fault tolerance we will be examining the case study of the automobile. In cars, constantly sensors receive and process input, based on the output result the airbags or the abs system can be activated. However, these two subsystem are crucial for the safety of the passengers. Thus, we cannot risk wrong calculation of the acceleration of a car during a car crash since this could result in fatal consequences. Therefore, we need to design our software in a way that can guarantee us, as much as possible, the correctness of the output of the system.

Fault Injection in embedded systems

In the previous section we discussed the importance of fault tolerance in embedded computing. However, developers need to be able to asses whether their fault tolerant policies work properly. In order to achieve this they need to manual insert error in their code and check what will happen if not everything goes as planned. Therefore, we see the need to find out ways to manually insert error in a system to stress test it. Currently, there are two possible approaches: First of all, the engineer can insert error in the hardware level. This refers to, exposing the embedded system to harmful environmental conditions such as electromagnetic radiation, short circuiting part of the system e.t.c. These however, destroy the system. In order to avoid system destruction in testing developers can insert error in the software layer. In this case they manually edit the contents of

the memory or flip the bits of variables to check how the system will react. These two methods help the developers test the system tolerance to error.

Thesis Scope

As industry 4.0 emerges the usage of off the shelf embedded systems steadily increases. However, as these systems are deployed in time and error critical scenarios we need to be able to verify the correctness of the system output. In the past few years a new subcategory of embedded systems has emerged named vision processing units. This thesis focuses on one of the most power efficient vision processing unit called Intel Movidius Myriad 2. Since, these platforms are new there has not been much effort to port commonly used fault tolerant policies that can be easily developed, or just used via an existing library. Therefore, our goal is to port some of these commonly used techniques to the Myriad 2 platform. To be more precise we developed multiple fault tolerant policies that correct erroneous content in multiple scenarios such as: the SHAVEs receive wrong data from LEON OS, SHAVE get restored after ruining their own data, LEON OS reschedules data of broken shaves to working ones e.t.c. All of the above methods have one thing in common we use cyclic redundancy checks to verify whether there is any error in the system. Furthermore, we developed two types of voting systems as a method to correct the output of the system in the case of error. These systems, are commonly used in distributed systems. Last but not least, we developed multiple methods that are used to insert error in the platform on the software level. These methods can be then used by dev teams to speed up the testing of their products accelerating their time to market.

Chapter 2

Development of Fault Injection Techniques on Myriad2

Introduction

Fault injection testing is a type of software testing that intentionally introduces defects into a system to ensure that it can withstand and recover from them. Fault injection testing is commonly performed prior to deployment to identify any potential flaws introduced during production. Fault injection testing, like stress testing, is intended to find specific flaws in a hardware or software system so that they can be rectified or avoided.

Fault injection testing in software can be done either at compile time or during runtime. Compile-time injection is a testing technique that involves changing the source code to simulate software system flaws. Modifications or mutations to existing code, such as changing a line of code to reflect a different value, can be used to accomplish these changes. Testers can also modify code by adding or inserting new code, such as additional logic values.

A software trigger will be used to start injecting a fault into the software while it is operating. A time-based trigger is one that can be set to inject a fault at a specific moment. Trap mechanisms, which interrupt software at a certain position in the code or event in the system, can also be used to set triggers. An interrupt-based trigger is what this is called.

Apart from software based error injection we also have hardware level fault injection. In this case, instead of using a software trigger we introduce defects in the hardware level of the system. Fore example, we introduce electromagnetic transmissions that might result in bit flips in the system memory. However, this method is not easily used since it could easily result in heavy system damage or system destruction.

Other forms of testing, like as chaos engineering, are easily similar to fault injection. Fault injection, on the other hand, is distinct in that it necessitates a unique approach to test a single condition. Fault injection testing can also be used on hardware, as it simulates hardware failures such shorted circuit board connections. Fault injection testing has the following advantages:

- Increased software durability.
- Allows developers to preview the effects of defects or problems before they appear in production.
- Allows developers to fix previously unknown issues before they are released.

Fault injection can be done manually without the use of any tools; however, tools can be used to assist in the automation of the process.

Library-level Fault Injector (LFI) [13], for example, will automatically detect defects and inject faults between libraries and applications. Another automatic fault injection tool is Fault Tolerance and Performance Evaluator (FTAPE)[14], which allows users to inject errors into memory and disk access. The Xception tool[15] can also aid automate the use of software triggers, which are used to store problems in memory.

In this thesis, since there is no already developed tool, we have developed multiple techniques which introduce error in the Myriad 2 platform on a software level. In this chapter we will be examining the developed methods in depth.

Method 1: Transfer of corrupted data from the processor LEON OS

Below we can see the general programming paradigm of Intel Myriad 2:

1. The LEON OS receives the input data.
2. Transfer data from LOS to Shaves.
3. Processing the data in the Shaves.
4. Return data back to LOS after processing is complete.

The above steps are repeated until the input data processing is complete. Therefore, in the above steps we also see one of the error insertion methods implemented in this section. More specifically in the context of transferring corrupted input data from the LOS corrupted data is transferred from the Shaves for processing. Therefore, the Shaves process incorrect data and as expected return incorrect results. The above process for error insertion is schematically described in the diagram below:

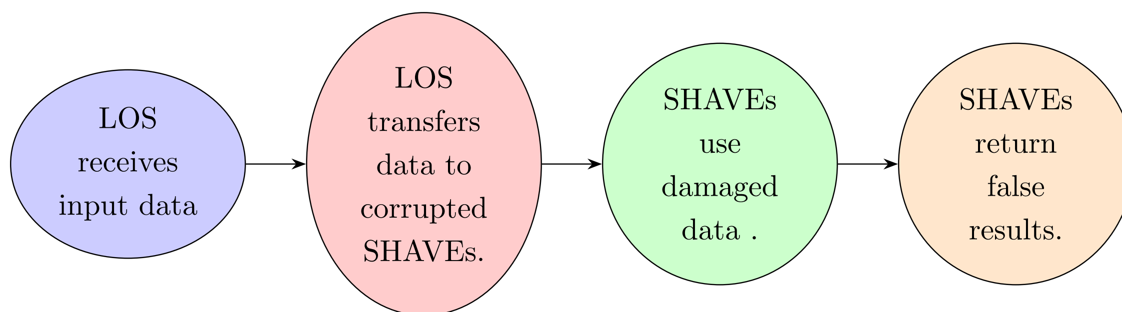


Figure 2.1: Fault Injection Method: LOS transfers corrupted data to Shave.

Down below we can see the way these corrupted data get transferred to the SHAVE co-processors in our code base:

```
. . .
. . .
. . .
int flag = 0;
for (int i = 0; i < SHAVES_USED; i++) {
    flag =0;
    for (int j=0; j<RAND_COUNT;j++){
        if(broken_shaves[j]==i){
            flag =1;
        }
    }
    if(flag==1){
        shv_inFrame1[i] = (u8 *) &corrupted_inputFrame1024[0];
        shv_inFrame2[i] = (u8 *) &corrupted_inputFrame1024[0];
    }
}
. . .
. . .
. . .
for (i = 0; i < SHAVES_USED; i++) {
    threadArgs[i].inFrame1 = shv_inFrame1[i];
    threadArgs[i].outFrame1 = shv_outFrame1[i];
    threadArgs[i].inFrame2 = shv_inFrame2[i];
    threadArgs[i].outFrame2 = shv_outFrame2[i];
    threadArgs[i].shave_id = (u8)i;
    threadArgs[i].is_broken = false;
    threadArgs[i].shave_realized_internal_error = false;
    for(int j=0; j<RAND_COUNT; j++){
        if(i == broken_shaves[j]){
            if(threadArgs[i].is_broken == false){
                threadArgs[i].is_broken = true;
                printf("Shave Number %d is damaged \n",i);
            }
        }
    }
}
```

Inspecting the above code we can observe two things. We can see that the error injection method is split into two discrete sections:

1. First of all we have an initialization section that depending on the status of the SHAVE will forward the same, erroneous, input data to all broken SHAVEs.

2. Secondly, after the SHAVE data initialization is complete we can see that we pass a flag that notifies us about whether a SHAVE is broken or not. This is included as a verification system in order to conclude about the working status of the SHAVE after the program execution.

3. Apart from the arguments mentioned above we forward a flag called shave_realised_internal_error. This flag is used for the SHAVEs to figure out whether there is something wrong with their data on their own.

Method 2: Input data corruption from SHAVEs

Following the basic programming assumption mentioned in the previous section, i.e., transferring input data from the various system peripherals to LEON OS and from LOS to SHAVEs, we observe a new case of error insertion. What happens if the input data is correctly transferred to SHAVEs from LOS but the SHAVEs are compromised? In this case corrupted SHAVEs can start corrupting the input data. We simulate this behavior with this error insertion technique.

Below we show and illustrate this technique:

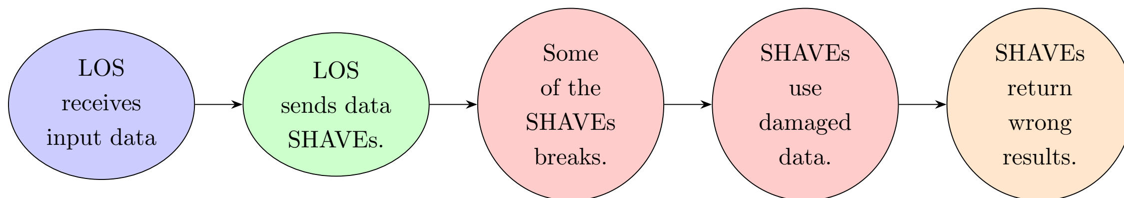


Figure 2.2: Fault Injection Method: SHAVEs corrupt their own data.

As always we will also be displaying the code that recreates the above flow diagram in our software system.

First of all, we will be displaying the code that is executed within LEON OS.

```
. . .
. . .
. . .
for (int i = 0; i < SHAVES_USED; i++) {
    shv_inFrame1[i] = (u8 *)(&inputFrame[FRAME_WIDTH TASK_FRAME_HEIGHT * i]);
    shv_outFrame1[i] = (u8 *)(&outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i
]);
    shv_inFrame2[i] = (u8 *)(&inputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i +
FRAME_WIDTH * FRAME_HEIGHT / 2]);
    shv_outFrame2[i] = (u8 *)(&outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i
+ FRAME_WIDTH * FRAME_HEIGHT / 2]);
    }
. . .
. . .
. . .
for (i = 0; i < SHAVES_USED; i++) {
    threadArgs[i].inFrame1 = shv_inFrame1[i];
    threadArgs[i].backup_inFrame1 = shv_inFrame1[i];
    threadArgs[i].outFrame1 = shv_outFrame1[i];
    threadArgs[i].inFrame2 = shv_inFrame2[i];
    threadArgs[i].backup_inFrame2 = shv_inFrame2[i];
    threadArgs[i].outFrame2 = shv_outFrame2[i];
    threadArgs[i].shave_id = (u8)i;
    threadArgs[i].is_broken = false;
    threadArgs[i].array_of_realization_flags = & realization_flags[i];
    for(int j=0; j<RAND_COUNT; j++){
        if(i == broken_shaves[j]){
            if(threadArgs[i].is_broken == false){
                threadArgs[i].is_broken = true;
                printf("Shave Number %d is damaged \n",i);
            }
        }
    }
}
```

Accordingly, the error injection code executed within the SHAVES.

```
. . .
. . .
. . .
if(thArgs->is_broken==true){
    extern u8 corrupted_inputFrame1024[];
    inFrame1 = &corrupted_inputFrame1024[0];
    inFrame2 = &corrupted_inputFrame1024[0];
}
. . .
. . .
. . .
```

Inspecting the above code snippets we can deduce that:

1. In this case LEON OS schedules all the input data correctly.
2. Apart from the input frame and output frames we also send a backup frame to the SHAVEs. This is not yet relevant, but will be of importance later in the fault tolerant policies.
3. The error injection process is: Once a SHAVE realizes he is "broken" it uses as input data a corrupted input frame that was provided at system initialization.

Method 3: LEON OS corrupts variables shared with SHAVEs.

In multicore programming it is quite common to have common variables between different computing cores. Similarly in the Myriad 2 platform it is possible to have shared variables between SHAVEs and LEON OS. Therefore, there is a possibility that in such a programming assumption, a failure in LOS could lead to errors in the SHAVEs. Therefore, in this technique we have appropriately configured the code so that a change due to a simulated failure in LOS will cause SHAVEs to malfunction in real-time.

The previous technique is shown below in steps:

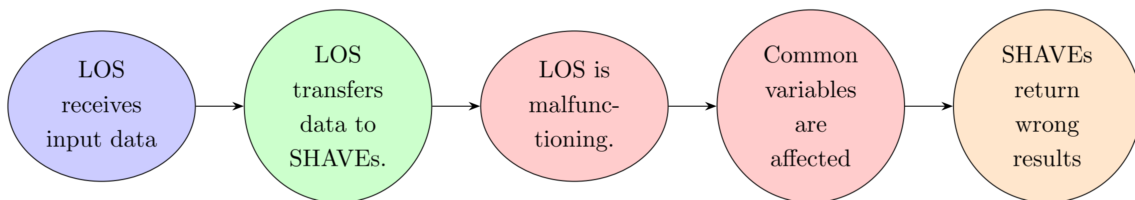


Figure 2.3: Fault Injection Method: LOS corrupts shared variable

As always we will also be displaying the code that recreates the above flow diagram in our software system.

```
1    . . .
2    . . .
3    for (int i = 0; i < SHAVES_USED; i++) {
4        shv_inFrame1[i] = (u8 *)(&inputFrame[FRAME_WIDTH TASK_FRAME_HEIGHT * i]);
5        shv_outFrame1[i] = (u8 *)(&outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i
     ]);
6        shv_inFrame2[i] = (u8 *)(&inputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i +
     FRAME_WIDTH * FRAME_HEIGHT / 2]);
7        shv_outFrame2[i] = (u8 *)(&outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i
     + FRAME_WIDTH * FRAME_HEIGHT / 2]);
8        }
9    . . .
10   . . .
11   for (i = 0; i < SHAVES_USED; i++) {
12               threadArgs[i].inFrame1 = shv_inFrame1[i];
13               threadArgs[i].backup_inFrame1 = shv_inFrame1[i];
14               threadArgs[i].outFrame1 = shv_outFrame1[i];
15               threadArgs[i].inFrame2 = shv_inFrame2[i];
16               threadArgs[i].backup_inFrame2 = shv_inFrame2[i];
17               threadArgs[i].outFrame2 = shv_outFrame2[i];
18               threadArgs[i].shave_id = (u8)i;
19               threadArgs[i].is_broken = false;
20               threadArgs[i].shave_realized_internal_error = false;
21                   for(int j=0; j<RAND_COUNT; j++){
22                       if(i == broken_shaves[j]){
23                           if(threadArgs[i].is_broken == false){
24                               threadArgs[i].is_broken = true;
25                               printf("Shave Number %d is damaged \n",i);
26                       }
27                   }
28               }
29   }
30   for (int l =0; l<RAND_COUNT; l++){
31       switch (broken_shaves[l])
32       {
33       case 0:
34           Example0_inFrame1 = (u8*) (&corrupted_inputFrame1024[FRAME_WIDTH *
     TASK_FRAME_HEIGHT * i]);
35           Example0_inFrame2 = (u8 *) (&corrupted_inputFrame1024[FRAME_WIDTH *
     TASK_FRAME_HEIGHT * i]);
36           break;
37       case 1:
38           Example1_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
     TASK_FRAME_HEIGHT * i]);
39           Example1_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
     TASK_FRAME_HEIGHT * i]);
40           break;
41       case 2:
42           Example2_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
     TASK_FRAME_HEIGHT * i]);
43           Example2_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
     TASK_FRAME_HEIGHT * i]);
44           break;
45       case 3:
46           Example3_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
     TASK_FRAME_HEIGHT * i]);
```

```
47             Example3_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
48             break;
49         case 4:
50             Example4_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
51             Example4_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
52             break;
53         case 5:
54             Example5_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
55             Example5_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
56             break;
57         case 6:
58             Example6_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
59             Example6_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
60             break;
61         case 7:
62             Example7_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
63             Example7_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
64             break;
65         case 8:
66             Example8_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
67             Example8_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
68             break;
69         case 9:
70             Example9_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
71             Example9_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
72             break;
73         case 10:
74             Example10_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
75             Example10_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
76             break;
77         case 11:
78             Example11_inFrame1 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
79             Example11_inFrame2 = (u8 *)(&corrupted_inputFrame1024[FRAME_WIDTH *
    TASK_FRAME_HEIGHT * i]);
80             break;
81         default:
82             break;
83         }
84     }
```

Inspecting the above code snippets we can deduce that:

1. In this case LEON OS schedules all the input data correctly.
2. Based on whether a SHAVE is broken or not then a corrupted input frame is used as input.
3. In this case we do not only provide the ”standard” input frames we also provide a backup input frame. This is not related to the error injection method. However, it will be used later on to recover from system failure.
4. The error injection process happens by changing the different input frames that each SHAVE has.

Method 4: Fault Injection in instruction memory

One of the key features of the Myriad 2 platform is the fact that it has a unified memory space. This means that regions of DDR and CMX are accessible from all computing units in the system. It should be noted, however, that memory access is not uniform which may cause race conditions. Due to the above properties of Myriad 2, a malfunctioning part may corrupt data of other subsystems. Finally, the code of Shaves is stored in specific memory locations specified by the developer, thus even the code run by SHAVEs can be corrupted.

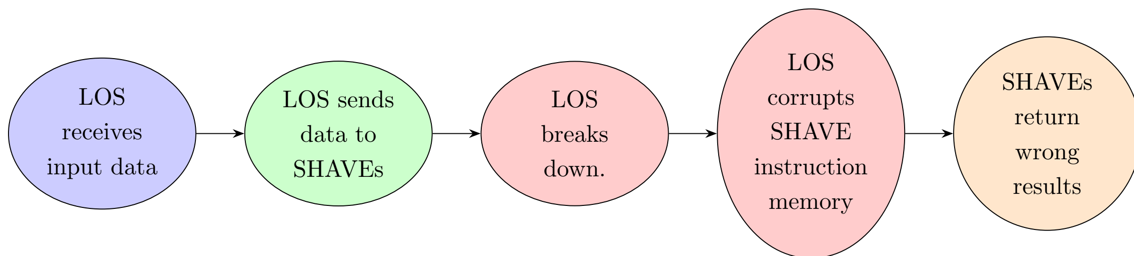


Figure 2.4: Fault Injection Method: LOS corrupts shared memory space

The code used to simulate the above flow chart can be viewed down below:

```
    . . .
    . . .
    int * code_shave_0  = (int *) 0x80001AAB;
    int * code_shave_1  = (int *) 0x80021AAB;
    int * code_shave_2  = (int *) 0x80041AAB;
    int * code_shave_3  = (int *) 0x80061AAB;
    int * code_shave_4  = (int *) 0x80081AAB;
    int * code_shave_5  = (int *) 0x800A1AAB;
    int * code_shave_6  = (int *) 0x800C1AAB;
    int * code_shave_7  = (int *) 0x800E1AAB;
    int * code_shave_8  = (int *) 0x80101AAB;
    int * code_shave_9  = (int *) 0x80121AAB;
    int * code_shave_10 = (int *) 0x80141AAB;
    int * code_shave_11 = (int *) 0x80161AAB;
    . . .
    . . .
    for (int i = 0; i < SHAVES_USED; i++) {
        shv_inFrame1[i] = (u8 *)(&inputFrame[FRAME_WIDTH TASK_FRAME_HEIGHT * i]);
        shv_outFrame1[i] = (u8 *)(&outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i
    ]);
```

```
20      shv_inFrame2[i] = (u8 *)(&inputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i +
    FRAME_WIDTH * FRAME_HEIGHT / 2]);
21      shv_outFrame2[i] = (u8 *)(&outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i
    + FRAME_WIDTH * FRAME_HEIGHT / 2]);
22      }
23  . . .
24  . . .
25  for (i = 0; i < SHAVES_USED; i++) {
26              threadArgs[i].inFrame1 = shv_inFrame1[i];
27              threadArgs[i].outFrame1 = shv_outFrame1[i];
28              threadArgs[i].inFrame2 = shv_inFrame2[i];
29              threadArgs[i].outFrame2 = shv_outFrame2[i];
30              threadArgs[i].shave_id = (u8)i;
31              threadArgs[i].is_broken = false;
32              threadArgs[i].shave_realized_internal_error = false;
33                  for(int j=0; j<RAND_COUNT; j++){
34                      if(i == broken_shaves[j]){
35                          if(threadArgs[i].is_broken == false){
36                              threadArgs[i].is_broken = true;
37                              printf("Shave Number %d is damaged \n",i);
38                      }
39                  }
40              }
41  }
42  . . .
43  . . .
44  for (int l =0; l<RAND_COUNT; l++){
45      switch (broken_shaves[l]){
46      case 0:
47          memset(code_shave_0, 0, CODE_HARM_BYTES* sizeof(char));
48          break;
49      case 1:
50          memset(code_shave_1, 0, CODE_HARM_BYTES* sizeof(char));
51          break;
52      case 2:
53          memset(code_shave_2, 0, CODE_HARM_BYTES* sizeof(char));
54          break;
55      case 3:
56          memset(code_shave_3, 0, CODE_HARM_BYTES* sizeof(char));
57          break;
58      case 4:
59          memset(code_shave_4, 0, CODE_HARM_BYTES* sizeof(char));
60          break;
61      case 5:
62          memset(code_shave_5, 0, CODE_HARM_BYTES* sizeof(char));
63          break;
64      case 6:
65          memset(code_shave_6, 0, CODE_HARM_BYTES* sizeof(char));
66      case 7:
67          memset(code_shave_7, 0, CODE_HARM_BYTES* sizeof(char));
68          break;
69      case 8:
70          memset(code_shave_8, 0, CODE_HARM_BYTES* sizeof(char));
71          break;
72      case 9:
73          memset(code_shave_9, 0, CODE_HARM_BYTES* sizeof(char));
74          break;
75      case 10:
```

```
76              memset(code_shave_10, 0, CODE_HARM_BYTES* sizeof(char));
77              break;
78          case 11:
79              memset(code_shave_11, 0, CODE_HARM_BYTES* sizeof(char));
80              break;
81          default:
82              break;
83          }
84      }
```

Inspecting the above code snippets we can deduce that:

1. In this case LEON OS schedules all the input data correctly.
2. Based on whether a SHAVE is broken or not we manually change the contents of its memory setting everything to 0.
3. The error injection process happens by changing the contents of shave memory.
4. The number of bytes we decided to harm have been chosen arbitrarily but are the same for all the SHAVEs.

