ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αξιοποίηση κωδίκων αποκατάστασης σφαλμάτων σε κατανεμημένα συστήματα αποθήκευσης υψηλών επιδόσεων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Μπαρμπαρής

Επιβλέπων: Νεκτάριος Κοζύρης,
Αν. Καθηγητής ΕΜΠ

Αθήνα, Ιανουάριος 2011

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αξιοποίηση κωδίκων αποκατάστασης σφαλμάτων σε κατανεμημένα συστήματα αποθήκευσης υψηλών επιδόσεων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Μπαρμπαρής

Επιβλέπων: Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Φεβρουαρίου 2011.

………………………..	………………………..	………………………..
Ν. Κοζύρης	Π. Τσανάκας	Ν. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2011

……………………………….

Κωνσταντίνος Μπαρμπαρής
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Copyright © Κωνσταντίνος Μπαρμπαρής, 2010
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμηματικά 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 το συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν το συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σκοπός της παρούσας εργασίας είναι ο σχεδιασμός και η υλοποίηση ενός RAID συστήματος (RAID EC), το οποίο θα αξιοποιεί τη θεωρία των κωδίκων αποκατάστασης σφαλμάτων, και πιο συγκεκριμένα του Erasure-Code, με σκοπό την επίτευξη της βέλτιστης ισορροπίας μεταξύ ταχύτητας απόκρισης και προστασίας των δεδομένων σε ένα κατανεμημένο αποθηκευτικό σύστημα. Ένα τέτοιο RAID σύστημα μπορεί να υποστηρίζει m πλεονάζοντες δίσκους, ώστε η ζητούμενη πληροφορία να αποθηκεύεται σε n+m δίσκους και να είναι δυνατή η ανάκτησή της από οποιουσδήποτε n δίσκους. Για το σχεδιασμό και την υλοποίηση του συστήματος RAID EC χρησιμοποιήθηκε η κωδικοποίηση των Reed-Solomon από τους κώδικες αποκατάστασης σφαλμάτων Erasure Code και οι μαθηματικές θεωρίες των σωμάτων Γκαλουά (Galois fields).

Το RAID EC αποτελεί ουσιαστικά μια επέκταση των RAID 5 και RAID 6, η οποία προσφέρει μεγαλύτερη ασφάλεια στα δεδομένα προς αποθήκευση, αλλά και παραπλήσιους χρόνους απόκρισης επιλέγοντας τον κατάλληλο τρόπο υλοποίησης. Το RAID EC, ως ένα σύστημα που μπορεί να διαχειριστεί ένα μεγάλο όγκο δεδομένων εξασφαλίζοντας την ασφάλεια, αλλά και την αποδοτική διαχείρισή του, μπορεί να βρει εφαρμογή σε πολλά συστήματα με αυξημένες απαιτήσεις ως προς τη διαχείριση των δεδομένων τους, όπως σε server web εφαρμογών ή κατανεμημένα υπολογιστικά συστήματα.

Λέξεις-Κλειδιά: Κατανεμημένα συστήματα αποθήκευσης υψηλών επιδόσεων,Erasure Code, RAID, Reed – Solomon, Σώματα Γκαλουά, Πλεονάζουσα Πληροφορία Ισοτιμίας, Μέθοδος Απαλοιφής Gauss, Αποθηκευτικά Συστήματα

Abstract

The purpose of this study is to design and implement a RAID system (RAID EC), which utilizes the theory of error recovery code, and particularly Erasure-Code, in order to achieve the optimal balance between performance and data protection in an expanded storage system. Such a RAID system can support m redundant disks; as a result the requested information is stored on n+m disks, while it can be recovered using any n disks of the n+m disks. The Reed-Solomon error-correction code, from the optimal Erasure Codes, along with the mathematical theories of Galois fields have been used in order the RAID EC system to be designed and implemented.

The RAID EC is practically an extension of RAID 5 and RAID 6, which provides better protection for the system's data, while by choosing the appropriate implementation method it can achieve similar, if not better, response times than RAID 5 and RAID 6. The RAID EC as a system that can ensure the management of a large volume of data in a most efficient and safe way, could find application in many systems with increased demands on information storage, such as a web applications server or high-end computer systems.

Keywords: High-performance distributed storage system, Erasure Code, RAID, Reed – Solomon, Galois fields, Parity, Gauss Elimination, Storage Systems

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου υπό την επίβλεψη του Αναπληρωτή Καθηγητή Νεκτάριου Κοζύρη.

Θα ήθελα να ευχαριστήσω θερμά τον κύριο Κοζύρη για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο τομέα της επιστήμης των υπολογιστών στο Εργαστήριο Υπολογιστικών Συστημάτων, καθώς και για τις πολύτιμες συμβουλές του καθ' όλη τη διάρκεια περάτωσης της εργασίας.

Η διπλωματική αυτή αποτελεί έμπνευση του διδακτορικού Αντώνη Χαζάπη και του υποψήφιου διδακτορικού Αναστάσιου Νάνου. Η βοήθεια τους ήταν ανεκτίμητη τόσο για τις καίριες συμβουλές και υποδείξεις τους στο σχεδιασμό και την υλοποίηση του Erasure Code για RAID συστήματα, όσο και για τις γνώσεις που μου μετέδωσαν. Χωρίς τη συμβολή τους η διπλωματική εργασία δε θα ήταν εφικτή.

Επιπρόσθετα, θα ήθελα να ευχαριστήσω και όλα τα μέλη του εργαστηρίου Υπολογιστικών Συστημάτων για τις πολύτιμες συμβουλές τους.

Περιεχόμενα

Περίληψη........5
Abstract........7
Ευχαριστίες........9
Κατάλογος Πινάκων........12
Κατάλογος Σχημάτων........13
Κεφάλαιο 1........15
Εισαγωγή........15
1.1 Σκοπός........16
1.2 Η δομή του κειμένου........17
Κεφάλαιο 2........18
Θεωρητικό Υπόβαθρο........18
2.1 Αρχιτεκτονικές RAID........18
2.1.1 RAID Level 5........19
2.1.2 RAID Level 6........27
2.2 Erasure Code........36
2.2.1 Reed-Solomon Κωδικοποίηση........37
2.2.2 Σώματα Galois........39
2.2.3 Μέθοδος απαλοιφής Gauss........43
Κεφάλαιο 3........46
Σχεδιασμός........46
3.1 Εφαρμογή Erasure Code σε RAID συστήματα........47
3.1.1 Εφαρμογή των σωμάτων Galois στα RAID EC........48
3.1.2 Μεθοδολογία κατασκευής του πίνακα B για την κωδικοποίηση Reed-Solomon........51
3.1.3 Μεθοδολογία ανάκτησης χαμένων δίσκων δεδομένων.........52
3.2 Αρχιτεκτονική του RAID EC........53
3.2.1 Ανάγνωση δεδομένων από το RAID EC (Read)........56
3.2.2 Εγγραφή δεδομένων στο RAID EC (Write)........58
3.2.3 Υποβάθμιση του RAID EC (Degrade)........62
3.2.4 Αποκατάσταση του RAID EC (Recover)........63
Κεφάλαιο 4........65

Υλοποίηση 65
4.1 Το vRAID και η σχέση του με την υλοποίηση του RAID EC..66
4.2 Υλοποίηση των βασικών διεργασιών του Erasure Code..73
4.2.1 Υλοποίηση των σωμάτων Galois 74
4.2.2 Υλοποίηση του πίνακα B 78
4.2.3 Υλοποίηση της μεθόδου απαλοιφής Gauss 80
4.3 Υλοποίηση του RAID EC 84
4.3.1 Αρχικοποίηση του συστήματος RAID EC..86
4.3.2 Υλοποίηση της διεργασίας read...88
4.3.3 Υλοποίηση της διεργασίας write 104
4.3.4 Υλοποίηση της διεργασίας degrade 125
4.3.5 Υλοποίηση της διεργασίας recover 127
Κεφάλαιο 5..132
Αξιολόγηση και Εφαρμογές..132
5.1 Αξιολόγηση του συστήματος RAID EC 133
5.2 Επεκτάσεις και Εφαρμογές 140
Βιβλιογραφία 143

Κατάλογος Πινάκων

ΠΙΝΑΚΑΣ 1: ΚΑΤΑΝΟΜΗ PARITIES ΣΤΟ RAID EC (N=5, M=8)....................................54
ΠΙΝΑΚΑΣ 2: ΧΡΟΝΟΣ ΑΠΟΚΡΙΣΗΣ ΤΩΝ READ ΚΑΙ WRITE ΓΙΑ ERASURE-CODE......134
ΠΙΝΑΚΑΣ 3: ΧΡΟΝΟΣ ΑΠΟΚΡΙΣΗΣ ΤΩΝ READ ΚΑΙ WRITE ΓΙΑ RAID-5........................134
ΠΙΝΑΚΑΣ 4: ΧΡΟΝΟΣ RECOVER ΓΙΑ ERASURE-CODE 137

Κατάλογος Σχημάτων

ΣΧΗΜΑ 1: ΚΑΤΑΝΟΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ PARITY ΓΙΑ RAID 5....21
ΣΧΗΜΑ 2: ΑΝΤΙΣΤΟΙΧΙΑ DAG ΜΕ ΠΕΡΙΠΤΩΣΕΙΣ READ ΚΑΙ WRITE ΓΙΑ RAID 5....22
ΣΧΗΜΑ 3: DAG: NONREDUNDANT READ....22
ΣΧΗΜΑ 4: DAG: DEGREADED-MODE READ....23
ΣΧΗΜΑ 5: DAG: SMALL-WRITE GRAPH....24
ΣΧΗΜΑ 6: DAG: RECONSTRUCT-WRITE GRAPH....25
ΣΧΗΜΑ 7: DAG: LARGE-WRITE GRAPH....26
ΣΧΗΜΑ 8: DAG: NONREDUNDANT WRITE....26
ΣΧΗΜΑ 9: ΚΑΤΑΝΟΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ PARITY ΓΙΑ RAID 5....28
ΣΧΗΜΑ 10: ΑΝΤΙΣΤΟΙΧΙΑ DAG ΜΕ ΠΕΡΙΠΤΩΣΕΙΣ READ ΚΑΙ WRITE ΓΙΑ RAID 6 (ΜΕΡΟΣ Α)....29
ΣΧΗΜΑ 11: ΑΝΤΙΣΤΟΙΧΙΑ DAG ΜΕ ΠΕΡΙΠΤΩΣΕΙΣ READ ΚΑΙ WRITE ΓΙΑ RAID 6 (ΜΕΡΟΣ Β)....29
ΣΧΗΜΑ 12: DAG: PQ DOUBLE-DEGRADED-READ....31
ΣΧΗΜΑ 13: DAG: PQ DEGRADED-DP-READ....31
ΣΧΗΜΑ 14: DAG: PQ SMALL-WRITE....32
ΣΧΗΜΑ 15: DAG: PQ RECONSTRUCT-WRITE....33
ΣΧΗΜΑ 16: DAG: PQ LARGE-WRITE....34
ΣΧΗΜΑ 17: DAG: PQ DOUBLE-DEGRADED-WRITE....35
ΣΧΗΜΑ 18: VENDERMONDE ΠΙΝΑΚΑΣ....37
ΣΧΗΜΑ 19: ΜΗ-ΜΗΔΕΝΙΚΑ ΣΤΟΙΧΕΙΑ ΤΟΥ GF(16)....43
ΣΧΗΜΑ 20: ΔΙΑΓΡΑΜΜΑ ΡΟΗ RAID EC....56
ΣΧΗΜΑ 21: ΔΙΑΓΡΑΜΜΑ ΡΟΗ ΤΗΣ ΔΙΕΡΓΑΣΙΑΣ READ....57
ΣΧΗΜΑ 22: ΔΙΑΓΡΑΜΜΑ ΡΟΗΣ ΔΙΕΡΓΑΣΙΑΣ WRITE (ΟΠΟΥ NWD_OK= TRUE, ΕΑΝ ΟΙ ΔΙΣΚΟΙ ΔΕΔΟΜΕΝΩΝ ΠΟΥ ΔΕΝ ΕΙΝΑΙ ΠΡΟΣ ΕΓΓΡΑΦΗ ΕΙΝΑΙ ΔΙΑΘΕΣΙΜΟΙ, WD_OK= TRUE, ΕΑΝ ΟΙ ΔΙΣΚΟΙ ΔΕΔΟΜΕΝΩΝ ΠΟΥ ΕΙΝΑΙ ΠΡΟΣ ΕΓΓΡΑΦΗ ΕΙΝΑΙ ΔΙΑΘΕΣΙΜΟΙ ΚΑΙ LHR=TRUE, ΕΑΝ ΟΙ ΔΙΣΚΟΙ ΔΕΔΟΜΕΝΩΝ ΠΡΟΣ ΕΓΓΡΑΦΗΣ ΕΙΝΑΙ ΠΕΡΙΣΣΟΤΕΡΟΙ ΑΠΟ $n/2$)....59
ΣΧΗΜΑ 23: ΔΙΑΓΡΑΜΜΑ ΡΟΗΣ ΔΙΕΡΓΑΣΙΑΣ DEGRADE....62
ΣΧΗΜΑ 24: ΔΙΑΓΡΑΜΜΑ ΡΟΗΣ ΔΙΕΡΓΑΣΙΑΣ RECOVER....63
ΣΧΗΜΑ 25: ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΤΟΥ VRAID....67
ΣΧΗΜΑ 26: Ο ΧΡΟΝΟΣ ΑΠΟΚΡΙΣΗΣ (ΑΝΑΓΟΜΕΝΟΣ ΣΤΟ ΜΙΚΡΟΤΕΡΟ ΧΡΟΝΟ ΤΟΥ ERASURE-CODE READ) ΤΟΥ READ ΚΑΙ WRITE ΓΙΑ ERASURE-CODE ΚΑΙ RAID-5 ΣΕ ΑΡΧΕΙΟ ΤΩΝ 50 MB....135
ΣΧΗΜΑ 27: Ο ΧΡΟΝΟΣ ΑΠΟΚΡΙΣΗΣ (ΑΝΑΓΟΜΕΝΟΣ ΣΤΟ ΜΙΚΡΟΤΕΡΟ ΧΡΟΝΟ ΤΟΥ ERASURE-CODE READ) ΤΟΥ READ ΚΑΙ WRITE ΓΙΑ ERASURE-CODE ΚΑΙ RAID-5 ΣΕ ΑΡΧΕΙΟ ΤΩΝ 100 MB....135
ΣΧΗΜΑ 28: Ο ΛΟΓΟΣ ΤΟΥ ΧΡΟΝΟΥ ΑΠΟΚΡΙΣΗΣ ΤΟΥ READ ΚΑΙ WRITE ΓΙΑ ERASURE-CODE ΚΑΙ RAID-5 ΓΙΑ ΑΡΧΕΙΟ 100MB ΠΡΟΣ ΑΥΤΟΝ ΓΙΑ ΑΡΧΕΙΟ 50MB....136

ΣΧΗΜΑ 29: Ο ΧΡΟΝΟΣ ΑΠΟΚΡΙΣΗΣ (ΑΝΑΓΟΜΕΝΟΣ ΣΤΟ ΜΙΚΡΟΤΕΡΟ ΧΡΟΝΟ) ΤΟΥ RECOVER ΓΙΑ ΤΟ ERASURE-CODE ΚΑΙ ΔΙΣΚΟΥΣ ΜΕΓΕΘΟΥΣ 1000 MB 138
ΣΧΗΜΑ 30: Ο ΧΡΟΝΟΣ ΑΠΟΚΡΙΣΗΣ (ΑΝΑΓΟΜΕΝΟΣ ΣΤΟ ΜΙΚΡΟΤΕΡΟ ΧΡΟΝΟ) ΤΟΥ RECOVER ΓΙΑ ΤΟ ERASURE-CODE ΚΑΙ ΔΙΣΚΟΥΣ ΜΕΓΕΘΟΥΣ 100 MB 138
ΣΧΗΜΑ 31: Ο ΛΟΓΟΣ ΤΟΥ ΧΡΟΝΟΥ ΑΠΟΚΡΙΣΗΣ ΤΟΥ RECOVER ΓΙΑ ΤΟ ERASURE-CODE ΓΙΑ ΜΕΓΕΘΟΣ ΔΙΣΚΩΝ 1000MB ΠΡΟΣ ΑΥΤΟΝ ΓΙΑ ΔΙΣΚΟΥΣ ΜΕΓΕΘΟΥΣ 100MB 139

Κεφάλαιο 1

Εισαγωγή

Οι βασικές λειτουργίες ενός υπολογιστικού συστήματος είναι η επεξεργασία και η αποθήκευση πληροφορίας. Τα υποσυστήματα των υπολογιστικών συστημάτων τα οποία ασχολούνται με την αποθήκευση και ανάκληση των δεδομένων ονομάζονται συστήματα αποθήκευσης. Τα τελευταία ορίζονται ως ο συνδυασμός των συσκευών μνήμης και των αλγορίθμων διαχείρισης και ελέγχου των αποθηκευμένων πληροφοριών.

Λόγω αυτής της κυριότητάς τους στο χώρο των υπολογιστών, τα συστήματα αποθήκευσης τυγχάνουν, όλο και πιο συχνά, το ενδιαφέρον της ερευνητικής κοινότητας. Στόχος αυτής της ερευνητικής δραστηριότητας είναι η επίτευξη της μεγαλύτερης δυνατής ταχύτητας μεταφοράς δεδομένων, αλλά και η ανάπτυξη μηχανισμών για την προστασία και διατήρηση της σχετικής πληροφορίας από σφάλματα και αστοχίες υλικού. Με σκοπό τη βελτιστοποίηση και των 2 τομέων των συστημάτων αποθήκευσης (την ταχύτητα και την ακεραιότητα της πληροφορίας) και δεδομένου του χαμηλού κόστους των δίσκων, τα RAID συστήματα άρχισαν να γίνονται όλο και πιο δημοφιλή.

Οι πλεονάζουσες συστοιχίες ανεξάρτητων δίσκων (**Redundant Array of Independent Disks**) ή πλεονάζουσες συστοιχίες φθηνών δίσκων (**Redundant Array of Inexpensive Disks**) όπως ονομάστηκαν αποτελούν ομάδες δίσκων που χρησιμοποιούνται από κοινού, με σκοπό τη δημιουργία ενός αξιόπιστου αποθηκευτικού μέσου υψηλών επιδόσεων. Ο όρος RAID περιλαμβάνει όλα εκείνα τα αποθηκευτικά συστήματα που διαμοιράζουν τα δεδομένα και διατηρούν πολλαπλά αντίγραφά τους σε ένα σύνολο από δίσκους, τα οποία κάτω από ένα σύστημα RAID εμφανίζονται ως ένας εικονικός δίσκος για το εκάστοτε λειτουργικό σύστημα. Με τον τρόπο αυτό εξασφαλίζουν την ακεραιότητα των δεδομένων και βελτιώνουν την απόδοση των λειτουργιών ανάγνωσης και εγγραφής για μεγάλους αποθηκευτικούς χώρους.

1.1 Σκοπός

Σκοπός της παρούσας εργασίας αποτελεί η μελέτη και ο σχεδιασμός μίας αρχιτεκτονικής RAID, η οποία θα μπορούσε να αξιοποιήσει τη θεωρία των κωδίκων αποκατάστασης σφαλμάτων και πιο συγκεκριμένα του Erasure Code, καθώς και η ανάπτυξη μιας υλοποίησης της αρχιτεκτονικής αυτής σε επίπεδο λογισμικού.

Οι διάφορες υπάρχουσες αρχιτεκτονικές RAID είναι γνωστές ως RAID levels με τις 0, 1 και 5 να είναι οι πιο δημοφιλείς. Η καθεμία από αυτές έχει τα δικά της χαρακτηριστικά με τα δικά της πλεονεκτήματα και μειονεκτήματα στην ταχύτητα και τη διατήρηση της ακεραιότητας της πληροφορίας.

- Το RAID level 0 αποτελείται από συστοιχίες δίσκων στις οποίες αποθηκεύεται κατανεμημένα μόνο η ζητούμενη πληροφορία χωρίς πλεονάζουσα πληροφορία ισοτιμίας (parity), με αποτέλεσμα να επιτυγχάνεται μεγάλη ταχύτητα εγγραφής και ανάγνωσης, ενώ δεν προσφέρει καμία προστασία στην αποθηκευμένη πληροφορία από σφάλματα και αστοχίες των δίσκων.

- Το RAID 1, το οποίο είναι γνωστό και ως mirror, έχει ακριβώς το διπλάσιο αριθμό δίσκων από αυτόν που χρειάζεται για να αποθηκεύσει τη ζητούμενη πληροφορία, ώστε να διατηρεί ακριβής αντίγραφο της πληροφορίας. Η αρχιτεκτονική αυτή προφέρει πολύ μεγάλη προστασία της πληροφορίας που αποθηκεύει, χωρίς όμως να υπάρχει κάποιο ιδιαίτερο κέρδος σε ταχύτητα, ενώ παράλληλα κοστίζει πολύ σε αποθηκευτικό χώρο.

- Το RAID 5, το οποίο αποτελεί πλέον την πιο διαδεδομένη αρχιτεκτονική RAID, διαθέτει έναν πλεονάζοντα δίσκο, ώστε μαζί με την πληροφορία να αποθηκεύεται και πλεονάζουσα πληροφορία ισοτιμίας (parity). Με τον τρόπο αυτό η πληροφορία παραμένει ακέραιη σε περίπτωση που ένας οποιοσδήποτε δίσκος χαθεί από σφάλμα ή αστοχία υλικού. Κατά συνέπεια το RAID 5 επιτυγχάνει ταχύτητες εγγραφής και ανάγνωσης κοντά με αυτές του RAID 0 και επιπρόσθετα παρέχει προστασία στην αποθηκευμένη πληροφορία.

Το RAID 5, επομένως, αποτελεί μια ενδιάμεση λύση σε σχέση με το RAID 0 και RAID 1. Η αρχιτεκτονική RAID, όμως, που μελετάται στην παρούσα εργασία στοχεύει σε μία καλύτερη ισορροπία μεταξύ της ταχύτητας αποθήκευσης και ανάκτησης της πληροφορίας και της προστασία αυτής, ιδιαίτερα στις περιπτώσεις που ο αριθμός των δίσκων στη συστοιχία είναι αρκετά μεγάλος. Η αρχιτεκτονική του RAID Erasure Code (RAID EC) μπορεί να υποστηρίξει m πλεονάζοντες δίσκους, ώστε να αποθηκεύεται μαζί

με τη ζητούμενη πληροφορία και m πλεονάζουσα πληροφορία ισοτιμίας (parity). Ως αποτέλεσμα, θεωρώντας ότι n δίσκοι προορίζονται για την αποθήκευση της ζητούμενης πληροφορίας και m για την πλεονάζουσα (m<n), αν είναι διαθέσιμοι οποιοιδήποτε n δίσκοι από τους n+m, η αποθηκευμένη πληροφορία μπορεί να ανακτηθεί πλήρως. Η δυνατότητα υπολογισμού της m πλεονάζουσας πληροφορίας, όπως και η ανάκτηση της ζητούμενης πληροφορίας από την πρώτη, παρέχεται, χρησιμοποιώντας τον κώδικα αποκατάστασης σφαλμάτων Erasure Code.

1.2 Η δομή του κειμένου

Στο δεύτερο κεφάλαιο της εργασίας γίνεται αναφορά στο θεωρητικό υπόβαθρο που απαιτείται για τη μελέτη και το σχεδιασμό της αρχιτεκτονικής RAID EC, αλλά και της υλοποίησής της. Στο κεφάλαιο αυτό, γίνεται αναφορά στις υπάρχουσες αρχιτεκτονικές RAID και ιδιαίτερα στις RAID 5 και 6, όντας πιο κοντά στην αρχιτεκτονική του RAID EC. Επιπρόσθετα, περιέχει εκτενή αναφορά στους κώδικες αποκατάστασης σφαλμάτων Erasure Code και ειδικότερα στον Reed-Solomon code, ο οποίος αποτελεί υλοποίηση του βέλτιστου Erasure Code για m>1 πλεονάζουσα πληροφορία. Αναφορά, επίσης, γίνεται και στις μαθηματικές θεωρίες των σωμάτων Γκαλουά (Galois fields) και της μεθόδου απαλοιφής Gauss (Gauss Elimination), οι οποίες είναι απαραίτητες για τους υπολογισμούς που λαμβάνουν χώρα στην αρχιτεκτονική RAID EC.

Στο τρίτο κεφάλαιο περιγράφεται ο σχεδιασμός της αρχιτεκτονικής RAID EC. Σε κάθε περίπτωση περιγράφονται οι σχεδιαστικές επιλογές, καθώς και οι όποιες εναλλακτικές αυτών. Η κάθε μία αιτιολογείται και τίθενται οι σχετικοί στόχοι που θα πρέπει να ικανοποιούνται από τα αποτελέσματα.

Στο τέταρτο κεφάλαιο γίνεται περιγραφή της υλοποίησης και των επιμέρους επιλογών αυτής. Πιο συγκεκριμένα, γίνεται μια αναφορά στις βασικές συναρτήσεις του κώδικα, οι οποίες υλοποιούν τις βασικές λειτουργίες που λαμβάνουν μέρος στην αρχιτεκτονική RAID EC, και στις όποιες επιλογές υλοποίησης των αλγορίθμων αυτών.

Τέλος, στο πέμπτο κεφάλαιο περιγράφονται τα αποτελέσματα από την υλοποίηση της αρχιτεκτονικής RAID EC, όπως και πιθανές βελτιώσεις και επεκτάσεις αυτής.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

Το κεφάλαιο αυτό έχει ως σκοπό την εισαγωγή στο σχεδιασμό και την υλοποίηση της αρχιτεκτονικής RAID Erasure Code και χωρίζεται σε δύο ενότητες: η πρώτη αφορά την περιγραφή των αρχιτεκτονικών RAID, που ήδη χρησιμοποιούνται ευρέως, και κυρίως των RAID 5 και 6, και η δεύτερη στην περιγραφή του Erasure Code και του θεωρητικού υπόβαθρου που σχετίζεται με αυτό.

Αναλυτικότερα, στην πρώτη ενότητα αναφέρονται εκτενώς η αρχιτεκτονική, οι λειτουργίες, και οι πρακτικές των RAID 5 και 6, όπως και η αιτιολόγηση των σχετικών επιλογών. Η σχεδίαση της αρχιτεκτονικής RAID EC στηρίζεται στην παραπάνω πληροφορία, η οποία είναι απαραίτητη για την κατανόηση των όποιων σχεδιαστικών επιλογών.

Στη δεύτερη ενότητα περιγράφεται σε σημαντικό βαθμό ο Erasure Code και πιο συγκριμένα ο Reed-Solomon code, τον οποίο κώδικα χρησιμοποιεί το RAID EC που σχεδιάζεται και υλοποιείται στην παρούσα εργασία. Επιπρόσθετα, παρατίθεται το θεωρητικό υπόβαθρο αναφορικά με τα σώματα Galois και τη μεθοδολογία απαλοιφής Gauss, οι οποίες χρησιμοποιούνται επίσης στο RAID EC για την πραγματοποίηση των όποιων υπολογισμών.

2.1 Αρχιτεκτονικές RAID

Τα κύρια χαρακτηριστικά των RAID συστημάτων είναι ότι, αρχικά, αποτελούνται από μια ομάδα φυσικών οδηγών δίσκου, την οποία το εκάστοτε λειτουργικό σύστημα βλέπει ως ένα λογικό δίσκο. Επιπρόσθετα, τα δεδομένα κατανέμονται μεταξύ των φυσικών

δίσκων μιας συστοιχίας, ώστε στην κάθε εγγραφή ή ανάγνωση οι κλήσεις στους επιμέρους φυσικούς δίσκους να είναι όσο το δυνατόν πιο κατανεμημένες. Τέλος, ένα από κυριότερα χαρακτηριστικά των RAID συστημάτων είναι η χρήση των πλεοναζόντων δίσκων για να αποθηκευθεί η πληροφορία ισοτιμίας (parity), ώστε να είναι εφικτή η ανάκτηση των δεδομένων σε περίπτωση απώλειας κάποιου δίσκου. Το τελευταίο αυτό χαρακτηριστικό δεν αναφέρεται στο RAID 0 λόγω της έλλειψης πλεονάζουσας πληροφορίας στην αρχιτεκτονική αυτή.

Πλεονεκτήματα των συστημάτων RAID αποτελούν η βελτίωση της επίδοσης της εγγραφής και ανάγνωσης (I/O), επιτρέποντας την αύξηση της χωρητικότητας των δεδομένων, όπως και η αυξημένη αξιοπιστία του συστήματος απέναντι σε απώλειες δίσκων, επιτρέποντας την ανάκτηση δεδομένων μετά από απώλεια ενός ή περισσότερων δίσκων, ανάλογα με το επίπεδο RAID. Παραδείγματος χάριν το RAID 0 έχει τις καλύτερες επιδόσεις I/O, αλλά έχει και τη μικρότερη αξιοπιστία ως προς την ακεραιότητα των αποθηκευμένων δεδομένων. Σε αντίθεση, το RAID 1 διαθέτει τις χειρότερες επιδόσεις I/O, αλλά εξασφαλίζει τη μεγαλύτερη αξιοπιστία για τη διατήρηση των δεδομένων. Τα RAID 5 και 6 βρίσκονται ενδιάμεσα των RAID 0 και 1 και αποτελούν αρχιτεκτονικές με πολύ καλές I/O επιδόσεις και καλή αξιοπιστία της διατήρησης της ακεραιότητας των δεδομένων, όταν ο αριθμός των δίσκων που χρησιμοποιούνται δεν είναι πολύ μεγάλος.

2.1.1 RAID Level 5

Το RAID 5, όπως και το 6, χρησιμοποιούν τεχνική ανεξάρτητης προσπέλασης. Σε μία συστοιχία (stripe) ανεξάρτητης προσπέλασης, κάθε δίσκος – μέλος λειτουργεί ανεξάρτητα, ώστε οι ξεχωριστές αιτήσεις I/O να μπορούν να εξυπηρετούνται παράλληλα. Για το λόγο αυτό, οι συστοιχίες ανεξάρτητης πρόσβασης είναι καταλληλότερες για εφαρμογές που απαιτούν μεγάλες συχνότητες αιτήσεων I/O, από εφαρμογές όπου απαιτούνται μεγάλες ταχύτητες μεταφοράς δεδομένων.

Το RAID 5 (όπως και το 6) χρησιμοποιεί κατανομή των δεδομένων σε ακολουθίες της τάξης του block.

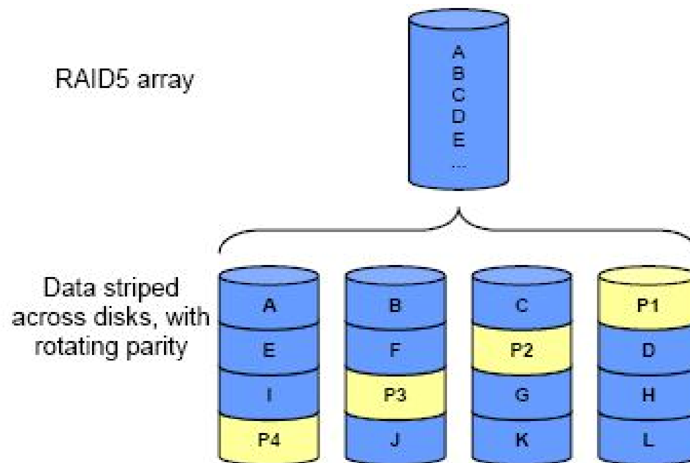
Το RAID 5 χρησιμοποιεί ένα parity και επομένως είναι σε θέση να ανακτήσει τα δεδομένα ύστερα από βλάβη σε ένα μόνο δίσκο. Έτσι μια συστοιχία του, της οποίας τα δεδομένα χρήστη απαιτούν N δίσκους, αποτελείται από N+1 δίσκους. Το parity κάθε συστοιχίας υπολογίζεται με τη βοήθεια της λογικής συνάρτησης XOR. Παραδείγματος χάριν για N=3, το parity P υπολογίζεται σε μια συστοιχία ως εξής: $P = X_0 \oplus X_1 \oplus X_2$, όπου X_0, X_1, X_2 τα δεδομένα (blocks) του χρήστη στους υπόλοιπους σκληρούς (για τη

συστοιχία αυτή). Έτσι στην περίπτωση που χάσουμε τον δίσκο του X_1, μπορούμε να ανακτήσουμε την πληροφορία του X_1: $X_1 = X_0 \oplus X_2 \oplus P$.

Σε κάθε αίτηση εγγραφής θα πρέπει να ενημερώνεται και το parity. Για το παράδειγμα που χρησιμοποιήσαμε και προηγουμένως, αν σε μία συστοιχία γίνει μία εγγραφή μόνο στο X_1 και η νέα τιμή του είναι X'_1, τότε το νέο parity P΄ της συστοιχία θα μπορούσε να υπολογιστεί ως εξής: $P' = P \oplus X_1 \oplus X_1'$, όπου P η προηγούμενη τιμή του parity. Επομένως, για τον υπολογισμό του νέου parity της συστοιχίας θα πρέπει να διαβαστεί η παλιά ακολουθία του χρήστη καθώς και το παλιό parity. Έτσι, κάθε εγγραφή ακολουθίας περιλαμβάνει δύο αναγνώσεις και δύο εγγραφές. Στην περίπτωση όμως μιας μεγαλύτερου μεγέθους εγγραφής I/O, που περιλαμβάνει αλλαγή όλων των δεδομένων χρήστη της συστοιχίας, το parity μπορεί να ενημερωθεί παράλληλα με τους δίσκους δεδομένων και δεν υπάρχουν επιπλέον αναγνώσεις. Επομένως σε κάθε περίπτωση, κάθε λειτουργία εγγραφής πρέπει απαραίτητα να περιλαμβάνει και το parity.

Το RAID 5, λόγω της συνεχούς και συχνής χρήσης του parity και για να αποφύγει την συμφόρηση ενός δίσκου (το RAID 4 χρησιμοποιεί ένα συγκεκριμένο δίσκο για την αποθήκευση του parity, με αποτέλεσμα να εμφανίζεται σ' αυτόν συμφόρηση αιτήσεων I/O και να καθυστερείται όλο το σύστημα), κατανέμει τα parities σε όλους τους δίσκους. Επιπρόσθετα, η κατανομή του parity που υιοθετεί το RAID 5 έχει επιπρόσθετα το πλεονέκτημα ότι μοιράζει τα δεδομένα σε όλους τους δίσκους (N+1) και όχι μόνο σε N. Το γεγονός αυτό επιτρέπει σε όλους του δίσκους να συμμετέχουν σε αιτήσεις ανάγνωσης και έτσι να μειώνουν τη συμφόρηση τους, σε αντίθεση με συστήματα που χρησιμοποιούν ένα συγκεκριμένο δίσκο για τα parities, ο οποίος δεν θα συμμετέχει σε αιτήσεις ανάγνωσης.

Σύμφωνα και με τα παραπάνω, η μέθοδος κατανομής του parity στους δίσκους επηρεάζει σε σημαντικό βαθμό την επίδοση του συστήματος. Στο Σχήμα 1 φαίνεται (σύμφωνα με τους Lee και Katz) η καλύτερη μέθοδος κατανομής του parity από αυτές που έχουν εξεταστεί και ονομάζεται αριστερά συμμετρική (left-symmetric) κατανομή του parity. Το πλεονέκτημα της μεθόδου αυτής είναι ότι πάντα, όταν διατρέχονται σειριακά οι ακολουθίες χρήστη των συστοιχιών, θα συναντάται πρώτα κάθε δίσκος μια φορά πριν συναντηθεί κάποιος για δεύτερη. Αυτό μεταφράζεται σε μείωση της συμφόρησης αιτήσεων I/O στους δίσκους και καλύτερη απόδοση.



Σχήμα 1: Κατανομή δεδομένων και parity για RAID 5

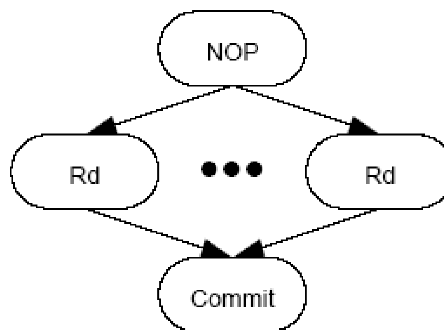
Πιο αναλυτικά οι λειτουργίες του RAID 5 παρουσιάζονται στη συνέχεια με τη βοήθεια κατευθυνόμενων ακυκλικών γράφων (DAGs = Directed Acyclic Graphs). Οι κόμβοι των γράφων αναπαριστούν απλές λειτουργίες, όπως ανάγνωση (R), εγγραφή (W) και τη λογική πράξη XOR (δηλαδή τον υπολογισμό του parity). Οι ακμές του αναπαριστούν τη διασύνδεση των διαφόρων λειτουργιών, δηλαδή τη μεταφορά δεδομένων και σημάτων ελέγχου. Το RAID 5, για να είναι σε θέση να εξασφαλίζει τη δημιουργία-απόκτηση της σωστής πληροφορίας (για την κάθε συστοιχία) και του αντίστοιχου parity από τυχόν σφάλματα κατά την εκτέλεση μία αίτησης I/O, ορίζει κάποια σημεία «ελέγχου» στη ροή των δεδομένων. Τα σημεία αυτά συμβολίζονται στους γράφους ως “C” ή “Commit”. Στα σημεία αυτά καταλήγουν τα σήματα ελέγχου των λειτουργιών που εκτελέστηκαν προηγουμένως και τα οποία είναι fail ή pass, για την περίπτωση που έχει εμφανιστεί σφάλμα κατά την εκτέλεση της ή όχι αντίστοιχα. Αν το σύνολο των σημάτων που φτάνει σε αυτά είναι pass, τότε συνεχίζεται κανονικά η εκτέλεση των λειτουργιών. Στην αντίθετη περίπτωση σταματάει την κανονική ροή εκτέλεση και ανάλογα με την πηγή του προβλήματος είτε αποτυγχάνει, είτε χρησιμοποιεί άλλη μέθοδο ικανοποίησης της αίτησης I/O (δηλαδή άλλο DAG). Τέλος, υπάρχει και ο κόμβος “NOP”, ο οποίος δεν αντιστοιχεί σε καμία λειτουργία και απλά εξασφαλίζει ότι κάθε γράφος έχει μοναδική αρχή και τέλος.

Το παρακάτω σχήμα (Σχήμα 2) παρουσιάζει ποιο DAG αντιστοιχεί σε κάθε αίτηση I/O ανάλογα με τη βλάβη δίσκου που υφίσταται (αν υφίσταται).

Request	Disk Faults	Graph
read	none	nonredundant read
read	data disk	degraded read
read	parity disk	nonredundant read
write < 50% of codeword	none	small write
write > 50% and < 100%	none	reconstruct write
write entire codeword	none	large write
write	data disk	reconstruct write
write	parity disk	nonredundant write

Σχήμα 2: Αντιστοιχία DAG με περιπτώσεις read και write για RAID 5

Στον DAG "nonredundant read", το οποίο φαίνεται στη συνέχεια, διαβάζονται, απλώς, τα δεδομένα χρήστη από τους δίσκους (αφού δεν έχει συμβεί βλάβη σε δίσκο όπου υπάρχει πληροφορία η οποία απαιτείται από την αίτηση ανάγνωσης και έτσι δεν συμβάλει και το parity).