Chapter 3

Development of Fault-Tolerant Architectures on Myriad2

Introduction

By preventing disruptions caused by a single point of failure, fault tolerant computer systems ensure business continuity and high availability. As a result, fault tolerance solutions are typically focused on mission-critical applications or systems. There are multiple levels of fault tolerance in fault tolerant computing: At the most basic level, consider the capacity to respond to a power outage. Multiprocessor fault tolerant computing: many processors work together to examine input and output for mistakes and rectify them promptly. Fault tolerance software may be included in the OS interface, allowing programmers to double-check vital data at key moments during a transaction. Fault-tolerant systems prevent service interruptions by automatically replacing faulty components with backup components.

Some examples are:

- Hardware with backup operating systems that are identical or comparable. A fault tolerant server, for example, is one that has an identical fault tolerant server mirroring all operations in backup and running in parallel.
- Hardware fault tolerance in the form of redundancy may make any component or system significantly safer and more reliable by removing single points of failure. Software systems are backed up by other software instances. If you continually replicate your customer database, for example, processes in the primary database can be diverted to the secondary database if the first fails.
- Alternative power sources that can take over automatically during power outages can assist avoid a system problem and ensure that no service is lost.

The most significant part of this thesis revolves around designing and implementing multiple fault tolerant architectures. These architectures will be thoroughly explained in the following sections of this chapter.

A key step in troubleshooting and repairing a fault is to detect the fault in the system. This detection must occur with absolute accuracy as it is the key weapon in the design of a fault tolerant system. In this thesis, a fairly widespread technique for fault detection called Cyclic Redundancy Check[11] was used.

Fault tolerant architectures that rely on C.R.C

Error detection with Cyclic Redundancy Check

The Cyclic Redundancy Check (C.R.C) is an error detection code often used in digital networks and storage devices to detect random changes in data. Blocks of data entering systems have a check value based on the polynomial division of their contents. During retrieval the calculation is repeated and if the check value has changed then there is a change in the stored data. The reason that C.R.C is so widespread is that they are easy to implement in hardware, easy to analyze mathematically, and easy to detect common errors due to noise. As their output is of a certain length, C.R.C are often used as hash functions[12]. As mentioned earlier the main reason for the high performance of Myriad 2 is due to the fast VLIW SHAVE co-processors. The general programming method of this platform is based on the following steps:

1. The LEON OS receives the input data.

2. Transfer data from LOS to SHAVEs.

3. Processing the data in SHAVEs.

4. Return data back to LOS after processing is complete.

The above steps are repeated until the input data processing is complete.

Therefore, in order to detect changes in the data, we must pass the input data through a C.R.C function and at regular intervals, or at least before processing it, pass it again to check if it has been altered. The following shows the error detection process using Cyclic Redundancy Check.

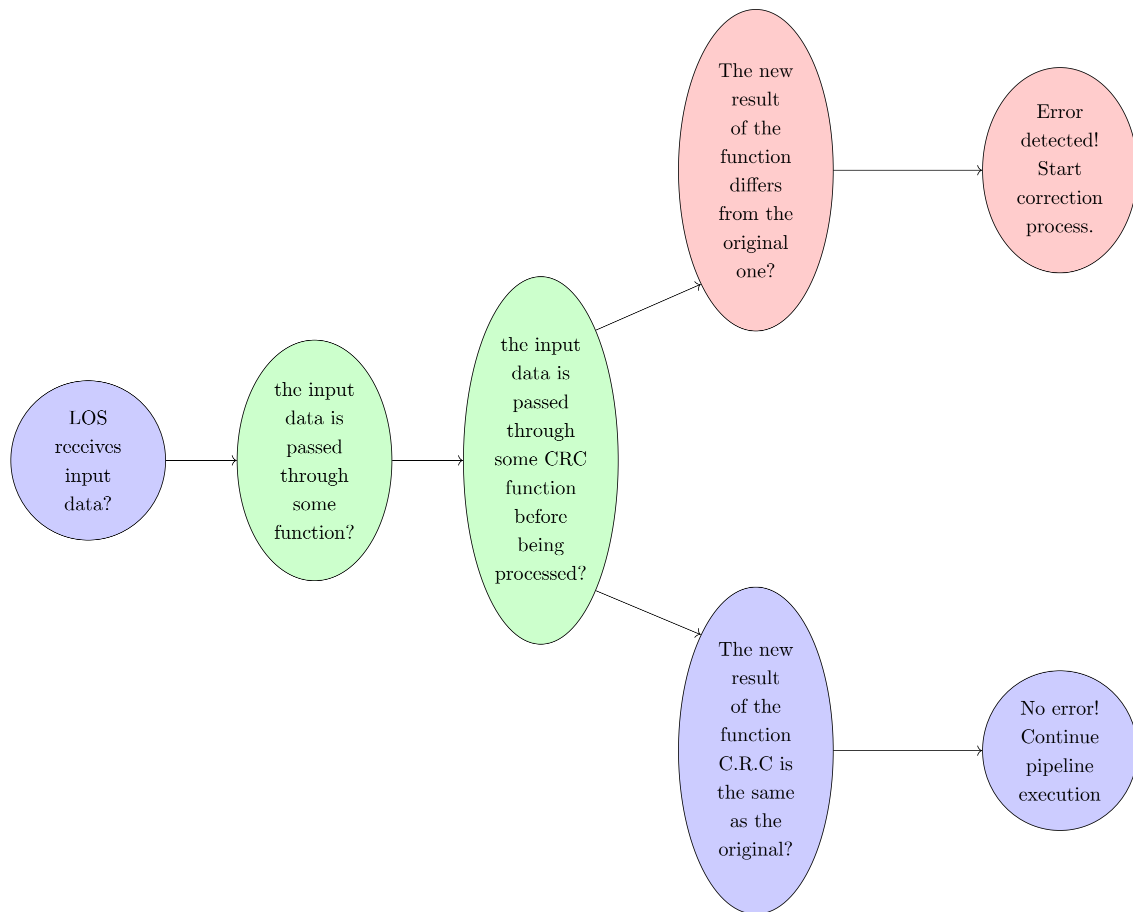


Figure 3.1: Fault Detection: using CRC to detect data corruption

Fault tolerant architecture 1: LEON OS transmits broken data to SHAVEs

In a previous section we mentioned that LEONOS is the main orchestrator of the Myriad operation as it is responsible for transferring data to Shaves at various stages of the programming pipeline. Therefore, it makes special sense to perform various checks on the data as it is transferred to the various co-processors. In this particular policy, LOS passes through a C.R.C function the input data as soon as it receives it from a peripheral. Also, after passing this data through the C.R.C. function it creates a temporary backup of this data. Then, before sending them to the SHAVEs it passes them again through a C.R.C. function and checks if the return value of the function will be different. If the value matches the previous one then it passes the data to SHAVEs normally. If, the value is different then, LOS has suffered some kind of failure and we cannot trust the data it is trying to transfer to SHAVEs. Therefore, it routes the data it knows is correct to the SHAVEs and then restores the corrupted data to its original state via the backup it had created once it received it and routes that data back to the SHAVEs. Below is a flowchart of the above process in order to fully understand it:

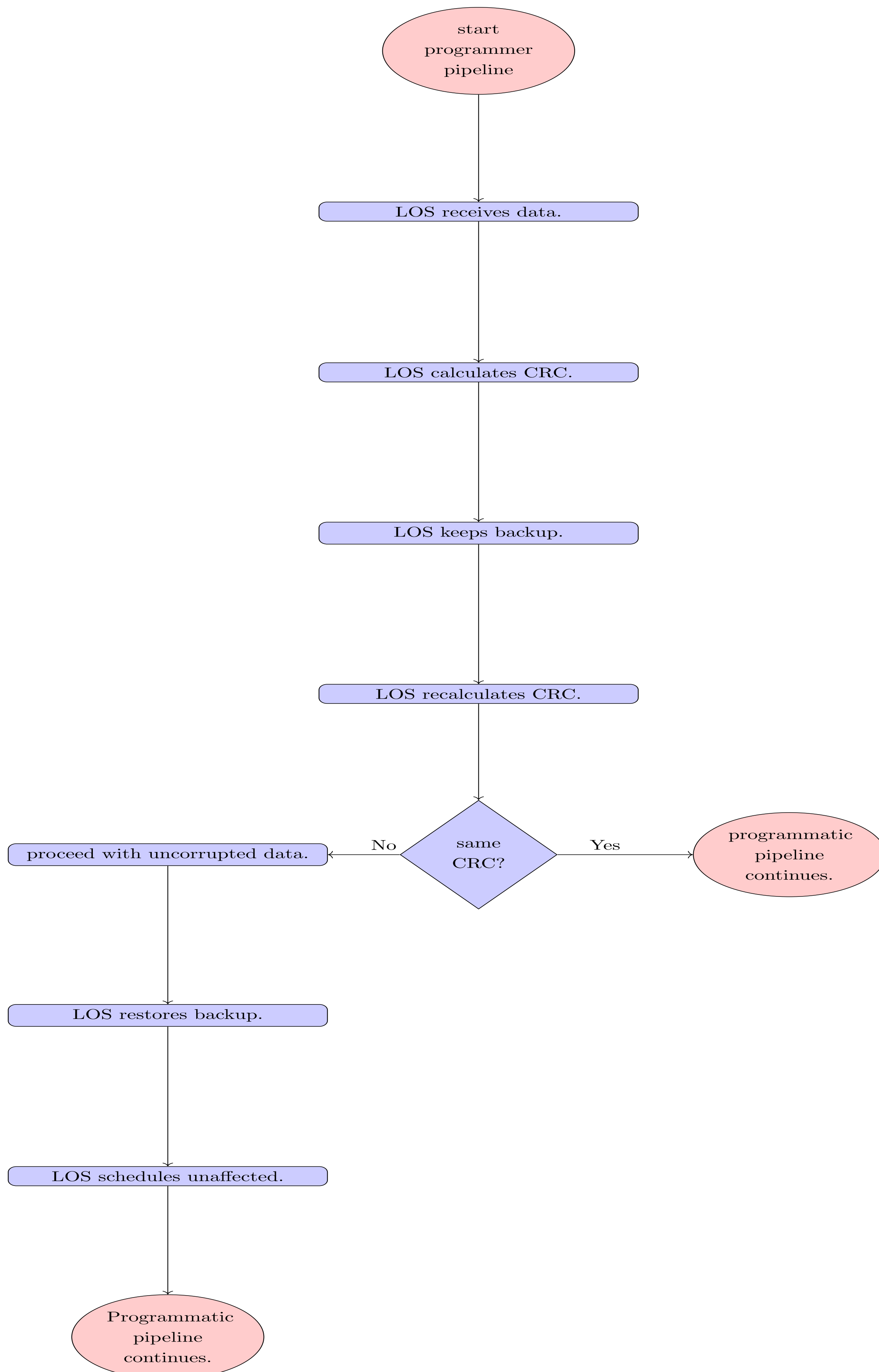


Figure 3.2: Fault Tolerant Policy 1: Using C.R.C to detect data corruption LOS reschedules Shave data

The code that assists in the execution of the above flow chart can be seen down bellow:

```
    . . .
    . . .
    . . .
u8 broken_shaves_number=0;
    for (int i=0; i< SHAVES_USED;i++){
        sc = OsDrvTimerGetSystemTicks64(&hashing_start_ticks_0);
        array_of_crcs_after_mistake_insertion[i] = swcCalcCrc32(shv_inFrame1[i],
    sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT,be_pointer);
        sc = OsDrvTimerGetSystemTicks64(&hashing_end_ticks_0);
        if(array_of_crcs[i]!= array_of_crcs_after_mistake_insertion[i]){
                    los_broken_shaves[broken_shaves_number] = i;
                    broken_shaves_number++;
        }
    }

sc = OsDrvTimerGetSystemTicks64(&scheduling_startTicks);
if (sc == MYR_DRV_NOT_INITIALIZED)
    exit(sc);
// Resceduler
// Find number of working shaves and store them in an array
number_of_working_shaves = 0;
for(i=0; i<SHAVES_USED; i++){
    bool working_flag = true;
    for(int j=0; j<broken_shaves_number;j++){
        if(i==los_broken_shaves[j]){
            working_flag = false;
        }
    }
    if(working_flag && (broken_shaves_number != SHAVES_USED)){
        los_working_shaves[number_of_working_shaves] = i;
        number_of_working_shaves++;
    }
}

u8 idx = 0;

sc = OsDrvTimerGetSystemTicks64(&second_run_startTicks);
    if (sc == MYR_DRV_NOT_INITIALIZED)
        exit(sc);

//Case were number of broken shaves is equal to or less than the snumber of working
     shaves
if(number_of_working_shaves == 0){
    for (int i = 0; i < broken_shaves_number; i++) {
            threadArgs[i].inFrame1 =
                (u8 *)(&inputFrame1024[FRAME_WIDTH * TASK_FRAME_HEIGHT * i]);
            threadArgs[i].inFrame2 =
                (u8 *)(&inputFrame1024[FRAME_WIDTH * TASK_FRAME_HEIGHT * i +
                                        FRAME_WIDTH * FRAME_HEIGHT / 2]);
            threadArgs[i].outFrame1 =
                (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i]);
            threadArgs[i].outFrame2 =
                (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT * i +
                                FRAME_WIDTH * FRAME_HEIGHT / 2]);
        }

    for (u32 i = 0; i < broken_shaves_number; i++) {
```

```
56
57          sc = OsDrvSvuOpenShave(handlers+i, i, OS_MYR_PROTECTION_SEM);
58          if (sc == OS_MYR_DRV_SUCCESS) {
59              sc = OsDrvSvuResetShave(handlers+i);
60              if (sc)
61                  exit(sc);
62              sc = OsDrvSvuSetAbsoluteDefaultStack(handlers+i);
63              if (sc)
64                  exit(sc);
65
66              sc = OsDrvSvuStartShaveCC(handlers+i, (u32)entryPoints[i], "i",
67                                              (void *)&threadArgs[i]);
68
69              if (sc)
70                  exit(sc);
71          } else {
72              printf("Error cannot open shave %lu\n", i);
73              exit(sc);
74          }
75      }
76
77      for (u32 i = 0; i < broken_shaves_number; i++) {
78       sc = OsDrvSvuWaitShaves(1, handlers+i,1000,
79                                  &running);
80          if (sc)
81              exit(sc);
82          sc = OsDrvSvuCloseShave(handlers+i);
83          if (sc)
84              exit(sc);
85      }
86  }
87  else if(number_of_working_shaves >= broken_shaves_number){
88      idx = 0 ;
89      sc = OsDrvTimerGetSystemTicks64(&scheduling_startTicks);
90      if (sc == MYR_DRV_NOT_INITIALIZED)
91          exit(sc);
92
93      for(i=0; i<number_of_working_shaves; i++){
94          while(idx<broken_shaves_number){
95              threadArgs[los_working_shaves[i]].inFrame1 = (u8 *)(&inputFrame1024[
        FRAME_WIDTH * TASK_FRAME_HEIGHT * los_broken_shaves[idx]]);
96              threadArgs[los_working_shaves[i]].inFrame2 = (u8 *)(&inputFrame1024[
        FRAME_WIDTH * TASK_FRAME_HEIGHT * los_broken_shaves[idx] + FRAME_WIDTH *
        FRAME_HEIGHT / 2]);
97              threadArgs[los_working_shaves[i]].outFrame1 =
98                  (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT *
        los_broken_shaves[idx]]);
99              threadArgs[los_working_shaves[i]].outFrame2 =
100                 (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT *
        los_broken_shaves[idx] +
101                                     FRAME_WIDTH * FRAME_HEIGHT / 2]);
102             idx++;
103             break;
104         }
105     }
106
107     sc = OsDrvTimerGetSystemTicks64(&scheduling_endTicks);
108     if (sc == MYR_DRV_NOT_INITIALIZED)
```

```
109         exit(sc);
110     // Rescedule new shaves
111     // Working shaves - broken shaves is the list of working shaves used inside
    theh array
112     // Rescedule the shaves found in the los_working_shaves_array and wait
113     temp_system_ticks = temp_system_ticks +  scheduling_endTicks -
    scheduling_startTicks ;
114 
115     for (u32 i = 0; i < number_of_working_shaves; i++) {
116         sc = OsDrvSvuOpenShave(handlers+(los_working_shaves[i]),los_working_shaves[
    i],OS_MYR_PROTECTION_SEM);
117         if(sc == OS_MYR_DRV_SUCCESS){
118         sc = OsDrvSvuResetShave(handlers+(los_working_shaves[i]));
119         if (sc)
120             exit(sc);
121         sc = OsDrvSvuSetAbsoluteDefaultStack(handlers+(los_working_shaves[i]));
122         if (sc)
123             exit(sc);
124         sc = OsDrvSvuStartShaveCC(handlers+(los_working_shaves[i]),
125                                   (u32)entryPoints[(los_working_shaves[i])], "i",
126                                   (void *)&threadArgs[(los_working_shaves[i])]);
127 
128         if (sc)
129             exit(sc);
130         }
131         else {
132         printf("Error cannot open shave %d\n", (los_working_shaves[i]));
133         exit(sc);
134         }
135     }
136 
137     for (u32 i = 0; i < number_of_working_shaves; i++) {
138         sc = OsDrvSvuWaitShaves(1, handlers+(los_working_shaves[i]),1000,
139                                 &running);
140         if (sc)
141             exit(sc);
142         sc = OsDrvSvuCloseShave(handlers+(los_working_shaves[i]));
143         if (sc)
144             exit(sc);
145     }
146     // Rescedule new shaves
147     // Working shaves - broken shaves is the list of working shaves used inside the
     array
148     // Rescedule the shaves found in the los_working_shaves_array and wait
149 }
150 
151 // Case were more shaves are broken than working
152 else if (number_of_working_shaves < broken_shaves_number){
153     idx = 0;
154     u8 broken_shaves_remaining = broken_shaves_number;
155     while(broken_shaves_remaining > number_of_working_shaves){
156         sc = OsDrvTimerGetSystemTicks64(&scheduling_startTicks);
157         for(i=0; i<number_of_working_shaves; i++){
158             while(idx<broken_shaves_number){
159             threadArgs[los_working_shaves[i]].inFrame1 = (u8 *)(&inputFrame1024[
    FRAME_WIDTH * TASK_FRAME_HEIGHT * los_broken_shaves[idx]]);
160             threadArgs[los_working_shaves[i]].inFrame2 = (u8 *)(&inputFrame1024[
    FRAME_WIDTH * TASK_FRAME_HEIGHT * los_broken_shaves[idx] + FRAME_WIDTH *
```

```
FRAME_HEIGHT / 2]);
161             threadArgs[los_working_shaves[i]].outFrame1 =
162                 (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT *
los_broken_shaves[idx]]);
163             threadArgs[los_working_shaves[i]].outFrame2 =
164                 (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT *
los_broken_shaves[idx] +
165                                 FRAME_WIDTH * FRAME_HEIGHT / 2]);
166                 idx++;
167                 break;
168             }
169         }
170     broken_shaves_remaining = broken_shaves_remaining - number_of_working_shaves;
171     sc = OsDrvTimerGetSystemTicks64(&scheduling_endTicks);
172     temp_system_ticks = temp_system_ticks + (scheduling_endTicks -
scheduling_startTicks);
173
174
175     // Rescedule new shaves
176     // Working shaves - broken shaves is the list of working shaves used inside
theh array
177     // Rescedule the shaves found in the los_working_shaves_array and wait
178         for (u32 i = 0; i < number_of_working_shaves; i++) {
179             sc = OsDrvSvuOpenShave(handlers+(los_working_shaves[i]),
los_working_shaves[i],OS_MYR_PROTECTION_SEM);
180             if(sc == OS_MYR_DRV_SUCCESS){
181                 sc = OsDrvSvuResetShave(handlers+(los_working_shaves[i]));
182                 if (sc)
183                     exit(sc);
184                 sc = OsDrvSvuSetAbsoluteDefaultStack(handlers+(los_working_shaves[i
]));
185                 if (sc)
186                     exit(sc);
187                 sc = OsDrvSvuStartShaveCC(handlers+(los_working_shaves[i]),
188                                     (u32)entryPoints[(los_working_shaves[i])], "i"
,
189                                     (void *)&threadArgs[(los_working_shaves[i])]);
190                 if (sc)
191                     exit(sc);
192         } else{
193             printf("Error cannot open shave %d\n", (los_working_shaves[i]));
194             exit(sc);
195             }
196         }
197
198         for (u32 i = 0; i < number_of_working_shaves; i++) {
199             sc = OsDrvSvuWaitShaves(1, handlers+(los_working_shaves[i]),1000,
200                                     &running);
201             if (sc)
202                 exit(sc);
203             sc = OsDrvSvuCloseShave(handlers+(los_working_shaves[i]));
204             if (sc)
205                 exit(sc);
206             }
207         }
208
209         // We are finally to a lessser number of broken shaves than working shaves
210         // Rescedule Final  shaves
```

```
211         // Working shaves - broken shaves is the list of working shaves used inside
     theh array
212         // Rescedule the shaves found in the los_working_shaves_array and wait
213         sc = OsDrvTimerGetSystemTicks64(&scheduling_startTicks);
214         for(i=0; i<number_of_working_shaves; i++){
215             while(idx<broken_shaves_number){
216             threadArgs[los_working_shaves[i]].inFrame1 = (u8 *)(&inputFrame1024[
    FRAME_WIDTH * TASK_FRAME_HEIGHT * los_broken_shaves[idx]]);
217             threadArgs[los_working_shaves[i]].inFrame2 = (u8 *)(&inputFrame1024[
    FRAME_WIDTH * TASK_FRAME_HEIGHT * los_broken_shaves[idx] + FRAME_WIDTH *
    FRAME_HEIGHT / 2]);
218             threadArgs[los_working_shaves[i]].outFrame1 =
219                 (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT *
    los_broken_shaves[idx]]);
220             threadArgs[los_working_shaves[i]].outFrame2 =
221                 (u8 *)(&shave_outputFrame[FRAME_WIDTH * TASK_FRAME_HEIGHT *
    los_broken_shaves[idx] +
222                                     FRAME_WIDTH * FRAME_HEIGHT / 2]);
223                 idx++;
224                 break;
225             }
226         }
227         sc = OsDrvTimerGetSystemTicks64(&scheduling_endTicks);
228         temp_system_ticks = temp_system_ticks + (scheduling_endTicks -
    scheduling_startTicks);
229
230
231         for (u32 i = 0; i < number_of_working_shaves; i++) {
232             sc = OsDrvSvuOpenShave(handlers+(los_working_shaves[i]),
    los_working_shaves[i],OS_MYR_PROTECTION_SEM);
233             if(sc == OS_MYR_DRV_SUCCESS){
234             //printf("Started shave %d\n", los_working_shaves[i]);
235             sc = OsDrvSvuResetShave(handlers+(los_working_shaves[i]));
236                 if (sc)
237                 exit(sc);
238                 sc = OsDrvSvuSetAbsoluteDefaultStack(handlers+(los_working_shaves[i
    ]));
239                 if (sc)
240                     exit(sc);
241
242                 sc = OsDrvSvuStartShaveCC(handlers+(los_working_shaves[i]),
243                                     (u32)entryPoints[(los_working_shaves[i])],
     "i",
244                                     (void *)&threadArgs[(los_working_shaves[i
    ])]);
245
246                 if (sc)
247                     exit(sc);
248             } else{
249                 printf("Error cannot open shave %d\n", (los_working_shaves[i]));
250                 exit(sc);
251             }
252         }
253
254     for (u32 i = 0; i < number_of_working_shaves; i++) {
255         //printf("Waiting for shave number %ld\n",los_working_shaves[i]);
256         sc = OsDrvSvuWaitShaves(1, handlers+(los_working_shaves[i]),1000,
257                                 &running);
```

```
        //printf("Shave Number %ld returned \n",los_working_shaves[i]);
        if (sc)
            exit(sc);
        sc = OsDrvSvuCloseShave(handlers+(los_working_shaves[i]));
        if (sc)
            exit(sc);
        }

}
    . . .
    . . .
    . . .
```