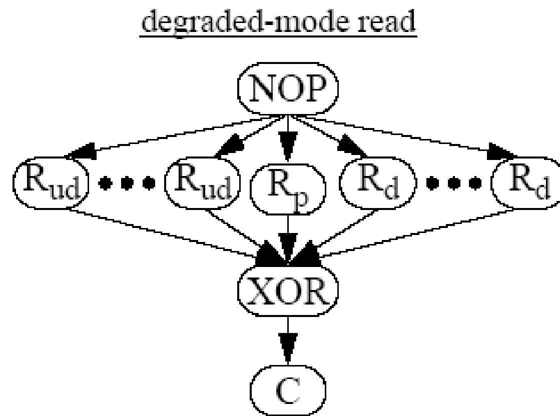


Nonredundant Read

Σχήμα 3: DAG: Nonredundant Read

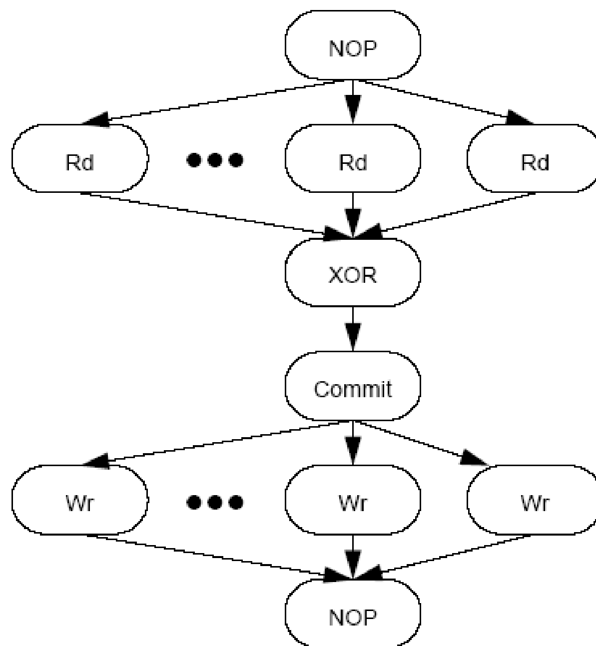
Στον DAG "degraded read", ο οποίος φαίνεται στη συνέχεια, εφόσον εντοπίζεται βλάβη σε δίσκο όπου υπάρχει επιθυμητή πληροφορία χρήστη, διαβάζονται μόνο όσες έχουν τα δεδομένα χρήστη (για τη/τις συστοιχία/ίες), αλλά και το parity ώστε να ανακτηθεί η

χαμένη πληροφορία. Στην περίπτωση που ζητούνται και κάποια δεδομένα χρήστη, τα οποία δεν βρίσκονται σε ίδια συστοιχία με δεδομένα χρήστη που έχουν ζητηθεί, αλλά βρίσκονται στον κατεστραμμένο δίσκο, και συνεπώς δεν έχουν ήδη διαβαστεί για την ανάκτηση των δεύτερων, διαβάζονται επιπρόσθετα (R_{ud}).



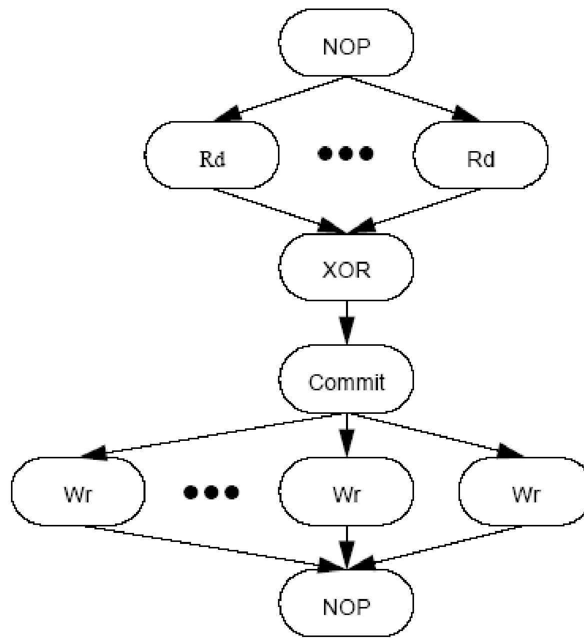
Σχήμα 4: DAG: degreaded-mode read

Στον DAG "small write", ο οποίος παρουσιάζεται στη συνέχεια, επειδή εφαρμόζεται στην περίπτωση όπου στη συστοιχία αλλάζουν λιγότερα από τα μισά δεδομένα χρήστη, με σκοπό να ενημερωθεί σωστά το νέο parity, διαβάζεται η παλιά τιμή του parity και των δεδομένων που θα αλλαχτούν. Αφού υπολογιστεί το νέο αυτό parity, οι νέες τιμές των δεδομένων χρήστη γράφονται στους αντίστοιχους δίσκους.



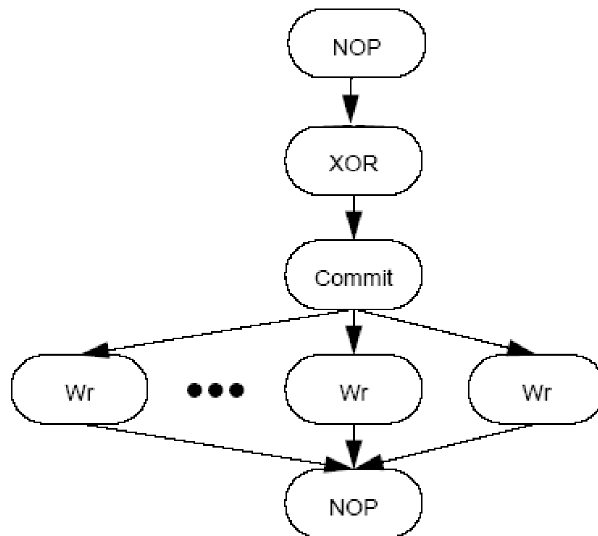
Σχήμα 5: DAG: Small-Write Graph

Στον DAG “reconstruct write”, ο οποίος φαίνεται στη συνέχεια, καθώς εφαρμόζεται στην περίπτωση όπου το μεγαλύτερο μέρος των δεδομένων χρήστη (αλλά όχι όλο) αλλάζει στη συστοιχία, για τον υπολογισμό του νέου parity διαβάζονται μόνο τα δεδομένα χρήστη που δεν αλλάζουν (αφού μαζί με τις νέες τιμές των υπόλοιπων δεδομένων υπολογίζεται το νέο parity), επιτυγχάνοντας λιγότερες αναγνώσεις για τον υπολογισμό του parity από ότι αν ακολουθούσαμε το προηγούμενο DAG (εφόσον τα περισσότερα δεδομένα χρήστη αλλάζουν). Στη συνέχεια, γράφονται στους αντίστοιχους δίσκους οι νέες τιμές του parity και των δεδομένων χρήστη.



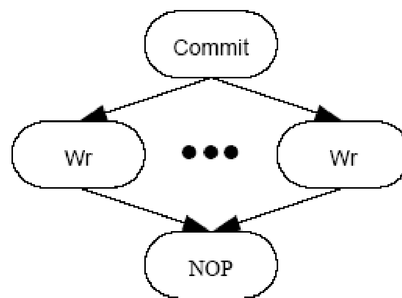
Σχήμα 6: DAG: Reconstruct-Write Graph

Στον DAG "large write", ο οποίος παρατίθεται στη συνέχεια, καθότι εφαρμόζεται στην περίπτωση που όλα τα δεδομένα χρήστη της συστοιχίας αλλάζουν, δεν χρειάζεται να γίνει καμία ανάγνωση, αφού η νέα τιμή του parity υπολογίζεται από της νέες τιμές των δεδομένων. Έπειτα, γράφονται στους αντίστοιχους δίσκους οι νέες τιμές του parity και των δεδομένων χρήστη.



Σχήμα 7: DAG: Large-Write Graph

Στον DAG "nonredundant write", ο οποίος φαίνεται στη συνέχεια, δεν χρειάζεται να γίνει υπολογισμός, καθώς θεωρείται ότι ο δίσκος όπου βρίσκεται το parity είναι κατεστραμμένος και επομένως απλά γράφονται οι νέες τιμές των δεδομένων.



Nonredundant Write

Σχήμα 8: DAG: Nonredundant Write

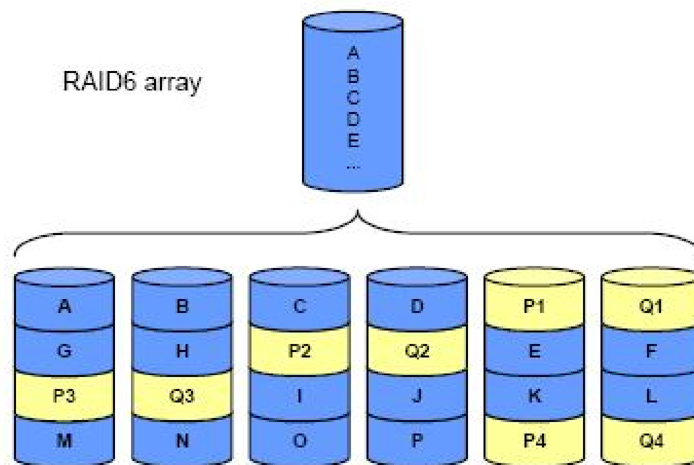
Τέλος, πρέπει να σημειωθεί ότι, στην περίπτωση που έχουμε αίτηση εγγραφής και υπάρχει βλάβη σε ένα δίσκο με δεδομένα χρήστη, δύναται να εφαρμοστεί είτε το DAG "small write" είτε το "reconstruct write", αναλόγως, βέβαια, με το κατά πόσο ο εσφαλμένος δίσκος συμπεριλαμβάνεται σε αυτούς προς εγγραφής (συνθήκη η οποία δεν παρουσιάζεται στον Πίνακα 1). Συνεπώς, στην περίπτωση την οποία πρόκειται να εγγραφούν δεδομένα στον εσφαλμένο δίσκο, χρησιμοποιείται το DAG "reconstruct write", διότι δεν υπάρχει παλιά τιμή για τα δεδομένα αυτά. Επιπρόσθετα, είναι επιθυμητό, στον υπολογισμό της νέας τιμής του P να λαμβάνει μέρος και η πληροφορία που αλλάζει, αλλά τελικά δεν αποθηκεύεται, ώστε να είναι δυνατή η ανάκτηση των δεδομένων αυτών. Στην αντίθετη περίπτωση δε, χρησιμοποιούμε το DAG "small write", διότι κάποια δεδομένα, τα οποία δεν αλλάζουν, έχουν πλέον χαθεί και στον υπολογισμό του νέου P επιθυμείται να λαμβάνει μέρος η παλιά τιμή του, η οποία εμπεριέχει την χαμένη πληροφορία, ώστε το νέο P να είναι σε θέση να την ανακτήσει.

2.1.2 RAID Level 6

Σε ένα σύστημα RAID με x δίσκους η πιθανότητα ένας από αυτούς να πάθει κάποια βλάβη είναι σημαντική. Πιο συγκεκριμένα, αν ο μέσος χρόνος για τη βλάβη ενός δίσκου είναι F, τότε ο μέσος χρόνος για τη βλάβη ενός δίσκου σε ένα σύστημα από x δίσκους είναι F/x. Επομένως, καθώς αυξάνενται ο αριθμός των δίσκων των συστημάτων RAID, αυξάνεται αρκετά η πιθανότητα εμφάνισης βλάβης σε δίσκο του συστήματος και μάλιστα σε περισσότερους του ενός ταυτόχρονα.

Όπως έχει ήδη αναφερθεί, το RAID 5 μπορεί να αντιμετωπίσει επιτυχώς μόνο μία βλάβη δίσκου και, σε περίπτωση που εμφανιστεί βλάβη και σε ένα δεύτερο δίσκο, η πληροφορία που βρίσκεται και στους δύο δίσκους χάνεται. Έτσι, για να αυξηθεί η αξιοπιστία του απέναντι στις βλάβες δίσκων, αναπτύχθηκε το RAID 6, το οποίο αποτελεί ουσιαστικά επέκταση του RAID 5 ώστε να μπορεί να ανακτήσει τα δεδομένα ύστερα από βλάβη σε δύο δίσκους ταυτόχρονα. Η διαφορά του RAID 6 από το 5 είναι ότι το πρώτο διαθέτει 2 parity P,Q, αντί ενός, και για το λόγο αυτό το RAID 6 καλείται και P+Q Redundancy. Τα P και Q υπολογίζονται με διαφορετικό αλγόριθμο, ώστε να μπορούν να ανακτήσουν τα δεδομένα ύστερα από ταυτόχρονη βλάβη σε δύο δίσκους που περιέχουν δεδομένα χρήστη (προφανώς αν εμφανιστεί ταυτόχρονα βλάβη και σε τρίτο δίσκο, τότε τα δεδομένα χάνονται). Το P υπολογίζεται, όπως και στο RAID 5, με τη λογική πράξη XOR από τα δεδομένα χρήστη της κάθε συστοιχίας και το Q με τον αλγόριθμο Reed-Solomon (τον οποίο θα εξετάσουμε στη συνέχεια). Επειδή τα P και Q αποθηκεύονται σε διαφορετικά τμήματα διαφορετικών δίσκων, μια συστοιχία RAID 6 της οποίας τα δεδομένα χρήστη απαιτούν N δίσκους αποτελείται από N+2 δίσκους.

Και στο RAID 6, όπως στο 5, τα parities κατανέμονται σε όλους τους δίσκους, όπως εξάλλου φαίνεται και στο Σχήμα 9, με σκοπό τη βελτίωση της απόδοσης του συστήματος (όπως εξηγήσαμε για το RAID 5). Χρησιμοποιείται και εδώ η ίδια μέθοδος κατανομής, η αριστερά συμμετρική (left-symmetric) κατανομή, ώστε να ισχύει και εδώ η πρόταση: πάντα, όταν διατρέχονται σειριακά οι ακολουθίες χρήστη των συστοιχιών, συναντάται κάθε δίσκος πρώτα μια φορά, πριν συναντηθεί κάποιος για δεύτερη φορά.



Σχήμα 9: Κατανομή δεδομένων και parity για RAID 5

Οι ακριβείς λειτουργίες του RAID 6 θα παρουσιαστούν και εδώ με τη βοήθεια των DAG. Για αυτά ισχύει ότι έχει ειπωθεί κατά την ανάλυση του RAID 5, μόνο που έχουν προστεθεί και οι κόμβοι Q και $\overline{Q}$ που συμβολίζουν τον υπολογισμό του Q και τους υπολογισμούς για την ανάκτηση πληροφοριών μέσω του Q αντίστοιχα.

Τα παρακάτω σχήματα (Σχήμα 10, 11) παρουσιάζουν ποιο DAG αντιστοιχεί σε κάθε αίτηση I/O, ανάλογα με τις βλάβες δίσκων που υφίστανται (στην περίπτωση που υφίστανται).

Request	Disk Faults	Graph
read	none	nonredundant read
read	single data disk	degraded read
read	parity disk	nonredundant read
read	Q disk	nonredundant read
read	two data disks	PQ double-degraded read
read	data + parity disks	PQ degraded-DP read
read	data + Q disks	degraded read

Σχήμα 10: Αντιστοιχία DAG με περιπτώσεις read και write για RAID 6 (μέρος α)

Request	Disk Faults	Graph
read	parity + Q disks	nonredundant read
write < 50% of codeword	none	PQ small write
write < 50% of codeword	parity	PQ small write, P omitted
write < 50% of codeword	Q	small write
write > 50% and < 100%	none	PQ reconstruct write
write > 50% and < 100%	parity	PQ reconstruct, P omitted
write > 50% and < 100%	Q	reconstruct write
write 100%	none	PQ large write
write 100%	parity	PQ large write, P omitted
write 100%	Q	large write
write	one data disk	PQ reconstruct write
write	two data disks	PQ double-degraded write
write	data + parity disks	PQ reconstruct, P omitted
write	data + Q disks	reconstruct write
write	parity + Q disks	nonredundant write

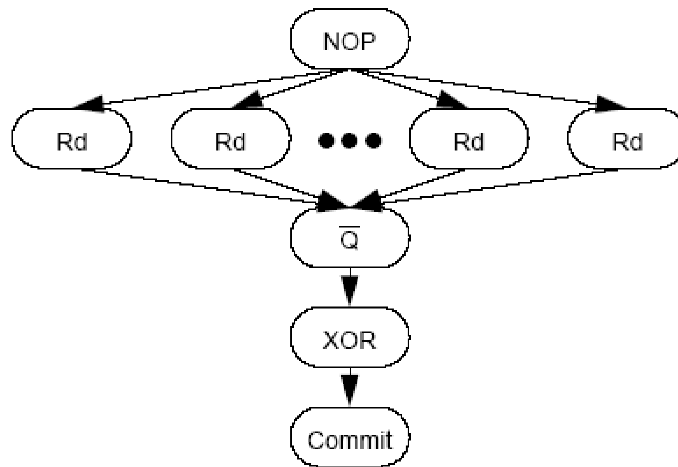
Σχήμα 11: Αντιστοιχία DAG με περιπτώσεις read και write για RAID 6 (μέρος β)

Πριν την παρουσίαση των DAG κρίνεται απαραίτητο να γίνει μια αναφορά για τον τρόπο υπολογισμού του Q. Για τον υπολογισμό του χρησιμοποιείται η πεδιακή άλγεβρα του Galois. Εάν τα δεδομένα χρήστη σε μία συστοιχία απαιτούν Ν δίσκους και αυτά είναι τα:

D_0, D1, D_2, … D_{n-1}, τότε το Q υπολογίζεται από την σχέση: $Q=2^0D_0+2^1D_1+2^2D_2+ \ldots +2^{n-1}D_{n-1}$. Στην περίπτωση που η τιμή μιας πληροφορίας χρήστη D_x αλλάξει στην D_x', τότε η νέα τιμή του Q, Q΄, υπολογίζεται ως εξής: $Q'=Q+2^xD_x+2^xD_x'$. Στην περίπτωση που υπάρχει βλάβη στο δίσκο του Dx, μπορούμε να ανακτήσουμε τα δεδομένα του με την βοήθεια του Q ως εξής: $Q_x+2^xD_x=Q$, όπου Q_x η τιμή του Q θεωρώντας το $D_x=0$. Στον προηγούμενο τύπο επειδή ο μόνος άγνωστος είναι το D_x, μπορούμε να ανακτήσουμε την τιμή του. Στην περίπτωση που πάθουν βλάβη δύο δίσκοι με δεδομένα χρήστη, δηλαδή χαθούν τα δεδομένα των Dx και Dy, τότε μπορούμε να τα ανακτήσουμε με τη βοήθεια των P και Q ως εξής: $P_{xy}+D_x+D_y=P$ $Q_{xy}+2^xD_x+2^yD_y=Q$, όπου P_{xy} και Q_{xy} είναι οι τιμές των P και Q θεωρώντας τα $D_x=D_y=0$.Επειδή στο παραπάνω σύστημα οι μόνοι άγνωστοι είναι τα D_x και D_y, οι τιμές μπορούν να υπολογιστούν. Υπενθυμίζεται ότι όλες οι πράξεις γίνονται χρησιμοποιώντας την πεδιακή άλγεβρα του Galois, και συγκεκριμένα σε πεδίο με 2^w στοιχεία (δηλαδή σε $GF(2^w)$), όπου w το μέγεθος (σε bits) των ακολουθιών του κάθε δίσκου που συμμετέχει στις συστοιχίες. Αυτή η υλοποίηση του Reed-Solomon κώδικα δεν είναι η μοναδική. Γενικά απαιτεί τον υπολογισμό του Q ως ένα γραμμικό συνδυασμό των D_0, D1, D_2, … D_{n-1}. Οι συντελεστές αυτών, δηλαδή, θα μπορούσαν να είναι αντίστοιχα οι 1,2,3,…,n, αλλά η επιλογή των 2^n ως συντελεστών φαίνεται να εξυπηρετεί τις πράξεις.

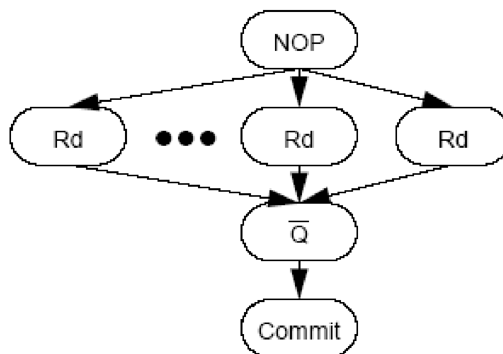
Τα DAG “nonredundant read”, “degraded read”, “small write”, “reconstruct write”, “large write” και “nonredundant write” είναι αυτά που παρουσιάστηκαν κατά την ανάλυση του RAID 5. Αυτό θεωρείται λογικό, καθώς τα συγκεκριμένα αναφέρονται στις περιπτώσεις όπου υπάρχει βλάβη στο δίσκο στον οποίο βρίσκεται το Q ή όταν υπάρχει αίτηση ανάγνωσης και προκύπτει βλάβη σε ένα μόνο δίσκο, ο οποίος περιέχει δεδομένα χρήστη ή το P. Σ’ αυτές τις περιπτώσεις το RAID 6 συμπεριφέρεται ως 5.

Στο DAG “PQ double-degraded read”, το οποίο φαίνεται στη συνέχεια και εφαρμόζεται στην περίπτωση που δύο δίσκοι με δεδομένα χρήστη έχουν πάθει βλάβη, διαβάζονται όλα τα υπόλοιπα δεδομένα χρήστη της συστοιχίας, μαζί με τα P και Q (τα δυο ακραία Rd στο γράφο), ώστε να ανακτηθούν τα δεδομένα που έχουν χαθεί, βάσει όσων αναφέρθηκαν προηγουμένως.



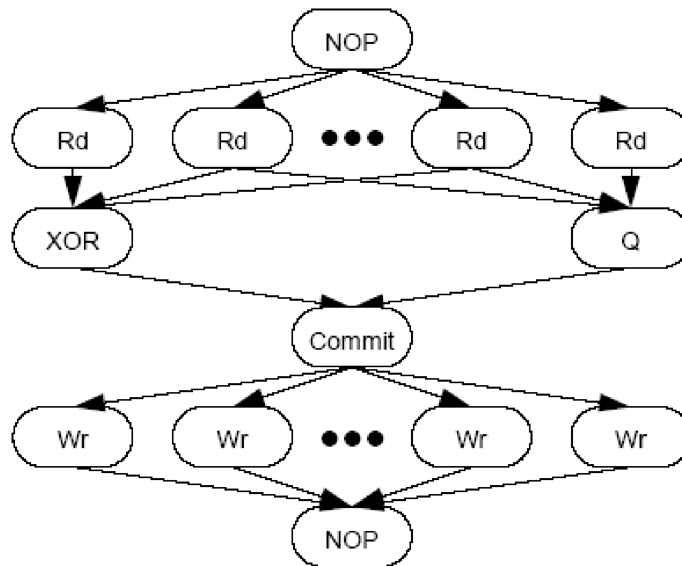
Σχήμα 12: DAG: PQ Double-Degraded-Read

Στο DAG "PQ degraded-DP read", το οποίο φαίνεται στη συνέχεια και εφαρμόζεται στην περίπτωση που ένας δίσκος με δεδομένα χρήστη και ο δίσκος που περιέχει το P έχουν βλάβη, διαβάζονται τα υπόλοιπα δεδομένα χρήστη της συστοιχίας, όπως και το Q, για την ανάκτηση της χαμένης πληροφορίας χρήστη με τον τρόπο που έχει ήδη αναφερθεί.



Σχήμα 13: DAG: PQ Degraded-DP-Read

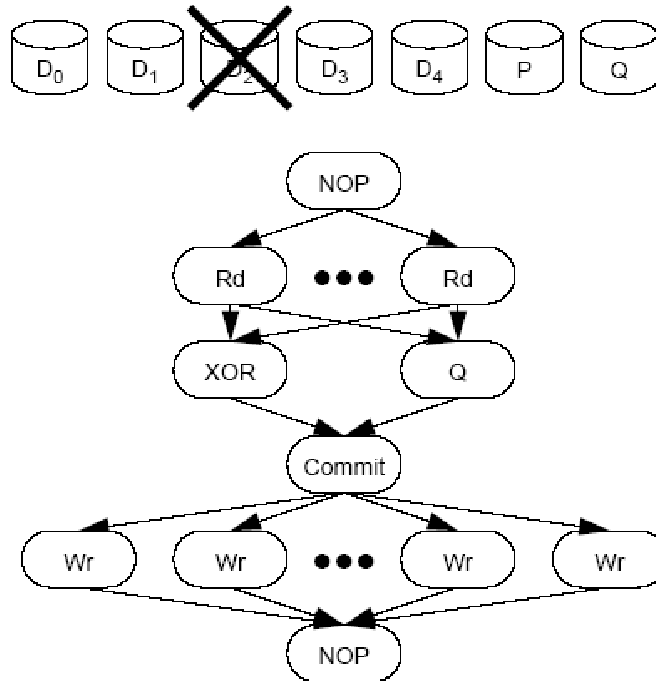
Στο DAG “PQ small write”, το οποίο παρουσιάζεται παρακάτω και χρησιμοποιείται στην περίπτωση που λιγότερα από τα μισά δεδομένα χρήστη της συστοιχίας αλλάζουν και δεν υπάρχει καμία βλάβη δίσκου, διαβάζονται οι παλιές τιμές των δεδομένων χρήστη, οι οποίες αλλάζουν μαζί με τις παλιές τιμές των P και Q, ώστε να υπολογιστεί η νέα τιμή των P (με βάση αυτά που ειπώθηκαν στην ανάλυση του RAID 5) και Q (με βάση αυτά που έχουν ειπωθεί). Ακολούθως, γράφονται στους αντίστοιχους δίσκους οι νέες τιμές των δεδομένων χρήστη και των P και Q. Επιπρόσθετα, το συγκεκριμένο DAG χρησιμοποιείται και στην περίπτωση την οποία υπάρχει βλάβη στο δίσκο όπου αποθηκεύεται το P, με τη διαφορά ότι δεν εφαρμόζονται οι διαδικασίες για τον υπολογισμό τις νέας τιμής του P.



Σχήμα 14: DAG: PQ Small-Write

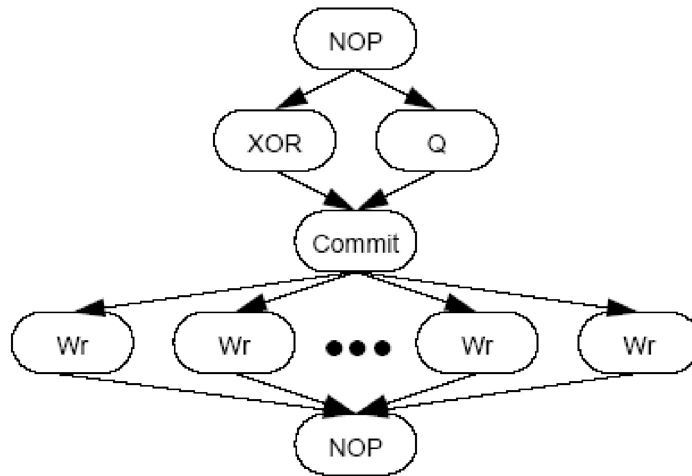
Στο DAG “PQ reconstruct write”, το οποίο παρατίθεται παρακάτω και εφαρμόζεται στην περίπτωση που περισσότερα από τα μισά αλλά όχι όλα τα δεδομένα χρήστη της συστοιχίας αλλάζουν και καμία βλάβη δίσκου δεν υφίσταται, διαβάζονται όλα τα υπόλοιπα δεδομένα χρήστη της συστοιχίας τα οποία δεν αλλάζουν, μαζί με τα P και Q, ώστε μαζί με της νέες τιμές των δεδομένων να μπορούν να υπολογιστούν και οι νέες τιμές των P και Q. Κατά αυτό τον τρόπο επιτυγχάνονται λιγότερες αναγνώσεις. Έπειτα,

οι νέες τιμές αναγράφονται στους αντίστοιχους δίσκους. Επειδή το DAG αυτό εφαρμόζεται και όταν προκύπτει βλάβη στο δίσκο όπου βρίσκεται το P, δεν διεξάγεται ο υπολογισμός του P στη συγκεκριμένη περίπτωση.



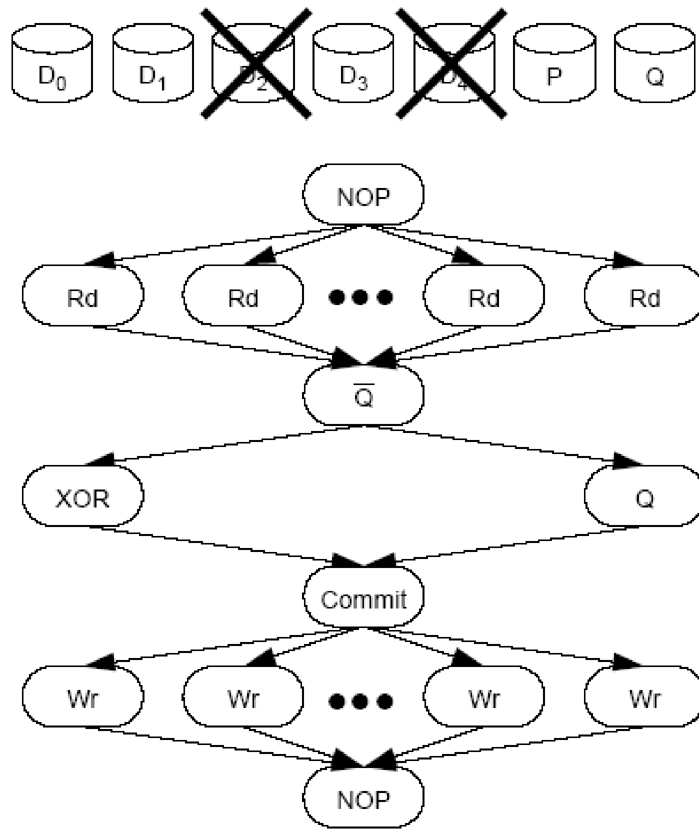
Σχήμα 15: DAG: PQ Reconstruct-Write

Στο DAG "PQ large write", το οποίο φαίνεται στη συνέχεια και εφαρμόζεται στην περίπτωση που όλα τα δεδομένα χρήστη της συστοιχίας αλλάζουν, και εφόσον δεν υφίσταται καμία βλάβη σε δίσκο, δεν χρειάζεται να γίνει καμία ανάγνωση και οι νέες τιμές των P και Q υπολογίζονται άμεσα από τις νέες τιμές των δεδομένων χρήστη. Στη συνέχεια, όλες οι νέες τιμές δεδομένων χρήστη, και των P και Q, γράφονται στους αντίστοιχους δίσκους. Επειδή το DAG αυτό χρησιμοποιείται και στην περίπτωση όπου υπάρχει βλάβη στο δίσκο που βρίσκεται το P, δεν διεξάγεται ο υπολογισμός του P σε αυτή την περίπτωση.



Σχήμα 16: DAG: PQ Large-Write

Στο DAG "PQ double-degraded write", το οποίο παρουσιάζεται στη συνέχεια και εφαρμόζεται στην περίπτωση που δύο δίσκοι με δεδομένα χρήστη έχουν βλάβη, ενώ ο ένας από αυτούς ανήκει στους δίσκους χρήστη προς εγγραφή, διαβάζονται όλα τα δεδομένα χρήστη της συστοιχίας που διασώζονται στους δίσκους (που είναι διαθέσιμοι), όπως και τα P και Q, ώστε να μπορούν να ανακτηθούν τα δεδομένα του δεύτερου δίσκου που έπαθε βλάβη και δεν περιλαμβάνεται στους προς εγγραφή δίσκους. Έπειτα, χρησιμοποιώντας την πληροφορία που ανακτήθηκε και τις τιμές των δεδομένων χρήστη της συστοιχία που είχαν διαβαστεί προηγουμένως, και οι οποίες δεν αλλάζουν, μαζί με τις νέες τιμές δεδομένων χρήστη, υπολογίζονται οι νέες τιμές των P και Q. Οι τιμές αυτές μπορούν να χρησιμοποιηθούν για την ανάκτηση των δεδομένων και των δύο εσφαλμένων δίσκων. Τέλος, οι νέες τιμές αναγράφονται στους αντίστοιχους δίσκους (αν αυτό είναι δυνατόν, καθώς ο ένας παρουσιάζει βλάβη όπως προαναφέρθηκε). Στην περίπτωση όπου δεν πρόκειται να εγγραφεί κανένα δεδομένο στους δύο εσφαλμένους δίσκους χρήστη, τότε χρησιμοποιείται το DAG "PQ small write", ώστε στον υπολογισμό των νέων P και Q να λαμβάνουν μέρος και οι παλιές τιμές τους, οι οποίες περιέχουν τα δεδομένα των εσφαλμένων δίσκων. Στην αντίθετη περίπτωση, όπου και στους δύο δίσκους με βλάβη πρόκειται να γραφούν δεδομένα, χρησιμοποιείται το DAG "PQ reconstruct write", εφόσον οι τιμές των δεδομένων χρήστη της συστοιχίας που δεν αλλάζουν είναι όλες διαθέσιμες, ενώ οι παλιές τιμές κάποιων από τα δεδομένα που αλλάζουν δεν είναι διαθέσιμες.



Σχήμα 17: DAG: PQ Double-Degraded-Write

Τέλος πρέπει να σημειωθεί ότι στην περίπτωση που έχουμε αίτηση εγγραφής και υπάρχει βλάβη σε ένα δίσκο με δεδομένα χρήστη (και ίσως και βλάβη στο δίσκο στον οποίο αποθηκεύεται το P), τότε εφαρμόζεται είτε το DAG "PQ small write" είτε το "PQ reconstruct write", ανάλογα με το κατά πόσο ο εσφαλμένος δίσκος συμπεριλαμβάνεται σε αυτούς προς εγγραφή (συνθήκη που δεν περιγράφεται στον Πίνακα 2). Πιο συγκεκριμένα, στην περίπτωση που πρόκειται να εγγραφούν δεδομένα στον εσφαλμένο δίσκο, χρησιμοποιείται το DAG "PQ reconstruct write", διότι δεν υπάρχει παλιά τιμή για τα δεδομένα αυτά και επιθυμούμε στον υπολογισμό των νέων τιμών των P και Q να περιέχεται και η πληροφορία που αλλάζει, αλλά τελικά δεν αποθηκεύεται, με σκοπό να μπορεί να ανακτηθεί στο μέλλον. Στην αντίθετη περίπτωση χρησιμοποιούμε το DAG "PQ small write", διότι κάποια δεδομένα που δεν αλλάζουν έχουν πλέον χαθεί και στον υπολογισμό των νέων P και Q επιθυμούμε να λαμβάνουν μέρος οι παλιές τιμές των P και

Q, οι οποίες εμπεριέχουν την χαμένη πληροφορία. Κατά αυτό τον τρόπο θα είναι δυνατόν να ανακτηθούν και από τα νέα P και Q.

2.2 Erasure Code

Στη Θεωρία της Πληροφορίας το Erasure Code αποτελεί κωδικοποίηση άμεσης διόρθωσης λαθών (Forward Error Correction - FEC), η οποία μετασχηματίζει ένα μήνυμα k συμβόλων σε ένα μεγαλύτερο, των n συμβόλων (n>k), ώστε το αρχικό μήνυμα να μπορεί να ανακτηθεί από ένα υποσύνολο των n συμβόλων.

Οι βέλτιστες εκδόσεις του Erasure Code έχουν την ιδιότητα ότι το αρχικό μήνυμα μπορεί να ανακτηθεί χρησιμοποιώντας οποιαδήποτε k από τα n σύμβολα του εκτεταμένου μηνύματος.

Ο έλεγχος ισοτιμίας αναφέρεται στην απλή περίπτωση που το n=k+1. Για ένα σύνολο k τιμών $\{\upsilon_i\}_{1\le i\le k}$, μια τιμή ισοτιμίας (check-sum) υπολογίζεται και προστίθεται στο αρχικό μήνυμα των k συμβόλων: $\upsilon_{k+1} = -\sum_{i=1}^{k}\upsilon_i$. Χρησιμοποιώντας το σύνολο των k+1 τιμών $\{\upsilon_i\}_{1\le i\le k+1}$, είναι δυνατή η ανάκτηση της τιμής υ_e, που τυχόν μπορεί να έχει χαθεί , βάση των υπολειπόμενων τιμών: $\upsilon_e = -\sum_{i=1, i\ne e}^{k+1}\upsilon_i$.

Στη γενικότερη περίπτωση, οι τιμές των συμβόλων των μηνυμάτων ανήκουν σε ένα περασμένο σύνολο τιμών F και κατασκευάζεται ένα πολυώνυμο Lagrance p(x) βαθμού k, ώστε η τιμή του p(i) να είναι ίση με την τιμή του συμβόλου i. Χρησιμοποιώντας αυτό το πολυώνυμο κατασκευάζονται και τα επιπρόσθετα σύμβολα, ώστε το μήνυμα να φτάσει στο μέγεθος των n συμβόλων. Στη συνέχεια, για την ανάκτηση τυχόν χαμένων συμβόλων του αρχικού μηνύματος, χρησιμοποιείται επίσης ένα πολυώνυμο παρεμβολής και οποιαδήποτε k σύμβολα του εκτεταμένου μηνύματος.

Υλοποίηση αυτής της γενικής διαδικασίας γίνεται από τους Reed-Solomon κώδικες, με τιμές να ανήκουν σε πεπερασμένο σώμα, και χρησιμοποιώντας πίνακες Vandermonde.

2.2.1 Reed-Solomon Κωδικοποίηση

Οι Reed-Solomon (RS) κώδικες ανήκουν στην κατηγορία των γραμμικών, μη δυαδικών κυκλικών κωδίκων ελέγχου και περιγράφουν ένα συστηματικό τρόπο δημιουργίας κώδικα, ο οποίος είναι σε θέση να εντοπίσει και να διορθώσει πολλαπλά και τυχαία σφάλματα στα σύμβολα μηνυμάτων. Προσθέτοντας στο μήνυμα t σύμβολα ισοτιμίας, ένας RS κώδικας μπορεί να εντοπίσει οποιοδήποτε συνδυασμό έως t εσφαλμένων συμβόλων και να διορθώσει μέχρι $\left\lfloor \frac{t}{2} \right\rfloor$ σύμβολα. Επιπρόσθετα ως Erasure Code, έχει τη δυνατότητα να ανακτήσει έως t γνωστά χαμένα σύμβολα. Η τιμή του t είναι σχεδιαστική επιλογή του κώδικα και μπορεί να κυμανθεί εντός μεγάλου εύρους τιμών.

Στη Reed-Solomon κωδικοποίηση, τα αρχικά σύμβολα του μηνύματος ορίζονται ως συντελεστές του χαρακτηριστικού πολυωνύμου p(x) και ανήκουν σε πεπερασμένα σώματα, τα οποία συνήθως είναι Galois fields GF(q), όπου q είναι δύναμη ενός αριθμού βάσης $q=p^m$ με m θετικό ακέραιο. Η αρχική διαδικασία ήταν η δημιουργία των n κωδικοποιημένων συμβόλων από τα αρχικά k σύμβολα, χρησιμοποιώντας το πολυώνυμο p(x) και τον υπολογισμό και των n συμβόλων με n>k, με τον αποδέκτη να χρησιμοποιεί τεχνικές παρεμβολής για την ανάκτηση του αρχικού μηνύματος. Στη συνέχεια, όμως, θεωρώντας τους RS κώδικες υποκατηγορία των κυκλικών BCH κωδίκων, τα κωδικοποιημένα σύμβολα παράγονται από τους συντελεστές του πολυωνύμου που δημιουργείται από το πολλαπλασιασμό του p(x) με πολυώνυμο γεννήτριας κυκλικού κώδικα.