In the code listing above we can see the important section of the above flow chart. First of all, we expose the way we figure out whether a SHAVE has corrupted data or not. If a SHAVE is broken it gets inserted inside an array that keeps the ids of all the broken SHAVEs. Apart from that we also expose the scheduling logic used by LEON OS. The scheduler has three discrete cases: First of all, we have the case where the number of SHAVEs broken is less than the number of working SHAVEs. In this case we just reschedule all the data of the broken SHAVEs to any of the working ones. The second case is having more broken SHAVEs than working. In this case we need to re execute the workload in chunchs of the number of working SHAVEs. However, this results in a time overhead since we do not use all the "available resources". Lastly, we have the case were all the SHAVEs are broken. In this case, we need to reset all SHAVEs and just re execute everything.

Fault tolerant architecture 2: System restoration from error in SHAVEs data, caused by SHAVEs

In the context of this policy, SHAVEs after receiving their data from LOS pass it through a C.R.C function. They then store the value returned by this function in order to be able to understand in the future if the data has been corrupted. Before processing the data, we pass it through this function again so that there is no doubt about whether it has been altered. In case an error is detected then the system goes into recovery mode as this means that there is some damage to the SHAVEs. More specifically, we distinguish two cases. In the first case, each damaged SHAVE takes care of its own self-feeding. Therefore, it restores its data from the available backup created by LOS when it received the data from the corresponding peripheral. Subsequently, after restoring the correct data it runs the code in order to compute the correct data. The other case we distinguish is the following: The LOS checks if the data has been corrupted. If corrupted, it restores the data to its original, uncorrupted, state and re-routes it to SHAVEs that it knows are working, in order to make sure the input data is processed correctly. It then resets the broken SHAVEs in order to be able to use them in the future.

Below are two flowcharts for each of the policies just mentioned. The first one corresponds to the case where SHAVEs heal their own error. The second one showcases the case where SHAVEs inform LOS with regard to the error and he then reschedules the data to working SHAVEs.

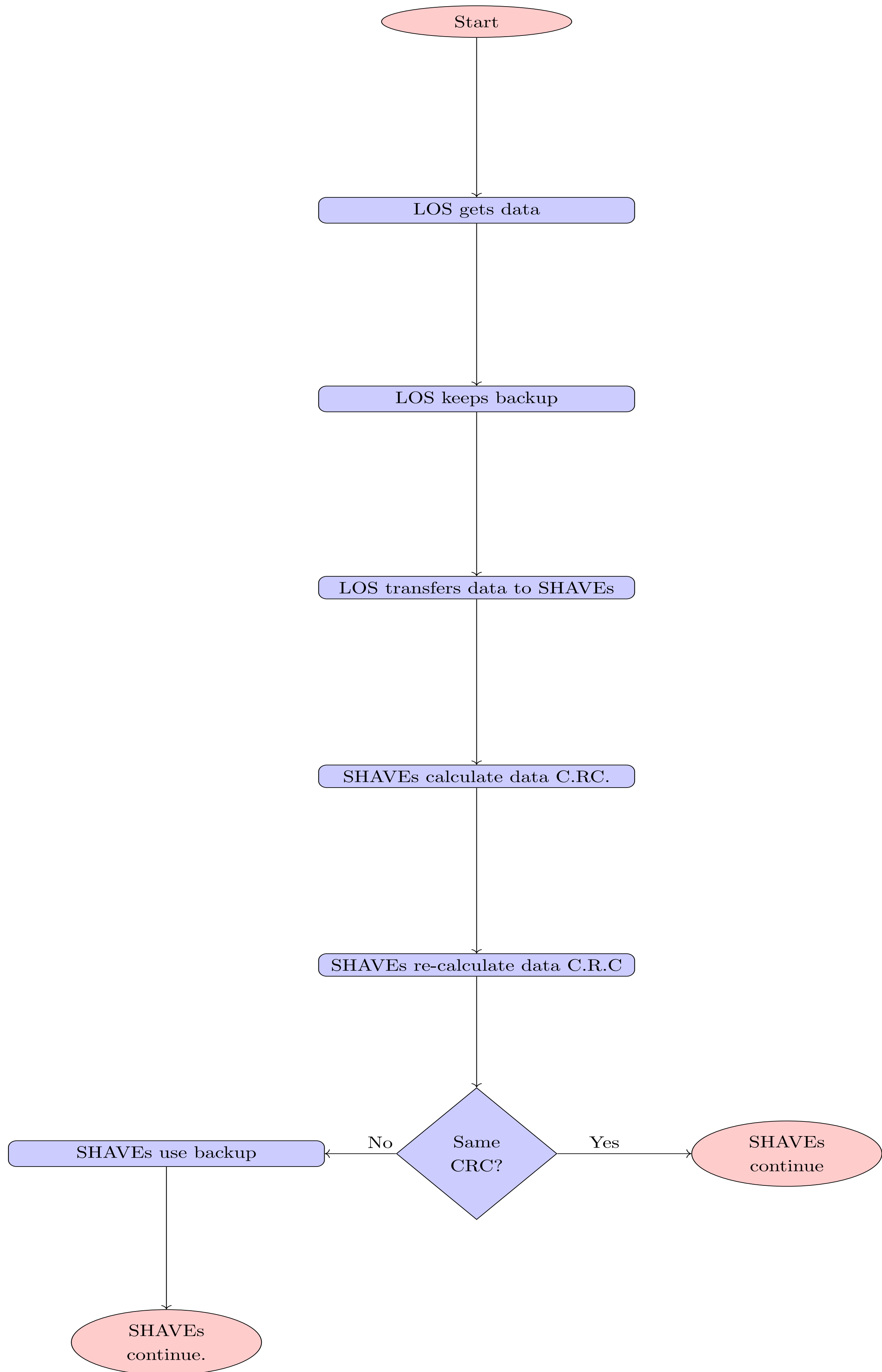


Figure 3.3: SHAVEs corrupt their data, SHAVEs amend error!

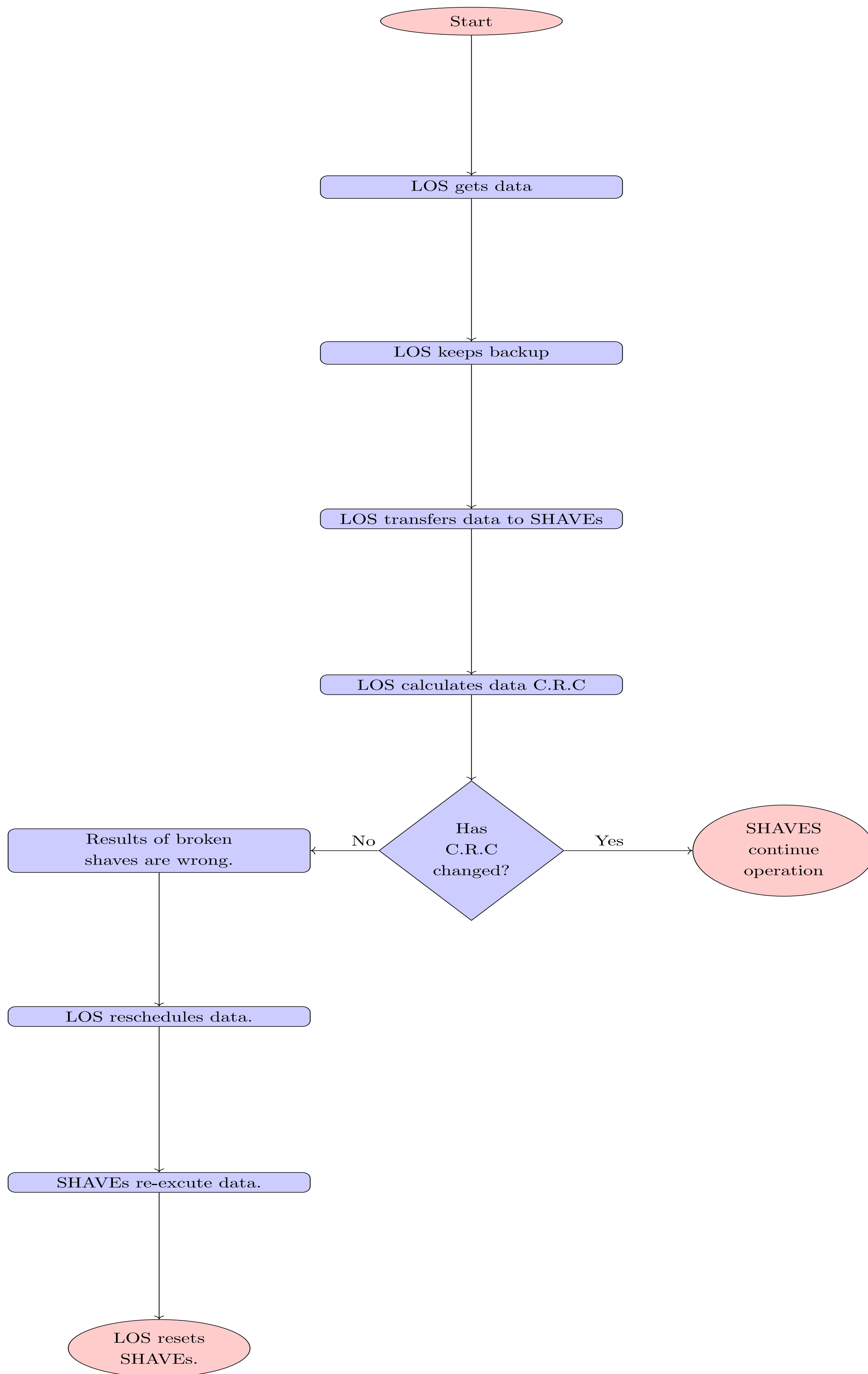


Figure 3.4: SHAVEs corrupt their data, LOS amends error!

Inspecting the above flow charts as we can see there are two discrete cases for this fault tolerant policy. This derives from the fact that if a SHAVE realizes there is an internal error then we are not sure whether we can rely on its output or not. Therefore, we introduce LOS as a scheduler in order to make sure that we use only properly working SHAVEs upon error correction. In the case of LOS fixing the output the same scheduler as the previous section was used. Therefore, we will not provide the source code in this section. Lastly, down below we can see the code that gets invoked in the case where SHAVEs use a backup to execute their workload:

```
1     CONV2D_ThreadArgs *thArgs = (CONV2D_ThreadArgs *)thArgsp;
2     u8 *inFrame1 = thArgs->inFrame1;
3     u8 *outFrame1 = thArgs->outFrame1;
4     u8 *inFrame2 = thArgs->inFrame2;
5     u8 *outFrame2 = thArgs->outFrame2;
6     u8 shave_id = thArgs->shave_id;
7     u32 crc = swcCalcCrc32(inFrame1,sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
8     extern u8 corrupted_inputFrame1024[];
9     if(thArgs->is_broken==true){
10        inFrame1 = &corrupted_inputFrame1024[0];
11        inFrame2 = &corrupted_inputFrame1024[0];
12        u32 crc_after_error_injection = swcCalcCrc32(inFrame1,sizeof(u8)*
      FRAME_WIDTH * TASK_FRAME_HEIGHT);
13    if((crc_after_error_injection != crc) && (thArgs->is_broken==true)){
14        thArgs->shave_realized_internal_error = true;
15        inFrame1 = thArgs->backup_inFrame1;
16        inFrame2 = thArgs->backup_inFrame2;
17    }
```

In the above code snippet we see that SHAVEs calculate the C.R.C of their data. Once this check is calculated then the error is inserted by SHAVEs if they are broken. After that, SHAVEs recalculate the C.R.C of their data and if it is different than the origincal value then they use the backup frames provided on their initialization by LEON OS. After that they continue with the execution of their intended kernel.

Fault tolerant architecture 3: System restoration from error in variable shared with SHAVEs

In this particular policy the system has been working properly for some time. But, SHAVES and LEON OS use common variables. But after the failure, the common variables of the two aforementioned subsystems change. This event in turn leads to corruption of the data processed by the system and consequently, if not corrected, to an incorrect result. In order to prevent the incorrect operation of the system, the SHAVES pass the common data through a function that computes their C.R.C. Consequently, before processing this data, they check whether it has been corrupted and consequently two cases can be distinguished. The first case is that there is no corruption of the data and consequently no action should be taken. The second, is if the value of C.R.C is changed and consequently SHAVEs become aware that something is wrong. After, they realize that there is an error then two data reset policies are applied. In the first policy the "broken" SHAVEs use a local backup of the data they have stored. This ensures correct operation. In the second policy, instead of SHAVEs perceiving the problem in the data, LEON OS perceives it. The way it deduces this is to pass the data it sends to SHAVEs through a C.R.C function. This way it understands that the common data has been corrupted. It then re-routes this data to the SHAVEs that it knows are working correctly in order to process it again and ensure a correct result. The flowcharts below show how the above policies are executed.

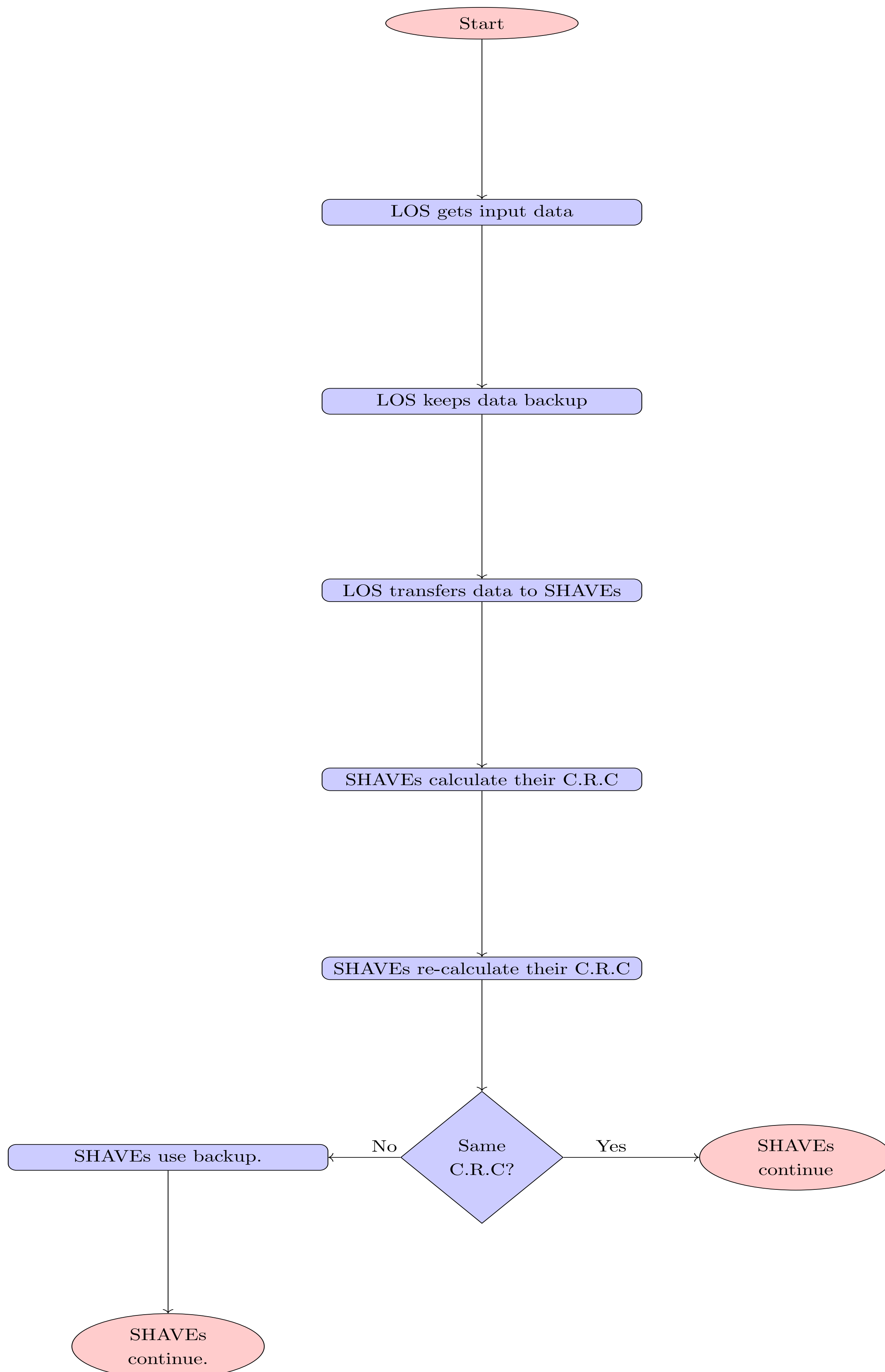


Figure 3.5: LOS corrupts shared variables, SHAVEs amend error!

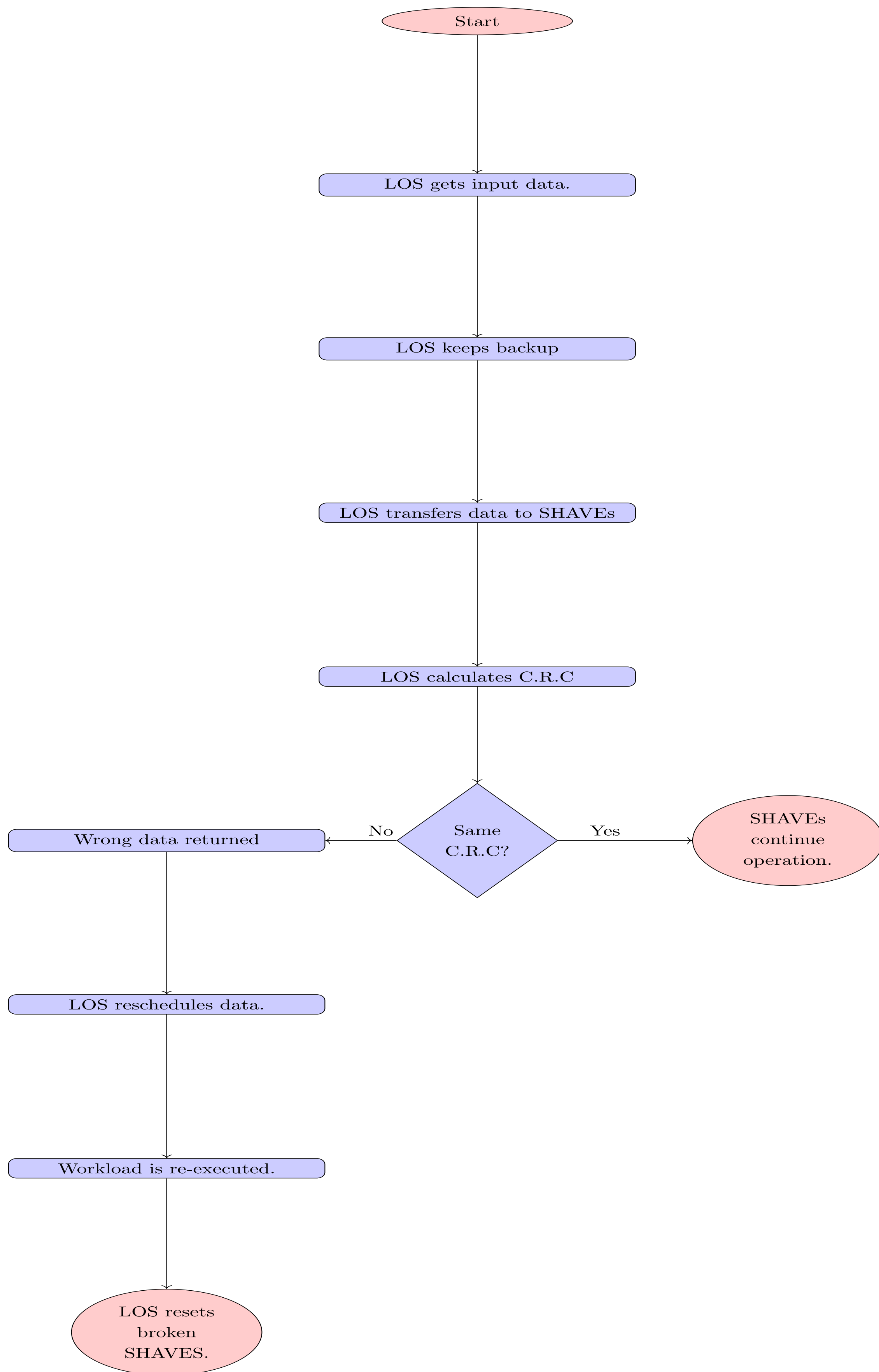


Figure 3.6: LOS corrupts shared variables, LOS amend error!

In this architecture the underlying code is similar to the previous section. The main difference relies in the way the error is injected and the fact that some variables. Furthermore, the scheduler is the same as the one mentioned previously, in the case of LOS correcting the data error, therefore it will not be included in this section either.

Fault tolerant architecture 4: System restoration from error in SHAVE instruction memory

Error recovery in the memory where the code of SHAVEs. A key feature of Myriad 2 is the fact that it has a unified memory Space. Therefore, checks can be made to see if any alteration has occurred throughout the memory space contents. Also, it should be mentioned that the part in memory in which the various SHAVEs code is manually defined in a config file. Therefore, changes may be made to the contents of not only in memory but also in the commands executed by SHAVEs. In this policy, LEON OS takes the C.R.C values of the memory-space occupied by the instructions of SHAVEs. By extension, LEON OS checks whether the contents of the instruction-memory of SHAVEs have been altered, by repeatedly calculating the C.R.C. of those memory locations. Since the code of SHAVEs does not change in run-time under normal circumstances any alteration of these memory locations are due to a hardware malfunction. So in such a case LEON OS senses the data corruption and considers invalid the results of SHAVES that have corrupted the memory storing So its next action is to route the input data that led to incorrect results to the functional SHAVEs. Finally, after the execution of the workload is complete, LEON OS resets the faulty SHAVEs. The following code snippet shows the code used to deduce whether a SHAVE coprocessor has any errors in its instruction memory harm.

```
1   . . .
2   . . .
3   . . .
4   for (int i=0; i< SHAVES_USED;i++){
5           array_of_memory_crcs[i] = swcCalcCrc32((u8 *) 0x80000000 +(i*0x00020000
    ),128000,be_pointer);
6
7   }
8   . . .
9   . . .
10  . . .
11  for (int i=0; i< SHAVES_USED;i++){
12      sc = OsDrvTimerGetSystemTicks64(&hashing_start_ticks_1);
13      array_of_memory_crcs_after_mistake_insertion[i] = swcCalcCrc32((u8 *) 0
    x80000000 +(i*0x00020000),128000,be_pointer);
14      sc = OsDrvTimerGetSystemTicks64(&hashing_end_ticks_1);
15      if(array_of_memory_crcs_after_mistake_insertion[i]!= array_of_memory_crcs[i
    ]){
16      los_broken_shaves[broken_shaves_number] = i;
17      broken_shaves_number++;
18      }
19  }
20  . . .
21  . . .
22  . . .
```

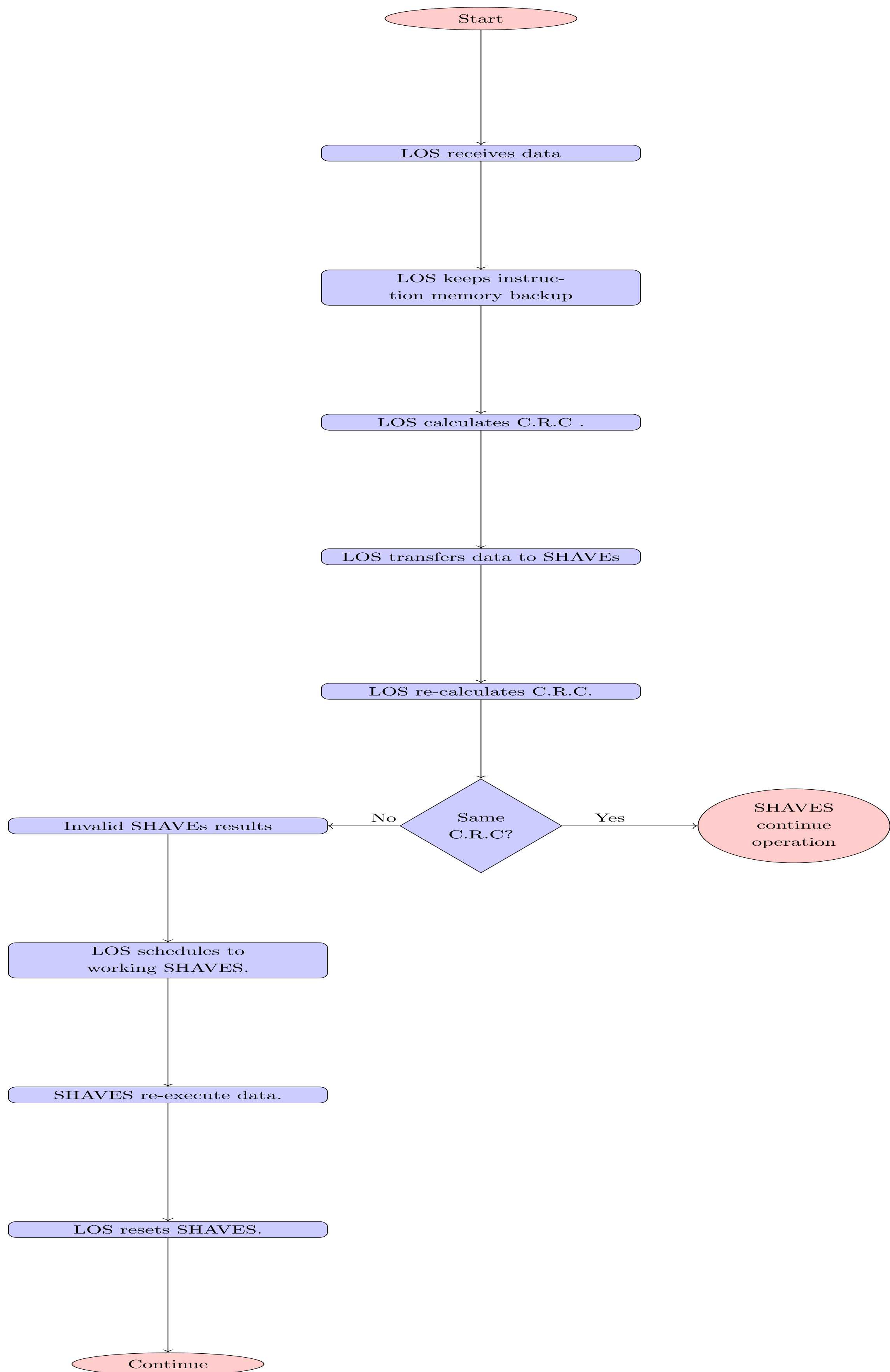


Figure 3.7: LOS correct error in instruction memory of SHAVEs by scheduling data to working shaves.

Fault tolerant architectures that use voting systems

A key requirement to ensure the system reliability of an embedded system is the existence of redundant resources. More specifically, there must always be an alternate path for potential failure. This redundancy may be either in memory, e.g. allocating data more than once, or in computational units, i.e. having more than one computational unit so that in the event of a failure the data can be routed to the backup. Therefore, in the case of Myriad 2 having processing multiple processing resources due to SHAVEs. Thus, some of them may be used in order to ensure the fault resilience of the system.

The best known technique for checking the correct operation of the various computing units used from embedded to cloud computing is N-voting systems. These systems allocate the available resources to ballots and each computing unit is assigned to a ballot. The computing units assigned to the same ballot have the same input data and also go through the same processing. Subsequently, each computational unit places the result of its calculations into the ballot. After each computing unit has finished the results are compared. After the comparison, the result obtained by the majority of the SHAVES is considered correct. From the above we come to the following conclusions:

- The number of processing units per ballot must be an odd number. If it was an even number then there might be ties in the results and we might not be able to deduce which is the correct result.
- The fact that a result has been voted by a majority as correct does not mean that it is correct. More specifically, let us consider the error as a function f that has as its set of values the set of data that will be affected by the error function. The result of this function is the erroneous input data a. Therefore, all the preceding can be summarized in the expression $f(x) = a$. Thus, all that is needed in order for two processing units to find the same result is that the function f and the data x are the same. Consequently, the majority may come up with incorrect results.

In the context of this paper, two different n-vector systems have been implemented.
These two systems are:

- 3-Voting System

  In this system, SHAVEs are assigned to a ballot every three. In this case each SHAVE processes the same data in the same way the same data. Finally, they place their results in their assigned ballot and we find the result with which the majority agrees.

- **5-Voting Sytem**

  In this case we have two different ballots and the SHAVEs are assigned by five to one of them. The reason why the SHAVEs assigned per ballot box is five is to avoid tie votes.

In the following subsection we will introduce a semantic representation of the aforementioned systems and the source code with regard to the voting process.

Fault tolerant architecture 5: 3 Voting system

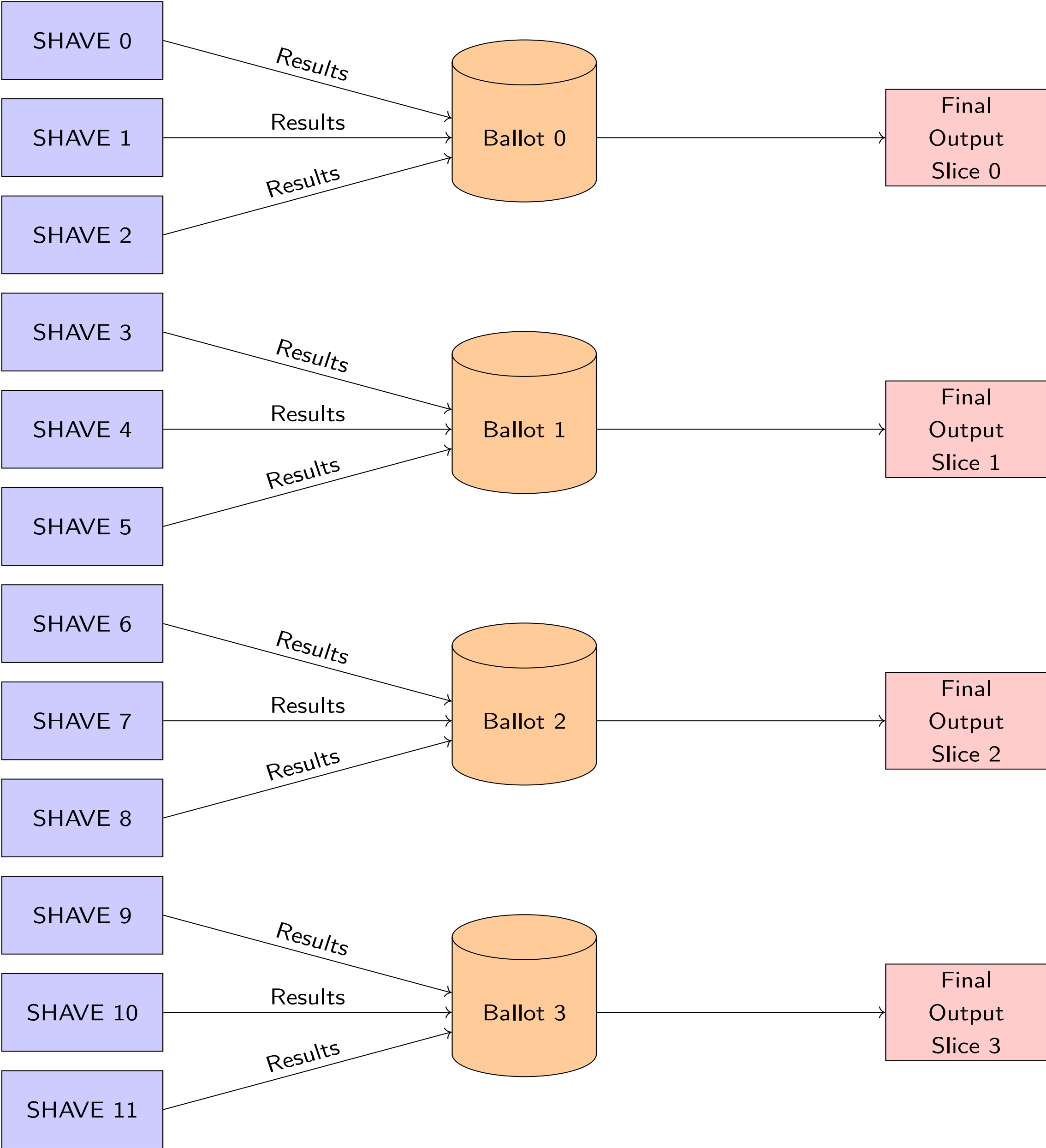


Figure 3.8: N-voting Systems: 4 Voter System.