Για τα συστήματα RAID χρησιμοποιείται μια εξειδικευμένη έκδοση των Reed-Solomon κωδίκων, η Vandermonde-RS, η οποία χρησιμοποιεί πίνακες Vandermonde για τον υπολογισμό της πλεονάζουσας πληροφορίας της ισοτιμίας. Vandermonde πίνακας θεωρείται ο πίνακας που τα στοιχεία της κάθε γραμμής του αποτελούν μία γεωμετρική πρόοδο. Ένα παράδειγμα ενός τέτοιου $m \times n$ πίνακα φαίνεται στο Σχήμα 18.

$$V = \begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \dots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \dots & \alpha_m^{n-1} \end{bmatrix}$$

Σχήμα 18: Vendermonde πίνακας

Η παρακάτω συνάρτηση (1) χρησιμοποιείται για τον υπολογισμό της πλεονάζουσας πληροφορίας ισοτιμίας.

$$FD = C$$

$$\begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & \cdots & f_{0,n-1} \\ f_{1,0} & f_{1,1} & f_{1,2} & \cdots & f_{1,n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ f_{m-1,0} & f_{m-1,1} & f_{m-1,2} & \cdots & f_{m-1,n-1} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix}$$

Στα πλαίσια της εφαρμογής της παραπάνω μεθοδολογίας στα RAID συστήματα, κατασκευάζεται ένας πίνακας Β, ο οποίος πρέπει να διαθέτει τις ακόλουθες ιδιότητες:

1. Να αποτελεί ένα πίνακα $(n+m)\times n$
2. Ο υποπίνακας $n \times n$ των n πρώτων γραμμών να είναι μοναδιαίος πίνακας
3. Κάθε υποπίνακας που δημιουργείται διαγράφοντας m γραμμές του πίνακα να είναι αντιστρέψιμος.

Ο πίνακας Β χρησιμοποιείται για την κατασκευή της ακόλουθης συνάρτησης (2):

$$BD = E$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & a_{m.3} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

Με τον τρόπο αυτό ο πίνακας στήλη Ε απεικονίζει τους n+m δίσκους, ο πίνακας στήλη D τους αρχικούς δίσκους δεδομένων και ο πίνακας Β τον τρόπο υπολογισμού των δεδομένων των n+m, όπου οι n από τους n+m δίσκους έχουν ακριβώς την αρχική πληροφορία και οι πλεονάζοντες m την πληροφορία ισοτιμία, η οποία υπολογίζεται συναρτήσει των n δίσκων.

Στην περίπτωση που χαθεί κάποιος από τους n+m δίσκους, τότε για την ανάκτηση της τυχόν χαμένης πληροφορίας μετασχηματίζουμε τη συνάρτηση (2), αφαιρώντας από τους πίνακες B και E τις αντίστοιχες γραμμές που αντιστοιχούν στους χαμένους δίσκους, δημιουργώντας τους πίνακες B΄ και E΄. Έτσι, χρησιμοποιώντας τη συνάρτηση: B΄D=E΄ και τη μέθοδο απαλοιφής Gauss, λόγω του ότι ο πίνακας είναι αντιστρέψιμος χάρις την ιδιότητα 3 του πίνακα B, μπορούμε να υπολογίσουμε τη χαμένη πληροφορία.

Ο πίνακας B δημιουργείται μετασχηματίζοντας ένα $(n+m)\times n$ πίνακα Vendermonde χρησιμοποιώντας μια σειρά από βασικούς μετασχηματισμούς πινάκων. Τα στοιχεία του πίνακα Vendermonde ορίζονται να είναι τα i^j και ο πίνακας να παίρνει την παρακάτω μορφή:

$$\begin{bmatrix} 0^0(=1) & 0^1(=0) & 0^2(=0) & \cdots & 0^{n-1}(=0) \\ 1^0 & 1^1 & 1^2 & \cdots & 1^{n-1} \\ 2^0 & 2^1 & 2^2 & \cdots & 2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ (n+m-1)^0 & (n+m-1)^1 & (n+m-1)^2 & \cdots & (n+m-1)^{n-1} \end{bmatrix}$$

Εξ'ορισμού, ο παραπάνω πίνακας έχει την ιδιότητα ότι κάθε υποπίνακας του, ο οποίος δημιουργείται διαγράφοντας m γραμμές του αρχικού, είναι αντιστρέψιμος. Επιπρόσθετα, κάθε πίνακας που δημιουργείται από τον παραπάνω πίνακα από μία σειρά βασικών μετασχηματισμών πινάκων, διατηρεί αυτή την ιδιότητα, αφού οι βασικοί μετασχηματισμοί πινάκων δεν αλλάζουν το βαθμό ενός πίνακα. Επομένως, η κατασκευή του πίνακα B είναι απλά θέμα εφαρμογής βασικών μετασχηματισμών στον προαναφερόμενο πίνακα Vendermonde, μέχρι οι πρώτες n γραμμές να σχηματίσουν ένα μοναδιαίο πίνακα.

2.2.2 Σώματα Galois

Οι πράξεις που αναφέρονται στον αλγόριθμο Reed-Solomon και χρησιμοποιούνται για τον υπολογισμών των «checksums» (πληροφορία ισοτιμίας) και την ανάκτηση των χαμένων πληροφοριών χρήστη μέσων των τελευταίων, δεν μπορούν να γίνουν με τις γνωστές πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμός και διαίρεση) που χρησιμοποιούμε στο σύνολο των πραγματικών αριθμών, αν και έχουμε να κάνουμε με αριθμούς (όλες οι πληροφορίες στην ουσία είναι δυαδικοί αριθμοί). Αυτό οφείλεται στο γεγονός ότι, δεδομένου του μεγέθους των blocks μεταξύ των οποίων θα γίνονται οι απαραίτητες πράξεις, έχουμε να κάνουμε με περασμένα σύνολα αριθμών. Συγκεκριμένα, αν το μέγεθος των blocks είναι w bit, τότε έχουμε ένα σύνολο με 2^w στοιχεία–αριθμούς

και οι πράξεις μας πρέπει να ορίζονται για όλα τα στοιχεία (εκτός από τη διαίρεση με το μηδέν) και κυρίως τα αποτελέσματα αυτών να είναι στοιχεία του συνόλου. Στην περίπτωση που δεν θα ισχύει το τελευταίο, δεν θα είναι δυνατός ο σωστός υπολογισμός των «checksums» και η ανάκτηση των ακριβών πληροφοριών που χάθηκαν λόγω σφάλματος σε δίσκο δεδομένων χρήστη. Αυτό, παραδείγματος χάριν, θα συμβεί αν χρησιμοποιηθεί η συνήθης άλγεβρα των πραγματικών αριθμών για τη διεξαγωγή των πράξεων αυτών. Σε αυτή την άλγεβρα θα είναι πολύ πιθανή η περίπτωση να προκύψει αποτέλεσμα που να μην μπορεί να παρασταθεί με w bit σε δυαδική μορφή, είτε προσθέτοντας ή πολλαπλασιάζοντας σχετικά μεγάλους αριθμούς, είτε παίρνοντας δεκαδικά πηλίκα. Παρόλα αυτά, όμως, το πρόγραμμα θα έδινε μία αλληλουχία bit στο block του αποτελέσματος της πράξης, η οποία δεν θα ήταν το πραγματικό αποτέλεσμα της πράξης, με άμεση συνέπεια να γίνονται λάθος υπολογισμοί και να αποτυγχάνει όλη η προσπάθεια αύξησης της ασφάλειας των δεδομένων. Επομένως, θα πρέπει να ορίσουμε δύο άλλες πράξεις στη θέση των συνηθισμένων πράξεων της πρόσθεσης και του πολλαπλασιασμού (ως γνωστό η αφαίρεση και η διαίρεση εξάγονται άμεσα από την πρόσθεση και τον πολλαπλασιασμό αντίστοιχα), ώστε με το δεδομένο σύνολο αριθμών να σχηματίζουν ένα σώμα.

Σώμα, γενικά, θεωρείται ένα αλγεβρικό σύστημα που αποτελείται από στοιχεία τα οποία ονομάζονται κοινώς αριθμοί και στο οποίο οι 4 συνηθισμένες πράξεις, πρόσθεση, αφαίρεση, πολλαπλασιασμός και διαίρεση, ορίζονται για όλα τα στοιχεία (εκτός από τη διαίρεση με το μηδέν) και έχουν όλες τις γνωστές ιδιότητες τους. Πιο αυστηρά, έστω Ε ένα μη κενό σύνολο στο οποίο έχουν οριστεί οι (εσωτερικές) πράξεις *, #. Θα λέμε ότι η (διατεταγμένη) τριάδα (Ε, *, #) είναι σώμα τότε, και μόνο τότε, αν για κάθε $\alpha \in E$, κάθε $\beta \in E$ και κάθε $\gamma \in E$, ισχύει:

1. (Ε, *) είναι αντιμεταθετική ομάδα με ουδέτερο στοιχείο της, έστω e. Δηλαδή:
   a. α*β=β*α [αντιμεταθετικότητα της *]
   b. (α*β)*γ=α*(β*γ) [προσεταιριστικότητα της *]
   c. α*e=e*α=α [ύπαρξη του ουδέτερου στοιχείου e, ως προς την *]
   d. α*α΄=α΄*α=e [ύπαρξη του συμμετρικού στοιχείου α΄, ως προς]
2. (E^*,#), όπου E^*=E-{e}, είναι αντιμεταθετική ομάδα με ουδέτερο στοιχείο της, έστω ε. Δηλαδή:
   a. α#β=β#α [αντιμεταθετικότητα της #]
   b. (α#β)#γ=α#(β#γ) [προσεταιριστικότητα της #]
   c. α#ε=ε#α=α [ύπαρξη του ουδέτερου στοιχείου ε, ως προς την #]
   d. α#α¨=α¨#α=ε [ύπαρξη του συμμετρικού στοιχείου α¨, ως προς την #]
3. Η πράξη # είναι επιμεριστική ως προς την πράξη #. Δηλαδή:
   a. α#(β*γ)=(α#β)*(α#γ)

Προφανώς τα e, ε, α΄ και α¨ ανήκουν στο Ε. Ένα παράδειγμα είναι το γνωστό σώμα των πραγματικών αριθμών, όπου το σύνολο Ε είναι το σύνολο των πραγματικών αριθμών, η πράξη * είναι η γνωστή πρόσθεση, η πράξη # ο γνωστός πολλαπλασιασμός, e=0 και ε=1.

Στην περίπτωση μας όμως έχουμε να κάνουμε με πεπερασμένο σώμα (αφού το E είναι πεπερασμένο) με 2^w στοιχεία.

Το πιο απλό πεπερασμένο σώμα είναι η μετρική (modular) κλάση Z_n με n πρώτο αριθμό. Τα μέλη του Z_n είναι οι ακέραιοι 0,1,…,n-1, που πρέπει να θεωρούνται ως υπόλοιπα διαιρέσεων δια του n. Για να προσθέσουμε δύο στοιχεία α, β του Z_n, διαιρούμε το συνηθισμένο άθροισμα α + β των στοιχείων αυτών δια του n. Τότε το υπόλοιπο υ της διαιρέσεως είναι το ζητούμενο άθροισμα, δηλαδή α + β = υ (mod n). Ομοίως, για να πολλαπλασιάσουμε δύο στοιχεία α, β του Z_n, διαιρούμε το συνηθισμένο γινόμενο α*β των στοιχείων αυτών δια του n. Τότε το υπόλοιπο υ της διαιρέσεως είναι το ζητούμενο γινόμενο, δηλ. α * β = υ (mod n). Στην περίπτωση που το n είναι ένας πρώτος αριθμός p, ο πολλαπλασιασμός στο Z_p είναι (εκτός επί 0) αντιστρέψιμος, οπότε το Z_p είναι σώμα ως προς την πρόσθεση και τον πολλαπλασιασμό modulo p (τα υπόλοιπα αξιώματα για να είναι το Z_n σώμα ως προς τις προαναφερόμενες πράξεις ισχύουν για κάθε n ακέραιο). Όταν n=rs, όπου r, s φυσικοί αριθμοί με $1<r,s<n$ (πράγμα που συμβαίνει μόνο όταν ο n δεν είναι πρώτος), τότε είναι δυνατή η ισότητα rs=0 στο Z_n, μολονότι είναι $r \neq 0$ και $s \neq 0$. Επομένως, όταν ο n δεν είναι πρώτος, το Z_n δεν είναι ούτε καν ακέραια περιοχή. Έτσι, συνάγεται ότι το Z_n είναι σώμα, όταν και μόνο όταν ο n είναι πρώτος. Άρα και το ζητούμενο σώμα δεν μπορεί να είναι το Z_n, αφού το n θα ήταν 2^w, ο οποίος είναι ένας άρτιος αριθμός.

Κάθε πεπερασμένο σώμα όμως, με χαρακτηριστική p (χαρακτηριστική ενός σώματος είναι ο ελάχιστος θετικός ακέραιος n, για τον οποίο ισχύει n1=0), είναι μια απλή αλγεβρική επέκταση του πρώτου υποστρώματος του Z_p. Λεπτομερέστερα, κάθε πεπερασμένο σώμα με χαρακτηριστική p περιέχει, για κάποιο θετικό ακέραιο n, p^n στοιχεία και είναι ισόμορφο προς το σώμα $Z_p[x]/(f(x))$ (δηλαδή είναι μία αμφιμονοσήμαντη απεικόνιση επί αυτού), για κάποιο μονικό ανάγωγο πολυώνυμο f(x) στο $Z_p[x]$ βαθμού n (μονικό πολυώνυμο στο $Z_p[x]$ λέγεται όταν όλοι οι συντελεστές του είναι στοιχεία του Z_p και ειδικά συντελεστής 1 για το x^n και ανάγωγο όταν δεν μπορεί να αναλυθεί σε γινόμενο πολυωνύμων μικρότερου βαθμού με συντελεστές στο Z_p). Με $Z_p[x]$ συμβολίζεται η περιοχή όλων των πολυωνύμων επί του Z_p (δηλαδή με συντελεστές στο Z_p) ως προς την άγνωστη x. Με (f(x)) συμβολίζεται τα πολλαπλάσια του f(x), που αποτελούν ένα ιδεώδες της περιοχής $Z_p[x]$, αφού f(x) μονικό ανάγωγο πολυώνυμο στο $Z_p[x]$. Τα στοιχεία του σώματος-πηλίκου $Z_p[x]/(f(x))$ είναι οι κλάσεις υπολοίπων ως προς το ιδεώδες (f(x)). Κάθε κλάση υπολοίπων του ιδεώδους (f(x)) περιέχει ένα ακριβώς πολυώνυμο βαθμού μικρότερου από τον βαθμό n του f(x). Το πολυώνυμο r(x) του $Z_p[x]$, που ανήκει στην κλάση (f(x))+p(x)=p(x)+(f(x)), όπου $p(x) \in Z_p[x]$, μπορεί να προσδιοριστεί από οποιοδήποτε μέλος της κλάσης –π.χ. από το p(x)- ως υπόλοιπο της διαίρεσης του μέλους-πολυωνύμου δια του f(x). Κατά συνέπεια, τα πολυώνυμα του $Z_p[x]$ με βαθμό μικρότερο από τον βαθμό n του f(x), όπως είναι το r(x) και το s(x), μπορούν να χρησιμοποιηθούν για να παραστήσουν συμβολικά τα στοιχεία του $Z_p[x]/(f(x))$. Τα

πολυώνυμα αυτά μπορούν να προστεθούν, όπως τα συνηθισμένα πολυώνυμα, μόνο που πρέπει να έχουμε υπόψη ότι οι συντελεστές τους είναι στοιχεία του σώματος Z_p και η πράξη της πρόσθεσης επί αυτών πρέπει να είναι η αντίστοιχη ως προς την οποία έχει οριστεί στο σώμα, οπότε έτσι ορίζεται η πρόσθεση στο $Z_p[x]/(f(x))$. Για να ορίσουμε τον πολλαπλασιασμό στο $Z_p[x]/(f(x))$ εργαζόμαστε ως εξής: ως γινόμενο των r(x) και s(x) στο $Z_p[x]/(f(x))$ ορίζουμε το υπόλοιπο της διαίρεσης του συνηθισμένου γινομένου τους (το γινόμενο των συντελεστών γίνεται βάσει της αντίστοιχης πράξης ως προς την οποία έχει οριστεί το σώμα Z_p, αφού αυτοί αποτελούν στοιχεία τους) δια του f(x).

Τα πεπερασμένα σώματα κατασκευάστηκαν για πρώτη φορά από τον Γκαλουά (Galois), γι' αυτό και λέγονται επίσης σώματα Γκαλουά (Galois Fields). Αν ο p είναι πρώτος και ο n θετικός ακέραιος, τότε το σώμα Γκαλουά τάξεως p^n συμβολίζεται με $GF(p^n)$. Επομένως, το ζητούμενο σώμα, βάσει του οποίου θα εκτελεστούν οι πράξεις του αλγορίθμου του Reed-Solomon, είναι το σώμα $GF(2^w)$. Για τα σώματα Γκαλουά ισχύει το εξής θεώρημα: Η πολλαπλασιαστική ομάδα του $GF(p^n)$ είναι κυκλική τάξεως p^n-1, δηλαδή τα μη-μηδενικά στοιχεία του $GF(p^n)$ είναι όλα δυνάμεις ενός στοιχείου (γεννήτορα) x τάξεως p^n-1, δηλαδή το x είναι αρχική ρίζα της μονάδας 1 του $GF(p^n)$ τάξεως p^n-1.

Για παράδειγμα, ας δούμε το πεπερασμένο σώμα $GF(2^4)= GF(16)\cong Z_2[x]/\left(x^4+x+1\right)$. Αν x είναι γεννήτορας της (κυκλικής) πολλαπλασιαστικής ομάδας του GF(16), τότε το GF(16) αποτελείται από τα στοιχεία 0, x, x^2, x^3, x^4,…,x^{15}=1. Κατά συνέπεια, τα μη-μηδενικά στοιχεία του GF(16) εκφράζονται από τις παρακάτω σχέσεις:

$$
\begin{aligned}
x &= x \\
x^2 &= x^2 \\
x^3 &= x^3 \\
x^4 &= x+1 \\
x^5 &= x^2+x \\
x^6 &= x^3+x^2 \\
x^7 &= x^3+x+1 \\
x^8 &= x^2+1 \\
x^9 &= x^3+x \\
x^{10} &= x^2+x+1 \\
x^{11} &= x^3+x^2+x \\
x^{12} &= x^3+x^2+x+1 \\
x^{13} &= x^3+x^2+1 \\
x^{14} &= x^3+1 \\
x^{15} &= x^0 = 1
\end{aligned}
$$

Σχήμα 19: Μη-μηδενικά στοιχεία του GF(16)

Αν το x είναι γεννήτορας της πολλαπλασιαστικής ομάδας του GF(16), τότε τα στοιχεία x, x^2, x^4, x^7, x^8, x^{11}, x^{13}, x^{14} των οποίων οι εκθέτες είναι πρώτοι προς τον 15, αποτελούν όλους τους γεννήτορες της πολλαπλασιαστικής ομάδας του GF(16).

2.2.3 Μέθοδος απαλοιφής Gauss

Η μέθοδος απαλοιφής Gauss αποτελεί την πιο ενδεδειγμένη μέθοδο επίλυσης γραμμικών συστημάτων γενικής μορφής με Υπολογιστή. Η βασική ιδέα της μεθόδου αυτής είναι η μετατροπή του συστήματος Ax=b σε ένα ισοδύναμο σύστημα A΄x=b΄, όπου ο πίνακας A΄ είναι άνω-τριγωνικός, δηλαδή τέτοιος ώστε $a_{ij} = 0$ για i>j, οπότε το νέο σύστημα A΄x=b΄ λύνεται εύκολα με διαδοχικές αντικαταστάσεις. Στην περίπτωση όπου ο πίνακας A είναι τετραγωνικός $n \times n$ και ομαλός, δηλαδή με $|A| \neq 0$ (όπως σημαίνει και στον πίνακα B΄ της ενότητας 2.2.1), η μέθοδος Gauss εφαρμόζεται σε 2 φάσεις: την τριγωνοποίηση και την πίσω-αντικατάσταση.

I. Τριγωνοποίηση

Απαλείφουμε διαδοχικά, για k=1, …, n-1 την άγνωστο x_k από τις εξισώσεις k+1 μέχρι n. Στο βήμα k, παίρνουμε το ακόλουθο σύστημα, με νέους συντελεστές a_{ij} και δεύτερα μέλη b_i

$$
(1) \qquad
\begin{array}{c|c}
 & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1k}x_k + \ldots + a_{1n}x_n = b_1 \\
 & \vdots \qquad\qquad \vdots \\
 & a_{kk}x_{kk} + \ldots + a_{kn}x_n = b_k \\
p_{k+1} & a_{k+1,k}x_k + \ldots + a_{k+1,n}x_n = b_{k+1} \\
 & \vdots \qquad\qquad \vdots \\
p_i = \dfrac{a_{ik}}{a_{kk}} & 0 \qquad a_{ik}x_k + \ldots + a_{in}x_n = b_i \\
 & \vdots \qquad\qquad \vdots \\
p_n & a_{nk}x_k + \ldots + a_{nn}x_n = b_n
\end{array}
$$

Ας εξετάσουμε τώρα την απαλοιφή του x_k στο βήμα k.

Θεωρούμε το υποσύστημα των εξισώσεων k μέχρι n. Αν $a_{kk} = 0$, ανταλλάσουμε πρώτα την εξίσωση k με μία εξίσωση i, i>k, με $a_{ik} \neq 0$ (ή την άγνωστο x_k με μία άγνωστο x_j, j>k, με $a_{kj} \neq 0$). Υπάρχει τέτοιος συντελεστής, αλλιώς είναι φανερό ότι $|A| = 0$.

Μετά, για i=k+1, …, n, πολλαπλασιάζουμε την εξίσωση k με $p_i = a_{ik}/a_{kk}$ και την αφαιρούμε από την εξίσωση i, απαλείφοντας έτσι την άγνωστο x_k από τις εξισώσεις k+1 μέχρι n. Παίρνουμε έτσι

$$
(2) \qquad
\left|
\begin{array}{lll}
p_i = a_{ik}/a_{kk} & i = k+1,\ldots,n & \\
a_{ij} = a_{ij} - p_i a_{kj} & i = k+1,\ldots,n & j = k,\ldots,n \\
b_1 = b_i - p_i b_k & i = k+1,\ldots,n &
\end{array}
\right.
$$

Σημειωτέον ότι το σύστημα που προκύπτει είναι ισοδύναμο με το αρχικό σύστημα (1). Αν εφαρμόσουμε την απαλοιφή για k = 1, …, n-1, καταλήγουμε έτσι σε ένα τριγωνικό σύστημα.

(3)

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{22}x_{22} + \ldots + a_{2n}x_n &= b_2 \\
\vdots \qquad\qquad & \quad \vdots \\
a_{kk}x_k + \ldots + a_{kn}x_n &= b_k \\
\vdots \qquad & \quad \vdots \\
a_{nn}x_n &= b_n
\end{aligned}
$$

II. Πίσω-αντικατάσταση

Ξεκινώντας τότε από $x_n = b_n / a_{nn}$ και έχοντας υπολογίσει διαδοχικά τις αγνώστους $x_{n-1}, x_{n-2}, \ldots, x_{k-1}$, από το σύστημα (4), υπολογίζουμε το x_k από την εξίσωση k

(4)

$$x_k = \frac{1}{a_{kk}}\left(b_k - \sum_{j=k+1}^{n} a_{kj}x_j\right), k = n-1, \ldots, k-1$$

Στη συνέχεια, παρατηρούμε ότι για λόγους αριθμητικής ευστάθειας, σε κάθε βήμα της τριγωνοποίησης, και πριν απαλείψουμε το x_k, ανταλλάσουμε συνήθως την εξίσωση k με την εξίσωση $i \geq k$ που έχει το μεγαλύτερο συντελεστή a_{ik} σε απόλυτη τιμή. Η πράξη αυτή καλείται μερική οδήγηση κατά στήλη. Μπορούμε επίσης να ανταλλάξουμε την άγνωστο x_k με την άγνωστο x_j, $j \geq k$, με το μεγαλύτερο συντελεστή a_{kj} σε απόλυτη τιμή. Αυτό λέγεται τότε μερική οδήγηση κατά γραμμή. Στην περίπτωση αυτή, όμως, οι άγνωστοι πρέπει στο τέλος να αναδιαταχθούν. Τέλος, στην πλήρη οδήγηση διαλέγουμε, σε κάθε βήμα της τριγωνοποίησης, το μεγαλύτερο συντελεστή a_{ij}, $i \geq k$, $j \geq k$ σε απόλυτη τιμή, και ανταλλάσουμε τις εξισώσεις k και i και τις αγνώστους k και j. Η διαδικασία, όμως, αυτή είναι περισσότερο χρονοβόρα.

Επιπρόσθετα, για την εφαρμογή της μεθόδου σε Υπολογιστή, χρησιμοποιούμε τον λεγόμενο επαυξημένο πίνακα του συστήματος

$$[A|b] = \left[\begin{array}{ccc|c} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & bn \end{array}\right]$$

Κεφάλαιο 3

Σχεδιασμός

Στο κεφάλαιο αυτό περιγράφεται ο σχεδιασμός και η αρχιτεκτονική του RAID Erasure Code συστήματος, όπως αυτό αναπτύχθηκε στα πλαίσια της παρούσας εργασίας, και χωρίζεται σε δύο ενότητες: η πρώτη αφορά την περιγραφή του τμήματος του σχεδιασμού του RAID EC που αναφέρεται στην εφαρμογή του Erasure Code σε ένα RAID σύστημα, ενώ στη δεύτερη ενότητα περιγράφεται η συνολική αρχιτεκτονική του RAID EC και οι επιμέρους βασικές διεργασίες που εκτελεί.

Αναλυτικότερα, στην πρώτη ενότητα αναπτύσσεται ο επακριβής τρόπος με τον οποίο η κωδικοποίηση Reed-Solomon χρησιμοποιείται σε ένα RAID σύστημα για την επίτευξη του RAID EC και της δυνατότητας να χρησιμοποιούνται m πλεονάζοντες δίσκοι, ώστε να επιτυγχάνουν την ανάκτηση ολόκληρης της αποθηκευμένης πληροφορίας ακόμη και όταν σε ένα σύστημα n+m δίσκων έχουν χαθεί οι m. Περιγράφεται η μεθοδολογία με την οποία επιτυγχάνονται για το RAID EC οι απαιτούμενες πράξεις στο σώμα Galois, ο υπολογισμός του πίνακα B της Reed-Solomon κωδικοποίησης και η εφαρμογή της μεθόδου απαλοιφής Gauss.

Στη δεύτερη ενότητα λαμβάνει χώρα η περιγραφή της συνολικής αρχιτεκτονικής του RAID EC. Γίνεται αναφορά στη δομή που έχει και στον τρόπο κατανομής των δεδομένων αποθήκευσης και των parities στη συστοιχία των n+m δίσκων. Περιγράφεται ο τρόπος λειτουργίας του συστήματος και η ροή των βασικών διεργασιών του, όπως και ο τρόπος με τον οποίο αλληλεπιδρούν οι τελευταίες. Τέλος, προδιαγράφεται σε μεγάλο βαθμό η λογική και μεθοδολογία που ακολουθείται από τις βασικές διεργασίες του RAID EC.

Η αρχιτεκτονική, η οποία έχει αναπτυχθεί στην παρούσα εργασία, έχει σχεδιαστεί ώστε να έχει μια αρκετά ευέλικτη υλοποίηση, τόσο σε user level όσο και σε kernel level και

embedded συστήματα. Επιπρόσθετα, ως δίσκος δεδομένων μπορεί να θεωρηθεί οποιοδήποτε αποθηκευτικό μέσο, όπως block device ή file.

3.1 Εφαρμογή Erasure Code σε RAID συστήματα

Βάσει των όσων προαναφέρθηκαν στην παράγραφο 2.2.1, χρησιμοποιώντας την συνάρτηση BD=R της κωδικοποίησης Reed-Solomon, η οποία παρουσιάζεται και στη συνέχεια, είναι δυνατόν να αποθηκεύεται πληροφορία n λέξεων σε n+m λέξεις και στη συνέχεια να μπορεί να ανακτηθεί η αρχική πληροφορία από οποιεσδήποτε n λέξεις των n+m λέξεων, χρησιμοποιώντας τη μέθοδο απαλοιφής Gauss.

$$BD = E$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & a_{m.3} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

Αντικαθιστώντας τις λέξεις δεδομένων με τα κομμάτια δεδομένων (chunk) που ανήκουν στον κάθε δίσκο του συστήματος RAID στο εκάστοτε stripe του τελευταίου, και χρησιμοποιώντας την παραπάνω συνάρτηση, είναι δυνατόν να υπολογιστεί η αντίστοιχη πληροφορία για τους πλεονάζοντες δίσκους (parity). Ο τρόπος, με τον οποίο κατανέμονται τα δεδομένα προς αποθήκευση και τα αντίστοιχα parities στους δίσκους για το εκάστοτε stripe, θα αναφερθεί στην επόμενη ενότητα. Στη συνέχεια, στην περίπτωση που χαθούν έως m δίσκος από τους n+m, είναι δυνατή η ανάκτηση τους χρησιμοποιώντας τη συνάρτηση BD=R και τη μέθοδο απαλοιφής του Gauss,.

3.1.1 Εφαρμογή των σωμάτων Galois στα RAID EC

Βάσει των όσων έχουν περιγραφεί στην παράγραφο 2.2.3, το σώμα στο οποίο θα γίνουν οι πράξεις που απαιτούνται από τον αλγόριθμο Reed-Solomon, για block μεγέθους w bit, θα είναι το $GF(2^w)$. Επειδή τα στοιχεία του $GF(2^w)$ είναι πολυώνυμα με συντελεστές στο Z_2 και βαθμό μικρότερο του w, θα αντιστοιχίζουμε τις τιμές των block με τα στοιχεία του $GF(2^w)$ ως εξής: Η τιμή του i-οστού bit του block θα αντιστοιχεί στην τιμή του συντελεστή του x^i όρου του πολυωνύμου, όπου i=0,1,2,…,w-1. Επομένως, οι πράξεις θα γίνουν σύμφωνα με το σώμα $GF(2^w)$ (το οποίο είναι ισόμορφο με το $Z_2[x]/(p(x))$ για p(x) βαθμού w). Η υλοποίηση αυτών φαίνεται στη συνέχεια. Επισημαίνεται ότι στο σώμα Z_2, η πρόσθεση ορίζεται με τη λογική πράξη XOR και ο πολλαπλασιασμός με την AND.

Η πράξη της πρόσθεσης στο $GF(2^w)$, σύμφωνα με την προηγούμενη ενότητα, υλοποιείται προσθέτοντας τα αντίστοιχα πολυώνυμα. Επειδή όμως οι συντελεστές των πολυωνύμων αποτελούν στοιχεία του σώματος Z_2, στην ουσία εκτελείται η πράξη XOR στους αντίστοιχους συντελεστές των πολυωνύμων. Άρα, δεδομένης της αντιστοίχησης των τιμών των block στα στοιχεία του $GF(2^w)$, η υλοποίηση της πράξης της πρόσθεσης γίνεται εκτελώντας την πράξη XOR στα bit των αντίστοιχων block.

Επειδή το σώμα έχει χαρακτηριστική 2, η πράξη της αφαίρεση του σώματος είναι η ίδια με την πράξη της πρόσθεση, που είδαμε προηγουμένως. Αυτό συμβαίνει διότι όταν η χαρακτηριστική είναι 2 σημαίνει ότι 2*1=0, άρα και 2*x=0, όπου x οποιοδήποτε στοιχείο του $GF(2^w)$, δηλαδή $x + x = 0 \Rightarrow x = -x$. Επομένως $y - x = y + (-x) = y + x$, όπου x, y δύο οποιαδήποτε στοιχεία του $GF(2^w)$.

Η πράξη του πολλαπλασιασμού στο $GF(2^w)$, σύμφωνα με τα όσα έχουμε ήδη ειπωθεί, υλοποιείται πολλαπλασιάζοντας, αρχικά, τα αντίστοιχα πολυώνυμα και στη συνέχεια διαιρώντας το αποτέλεσμα αυτό με το πολυώνυμο p(x) (του σώματος $Z_2[x]/(p(x))$ που είναι ισόμορφο με το $GF(2^w)$) και παίρνοντας ως αποτέλεσμα της πράξης του πολλαπλασιασμού μας, το υπόλοιπο της διαίρεσης αυτής (υπενθυμίζουμε ότι οι πράξεις στους συντελεστές των πολυωνύμων γίνονται στο Z_2, στο οποίο και ανήκουν).

Η πράξη της διαίρεσης στο $GF(2^w)$ πραγματοποιείται μέσω της πράξης του πολλαπλασιασμού που είδαμε προηγουμένως. Πιο συγκεκριμένα, για να υπολογιστεί το πηλίκο x/y, όπου x, y στοιχεία του $GF(2^w)$, υπολογίζεται ο αντίστροφος y^{-1} του y (ο οποίος υπάρχει αφού έχουμε να κάνουμε με σώμα) και παίρνουμε το ζητούμενο πηλίκο από τον πολλαπλασιασμό $x*y^{-1}$. Όμως, ο αλγόριθμος που θα ακολουθηθεί για την

υλοποίηση της διαίρεσης, όπως και του πολλαπλασιασμού, θα είναι διαφορετικός και θα αναλυθεί στη συνέχεια.

Μπορεί σχετικά εύκολα να κατασκευαστεί αλγόριθμος που να υλοποιεί την πράξη του πολλαπλασιασμού, έτσι όπως περιγράφηκε προηγουμένως. Το πρόβλημα είναι η δημιουργία του αλγορίθμου για τη διαίρεση. Για να υλοποιηθεί η διαίρεση θα πρέπει, σύμφωνα με όσα έχουν προαναφερθεί, να υπολογιστεί αρχικά ο αντίστροφος του διαιρέτη και, έτσι, η διαίρεση να μετατραπεί σε πολλαπλασιασμό. Ο αντίστροφος ενός στοιχείου r του GF(2^w) θεωρείται το στοιχείο r΄ του GF(2^w) για το οποίο ισχύει: το υπόλοιπο της διαίρεσης του γινομένου των r και r΄ με το p είναι ίσο με 1. Η διαδικασία, όμως, αυτή δεν είναι αντιστρέψιμη και ο υπολογισμός του αντιστρόφου μπορεί να γίνει μόνο μέσω δοκιμών από το σύνολο των στοιχείων του σώματος. Οι δοκιμές μπορεί να αναφέρονται είτε στη θέση του r΄, όπου στην περίπτωση αυτή θα πρέπει κάθε φορά να διατρέχεται όλο το σώμα, είτε στη θέση π του πηλίκου της διαίρεσης του γινομένου rr΄ με το p (το r΄ υπολογίζεται βάση του τύπου: $r \cdot r' = \pi \cdot p + 1 \Rightarrow r' = \frac{\pi \cdot p + 1}{r}$), όπου εδώ μικραίνει το εύρος της δοκιμής, εφόσον το π είναι πολυώνυμο βαθμού το πολύ w-2 (έχοντας να κάνουμε σώμα GF(2^w)). Και στις δύο περιπτώσεις, πάντως, η διαδικασία είναι ασύμφορη, σπαταλώντας υπολογιστική ισχύ και χρόνο, ιδίως καθώς το μέγεθος του σώματος αυξάνεται.

Οι πράξεις, όμως, του πολλαπλασιασμού και της διαίρεσης μπορεί να γίνουν πολύ εύκολα, παρατηρώντας ότι τα μη-μηδενικά στοιχεία του GF(2^w) αντιστοιχούν όλα σε δυνάμεις ενός στοιχείου (γεννήτορα) x τάξεως 2^w-1, όπως είδαμε και προηγουμένως. Συνεπώς, οι πράξεις του πολλαπλασιασμού και της διαίρεσης μπορούν να αναχθούν σε πράξεις δυνάμεων. Συγκεκριμένα, ο πολλαπλασιασμός μπορεί να διεξαχθεί προσθέτοντας τους εκθέτες των δυνάμεων, στους οποίους αντιστοιχούν οι δύο πολλαπλασιαστικοί όροι, και η διαίρεση αφαιρώντας του αντίστοιχους εκθέτες (αφαιρείται ο εκθέτης της δύναμης που αντιστοιχεί στον διαιρέτη από τον εκθέτη της δύναμης που αντιστοιχεί στο διαιρετέο). Η πολλαπλασιαστική ομάδα του GF(2^w), όμως, είναι κυκλικής τάξεως 2^w-1, άρα για να αναχθούν με ακρίβεια οι πράξεις του πολλαπλασιασμού και της διαίρεσης στις αντίστοιχες πράξεις εκθετών, θα πρέπει στην περίπτωση που το άθροισμα των εκθετών προκύπτει μεγαλύτερο του 2^w-1, να παίρνουμε σαν αποτέλεσμα του πολλαπλασιασμού τη δύναμη με εκθέτη το άθροισμα των εκθετών modulo 2^w-1 (δηλαδή το υπόλοιπο της διαίρεσης του αθροίσματος των εκθετών με το 2^w-1). Στην περίπτωση, δε, που η διαφορά των εκθετών είναι μικρότερη του μηδενός, να παίρνουμε ως αποτέλεσμα της διαίρεσης τη δύναμη με εκθέτη το άθροισμα της διαφοράς με το 2^w-1, ώστε να προκύπτει θετικός εκθέτης μικρότερος του 2^w-1 (που πάντα θα προκύπτει, αφού η διαφορά δεν μπορεί να είναι μικρότερη του –(2^w-1), διότι οι εκθέτες μπορούν να πάρουν τιμές στο διάστημα [0, 2^w-1]). Επομένως, οι πράξεις του πολλαπλασιασμού και της διαίρεσης μπορούν να υλοποιηθούν πολύ εύκολα με μία

πρόσθεση και μία αφαίρεση αντίστοιχα, αρκεί να μπορούμε να αντιστοιχίζουμε κάθε φορά το κάθε στοιχείο του $GF(2^w)$ στην αντίστοιχη δύναμη του γεννήτορά (x) του (για την ακρίβεια στον εκθέτη της δύναμη) και αντίστροφα.