In the case of the 3 voting system down below we can see the implementation of the voting system. It explores all the possible combinations of 2 when we have 3 elements in total. Based on whether a consensus is reached within a ballot the corresponding result is forwarded as the final output. Last but not least, if no consensus is reached then a result is forwarded at random since we cannot trust any of the SHAVEs.

```
1     int choice;
2    int comparison = memcmp(ballot_0_outFrame1[0], ballot_1_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
3        if(comparison == 0){
4            // printf("Shave 0 and shave 4 have the same output\n");
5            memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
6            memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
7            goto end_balot_1;
8        }
9        comparison = memcmp(ballot_0_outFrame1[0], ballot_2_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
10       if (comparison == 0){
11           // printf("Shave 0 and shave 8 have the same output\n");
12           memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
13           memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
14           goto end_balot_1;
15       }
16       comparison = memcmp(ballot_1_outFrame1[0], ballot_2_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
17       if(comparison == 0){
18           // printf("Shave 4 and shave 8 have the same output\n");
19           memcpy(shv_outFrame1[0], ballot_1_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
20           memcpy(shv_outFrame2[0], ballot_1_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
21           goto end_balot_1;
22       }
23       choice = random(0, 2, startTicks);
24       // printf("No concenus in ballot 0\n");
25       if(choice == 0){
26           // printf("Copying output from shave 0\n");
27           memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
28           memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
29       }
30       else if(choice == 1){
31           // printf("Copying output from shave 4\n");
32           memcpy(shv_outFrame1[0], ballot_1_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
33           memcpy(shv_outFrame2[0], ballot_1_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
34       }
35       else{
36           // printf("Copying output from shave 8\n");
37           memcpy(shv_outFrame1[0], ballot_2_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
38           memcpy(shv_outFrame2[0], ballot_2_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
```

```
     TASK_FRAME_HEIGHT);
39         }
40 
41 end_balot_1:
42     int comparison_2 = memcmp(ballot_0_outFrame1[1], ballot_1_outFrame1[1], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
43         if(comparison_2 == 0){
44             // printf("Shave 1 and shave 5 have the same output\n");
45             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
46             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
47             goto end_balot_2;
48         }
49         comparison_2 = memcmp(ballot_0_outFrame1[1], ballot_2_outFrame1[1], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
50         if (comparison_2 == 0){
51             // printf("Shave 1 and shave 9 have the same output\n");
52             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
53             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
54             goto end_balot_2;
55         }
56         comparison_2 = memcmp(ballot_1_outFrame1[1], ballot_2_outFrame1[1], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
57         if(comparison_2 == 0){
58             // printf("Shave 5 and shave 9 have the same output\n");
59             memcpy(shv_outFrame1[1], ballot_1_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
60             memcpy(shv_outFrame2[1], ballot_1_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
61             goto end_balot_2;
62         }
63         choice = random(0, 2, startTicks);
64         // printf("No concenus in ballot 1\n");
65         if(choice == 0){
66             // printf("Copying output from shave 1\n");
67             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
68             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
69         }
70         else if(choice == 1){
71             // printf("Copying output from shave 5\n");
72             memcpy(shv_outFrame1[1], ballot_1_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
73             memcpy(shv_outFrame2[1], ballot_1_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
74         }
75         else{
76             // printf("Copying output from shave 9\n");
77             memcpy(shv_outFrame1[1], ballot_2_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
78             memcpy(shv_outFrame2[1], ballot_2_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
79         }
80 end_balot_2:
```

```
81      int comparison_3 = memcmp(ballot_0_outFrame1[2], ballot_1_outFrame1[2], sizeof(
        u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
82          if(comparison_3 == 0){
83              // printf("Shave 2 and shave 6 have the same output\n");
84              memcpy(shv_outFrame1[2], ballot_0_outFrame1[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
85              memcpy(shv_outFrame2[2], ballot_0_outFrame2[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
86              goto end_balot_3;
87          }
88          comparison_3 = memcmp(ballot_0_outFrame1[2], ballot_2_outFrame1[2], sizeof(
        u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
89          if (comparison_3 == 0){
90              // printf("Shave 2 and shave 10 have the same output\n");
91              memcpy(shv_outFrame1[2], ballot_0_outFrame1[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
92              memcpy(shv_outFrame2[2], ballot_0_outFrame2[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
93              goto end_balot_3;
94          }
95          comparison_3 = memcmp(ballot_1_outFrame1[2], ballot_2_outFrame1[2], sizeof(
        u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
96          if(comparison_3 == 0){
97              // printf("Shave 6 and shave 10 have the same output\n");
98              memcpy(shv_outFrame1[2], ballot_1_outFrame1[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
99              memcpy(shv_outFrame2[2], ballot_1_outFrame2[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
100             goto end_balot_3;
101         }
102         choice = random(0, 2, startTicks);
103         // printf("No concenus in ballot 2\n");
104         if(choice == 0){
105             // printf("Copying output from shave 2\n");
106             memcpy(shv_outFrame1[2], ballot_0_outFrame1[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
107             memcpy(shv_outFrame2[2], ballot_0_outFrame2[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
108         }
109         else if(choice == 1){
110             // printf("Copying output from shave 6\n");
111             memcpy(shv_outFrame1[2], ballot_1_outFrame1[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
112             memcpy(shv_outFrame2[2], ballot_1_outFrame2[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
113         }
114         else{
115             // printf("Copying output from shave 10\n");
116             memcpy(shv_outFrame1[2], ballot_2_outFrame1[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
117             memcpy(shv_outFrame2[2], ballot_2_outFrame2[2],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
118         }
119
120 end_balot_3:
121     int comparison_4 = memcmp(ballot_0_outFrame1[3], ballot_1_outFrame1[3], sizeof(
        u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
122         if(comparison_4 == 0){
```

```
123             // printf("Shave 3 and shave 7 have the same output\n");
124             memcpy(shv_outFrame1[3], ballot_0_outFrame1[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
125             memcpy(shv_outFrame2[3], ballot_0_outFrame2[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
126             goto end_balot_4;
127         }
128         comparison_4 = memcmp(ballot_0_outFrame1[1], ballot_2_outFrame1[1], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
129         if (comparison_2 == 0){
130             // printf("Shave 3 and shave 11 have the same output\n");
131             memcpy(shv_outFrame1[3], ballot_0_outFrame1[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
132             memcpy(shv_outFrame2[3], ballot_0_outFrame2[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
133             goto end_balot_4;
134         }
135         comparison_4 = memcmp(ballot_1_outFrame1[1], ballot_2_outFrame1[1], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
136         if(comparison_4 == 0){
137             // printf("Shave 7 and shave 11 have the same output\n");
138             memcpy(shv_outFrame1[3], ballot_1_outFrame1[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
139             memcpy(shv_outFrame2[3], ballot_1_outFrame2[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
140             goto end_balot_4;
141         }
142         // printf("No concenus in ballot 3\n");
143         choice = random(0, 2, startTicks);
144         if(choice == 0){
145             // printf("Copying output from shave 3\n");
146             memcpy(shv_outFrame1[3], ballot_0_outFrame1[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
147             memcpy(shv_outFrame2[3], ballot_0_outFrame2[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
148         }
149         else if(choice == 1){
150             // printf("Copying output from shave 7\n");
151             memcpy(shv_outFrame1[3], ballot_1_outFrame1[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
152             memcpy(shv_outFrame2[3], ballot_1_outFrame2[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
153         }
154         else{
155             // printf("Copying output from shave 11\n");
156             memcpy(shv_outFrame1[3], ballot_2_outFrame1[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
157             memcpy(shv_outFrame2[3], ballot_2_outFrame2[3],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
158         }
159 end_balot_4:
```

Fault tolerant architecture 6: 5 Voting system

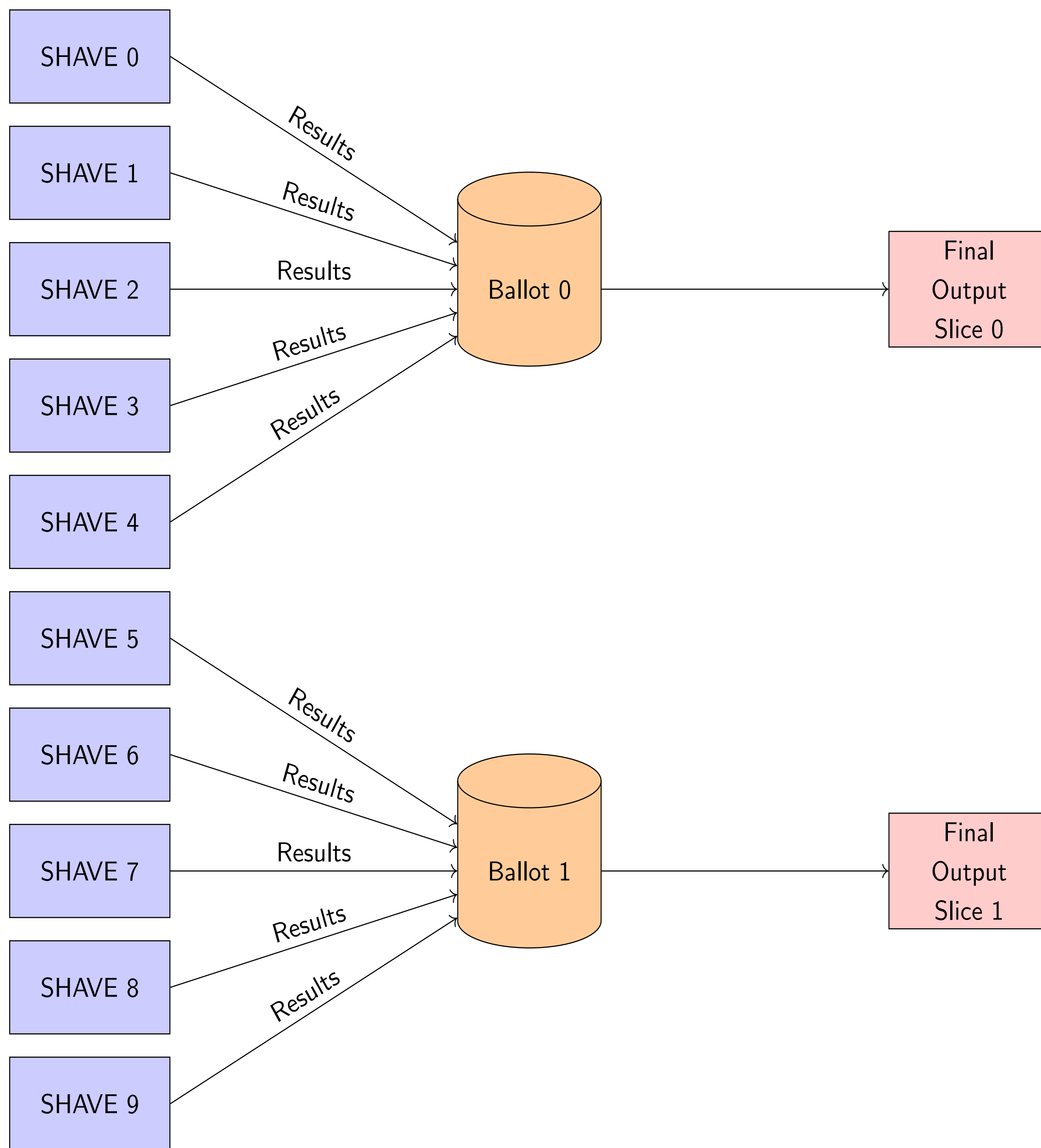


Figure 3.9: N-voting Systems: 2 Voter System.

In the case of the 5 voting system we have more possible combinations since $\binom{5}{3} = 10$. Therefore, we need to enumerate all the possible combinations in order to assert that the voting system will be working as desired. Furthermore, on the diagram shown above we can observe that only 10 SHAVEs out of the 12 that are in total available are used. This design choice was taken, in order to avoid any issues that might arise in case of ties. Down below, is the source code of the voting system:

```
        int comparison_1_1 = memcmp(ballot_0_outFrame1[0], ballot_1_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
    int comparison_1_2 = memcmp(ballot_0_outFrame1[0], ballot_2_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
        if((comparison_1_1 == 0) && (comparison_1_2 == 0)){
            memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            goto end_balot_1;
        }
    comparison_2_1 = memcmp(ballot_0_outFrame1[0], ballot_1_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
    comparison_2_2 = memcmp(ballot_0_outFrame1[0], ballot_3_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
        if((comparison_2_1 == 0) && (comparison_2_2 == 0)){
            memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            goto end_balot_1;
        }
    comparison_3_1 = memcmp(ballot_0_outFrame1[0], ballot_1_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
    comparison_3_2 = memcmp(ballot_0_outFrame1[0], ballot_4_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
        if((comparison_3_1 == 0) && (comparison_3_2 == 0)){
            memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            goto end_balot_1;
        }
    comparison_4_1 = memcmp(ballot_0_outFrame1[0], ballot_2_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
    comparison_4_2 = memcmp(ballot_0_outFrame1[0], ballot_3_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
        if((comparison_4_1 == 0) && (comparison_4_2 == 0)){
            memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
            goto end_balot_1;
        }
    comparison_5_1 = memcmp(ballot_0_outFrame1[0], ballot_3_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
    comparison_5_2 = memcmp(ballot_0_outFrame1[0], ballot_4_outFrame1[0], sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
        if((comparison_5_1 == 0) && (comparison_5_2 == 0)){
            memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT);
```

```
33            memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
34            goto end_balot_1;
35        }
36    comparison_6_1 = memcmp(ballot_0_outFrame1[0], ballot_2_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
37    comparison_6_2 = memcmp(ballot_0_outFrame1[0], ballot_4_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
38        if((comparison_6_1 == 0) && (comparison_6_2 == 0)){
39            memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
40            memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
41            goto end_balot_1;
42        }
43    comparison_7_1 = memcmp(ballot_2_outFrame1[0], ballot_3_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
44    comparison_7_2 = memcmp(ballot_2_outFrame1[0], ballot_4_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
45        if((comparison_7_1 == 0) && (comparison_7_2 == 0)){
46            memcpy(shv_outFrame1[0], ballot_2_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
47            memcpy(shv_outFrame2[0], ballot_2_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
48            goto end_balot_1;
49        }
50    comparison_8_1 = memcmp(ballot_1_outFrame1[0], ballot_2_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
51    comparison_8_2 = memcmp(ballot_1_outFrame1[0], ballot_3_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
52        if((comparison_8_1 == 0) && (comparison_8_2 == 0)){
53            memcpy(shv_outFrame1[0], ballot_1_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
54            memcpy(shv_outFrame2[0], ballot_1_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
55            goto end_balot_1;
56        }
57    comparison_9_1 = memcmp(ballot_1_outFrame1[0], ballot_3_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
58    comparison_9_2 = memcmp(ballot_1_outFrame1[0], ballot_4_outFrame1[0], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
59        if((comparison_9_1 == 0) && (comparison_9_2 == 0)){
60            memcpy(shv_outFrame1[0], ballot_1_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
61            memcpy(shv_outFrame2[0], ballot_1_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
62            goto end_balot_1;
63        }
64    comparison_10_1 = memcmp(ballot_1_outFrame1[0], ballot_2_outFrame1[0], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
65    comparison_10_2 = memcmp(ballot_1_outFrame1[0], ballot_4_outFrame1[0], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
66        if((comparison_10_1 == 0) && (comparison_10_2 == 0)){
67            memcpy(shv_outFrame1[0], ballot_1_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
68            memcpy(shv_outFrame2[0], ballot_1_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
69            goto end_balot_1;
```

```
70          }
71          choice = random(0, 4, startTicks);
72          // printf("No concenus in ballot 0\n");
73          if(choice == 0){
74              // printf("Copying output from shave 0\n");
75              memcpy(shv_outFrame1[0], ballot_0_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
76              memcpy(shv_outFrame2[0], ballot_0_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
77          }
78          else if(choice == 1){
79              // printf("Copying output from shave 4\n");
80              memcpy(shv_outFrame1[0], ballot_1_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
81              memcpy(shv_outFrame2[0], ballot_1_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
82          }
83          else if(choice == 2){
84              // printf("Copying output from shave 4\n");
85              memcpy(shv_outFrame1[0], ballot_2_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
86              memcpy(shv_outFrame2[0], ballot_2_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
87          }
88          else if(choice == 3){
89              // printf("Copying output from shave 4\n");
90              memcpy(shv_outFrame1[0], ballot_3_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
91              memcpy(shv_outFrame2[0], ballot_3_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
92          }
93          else{
94              // printf("Copying output from shave 8\n");
95              memcpy(shv_outFrame1[0], ballot_4_outFrame1[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
96              memcpy(shv_outFrame2[0], ballot_4_outFrame2[0],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
97          }
98
99  end_balot_1:
100     comparison_1_1 = memcmp(ballot_0_outFrame1[1], ballot_1_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
101     comparison_1_2 = memcmp(ballot_0_outFrame1[1], ballot_2_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
102         if((comparison_1_1 == 0) && (comparison_1_2 == 0)){
103             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
104             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
105             goto end_balot_1_1;
106         }
107     comparison_2_1 = memcmp(ballot_0_outFrame1[1], ballot_1_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
108     comparison_2_2 = memcmp(ballot_0_outFrame1[1], ballot_3_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
109         if((comparison_2_1 == 0) && (comparison_2_2 == 0)){
110             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
```

```
111             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
112             goto end_balot_1_1;
113         }
114     comparison_3_1 = memcmp(ballot_0_outFrame1[1], ballot_1_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
115     comparison_3_2 = memcmp(ballot_0_outFrame1[1], ballot_4_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
116         if((comparison_3_1 == 0) && (comparison_3_2 == 0)){
117             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
118             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
119             goto end_balot_1_1;
120         }
121     comparison_4_1 = memcmp(ballot_0_outFrame1[1], ballot_2_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
122     comparison_4_2 = memcmp(ballot_0_outFrame1[1], ballot_3_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
123         if((comparison_4_1 == 0) && (comparison_4_2 == 0)){
124             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
125             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
126             goto end_balot_1_1;
127         }
128     comparison_5_1 = memcmp(ballot_0_outFrame1[1], ballot_3_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
129     comparison_5_2 = memcmp(ballot_0_outFrame1[1], ballot_4_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
130         if((comparison_5_1 == 0) && (comparison_5_2 == 0)){
131             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
132             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
133             goto end_balot_1_1;
134         }
135     comparison_6_1 = memcmp(ballot_0_outFrame1[1], ballot_2_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
136     comparison_6_2 = memcmp(ballot_0_outFrame1[1], ballot_4_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
137         if((comparison_6_1 == 0) && (comparison_6_2 == 0)){
138             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
139             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
140             goto end_balot_1_1;
141         }
142     comparison_7_1 = memcmp(ballot_2_outFrame1[1], ballot_3_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
143     comparison_7_2 = memcmp(ballot_2_outFrame1[1], ballot_4_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
144         if((comparison_7_1 == 0) && (comparison_7_2 == 0)){
145             memcpy(shv_outFrame1[1], ballot_2_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
146             memcpy(shv_outFrame2[1], ballot_2_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
147             goto end_balot_1_1;
```

```
148         }
149     comparison_8_1 = memcmp(ballot_1_outFrame1[1], ballot_2_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
150     comparison_8_2 = memcmp(ballot_1_outFrame1[1], ballot_3_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
151         if((comparison_8_1 == 0) && (comparison_8_2 == 0)){
152             memcpy(shv_outFrame1[1], ballot_1_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
153             memcpy(shv_outFrame2[1], ballot_1_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
154             goto end_balot_1_1;
155         }
156     comparison_9_1 = memcmp(ballot_1_outFrame1[1], ballot_3_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
157     comparison_9_2 = memcmp(ballot_1_outFrame1[1], ballot_4_outFrame1[1], sizeof(u8
    )*FRAME_WIDTH * TASK_FRAME_HEIGHT );
158         if((comparison_9_1 == 0) && (comparison_9_2 == 0)){
159             memcpy(shv_outFrame1[1], ballot_1_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
160             memcpy(shv_outFrame2[1], ballot_1_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
161             goto end_balot_1_1;
162         }
163     comparison_10_1 = memcmp(ballot_1_outFrame1[0], ballot_2_outFrame1[0], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
164     comparison_10_2 = memcmp(ballot_1_outFrame1[0], ballot_4_outFrame1[0], sizeof(
    u8)*FRAME_WIDTH * TASK_FRAME_HEIGHT );
165         if((comparison_10_1 == 0) && (comparison_10_2 == 0)){
166             memcpy(shv_outFrame1[1], ballot_1_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
167             memcpy(shv_outFrame2[1], ballot_1_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
168             goto end_balot_1_1;
169         }
170     choice = random(0, 4, startTicks);
171     if(choice == 0){
172             memcpy(shv_outFrame1[1], ballot_0_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
173             memcpy(shv_outFrame2[1], ballot_0_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
174     }
175     else if(choice == 1){
176             memcpy(shv_outFrame1[1], ballot_1_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
177             memcpy(shv_outFrame2[1], ballot_1_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
178     }
179     else if(choice == 2){
180             memcpy(shv_outFrame1[1], ballot_2_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
181             memcpy(shv_outFrame2[1], ballot_2_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
182     }
183     else if(choice == 3){
184             memcpy(shv_outFrame1[1], ballot_3_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
185             memcpy(shv_outFrame2[1], ballot_3_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
     TASK_FRAME_HEIGHT);
```

```
186     }
187     else{
188             memcpy(shv_outFrame1[1], ballot_4_outFrame1[1],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
189             memcpy(shv_outFrame2[1], ballot_4_outFrame2[1],sizeof(u8)*FRAME_WIDTH *
         TASK_FRAME_HEIGHT);
190     }
191
192 end_balot_1_1:
```

Chapter 4

Experimental Evaluation

Experimental Setup

Platform

The platform used for the experimental Evaluation, as mentioned previously is Intel Movidius Myriad 2.

Below we see the architecture of the aforementioned microprocessors:

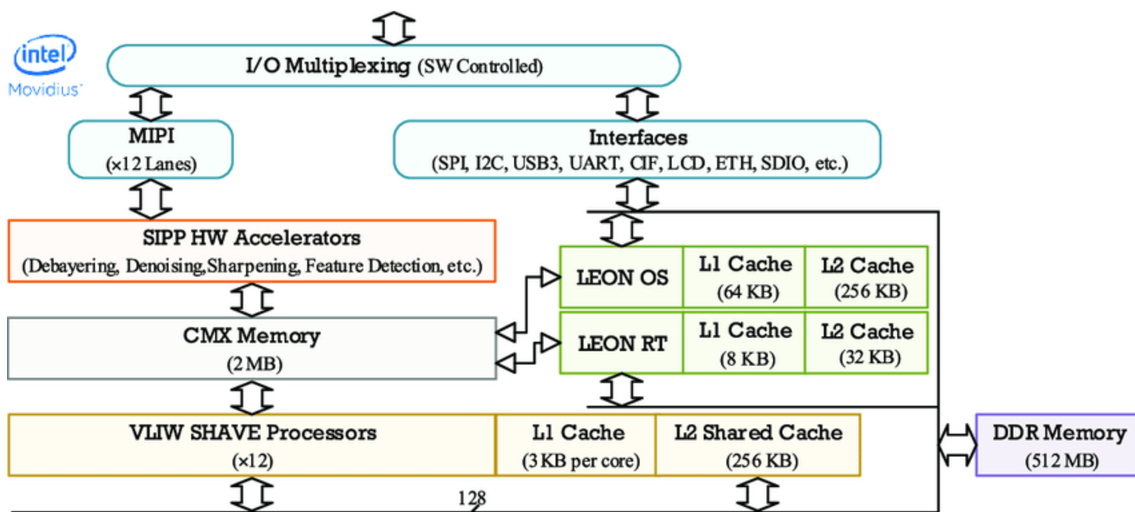


Figure 4.1: Myriad 2[1]

As can be seen from the above figure Myriad 2 has two 32 bit LEON4 general purpose RISC SPARCv8 processors of (LEON OS, LEON RT) architecture. The LEON OS supports a Real Time Operating System called RTEMS while the LEON RT is responsible for peripheral management. Responsible for the high throughput of the system are the Streaming Hybrid Architecture Vector Engines (SHAVEs) architecture kernels 128 bit, VLIW and SIMD. The orchestration of the operation of the SHAVEs is performed by the aforementioned general purpose processors. Finally, Myriad 2 has a set of hardware filters called Streaming Image Processing Pipeline (SIPP)

The memory organization of of Myriad 2 has both DDR and DRAM. In addition, special mention should be made of the Connection Matrix Memory (CMX) which is used as NUMA ScratchPad memory and contributes significantly to the high performance of the system. The

transfer of data from DDR to CMX, and vice versa, is done via a Direct Memory Access engine. In addition, LEON OS as well as LEON RT have L1 cache for both data and instructions as well as L2 cache. Finally, the SHAVEs each have a separate L1 cache and a shared L2 cache.

In this thesis, we used the SHAVEs co-processor and the CMX. Which, where used to speed up our benchmarks. The benchmarks will be more thoroughly explained in the following section.

Benchmarks

In order to be able to quantify the performance of the proposed models we had to develop and use programs that correspond to a realistic computational load for the platform Myriad 2. Since this module is used in machine vision applications it makes sense that the benchmarks developed correspond to computationally difficult operations on matrix elements. In these programs, we measured their performance under normal operation and under stress conditions in order to be able to measure the efficiency of both Fault-Tolerant policies and error-introduction techniques. A more detailed presentation of the benchmarks used follows below.

2D Convolution

One of the most classic digital signal processing operations is convolution. The operation of convolution is defined between two signals of at least one dimension. In one-dimensional convolution, the instantaneous values of signals that make overlap are summed and multiplied together. In addition, it should be noted that one of the two signals is inverted. The mathematical representation of convolution for discrete signals is:

$$(f * g)(t) = \sum_{-\infty}^{+\infty} f(\tau)g(t-\tau)$$

However, in addition to discrete signals, there are also continuous signals. In this case the convolution is defined as:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau$$

This definition extends to more than one dimension. The main difference between convolution in one and two dimensions is the fact that one of the two signals is inverted twice. The aforementioned process is used extensively in various widely used machine vision pipelines such as edge detection in an image. In the case of images two matrices are distinguished as the convolved signals are defined as a large two-dimensional matrix and a smaller one. The smaller matrix is called the kernel. The aforementioned properties of this operation make convolution a suitable benchmark to measure the performance of policies designed and implemented for both fault tolerance and error injection

2D Binning

Binning is a process in which the contents of pixel of adjacent regions are combined into a super-pixel and is a common process to reduce image noise by increasing the signal-to-noise ratio of the image. Usually, binning occurs in groups of four pixels to form a quad (range $2x2$). However, there are cases in which hexagons of pixels (range $4x4$). The reason for using $4x4$ regions is the fact that they increase the signal-to-noise ratio fourfold but under-square the resolution. Finally, 2D Binning is also done in $2x1$ and $1x2$ regions. However, this usage is not widespread in commercial applications. The main reason for using Binning is the fact that it increases the signal-to-noise ratio

which is one of the key metrics in image quality. In addition, there are two different categories of: summation-binning and the average-binning. Which of the two, however, brings about the optimal result depends very much on the application. With less noise the data can be subjected to higher levels of amplification in the pre-processing process. Therefore, 2D Binning is a particularly useful benchmark to quantify the performance of the thesis techniques.

Fault Injection

Theoretical background on metrics

The following metrics are used in all the error injection methods:

- Error Rate: The percentage of errors in the final output of SHAVEs. This is compared with a single core execution of the desired program which are called "Golden Data". Responsible for this execution is LEON OS.

- Mean Absolute Error: In statistics, mean absolute error (MAE) is a measure of errors between paired observations expressing the same phenomenon. Examples of Y versus X include comparisons of predicted versus observed, subsequent time versus initial time, and one technique of measurement versus an alternative technique of measurement. MAE is calculated as the sum of absolute errors divided by the sample size: $\mathrm{MAE} = (\frac{1}{n})\sum_{i=1}^{n}|y_i - x_i|$

- Mean Relative Error: The approximation error in a data value is the discrepancy between an exact value and some approximation to it. This error can be expressed as a relative error (the absolute error divided by the data value).

- Max Error: The highest error value.

- Max Relative Error: The highest mean relative error value.

- PSNR Error: Peak signal-to-noise ratio (PSNR) is an engineering term for the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. Because many signals have a very wide dynamic range, PSNR is usually expressed as a logarithmic quantity using the decibel scale. PSNR is commonly used to quantify reconstruction quality for images and video subject to lossy compression.

Fault injection design choices

The following design choices where taken with regard to error injection:

- The results following were inserted in 3, 6, 9 and 12 SHAVEs. 12 SHAVEs were chosen in order to simulate complete system breakdown.

- We designed 4 fault injection methods.

- The methods devised, inserted error in: Instruction memory of SHAVEs, Data transferred to SHAVEs, Shared Variable with SHAVEs, SHAVEs harm their own data.

Fault injection method: LEON OS corrupts memory contents of SHAVEs

This subsection presents the results of experimental measurements regarding the aliasing of the memory contents of SHAVEs by LEON OS for both benchmarks:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	13.5	24.2	31.5	44
Mean absolute error	235.6	137.7	170	172
Mean relative error	15	15	15	15
Max error	255	255	255	255
Max relative error	254	254	254	254
PSNR error (db)	9	9	7	6

Table 4.1: LEON OS corrupts SHAVE instruction memory: Results of 2D Convolution

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	8.5	12	16	24
Mean absolute error	33.6	15.2	26	18
Mean relative error	2	0.9	1.5	1
Max error	254	176	254	176
Max relative error	241	80	241	166
PSNR error (db)	35	42	34	37

Table 4.2: LEON OS corrupts SHAVE instruction memory: Results of 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The error rate increases as the number of corrupted SHAVEs increases.
- In both convolution and binning the final error does not exceed 50%. This is due to the number of memory-bytes being corrupted.
- All metrics except the average absolute error remain broadly constant in the case of convolution while in binning it changes.
- The very high max error is due to the fact that during corruption several memory elements are given a value of 255 instead of the original value of 0.

Fault injection method: LEON OS sends corrupted data to SHAVEs

This subsection presents the results of experimental measurements with reference to the aliasing of the input data of SHAVEs by LEON OS for both benchmarks:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	25	50	74.1	99.3
Mean absolute error	8.6	8.6	8.6	7.9
Mean relative error	0.1	0.1	0.1	0.1
Max error	155	160	160	160
Max relative error	6	6	7	7
PSNR error (db)	33	31	29	28

Table 4.3: LEON OS sends corrupted data to SHAVEs: Results of 2D Convolution

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	25	48	74	97
Mean absolute error	8.7	8.7	8.7	8.7
Mean relative error	0.3	0.3	0.3	0.3
Max error	72	72	72	72
Max relative error	7	7	7	7
PSNR error (db)	36	36	36	36

Table 4.4: LEON OS sends corrupted data to SHAVEs: Results of 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The error rate increases as the number of corrupted SHAVE increases.
- In both convolution and binning the final error marginally reaches 100%.
- All metrics remain broadly stable in both the convolution and binning case.
- The values of the different metrics depend on the error introduced. Therefore, their values remain broadly constant as the error is the same for each SHAVE.

Fault injection method: LEON OS corrupts variables shared with SHAVEs

In this subsection we present the results of experimental measurements regarding the corruption of variables common between SHAVEs and LEON OS for the two metroprograms:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	24	500	72.7	97.8
Mean absolute error	9.5	10	9.5	10
Mean relative error	0.5	0.4	0.4	0.3
Max error	79	155	160	160
Max relative error	64	73	73	73
PSNR error (db)	32	28	27	25

Table 4.5: LEON OS corrupts variable shared with SHAVEs: Results for 2D Convolution

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	23.5	48.2	73.4	99
Mean absolute error	14	14.8	14.5	14.6
Mean relative error	0.9	0.7	0.7	0.8
Max error	178	178	178	178
Max relative error	176	176	173	176
PSNR error (db)	34	31	29	28

Table 4.6: LEON OS corrupts variable shared with SHAVEs: Results for 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The error rate increases as the number of corrupted SHAVEs increases.
- In both convolution and binning the final error marginally reaches 100%.
- All metrics remain broadly stable in both the convolution and binning case.
- The values of the different metrics depend on the error introduced. Therefore, their values remain broadly constant as the error is the same for each SHAVE.

Fault injection method: SHAVEs harm their own data

This subsection presents the results of experimental measurements in reference to the destruction of SHAVEs data by the SHAVEs themselves for both metroprograms:

2D Convolution				
# of broken SHAVE	3	6	9	12
Error rate %	24	500	74.7	99.3
Mean absolute error	8	7.9	8	8
Mean relative error	0.1	0.1	0.1	0.1
Max error	121	155	160	160
Max relative error	7	7	7	77
PSNR error (db)	34	31	29	28

Table 4.7: SHAVEs corrupt their own data: Results of 2D Convolution

2D Binning				
# of broken SHAVE	3	6	9	12
Error rate %	24	47.5	74.6	99
Mean absolute error	9.3	9.3	9.6	9.4
Mean relative error	0.2	0.2	0.2	0.2
Max error	165	165	165	165
Max relative error	12	8	14	14
PSNR error (db)	39	36	34	33

Table 4.8: SHAVEs corrupt their own data: Results of 2D Convolution

Analyzing the above tables we observe the following for both benchmarks:

- The error rate increases as the number of corrupted SHAVEs increases.
- In both convolution and binning the final error marginally reaches 100%.
- All metrics remain broadly stable in both the convolution and binning case.
- The values of the different metrics depend on the error introduced. Therefore, their values remain broadly constant as the error is the same for each SHAVE.

Fault Tolerance

In this subsection, we present the results of the experimental measurements for the error recovery policy in the case of an error in the code memory of the SHAVEs coprocessors for the two metroprograms.

2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	81 ms	81 ms	65 ms	65 ms
2. Hashing memory / SHAVE	4.5 ms	4.5 ms	4.5 ms	4.5 ms
3. Rescheduling run time	0.003 ms	0.004 ms	0.006 ms	0 ms
4. SHAVE rerun time	81 ms	81 ms	87 ms	114 ms
5. Total run time	216 ms	217 ms	206 ms	233 ms
8. Error percentage	0%	0%	0%	0%

Table 4.9: Leon OS corrects SHAVE instruction memory harm, results for 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	2 ms	2 ms	2 ms	2 ms
2. Hashing memory / SHAVE	3 ms	3 ms	3 ms	3 ms
3. Rescheduling run time	0.002 ms	0.002 ms	0.003 ms	0 ms
4. SHAVE rerun time	2 ms	2 ms	8 ms	38 ms
5. Total run time	49 ms	49 ms	55 ms	85 ms
8. Error percentage	0.2%	0.4%	0.5%	0%

Table 4.10: Leon OS corrects SHAVE instruction memory harm, results for 2D Binning

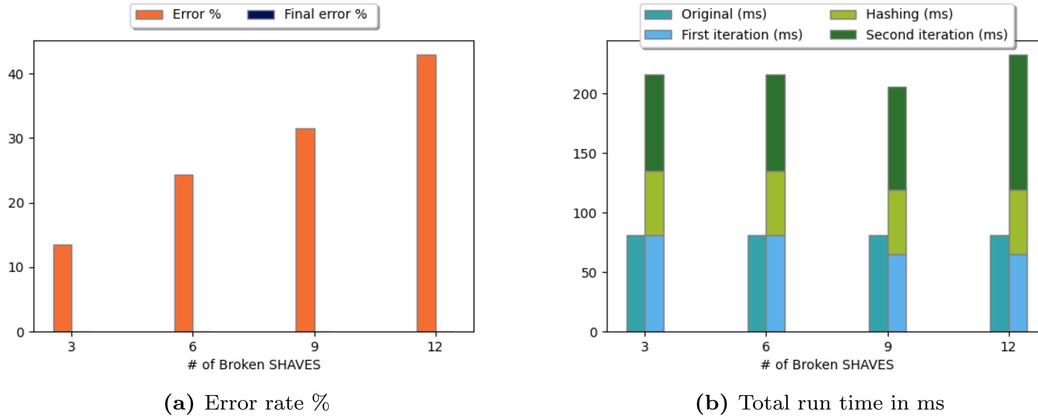
In addition, the results are also shown below in graphical form in relation to both the initial run-time and the initial error rate:

Analyzing the above tables we observe the following for both benchmarks:

- The initial data execution time of the SHAVEs is constant regardless of the SHAVEs being broken.
- Noticeable overhead in runtime is placed by the hashing of the data.
- The rerouting time is noticeably short and in the case where all SHAVEs are broken zero as in this case all SHAVEs get reset.
- The final error remaining is very small.

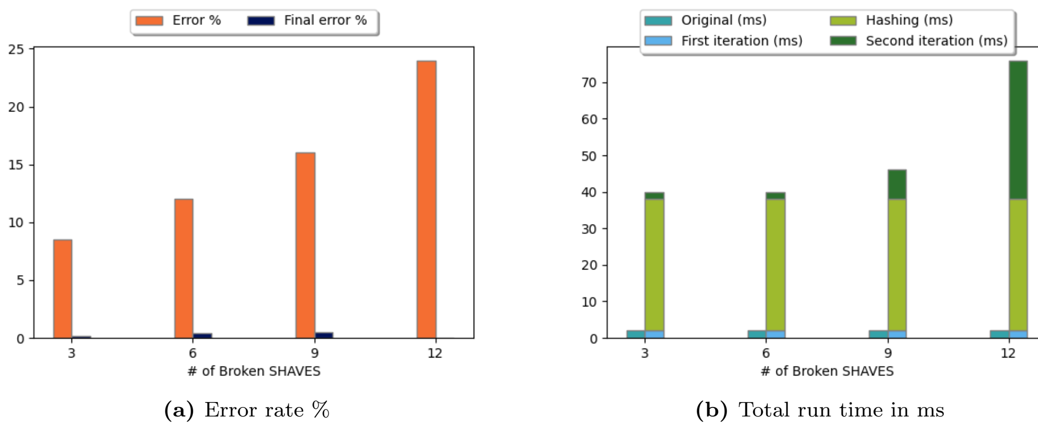
Observing the above diagrams we come to the following observations:

- The error rate is very small compared to the original error introduced.



(a) Error rate % **(b)** Total run time in ms

Figure 4.2: Leon OS corrects SHAVE instruction memory harm, results for 2D Convolution



(a) Error rate % **(b)** Total run time in ms

Figure 4.3: Leon OS corrects SHAVE instruction memory harm, results for 2D Binning

- In the case of run-time, we observe that the run-time increases as the total number of SHAVEs that are broken increases. This makes sense as the more they increase the more SHAVEs need to re-execute their workload.
- The time overhead placed by the time taken by data hashing is noticeable.
- The execution time in the case of two-dimensional convolution after applying the error correction policy is about 3 times the original execution time. Which is reasonable since correctness checks are required but also to rerun the data twice.
- The runtime in the case of 2D binning is around 25 times longer than the original. This increase is due to the fact that 2D binning is considerably faster than convolution.

Fault tolerant policy: LEON OS corrects error in data transferred to SHAVEs

In this subsection, we present the results of the experimental measurements for the error recovery policy on the data transferred by LEONOS to the SHAVE co-processors for the two benchmarks.

2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	81 ms	81 ms	81 ms	81 ms
2. Hashing data / SHAVE	7.4 ms	7.4 ms	7.4 ms	7.4 ms
3. Rescheduling run time	0.002 ms	0.003 ms	0.004 ms	0 ms
4. SHAVE rerun time	81 ms	86 ms	87 ms	81 ms
5. Total run time	252 ms	257 ms	258 ms	252 ms
8. Error percentage	0%	0.2%	0.2%	0%

Table 4.11: LEON OS corrects corrupted data transferred, results for 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	2 ms	2 ms	2 ms	2 ms
2. Hashing data / SHAVE	3.4 ms	3.4 ms	3.4 ms	3.4 ms
3. Rescheduling run time	0.002 ms	0.002 ms	0.003 ms	0 ms
4. SHAVE rerun time	8.3 ms	15.9 ms	7.7 ms	2 ms
5. Total run time	52 ms	59 ms	50 ms	44 ms
8. Error percentage	1.5%	2.6%	0.4%	0%

Table 4.12: LEON OS corrects corrupted data transferred, results for 2D Binning

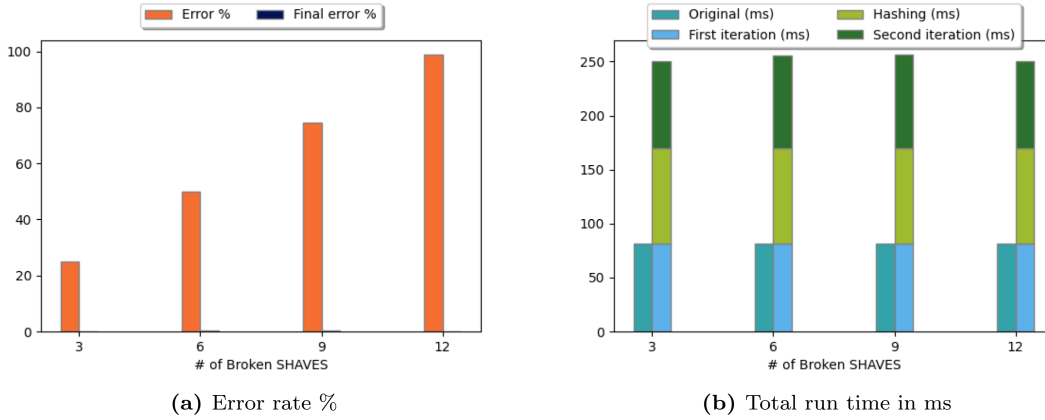
In addition, the results are also shown below in graphical form in relation to both the initial run-time and the initial error rate:

Analyzing the above tables we observe the following for both benchmarks:

- The initial data execution time of the SHAVEs is constant regardless of the SHAVEs being broken.
- Noticeable overhead in run-time is placed by the hashing of the data.
- The rescheduling time is noticeably short and in the case where all SHAVEs are broken zero as in this case all SHAVEs get reset.
- The final error remaining is very small.

Observing the above diagrams we come to the following observations:

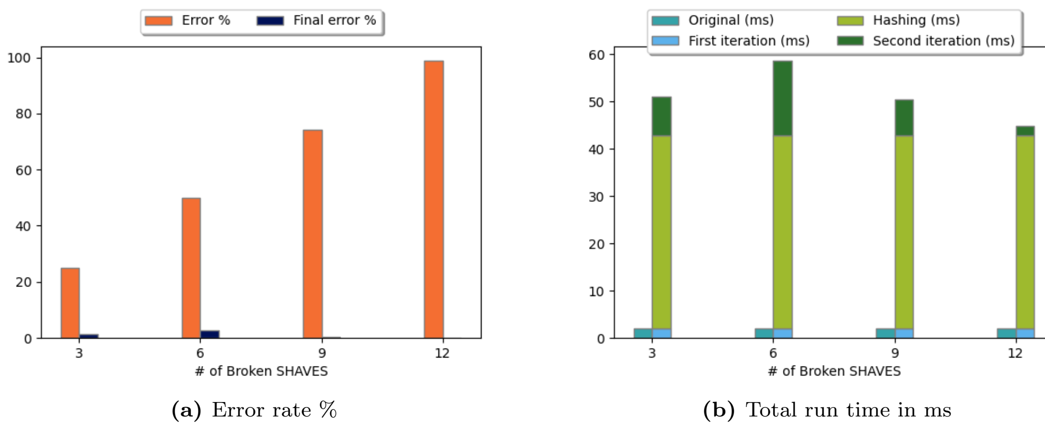
- The error rate is very small compared to the original error introduced.
- In the case of run-time, we observe that the run-time increases as the total number of SHAVE that are broken breaks down. This makes sense as the more they increase the more SHAVEs need to re-execute their workload.
- The time overhead placed by the time taken by data hashing is noticeable.