Ο πιο εύκολος και γρήγορος τρόπος να γίνει αυτή η αμφίδρομη αντιστοίχηση μεταξύ στοιχείων $GF(2^w)$ και των δυνάμεων του γεννήτορα (x), οποιαδήποτε στιγμή χρειαστεί, είναι η κατασκευή δυο πινάκων, του gflog και gfilog. Ο πίνακας gflog θα προσομοιώνει τη λειτουργία του λογαρίθμου (δηλαδή την αντιστοίχηση του στοιχείου του $GF(2^w)$ με τη δύναμη του γεννήτορα). Επομένως, ο gflog θα έχει μέγεθος 2^w και θα συμπληρωθεί με τέτοιο τρόπο, ώστε στη θέση gflog[y] να αναγράφεται η δύναμη του γεννήτορα που δίνει το στοιχείο του $GF(2^w)$, το οποίο αντιστοιχεί στη δυαδική τιμή y του block μνήμης. Για την ακρίβεια, ο gflog χρειάζεται να έχει μέγεθος 2^w-1, αφού το μηδενικό στοιχείο του $GF(2^w)$ δεν μπορεί να εκφραστεί ως δύναμη του γεννήτορα x, παρόλα αυτά, όμως, διατηρείται το μέγεθος του σε 2^w για ευκολότερη αντιστοίχηση μεταξύ των τιμών των block και των δυνάμεων του γεννήτορα. Αντίστοιχα, ο πίνακας gfilog θα προσομοιώνει την αντίστροφη λειτουργία του λογαρίθμου (δηλαδή την αντιστοίχηση της δύναμης του γεννήτορα με το στοιχείο του $GF(2^w)$). Θα έχει μέγεθος 2^w-1 και θα συμπληρωθεί με τέτοιο τρόπο ώστε στη θέση gflog[z] να αναγράφεται η δυαδική τιμή του block μνήμης που αντιστοιχεί στο μη-μηδενικό στοιχείο x^z του $GF(2^w)$ (όπου x ο γεννήτορας της πολλαπλασιαστικής ομάδας του $GF(2^w)$).

Για τη συμπλήρωση των πινάκων gflog και gfilog διατρέχουμε τα μη-μηδενικά στοιχεία x^z του $GF(2^w)$ σειριακά, βάσει του z. Για να υπολογίσουμε τιμές των blocks που αντιστοιχούν στα αντίστοιχα στοιχεία x^z του $GF(2^w)$, θα πρέπει αρχικά να αντιστοιχίσουμε τα τελευταία με τα πολυωνυμικά στοιχεία του $GF(2^w)$. Η μέθοδος για να γίνει η τελευταία αντιστοιχία είναι να διαιρούμε το x^z με το μονικό ανάγωγο πολυώνυμο f(x) του σώματος $Z_2[x]/(f(x))$, βαθμού w, ώστε το $Z_2[x]/(f(x))$ να είναι ισόμορφο του $GF(2^w)$, και παίρνουμε το υπόλοιπο της διαίρεσης αυτής, το οποίο θα είναι πολυώνυμο βαθμού μικρότερου του w, αφού το f(x) είναι βαθμού w. Στην περίπτωση αυτή, όμως, θα έπρεπε να εκτελέσουμε 2^w-z διαιρέσεις, οι οποίες θα αποκτούν όλο και μεγαλύτερο βαθμό δυσκολίας καθώς το z θα ανεβαίνει. Για το λόγο αυτό και η μέθοδος που επιλέγεται για να πάρουμε τα στοιχεία του $GF(2^w)$, με τη σειρά που υπαγορεύεται από το σειριακή προσπέλαση των στοιχείων x^z , είναι: αντί να παίρνουμε το στοιχείο x^z και κάθε φορά να αυξάνουμε την τιμή του z κατά ένα, να ξεκινήσουμε με το στοιχείο x και κάθε φορά να πολλαπλασιάζουμε το προηγούμενο στοιχείο με x αντιμετωπίζοντάς τα ως πολυώνυμα. Προφανώς, ο πολλαπλασιασμός αυτός με το x για τη δυαδική τιμή του block μνήμης ισοδυναμεί με μία μετατόπιση των ψηφίων του κατά μία θέση αριστερότερα. Όταν προκύπτει πολυώνυμο τάξης w, το διαιρούμε με το f(x) το οποίο είναι πολυώνυμο βαθμού w, και, για το βήμα αυτό, παίρνουμε για στοιχείο του $GF(2^w)$ το υπόλοιπο της διαίρεσης αυτής. Ο υπολογισμός του υπόλοιπου της διαίρεσης, στην περίπτωση αυτή (στην οποία διαιρετέος και διαιρέτης είναι του ίδιου βαθμού και οι

συντελεστές των πολυωνύμων ανήκουν στο σώμα Z_2), γίνεται απλά, εκτελώντας ένα XOR μεταξύ των bit των block μνήμης του διαιρετέου και του διαιρέτη. Συνεπώς, σε κάθε βήμα για z = i τοποθετούμε στη θέση gfilog[i] την τιμή του υπολογίζεται (val) και στη θέση gflog[val] την τιμή του i. Με τον τρόπο αυτό συμπληρώνονται οι πίνακες gflog και gfilog.

3.1.2 Μεθοδολογία κατασκευής του πίνακα B για την κωδικοποίηση Reed-Solomon

Σημαντικό κομμάτι στο σχεδιασμό του RAID EC είναι ο καθορισμός της μεθοδολογίας κατασκευής του πίνακα B για n+m δίσκους και σκοπός αυτής της παραγράφου είναι η παρουσίαση της μεθοδολογίας υπολογισμού του πίνακα αυτού.

Όπως έχει ήδη αναφερθεί στην παράγραφο 2.2.1, ο πίνακας B σχηματίζεται μετά από μια σειρά διαδοχικών μετασχηματισμών στον παρακάτω $(n+m)\times n$ πίνακα Vendermonde:

$$\begin{bmatrix} 0^0(=1) & 0^1(=0) & 0^2(=0) & \cdots & 0^{n-1}(=0) \\ 1^0 & 1^1 & 1^2 & \cdots & 1^{n-1} \\ 2^0 & 2^1 & 2^2 & \cdots & 2^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ (n+m-1)^0 & (n+m-1)^1 & (n+m-1)^2 & \cdots & (n+m-1)^{n-1} \end{bmatrix}$$

Οι παραπάνω μετασχηματισμοί, οι οποίοι απαιτούνται για να μετατρέψουν τις n πρώτες γραμμές του παραπάνω Vendermonde πίνακα σε μοναδιαίο πίνακα, παρουσιάζονται στη συνέχεια:

1. Υποθέτουμε ότι οι πρώτες i-1 γραμμές, i<n, του πίνακα βρίσκονται ήδη στη σωστή μορφή για το σχηματισμό του μοναδιαίου πίνακα στις n γραμμές. Στο κάθε βήμα θα μετατρέπεται η γραμμή i στην κατάλληλη γραμμή για το μοναδιαίο $n\times n$ πίνακα, δηλαδή θα είναι όλα τα στοιχεία της γραμμής μηδενικά, εκτός του i-οστού στοιχείου της γραμμής, το οποίο πρέπει να είναι ίσο με την μονάδα. Κατά αυτό τον τρόπο, στην περίπτωση που το i-οστό στοιχείο της γραμμής i είναι ίσο με το μηδέν, εντοπίζεται μία στήλη j με j>i, ώστε το j-οστό στοιχείο της γραμμής i να είναι μη-μηδενικό, και ανταλλάσουμε τη στήλη i με τη στήλη j. Μια τέτοια στήλη j είναι σίγουρο ότι υπάρχει, διαφορετικά οι πρώτες n γραμμές του πίνακα

δε θα δημιουργούσαν έναν αντιστρέψιμο πίνακα, πράγμα το οποίο είναι άτοπο, αφού εξορισμού κάθε υποπίνακας ενός Vendermonde πίνακα, ο οποίος σχηματίζεται με τη διαγραφή m γραμμών, είναι αντιστρέψιμος. Επιπρόσθετα, επειδή j>i, η ανταλλαγή των στηλών i και j δεν μεταβάλλουν τις i-1 πρώτες γραμμές του πίνακα. Στο βήμα αυτό οι i-1 πρώτες γραμμές έχουν όλα τα κ-οστά στοιχεία τους με k>i-1ίσο με μηδέν.

2. Έστω ότι $f_{i,i}$ η τιμή του i-οστού στοιχείου της i-οστής γραμμής και $f_{i,i}^{-1}$ ο πολλαπλασιαστικός αντίστροφος του $f_{i,i}$, δηλαδή $f_{i,i} * f_{i,i}^{-1} = 1$, ο οποίος σίγουρα υπάρχει αφού $f_{i,i} \neq 0$ έπειτα το βήμα 1. Στην περίπτωση τώρα που $f_{i,i} \neq 1$, αντικαθιστούμε τη στήλη C_i με την $f_{i,i}^{-1} * C_i$.
3. Πλέον έχει επιτευχθεί το i-οστό στοιχείο της γραμμής i να είναι ίσο με την μονάδα, δηλαδή $f_{i,i} = 1$. Στη συνέχεια, για κάθε στήλη $j \neq i$ και j-οστό στοιχείο της γραμμής i που είναι διάφορο του μηδενός, $f_{i,j} \neq 0$, αντικαθιστούμε τη στήλη C_j με $C_j - f_{i,j}C_i$. Στο τέλος αυτού του βήματος οι i πρώτες γραμμές του πίνακα θα είναι ταυτόσημες με τις i πρώτες γραμμές ενός μοναδιαίου πίνακα $n \times n$ (i<n) και με τον πίνακα να εξακολουθεί να διατηρεί την ιδιότητα οι $n \times n$ υποπίνακες του να είναι αντιστρέψιμοι.
4. Επαναλαμβάνονται τα παραπάνω βήματα για όλες τις n πρώτες γραμμές του πίνακα, σχηματίζοντας κατά αυτό τον τρόπο ένα μοναδιαίο πίνακα με της n πρώτες γραμμές και ολοκληρώνοντας έτσι τη δημιουργία του πίνακα B.

3.1.3 Μεθοδολογία ανάκτησης χαμένων δίσκων δεδομένων

Στα πλαίσια του RAID EC, η περίπτωση απώλειας δίσκου που απαιτεί ανάκτηση δεδομένων, είναι η απώλεια δίσκων δεδομένων. Έτσι, στην περίπτωση που k δίσκοι δεδομένων και p πλεονάζοντες δίσκοι με $k + p \leq m$ έχουν χαθεί, η διαδικασία από το RAID EC που ακολουθείται για την ανάκτηση των k δίσκων δεδομένων είναι η ακόλουθη.

Καταρχήν, αφαιρούνται από τους πίνακες B και R οι k+p γραμμές που αντιστοιχούν στους χαμένους δίσκους. Από τη συνάρτηση που δημιουργείται, κατασκευάζεται ένα σύστημα εξισώσεων από τις πρώτες k γραμμές των τελευταίων m-p γραμμών των πινάκων D και R ($k \leq m - p$, αφού $k + p \leq m$).

Με αυτό τον τρόπο σχηματίζεται ένα σύστημα k εξισώσεων με k αγνώστους. Για να είναι δυνατή η εφαρμογή της μεθόδου της απαλοιφής Gauss, όπως αυτή περιγράφηκε στην παράγραφο 2.2.3, μεταφέρουμε τις σταθερές των εξισώσεων του συστήματος στο αριστερό μέρος αυτών. Ως αποτέλεσμα δημιουργείται ένα σύστημα εξισώσεων που μπορεί να απεικονιστεί ως Ax=b με A τετραγωνικό $k \times k$ πίνακα και ομαλό ($|A| \neq 0$), αφού η απόλυτη τιμή της ορίζουσα του A έχει την ίδια τιμή με την απόλυτη τιμή της ορίζουσας του υποπίνακα του πίνακα B, ο οποίος έχει δημιουργηθεί αφαιρώντας τις αντίστοιχες k+p γραμμές. Το γεγονός ότι οι δύο ορίζουσες είναι ταυτόσημες, οφείλεται στο ότι οι n-k πρώτες γραμμές του υποπίνακα του πίνακα B προέρχονται από γραμμές μοναδιαίου πίνακα. Επομένως, η ορίζουσα υπολογίζεται από την ορίζουσα του υποπίνακα, ο οποίος σχηματίζεται αφαιρώντας τις n-k πρώτες γραμμές και τις στήλες, οι οποίες γραμμές έχουν τιμή ίση με τη μονάδα, πολλαπλασιασμένη με +1 ή -1 ανάλογα με το ποιο από τα στοιχεία των n-k πρώτων γραμμών είναι ίσο με την μονάδα. Ο υποπίνακας, όμως, αυτός ταυτίζεται με τον πίνακα A και επομένως η ορίζουσα του δεν μπορεί να είναι μηδενική. Συνεπώς, χρησιμοποιώντας το σύστημα Ax=b και τη μεθοδολογία απαλοιφής του Gauss είναι δυνατή η ανάκτηση της χαμένης πληροφορίας των απολεσθέντων k δίσκων.

3.2 Αρχιτεκτονική του RAID EC

Όπως τα RAID συστήματα 5 και 6, έτσι και το RAID EC χρησιμοποιεί κατανομή των δεδομένων προς αποθήκευση, μαζί με τα parities, σε ακολουθίες τμημάτων πληροφορίας που ονομάζονται chunk, ανά stripe. Και εδώ, για να αποφευχθεί η συμφόρηση κάποιων συγκεκριμένων δίσκων, λόγω της απαίτησης της ενημέρωσης των parities σε κάθε εγγραφή, τα parities κατανέμονται στο σύνολο των δίσκων. Η κατανομή των parities έχει σαν αποτέλεσμα τα δεδομένα να μοιράζονται σε όλους τους δίσκους (n+m), και όχι μόνο σε n δίσκους, και ο τρόπος κατανομής των parities να επηρεάζει σε σημαντικό βαθμό την επίδοση του συστήματος. Η μέθοδος κατανομής που επιλέγεται, όπως και στα RAID 5 και 6, είναι η αριστερά συμμετρική (left-symmetric), η οποία κατά τους Lee και Katz είναι η βέλτιστη. Το πλεονέκτημα της μεθόδου αυτής είναι ότι, πάντα, όταν διατρέχονται σειριακά οι ακολουθίες χρήστη των συστοιχιών, θα συναντάται πρώτα κάθε δίσκος μια φορά πριν συναντηθεί κάποιος για δεύτερη. Αυτό μεταφράζεται σε μείωση της συμφόρησης αιτήσεων I/O στους δίσκους και καλύτερη απόδοση. Ένα παράδειγμα εφαρμογής της κατανομής αυτής παρουσιάζεται στον παρακάτω πίνακα (Πίνακας 1), όπου θεωρείται n=5 και m=3.

Πίνακας 1: Κατανομή parities στο RAID EC (n=5, m=8)

Stripes \ Disks	D0	D1	D2	D3	D4	D5	D6	D7
S0	**0**	**1**	**2**	**3**	**4**	**P1**	**P2**	**P3**
S1	**8**	**9**	**P1**	**P2**	**P3**	**5**	**6**	**7**
S2	**P2**	**P3**	**10**	**11**	**12**	**13**	**14**	**P1**
S3	**16**	**17**	**18**	**19**	**P1**	**P2**	**P3**	**15**
S4	**24**	**P1**	**P2**	**P3**	**20**	**21**	**22**	**23**
S5	**P3**	**25**	**26**	**27**	**28**	**29**	**P1**	**P2**
S6	**32**	**33**	**34**	**P1**	**P2**	**P3**	**30**	**31**
S7	**P1**	**P2**	**P3**	**35**	**36**	**37**	**38**	**39**
S8	**40**	**41**	**42**	**43**	**44**	**P1**	**P2**	**P3**

Οι βασικές διεργασίες του RAID EC, όπως και σε κάθε RAID σύστημα, είναι η ανάγνωση δεδομένων (read), η εγγραφή δεδομένων (write), η διαδικασία υποβάθμισης (degrade), η οποία εξασφαλίζει τη σωστή λειτουργία του RAID EC στην περίπτωση απώλειας ενός ή περισσότερων δίσκων, και η αποκατάσταση χαμένου δίσκου με ένα νέο ελεύθερο (spare) δίσκο (recover), ώστε να επανέλθει στην κανονική λειτουργία το σύστημα με όλους τους n+m δίσκους του διαθέσιμους. Οι διεργασίες αυτές αναπτύσσονται περισσότερο στις επόμενες παραγράφους.

Επίσης, κατά την αρχικοποίηση ενός RAID EC σε ένα υπολογιστικό σύστημα λαμβάνουν χώρα δυο επιπρόσθετες διεργασίες: ο έλεγχος εφικτότητας λειτουργίας του RAID EC (check) και η αρχικοποίηση του συστήματος (init).

Κατά τη διεργασία του check εκτελούνται δύο βασικοί έλεγχοι, οι οποίοι έχουν εξαιρετική σημασία για τη βιωσιμότητα του συστήματος: ο αριθμός των διαθέσιμων δίσκων για το σύστημα, όπου στην περίπτωση που είναι ένας ή κανένας το RAID EC δεν ενεργοποιείται, και το μέγεθος του chunk (chunksize), το οποίο πρέπει να είναι δύναμη του 2 και μεγαλύτερο του μηδενός, ώστε να μπορεί να φιλοξενήσει δεδομένα, καθώς, σε διαφορετική περίπτωση, το σύστημα αποτυγχάνει.

Στα πλαίσια της διεργασίας init λαμβάνουν μέρος οι επόμενες ενέργειες:

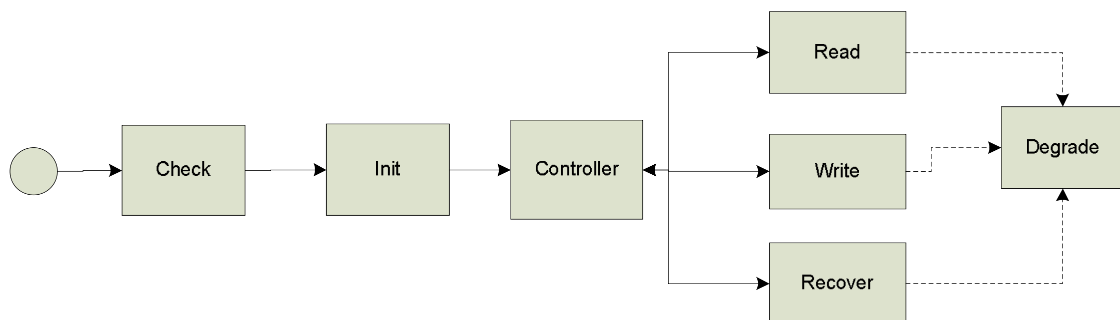
Κανονικοποιούνται τα μεγέθη των δίσκων που θα χρησιμοποιηθούν από το RAID EC, δηλαδή ελέγχεται το μέγεθος όλων των δίσκων και δηλώνονται όλοι στο μέγεθος του μικρότερου, ώστε να είναι δυνατές οι όποιες λειτουργίες του συστήματος. Το RAID EC, όπως και κάθε RAID σύστημα, υποθέτει ότι όλοι οι δίσκοι του έχουν το ίδιο μέγεθος, ώστε να είναι δυνατή η κατανομή των δεδομένων και ο καθορισμός του αριθμού των stripes. Στη συνέχεια γίνεται η αρχικοποίηση κάποιων από τις βασικότερες παραμέτρους του συστήματος, βάσει των παραμέτρων που δόθηκαν κατά την ενεργοποίηση αυτού.

Κατασκευάζονται οι βασικοί πινάκες gflog και gfilog (παράγραφος 2.2.3), ώστε να μπορούν να γίνονται οι πράξεις στο σώμα Galois και ο πίνακας B της κωδικοποίησης Reed-Solomon, καλώντας τις κατάλληλες διεργασίες. Τέλος, κρίνεται ότι το σύστημα λειτουργεί στο κανονικό του mode χωρίς να έχει απώλεια κάποιου δίσκου (λεπτομερέστερη αναφορά στα mode λειτουργίας του συστήματος θα γίνει κατά την ανάπτυξη της διεργασίας degrade).

O RAID EC, που έχει σχεδιαστεί στην παρούσα εργασία, έχει σαν βασικές παραμέτρους ενεργοποίησης τον αριθμό των δίσκων με τον οποίο θα λειτουργεί (δηλαδή το n+m), τον αριθμό (m) των δίσκων που θα χρησιμοποιηθούν ως πλεονάζοντες για την καταχώρηση της επιπρόσθετης πληροφορίας των parities, τον αριθμό των spare δίσκων, το μέγεθος των chunks και το μέγεθος των sector, τα οποία αποτελούν υπομονάδα των chunks και το μικρότερο τμήμα μνήμης που επιτρέπεται να διαχειρίζεται το RAID EC σύστημα. Το σύστημα έχει σχεδιαστεί έτσι, ώστε ο διαχειριστής του να έχει τη δυνατότητα να επιλέξει τιμές αυτών των παραμέτρων με όποιον τρόπο κρίνει ότι θα αποδώσει καλύτερα το σύστημα, καθώς οι τιμές αυτές το επηρεάζουν σημαντικά.

Η ροή των προαναφερόμενων διεργασιών του RAID EC συνοψίζεται στο παρακάτω διάγραμμα (Σχήμα 20). Το σύστημα έχει σχεδιαστεί, όπως φαίνεται και στο σχετικό διάγραμμα, με τέτοιο τρόπο, ώστε οι τέσσερεις βασικές διεργασίες του (read, write, recover, degrade) να μπορούν να εκτελούνται παράλληλα για να είναι σε θέση να διαχειρίζονται πολλαπλά αιτήματα. Επιπρόσθετα, για τον ίδιο λόγο, οι διεργασίες read, write και degrade είναι σχεδιασμένες να εκτελούνται παράλληλα με τον εαυτό τους. Μπορούν να εκτελούνται παράλληλα περισσότερα του ενός read, write ή degrade, ενώ πάντα μπορεί να εκτελείται μόνο ένα recover. Ο λόγος που συμβαίνει αυτό είναι ότι οι read, write και degrade εξυπηρετούν τα αιτήματα ανάγνωσης και εγγραφής από και προς το RAID EC, ενώ το recover αποκαθιστά ολόκληρο το σύστημα ύστερα από απώλεια ενός ή περισσότερων δίσκων και εκτελείται στο υπόβαθρο του συστήματος.

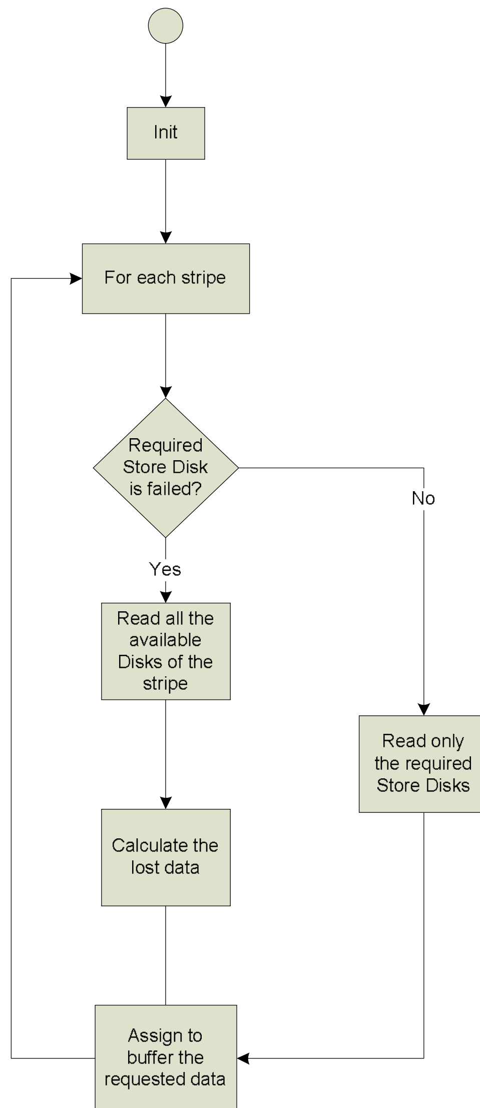
O controller, ο οποίος εμφανίζεται στο Σχήμα 20, είναι η κύρια διεργασία του συστήματος, η οποία δέχεται τα εκάστοτε αιτήματα ανάγνωσης και εγγραφής από και προς το RAID EC και καλεί τις κατάλληλες διεργασίες προς ικανοποίηση αυτών. Επιπλέον, ελέγχει τις απαντήσεις των διεργασιών, τις οποίες και καλεί στην περίπτωση εμφάνισης απώλειας δίσκου, και σε συνάρτηση της καταστάσεως στην οποία βρίσκεται το σύστημα, μπορεί είτε να ενεργοποιήσει την αποκατάσταση αυτού (καλεί την recover), είτε να τερματίσει το σύστημα ως εσφαλμένο.



Σχήμα 20: Διάγραμμα ροή RAID EC

3.2.1 Ανάγνωση δεδομένων από το RAID EC (Read)

Η διεργασία read του RAID EC είναι η υπεύθυνη για την ανάγνωση δεδομένων από το RAID σύστημα. Για να επιτευχθεί αυτό ακολουθούνται δυο μέθοδοι: ανάγνωση των ζητούμενων δεδομένων κατευθείαν από τους δίσκους που τα περιέχουν, εφόσον κανένας από αυτούς δεν έχει χαθεί, ενώ στην αντίθετη περίπτωση διαβάζονται όλοι οι δίσκοι δεδομένων, που είναι ακόμη διαθέσιμοι, και τόσα parity, όσοι είναι οι χαμένοι δίσκοι δεδομένων, ώστε να σχηματιστεί το σύστημα εξισώσεων που περιγράφεται στην παράγραφο 3.1.3 και να ανακτηθούν τα χαμένα δεδομένα. Στο παρακάτω διάγραμμα (Σχήμα 21) περιγράφεται η ροή της διεργασία read.



Σχήμα 21: Διάγραμμα ροή της διεργασίας read

Λόγο του τρόπου κατανομής των δεδομένων και των parities στους δίσκους ανά stripe, η διεργασία read έχει σχεδιαστεί με τρόπο που να είναι προσανατολισμένη ανά stripe. Όλες οι διεργασίες της έχουν προδιαγραφεί να εκτελούνται ανά stripe. Πιο συγκεκριμένα, αρχικά εντοπίζει τα stripes του συστήματος RAID EC στα οποία βρίσκεται η αιτούμενη πληροφορία και στη συνέχεια για κάθε stripe ελέγχει, πρωταρχικά, αν οι χαμένοι δίσκοι είναι δίσκοι δεδομένων, οι οποίοι περιέχουν δεδομένα που έχουν κλιθεί να αναγνωστούν. Στην περίπτωση που για το συγκεκριμένο stripe οι δίσκοι δεδομένων που περιέχουν τη ζητούμενη πληροφορία είναι διαθέσιμοι, τότε απλά γίνεται η ανάγνωση των συγκεκριμένων chucks (ή υποσυνόλου αυτών αναλόγως του όγκου των δεδομένων που ζητούνται από το συγκεκριμένο stripe) από τους δίσκους αυτούς. Στην αντίθετη

περίπτωση, γίνεται ανάγνωση όλων των αντίστοιχων chucks (ή υποσυνόλου αυτών) των διαθέσιμων δίσκων δεδομένων για το δεδομένο stripe, όπως και τόσων διαθέσιμων parities του stripe, όσοι είναι οι δίσκοι δεδομένων που έχουν χαθεί στο stripe αυτό, με σκοπό να μπορεί να ανακτηθεί η χαμένη πληροφορία. Όπως είναι κατανοητό, και οι υπολογισμοί για την ανάκτηση των χαμένων δεδομένων γίνονται ανά stripe. Κατά αυτό τον τρόπο, επαναλαμβάνοντας την παραπάνω διαδικασία για κάθε stripe στα οποία αναφέρεται η ζητούμενη πληροφορία, επιτυγχάνεται η ικανοποίηση του αιτήματος για ανάγνωση δεδομένων από το RAID EC σύστημα.

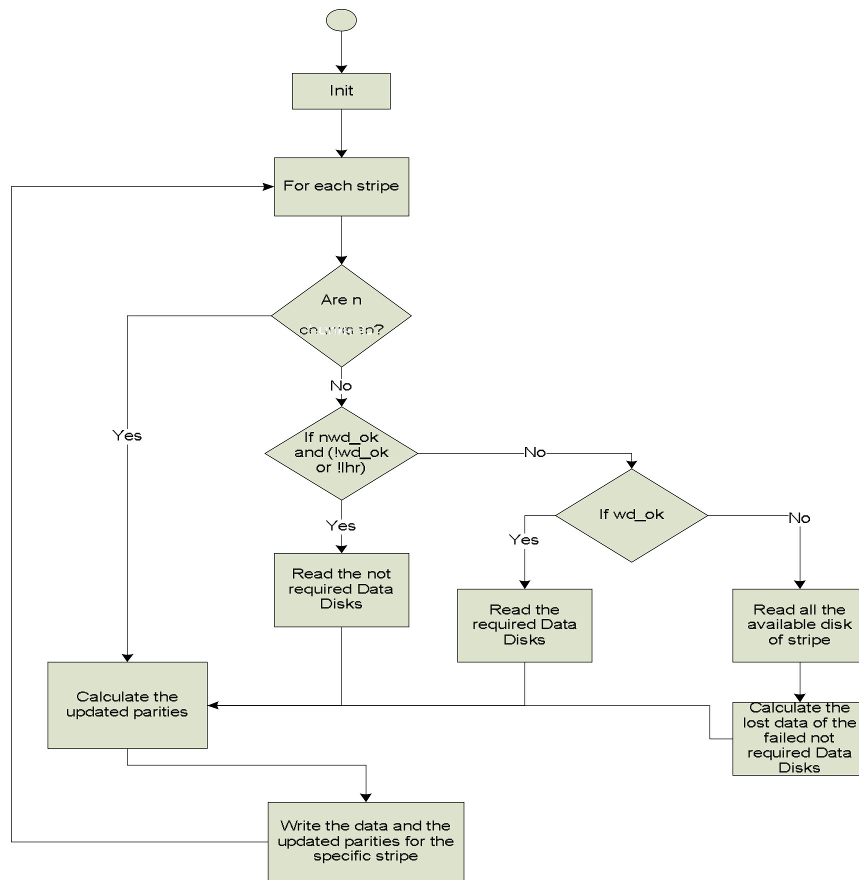
Στην περίπτωση κατά την οποία χαθεί κάποιος δίσκος εν εξελίξει της διεργασίας read, το οποίο γίνεται αντιληπτό από την αποτυχία ανάγνωσης από το δίσκο αυτό, τότε η συγκεκριμένη διεργασία ακυρώνεται και ανάλογο μήνυμα σφάλματος επιστρέφεται στο σύστημα RAID EC, και πιο συγκεκριμένα στον Controller αυτού, αφού πρώτα έχει γίνει κλήση της διεργασίας degrade. Στην περίπτωση που το σύστημα είναι βιώσιμο, έπειτα από την απώλεια του δίσκου, τότε ο Controller είναι υπεύθυνος για να επαναλάβει το αίτημα ανάγνωσης.

3.2.2 Εγγραφή δεδομένων στο RAID EC (Write)

Η διεργασία write του RAID EC είναι υπεύθυνη για την εγγραφή δεδομένων στο RAID σύστημα. Σε κάθε αίτημα εγγραφή δεδομένων η διεργασία write θα πρέπει να ενημερώνει και τις τιμές των parities, ώστε να διατηρείται η ικανότητα του συστήματος να ανακτά τα χαμένα δεδομένα ύστερα από απώλεια δίσκων. Αυτό μπορεί να επιτευχθεί με δύο τρόπους. Είτε υπολογίζοντας τα parities από τα δεδομένα όλων των δίσκων, χρησιμοποιώντας τον πίνακα B της κωδικοποίησης Reed-Solomon, είτε χρησιμοποιώντας τα δεδομένα προς εγγραφή, τα δεδομένα που θα γίνουν overwrite και την προηγούμενη τιμή των parities. Ο δεύτερος τρόπος υπολογισμού είναι εφικτός χάρις την πεδιακή άλγεβρα του Galois που, όπως έχει ήδη παρουσιαστεί, βάσει αυτής διεξάγονται όλες οι πράξεις στο RAID EC. Πιο συγκεκριμένα, θεωρώντας ότι μία συστοιχία αποτελείται από n δίσκους δεδομένων: d_1, d_2, d_3, …, d_n και p_j ένα από τα m δίσκους parities (j<m), τότε $p_j = a_{1,j}d_1 + a_{2,j}d_2 + a_{3,j}d_3 + \ldots + a_{n,j}d_n$. Στην περίπτωση που το d_i αλλάξει στο d_i', τότε το νέο parity p_j' υπολογίζεται ως εξής, αφού $x = -x$ για οποιοδήποτε στοιχείο του GF(2^w), του οποίου στοιχεία έχουν θεωρηθεί τα block των δεδομένων του RAID EC :

$$p_j' = a_{1,j}d_1 + a_{2,j}d_2 + \ldots + a_{i,j}d_i' + \ldots + a_{n,j}d_n \Leftrightarrow$$
$$p_j' = \left(a_{1,j}d_1 + \ldots + a_{i-1,j}d_{i-1} + a_{i+1,j}d_{i+1} + \ldots + a_{n,j}d_n\right) + a_{i,j}d_i' \Leftrightarrow$$
$$p_j' = \left(p_j - a_{i,j}d_i\right) + a_{i,j}d_i' \Leftrightarrow p_j' = p_j + a_{i,j}d_i + a_{i,j}d_i'$$

Στο παρακάτω διάγραμμα (Σχήμα 22) παρουσιάζεται η ροή της διεργασίας και ο τρόπος που χρησιμοποιούνται οι παραπάνω διαδικασίες για την επίτευξη σωστής και αποδοτικής εγγραφής δεδομένων στο RAID EC, διατηρώντας παράλληλα την ικανότητα ανάκτησης των χαμένων δεδομένων. Η αποδοτικότητα επιτυγχάνεται περιορίζοντας όσο το δυνατόν περισσότερο τον αριθμό κλήσεων για αναγνώσεις στους δίσκους του RAID EC συστήματος.



Σχήμα 22: Διάγραμμα ροής διεργασίας write (όπου nwd_ok= true, εάν οι δίσκοι δεδομένων που δεν είναι προς εγγραφή είναι διαθέσιμοι, wd_ok= true, εάν οι δίσκοι δεδομένων που είναι προς εγγραφή είναι διαθέσιμοι και lhr=true, εάν οι δίσκοι δεδομένων προς εγγραφή είναι περισσότεροι από $n/2$)

Όπως και στην περίπτωση του read, έτσι και η διεργασία write έχει σχεδιαστεί ώστε να είναι προσανατολισμένη ανά stripe. Πιο συγκεκριμένα, έχοντας εντοπίσει τα stripe του συστήματος RAID EC, στα οποία πρόκειται να αποθηκευτούν τα δεδομένα του αιτήματος, ελέγχει, για κάθε ένα από αυτά, το μέγεθος των δεδομένων που θα γραφτούν στο stripe αυτό, όπως και το αν και ποιοι δίσκοι είναι χαμένοι.

Συνεπώς, στην περίπτωση που για το δεδομένο stripe τα δεδομένα προς εγγραφή καταλαμβάνουν και τους n δίσκους του RAID EC, η διεργασία κλειδώνει για λόγους συγχρονισμού όλους τους δίσκους δεδομένων, όπως και τους δίσκους parity (ή τους διαθέσιμους από αυτούς, εφόσον υπάρχουν εσφαλμένοι), ώστε να μην επιχειρεί πάνω από μια διεργασία να μεταβάλλει την πληροφορία που διατηρεί ένας δίσκος, αφού είναι δυνατόν να εκτελούνται παράλληλα πάνω από μια διεργασία write. Στη συνέχεια υπολογίζονται τα νέα parities για το stripe, χρησιμοποιώντας τα δεδομένα προς εγγραφή και τις εξισώσεις που προκύπτουν από την κωδικοποίηση Reed – Solomon. Τέλος, εγγράφονται τα νέα δεδομένα μαζί με τα νέα parities, ξεκλειδώνοντας παράλληλα τους αντίστοιχους δίσκους (για την ακρίβεια τα αντίστοιχα chunk των δίσκων).

Στην περίπτωση που οι δίσκοι δεδομένων, στους οποίους πρέπει να γραφτεί η εισερχόμενη πληροφορία (βάσει του αιτήματος που έχει δεχτεί το σύστημα RAID EC), είναι διαθέσιμοι και ο αριθμός των δίσκων αυτών είναι μικρότερος του $\frac{n}{2}$ ή κάποιος από τους υπόλοιπους δίσκους δεδομένων έχει αναγνωριστεί ως εσφαλμένος, η write διαβάζει την υπάρχουσα πληροφορία των δίσκων δεδομένων στους οποίους πρόκειται να εγγραφούν τα νέα δεδομένα, όπως και τους διαθέσιμους δίσκους parity, και στη συνέχεια τους κλειδώνει. Χρησιμοποιώντας τη μεθοδολογία που αναπτύχθηκε στην αρχή της παραγράφου αυτής (3.2.2), τα δεδομένα προς εγγραφή και τα δεδομένα που αναγνώσθηκαν στο προηγούμενο βήμα, υπολογίζονται τα νέα parities. Τα νέα δεδομένα μαζί με τα νέα parities εγγράφονται στους κατάλληλους δίσκους, ξεκλειδώνοντας παράλληλα τους δίσκους που προηγουμένως είχαν κλειδωθεί.

Στην περίπτωση που καμία από τις δύο παραπάνω περιπτώσεις δεν ικανοποιείται και οι δίσκοι δεδομένων, στους οποίους δεν προορίζεται να γραφτούν δεδομένα, είναι όλοι διαθέσιμοι, η διεργασία διαβάζει τα περιεχόμενα των δίσκων αυτών και κλειδώνει όλους του δίσκους δεδομένων στο οποίους θα εγγραφούν τα νέα δεδομένα, όπως και τους δίσκους των parities. Χρησιμοποιώντας τα νέα δεδομένα, όσο και εκείνα που αναγνώστηκαν προηγουμένως, υπολογίζονται τα νέα parities. Τα νέα δεδομένα μαζί με τα νέα parities εγγράφονται στους κατάλληλους δίσκους, ξεκλειδώνοντας παράλληλα τους αντίστοιχους δίσκους.

Στην περίπτωση τώρα που δεν ικανοποιείται καμία από τις παραπάνω συνθήκες, σημαίνει ότι υπάρχουν εσφαλμένοι δίσκοι τόσο σε αυτούς που πρόκειται να αποθηκευτούν τα νέα δεδομένα, όσο και στους υπόλοιπους δίσκους δεδομένων. Έτσι, η

write διαβάζει τα περιεχόμενα όλων των διαθέσιμων δίσκων, όπως και όλων των διαθέσιμων δίσκων parities, ενώ παράλληλα κλειδώνει τους δίσκους στους οποίους θα εγγραφούν τα νέα δεδομένα και τα ενημερωμένα parities. Διαθέτοντας όλη την παραπάνω πληροφορία και χρησιμοποιώντας την κωδικοποίηση Reed – Solomon, ανακτάται η παλιά πληροφορία των χαμένων δίσκων δεδομένων, στους οποίους θα εγγραφούν τα νέα δεδομένα. Εν συνεχεία υπολογίζονται τα νέα parities, χρησιμοποιώντας τη διαδικασία που περιγράφηκε και στην $2^η$ περίπτωση. Τα νέα δεδομένα μαζί με τα νέα parities εγγράφονται στους κατάλληλους δίσκους, ξεκλειδώνοντας παράλληλα τους αντίστοιχους δίσκους.