(a) Error rate %

(b) Total run time in ms

Figure 4.4: LEON OS corrects corrupted data transferred, comparative results for 2D Convolution



(a) Error rate %

(b) Total run time in ms

Figure 4.5: LEON OS corrects corrupted data transferred, comparative results for 2D Binning

- The execution time in the case of two-dimensional convolution after applying the error correction policy is about 3 times the original execution time. Which is reasonable since correctness checks are required but also to rerun the data twice.
- The run-time in the case of 2D binning is around 25 times longer than the original. This increase is due to the fact that 2D binning is considerably faster than convolution.
- Increase in rerun time as the number of SHAVEs that are broken changes is due to the fact that the more co-processors that are broken the more their workload is executed in a stream. That is, if 9 SHAVEs are broken then the functional SHAVEs available to execute their workload is 3. Therefore, the data of the broken ones should be executed in threes. Finally, because of this wait, additional delay is introduced.

Fault tolerant policy: LEON OS corrects error introduced in variables shared with SHAVEs

In this subsection, we present the results of the experimental measurements for the error recovery policy in variable joint between the LEONOS and SHAVE co-processors for the two metro-programs.

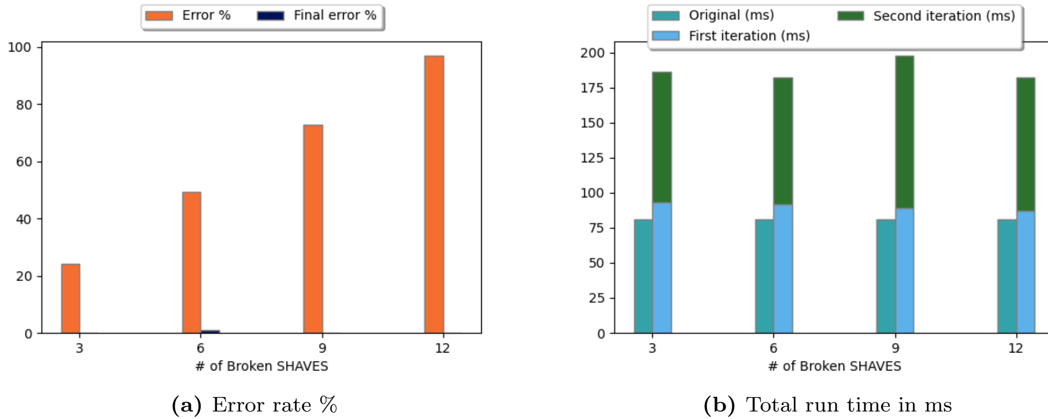
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	93 ms	92 ms	89 ms	87 ms
2. Rescheduling run time	0.002 ms	0.003 ms	0.004 ms	0 ms
3. SHAVE rerun time	93 ms	90 ms	109 ms	95 ms
4. Total run time	186 ms	182 ms	199 ms	183 ms
5. Error percentage	0%	1%	0%	0%

Table 4.13: Leon OS corrects variables shared with SHAVEs, results for 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	31 ms	33 ms	24 ms	21 ms
2. Rescheduling run time	0.001 ms	0.001 ms	0.003 ms	0 ms
3. SHAVE rerun time	27 ms	22 ms	62 ms	32 ms
4. Total run time	58 ms	55 ms	89 ms	53 ms
5. Error percentage	0%	0%	0%	0%

Table 4.14: Leon OS corrects variables shared with SHAVEs, results for 2D Binning

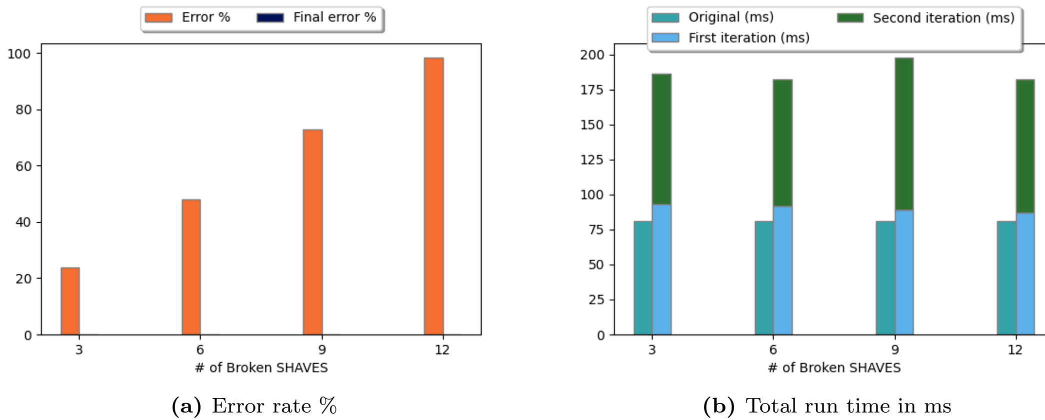
In addition, the results are also shown below in graphical form in relation to both the initial runtime and the initial error rate:



(a) Error rate %

(b) Total run time in ms

Figure 4.6: Leon OS corrects variables shared with SHAVEs, comparative results for 2D Convolution



(a) Error rate %

(b) Total run time in ms

Figure 4.7: Leon OS corrects variables shared with SHAVEs, comparative results for 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The initial execution time of the data from the SHAVEs is steadily decreasing depending on the SHAVEs that are broken.
- Now the hashing of the data is parallelized and thus does not take as long as the previous cases.
- The rerouting time is noticeably short and in the case where all SHAVEs are broken is zero as in this case all SHAVEs get reset.
- The final error remaining is marginally 0.

Observing the above diagrams we come to the following observations:

- The error rate is minimal compared to the original error introduced.
- In the case of run-time, we observe that the run-time increases as the total number of SHAVEs that are broken breaks down. This makes sense as the more they increase the more SHAVEs need to re-execute their workload.
- The time overhead placed by the time taken by data hashing is noticeable.
- The execution time in the case of two-dimensional convolution after applying the error correction policy is about 2.5 times the original execution time. Which is reasonable since correctness checks are required but also to rerun the data twice. The run-time in the case of 2D binning is around 25-30 times longer than the original. This increase is due to the fact that 2D binning is considerably faster than convolution.
- Increase in rerun time as the number of SHAVEs that are broken changes is due to the fact that the more co-processors that are broken the more their workload is executed in a stream. That is, if 9 SHAVEs are broken then the functional SHAVEs available to execute their workload is 3. Therefore, the data of the broken ones should be executed in threes. Finally, because of this wait, additional delay is introduced. An exception, is the case when all SHAVEs are broken as in this case all SHAVEs need to be restarted.

Fault tolerant policy: SHAVEs correct the variables shared with LEON OS

In this subsection, we present the results of experimental measurements on the error recovery policy on variable joint between the LEONOS and SHAVE co-processors for the two benchmarks. In this particular case, SHAVEs this corrects the error and does not notify the LEON OS.

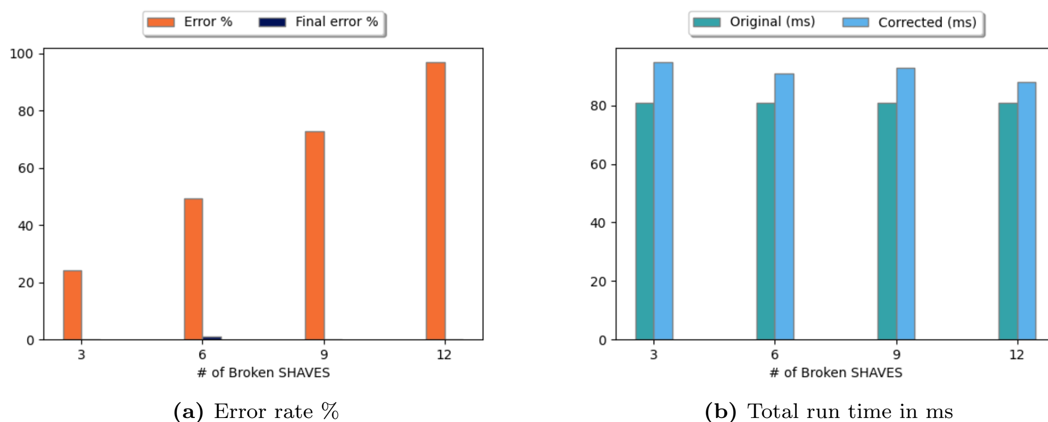
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	95 ms	91 ms	93 ms	88 ms
2. Error percentage	0%	1%	0%	0%

Table 4.15: SHAVEs correct variables shared with LEON OS, results for 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	31 ms	33 ms	24 ms	21 ms
2. Error percentage	0%	0%	0%	0%

Table 4.16: SHAVEs correct variables shared with LEON OS, results for 2D Binning

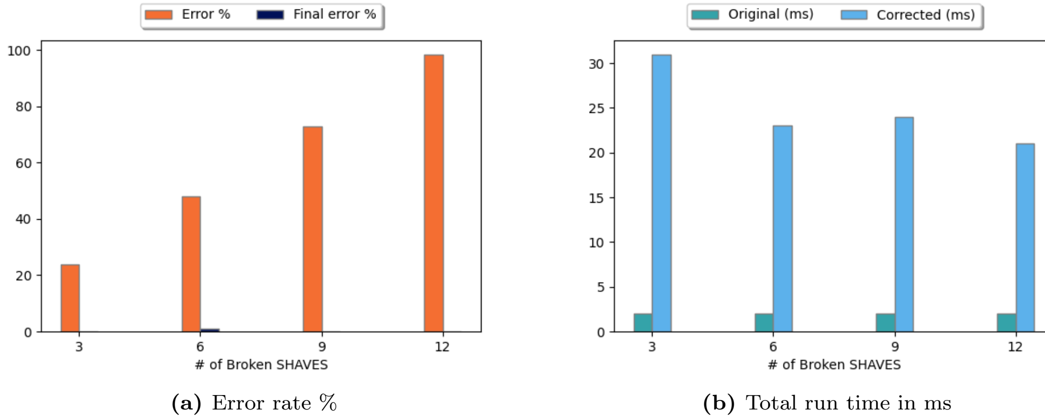
In addition, the results are also shown below in graphical form in relation to both the initial run-time and the initial error rate:



(a) Error rate %

(b) Total run time in ms

Figure 4.8: SHAVEs correct variables shared with LEON OS, comparative results for 2D Confolution



(a) Error rate %

(b) Total run time in ms

Figure 4.9: SHAVEs correct variables shared with LEON OS, comparative results for 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The initial execution time of the data from the SHAVEs is steadily decreasing depending on the SHAVEs that are broken.
- Now the hashing of the data is parallelized and thus does not take as long as the previous cases.
- The rescheduling time is now non-existent as thesSHAVEs are aware of their error and correct it themselves using a backup frame they have.
- The runtime so far is the shortest observed as no re-run is required.
- The final error that remains is marginally 0

Observing the above diagrams we come to the following observations:

- The error rate is minimal compared to the original error introduced.
- The time overhead placed by the time taken by data hashing is noticeable but smaller than the original parallelism ratio.
- The execution time in the case of two-dimensional convolution after applying the error correction policy is about 1.2 times the original execution time. This makes sense since correctness checks are required and no longer every SHAVEs needs to run its workload twice.
- The runtime in the case of 2D binning is around 15 times longer than the original. This increase is due to the fact that 2D binning is much faster than convolution and the overhead placed due to data hashing is much longer than the runtime of the benchmark itself.

Fault tolerant policy: LEON OS correct error introduced by SHAVEs themselves

In this subsection, we present the results of experimental measurements for the self-inflicted error recovery policy. In the present case, LEON OS is responsible for correcting the error after being informed of its existence by the problematic co-processors. Below are the results for both benchmarks.

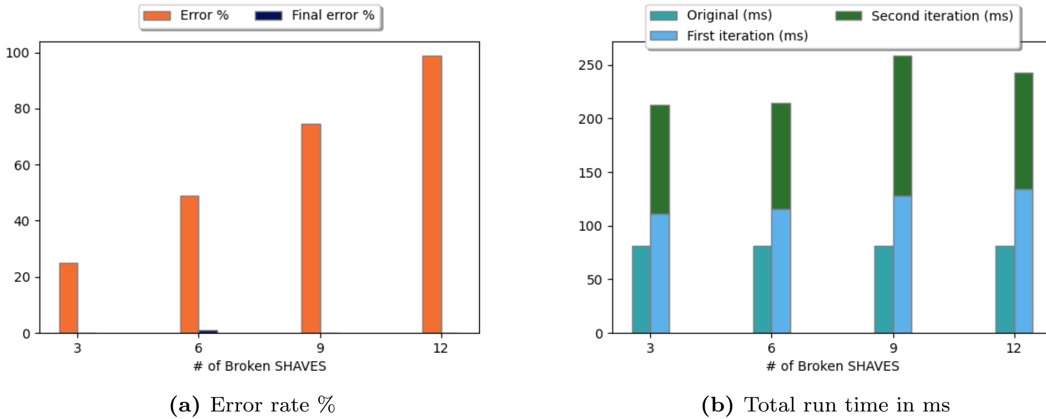
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	111 ms	116 ms	128 ms	134 ms
2. Rescheduling run time	0.002 ms	0.002 ms	0.004 ms	0 ms
3. SHAVE rerun time	102 ms	99 ms	131 ms	109 ms
4. Total run time	213 ms	215 ms	259 ms	243 ms
5. Error percentage	0%	1%	0%	0%

Table 4.17: LEON OS corrects data destroyed by SHAVEs, results for 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	81 ms	98 ms	126 ms	142 ms
2. Rescheduling run time	0.001 ms	0.002 ms	0.003 ms	0 ms
3. SHAVE rerun time	52 ms	47 ms	122 ms	62 ms
4. Total run time	134 ms	146 ms	249 ms	204 ms
5. Error percentage	0%	1%	0%	0%

Table 4.18: LEON OS corrects data destroyed by SHAVEs, results for 2D Binning

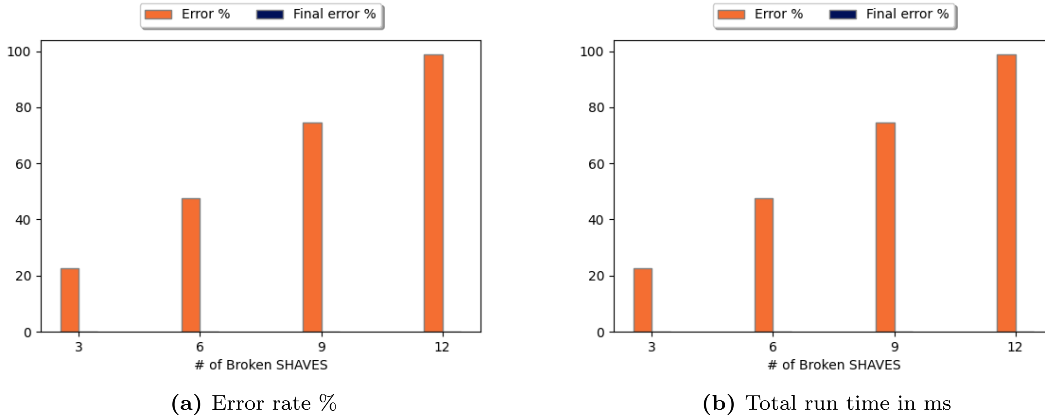
In addition, the results are also shown below in graphical form in relation to both the initial runtime and the initial error rate:



(a) Error rate %

(b) Total run time in ms

Figure 4.10: LEON OS corrects data destroyed by SHAVEs, comparative results for 2D Convolution



(a) Error rate %

(b) Total run time in ms

Figure 4.11: LEON OS corrects data destroyed by SHAVEs, comparative results for 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The initial execution time of the data from the SHAVEs is steadily increasing depending on the SHAVEs that are broken. This is due to the error insertion process.
- Now the hashing of the data is parallelized and thus does not take as long as the previous cases.
- The rerouting time is noticeably short and in the case where all SHAVEs are corrupted zero, as in this case all SHAVEs become resets.
- The final error remaining is marginally 0.

Observing the above diagrams we come to the following observations:

- The error rate is minimal compared to the original error introduced.
- In the case of run-time, we observe that the run-time increases as the total number of SHAVEs that are broken breaks down. This makes sense as the more they increase the more SHAVEs need to re-execute their workload.
- The time overhead placed by the time taken by data hashing is noticeable.
- The execution time in the case of two-dimensional convolution after applying the error correction policy is about 2.5 times the original execution time. Which is reasonable since correctness checks are required but also to rerun the data twice. execution time in the case of 2D binning is more than 30 times longer than the original. This increase is due to the fact that 2D binning is considerably faster than convolution.
- Increase in rerun time as the number of SHAVEs that are broken changes is due to the fact that the more co-processors that are broken the more their workload is executed in a stream. That is, if 9 SHAVEs are broken then the functional SHAVEs available to execute their workload is 3. Therefore, the data of the broken ones should be executed in threes. Finally, because of this wait, additional delay is introduced. An exception, is the case when all SHAVEs are broken as in this case all SHAVEs need to be restarted.

Fault tolerant policy: SHAVEs correct the error they introduced on their own.

In this subsection, we present the results of experimental measurements of the error recovery policy induced by SHAVEs on their own data for the two metroprograms. In this particular case, SHAVEs this corrects the error and does not notify the LEON OS.

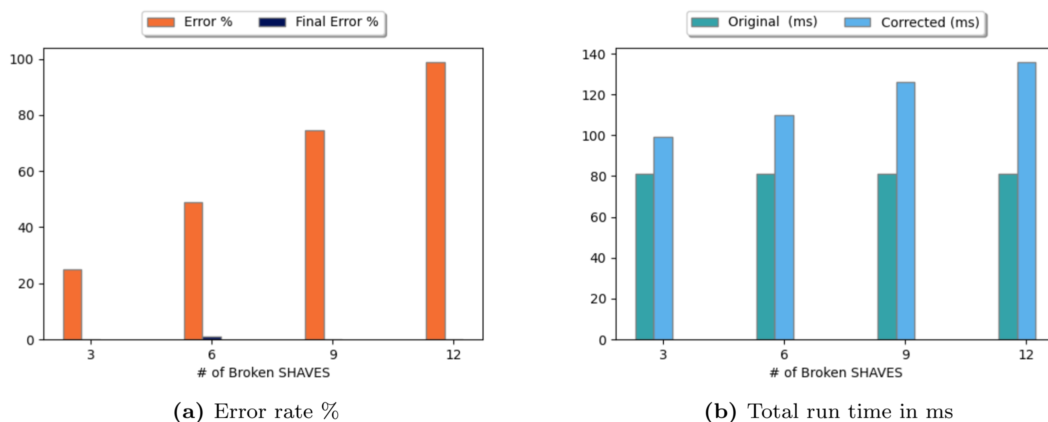
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	99 ms	110 ms	126 ms	136 ms
2. Error percentage	0%	1%	0%	0%

Table 4.19: SHAVEs correct data destroyed by SHAVEs, results for 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	52 ms	62 ms	76 ms	85 ms
2. Error percentage	0%	0%	0%	0%

Table 4.20: SHAVEs correct data destroyed by SHAVEs, results for 2D Binning

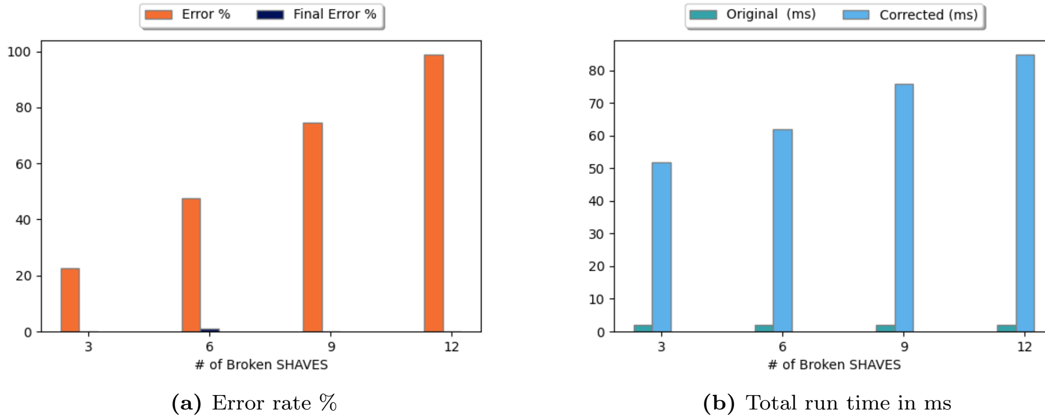
In addition, the results are also shown below in graphical form in relation to both the initial run-time and the initial error rate:



(a) Error rate %

(b) Total run time in ms

Figure 4.12: SHAVEs correct data destroyed by SHAVEs, comparative results for 2D Convolution



(a) Error rate %

(b) Total run time in ms

Figure 4.13: SHAVEs correct data destroyed by SHAVEs, comparative results for 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The initial execution time of the data from the SHAVEs is steadily increasing depending on the SHAVEs that are broken. This increase is due to the introduction of the error.
- Now the hashing of the data is parallelized and thus does not take as much time as the previous cases.
- The rerouting time is now non-existent as the SHAVEs are aware of their error and correct it themselves using a backup frame they have.
- The run-time so far is the second shortest observed as no re-run is required.
- The final error that remains is marginally 0%.