Για να επιτευχθεί η μέγιστη δυνατή επίδοση στο σύστημα RAID EC, τα κλειδώματα που λαμβάνουν χώρα ανά stripe στα παραπάνω βήματα, δεν γίνονται ανά chunk αλλά ανά sector του chunk. Αυτό έχει σαν αποτέλεσμα, εάν δύο διεργασίες write εκτελούνται παράλληλα και έχουν και οι δύο να ενημερώσουν το ίδιο chunk δίσκου, αλλά όχι τα ίδια sectors, να είναι δυνατόν να εκτελούνται παράλληλα χωρίς να περιμένει η μία να ολοκληρωθεί η άλλη.

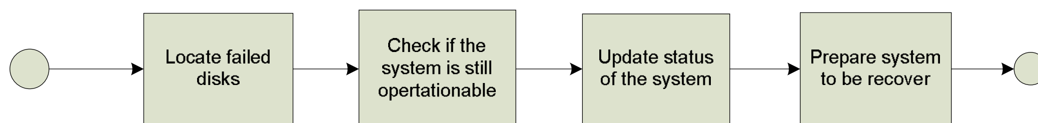
Και στη διεργασία write, στην περίπτωση που χαθεί κάποιος από τους δίσκους όσο αυτή βρίσκεται εν εξελίξει, πράγμα που γίνεται αντιληπτό από την αποτυχία ανάγνωσης ή εγγραφής από το ή στο δίσκο αυτό, τότε η συγκεκριμένη διεργασία τερματίζεται και κατάλληλο μήνυμα σφάλματος επιστρέφεται στο σύστημα RAID EC, και πιο συγκεκριμένα στον Controller αυτού, αφού πρώτα έχει γίνει κλήση της διεργασίας degrade. Στην περίπτωση που το σύστημα είναι βιώσιμο, έπειτα από την απώλεια του δίσκου, τότε ο Controller είναι υπεύθυνος για να επαναλάβει το αίτημα ανάγνωσης. Στο σημείο αυτό πρέπει να επισημανθεί ότι η write έχει σχεδιαστεί με τρόπο που ο τερματισμός της λόγω εντοπισμού εσφαλμένου δίσκου να γίνεται έτσι, ώστε να εξασφαλίζεται η συνέχεια της σωστής λειτουργίας του RAID EC, στην περίπτωση, βέβαια, που εξακολουθεί να είναι βιώσιμο.

Πιο συγκεκριμένα, επειδή στην write τα αιτήματα εγγραφής στον κάθε δίσκο εκτελούνται παράλληλα, στην περίπτωση που εντοπιστεί σφάλμα σε ένα δίσκο από κάποια από τα αιτήματα, τα υπόλοιπα συνεχίζουν κανονικά. Έτσι, πριν επιστρέψει, η διεργασία θα έχει ολοκληρώσει της απαιτούμενες ενέργειες (εγγραφή και ξεκλείδωμα) στους διαθέσιμους δίσκους, όταν φυσικά ο εσφαλμένος δίσκος εντοπιστεί κατά την διαδικασία εγγραφής των νέων δεδομένων και parities στον RAID EC. Στην περίπτωση που εντοπιστεί κατά τη διαδικασία συλλογής της επιπλέον πληροφορία για τον υπολογισμό των νέων parities, σύμφωνα με τις μεθοδολογίες που εξετάστηκαν προηγουμένως, η write διαθέτει μηχανισμό για να ξεκλειδώσει τους δίσκους που κλειδώθηκαν πριν επιστρέψει. Οι παραπάνω ενέργειες είναι απαραίτητες, διότι εάν κάποιος δίσκος παρέμενε κλειδωμένος μετά τον τερματισμό της διεργασίας, δεν θα μπορούσε να ξεκλειδωθεί από καμία άλλη κλήση της διεργασίας write και το συγκεκριμένο τμήμα του δίσκου δεν θα μπορούσε να ξαναενημερωθεί (μια διεργασία

write μπορεί να ξεκλειδώσει μόνο τους δίσκους που έχει κλειδώσει αυτή, ώστε να επιτυγχάνεται συγχρονισμός).

3.2.3 Υποβάθμιση του RAID EC (Degrade)

Η διεργασία degrade του συστήματος RAID EC εκτελείται όποτε εντοπίζεται σφάλμα σε δίσκο του συστήματος και έχει σαν στόχο τη διερεύνηση του κατά πόσο το σύστημα είναι βιώσιμο, την ενημέρωση του συστήματος με το κατάλληλο status, την εκτέλεση των απαραίτητων ενεργειών, ώστε το σύστημα να εξακολουθεί να λειτουργεί με χαμένους δίσκους (εφόσον είναι λιγότεροι από m), και τέλος την προετοιμασία για την αποκατάσταση του συστήματος. Στο σημείο αυτό πρέπει να επισημανθεί ότι στο σχεδιασμό του συστήματος έχει γίνει η υπόθεση, σύμφωνα με την οποία από τη στιγμή που εντοπιστεί σφάλμα έστω και σε μία προσπάθεια ανάγνωσης ή εγγραφής σε ένα δίσκο του συστήματος, ο δίσκος αυτό θεωρείται ότι έχει χαθεί. Στο παρακάτω διάγραμμα (Σχήμα 23) παρουσιάζεται η ροή των ενεργειών της διεργασία degrade.



Σχήμα 23: Διάγραμμα ροής διεργασίας degrade

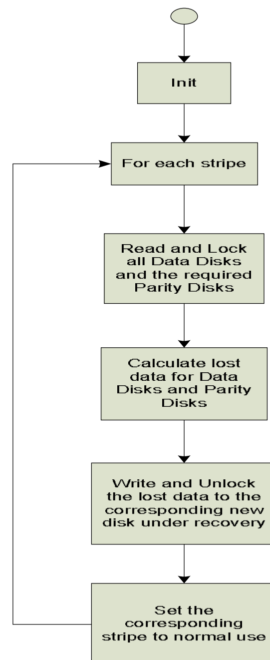
Αναλυτικότερα, η διεργασία του degrade, εκμεταλλευόμενη το γεγονός ότι ο εντοπισμός σφάλματος συνοδεύεται με τον χαρακτηρισμό του συγκριμένου δίσκου ως εσφαλμένου, αρχικά διατρέχοντας τους δίσκους του συστήματος εντοπίζει του εσφαλμένους δίσκους και ενημερώνει κατάλληλα το σύστημα. Λεπτομερέστερα, ελέγχει κατά πόσο το σύστημα εξακολουθεί να είναι βιώσιμο, δηλαδή αν ο αριθμός των εσφαλμένων δίσκων είναι μικρότερος του m. Στη συνέχεια ενημερώνεται το σύστημα, εφόσον αυτό είναι βιώσιμο, για το πόσοι και ποιοι δίσκοι είναι εσφαλμένοι, ώστε να είναι μπορεί να συνεχίζει να λειτουργεί σε degrade mode. Τέλος, εντοπίζει από τους spare δίσκους εκείνους που είναι διαθέσιμοι και τους δηλώνει ως δίσκους που θα αντικαταστήσουν τους χαμένους ύστερα από τη διαδικασία του recover.

Τέλος σημειώνεται ότι, ενώ το RAID EC σύστημα έχει σχεδιαστεί με σκοπό να μπορούν να εκτελούνται παράλληλα περισσότερες της μίας διεργασίας degrade, οι ενέργειες που εκτελούνται από αυτήν πρέπει να εκτελούνται σειριακά, ώστε να σχηματίζεται η σωστή

εικόνα της κατάστασης του συστήματος. Για το λόγο αυτό έχει αναπτυχθεί μηχανισμός συγχρονισμού των διεργασιών μέσω κλειδωμάτων.

3.2.4 Αποκατάσταση του RAID EC (Recover)

Η διεργασία του recover καλείται από το σύστημα του RAID EC και πιο συγκεκριμένα από το σχετικό Controller, όποτε εντοπίζεται ότι κάποιος δίσκο είναι εσφαλμένος, με σκοπό την αποκατάστασή του από την οπτική του συστήματος. Για την ακρίβεια, η διαδικασία που λαμβάνει χώρα είναι η αντικατάσταση του εσφαλμένου δίσκου με ένα νέο, εγγράφοντας την πληροφορία του εσφαλμένου δίσκου. Η διαδικασία αυτή πρέπει να είναι δυναμική και να εκτελείται σε παραλληλία με τις υπόλοιπες διεργασίες του συστήματος, αφού το σύστημα κατά τη διαδικασία της αποκατάστασης παραμένει λειτουργικό. Στο συγκεκριμένο σημείο πρέπει να επισημανθεί ότι κάθε φορά εκτελείται μόνο μια διεργασία recover και ότι στην περίπτωση που εντοπιστεί σφάλμα σε δίσκο του συστήματος, πριν κλιθεί η recover, το σύστημα περιμένει να ολοκληρωθούν τα εν εξελίξει αιτήματα εγγραφής και ανάγνωσης, παγώνοντας την εκτέλεση νέων αιτημάτων. Στόχος είναι όταν κλιθεί η recover να έχουν εντοπιστεί όλοι οι πιθανά εσφαλμένοι δίσκοι. Στο παρακάτω διάγραμμα παρουσιάζεται η ροή των ενεργειών της διεργασίας recover (Σχήμα 24).



Σχήμα 24: Διάγραμμα ροής διεργασίας recover

Αναλυτικότερα, η recover ακολουθώντας τη βασική αρχιτεκτονική του RAID EC, διεξάγει την αποκατάσταση των εσφαλμένων δίσκων ανά stripe. Διατρέχοντας τη συστοιχία των δίσκων του RAID EC ανά stripe, διαβάζει όλους τους διαθέσιμους δίσκους και τους κλειδώνει, λόγω της επικείμενης εγγραφής στους νέους δίσκους προς αντικατάσταση των εσφαλμένων. Χρησιμοποιώντας την κωδικοποίηση Reed-Solomon ανακτάται η χαμένη πληροφορία για το stripe αυτό. Πιο συγκριμένα, αρχικά ανακτούνται τα χαμένα δεδομένα (εφόσον για το δεδομένο stripe υπάρχουν εσφαλμένοι δίσκοι δεδομένων) και ακολούθως, χρησιμοποιώντας και την πληροφορία των υπόλοιπων δίσκων δεδομένων, οι οποίοι είχαν διαβαστεί στο προηγούμενο βήμα, υπολογίζονται και τα χαμένα parities (εφόσον για το δεδομένο stripe υπάρχουν εσφαλμένοι δίσκοι πλεονασμού).

Όταν έχει ολοκληρωθεί η διαδικασία αποκατάστασης για το δεδομένο stripe, το συγκεκριμένο τμήμα των νέων δίσκων παραδίδεται στο σύστημα του RAID EC για κανονική χρήση, ώστε να επιτευχθεί η δυναμική λειτουργία της αποκατάστασης και η κανονική λειτουργία του συστήματος καθ’ όλη τη διάρκεια αυτής. Παραδίδοντας τα ανακτημένα μέρη του νέου δίσκου, δεν είναι υπεύθυνη αυτή η διεργασία για τη συνεχή ενημέρωσή τους, αλλά η ίδια η διεργασία write που θα εκτελεστεί για το stripe αυτό, αφού πλέον ο αναφερόμενος δίσκος δεν θα θεωρείται εσφαλμένος. Βάσει αυτών, φυσικά, γίνεται και αντιληπτή η συνάρτηση της θεώρησης ενός δίσκου ως εσφαλμένου με τα stripes. Η συνάρτηση αυτή, όμως, απλουστεύεται από το γεγονός ότι η ανάκτηση του κάθε stripe γίνεται σειριακά, ξεκινώντας από το 1° stripe της συστοιχίας.

Επομένως, εκτελώντας τις παραπάνω ενέργειες για όλα τα stripes της συστοιχίας δίσκων του RAID EC, η αποκατάσταση του συστήματος ολοκληρώνεται και ενημερώνεται η κατάσταση του RAID EC για λειτουργία σε normal mode. Στην περίπτωση τώρα που κατά τη διάρκεια εκτέλεσης της διεργασίας recover εμφανιστεί επιπρόσθετος εσφαλμένος δίσκος, τότε η διεργασία διακόπτεται, καλώντας τη διεργασία degrade και, αφού ολοκληρωθούν τα όποια εν εξελίξει αιτήματα εγγραφής ή ανάγνωσης στους δίσκους του συστήματος και εξασφαλιστεί επιπλέον ότι δεν έχει παραμείνει κλειδωμένος με υπαιτιότητα του recover, επιστρέφει στο σύστημα (πιο συγκεκριμένα στο Controller του συστήματος) με σχετικό error. Στη συνέχεια, και εφόσον το σύστημα παραμένει βιώσιμο και αποκαταστάσιμο ξανακαλείται η recover.

Κεφάλαιο 4

Υλοποίηση

Στο κεφάλαιο αυτό περιγράφεται μια προσπάθεια υλοποίησης της αρχιτεκτονικής του αλγορίθμου του RAID EC, όπως και των επιμέρους επιλογών της υλοποίησης. Η υλοποίηση αυτή επιλέχτηκε να γίνει σε user level για λόγους ευκολίας σε συνδυασμό με το γεγονός ότι ο κύριος στόχος είναι η επαλήθευση της σωστής λειτουργίας της αρχιτεκτονικής και η διασφάλιση της αυξημένης προστασίας της αποθηκευόμενης πληροφορίας από την επίτευξη υψηλών επιδόσεων. Αν και ο παρών σχεδιασμός αναφέρεται σε ένα αυτόνομο σύστημα για αποθήκευση πλεονάζουσας πληροφορίας σε επίπεδο block, αρχικός στόχος ήταν η ενσωμάτωσή του στο vRAID, ένα κατανεμημένο σύστημα αποθήκευσης που αναπτύχθηκε από το εργαστήριο Υπολογιστικών Συστημάτων του Εθνικού Μετσόβιου Πολυτεχνείου. Για το λόγο αυτό και διατηρήθηκε η σημασιολογία του. Έτσι το παρόν κεφάλαιο χωρίζεται σε τρεις ενότητες: στην πρώτη γίνεται αναφορά στο vRAID και στη σχέση του με την παρούσα υλοποίηση, στη δεύτερη περιγράφεται ο τρόπος υλοποίησης των τμημάτων του RAID EC, που αναφέρονται στην εφαρμογή του Erasure Code στο RAID σύστημα, και στην τρίτη ενότητα περιγράφεται η υλοποίηση των βασικών διεργασιών του συστήματος.

Αναλυτικότερα, στην πρώτη ενότητα αναπτύσσεται μία σύντομη περιγραφή της αρχιτεκτονικής του vRAID και της σχέσης αυτής με την παρούσα υλοποίηση του RAID EC. Με αφετηρία την παρουσίαση αυτή γίνεται και μία αναφορά στις βασικότερες δομές του κώδικα που υλοποιεί το σύστημα RAID EC.

Στη δεύτερη ενότητα γίνεται παρουσίαση του τρόπου υλοποίησης της μεθοδολογίας πραγματοποίησης των απαιτούμενων πράξεων στο σώμα Galois, του υπολογισμού του πίνακα B της Reed-Solomon κωδικοποίησης και της εφαρμογής της μεθόδου απαλοιφής Gauss.

Στην τρίτη ενότητα περιγράφεται ο τρόπος υλοποίησης των βασικών διεργασιών του συστήματος RAID EC. Αναλυτικότερα, λαμβάνει χώρα μια εκτενή παρουσίαση του τρόπου υλοποίησης των παρακάτω διεργασιών:

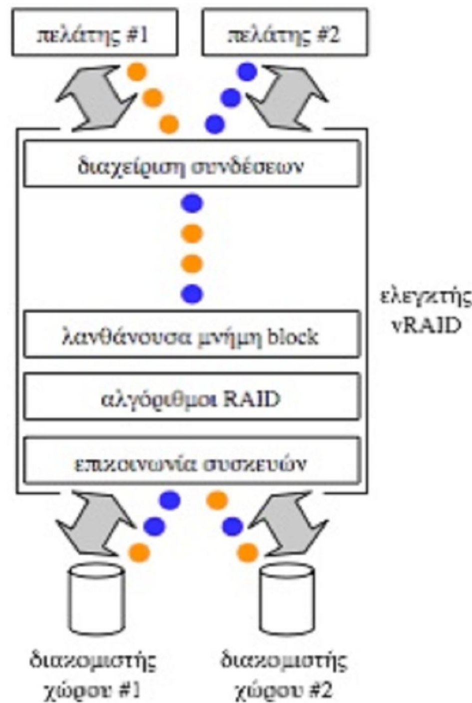
- **check**, η οποία εκτελεί τους αρχικούς ελέγχους για την ενεργοποίηση του συστήματος RAID EC,
- **init**, η οποία εκτελεί ενέργειες αρχικοποίησης του συστήματος απαραίτητες για την ενεργοποίησή του,
- **read**, η οποία πραγματοποιεί τα αιτήματα ανάγνωσης,
- **write**, η οποία ικανοποιεί τα αιτήματα εγγραφής,
- **degrade**, η οποία εκτελείται κάθε φορά που αποτυγχάνει κάποια κλήση ανάγνωσης ή εγγραφής του συστήματος για τον εντοπισμό εσφαλμένων δίσκων και την εκτέλεση των κατάλληλων ενεργειών για την εξασφάλιση της λειτουργίας του συστήματος (εφόσον το σύστημα παραμένει βιώσιμο) και, τέλος,
- **recover**, η οποία πραγματοποιεί την αποκατάσταση του συστήματος ύστερα από απώλεια δίσκου.

Στο σημείο αυτό πρέπει να σημειωθεί ότι η παρούσα υλοποίηση έχει γίνει έτσι ώστε στη θέση των δίσκων να μπορεί να βρίσκεται οποιοδήποτε αποθηκευτικό μέσο, όπως file, block device, Network Block Device (NBD) ή οποιαδήποτε μορφή εικονικού μέσου αποθήκευσης.

4.1 Το vRAID και η σχέση του με την υλοποίηση του RAID EC

Το vRAID είναι ένα κατανεμημένο σύστημα αποθήκευσης που αναπτύχθηκε από το εργαστήριο Υπολογιστικών Συστημάτων του Εθνικού Μετσόβιου Πολυτεχνείου. Αποτελείται από 3 διακριτά υποσυστήματα και η ομαλή του λειτουργία στηρίζεται στη σωστή και εύρυθμη συνεργασία τους: τον ελεγκτή, το διακομιστή δεδομένων και τον οδηγό συσκευής. Ο οδηγός συσκευής εκτελείται στο σύστημα του χρήστη και αποτελεί τη γέφυρα επικοινωνίας μεταξύ του λειτουργικού συστήματος που προσπελαύνει το vRAID και τον/τους ελεγκτή/ές. Ο ελεγκτής τοποθετείται στο ενδιάμεσο μονοπάτι των αιτήσεων (για εγγραφή ή ανάγνωση) μεταξύ των χρηστών και των φυσικών μέσων που προσφέρονται από τους διακομιστές δεδομένων. Ο ελεγκτής δέχεται τις αιτήσεις σε μορφή block από συστήματα που παράγουν ή καταναλώνουν πληροφορία και με τη βοήθεια του κατάλληλου αλγορίθμου RAID, μετασχηματίζει τις πράξεις ώστε να

αποσταλούν στους δίσκους. Από τη μία μεριά φτάνουν στον ελεγκτή αιτήσεις που αναφέρονται σε διευθύνσεις block του συνολικού ιδεατού χώρου και οι οποίες μεταδίδονται σύμφωνα με κάποιο γενικό πρωτόκολλο ανταλλαγής block δεδομένων. Από την άλλη μεριά τα μηνύματα που ταξιδεύουν μεταξύ των ελεγκτών και των διακομιστών δεδομένων αναφέρονται σε πραγματικές διευθύνσεις της φυσικής συσκευής και μπορεί να μεταφράζονται σε κάποια ειδική μορφή για τον τύπο συνδεσμολογίας του δίσκου.



Σχήμα 25: Αρχιτεκτονική του vRAID

Εσωτερικά ο ελεγκτής αποτελείται από μια σειρά επιπέδων, όπως φαίνεται και στο Σχήμα 25. Αυτή η προσέγγιση αφενός διευκολύνει την υλοποίηση και αφετέρου δίνει τη δυνατότητα να συμπληρωθούν ή να αλλαχτούν κάποιοι αλγόριθμοι στο μέλλον ανάλογα με τις απαιτήσεις του χρήστη του συστήματος. Αναλυτικότερα, τα επίπεδα αυτά είναι:

- Διαχείριση συνδέσεων: Είναι δυνατόν πολλαπλά συστήματα να πρέπει να συνδέονται με ένα κοινό σύστημα αποθήκευσης. Στο vRAID δίνεται η δυνατότητα αυτή μέσω του συγκεκριμένου επιπέδου, που διαχειρίζεται τις ενεργές συνδέσεις με τους χρήστες (με τους οδηγούς συσκευής). Το επίπεδο αυτό μπορεί να διαχειριστεί θεωρητικά απεριόριστες, παράλληλες και ταυτόχρονες συνδέσεις με οποιοδήποτε αριθμό χρηστών. Σε αυτό το επίπεδο υλοποιούνται οι απαραίτητοι μηχανισμοί για τη δημιουργία και διατήρηση συνδέσεων με το vRAID και το πρωτόκολλο μεταφοράς block δεδομένων. Επιπρόσθετα, το

επίπεδο διαχείρισης συνδέσεων επιτρέπει στα υπόλοιπα επίπεδα του vRAID να λειτουργούν ανεξάρτητα από τον αποστολέα της αίτησης. Η πληροφορία του αποστολέα μιας αίτησης κωδικοποιείται σε ένα μονικό αριθμό κατά την είσοδό της στο σύστημα. Ο αριθμός αυτός επιτρέπει στο επίπεδο διαχείρισης συνδέσεων να δρομολογήσει σωστά τα μηνύματα που επιστρέφονται από τα παρακάτω επίπεδα, πίσω στους χρήστες.

- Λανθάνουσα μνήμη block: Οι λανθάνουσες μνήμες χρησιμοποιούνται γενικά για την αύξηση των επιδόσεων συστημάτων που μπορεί να επεξεργάζονται πολλές φορές τα ίδια δεδομένα. Στο vRAID η λανθάνουσα μνήμη υλοποιείται στο επίπεδο των διευθύνσεων του ιδεατού δίσκου, οπότε πολλαπλές, συνεχόμενες εντολές που αντιστοιχούν σε ίδιες διευθύνσεις μπορούν να εξυπηρετούνται από αυτό το επίπεδο, χωρίς να απαιτείται η μεταφορά της αίτησης στο επίπεδο RAID. Η καθυστέρηση που μπορεί να αντιστοιχεί στην εκτέλεση κάθε εντολής εξαρτάται από το επίπεδο RAID, αλλά σε γενικές γραμμές κάθε εισερχόμενη αίτηση προκαλεί τη δημιουργία και την αποστολή πολύ περισσότερων αιτήσεων προς τους δίσκους. Η προσθήκη ενός επιπέδου λανθάνουσας μνήμης εντός του ελεγκτή αποθήκευσης οφείλεται και στο γεγονός ότι, για τη λειτουργία του συστήματος από πολλαπλούς χρήστες, θα πρέπει στο σημείο που εκτελούνται οι οδηγοί συσκευής να απενεργοποιούνται οι αντίστοιχες λανθάνουσες μνήμες του λειτουργικού συστήματος. Το συγκεκριμένο επίπεδο βοηθά στην αύξηση της συνολικής απόδοσης, ειδικά όταν οι παραπάνω μνήμες απενεργοποιούνται.

- Αλγόριθμοι RAID: Στο επίπεδο των αλγορίθμων RAID γίνεται η μετάφραση της αίτησης του χρήστη σε αιτήσεις προς τους διακομιστές δεδομένων. Το επίπεδο αυτό αναλαμβάνει την τήρηση της ακεραιότητας των δεδομένων και την ανάνηψη του συστήματος σε περίπτωση που κάποιος δίσκος αποσυνδεθεί ή σταματήσει να λειτουργεί.

- Επικοινωνία συσκευών: Στο τελευταίο επίπεδο γίνεται η επικοινωνία με τις φυσικές συσκευές. Στο επίπεδο αυτό υλοποιείται το πρωτόκολλο μεταφοράς block από και προς τους δίσκους (τους διακομιστές δεδομένων) και γίνεται η διαχείριση όλων των σχετικών δικτυακών συνδέσεων. Τα σφάλματα στην επικοινωνία αυτού του επιπέδου είναι εξαιρετικά σημαντικά και αντιστοιχούν σε σφάλματα στα μέσα αποθήκευσης.

Ο σχεδιασμός της παρούσας υλοποίησης του αλγορίθμου του RAID EC έγινε με γνώμονα τη μελλοντική χρησιμοποίησή του από τον ελεγκτή του vRAID. Κατά αυτό τον τρόπο η υλοποίηση των βασικών διεργασιών του RAID EC (check, init, read, write, degrade και recover), οι οποίες αποτελούν και τις βασικές διαδικασίες που απαιτεί το επίπεδο των αλγορίθμων RAID του ελεγκτή του vRAID για την επίτευξη της σωστής λειτουργίας του, έλαβε χώρα δίνοντας κυρίως έμφαση στη μετάφραση των αιτημάτων

του χρήστη σε αιτήματα προς τους διακομιστές δεδομένων, ώστε να διατηρείται η ακεραιότητα των δεδομένων και να επιτυγχάνεται η αποκατάσταση του συστήματος σε περίπτωση ύπαρξης εσφαλμένου δίσκου. Ακολουθώντας αυτή την κατεύθυνση, στον κώδικα υλοποιήθηκε ξεχωριστή μέθοδος που πραγματοποιεί την επικοινωνία με τις φυσικές συσκευές και τη διαχείριση των σχετικών αιτημάτων. Στη μελλοντική ενσωμάτωση του RAID EC στο VRAID, οι λειτουργίες αυτές θα χειρίζονται αποκλειστικά από συγκεκριμένες διεργασίες του τελευταίου. Επιπρόσθετα, ο παρών σχεδιασμός δεν περιλαμβάνει την πλήρη υλοποίηση κάποιου ελεγκτή (όπως αυτός του VRAID) που θα είναι υπεύθυνος για την κλήση και διαχείριση των βασικών διεργασιών του RAID EC, καθώς εστιάζει κυρίως στην αρχιτεκτονική του RAID EC και στις καθ'αυτού διεργασίες του. Η δομή ενός τέτοιου ελεγκτή καθορίζεται περισσότερο από το γενικότερο σύστημα στο οποίο καλείται να ενσωματωθεί η αρχιτεκτονική του RAID EC παρά από την ίδια την αρχιτεκτονική.

Αποτέλεσμα της συσχέτισης της υλοποίησης του αλγορίθμου του RAID EC με το vRAID αποτελούν και οι παρακάτω δομές, οι οποίες χρησιμοποιούνται και στην παρούσα υλοποίηση.

```
struct vraid_device {
        int                     fd;             //Device descriptor
        uint64_t                mediasize;  //The actual size of the device in sectors
        uint8_t                 status;         //Device status
        pthread_mutex_t  mutex;         //Pthread mutex in order to synchronize the call to the
                                                //changes on this device
        uint8_t                 *lock;          //Implement the Locks on the device
};

struct vraid_request {
        uint32_t   sequence;    //The index of the request to the disks
        uint8_t     command;   //The command type for the disks
        uint64_t   offset;      //The offset of the disk that the specific command is referring to
        uint32_t   ength;       //The length of the data that the specific command should be applied
};

struct vraid_info {
        uint64_t     mediasize;  //The size of the disks in sectors
        uint32_t     sectorsize; //The size of a sector (block) in bytes
        uint32_t     chunksize;  //The size of a chunk in sectors
        uint8_t       raidstatus; //The status of the RAID system.

        struct vraid_device     *device;
        uint8_t                         devicenum;         //The number of the RAID system's disks: n+m
        struct vraid_device     *spare;
        uint8_t                         sparenum;          //The number of spare disks for the RAID system

        int                             paritynum;         //The number of parity disk to be used: m
};

struct raidEC_command {
        struct vraid_device   *device;
        struct vraid_request  request;
        unsigned char   *data;
        uint8_t  success;        //success = 1, in case the command has been executed successfully
        uint8_t   precommand;
};
```

Όπως γίνεται φανερό, η δομή *vraid_device* διατηρεί βασικές πληροφορίες για τον κάθε δίσκο του συστήματος. Η χρησιμοποίηση της παραμέτρου lock στοχεύει στην υλοποίηση της διαδικασίας κλειδώματος, η οποία, όπως είχε αναφερθεί και στο κεφάλαιο 3, απαιτείται κατά τη διαδικασία εγγραφής σε δίσκους του συστήματος. Πιο συγκεκριμένα η παράμετρος lock είναι ένας πίνακα μεγέθους ίσος με το μέγεθος του δίσκου, με κάθε στοιχείο του να αντιστοιχεί σε κάθε sector του δίσκου και να παίρνει την τιμή 1 όταν είναι κλειδωμένο από κάποια διεργασία του συστήματος. Στην αντίθετη περίπτωση παίρνει την τιμή 0.

Οι δομές *vraid_request*, *raidEC_command* και *vraid_info* διατηρούν τις βασικές πληροφορίες για τον τύπο του αιτήματος προς το δίσκο του συστήματος, το ίδιο το αίτημα και το RAID σύστημα αντιστοίχως.

Στη συνέχεια, αναπτύχθηκαν και οι παρακάτω δομές για τη διατήρηση της βασικής πληροφορίας συστήματος, συγκεκριμένα για το RAID EC.

```
struct FDstr {
        int disk;       //The number of disk
        long stripe;  //0 - stripe is the stripes that are recovered. Initial value: -1
};

struct raidEC_info {
        struct vraid_device **device;
        int nRD;              //The number of RAID EC disks: n+m
        int nSD,nCsD;      //nSD: the number of store disks (n), nCsD: the number of
                              //Checkcsum disks (m)
        int nFD;            //The number of failed disks
        struct FDstr *failedDisks;  //The table of the failed disks
        int* eqFD;                    //The failed disks for the given stripe
        int eqFDnum;                //The number of failed disks for the given stripe
        int nGET;                    //The number of the faild store disks for the given stripe
        unsigned char ** GETable;  //The table of Gauss Elimination method
        int ** BMatrix;                //The B matrix of Reed-Solomon code
        pthread_mutex_t     mutex;      //Pthread mutex in order to synchronize changes
                                        //on system's status.
        int spareleft;          //The number of the unused spare disks
};
```

Η δομή *FDstr* και ειδικότερα η παράμετρος *failedDisks* του αλγορίθμου RAID EC χρησιμοποιείται για τη δήλωση των εσφαλμένων δίσκων του συστήματος. Όπως έχει αναφερθεί στο σχεδιασμό του συστήματος RAID EC και πιο συγκεκριμένα στην περιγραφή της διεργασίας recover (παράγραφος 3.2.4), η αποκατάσταση ενός δίσκου πραγματοποιείται σταδιακά ανά stripe. Για το λόγο αυτό και στη δομή FDstr, η παράμετρος stripe αναφέρεται στο βαθμό που έχει αποκατασταθεί ο δίσκος. Πιο συγκεκριμένα αναφέρεται στο stripe μέχρι το οποίο έχει αποκατασταθεί ο δίσκος (παίρνει την τιμή -1 στην περίπτωση που δεν έχει γίνει καμία διαδικασία αποκατάσταση στο δίσκο αυτό), επισημαίνοντας παράλληλα για ποια stripe του συστήματος ο συγκεκριμένος δίσκος θεωρείται εσφαλμένος και για ποια όχι.

Η παρακάτω συνάρτηση (*excCommand*) δημιουργήθηκε στην προσπάθεια διαχωρισμού του κώδικα που είναι υπεύθυνος για την πραγματοποίηση των αιτημάτων προς τους δίσκους του συστήματος από τον κύριο κώδικα που υλοποιεί τον αλγόριθμο του RAID EC. Στη συνέχεια, παρουσιάζεται το τμήμα του κώδικα της *excCommand* που υλοποιεί τα διάφορα αιτήματα. Όπως παρατηρούμε από τον παρακάτω κώδικα, στην περίπτωση αποτυχίας ανάγνωσης ή εγγραφής σε κάποιο δίσκο (δηλαδή όταν αποτύχουν οι

συναρτήσεις *pread* και *pwrite* αντίστοιχα), τότε αυτός μαρκάρεται ως εσφαλμένος και καλείται η διεργασία degrade. Επιπρόσθετα παρατηρείται ο τρόπος χρησιμοποίησης του πίνακα *lock* της δομής *vraid_device* για την επίτευξη του κλειδώματος τμήματος ενός δίσκου.

```
switch(command->request.command){
    case VRAID_COMMAND_RLOCK:
    case VRAID_COMMAND_READ:
            r=pread(command->device->fd, command->data, command->request.length*vraid.sectorsize,
                    command->request.offset*vraid.sectorsize);
            if(r<0){
                command->device->status = VRAID_DEVICE_ERROR;
                raidEC_degrade();
                command->success = VRAID_ERROR_DISKIO;
                break;
            }
            if(command->request.command == VRAID_COMMAND_READ)
                break;
    case VRAID_COMMAND_LOCK:
            pthread_mutex_lock(&command->device->mutex);
            for(i=command->request.offset;i<(command->request.offset+command->request.length);i++){
                if(command->device->lock[i]==0)
                    command->device->lock[i]=1;
                else {
                                        for(j=command->request.offset;j<i;j++)
                                                command->device->lock[i]=0;
                                        command->success = VRAID_ERROR_LOCK;
                                        break;
                                }
                        }
                        pthread_mutex_unlock(&command->device->mutex);
                        break;
```

```
    case VRAID_COMMAND_WUNLOCK:
    case VRAID_COMMAND_WRITE:
            r=pwrite(command->device->fd, command->data, command->request.length*vraid.sectorsize,
                    command->request.offset*vraid.sectorsize);
            if(r<0){
                command->device->status = VRAID_DEVICE_ERROR;
                raidEC_degrade();
                command->success = VRAID_ERROR_DISKIO;
                break;
            }
            if(command->request.command == VRAID_COMMAND_WRITE)
                break;
    case VRAID_COMMAND_UNLOCK:
            pthread_mutex_lock(&command->device->mutex);
            for(i=command->request.offset;i<(command->request.offset+command->request.length);i++){
                command->device->lock[i]=0;
            }
            pthread_mutex_unlock(&command->device->mutex);
            break;
}
```

Λόγω έλλειψης υποδομής αντίστοιχης με το VRAID στην παρούσα υλοποίηση, η διαχείριση των αιτημάτων προς τους δίσκους του συστήματος πραγματοποιείται εντός των διεργασιών read, write και recover. Κατά την εκτέλεση των παραπάνω διεργασιών, για κάθε stripe στο οποίο αναφέρονται, κατασκευάζονται αιτήματα προς τους δίσκους χρησιμοποιώντας την δομή *raidEC_command*. Ύστερα από την κατασκευή των αιτημάτων, για καθένα από αυτά εκτελείται η συνάρτηση *excCommand* ως παράλληλη διεργασία (thread).

4.2 Υλοποίηση των βασικών διεργασιών του Erasure Code

Στην ενότητα αυτή παρουσιάζεται η υλοποίηση των βασικών τμημάτων του RAID EC που απαιτούνται για την εφαρμογή του Erasure Code σε ένα RAID σύστημα. Αναλυτικότερα, θα περιγραφεί ο τρόπος με τον οποίο υλοποιούνται οι τεχνικές πραγματοποίησης των πράξεων σε σώματα Galois, ο υπολογισμός του πίνακα B της κωδικοποίησης Reed – Solomon και της μεθόδου απαλοιφής Gauss για την επίλυση γραμμικών συστημάτων. Οι παραπάνω υλοποιήσεις βασίζονται στις προδιαγραφές που περιγράφηκαν στην ενότητα 3.1.

4.2.1 Υλοποίηση των σωμάτων Galois

Για την υλοποίηση των σωμάτων Galois εφαρμόζεται η μεθοδολογία που περιγράφηκε στην παράγραφο 3.1.1 και η χρήση των πινάκων gflog και gfilog. Το σώμα Galois που θα χρησιμοποιηθεί για την παρούσα υλοποίηση του RAID EC είναι το $GF(2^8)$, δηλαδή το σώμα Galois με βαθμό w = 8. Ο λόγος επιλογής του συγκεκριμένου σώματος Galois έγινε ώστε να ταυτίζεται το σώμα με τις τιμές ενός byte, το οποίο αποτελεί την βασική μονάδα αποθήκευσης δεδομένων, αλλά και για πρακτικούς λόγους υλοποίησης του σώματος και των αντίστοιχων πράξεων εντός αυτού. Η επιλογή ενός μεγαλύτερου βαθμού του σώματος θα οδηγούσε στη δημιουργία αρκετά ογκώδων και δύσχρηστων πινάκων gflog και gfilog.

Συνεπώς, η μεταβλητή pp αντιπροσωπεύει το μονικό πολυώνυμο f(x) βαθμού w του $Z_2[x]/(f(x))$, ώστε το τελευταίο να είναι ισόμορφο με το $GF(2^w)$, και πιο συγκεκριμένα η τιμή του να ισούται με την οκταδική τιμή ενός byte μνήμης (το οποίο έχει μέγεθος 8 bit και το κάθε byte απεικονίζεται και από ένα στοιχείο του $GF(2^8)$) προς αποθήκευση. Στη συγκεκριμένη περίπτωση που w=8 το f(x) είναι το $x^8 + x^4 + x^3 + x^2 + 1$ και η τιμή του pp είναι 0435(οκταδικός αριθμός). Ο κώδικας που υλοποιεί το σχηματισμό των πινάκων gflog και gfilog παρουσιάζεται στη συνέχεια.

```
unsigned int pp=0435;
unsigned short *gflog,*gfilog;

int init_tables(){
      int i;
      unsigned short val;
      gflog=(unsigned short *)malloc(sizeof(unsigned short)*256);
      gfilog=(unsigned short *)malloc(sizeof(unsigned short)*255);
      val=1;
      gfilog[0]=val;
      gflog[1]=0;
      for(i=1;i<255;i++){
            val=val<<1;                         //Υλοποίηση του πολλαπλασιασμού με το x
            if(val&0400) val= val^pp;           //Υλοποίηση της διαίρεσης με το f(x), στη
                                          //περίπτωση που προκύπτει πολυώνυμο βαθμού
                                                //μεγαλύτερου του w
            gfilog[i]=val;
            gflog[val]=i;
      }
      return 0;
}
```

Ο κώδικας που υλοποιεί τις πράξεις της πρόσθεσης, αφαίρεσης, πολλαπλασιασμού, και διαίρεσης φαίνεται στη συνέχεια. Οι πράξεις της πρόσθεσης και της αφαίρεσης, όπως αναπτύχθηκε στο πρώτο κεφάλαιο, ταυτίζονται και υλοποιούνται με ένα XOR στους αντίστοιχους συντελεστές των πολυωνύμων (δηλαδή στα αντίστοιχα bit των block μνήμης). Η μέθοδος που τις υλοποιεί φαίνεται στη συνέχεια.

```
unsigned char gf_add(unsigned char x,unsigned char y){
	return x^y;
}
```

Η πράξη του πολλαπλασιασμού υλοποιείται με την παρακάτω συνάρτηση που την ανάγει σε απλή πρόσθεση εκθετών, εκμεταλλευόμενοι και το γεγονός ότι οι γεννήτορες είναι μία κυκλική ομάδα.

```
int gf_mult(int x,int y){
	int s;
	if(x==0 || y==0) return 0;
	s=gflog[x]+gflog[y];
	if(s>=255) s-=255;
	return gfilog[s];

}
```

Η πράξη της διαίρεσης υλοποιείται με την παρακάτω συνάρτηση που την ανάγει αντίστοιχα σε απλή αφαίρεση εκθετών.

```
int gf_div(int x,int y){
	int s;
	if(x==0) return 0;
	if(y==0) return -1;
	s=gflog[x]-gflog[y];
	if(s<0) s+=255;
	return gfilog[s];
}
```

Η πράξη της δύναμης υλοποιείται και αυτή με αντίστοιχο τρόπο, μετατρέποντας την πράξη σε ένα απλό πολλαπλασιασμό μεταξύ του εκθέτη και της δύναμης, όπως φαίνεται και στην παρακάτω συνάρτηση.

```
int gf_pow(int x,int n){
	int s;
	if(n==0) return 1;
	if(n==1) return x;
	if(x==0) return 0;
	if(x==1) return 1;
	s=n*gflog[x];
	while(s>=255) s-=255;
	return gfilog[s];
}
```

Επειδή όμως η βασική μονάδα ανάγνωσης και εγγραφής για το RAID EC και γενικότερα τα RAID συστήματα είναι το block, κατασκευάστηκαν οι παρακάτω συναρτήσεις ώστε να ανάγουν τις απαραίτητες πράξεις σε ένα block. Σε κάθε περίπτωση όμως οι πράξεις πάλι θα γίνονται στο επίπεδο του byte και οι παρακάτω συναρτήσεις έχουν βοηθητικό ρόλο για τη διευκόλυνση της πραγματοποίησης των πράξεων, αφού οι κυριότερες διεργασίες του RAID EC χρησιμοποιούνται ως μονάδα μνήμης του block.