Observing the above diagrams we come to the following observations:

- The error rate is minimal compared to the original error introduced.
- The time overhead placed by the time taken by data hashing is noticeable but less than the original parallelism ratio.
- The execution time in the case of two-dimensional convolution after applying the error correction policy is about 1.5 times the original execution time. This makes sense since correctness checks are required and no longer every SHAVE needs to run its workload twice.
- The run-time in the case of 2D binning is more than 30 times longer than the original. This increase is due to the fact that 2D binning is much faster than convolution, and the overhead placed reason data hashing is much longer than the run-time of the benchmark itself. In addition, there is a time overhead for introducing the error leading to increased run-time.

Fault tolerant policy: 3 Voting System, LEON OS corrects error in SHAVEs

In this subsection, we present the results of experimental measurements for the policy of recovering from erroneous data transfer from LEON OS to SHAVEs for both benchmarks. In this particular case the system used for the annealing is a three-voter system.

Below are the relevant tables of results:

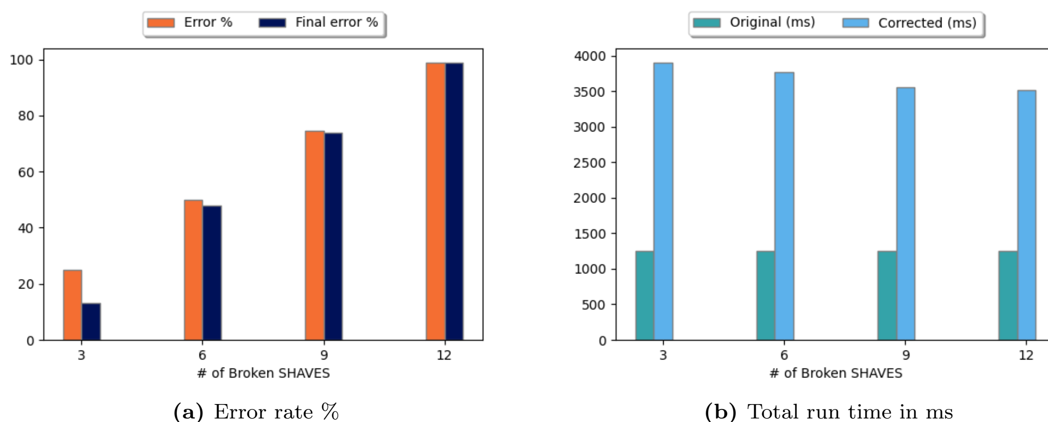
2D Convolution				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	3908 ms	3765 ms	3559 ms	3518 ms
2. Error percentage	13%	48%	74%	99%

Table 4.21: 3 Voting system, results for 2D Convolution

2D Binning				
# of Broken SHAVES	3	6	9	12
1. Initial SHAVE run time	15 ms	14 ms	14 ms	14 ms
2. Error percentage	11%	48%	69%	99%

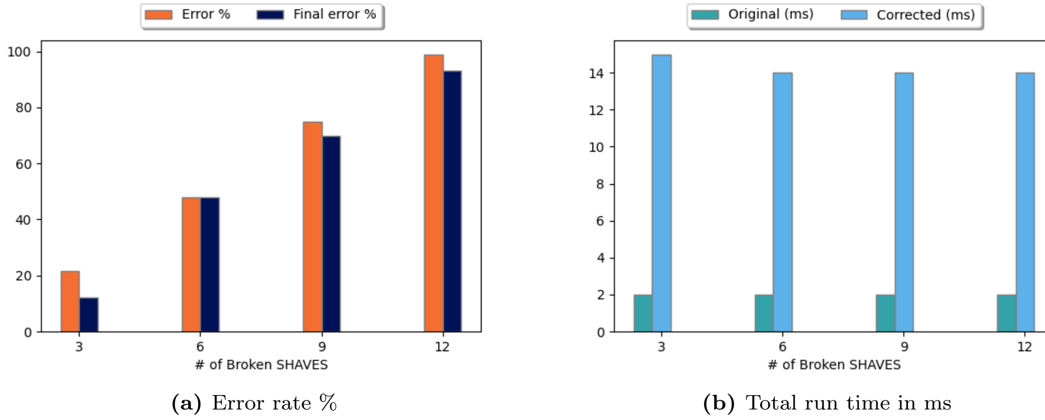
Table 4.22: 3 Voting system, results for 2D Binning

In addition, the results in relation to both the initial run-time and the initial error rate are shown below:



(a) Error rate % **(b)** Total run time in ms

Figure 4.14: 3 Voting system, comparative results for 2D Convolution



(a) Error rate %

(b) Total run time in ms

Figure 4.15: 3 Voting system, comparative results for 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The execution time is relatively constant.
- No more hashing is performed.
- The rerouting time is now non-existent as there is no re-execution of workload. In this case, redundancy is simply introduced in order to compare and recalculate results.
- The noticeable increase in time is due to the fact that we have not enabled DMA for data transfer to and from CMX.
- The final error is noticeable.
- The error is due to the fact that if more than half of the SHAVEs are corrupted then at least half of the ballots will return incorrect results.

Observing the above charts we come to the following observations:

- The error rate is comparable to the original error introduced.
- The time overhead is due to resource under-utilization.
- The run-time in the case of 2D convolution and 2D binning after applying the error correction policy is about 4 times the original run-time. Which is reasonable since SHAVEs execute the same load in groups of three.
- In the case of 2D binning the new execution time is better than the time of the methods that required data hashing as the time required for data hashing was significantly longer than the execution time of the benchmark under-utilizing the available resources. In contrast, in the case of convolution where the time of data hashing is comparable the execution time is significantly worse than the previous methods.

Fault tolerant policy: 5 Voting System

In this subsection, we present the results of experimental measurements for the policy of recovering from erroneous data transfer from LEON OS to SHAVEs for both metroprograms. In this particular case, the system used for the recovery is a five-voter system.

Below are the relevant tables of results:

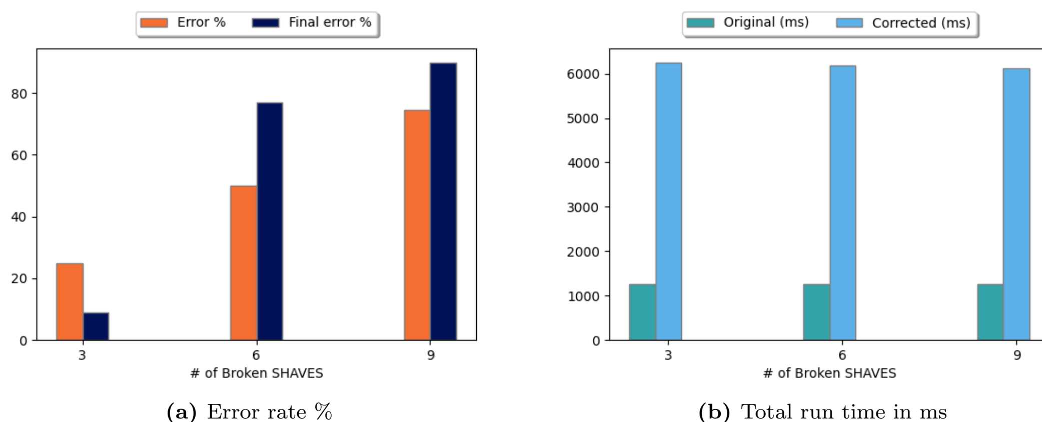
2D Convolution			
# of Broken SHAVES	3	6	9
1. Initial SHAVE run time	6257 ms	6185 ms	6130 ms
2. Error percentage	9%	77%	99%

Table 4.23: 5 Voting system, results of 2D Convolution

2D Binning			
# of Broken SHAVES	3	6	9
1. Initial SHAVE run time	28 ms	28 ms	28 ms
2. Error percentage	10%	54%	88%

Table 4.24: 5 Voting system, results of 2D Binning

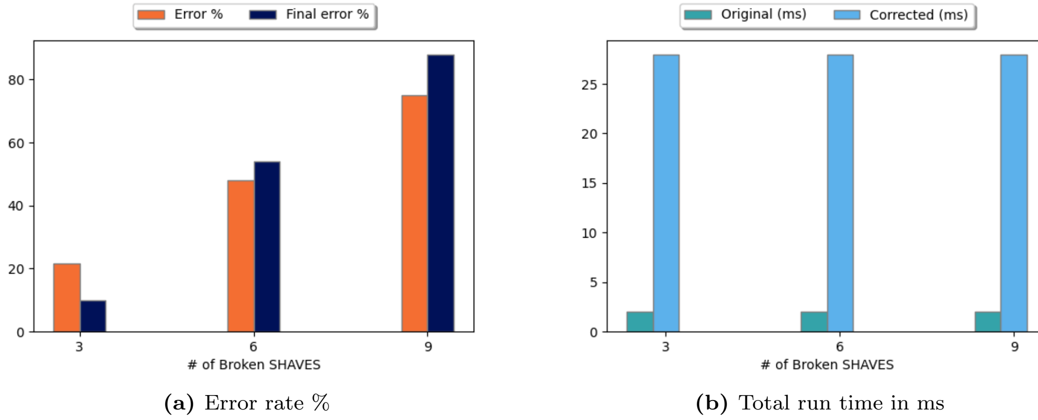
In addition, the results in relation to both the initial run-time and the initial error rate are shown below:



(a) Error rate %

(b) Total run time in ms

Figure 4.16: 5 Voting system, comparative results for 2D Convolution



(a) Error rate %

(b) Total run time in ms

Figure 4.17: 5 Voting system, comparative results for 2D Binning

Analyzing the above tables we observe the following for both benchmarks:

- The execution time is relatively constant.
- No more hashing is performed.
- The rerouting time is now non-existent as there is no re-execution of workload. In this case, redundancy is simply introduced in order to compare and recalculate results.
- The noticeable increase in time is due to the fact that we have not enabled DMA for data transfer to and from CMX.
- The final error is noticeable.
- The error is due to the fact that if more than half of the SHAVEs are corrupted then at least half of the ballots will return incorrect results.
- Two SHAVEs are not used at all.

Observing the above diagrams, we come to the following observations:

- The error rate is comparable to the original error introduced.
- The time overhead is due to under-utilization of SHAVEs.
- The run-time in the case of 2D convolution and 2D binning after applying the error correction policy is about 8 times the original run-time. Which makes sense since SHAVEs in groups of 5 execute the same load.
- In the case of 2D binning the new execution time is better than the time of the methods that required data hashing as the time required for data hashing was significantly longer than the execution time of the benchmark under-utilizing the available resources. In contrast, in the case of convolution where the time of data hashing is comparable the execution time is significantly worse than the previous methods.

Chapter 5

Conclusion and Future Work

Conclusion

In this thesis, we proposed multiple techniques to increase system reliability in case of failure on the Intel Movidius Myriad 2 V.P.U. These solutions, in their core introduced redundancy to ensure the correctness of the system output. These methods where then tested by using the error injection methods we also created. Lastly, we used two programs that are commonly encountered in computer vision pipelines to evaluate the performance of the proposed policies. These programs are 2D convolution which is used in computer vision to apply multiple classical filters and 2D binning which is a method for image sub-sampling

Observing our results, we can easily observe that the time overhead introduced by the plethora of checks needed to assert whether there is error is corrected is significant. However, the final output in most cases does not include any significant error rate after the policy has been applied. Furthermore, the time overhead introduce in the case of the 2D convolution is not that significant since the time overhead introduced by the error checking methods is of equal magnitude to the original run time. However, in the case of 2D image binning the time overhead introduce by error checking and rerunning is immense since this is a relatively fast benchmark, in one case a police resulted in a run time 50 times larger than the original one. Apart from the methods that use cyclic redundancy checks to asses whether there is error in the system we also developed some voting systems which group the SHAVES and forward the result that has been voting by the majority of the group. These policies, in the case of 2D binning despite the resource under utilization result in a final run time 4 to 8 times the original one which is acceptable. However, when it comes to the two dimensional convolution the final time overhead is significant especially when compared to the previous methods reaching in the case of the 5 voting system a run time 8 times the original one. Furthermore, due to their nature voting systems do not assert 0% error rate once they are applied. Therefore, if more than half the SHAVEs are broken the final output can have up to 99% of error.

Concluding, based on the observations we made from our experiments there is no "best" method and how appropriate a method is heavily depends on the quality of service we want our system to provide. Therefore, if we can accept a time overhead we can use the methods that use C.R.C to evaluate the correctness of the system. However, if we cannot support such a high run-time we could opt for a voting system to correct a subset off errors and not all of them.

Future Work

As this thesis reaches it's end the author has to propose the following future work:

- In this thesis all the fault tolerant policies do not run on run-time. Therefore, there is still potential to expand the methodologies proposed by this thesis in a complete fault tolerant framework that will allow the developer to quickly deploy them in real life scenarios.
- In the case of voting systems there is a sever resource under utilization in order to avoid cases where no consensus can be reached. Therefore, in these scenarios there is future work that could propose different ways to use the underutilized SHAVEs (e.g. the 2 co processors left unused in the case of the 5 voting system).
- In this thesis a simple C.R.C function was used. However, there might be other C.R.C functions that could reduce the overhead introduced by the cyclic redundancy checks.
- Some fault tolerant policies that are commonly found in bibliography have not been explored in this thesis. Some of them are, watchdog timers, lockstep system implementation e.t.c.
- In the case of voting systems we manually check the contents returned from each SHAVE to evaluate whether two SHAVEs have returned the same output. However, there is still room for improvement like using heuristic functions to deduce up to what point it is mandatory to check with a good probability of having a proper output.
- There are some subsystems that could be used to introduce extra reliability such as using LEON RT as a backup to LEON OS in case of system failure. Furthermore, we could use LEON OS not only as the orchestrator of the entire process but also as part of the fault tolerant policy introducing more safety mechanisms.
- Create an error injection tool that will be used to test developed software in Myriad 2.

Bibliography

[1] V. Leon, G. Lentaris, E. Petrongonas, D. Soudris, G. Furano, A. Tavoularis, and D. Moloney, "Improving Performance-Power-Programmability in Space Avionics with Edge Devices: VBN on Myriad2 SoC," *ACM Transactions on Embedded Computing Systems*, vol. 20, pp. 1–23, 03 2021.

[2] W. N. Toy, "Fault-tolerant computing," in *Advances in Computers*, vol. 26, 1987, pp. 201–279.

[3] A. Tom and R. Felix, *Safety-Critical Systems: The Convergence of High Tech and Human Factors*. Springer, 1996.

[4] B. Alfredo and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer, 2003.

[5] Y. Crouzet and K. Kanoun, "Chapter 3 - system dependability: Characterization and benchmarking," in *Dependable and Secure Systems Engineering*, ser. Advances in Computers, A. Hurson and S. Sedigh, Eds. Elsevier, 2012, vol. 84, pp. 93–139.

[6] L. Feinbube, L. Pirl, and A. Polze, "Software fault injection: A practical perspective," in *Dependability Engineering*, F. P. G. Márquez and M. Papaelias, Eds. Rijeka: IntechOpen, 2018, ch. 4. [Online]. Available: https://doi.org/10.5772/intechopen.70427

[7] E. Petrongonas, V. Leon, G. Lentaris, and D. Soudris, "ParalOS: A Scheduling & Memory Management Framework for Heterogeneous VPUs," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 221–228.

[8] I. Movidius, "Intel acquires movidius," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/movidius-vpu/myriad-x-product-brief.html

[9] cucumber, "Happy unhappy paths: Why you need to test both," 2019. [Online]. Available: https://cucumber.io/blog/test-automation/happy-unhappy-paths-why-you-need-to-test-both/

[10] D. Trawczynski and J. Sosnowski, "Fault injection testing of safety-critical applications," *Technical Transactions in Computer Sc. and Inf. Systems*, 01 2011.

[11] "Index," in *Essentials of Error-Control Coding Techniques*, H. Imai, Ed. Academic Press, 1990, pp. 327–337. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123707208500131

[12] J. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022000079900448

[13] P. D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, 2009, pp. 379–388.

[14] T. Tsai and R. Iyer, "Ftape: A fault injection tool to measure fault tolerance," 1995, pp. 339–346, funding Information: part by the Advanced Research Projects Agency (AR.PA) under contract DABT63-94-C-0045 and by NASA grant NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). The content of this paper does not necessarily reflect the position or policy of the government and no endoresement should be inferred. Publisher Copyright: © 1995, American Institute of Aeronautics and Astronautics Inc, AIAA. All rights reserved.; 10th Computing in Aerospace Conference, 1995 ; Conference date: 28-03-1995 Through 30-03-1995.

[15] D. Costa, H. Madeira, J. Carreira, and J. G. Silva, *Xception™: A Software Implemented Fault Injection Tool*. Boston, MA: Springer US, 2003, pp. 125–139. [Online]. Available: https://doi.org/10.1007/0-306-48711-X_8

[16] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[17] crunchbase, "Intel acquires movidius," 2016. [Online]. Available: https://www.crunchbase.com/acquisition/intel-acquires-movidius--ece69af0

[18] D. Jasnaik, "Fault-tolerance techniques in embedded system." 10 2014.

[19] F. Afonso, C. Silva, A. Tavares, and S. Montenegro, "Application-level fault tolerance in real-time embedded systems," in *2008 International Symposium on Industrial Embedded Systems*, 2008, pp. 126–133.

[20] I. Schagaev, E. Zouev, and K. Thomas, *Fault Tolerance: Theory and Concepts*, 01 2020, pp. 11–23.

[21] Z. Wang and N. Minsky, "Fault tolerance in heterogeneous distributed systems," 01 2014.

[22] M.-R. Motlagh, B. Kia, W. Ditto, and S. Sinha, "Fault tolerance and detection in chaotic computers." *I. J. Bifurcation and Chaos*, vol. 17, pp. 1955–1968, 06 2007.

[23] M. Stanisavljevic, A. Schmid, and Y. Leblebici, *Reliability, Faults, and Fault Tolerance*, 09 2011.

[24] D. Shah and A. Aslekar, "Iot in fault detection and prediction," 03 2022, pp. 1028–1033.

[25] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, pp. 171–186, 01 2004.

[26] R. Lenka, S. Padhi, and K. Nayak, "Fault injection techniques - a brief review," 10 2018, pp. 832–837.

[27] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75 – 82, 05 1997.

[28] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," 05 2013, pp. 1–10.

[29] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," 04 2009, pp. 502–506.

[30] M. Heing-Becker, T. Kamph, and S. Schupp, "Bit-error injection for software developers," 02 2014, pp. 434–439.

[31] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An empirical study of injected versus actual interface errors," 07 2014.

[32] D. Cotroneo and R. Natella, "Fault injection for software certification," *IEEE Security and Privacy Magazine*, vol. 11, pp. 38–45, 07 2013.

[33] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe, "Myriad 2: Eye of the computational vision storm," 08 2014, pp. 1–18.

[34] K.-Y. Fung and B. Goble, "Computation with error injection," *Mathematical and Computer Modelling - MATH COMPUT MODELLING*, vol. 11, pp. 1116–1121, 12 1988.

[35] C. Cheng, "Memory error injection mechanism," pp. 1070–1072, 08 2004.

[36] R. Velazco, "Method for error injection by interruptions," 07 2002.

[37] L. Zhang, B. Morin, B. Baudry, and M. Monperrus, "Maximizing error injection realism for chaos engineering with system calls," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, pp. 1–1, 03 2021.

[38] S. Mirkhani, H. Cho, S. Mitra, and J. Abraham, "Rethinking error injection for effective resilience," 01 2014, pp. 390–393.

[39] L. Wang, Z. Jia, S. Li, Y. Zhang, T. Wang, E. Xu, J. Wang, and X.-K. Liao, "Challenges and opportunities: an in-depth empirical study on configuration error injection testing," 07 2021, pp. 478–490.

[40] C.-K. Chang, S. Lym, N. Kelly, M. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for hpc," 11 2018, pp. 577–589.

[41] S. Jean, "Systems and methods for error injection in data storage systems," Patent, 04, 2014.

[42] V. Leon, C. Bezaitis, G. Lentaris, D. Soudris, E.-A. Papatheofanous, A. Kyriakos, A. Dunne, A. Samuelsson, and D. Steenari, "FPGA & VPU Co-Processing in Space Applications: Development and Testing with DSP/AI Benchmarks," in *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2021, pp. 1–5.