Το block υλοποιείται ως ένας πίνακα (array) από byte και επομένως οι πράξεις μεταξύ block ανάγονται σε πράξεις μεταξύ των αντίστοιχων byte αυτών. Έτσι οι παρακάτω συναρτήσεις υλοποιούν τις πράξεις της πρόσθεσης, αφαίρεσης, πολλαπλασιασμού και διαίρεσης, τόσο μεταξύ block, όσο και μεταξύ block και byte. Στη δεύτερη περίπτωση υλοποιείται η αντίστοιχη πράξη μεταξύ του κάθε byte του block και του συγκεκριμένου byte. Επιπρόσθετα υλοποιείται η πράξη της δύναμης n ενός block, όπου n η τιμή ενός συγκεκριμένου byte, εφαρμόζοντας την πράξη της δύναμης n στο κάθε byte του block. Τέλος, στις βοηθητικές συναρτήσεις πράξεων των block που παρουσιάζονται στην συνέχεια ανήκουν και αυτές που υλοποιούν την αρχικοποίηση των bytes ενός block με μία συγκεκριμένη τιμή, την πράξη της ανάθεσης μεταξύ block και την μετατροπή του διευθυνσιολογίου από block σε byte.

```
int block_block_add(block x, block y, block r){     //Υλοποιεί την πράξη της πρόσθεσης και αφαίρεσης
		long i;                                         //μεταξύ 2 block
		for(i=0;i<vraid.sectorsize;i++){
				r[i]=gf_add(x[i],y[i]);
		}
		return 0;
}
```

```
int block_add(block x , int y, block r){               //Υλοποιεί την πράξη της πρόσθεσης και αφαίρεσης
            long i;                                                //μεταξύ block και byte
            for(i=0;i<vraid.sectorsize;i++){
                        r[i]=gf_add(x[i],y);
            }
            return 0;
}

int block_block_mult(block x, block y, block r){    //Υλοποιεί την πράξη του πολλαπλασιασμού μεταξύ
            long i;                                                //2 block
            for(i=0;i<vraid.sectorsize;i++){
                        r[i]=gf_mult(x[i],y[i]);
            }
            return 0;
}

int block_mult(block x, int y, block r){               //Υλοποιεί την πράξη του πολλαπλασιασμού μεταξύ
            long i;                                                //block και byte
            for(i=0;i<vraid.sectorsize;i++){
                        r[i]=gf_mult(x[i],y);
            }
            return 0;
}

int block_block_div(block x, block y, block r){      //Υλοποιεί την πράξη της διαίρεσης μεταξύ 2 block
            long i;
            for(i=0;i<vraid.sectorsize;i++){
                        r[i]=gf_div(x[i],y[i]);
            }
            return 0;
}

int block_div(block x, int y, block r){                 //Υλοποιεί την πράξη της διαίρεσης μεταξύ
            long i;                                                //block και byte
            for(i=0;i<vraid.sectorsize;i++){
                        r[i]=gf_div(x[i],y);
            }
            return 0;
}

int block_pow(block x, int n, block r){                //Υλοποιεί την πράξη της δύναμης n ενός block
            long i;
            for(i=0;i<vraid.sectorsize;i++){
                        r[i]=gf_pow(x[i],n);
            }
            return 0;
}

uint64_t block_offset(uint32_t offset){                //Μετατρέπει το offset από block σε byte
            return offset*vraid.sectorsize;
}
```

```
int block_init(block x, int y){                    //Αρχικοποιεί τα byte ενός block
        int i;
        for(i=0;i<vraid.sectorsize;i++)
                x[i]=y;
        return 0;
}

int block_assign(block x, block y){                    //Υλοποιεί την πράξη της ανάθεση μεταξύ 2 block
        int i;
        for(i=0;i<vraid.sectorsize;i++){
                x[i]=y[i];
        }
        return 0;
}
```

4.2.2 Υλοποίηση του πίνακα B

Ο κώδικας που παρουσιάζεται στη συνέχεια αποτελεί μια προσπάθεια υπολογισμού του πίνακα B της κωδικοποίησης Reed – Solomon, βάση των όσων είχαν περιγραφεί στην ενότητα 3.1.2 του σχεδιασμού.

Πιο συγκεκριμένα, δημιουργεί έναν πίνακα $m \times (m+n)$ (όπου m οι δίσκοι δεδομένων raidEC.nSD και n οι δίσκοι parities raidEC.nCsD, ενώ m+n=raidEC.nRD), με τα στοιχεία του να παίρνουν τις τιμές j^i, όπου j η στήλη και i η γραμμή του στοιχείου. Ο πίνακας αυτός είναι ο $(n+m) \times n$ πίνακας Vendermonde, ο οποίος και περιγράφεται στην ενότητα 3.1.2, ανεστραμμένος για πρακτικούς λόγους. Με αυτό τον τρόπο είναι πιο εύκολη η υλοποίηση της ανταλλαγής των στηλών του αρχικού πίνακα κατά τη διάρκεια του μετασχηματισμού του. Στη συνέχεια, διατρέχεται ο πίνακας ανά γραμμή j και στην περίπτωση που το στοιχείο (j,j) έχει την τιμή 0, ανταλλάσσεται η γραμμή του πίνακα με τη γραμμή i>j, για την οποία το αντίστοιχο στοιχείο (i,j) είναι διάφορο του 0. Έπειτα για τη δεδομένη γραμμή, και εφόσον το στοιχείο (j,j) είναι διάφορο του 1, διαιρούνται όλα τα στοιχεία της γραμμής με την τιμή του στοιχείου (j,j). Ακολούθως, για κάθε γραμμή i του πίνακα, εκτός της γραμμής j και των γραμμών k, όπου το στοιχείο τους (k,j) είναι ίσο με το μηδέν, πολλαπλασιάζουμε τα στοιχεία της γραμμής j με το στοιχείο (i,j) και αφαιρούμε από τη γραμμή i τη γραμμή j. Τέλος, κατασκευάζεται ο πίνακας B αντιστρέφοντας τον παραπάνω πίνακα και παίρνοντας τις n (raidEC.nCsD) ως πρώτες γραμμές.

```
int calBMatrix(){
	int i,j,k;
	uint8_t p,m;
	uint8_t ** h;
	uint8_t * h1;

	h=(uint8_t **)malloc(sizeof(uint8_t*)*raidEC.nSD);
	for(i=0;i<raidEC.nSD;i++){
		h[i]=(uint8_t *)malloc(sizeof(uint8_t)*raidEC.nRD);
		for(j=0;j<raidEC.nRD;j++)
			h[i][j]=gf_pow(j,i);
	}
	for(j=1;j<raidEC.nSD;j++){
		if(h[j][j]==0){
			for(i=j+1;i<raidEC.nSD;i++)
				if(h[i][j]!=0) break;
			h1=h[i];
			h[i]=h[j];
			h[j]=h1;
		}
		if(h[j][j]!=1){
			p=gf_div(1,h[j][j]);
			for(i=0;i<raidEC.nRD;i++)
				h[j][i]=gf_mult(p,h[j][i]);
		}
		for(i=0;i<raidEC.nSD;i++){
			if(i==j || h[i][j]==0) continue;
			m=h[i][j];
			for(k=0;k<raidEC.nRD;k++){
				p=gf_mult(m,h[j][k]);
				h[i][k]=gf_add(h[i][k],p);
			}
		}
	}
	raidEC.BMatrix=(int **) malloc(sizeof(int *)*raidEC.nCsD);
	for(i=0;i<raidEC.nCsD;i++){
		raidEC.BMatrix[i]=(int *)malloc(sizeof(int)*raidEC.nSD);
		for(j=0;j<raidEC.nSD;j++)
			raidEC.BMatrix[i][j]=h[j][i+raidEC.nSD];
	}
	for(i=0;i<raidEC.nSD;i++)
		if (h[i]) free(h[i]);
	free(h);
	return 1;
}
```

Στο σημείο αυτό αξίζει να σημειωθεί ότι τα στοιχεία του πίνακα B είναι byte, μιας και όλες οι πράξεις ανάγονται σε πράξεις μεταξύ byte, όπως είδαμε και στην προηγούμενη ενότητα.

4.2.3 Υλοποίηση της μεθόδου απαλοιφής Gauss

Στην ενότητα αυτή παρουσιάζεται ο κώδικας που υλοποιεί τη μεθοδολογία υπολογισμού της χαμένης πληροφορίας, όπως αυτή περιγράφηκε στα προηγούμενα κεφάλαια και πιο συγκεκριμένα στις ενότητες 2.2.3 και 3.1.3.

Εναρκτήριο βήμα για την ανάκτηση των χαμένων πληροφοριών, χρησιμοποιώντας την μέθοδο απαλοιφής του Gauss, είναι ο υπολογισμός του πίνακα του συστήματος των σχηματιζόμενων εξισώσεων. Ο υπολογισμός όμως του πίνακα A της μεθόδου απαλοιφής Gauss εξαρτάται από το stripe, στο οποίο αναφέρεται ο αλγόριθμος του RAID EC τη δεδομένη στιγμή. Η εξάρτηση αυτή οφείλεται στον τρόπο κατανομής της πληροφορίας και των parities στους δίσκους του συστήματος. Έτσι, ανάλογα με το αναφερόμενο stripe, διαφορετικά τμήματα της πληροφορίας ή διαφορετικά parities θεωρούνται εσφαλμένα/χαμένα. Επομένως, της κατασκευής του πίνακα A πρέπει να προηγείται ο εντοπισμός, για το συγκεκριμένο κάθε φορά stripe, των τμημάτων της πληροφορίας και των parities που θεωρούνται χαμένα. Προς υλοποίηση αυτού κατασκευάζεται κάθε φόρα ο πίνακας eqFD που απεικονίζει την κατάσταση του συγκεκριμένου stripe. Αναλυτικότερα, διατρέχεται ο πίνακας failedDisks, ο οποίος περιέχει την πληροφορία σχετικά με το ποιος δίσκος και για ποια stripe θεωρείται εσφαλμένος, εντοπίζοντας για το συγκεκριμένο stripe ποιοι είναι οι εσφαλμένοι δίσκοι. Βάσει της πληροφορίας αυτής κατασκευάζεται ο πίνακας eqFD, ώστε να περιέχει ταξινομημένα τα εσφαλμένα chunks δεδομένων και parities για το συγκεκριμένο stripe, χρησιμοποιώντας τις πρώτες θέσεις του πίνακα για τα chunk δεδομένων, χαρακτηρίζοντας το πρώτο chunk δεδομένων του stripe ως το 0 και τις υπόλοιπες θέσεις για τα chunk των parities, χαρακτηρίζοντας το πρώτο parity chunk ως το 0. Ο λόγος για τον οποίο χρησιμοποιείται η δομή αυτή για τον πίνακα eqFD είναι το γεγονός ότι όλοι οι υπολογισμοί στο RAID EC γίνονται ανά stripe και ανάγονται στο πρώτο stripe του συστήματος, όπου οι πρώτοι m δίσκοι είναι οι δίσκοι δεδομένων και οι υπόλοιποι n τα parities. Στο σημείο αυτό κρίνεται απαραίτητο να αναφερθεί ότι η ενημέρωση του πίνακα failedDisks γίνεται από τις διεργασίες degrade και recover. Επιπρόσθετα, κατά τη διαδικασία κατασκευής του πίνακα eqFD υπολογίζεται και η τιμή της παραμέτρου nGET, η οποία είναι ίση με τον αριθμών των εσφαλμένων chunks δεδομένων για το συγκεκριμένο stripe. Στη συνέχεια παρουσιάζεται ο κώδικας που υλοποιεί τη διεργασία κατασκευής του πίνακα eqFD.

```
int create_eqFD(uint32_t n_stripe){
	int i,j,k;
	uint32_t data_disk,parity_disk;
	raidEC.nGET=0;
	raidEC.eqFDnum=0;
	if(raidEC.nFD==0)
		return 0;
	for(i=0;i<raidEC.nFD;i++)
		if(n_stripe>raidEC.failedDisks[i].stripe)
			raidEC.eqFDnum++;
	if(raidEC.eqFD==0)
		return 0;
	data_disk=(n_stripe*raidEC.nSD)%raidEC.nRD;
	parity_disk=((n_stripe+1)*raidEC.nSD)%raidEC.nRD;
	if(data_disk+raidEC.nSD<=raidEC.nRD)
		for(i=0,j=0,k=raidEC.eqFDnum-1;i<raidEC.nFD;i++) {
		if(n_stripe>raidEC.failedDisks[i].stripe &&
			  raidEC.failedDisks[i].disk>=data_disk &&
			  raidEC.failedDisks[i].disk<data_disk+raidEC.nSD){
			raidEC.nGET++;
				raidEC.eqFD[j]=raidEC.failedDisks[i].disk-data_disk;
				j++;
			}
			else if(n_stripe>raidEC.failedDisks[i].stripe){
				raidEC.eqFD[k]=raidEC.failedDisks[i].disk-parity_disk;
				if(raidEC.eqFD[k]<0)
					raidEC.eqFD[k]+=raidEC.nRD;
				k--;
			}
		}
	else
		for(i=0,j=0,k=raidEC.eqFDnum-1;i<raidEC.nFD;i++)
			if(n_stripe>raidEC.failedDisks[i].stripe &&
			  (raidEC.failedDisks[i].disk<(data_disk+raidEC.nSD)%raidEC.nRD
			   || raidEC.failedDisks[i].disk>=data_disk)){
				raidEC.nGET++;
				raidEC.eqFD[j]=raidEC.failedDisks[i].disk-data_disk;
				if(raidEC.eqFD[j]<0)
					raidEC.eqFD[j]+=raidEC.nRD;
				j++;
			}
			else if(n_stripe>raidEC.failedDisks[i].stripe){
				raidEC.eqFD[k]=raidEC.failedDisks[i].disk-parity_disk;
				k--;
			}
	sort(raidEC.eqFD,raidEC.nGET);
	sort(raidEC.eqFD+raidEC.nGET,raidEC.eqFDnum-raidEC.nGET);
	return 0;
}
```

Ακολούθως, αρχικοποιείται ο πίνακας A της μεθόδου απαλοιφής Gauss χρησιμοποιώντας τον πίνακα eqFD, για το δεδομένο stripe του RAID EC συστήματος,

και τον πίνακα B της κωδικοποίησης Reed-Solomon, η κατασκευή του οποίου παρουσιάστηκε στην προηγούμενη ενότητα (4.2.2). Πιο συγκεκριμένα κατασκευάζεται ο πίνακας A σε μορφή επαυξημένου πίνακα. Το μέγεθος, επομένως, του πίνακα ορίζεται ως $nGET \times (nGET+1)$, όπου nGET ο αριθμός των εσφαλμένων chunks δεδομένων για το συγκεκριμένο stripe. Η κατασκευή του πίνακα A επιτυγχάνεται διατρέχοντας τον πίνακα eqFD, με σκοπό τον εντοπισμό των nGET πρώτων parities που είναι διαθέσιμα. Στη συνέχεια, βάση των parities που έχουν επιλεγεί, ο πίνακας συμπληρώνεται ανά γραμμή με τα στοιχεία του πίνακα B b_{ij}, με i να αντιστοιχεί στο αντίστοιχο parity και j να αναφέρεται στους εσφαλμένους δίσκους δεδομένων. Επιπρόσθετα, στη στήλη (nGET+1) του πίνακα A αναγράφεται το εκάστοτε parity. Με αυτό τον τρόπο σχηματίζεται ο επαυξημένος πίνακας A της μεθόδου απαλοιφής Guass, ο οποίος περιέχει τους συντελεστές του συστήματος, αλλά χωρίς το σταθερό μέρος των εξισώσεων το οποίο και θα προστεθεί στη συνέχεια. Παρακάτω εμφανίζεται ο κώδικας που υλοποιεί την αρχικοποίηση του πίνακα A.

```
int createGETable(){
	int i,j,n,k;
	raidEC.GETable=(unsigned char **) malloc(sizeof(char *)*raidEC.nGET);
	for(i=0,n=0;i<raidEC.nGET;i++,n++){
		raidEC.GETable[i]=(unsigned char *)malloc(sizeof(char)*(raidEC.nGET+1));
		for(k=raidEC.nGET;k<raidEC.eqFDnum;k++){
			if(n<raidEC.eqFD[k])
				break;
			if(n==raidEC.eqFD[k])
				n++;
		}
		for(j=0;j<raidEC.nGET;j++)
		   raidEC.GETable[i][j]=raidEC.BMatrix[n][raidEC.eqFD[j]];
		raidEC.GETable[i][j]=n;
	}
	return 0;
}
```

Στη συνέχεια παρουσιάζεται ο κώδικας που πραγματοποιεί τη μέθοδο απαλοιφής του Gauss στα πλαίσια του αλγορίθμου του RAID EC. Ακολουθώντας όσα αναφέρθηκαν στην ενότητα 2.2.3, η εφαρμογή της μεθόδου στην παρούσα υλοποίηση λαμβάνει χώρα ακολουθώντας τα 3 παρακάτω βήματα.

Πρώτο βήμα είναι η αναδιάταξη των στηλών του πίνακα A, ώστε στις αριστερότερες στήλες να βρίσκονται οι άγνωστοι που ενδιαφέρουν την εκάστοτε περίπτωση και να μη χρειάζεται να ολοκληρωθεί η διαδικασία της πίσω-αντικατάστασης για την ανάκτηση των δεδομένων που ενδιαφέρουν. Το σχετικό τμήμα του κώδικα που υλοποιεί τη διαδικασία αυτή φαίνεται στη συνέχεια. Για την επισήμανση των δίσκων δεδομένων, των

οποίων τα δεδομένα ζητούνται να ανακτηθούν, χρησιμοποιούνται οι μεταβλητές b, e και in. Οι μεταβλητές b και e δηλώνουν ένα σύνολο δίσκων i, ώστε να ισχύει $b \le i \le e$ και η μεταβλητή in υποδηλώνει κατά πόσο οι ενδιαφερόμενοι δίσκοι βρίσκονται εντός του προαναφερόμενου συνόλου δίσκων.

```
for(k=0,j=m=raidEC.nGET-1;j>=0;j--)
    if((in && (raidEC.eqFD[j]<b || raidEC.eqFD[j]>e))
        || (!in && raidEC.eqFD[j]>=b && raidEC.eqFD[j]<=e)){
        for(i=0;i<raidEC.nGET;i++)
            mat[i][k]=raidEC.GETable[i][j];          // Πίνακας nGET × nGET
        k++;
    }
    else{
        for(i=0;i<raidEC.nGET;i++)
            mat[i][m]=raidEC.GETable[i][j];
        m--;
    }
ret=m+1;
```

Δεύτερο βήμα είναι η διαδικασία της τριγωνοποίησης, η οποία αναλύθηκε στην ενότητα 2.2.3, και το τμήμα του κώδικα το οποίο την υλοποιεί εμφανίζεται στη συνέχεια.

```
for(i=0;i<raidEC.nGET-1;i++){
    d=mat[i][i];
    for(j=i+1;j<raidEC.nGET;j++){
        m=mat[j][i];
        m=gf_div(m,d);
        for(k=i+1;k<raidEC.nGET;k++){
            h=gf_mult(m,mat[i][k]);
            mat[j][k]=gf_add(h,mat[j][k]);
        }
        for(l=0;l<psize;l++){  //Το size του chunk των δίσκων δεδομένων που ενδιαφέρει
            block_mult(sm[i][l],m,hb);
            block_block_add(sm[j][l],hb,sm[j][l]);
        }
    }
}
```

Το τρίτο βήμα είναι η διαδικασία της πίσω-αντικατάστασης, η οποία αναλύθηκε στην ενότητα 2.2.3, και το τμήμα του κώδικα που την υλοποιεί φαίνεται στη συνέχεια.

```
for(i=raidEC.nGET-1;i>=ret;i--){
	for(l=0;l<psize;l++){
		block_div(sm[i][l],mat[i][i],sm[i][l]);
		for(j=i-1;j>=ret;j--){
			block_mult(sm[i][l],mat[j][i],hb);
			block_block_add(sm[j][l],hb,sm[j][l]);
		}
	}
}
```

4.3 Υλοποίηση του RAID EC

Στην ενότητα αυτή θα εξεταστεί ο τρόπος υλοποίησης των βασικών διεργασιών του συστήματος RAID EC, ακολουθώντας τα πρότυπα της αρχιτεκτονικής που περιγράφηκε στο κεφάλαιο 3 και πιο συγκεκριμένα στην ενότητα 3.2. Επιπρόσθετα θα αναπτυχθεί, όπου αυτό απαιτείται, ο λόγος για τον οποίο επιλέχθηκε η συγκεκριμένη μεθοδολογία υλοποίησης.

Αρχικά, θα παρουσιαστούν οι διεργασίες **check** και **init**, οι οποίες είναι υπεύθυνες για τη σωστή ενεργοποίηση του RAID EC.

Στη συνέχεια, περιγράφονται οι βασικές μέθοδοι του συστήματος, **read** και **write**, οι οποίες διαχειρίζονται όλα τα αιτήματα ανάγνωσης και εγγραφής στη συστοιχία των δίσκων.

Έπειτα, παρατίθεται η υλοποίηση των διεργασιών, **degrade** και **recover**, οι οποίες είναι υπεύθυνες για τη συνέχεια της σωστής λειτουργίας του συστήματος RAID EC ύστερα από σφάλμα σε κάποιο δίσκο του συστήματος.

Στο σημείο, όμως, αυτό πρέπει να τονιστεί ότι η ύπαρξη κάποιου **controller** στο σύστημα RAID EC, όπως διατυπώθηκε και στην ενότητα 3.2, και πιο συγκεκριμένα στο Σχήμα 20, είναι απαραίτητη για το συντονισμό των παραπάνω διεργασιών. Στην περίπτωση που το RAID EC θα ενσωματωνόταν στο vRAID, το ρόλο του **controller** αυτού θα τον είχε ο ελεγκτής του vRAID, ο οποίος παρουσιάστηκε στην ενότητα 4.1. Στην παρούσα, όμως, υλοποίηση του RAID EC, ο ρόλος αυτός για πρακτικούς λόγους ανατέθηκε στη main μέθοδο, αφού η μορφή και η δομή του σχετίζεται περισσότερο με το σύστημα το οποίο θα χρησιμοποιήσει το RAID EC και λιγότερο με το ίδιο το RAID EC. Τα χαρακτηριστικά που πρέπει να ικανοποιεί ο **controller** αυτός για να είναι δυνατή η λειτουργία του συστήματος RAID EC, τα οποία αποτελούν επιπρόσθετα και αναγκαίες

προϋποθέσεις για τη σωστή λειτουργία των βασικών διεργασιών του συστήματος, παρουσιάζονται στη συνέχεια:

- η δυνατότητα αποδοχής πολλαπλών αιτημάτων ανάγνωσης και εγγραφής και η εκτέλεση των αντίστοιχων διεργασιών read και write σε παράλληλα pthreads, προς ικανοποίηση των αιτημάτων.

- η εξασφάλιση της εκτέλεσης μιας μόνο διεργασίας recover κάθε φόρα. Αυτό εξασφαλίζεται με τον εξής τρόπο:

  1. Όταν εμφανιστεί σφάλμα σε κάποιο δίσκο, αυτό γίνεται αντιληπτό από τον controller, καθώς μια ή περισσότερες διεργασίες read ή write (ή και η recover) έχουν επιστρέψει σαν εσφαλμένες
  2. Στην περίπτωση αυτή ο controller περιμένει να επιστρέψουν όλες οι διεργασίες που βρίσκονται εν εξελίξει, ενώ παράλληλα έχει παγώσει την έναρξη νέων διεργασιών τοποθετώντας τα νέα αιτήματα ανάγνωσης και εγγραφής σε κάποια στοίβα. Στη στοίβα τοποθετούνται και τα αιτήματα που απέτυχαν λόγω του εσφαλμένου δίσκου
  3. Ελέγχει το status του RAID EC, το οποίο έχει ενημερώσει κατάλληλα τη διεργασία degrade, η οποία εκτελείται από την κάθε διεργασία που εντόπισε τον εσφαλμένο δίσκο και αποφασίζει κατά πόσο είναι βιώσιμο το σύστημα ή όχι. Στην περίπτωση που δεν είναι βιώσιμο, το σύστημα τερματίζει ως εσφαλμένο.
  4. Στην περίπτωση που το σύστημα είναι βιώσιμο, ο controller καλεί σε δικό του pthread τη διεργασία recover και παράλληλα ξεκινάει την ικανοποίηση των αιτημάτων της στοίβας του βήματος 2, καθώς και οποιουδήποτε νέου αιτήματος έρχεται. Στην περίπτωση που εμφανιστεί πάλι κάποιο νέο σφάλμα δίσκου, ο controller ακολουθεί πάλι την ίδια διαδικασία από το βήμα 1 και εκτελεί ένα νέο recover που εκτελεί την ανάκτηση του συστήματος από την αρχή.

Η ανάπτυξη ενός πλήρους controller σαν αυτού του vRAID, όπως ήδη αναφέραμε, βρίσκεται εκτός των στόχων της παρούσας εργασίας. Για το λόγο αυτό και δεν κατασκευάστηκε σχετικός κώδικας για τη διεργασία αυτή. Η μόνη σχετική υλοποίηση που έλαβε χώρα είναι στα πλαίσια του ελέγχου λειτουργίας των διεργασιών του RAID EC και των σχετικών μετρήσεων του συστήματος, η οποία και θα παρουσιαστεί στο επόμενο κεφάλαιο.

4.3.1 Αρχικοποίηση του συστήματος RAID EC

Όπως έχει ήδη περιγραφεί, για την αρχικοποίηση του συστήματος RAID EC απαιτείται η εκτέλεση 2 διεργασιών του συστήματος: της **check** και της **init**. Η **check** είναι υπεύθυνη για τον έλεγχο του κατά πόσο η δεδομένη συστοιχία δίσκων είναι κατάλληλη να υποστηρίξει το RAID EC, ενώ η **init** είναι υπεύθυνη για την κατάλληλη αρχικοποίηση του συστήματος.

Η διεργασία **check**, σύμφωνα με τις προδιαγραφές της ενότητας 3.2, ελέγχει τα εξής: τον αριθμός των διαθέσιμων δίσκων για το σύστημα, όπου στην περίπτωση που είναι ένας ή κανένας το RAID EC δεν ενεργοποιείται, και το μέγεθος του chunk (chunksize), το οποίο πρέπει να είναι δύναμη του 2 και μεγαλύτερο του μηδενός ώστε να μπορεί να φιλοξενήσει δεδομένα, σε διαφορετική περίπτωση το σύστημα αποτυγχάνει. Ο κώδικας που υλοποιεί το παραπάνω φαίνεται στη συνέχεια.

```
int
raidEC_check()
{
	if (vraid.devicenum <= 1)
		return XERROR_NOT_ENOUGH_DEVICES;
	if (vraid.chunksize == 0 || ((vraid.chunksize) & (vraid.chunksize - 1)) != 0)
		return XERROR_INCORRECT_CHUNKSIZE;
	return XERROR_NO_ERROR;
}
```

Η διεργασία init, όπως περιγράφηκε και στην ενότητα 3.2, είναι υπεύθυνη για τις παρακάτω ενέργειες. Την κανονικοποίηση του μεγέθους των δίσκων της συστοιχίας, ώστε αρχικά να έχουν όλοι το ίδιο μέγεθος και αυτό να είναι πολλαπλάσιο του μεγέθους ενός sector, όπου ως sector θεωρείται η μικρότερη μονάδα μνήμης που μπορεί να διαβαστεί ή να εγγραφεί σε ένα δίσκο. Καλεί τις διεργασίες **init_tables** και **calBMatrix**, που περιγράφηκαν στην ενότητα 4.2, και αρχικοποιούν τους βασικούς πινάκες gflog και gfilog ώστε να μπορούν να γίνονται οι πράξεις στο σώμα Galois, και τον πίνακα B της κωδικοποίησης Reed-Solomon αντίστοιχα. Τέλος, αρχικοποιεί τις βασικές δομές του συστήματος. Στο σημείο αυτό καθορίζονται και βασικά χαρακτηριστικά του συστήματος, όπως ο αριθμός των δίσκων δεδομένων και των δίσκων parities, τα οποία στη συνέχεια δεν είναι δυνατόν να μεταβληθούν. Στη συνέχεια παρουσιάζεται το τμήμα του κώδικα που υλοποιεί τη διεργασία αυτή.

```
int raidEC_init()
{
	int i, size_alert;
	int error;
	uint64_t minsize,j;

	// Normalize sizes to vraid.sectorsize
	size_alert = 0;
	minsize = vraid.device[0].mediasize;
	for (i = 1; i < vraid.devicenum; i++) {
		if (vraid.device[i].mediasize != minsize) size_alert = 1;
		if (vraid.device[i].mediasize < minsize) minsize = vraid.device[i].mediasize;
	}

	if (size_alert) {
		for (i = 0; i < vraid.devicenum; i++) {
			vraid.device[i].mediasize = minsize;
		}
	}

	vraid.mediasize = minsize;

	if(vraid.mediasize%vraid.chunksize > 0)
		vraid.mediasize-=vraid.mediasize%vraid.chunksize;

	raidEC.nRD = vraid.devicenum;
	raidEC.spareleft=vraid.sparenum;

	raidEC.device = (struct vraid_device **) malloc(sizeof(struct vraid_device *) * raidEC.nRD);
	if (raidEC.device == NULL) {
		error = VRAID_ERROR_MEMORY;
	}


	/* Initialize the internal RAID Erasure Code variables. */
	error = 0;

	for (i = 0; i < raidEC.nRD; i++) {
		raidEC.device[i] = &vraid.device[i];
		raidEC.device[i]->lock = (uint8_t *) malloc(sizeof(uint8_t)*vraid.mediasize);
		for(j=0;j<vraid.mediasize;j++)
			raidEC.device[i]->lock[j]=0;
	}

	for (i = 0; i < vraid.sparenum; i++) {
		vraid.spare[i].lock = (uint8_t *) malloc(sizeof(uint8_t)*vraid.mediasize);
		for(j=0;j<vraid.mediasize;j++)
			vraid.spare[i].lock[j]=0;
	}
```

```
        for(i=0;i<raidEC.nRD;i++){
                error = pthread_mutex_init(&raidEC.device[i]->mutex, NULL);
        }

        raidEC.nCsD=vraid.paritynum;
        raidEC.nSD=raidEC.nRD-raidEC.nCsD;
        raidEC.nFD=0;
        raidEC.eqFDnum=0;
        raidEC.failedDisks=(struct FDstr *) malloc(sizeof(struct FDstr)*raidEC.nCsD);
        raidEC.eqFD=(int *) malloc(sizeof(int)*raidEC.nCsD);
        raidEC.GETable=NULL;

        init_tables();
        calBMatrix();

        vraid.raidstatus = VRAID_STATUS_NORMAL;
        error = pthread_mutex_init(&raidEC.mutex, NULL);
        if(error ==0 )
          return XERROR_NO_ERROR;
        else
          return XERROR_INT_ERROR;
}
```

4.3.2 Υλοποίηση της διεργασίας read

Η διεργασία read είναι μία από τις βασικότερες διεργασίες του RAID EC και είναι υπεύθυνη για την ικανοποίηση των αιτημάτων για ανάγνωση από τη συστοιχία των δίσκων. Ακολουθώντας τον αλγόριθμο που περιγράφηκε στην ενότητα 3.2.1, η υλοποίηση της διεργασίας γίνεται με την παρακάτω μεθοδολογία.

Υπολογίζεται αρχικά ο αριθμός των chunk των δίσκων δεδομένων που πρέπει να αναγνωσθούν. Στη συνέχεια ελέγχεται το κατά πόσο η ζητούμενη πληροφορία βρίσκεται σε εσφαλμένο δίσκο. Για τον έλεγχο αυτό έχει κατασκευαστεί η μέθοδος **wdok**, η οποία αρχικά εντοπίζει την ακριβή θέση της ζητούμενης πληροφορίας με τη βοήθεια της μεθόδου **raidEC_locate_data** και στη συνέχεια, διατρέχοντας τη ζητούμενη περιοχή της συστοιχίας των δίσκων ανά stripe, ελέγχει για κάποιον εσφαλμένο δίσκο δεδομένων με τη βοήθεια της μεθόδου **sFD**. Η τελευταία δημιουργεί έναν πίνακα με τους εσφαλμένους δίσκους για το συγκεκριμένο stripe για το οποίο έγινε η κλήση της μεθόδου και έπειτα, ελέγχοντας τον πίνακα σε συνδυασμό με τα ενδιαφερόμενα chunk του συγκριμένου stripe, αποφαίνεται για το κατά πόσο αναφέρεται σε εσφαλμένο δίσκο δεδομένων. Ο κώδικας που υλοποιεί τις παραπάνω μεθόδους διαφαίνεται στη συνέχεια.

```
uint32_t raidEC_locate_data(uint64_t offset, uint32_t *parity_disk, uint32_t *data_disk, uint64_t *data_off)
{
	uint32_t n_chunk,n_stripe;
	n_chunk=offset/vraid.chunksize;
	n_stripe=n_chunk/raidEC.nSD;
	*parity_disk=((n_stripe+1)*raidEC.nSD)%raidEC.nRD;
	*data_disk=(n_stripe*raidEC.nSD+n_chunk%raidEC.nSD)%raidEC.nRD;
	*data_off=n_stripe*vraid.chunksize+offset%vraid.chunksize;
	return n_stripe;
}

/*search Failed Disks*/
int sFD(uint32_t b,uint32_t e,uint32_t n_stripe){
	int a,t,m,i;
	int * hfd;


	hfd=(int *) malloc(sizeof(int)*raidEC.nFD);
	a=0;
	for(i=0,t=0;i<raidEC.nFD;i++){
		if(n_stripe>raidEC.failedDisks[i].stripe){
			hfd[t]=raidEC.failedDisks[i].disk;
			t++;
		}
	}
	while(1){
		m=(int) (a+t)/2;
		if(hfd[m]>=b && hfd[m]<=e){
			free(hfd);
			return 0;
		}
		if(hfd[m]>b)
			t=m-1;
		else
			a=m+1;
		if(a>t) break;
	}
	free(hfd);
	return 1;
}


/*Επιστρέφει 1 αν οι δίσκοι δεδομένων b,...,e λειτουργούν
  και 0 αν έστω και ένας από αυτούς έχει σφάλμα*/
int wdok(uint64_t offset,uint32_t nchunk){
	uint32_t i,t,pd,dd1,dd2,stripenum,chunknum;
	uint64_t dof;

	if(raidEC.nFD == 0)
	   return 1;
	stripenum=raidEC_locate_data(offset,&pd,&dd1,&dof);
	chunknum=offset/vraid.chunksize;
```

```
        while(nchunk>0){
                nchunk--;
                for(dd2=dd1;nchunk>0;dd2++,chunknum++,nchunk--){
                        if(chunknum==(stripenum+1)*raidEC.nSD-1)
                                break;
                }
                if(dd2<raidEC.nRD){
                        if(sFD(dd1,dd2,stripenum) == 0)
                                return 0;
                }
                else{
                        if(sFD(dd1,raidEC.nRD-1,stripenum)==0)
                                return 0;
                        if(sFD(0,dd2%raidEC.nRD,stripenum)==0)
                                return 0;
                }
                stripenum++;
                dd1=(stripenum*raidEC.nSD)%raidEC.nRD;
        }

        return 1;
}
```

Η διεργασία read, στην περίπτωση που η ζητούμενη πληροφορία δεν εμπεριέχεται σε εσφαλμένο τμήμα κάποιου δίσκου δεδομένων, ζητάει απευθείας τα απαιτούμενα chuck από του δίσκους δεδομένων και σχηματίζει τη ζητούμενη πληροφορία, την οποία και επιστρέφει. Το τμήμα του κώδικα της διεργασία που υλοποιεί το παραπάνω παρουσιάζεται στη συνέχεια.

```
if(wdok(offset,nchunk)){
        rmode=0;
        command = (struct raidEC_command *)malloc(sizeof(struct raidEC_command) * nchunk);
        if (command == NULL) {
                return (VRAID_ERROR_MEMORY);
        }
        for (i = 0; i < nchunk; i++) {
                if (i == 0)
                        done = 0;
                else
                        done = (i * vraid.chunksize) - (offset % vraid.chunksize);
                todo = count - done;
                if (todo + ((offset + done) % vraid.chunksize) > vraid.chunksize)
        todo = vraid.chunksize - ((offset + done) % vraid.chunksize);
                raidEC_locate_data(offset + done, &parity_disk, &data_disk,  &data_off);
                command[commandsIndex].device = raidEC.device[data_disk];
                command[commandsIndex].request.sequence = commandsIndex;
                command[commandsIndex].request.command = VRAID_COMMAND_READ;
                command[commandsIndex].request.offset = data_off;
                command[commandsIndex].request.length = todo;
                command[commandsIndex].data = buf + (done * vraid.sectorsize);
                command[commandsIndex].success = VRAID_COMMAND_PENDING;
                commandsIndex++;
        }
        commands_tosend=nchunk;
}
```

Στην περίπτωση που μέρος της ζητούμενης πληροφορίας βρίσκεται σε εσφαλμένο τμήμα δίσκου δεδομένων, ο τρόπος ανάκτησης της πληροφορίας που ακολουθεί τη διεργασία διαφοροποιείται ανάλογα το μέγεθος της πληροφορίας και τη θέση αυτής στη συστοιχία των δίσκων. Οι διαχωρισμοί αυτοί γίνονται τόσο για πρακτικούς λόγους, όσο και για την επίτευξη καλύτερης απόδοσης του συστήματος. Κατά αυτόν τον τρόπο διακρίνονται οι παρακάτω περιπτώσεις:

- η ζητούμενη πληροφορία να έχει μέγεθος έως ένα chunk.
- η ζητούμενη πληροφορία να περιέχεται μόνο σε ένα stripe της συστοιχίας δίσκων του RAID EC συστήματος
- η ζητούμενη πληροφορία να εμπεριέχεται σε περισσότερα από ένα stripe του συστήματος

Στην πρώτη περίπτωση κατά την οποία η ζητούμενη πληροφορία έχει μέγεθος το πολύ ένα chunk, είναι βέβαιο ότι ο συγκεκριμένος δίσκος ή ακριβέστερα το συγκεκριμένο τμήμα του δίσκου είναι εσφαλμένο, διότι στην αντίθετη περίπτωση, η ζητούμενη πληροφορία δεν θα περιεχόταν σε εσφαλμένο τμήμα κάποιου δίσκου δεδομένων και η

διεργασία θα εκτελούσε τον κώδικα που περιγράφηκε προηγουμένως. Επομένως, σύμφωνα και με τον σχεδιασμό που παρουσιάζεται στην ενότητα 3.2.1, διαβάζεται η διαθέσιμη πληροφορία του συγκεκριμένου stripe τόσο από του δίσκους δεδομένων, όσο και από τους δίσκους parities, και εφαρμόζοντας τις μεθόδους της κωδικοποίησης Reed-Solomon και της απαλοιφής του Gauss, ανακτάται η ζητούμενη πληροφορία. Στο σημείο αυτό πρέπει να υπενθυμίσουμε ότι η εφαρμογή της κωδικοποίησης του Reed-Solomon στο RAID EC γίνεται ανά block (sector) και για το λόγο αυτό και τα δεδομένα που διαβάζονται από τους δίσκους θα έχουν το ίδιο μέγεθος με τη ζητούμενη πληροφορία. Στην συνέχεια παρουσιάζεται το συγκριμένο τμήμα του κώδικα που υλοποιεί τα παραπάνω.

```
else if(nchunk==1){
        rmode=1;
        command=(struct raidEC_command *)malloc(sizeof(struct raidEC_command) * (raidEC.nSD));
        commands_tosend=raidEC.nSD;
        hbuf=(unsigned char *)malloc(sizeof(unsigned char)*raidEC.nSD*count*vraid.sectorsize);
        todo=count;
        raidEC_locate_data(offset, &parity_disk, &data_disk, &data_off);
        fch=offset/vraid.chunksize;
        stripenum=fch/raidEC.nSD;
        disk_index=(stripenum*raidEC.nSD)%raidEC.nRD;
        done=0;
        create_eqFD(stripenum);
        for(j=0,i=0;i<raidEC.nSD;i++){   //read data
                if(j>=raidEC.nGET || i<raidEC.eqFD[j]){
                        command[commandsIndex].device = raidEC.device[disk_index];
                        command[commandsIndex].request.sequence = commandsIndex;
                        command[commandsIndex].request.command = VRAID_COMMAND_READ;
                        command[commandsIndex].request.offset = data_off;
                        command[commandsIndex].request.length = todo;
                        command[commandsIndex].data = hbuf + (done * vraid.sectorsize);
                        command[commandsIndex].success = VRAID_COMMAND_PENDING;
                        done+=todo;
                        commandsIndex++;
                }
                else
                        j++;
                disk_index++;
                if(disk_index>=raidEC.nRD)
                        disk_index=disk_index%raidEC.nRD;
        }
        createGETable();
```

```
        for(j=0;j<raidEC.nGET;j++,i++){     //read parities
                disk_index=parity_disk+raidEC.GETable[j][raidEC.nGET];
                if(disk_index>=raidEC.nRD)
                        disk_index=disk_index%raidEC.nRD;
                command[commandsIndex].device = raidEC.device[disk_index];
                command[commandsIndex].request.sequence = commandsIndex;
                command[commandsIndex].request.command = VRAID_COMMAND_READ;
                command[commandsIndex].request.offset = data_off;
                command[commandsIndex].request.length = todo;
                command[commandsIndex].data = hbuf + (done * vraid.sectorsize);
                command[commandsIndex].success = VRAID_COMMAND_PENDING;
                done+=todo;
                commandsIndex++;
        }
}
⋮
switch(rmode){
        ⋮
        case 1: createGETable();
              sm=(block **) malloc(sizeof(block *)*raidEC.nGET);
              for(i=0;i<raidEC.nGET;i++){
                  sm[i]=(block *) malloc(sizeof(block)*count);
                  for(j=0;j<count;j++)
                        sm[i][j]=(block) malloc(sizeof(unsigned char)*vraid.sectorsize);
              }
               calParity(hbuf,data_off%vraid.chunksize,count,fch%raidEC.nSD,1,sm);
               GausElim(sm,fch%raidEC.nSD,fch%raidEC.nSD,count,1);
               for(bufindex=0;bufindex<count;bufindex++)
                     block_assign(buf+block_offset(bufindex),sm[raidEC.nGET-1][bufindex]);
               break;
        ⋮
}
```

Στην περίπτωση που η ζητούμενη πληροφορία έχει μέγεθος μεγαλύτερο από ένα chunk, αλλά περιέχεται σε ένα μόνο stripe, θα πρέπει και εδώ να διαβαστεί η πληροφορία του συγκεκριμένου stripe τόσο των δίσκων δεδομένων, όσο και των parity. Αυτό συμβαίνει διότι κάποιος από τους ενδιαφερόμενους δίσκους είναι εσφαλμένος. Στην προσπάθεια τώρα να διαβαστεί η λιγότερη δυνατή πληροφορία από τους μη ενδιαφερόμενους δίσκους, ώστε να γίνει όσο το δυνατόν αποδοτικότερο το RAID EC σύστημα, πραγματοποιείται ένας έλεγχος για το ποιά chunks βρίσκονται σε εσφαλμένους δίσκους. Στην περίπτωση που μόνο το πρώτο ή μόνο το τελευταίο chunk της ζητούμενης πληροφορίας αναφέρεται σε εσφαλμένο δίσκο, τότε απαιτείται από τους υπόλοιπους δίσκους του stripe να διαβασθεί μόνο η πληροφορία μεγέθους του αντίστοιχου chunk, ώστε να ανακτηθεί η ζητούμενη πληροφορία, η οποία πιθανότατα θα είναι μικρότερη του μεγέθους του chunk (chunksize). Εάν όμως κάποιο ενδιάμεσο chunk ή και το πρώτο και

το τελευταίο chunk βρίσκονται σε εσφαλμένο δίσκο, τότε θα πρέπει να διαβασθούν ολόκληρα τα chunk των υπόλοιπων δίσκων (data και parity). Στη συνέχεια παρατίθεται ο κώδικας που υλοποιεί τα παραπάνω.

```
fch=offset/vraid.chunksize;
lch=fch+nchunk-1;
fsp=fch/raidEC.nSD;
lsp=lch/raidEC.nSD;
if(fsp==lsp){
        rmode=2;
        command = (struct raidEC_command *)malloc(sizeof(struct raidEC_command) * (raidEC.nSD));
        commands_tosend=raidEC.nSD;
        tdnmrc=0;
        if(lch-fch>1){
                if((lch-1)%raidEC.nSD>fch%raidEC.nSD){
                        if(!sFD((fch+1)%raidEC.nRD,(lch-1)%raidEC.nRD,fsp))
                                tdnmrc=vraid.chunksize;
                }
                else if(!sFD((fch+1)%raidEC.nRD,raidEC.nRD-1,fsp)
                        || !sFD(0,(lch-1)%raidEC.nRD,fsp))
                        tdnmrc=vraid.chunksize;
        }
        if(!tdnmrc){
                if(!sFD(fch%raidEC.nRD,fch%raidEC.nRD,fsp)
                   && !sFD(lch%raidEC.nRD,lch%raidEC.nRD,fsp))
                        tdnmrc=vraid.chunksize;
                else if(!sFD(fch%raidEC.nRD,fch%raidEC.nRD,fsp)){
                        tdnmrc=vraid.chunksize-(offset%vraid.chunksize);
                        ofnmrc=lsp*vraid.chunksize+offset%vraid.chunksize;
                }
                else if(!sFD(lch%raidEC.nRD,lch%raidEC.nRD,fsp)){
                        tdnmrc=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
                        ofnmrc=fsp*vraid.chunksize;
                }
        }
        hbufSize=(nchunk-1)*vraid.chunksize+(raidEC.nSD-nchunk+1)*tdnmrc;
        hbuf=(unsigned char *)malloc(sizeof(unsigned char)*hbufSize*vraid.sectorsize);
        disk_index=(fsp*raidEC.nSD)%raidEC.nRD;
        done=0;
        create_eqFD(fsp);
        for(j=0,i=0,k=0;k<raidEC.nSD;k++){    //read data
                if(j>=raidEC.nGET || k<raidEC.eqFD[j]){
                        command[commandsIndex].device = raidEC.device[disk_index];
                        command[commandsIndex].request.sequence = commandsIndex;
                        command[commandsIndex].request.command = VRAID_COMMAND_READ;
                        command[commandsIndex].data = hbuf + (done * vraid.sectorsize);
                        if(tdnmrc==vraid.chunksize || (fch%raidEC.nSD<k && lch%raidEC.nSD>k)
                           || (ofnmrc==lsp*vraid.chunksize+offset%vraid.chunksize
                          && k==lch%raidEC.nSD)
                           || (ofnmrc==fsp*vraid.chunksize && k==fch%raidEC.nSD)){
                                command[commandsIndex].request.offset = fsp*vraid.chunksize;
```

```
					command[commandsIndex].request.length = vraid.chunksize;
					done+=vraid.chunksize;
				}
				else{
					command[commandsIndex].request.offset = ofnmrc;
					command[commandsIndex].request.length = tdnmrc;
					done+=tdnmrc;
				}
				command[commandsIndex].success = VRAID_COMMAND_PENDING;
				i++;

				commandsIndex++;
			}
			else
				j++;
			disk_index++;
			if(disk_index>=raidEC.nRD)
				disk_index=disk_index%raidEC.nRD;
		}
		parity_disk=((fsp+1)*raidEC.nSD)%raidEC.nRD;
		createGETable();
		for(j=0;j<raidEC.nGET;j++,i++){    //read parities
			disk_index=parity_disk+raidEC.GETable[j][raidEC.nGET];
			if(disk_index>=raidEC.nRD)
				disk_index=disk_index%raidEC.nRD;
			command[commandsIndex].device = raidEC.device[disk_index];
			command[commandsIndex].request.sequence = commandsIndex;
			command[commandsIndex].request.command = VRAID_COMMAND_READ;
			command[commandsIndex].request.offset = ofnmrc;
			command[commandsIndex].request.length = tdnmrc;
			command[commandsIndex].data = hbuf + (done * vraid.sectorsize);
			command[commandsIndex].success = VRAID_COMMAND_PENDING;
			done+=tdnmrc;
			commandsIndex++;
		}
}
⋮
switch(rmode){
	⋮
	case 2: createGETable();
				sm=(block **) malloc(sizeof(block *)*raidEC.nGET);
				for(i=0;i<raidEC.nGET;i++){
				    sm[i]=(block *) malloc(sizeof(block)*tdnmrc);
				    for(j=0;j<tdnmrc;j++)
					sm[i][j]=(block) malloc(sizeof(unsigned char)*vraid.sectorsize);
				}
				calParity(hbuf,ofnmrc%vraid.chunksize,tdnmrc,fch%raidEC.nSD,nchunk,sm);
				k=GausElim(sm,fch%raidEC.nSD,lch%raidEC.nSD,tdnmrc,1);
				for(bufindex=0,hbufindex=0,hbufSize=0,j=0,i=0;i<=lch%raidEC.nSD;i++){
				    if(i==fch%raidEC.nSD){
					if(j<raidEC.nGET && i==raidEC.eqFD[j]){
					    if(tdnmrc==vraid.chunksize)
						for(l=offset%vraid.chunksize;l<vraid.chunksize;l++,
						    bufindex++)
						    block_assign(buf+block_offset(bufindex),sm[k][l]);
```

```
                    else
                        for(l=0;l<tdnmrc;l++,bufindex++)
                            block_assign(buf+block_offset(bufindex),sm[k][l]);
                    k++;
                    j++;
                }
                else{
                    hbufSize+=vraid.chunksize;
                    for(hbufindex+=offset%vraid.chunksize;hbufindex<hbufSize
                        ;hbufindex++,bufindex++)
                            block_assign(buf+block_offset(bufindex),
                                        hbuf+block_offset(hbufindex));
                }
            }
            else if(i>fch%raidEC.nSD && i<lch%raidEC.nSD){
                if(j<raidEC.nGET && i==raidEC.eqFD[j]){
                    for(l=0;l<vraid.chunksize;l++,bufindex++)
                        block_assign(buf+block_offset(bufindex),sm[k][l]);
                        k++;
                        j++;
                }
                else{
                    hbufSize+=vraid.chunksize;
                    for(;hbufindex<hbufSize;hbufindex++,bufindex++)
                        block_assign(buf+block_offset(bufindex),
                                    hbuf+block_offset(hbufindex));
                }
            }
            else if(i==lch%raidEC.nSD){
                if(j<raidEC.nGET && i==raidEC.eqFD[j]){
                    for(l=0;l<count- (nchunk-1)* vraid.chunksize +
                        offset % vraid.chunksize;l++,bufindex++)
                        block_assign(buf+block_offset(bufindex),sm[k][l]);
                    k++;
                    j++;
                }
                else{
                    hbufSize+=count- (nchunk-1)* vraid.chunksize
                                + offset % vraid.chunksize;
                    for(;hbufindex<hbufSize;hbufindex++,bufindex++)
                        block_assign(buf+block_offset(bufindex),hbuf
                                    +block_offset(hbufindex));
                }
            }
            else if(j<raidEC.nGET && i==raidEC.eqFD[j])
                j++;
            else{
                hbufindex+=tdnmrc;
                hbufSize+=tdnmrc;
            }
        }
        break;
    ⋮
}
```

Η τρίτη περίπτωση που διακρίνεται από τον κώδικα όταν κάποιος από τους ενδιαφερόμενους δίσκους είναι εσφαλμένος, είναι η περίπτωση που η ενδιαφερόμενη πληροφορία περιέχεται σε περισσότερα του ενός stripe. Στην περίπτωση αυτή, και στην προσπάθεια για την επίτευξη ενός αποδοτικότερου συστήματος, γίνονται οι παρακάτω έλεγχοι ανά stripe.

Για τα stripes, για τα οποία απαιτείται όλη η πληροφορία των data δίσκων, ελέγχεται αρχικά κατά πόσο οι εσφαλμένοι δίσκοι αποτελούν δίσκους δεδομένων για το συγκεκριμένο stripe. Στην περίπτωση που όλοι οι data δίσκοι του stripe δεν είναι εσφαλμένοι, τότε απλά διαβάζονται ολόκληρα τα chunk των δίσκων αυτών. Στην περίπτωση που για το συγκεκριμένο stripe υπάρχουν data δίσκοι που θεωρούνται εσφαλμένοι, τότε διαβάζονται ολόκληρα τα chunk όλων των διαθέσιμων δίσκων, data και parities για το stripe αυτό. Έπειτα, ανακτάται η ζητούμενη πληροφορία με την βοήθεια των μεθόδων της κωδικοποίησης Reed-Solomon και της απαλοιφής του Gauss.

Στην περίπτωση που το πρώτο και το τελευταίο stripe περιέχουν τη ζητούμενη πληροφορία, εφαρμόζεται η διαδικασία της προηγούμενης κατηγορίας, όπου η ζητούμενη πληροφορία περιορίζεται σε ένα μόνο stripe. Έτσι, για την κάθε περίπτωση (πρώτου ή τελευταίου stripe) ελέγχεται αρχικά εάν οι ζητούμενοι δίσκοι θεωρούνται εσφαλμένοι, και αν η απάντηση είναι θετική, ποιοι δίσκοι είναι αυτοί. Στην περίπτωση που είναι μόνο το πρώτο ή το τελευταίο (αντιστοίχως) chunk της ζητούμενης πληροφορίας, τότε ελέγχεται το μέγεθος του chunk που ενδιαφέρει. Αν αυτό είναι μικρότερο από το μέγεθος ολόκληρου του chunk, τότε από τους υπόλοιπους δίσκους του stripe θα διαβαστεί μόνο το αντίστοιχο κομμάτι των chunks τους, επιτυγχάνοντας τη μικρότερη δυνατή ανάγνωση δεδομένων. Στο σημείο αυτό πρέπει να επισημανθεί ότι το όφελος της διαδικασίας αυτής είναι περισσότερο εμφανές στην περίπτωση που το RAID EC σύστημα παραμετροποιηθεί ώστε τα stripe του να αποτελούνται από αρκετά ευμεγέθη chunks.

Το τμήμα του κώδικα που υλοποιεί τα παραπάνω φαίνεται στη συνέχεια:

```
for(commands_tosend=0,hbufSize=0,stripenum=fsp;stripenum<=lsp;stripenum++){
  rmode=3;
  tdnmrc=0;
  if(stripenum==fsp){
     if((fsp*raidEC.nSD)%raidEC.nRD>(raidEC.nRD-raidEC.nSD)){
        if(!sFD(fch%raidEC.nRD,raidEC.nRD-1,stripenum)
           || !sFD(0,((fsp+1)*raidEC.nSD-1)%raidEC.nRD,stripenum)){
                ofnmrc=stripenum*vraid.chunksize;
                tdnmrc=vraid.chunksize;
           }
      }
      else
         if(!sFD(fch%raidEC.nRD,((fsp+1)*raidEC.nSD-1)%raidEC.nRD,stripenum)){
             ofnmrc=stripenum*vraid.chunksize;
             tdnmrc=vraid.chunksize;
         }
      if(tdnmrc==0 && !sFD(fch%raidEC.nRD,fch%raidEC.nRD,stripenum)){
         tdnmrc=vraid.chunksize-(offset%vraid.chunksize);
         ofnmrc=fsp*vraid.chunksize+offset%vraid.chunksize;
      }
   }

  else if(stripenum==lsp){
     if((lsp*raidEC.nSD)%raidEC.nRD>lch%raidEC.nRD){
         if(!sFD((lsp*raidEC.nSD)%raidEC.nRD,raidEC.nRD-1,stripenum)
           || !sFD(0,lch%raidEC.nRD,stripenum)){
               ofnmrc=stripenum*vraid.chunksize;
               tdnmrc=vraid.chunksize;
           }
     }
     else
        if(!sFD((lsp*raidEC.nSD)%raidEC.nRD,lch%raidEC.nRD,stripenum)){
             ofnmrc=stripenum*vraid.chunksize;
             tdnmrc=vraid.chunksize;
        }
     if(tdnmrc==0 && !sFD(lch%raidEC.nRD,lch%raidEC.nRD,stripenum)){
         tdnmrc=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
         ofnmrc=lsp*vraid.chunksize;
      }
   }

   else{
     ofnmrc=stripenum*vraid.chunksize;
     tdnmrc=vraid.chunksize;
   }
```

```
if(tdnmrc==0){
  if(stripenum==fsp){
      rmode=4;
      commands_tosend=(fsp+1)*raidEC.nSD-fch;
      command = (struct raidEC_command *)malloc(sizeof(struct raidEC_command)
                    * commands_tosend);
      hbufSize=((fsp+1)*raidEC.nSD-fch)*vraid.chunksize-offset%vraid.chunksize;
      disk_index=fch%raidEC.nRD;
      done=0;
      for(i=0;i<commands_tosend;i++,disk_index++){
          command[commandsIndex].device = raidEC.device[disk_index%raidEC.nRD];
          command[commandsIndex].request.sequence = commandsIndex;
          command[commandsIndex].request.command = VRAID_COMMAND_READ;
          if(i==0){
              command[commandsIndex].request.offset = fsp*vraid.chunksize+offset%vraid.chunksize;
              command[commandsIndex].request.length = vraid.chunksize-(offset%vraid.chunksize);
              hbuf=(unsigned char *)malloc(sizeof(unsigned char)*(vraid.chunksize
                                        -(offset%vraid.chunksize))*vraid.sectorsize);
              done+=vraid.chunksize-(offset%vraid.chunksize);
          }
          else{
              command[commandsIndex].request.offset = fsp*vraid.chunksize;
              command[commandsIndex].request.length = vraid.chunksize;
              hbuf=(unsigned char *)malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
              done+=vraid.chunksize;
          }
          command[commandsIndex].data = hbuf;
          command[commandsIndex].success = VRAID_COMMAND_PENDING;
          commandsIndex++;
      }
  }

 else if(stripenum==lsp){
    if(rmode==4)
        rmode=6;
    else
        rmode=5;
    commands_tosend+=lch-lsp*raidEC.nSD+1;
    command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command)
                  * commands_tosend);
    hbufSize+=(lch-lsp*raidEC.nSD)*vraid.chunksize+count- (nchunk-1)* vraid.chunksize
                  + offset % vraid.chunksize;
    disk_index=(lsp*raidEC.nSD)%raidEC.nRD;
    for(;i<commands_tosend;i++,disk_index++){
      command[commandsIndex].device = raidEC.device[disk_index%raidEC.nRD];
      command[commandsIndex].request.sequence = commandsIndex;
```

```
            command[commandsIndex].request.command = VRAID_COMMAND_READ;
            if(disk_index==fch%raidEC.nRD){
                command[commandsIndex].request.offset = lsp*vraid.chunksize;
                command[commandsIndex].request.length = count- (nchunk-1)* vraid.chunksize
                                                         + offset % vraid.chunksize;
                hbuf=(unsigned char *)malloc(sizeof(unsigned char)*(count- (nchunk-1)* vraid.chunksize
                                        + offset % vraid.chunksize)*vraid.sectorsize);
                done+=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
            }
            else{
                command[commandsIndex].request.offset = lsp*vraid.chunksize;
                command[commandsIndex].request.length = vraid.chunksize;
                hbuf=(unsigned char *)malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
                done+=vraid.chunksize;
            }
            command[commandsIndex].data = hbuf;
            command[commandsIndex].success = VRAID_COMMAND_PENDING;
            commandsIndex++;
        }
    }
    continue;
}

commands_tosend+=raidEC.nSD;
if(stripenum==fsp)
     command = (struct raidEC_command *)malloc(sizeof(struct raidEC_command) * commands_tosend);
else
     command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command)*commands_tosend);
if(tdnmrc==vraid.chunksize)
     hbufSize+=raidEC.nSD*vraid.chunksize;
else if(stripenum==fsp)
    hbufSize+=((fsp+1)*raidEC.nSD-fch-1)*vraid.chunksize+(fch+1-fsp*raidEC.nSD)*tdnmrc;
else
    hbufSize+=(lch-lsp*raidEC.nSD)*vraid.chunksize+((lsp+1)*raidEC.nSD-lch)*tdnmrc;

disk_index=(stripenum*raidEC.nSD)%raidEC.nRD;
create_eqFD(stripenum);
for(j=0,i=0,k=0;k<raidEC.nSD;k++,disk_index++){
    if(j>=raidEC.nGET || k<raidEC.eqFD[j]){
        command[commandsIndex].device = raidEC.device[disk_index%raidEC.nRD];
        command[commandsIndex].request.sequence = commandsIndex;
        command[commandsIndex].request.command = VRAID_COMMAND_READ;
        if(tdnmrc==vraid.chunksize || (stripenum==fsp && fch<fsp*raidEC.nSD+k)
          || (stripenum==lsp && lch>lsp*raidEC.nSD+k)){
                command[commandsIndex].request.offset = stripenum*vraid.chunksize;
                command[commandsIndex].request.length = vraid.chunksize;
                hbuf=(unsigned char *)malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
                done+=vraid.chunksize;
        }
```

```
            else{
                command[commandsIndex].request.offset = ofnmrc;
                command[commandsIndex].request.length = tdnmrc;
                hbuf=(unsigned char *)malloc(sizeof(unsigned char)*tdnmrc*vraid.sectorsize);
                done+=tdnmrc;
            }
            command[commandsIndex].data = hbuf;
            command[commandsIndex].success = VRAID_COMMAND_PENDING;
            i++;
            commandsIndex++;
        }
        else
            j++;
    }
    /********************** Read Parities************************/
    parity_disk=((stripenum+1)*raidEC.nSD)%raidEC.nRD;
    createGETable();
    for(j=0;j<raidEC.nGET;j++,i++){
        disk_index=parity_disk+raidEC.GETable[j][raidEC.nGET];
        if(disk_index>=raidEC.nRD)
            disk_index=disk_index%raidEC.nRD;
        command[commandsIndex].device = raidEC.device[disk_index];
        command[commandsIndex].request.sequence = commandsIndex;
        command[commandsIndex].request.command = VRAID_COMMAND_READ;
        command[commandsIndex].request.offset = ofnmrc;
        command[commandsIndex].request.length = tdnmrc;
        hbuf=(unsigned char *)malloc(sizeof(unsigned char)*tdnmrc*vraid.sectorsize);
        command[commandsIndex].data = hbuf;
        command[commandsIndex].success = VRAID_COMMAND_PENDING;
        done+=tdnmrc;
        commandsIndex++;
    }
}
⋮
switch(rmode){
        ⋮
        default: sm=(block **) malloc(sizeof(block *)*raidEC.nFD);
                for(i=0;i<raidEC.nFD;i++){
                    sm[i]=(block *) malloc(sizeof(block)*vraid.chunksize);
                    for(j=0;j<vraid.chunksize;j++)
                        sm[i][j]=(block) malloc(sizeof(unsigned char)*vraid.sectorsize);
                }

                for(bufindex=0,hbufindex=0,done=0,hbufSize=0,stripenum=fsp;stripenum<=lsp;stripenum++){
                    create_eqFD(stripenum);
                    createGETable();
```

```
if(stripenum==fsp){
    if(rmode==4 || rmode==6){
         done=((stripenum+1)*raidEC.nSD-fch)*vraid.chunksize-offset%vraid.chunksize;
         for(;bufindex<done;bufindex++)
              block_assign(buf+block_offset(bufindex),hbuf+block_offset(hbufindex));
        hbufindex=bufindex;
        continue;
    }
    tdnmrc=command[raidEC.nSD-1].request.length;
    calParity(hbuf,ofnmrc%vraid.chunksize,tdnmrc,fch%raidEC.nSD,(stripenum+1)*raidEC.nSD
              -fch,sm);
    k=GausElim(sm,fch%raidEC.nSD,raidEC.nSD-1,tdnmrc,1);
    for(j=0,i=0;i<raidEC.nSD;i++){
        if(i==fch%raidEC.nSD){
           if(j<raidEC.nGET && i==raidEC.eqFD[j]){
              if(tdnmrc==vraid.chunksize)
                 for(l=offset%vraid.chunksize;l<vraid.chunksize;l++,bufindex++)
                       block_assign(buf+block_offset(bufindex),sm[k][l]);
              else
                 for(l=0;l<tdnmrc;l++,bufindex++)
                       block_assign(buf+block_offset(bufindex),sm[k][l]);
               k++;
               j++;
           }
           else{
              hbufSize+=vraid.chunksize;
              for(hbufindex+=offset%vraid.chunksize;hbufindex<hbufSize;hbufindex++,bufindex++)
                   block_assign(buf+block_offset(bufindex),hbuf+block_offset(hbufindex));
           }
           done=bufindex;
        }
        else if(i>fch%raidEC.nSD && i<raidEC.nSD){
           if(j<raidEC.nGET && i==raidEC.eqFD[j]){
               for(l=0;l<vraid.chunksize;l++,bufindex++){
                   block_assign(buf+block_offset(bufindex),sm[k][l]);
               }
               k++;
               j++;
           }
           else{
              hbufSize+=vraid.chunksize;
              for(;hbufindex<hbufSize;hbufindex++,bufindex++)
                   block_assign(buf+block_offset(bufindex),hbuf+block_offset(hbufindex));
           }
           done=bufindex;
        }
        else if(j<raidEC.nGET && i==raidEC.eqFD[j])
             j++;
        else{
             hbufindex+=tdnmrc;
             hbufSize+=tdnmrc;
        }
```

```
            }
            hbufindex+=raidEC.nGET*tdnmrc;
            hbufSize=hbufindex;
        }
        else if(stripenum==lsp){
            if(rmode>4){
                done+=((stripenum+1)*raidEC.nSD-fch-1)*vraid.chunksize+(offset+count
                        -1)%vraid.chunksize+1;
                for(;bufindex<done;bufindex++,hbufindex++){
                    block_assign(buf+block_offset(bufindex),hbuf+block_offset(hbufindex));
                }
                break;
            }
            tdnmrc=command[commands_tosend-1].request.length;
            calParity(hbuf+hbufindex,0,tdnmrc,0,lch-stripenum*raidEC.nSD+1,sm);
            k=GausElim(sm,0,lch%raidEC.nSD,tdnmrc,1);
            for(j=0,i=0;i<=lch%raidEC.nSD;i++){
                if(i>=0 && i<lch%raidEC.nSD){
                    if(j<raidEC.nGET && i==raidEC.eqFD[j]){
                        for(l=0;l<vraid.chunksize;l++,bufindex++)
                            block_assign(buf+block_offset(bufindex),sm[k][l]);
                        k++;
                        j++;
                    }
                    else{
                        hbufSize+=vraid.chunksize;
                        for(;hbufindex<hbufSize;hbufindex++,bufindex++)
                            block_assign(buf+block_offset(bufindex),hbuf+block_offset(hbufindex));
                    }
                    done=bufindex;
                }
                else if(i==lch%raidEC.nSD){
                    if(j<raidEC.nGET && i==raidEC.eqFD[j]){
                        for(l=0;l<count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
                            l++,bufindex++)
                            block_assign(buf+block_offset(bufindex),sm[k][l]);
                        k++;
                        j++;
                }
                else{
                    hbufSize+=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
                    for(;0<(int) (hbufSize-hbufindex);hbufindex++,bufindex++){
                        block_assign(buf+block_offset(bufindex),hbuf+block_offset(hbufindex));
                    }
                    done=bufindex;
                }
            }
        }
    }
    else{
        tdnmrc=vraid.chunksize;
        calParity(hbuf+hbufindex,0,vraid.chunksize,0,raidEC.nSD,sm);
        k=GausElim(sm,0,raidEC.nSD-1,vraid.chunksize,1);
        for(j=0,i=0;i<raidEC.nSD;i++){
            if(j<raidEC.nGET && i==raidEC.eqFD[j]){
```

```
                    for(l=0;l<vraid.chunksize;l++,bufindex++)
                        block_assign(buf+block_offset(bufindex),sm[k][l]);
                    k++;
                    j++;
                }
                else{
                    hbufSize+=vraid.chunksize;
                    for(;hbufindex<hbufSize;hbufindex++,bufindex++)
                        block_assign(buf+block_offset(bufindex),hbuf+block_offset(hbufindex));
                }
            }
            done=bufindex;
            hbufindex+=raidEC.nGET*tdnmrc;
            hbufSize=hbufindex;
        }
    }
    break;
}
```

4.3.3 Υλοποίηση της διεργασίας write

Η διεργασία write είναι μία από τις βασικότερες διεργασίες του RAID EC και είναι υπεύθυνη για την ικανοποίηση των αιτημάτων για εγγραφή στη συστοιχία των δίσκων. Ακολουθώντας τον αλγόριθμο που περιγράφηκε στην ενότητα 3.2.2, η υλοποίηση της διεργασίας γίνεται ακολουθώντας την παρακάτω μεθοδολογία.

Όπως περιγράφηκε και στην ενότητα 3.2.2, σκοπός της διεργασίας είναι τόσο η εγγραφή των ζητούμενων δεδομένων στη συστοιχία των δίσκων και εν γένει στο RAID EC σύστημα, αλλά και η σχετική ενημέρωση των parities, ώστε να είναι δυνατή η ανάκτηση της δεδομένης πληροφορία σε περίπτωση απώλειας κάποιου δίσκου. Επιπρόσθετα, η ενημέρωση των parities είναι απαραίτητη στην περίπτωση που κάποιος από τους δίσκους δεδομένων, όπου αναφέρεται η πληροφορία προς εγγραφή, είναι εσφαλμένος, ώστε να είναι δυνατή εκ των υστέρων η ανάκτηση της προς εγγραφής πληροφορία. Ο αλγόριθμος που περιγράφηκε στην προαναφερθείσα ενότητα, καθώς και η υλοποίηση που παρουσιάζεται στην παρούσα ενότητα, έχει τον επιπρόσθετο στόχο της επίτευξης των παραπάνω με τον καλύτερο δυνατό τρόπο. Αυτό πρακτικά ισοδυναμεί με την επίτευξη της εγγραφής της ζητούμενης πληροφορίας και της ενημέρωσης των parities με όσο το δυνατόν λιγότερες κλήσεις στους δίσκους του συστήματος. Επιπλέον, στην υλοποίηση που θα περιγραφεί στη συνέχεια γίνεται προσπάθεια οι κλήσεις ανάγνωσης, οι οποίες γίνονται στους δίσκους του συστήματος, να αναφέρονται στο μικρότερο δυνατό όγκο δεδομένων.

Η μεθοδολογία την οποία ακολουθεί η παρούσα υλοποίηση της διεργασίας, για την επίτευξη των παραπάνω, είναι η εξής. Αρχικά, εντοπίζονται τα stripes του συστήματος όπου αναφέρεται η πληροφορία προς εγγραφή. Έτσι στη συνέχεια εκτελεί ανά stripe τις παρακάτω ενέργειες. Διαβάζει, για το κάθε ενδιαφερόμενο stripe, τα chunk των δίσκων του RAID EC συστήματος, τα οποία περιέχουν την υπάρχουσα πληροφορία που απαιτείται ανά περίπτωση για να καταστεί δυνατή η ενημέρωση των parities, ενώ παράλληλα κλειδώνει τα chunk τα οποία πρόκειται να εγγραφούν από τη διαδικασία, δηλαδή τα chunk στα οποία αναφέρεται η πληροφορία προς εγγραφή, όπως και τα chunk των parities. Έπειτα, έχοντας μαζέψει την απαιτούμενη πληροφορία για όλα τα αναφερόμενα stripe για τη δεδομένη εγγραφή, η διεργασία κινείται πάλι ανά stripe για τον υπολογισμό των ενημερωμένων parities και την εγγραφή αυτών, μαζί με την πληροφορία προς εγγραφή, στους αντίστοιχους δίσκους του συστήματος, ξεκλειδώνοντάς τους παράλληλα.

Αναλυτικότερα, όπως αναφέρθηκε και προηγουμένως, τα chunks του συστήματος που διαβάζονται από τη διεργασία, διαφέρουν ανά περίπτωση και κατά αντίστοιχο τρόπο διαφοροποιείται και ο τρόπος υπολογισμού των νέων parities. Συνεπώς, βάσει και των όσων έχουν περιγραφεί στην ενότητα 3.2.2, η παρούσα υλοποίηση διαχωρίζει τις παρακάτω περιπτώσεις για το stripe που αναφέρεται κάθε φόρα:

1. Η πληροφορία προς εγγραφής ή τμήμα αυτής αναφέρεται σε ολόκληρη την πληροφορία δεδομένων του δεδομένου stripe.

2. Η πληροφορία προς εγγραφή δεν αναφέρεται σε ολόκληρη την πληροφορία δεδομένων του δεδομένου stripe.

    i. Όλα τα chunk δεδομένων του stripe, στα οποία δεν αναφέρεται η πληροφορία προς εγγραφή, είναι διαθέσιμα και τα υπόλοιπα chunk του stripe, στα οποία αναφέρεται η πληροφορία προς εγγραφή, είναι περισσότερα από τα πρώτα ή δεν είναι όλα διαθέσιμα.

    ii. Τα chunk του stripe, στα οποία αναφέρεται η πληροφορία προς εγγραφή, είναι όλα διαθέσιμα και είτε τα υπόλοιπα chunk των δίσκων δεδομένων δεν είναι όλα διαθέσιμα, είτε είναι περισσότερα από τα πρώτα.

    iii. Υπάρχουν εσφαλμένα chunk, τόσο ανάμεσα στα chunk όπου αναφέρεται η πληροφορία προς εγγραφή, όσο και στα υπόλοιπα chunk δεδομένων του stripe, αναφορικά με το συγκεκριμένο stripe.

Επομένως, η διεργασία διατρέχει τα stripe στα οποία αναφέρεται η πληροφορία προς εγγραφή και στην περίπτωση που εντοπιστεί stripe του οποίου όλα τα chunk είναι προς

εγγραφή, δηλαδή εάν ανήκει στην 1η περίπτωση, η μεθοδολογία που ακολουθείται είναι η παρακάτω. Επειδή, η πληροφορία προς εγγραφή καταλαμβάνει όλο τα chunk δεδομένων του συγκεκριμένου stripe, δεν είναι απαραίτητη η ανάγνωση κάποιας πληροφορία που υπάρχει ήδη εκεί. Από την εισερχόμενη για τη διεργασία πληροφορία μπορεί να υπολογιστούν πλήρως όλα τα parities του stripe και στη συνέχεια απλά να εγγραφεί στους διαθέσιμους δίσκους η αντίστοιχη πληροφορία (δεδομένα ή parities). Σε κάθε περίπτωση, όμως, θα πρέπει να κλειδωθούν όλα τα διαθέσιμα chunk του stripe καθ' όλη τη διάρκεια εκτέλεσης της διεργασίας, ώστε να εξασφαλιστεί ο σωστός συγχρονισμός των write και η μη αλλοίωση των δεδομένων που αποθηκεύονται στο RAID EC σύστημα. Το τμήμα του κώδικα που υλοποιεί τόσο το lock των chunk του stripe, όσο και αυτό που πραγματοποιεί τον υπολογισμό των parities και την εγγραφή της συνολικής πληροφορίας στους δίσκους του stripe, παρουσιάζεται στη συνέχεια.

```
⋮
/***************** Lock the corresponding chunk ******************/
wmode=(int *)malloc(sizeof(int)*(lsp-fsp+1));
command = (struct raidEC_command *)malloc(sizeof(struct raidEC_command));
hbuf=(unsigned char *)malloc(sizeof(unsigned char));
bm=(unsigned char *)malloc(sizeof(unsigned char)*vraid.sectorsize);
commands_tosend=0;
hbufSize=hbufindex=commandsIndex=0;
for(stripenum=fsp;stripenum<=lsp;stripenum++){
   printf("raidEC_write:RL: Inside first for stripe:%d...\n",stripenum);
   if((fsp<stripenum && stripenum<lsp)
     || (fsp<lsp && ((fsp==stripenum && fch%raidEC.nSD==0 && offset%vraid.chunksize==0)
     || (lsp==stripenum && (lch+1)%raidEC.nSD==0 && count- nchunk* vraid.chunksize +
     offset % vraid.chunksize==0))) || (fsp==lsp && count==raidEC.nSD*vraid.chunksize)){
      wmode[stripenum-fsp]=1;
      commands_tosend+=raidEC.nRD-raidEC.nFD;
      command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
      for(j=0,i=0;i<raidEC.nRD;i++){
         if(j<raidEC.nFD && i==raidEC.failedDisks[j].disk){
            j++;
            continue;
         }
         command[commandsIndex].device = raidEC.device[i];
         command[commandsIndex].request.sequence = commandsIndex;
         command[commandsIndex].request.command = VRAID_COMMAND_LOCK;
         command[commandsIndex].request.offset = stripenum*vraid.chunksize;
         command[commandsIndex].request.length = vraid.chunksize;
         command[commandsIndex].data = NULL;
         command[commandsIndex].success = VRAID_COMMAND_PENDING;
         commandsIndex++;
      }
   }
  ⋮
}
⋮
```

```
/********* Write and Unlock the corresponding data **********/
hbcp=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
for(commands_tosend=0,commandsIndex=0,bufindex=0,hbufindex=0,stripenum=fsp;stripenum<=lsp;stripenum++){
   parity_disk=((stripenum+1)*raidEC.nSD)%raidEC.nRD;
   if(wmode[stripenum-fsp]>1){
      if(fsp==lsp){
         csfch=fch;
         cslch=lch;
         ofcs=stripenum*vraid.chunksize+offset%vraid.chunksize;
         if(csfch==cslch)
             tdcs=count;
         else
             tdcs=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
      }
      else if(stripenum==fsp){
          csfch=fch;
          cslch=(stripenum+1)*raidEC.nSD-1;
          ofcs=stripenum*vraid.chunksize+offset%vraid.chunksize;
          tdcs=vraid.chunksize;
          if(csfch==cslch)
            tdcs-=ofcs%vraid.chunksize;
      }
     else if(stripenum==lsp){
           csfch=stripenum*raidEC.nSD;
           cslch=lch;
           ofcs=stripenum*vraid.chunksize;
           tdcs=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
     }
 }
create_eqFD(stripenum);
switch(wmode[stripenum-fsp]){
  case 1: commands_tosend+=raidEC.nRD-raidEC.nFD;
           command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
           for(k=raidEC.nGET,i=0;i<raidEC.nCsD;i++){
              if(k<raidEC.nFD && i==raidEC.eqFD[k]){
                 k++;
                 continue;
              }
            hbcp=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
            for(l=0;l<vraid.chunksize;l++)
              block_init(&hbcp[block_offset(l)],0);
            for(j=0;j<raidEC.nSD;j++){
              for(l=0;l<vraid.chunksize;l++){
                 block_mult(buf+block_offset(bufindex+j*vraid.chunksize+l),raidEC.BMatrix[i][j],bm);
                 block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
              }
            }
           command[commandsIndex].device = raidEC.device[(parity_disk+i)%raidEC.nRD];
           command[commandsIndex].request.sequence = commandsIndex;
           command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
           command[commandsIndex].request.offset = stripenum*vraid.chunksize;
           command[commandsIndex].request.length = vraid.chunksize;
           command[commandsIndex].data = hbcp;
           command[commandsIndex].success = VRAID_COMMAND_PENDING;
           commandsIndex++;
```

```
    }
    for(j=0,i=0;i<raidEC.nSD;i++){
      if(j<raidEC.nGET && i==raidEC.eqFD[j]){
        j++;
        bufindex+=vraid.chunksize;
        continue;
      }
      command[commandsIndex].device = raidEC.device[(stripenum*raidEC.nSD+i)%raidEC.nRD];
      command[commandsIndex].request.sequence = commandsIndex;
      command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
      command[commandsIndex].request.offset = stripenum*vraid.chunksize;
      command[commandsIndex].request.length = vraid.chunksize;
      command[commandsIndex].data = buf+block_offset(bufindex);
      command[commandsIndex].success = VRAID_COMMAND_PENDING;
      commandsIndex++;
      bufindex+=vraid.chunksize;
    }
    break;
  ⋮
  }
}
```

Στην περίπτωση που δεν είναι όλα τα chunk του δεδομένου stripe προς εγγραφή, τότε η διεργασία προχωράει στους παρακάτω 2 ελέγχους, οι οποίοι αποτελούν και σημαντικά κριτήρια απόφασης για τον αλγόριθμο της διεργασίας. Συγκεκριμένα, αρχικά πραγματοποιείται ένας έλεγχος για τα ακριβά όρια της πληροφορίας προς εγγραφή στα chunk του δεδομένου stripe, υπολογίζοντας παράλληλα και το ποσοστό των chunk που είναι προς εγγραφή σε σχέση με τα υπόλοιπα chunk δεδομένων. Έπειτα, με την βοήθεια των συναρτήσεων (wdok, raidEC_locate_data, sFD) που παρουσιάστηκαν και στην ενότητα 4.3.2, γίνεται έλεγχος για τον εντοπισμό των εσφαλμένων δίσκων δεδομένων και τον καθορισμό του κατά πόσο υπάρχουν εσφαλμένα chunk ανάμεσα σε αυτά προς εγγραφή ή σε αυτά που δεν είναι προς εγγραφή ή και στα δυο σύνολα των chunk. Στο σημείο αυτό πρέπει να επισημανθεί ότι στην περίπτωση που ένα chunk δεν είναι ολόκληρο προς εγγραφή θεωρείται, στα πλαίσια των παραπάνω ελέγχων, μέλος του συνόλου των chunk προς εγγραφή. Στο παρακάτω τμήμα κώδικα της διεργασίας write παρουσιάζεται η υλοποίηση των παραπάνω ελέγχων.

```
parity_disk=((stripenum+1)*raidEC.nSD)%raidEC.nRD;
if(fsp==lsp){
 csfch=fch;
 cslch=lch;
 ofcs=stripenum*vraid.chunksize+offset%vraid.chunksize;
 if(csfch==cslch)
  tdcs=count;
 else
  tdcs=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
}
else if(stripenum==fsp){
 csfch=fch;
 cslch=(stripenum+1)*raidEC.nSD-1;
 ofcs=stripenum*vraid.chunksize+offset%vraid.chunksize;
 tdcs=vraid.chunksize;
 if(csfch==cslch)
  tdcs-=ofcs%vraid.chunksize;
}
else if(stripenum==lsp){
 csfch=stripenum*raidEC.nSD;
 cslch=lch;
 ofcs=stripenum*vraid.chunksize;
 tdcs=count- (nchunk-1)* vraid.chunksize + offset % vraid.chunksize;
}
if(cslch+1-csfch<raidEC.nSD/2+raidEC.nSD%2)
 lhr=1;
else
 lhr=0;
if(cslch<csfch){
 if(sFD(csfch%raidEC.nRD,raidEC.nRD-1,stripenum) && sFD(0,cslch%raidEC.nRD,stripenum))
  wd_ok=1;
 else
  wd_ok=0;
 if(stripenum*raidEC.nSD<csfch){
  if(sFD((stripenum*raidEC.nSD)%raidEC.nRD,(csfch-1)%raidEC.nRD,stripenum))
   nwd_ok=1;
  else
   nwd_ok=0;
 }
 else
  nwd_ok=1;
 if(nwd_ok){
  if(cslch<(stripenum+1)*raidEC.nSD-1){
   if(sFD((cslch+1)%raidEC.nRD,((stripenum+1)*raidEC.nSD-1)%raidEC.nRD,stripenum))
    nwd_ok=1;
   else
    nwd_ok=0;
  }
  else
   nwd_ok=1;
  if(nwd_ok && csfch<cslch){
   if(ofcs%vraid.chunksize>0){
    if(sFD(csfch%raidEC.nRD,csfch%raidEC.nRD,stripenum))
     nwd_ok=1;
    else
```

```
        nwd_ok=0;
      }
      if(tdcs<vraid.chunksize){
        if(sFD(cslch%raidEC.nRD,cslch%raidEC.nRD,stripenum))
          nwd_ok=1;
        else
          nwd_ok=0;
      }
    }
  }
}
else{
  if(sFD(csfch%raidEC.nRD,cslch%raidEC.nRD,stripenum))
    wd_ok=1;
  else
    wd_ok=0;
  if(stripenum*raidEC.nSD<csfch){
    if((csfch-1)%raidEC.nRD<(stripenum*raidEC.nSD)%raidEC.nRD){
      if(sFD((stripenum*raidEC.nSD)%raidEC.nRD,raidEC.nRD-1,stripenum) && sFD(0,(csfch-
1)%raidEC.nRD,stripenum))
        nwd_ok=1;
      else
        nwd_ok=0;
    }
    else{
      if(sFD((stripenum*raidEC.nSD)%raidEC.nRD,(csfch-1)%raidEC.nRD,stripenum))
        nwd_ok=1;
      else
        nwd_ok=0;
    }
  }
  else
    nwd_ok=1;
  if(nwd_ok){
    if(cslch<(stripenum+1)*raidEC.nSD-1){
      if(((stripenum+1)*raidEC.nSD-1)%raidEC.nRD<(cslch+1)%raidEC.nRD){
        if(!sFD((cslch+1)%raidEC.nRD,raidEC.nRD-1,stripenum) || !sFD(0,((stripenum+1)*raidEC.nSD
                                                                        -1)%raidEC.nRD,stripenum))
          nwd_ok=0;
      }
      else{
        if(!sFD((cslch+1)%raidEC.nRD,((stripenum+1)*raidEC.nSD-1)%raidEC.nRD,stripenum))
          nwd_ok=0;
      }
    }
    if(nwd_ok && csfch<cslch){
      if(ofcs%vraid.chunksize>0 && !sFD(csfch%raidEC.nRD,csfch%raidEC.nRD,stripenum))
        nwd_ok=0;
      if(nwd_ok && tdcs<vraid.chunksize && !sFD(cslch%raidEC.nRD,cslch%raidEC.nRD,stripenum))
        nwd_ok=0;
    }
  }
}
```

Στη συνέχεια, βάσει των αποτελεσμάτων των παραπάνω ελέγχων, διακρίνουμε την περίπτωση που, για το δεδομένο stripe, τα chunks τα οποία δεν είναι προς εγγραφή είναι διαθέσιμα και τα υπόλοιπα chunks δεδομένων είτε είναι διαθέσιμα, αλλά περισσότερα σε αριθμό από τα πρώτα, είτε δεν είναι όλα διαθέσιμα. Στην περίπτωση αυτή, σύμφωνα και με τα όσα έχουν αναφερθεί και στην ενότητα 3.2.2, οι ενέργειες που λαμβάνουν χώρα από τη διεργασία είναι η ανάγνωση των chunk δεδομένων που δεν είναι προς εγγραφή και το κλείδωμα όλων των chunk (δεδομένων και parities) του δεδομένου stripe. Έπειτα, συνδυάζοντας τα δεδομένα που αναγνώσθηκαν προηγουμένως και τα δεδομένα προς εγγραφή, υπολογίζονται τα νέα ενημερωμένα parities. Με τον τρόπο αυτό επιτυγχάνεται ο υπολογισμός των νέων parities με τις όσο το δυνατόν λιγότερες κλήσεις (ανάγνωσης) προς τους δίσκους του RAID EC συστήματος. Το κλείδωμα των chunk γίνεται με σκοπό την αποφυγή της αλλοίωσης των δεδομένων του RAID EC συστήματος από ένα διαφορετικό αίτημα εγγραφής. Τέλος, η διεργασία αποθηκεύει τα νέα δεδομένα στα κατάλληλα chunk δεδομένων, όπως και τα ενημερωμένα parities στα αντίστοιχα chunks, και ξεκλειδώνει τα chunks του stripe.

Στην παραπάνω όμως περίπτωση εμφανίζονται οι επόμενες ειδικές υποπεριπτώσεις, οι οποίες θα πρέπει να τύχουν κατάλληλου και όσο το δυνατόν βέλτιστου χειρισμού. Η ύπαρξη αυτών των υποπεριπτώσεων οφείλεται σε 2 λόγους. Πρώτον, στο γεγονός ότι η πληροφορία προς εγγραφή είναι δυνατό να μην αναφέρεται μόνο σε ολόκληρα chunks και δεύτερον, ότι τα chunks που δεν είναι ολόκληρα προς εγγραφή θεωρούνται από τους ελέγχους της διεργασίας write ως chunks προς εγγραφή. Για το λόγο αυτό πρέπει να ακολουθήσουν επιπλέον έλεγχοι για την περίπτωση που περιγράφηκε προηγουμένως. Όπως είναι κατανοητό, το φαινόμενο αυτό είναι δυνατόν να εμφανιστεί μόνο στο αρχικό ή τελικό chunk από εκείνα που προορίζονται για εγγραφή. Επομένως, διακρίνονται οι παρακάτω 3 υποπεριπτώσεις: (1) στο δεδομένο stripe να βρίσκεται μόνο το αρχικό chunk από αυτά που είναι προς εγγραφή και να μην είναι ολόκληρο προς εγγραφή, (2) να βρίσκεται μόνο το τελευταίο chunk ή (3) να βρίσκεται και το αρχικό και το τελευταίο chunk, όταν τα δεδομένα προς εγγραφή καταλαμβάνουν χώρο μικρότερο από αυτό ενός stripe του συστήματος. Σε κάθε περίπτωση, ανάγνωση γίνεται μόνο στα απαραίτητα sectors του chunk τα οποία δεν είναι προς εγγραφή. Στη συνέχεια παρουσιάζεται ο κώδικας που υλοποιεί όλα τα παραπάνω.

```
if(nwd_ok && (!lhr || !wd_ok)){
  wmode[stripenum-fsp]=2;
  commands_tosend+=raidEC.nRD-raidEC.nFD;
  command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
  for(j=0,i=0;i<raidEC.nRD;i++){
    if(j<raidEC.nFD && i==raidEC.failedDisks[j].disk){
    j++;
    continue;
    }
    command[commandsIndex].device = raidEC.device[i];
    command[commandsIndex].request.sequence = commandsIndex;
    if((cslch<csfch && (i>=csfch%raidEC.nRD || i<=cslch%raidEC.nRD)) ||
      (csfch<=cslch && i>=csfch%raidEC.nRD && i<=cslch%raidEC.nRD)){
      command[commandsIndex].request.command = VRAID_COMMAND_LOCK;
      if(i==csfch%raidEC.nRD){
        command[i].request.offset = ofcs;
        if(csfch==cslch)
          command[commandsIndex].request.length = tdcs;
        else
          command[commandsIndex].request.length = vraid.chunksize-ofcs%vraid.chunksize;
      }
      else if(i==cslch%raidEC.nRD){
        command[commandsIndex].request.offset = stripenum*vraid.chunksize;
        command[commandsIndex].request.length = tdcs;
      }
      else{
        command[commandsIndex].request.offset = stripenum*vraid.chunksize;
        command[commandsIndex].request.length = vraid.chunksize;
      }
      command[commandsIndex].data = NULL;
    }
    else if((parity_disk<=raidEC.nRD-raidEC.nCsD && i>=parity_disk && i<parity_disk+raidEC.nCsD) ||
         (parity_disk>raidEC.nRD-raidEC.nCsD && (i>=parity_disk ||
         i<(parity_disk+raidEC.nCsD)%raidEC.nRD))){
      command[commandsIndex].request.command = VRAID_COMMAND_LOCK;
      if(csfch<cslch){
        command[commandsIndex].request.offset = stripenum*vraid.chunksize;
        command[commandsIndex].request.length = vraid.chunksize;
      }
      else{
        command[commandsIndex].request.offset = ofcs;
        command[commandsIndex].request.length = tdcs;
      }
      command[commandsIndex].data = NULL;
    }
    else{
      if(csfch==cslch)
      {
        hbufSize+=tdcs;
        hbuf=(unsigned char *) malloc(sizeof(unsigned char)*tdcs*vraid.sectorsize);
      }
      else
      {
        hbufSize+=vraid.chunksize;
        hbuf=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
```

```
        }
        command[commandsIndex].request.command = VRAID_COMMAND_READ;
        command[commandsIndex].data = hbuf;
        if(csfch==cslch){
            command[commandsIndex].request.offset = ofcs;
            command[commandsIndex].request.length = tdcs;
            hbufindex+=tdcs;
        }
        else{
            command[commandsIndex].request.offset = stripenum*vraid.chunksize;
            command[commandsIndex].request.length = vraid.chunksize;
            hbufindex+=vraid.chunksize;
        }
    }
    command[commandsIndex].success = VRAID_COMMAND_PENDING;
    commandsIndex++;
}
if(csfch==cslch)
    continue;
if(sFD(csfch%raidEC.nRD,csfch%raidEC.nRD,stripenum) && ofcs%vraid.chunksize>0){
    wmode[stripenum-fsp]=5;
    commands_tosend++;
    command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
    hbufSize+=ofcs%vraid.chunksize;
    hbuf=(unsigned char *) malloc(sizeof(unsigned char)*(ofcs%vraid.chunksize)*vraid.sectorsize);
    command[commandsIndex].device = raidEC.device[csfch%raidEC.nRD];
    command[commandsIndex].request.sequence = commandsIndex;
    command[commandsIndex].request.command = VRAID_COMMAND_READ;
    command[commandsIndex].request.offset = stripenum*vraid.chunksize;
    command[commandsIndex].request.length = ofcs%vraid.chunksize;
    command[commandsIndex].data = hbuf;
    command[commandsIndex].success = VRAID_COMMAND_PENDING;
    hbufindex=hbufSize;
    commandsIndex++;
}
if(sFD(cslch%raidEC.nRD,cslch%raidEC.nRD,stripenum) && tdcs<vraid.chunksize){
    if(wmode[stripenum-fsp]==2)
        wmode[stripenum-fsp]=6;
    else
        wmode[stripenum-fsp]=7;
    commands_tosend++;
    command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
    hbufSize+=vraid.chunksize-tdcs;
    hbuf=(unsigned char *) malloc(sizeof(unsigned char)*(vraid.chunksize-tdcs)*vraid.sectorsize);
    command[commandsIndex].device = raidEC.device[cslch%raidEC.nRD];
    command[commandsIndex].request.sequence = commandsIndex;
    command[commandsIndex].request.command = VRAID_COMMAND_READ;
    command[commandsIndex].request.offset = stripenum*vraid.chunksize+tdcs;
    command[commandsIndex].request.length = vraid.chunksize-tdcs;
    command[commandsIndex].data = hbuf;
    command[commandsIndex].success = VRAID_COMMAND_PENDING;
    hbufindex=hbufSize;
    commandsIndex++;
}
}
```

```
⋮
switch(wmode[stripenum-fsp]){
  ⋮
  case 2:
  case 5:
  case 6:
  case 7:
      commands_tosend+=raidEC.nCsD+raidEC.nGET-raidEC.nFD;
      command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
      if(csfch==cslch)
        todo=tdcs;
      else
        todo=vraid.chunksize;
      hbufSize=hbufindex;
      bufSize=bufindex;
      for(k=raidEC.nGET,i=0;i<raidEC.nCsD;i++){
        if(k<raidEC.nFD && i==raidEC.eqFD[k]){
          k++;
          continue;
        }
        hbcp=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
        for(l=0;l<todo;l++)
          block_init(&hbcp[block_offset(l)],0);
        for(hbufindex=hbufSize,j=0;j<raidEC.nRD;j++){
          if(cslch%raidEC.nRD<csfch%raidEC.nRD){
            if((j>=csfch%raidEC.nRD && j<raidEC.nRD) || (j>=0 && j<=cslch%raidEC.nRD)
              || (j>=parity_disk && j<parity_disk+raidEC.nCsD))
              continue;
            if(j==(stripenum*raidEC.nSD)%raidEC.nRD)
              h=0;
            else if(j==(cslch+1)%raidEC.nRD)
              h=(cslch+1)%raidEC.nSD;
            else
              h++;
          }
          else if(parity_disk>raidEC.nSD){
            if((j>=csfch%raidEC.nRD && j<=cslch%raidEC.nRD)
              || (j>=parity_disk && j<raidEC.nRD) || (j>=0 && j<raidEC.nCsD-raidEC.nRD+parity_disk))
              continue;
            if(j==(stripenum*raidEC.nSD)%raidEC.nRD)
              h=0;
            else if(j==(cslch+1)%raidEC.nRD)
              h=(cslch+1)%raidEC.nSD;
            else
              h++;
          }
          else{
            if((j>=csfch%raidEC.nRD && j<=cslch%raidEC.nRD)
              || (j>=parity_disk && j<parity_disk+raidEC.nCsD))
              continue;
            if(j==(stripenum*raidEC.nSD)%raidEC.nRD)
              h=0;
            else if(j==(cslch+1)%raidEC.nRD)
              h=(cslch+1)%raidEC.nSD;
```

```
            else if(j==0){
                if(csfch%raidEC.nRD<(stripenum*raidEC.nSD)%raidEC.nRD)
                    h=raidEC.nRD-(stripenum*raidEC.nSD)%raidEC.nRD;
                else
                    h=(cslch+1)%raidEC.nSD+raidEC.nRD-(cslch+1)%raidEC.nRD;
            }
            else
                h++;
        }
        for(l=0;l<todo;l++,hbufindex++){
            block_mult(&hbuf[block_offset(hbufindex)],raidEC.BMatrix[i][h],bm);
            block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
        }
    }
    if(csfch%raidEC.nSD>0)
        for(f=0;f<raidEC.nGET;f++){
            if(raidEC.eqFD[f]>(csfch-1)%raidEC.nSD)
                break;
        }
    else
        f=0;
    for(bufindex=bufSize,j=csfch%raidEC.nSD;j<=cslch%raidEC.nSD;j++){
        if(f>=raidEC.nGET || j<raidEC.eqFD[f])
            commands_tosend++;
        else
            f++;
        if(wmode[stripenum-fsp]>2){
            if(j==csfch%raidEC.nSD && wmode[stripenum-fsp]!=6){
                for(l=0;l<ofcs%vraid.chunksize;l++,hbufindex++){
                    block_mult(&hbuf[block_offset(hbufindex)],raidEC.BMatrix[i][j],bm);
                    block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                }
                h=vraid.chunksize;
            }
            else if(j==cslch%raidEC.nSD && wmode[stripenum-fsp]>5){
                for(l=tdcs;l<vraid.chunksize;l++,hbufindex++){
                    block_mult(&hbuf[block_offset(hbufindex)],raidEC.BMatrix[i][j],bm);
                    block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                }
                l=0;
                h=tdcs;
            }
        }
        else{
            l=0;
            h=todo;
        }
        for(;l<h;l++,bufindex++){
            block_mult(buf+block_offset(bufindex),raidEC.BMatrix[i][j],bm);
            block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
        }
    }
    command[commandsIndex].device = raidEC.device[(parity_disk+i)%raidEC.nRD];
    command[commandsIndex].request.sequence = commandsIndex;
    command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
```

```
            if(csfch==cslch)
                command[commandsIndex].request.offset = ofcs;
            else
                command[commandsIndex].request.offset = stripenum*vraid.chunksize;
            command[commandsIndex].request.length = todo;
            command[commandsIndex].data = hbcp;
            command[commandsIndex].success = VRAID_COMMAND_PENDING;
            commandsIndex++;
        }

        if(commandsIndex==commands_tosend)
            break;
        command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
        if(csfch%raidEC.nSD>0)
            for(j=0;j<raidEC.nGET;j++){
                if(raidEC.eqFD[j]>(csfch-1)%raidEC.nSD)
                    break;
            }
        else
            j=0;
        bufindex=bufSize;
        for(i=csfch%raidEC.nSD;i<=cslch%raidEC.nSD;i++,bufindex+=todo){
            if(csfch==cslch)
                todo=tdcs;
            else if(i==csfch%raidEC.nSD)
                todo=vraid.chunksize-ofcs%vraid.chunksize;
            else if(i==cslch%vraid.chunksize)
                todo=tdcs;
            else
                todo=vraid.chunksize;
            if(j<raidEC.nGET && i==raidEC.eqFD[j]){
                j++;
                continue;
            }
            command[commandsIndex].device = raidEC.device[(stripenum*raidEC.nSD+i)%raidEC.nRD];
            command[commandsIndex].request.sequence = commandsIndex;
            command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
            if(i==csfch%raidEC.nSD)
                command[commandsIndex].request.offset = ofcs;
            else
                command[commandsIndex].request.offset = stripenum*vraid.chunksize;
            command[commandsIndex].request.length = todo;
            command[commandsIndex].data = buf+block_offset(bufindex);
            command[commandsIndex].success = VRAID_COMMAND_PENDING;
            commandsIndex++;
        }
        break;
    ⋮
}
```

Η δεύτερη περίπτωση, η οποία διακρίνεται από τους βασικούς ελέγχους που παρουσιάστηκαν προηγουμένως, είναι η περίπτωση κατά την οποία τα chunk προς εγγραφή του δεδομένου stripe είναι όλα διαθέσιμα και είτε τα υπόλοιπα chunk των δίσκων δεδομένων δεν είναι όλα διαθέσιμα, είτε είναι περισσότερα από τα πρώτα. Στη συγκεκριμένη περίπτωση, ο υπολογισμός των ενημερωμένων parities γίνεται χρησιμοποιώντας την πληροφορία που υπάρχει ήδη στα chunk προς εγγραφή, σε συνδυασμό με τη νέα πληροφορία προς εγγραφή, όπως αναφέρεται και στην ενότητα 3.2.2 και έχει αναλυθεί εκτενέστερα σε προηγούμενες ενότητες. Με τον τρόπο αυτό, επιτυγχάνονται οι λιγότερο δυνατές κλήσεις ανάγνωσης προς τους δίσκους του συστήματος, αφού τα chunk προς εγγραφή είναι λιγότερα σε αριθμό σε σχέση με τα υπόλοιπα chunk του stripe. Επιπρόσθετα, στην περίπτωση που κάποιο από τα chunk προς εγγραφή (το αρχικό ή το τελευταίο) δεν είναι ολόκληρο προς εγγραφή, τότε διαβάζονται μόνο τα απαραίτητα sectors. Στο σημείο αυτό πρέπει να αναφερθεί ότι, στα πλαίσια αυτής της μεθοδολογίας και κατά τη διάρκεια αυτής, κλειδώνονται μόνο τα chunk προς εγγραφή (τα οποία και διαβάζονται) και τα chunk των parities. Παρακάτω παρουσιάζεται ο κώδικα που υλοποιεί τη διαχείριση της περίπτωσης αυτής.

```
else if(wd_ok){
   wmode[stripenum-fsp]=3;
   create_eqFD(stripenum);
   commands_tosend+=cslch-csfch+1;
   command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
   for(i=csfch;i<=cslch;i++){
      command[commandsIndex].device = raidEC.device[i%raidEC.nRD];
      command[commandsIndex].request.sequence = commandsIndex;
      command[commandsIndex].request.command = VRAID_COMMAND_RLOCK;
      if(i==csfch){
         command[commandsIndex].request.offset = ofcs;
         if(csfch==cslch){
            command[commandsIndex].request.length = tdcs;
            hbufSize+=tdcs;
            hbuf=(unsigned char *) malloc(sizeof(unsigned char)*tdcs*vraid.sectorsize);
         }
         else{
            command[commandsIndex].request.length = vraid.chunksize-ofcs%vraid.chunksize;
            hbufSize+=vraid.chunksize-ofcs%vraid.chunksize;
            hbuf=(unsigned char *) malloc(sizeof(unsigned char)*(vraid.chunksize
                                                  -ofcs%vraid.chunksize)*vraid.sectorsize);
         }
      }
      else if(i==cslch){
         command[commandsIndex].request.offset = stripenum*vraid.chunksize;
         command[commandsIndex].request.length = tdcs;
         hbufSize+=tdcs;
         hbuf=(unsigned char *) malloc(sizeof(unsigned char)*tdcs*vraid.sectorsize);
      }
      else{
         command[commandsIndex].request.offset = stripenum*vraid.chunksize;
```

```
        command[commandsIndex].request.length = vraid.chunksize;
        hbufSize+=vraid.chunksize;
        hbuf=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
      }
      command[commandsIndex].data = hbuf;
      command[commandsIndex].success = VRAID_COMMAND_PENDING;
      hbufindex=hbufSize;
      commandsIndex++;
    }
    commands_tosend+=raidEC.nCsD-raidEC.nFD+raidEC.nGET;
    command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
    for(j=raidEC.nGET,i=0;i<raidEC.nCsD;i++){
      if(j<raidEC.nFD && i==raidEC.eqFD[j]){
        j++;
        continue;
      }
      command[commandsIndex].device = raidEC.device[(parity_disk+i)%raidEC.nRD];
      command[commandsIndex].request.sequence = commandsIndex;
      command[commandsIndex].request.command = VRAID_COMMAND_RLOCK;
      if(csfch<cslch){
        command[commandsIndex].request.offset = stripenum*vraid.chunksize;
        command[commandsIndex].request.length = vraid.chunksize;
        hbufSize+=vraid.chunksize;
        hbuf=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
      }
      else{
        command[commandsIndex].request.offset = ofcs;
        command[commandsIndex].request.length = tdcs;
        hbufSize+=tdcs;
        hbuf=(unsigned char *) malloc(sizeof(unsigned char)*tdcs*vraid.sectorsize);
      }
      command[commandsIndex].data = hbuf;
      command[commandsIndex].success = VRAID_COMMAND_PENDING;
      hbufindex=hbufSize;
      commandsIndex++;
    }
}
⋮
switch(wmode[stripenum-fsp]){
⋮
    case 3: commands_tosend+=raidEC.nCsD+raidEC.nGET-raidEC.nFD;
        commands_tosend+=cslch-csfch+1;
        command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command)
                                                   * commands_tosend);
        hbufSize=hbufindex;
        bufSize=bufindex;
        for(h=0,k=raidEC.nGET,i=0;i<raidEC.nCsD;i++){
          if(k<raidEC.nFD && i==raidEC.eqFD[k]){
            k++;
            continue;
          }
          hbcp=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
          for(l=0;l<vraid.chunksize;l++)
            block_init(&hbcp[block_offset(l)],0);
```

```
        bufindex=bufSize;
        hbufindex=hbufSize;
        for(j=csfch;j<=cslch;j++){
          if(j==csfch){
            l=ofcs%vraid.chunksize;
            if(csfch==cslch)
              todo=tdcs;
            else
              todo=vraid.chunksize-ofcs%vraid.chunksize;
          }
          else{
            l=0;
            if(j==cslch)
              todo=tdcs;
            else
              todo=vraid.chunksize;
          }
          for(;l<todo;l++,bufindex++,hbufindex++){
            block_mult(buf+block_offset(bufindex),raidEC.BMatrix[i][j%raidEC.nSD],bm);
            block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
            block_mult(&hbuf[hbufindex],raidEC.BMatrix[i][j%raidEC.nSD],bm);
            block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
          }
        }
        if(csfch==cslch){
          l=ofcs%vraid.chunksize;
          todo=tdcs;
        }
        else{
          l=0;
          todo=vraid.chunksize;
        }
        for(hbufindex+=h;l<todo;l++,hbufindex++)
          block_block_add(&hbcp[block_offset(l)],&hbuf[block_offset(hbufindex)],&hbcp[block_offset(l)]);
        command[commandsIndex].device = raidEC.device[(parity_disk+i)%raidEC.nRD];
        command[commandsIndex].request.sequence = commandsIndex;
        command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
        command[commandsIndex].request.offset = stripenum*vraid.chunksize+l;
        command[commandsIndex].request.length = todo;
        command[commandsIndex].data = hbcp+block_offset(l);
        command[commandsIndex].success = VRAID_COMMAND_PENDING;
        commandsIndex++;
        h+=todo;
      }
      bufindex=bufSize;
      for(i=csfch;i<=cslch;i++){
        if(i==csfch){
          l=ofcs%vraid.chunksize;
          if(csfch==cslch)
            todo=tdcs;
          else
            todo=vraid.chunksize-ofcs%vraid.chunksize;
        }
        else{
          l=0;
```

```
                if(i==cslch)
                    todo=tdcs;
                else
                    todo=vraid.chunksize;
            }
            command[commandsIndex].device = raidEC.device[i%raidEC.nRD];
            command[commandsIndex].request.sequence = commandsIndex;
            command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
            command[commandsIndex].request.offset = stripenum*vraid.chunksize+l;
            command[commandsIndex].request.length = todo;
            command[commandsIndex].data = buf+block_offset(bufindex);
            command[commandsIndex].success = VRAID_COMMAND_PENDING;
            commandsIndex++;
            bufindex+=todo;
        }
        break;
    ⋮
}
```

Τέλος, διακρίνεται η περίπτωση, που για το δεδομένο stripe υπάρχουν μη διαθέσιμα chunks, τόσο από το σύνολο αυτών που είναι προς εγγραφή, όσο και από εκείνα που δεν είναι προς εγγραφή. Στην περίπτωση αυτή διαβάζονται όλα τα διαθέσιμα chunks, δεδομένων και parities, και στη συνέχεια, χρησιμοποιώντας της μεθόδους **calPariy** και **GausElim**, οι οποίες αναπτύχθηκαν προηγουμένως, συμβάλλουν στην υλοποίηση των μεθόδων της κωδικοποίησης Reed-Solomon και Gauss, υπολογίζοντας τα μη διαθέσιμα chunks προς εγγραφή. Με τον τρόπο αυτό, πλέον, είναι διαθέσιμη όλη η υπάρχουσα πληροφορία των chunks προς εγγραφή, όπως και η πληροφορία των parities chunks που είναι διαθέσιμη. Έτσι, χρησιμοποιώντας τη μεθοδολογία που αναπτύχθηκε στην ενότητα 3.2.2, μέσω της παραπάνω πληροφορίας και της πληροφορίας προς εγγραφή, υπολογίζονται οι νέες τιμές για τα parity chunks που είναι διαθέσιμα. Τα chunks που κλειδώνονται καθόλη τη διάρκεια της διαδικασίας είναι τα διαθέσιμα chunks δεδομένων προς εγγραφή και τα parity chunks, τα οποία και χρησιμοποιούνται για τον υπολογισμό των νέων parities. Στη συνέχεια, παρουσιάζεται ο κώδικας που υλοποιεί αυτή την υποπερίπτωση της διαδικασίας write.

```
else{
   wmode[stripenum-fsp]=4;
   create_eqFD(stripenum);
   commands_tosend+=raidEC.nRD-raidEC.nFD;
   command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command) * commands_tosend);
   if(csfch<cslch){
      hbufSize+=raidEC.nSD*vraid.chunksize;
   }
   else {
      hbufSize+=raidEC.nSD*tdcs;
   }
   for(j=0,i=0;i<raidEC.nSD;i++){
      if(j<raidEC.nGET && i==raidEC.eqFD[j]){
         j++;
         continue;
      }
      command[commandsIndex].device = raidEC.device[(stripenum*raidEC.nSD+i)%raidEC.nRD];
      command[commandsIndex].request.sequence = commandsIndex;
      if(i>=csfch%raidEC.nSD && i<=cslch%raidEC.nSD)
         command[commandsIndex].request.command = VRAID_COMMAND_RLOCK;
      else
         command[commandsIndex].request.command = VRAID_COMMAND_READ;
      if(csfch<cslch){
         command[commandsIndex].request.offset = stripenum*vraid.chunksize;
         command[commandsIndex].request.length = vraid.chunksize;
         hbuf=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
         hbufindex+=vraid.chunksize;
      }
      else{
         command[commandsIndex].request.offset = ofcs;
         command[commandsIndex].request.length = tdcs;
         hbuf=(unsigned char *) malloc(sizeof(unsigned char)*tdcs*vraid.sectorsize);
         hbufindex+=tdcs;
      }
      command[commandsIndex].data = hbuf;
      command[commandsIndex].success = VRAID_COMMAND_PENDING;
      commandsIndex++;
   }
   for(k=0,i=0;i<raidEC.nCsD;i++){
      if(j<raidEC.nFD && i==raidEC.eqFD[j]){
         j++;
         continue;
      }
      command[commandsIndex].device = raidEC.device[(parity_disk+i)%raidEC.nRD];
      command[commandsIndex].request.sequence = commandsIndex;
      if(k<raidEC.nGET){
         command[commandsIndex].request.command = VRAID_COMMAND_RLOCK;
         if(csfch<cslch){
            command[commandsIndex].request.offset = stripenum*vraid.chunksize;
            command[commandsIndex].request.length = vraid.chunksize;
            hbuf=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
            hbufindex+=vraid.chunksize;
         }
         else{
            command[commandsIndex].request.offset = ofcs;
```

```
            command[commandsIndex].request.length = tdcs;
            hbuf=(unsigned char *) malloc(sizeof(unsigned char)*tdcs*vraid.sectorsize);
            hbufindex+=tdcs;
        }
        command[commandsIndex].data = hbuf;
        k++;
    }
    else{
        command[commandsIndex].request.command = VRAID_COMMAND_LOCK;
        if(csfch<cslch){
            command[commandsIndex].request.offset = stripenum*vraid.chunksize;
            command[commandsIndex].request.length = vraid.chunksize;
        }
        else{
            command[commandsIndex].request.offset = ofcs;
            command[commandsIndex].request.length = tdcs;
        }
        command[commandsIndex].data = NULL;
    }
    command[commandsIndex].success = VRAID_COMMAND_PENDING;
    commandsIndex++;
  }
}
⋮
switch(wmode[stripenum-fsp]){
  ⋮
  case 4: if(csfch==cslch){
            todo=tdcs;
            i=j=csfch%raidEC.nSD;
        }
        else{
            todo=vraid.chunksize;
            if(ofcs%vraid.chunksize)
                i=(csfch+1)%raidEC.nSD;
            else
                i=csfch%raidEC.nSD;
            if(tdcs<vraid.chunksize)
                j=(cslch-1)%raidEC.nSD;
            else
                j=cslch%raidEC.nSD;
        }
        sm=(block **) malloc(sizeof(block *)*raidEC.nGET);
        for(k=0;k<raidEC.nGET;k++){
            sm[k]=(block *) malloc(sizeof(block)*todo);
            for(s=0;s<todo;s++)
                sm[k][s]=(block) malloc(sizeof(unsigned char)*vraid.sectorsize);
        }
        createGETable();
        calParity(hbuf,0,todo,raidEC.nRD,1,sm);
        s=r=GausElim(sm,i,j,todo,0);
        commands_tosend+=raidEC.nCsD+raidEC.nGET-raidEC.nFD;
        command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command)
                                                * commands_tosend);
        hbufSize=hbufindex;
```

```
    bufSize=bufindex;
    for(k=raidEC.nGET,i=0;i<raidEC.nCsD;i++){
        if(k<raidEC.nFD && i==raidEC.eqFD[k]){
            k++;
            continue;
        }
        hbcp=(unsigned char *) malloc(sizeof(unsigned char)*vraid.chunksize*vraid.sectorsize);
        for(l=0;l<todo;l++)
            block_init(&hbcp[block_offset(l)],0);
        hbufindex=hbufSize;
        bufindex=bufSize;
        for(s=r,f=0,j=0;j<raidEC.nSD;j++){
            if(j>=csfch%raidEC.nSD && j<=cslch%raidEC.nSD){

                if(csfch==cslch){
                    l=0;
                    h=tdcs;
                }
                else if(j==csfch%raidEC.nSD && ofcs%vraid.chunksize>0){
                    if(f<raidEC.nGET && j==raidEC.eqFD[f]){
                        for(l=0;l<ofcs%vraid.chunksize;l++){
                            block_mult(sm[s][block_offset(l)],raidEC.BMatrix[i][j],bm);
                            block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                        }
                        s++;
                    }
                    else{
                        for(l=0;l<ofcs%vraid.chunksize;l++,hbufindex++){
                            block_mult(&hbuf[block_offset(hbufindex)],raidEC.BMatrix[i][j],bm);
                            block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                        }
                    }
                    h=vraid.chunksize;
                }
                else{
                    l=0;
                    if(j==cslch%raidEC.nSD)
                        h=tdcs;
                    else
                        h=vraid.chunksize;
                }
                for(;l<h;l++,bufindex++){
                    block_mult(buf+block_offset(bufindex),raidEC.BMatrix[i][j],bm);
                    block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                    if(f>=raidEC.nGET || i<raidEC.eqFD[f])
                        hbufindex++;
                }
                if(csfch<cslch && l<vraid.chunksize){
                    if(f<raidEC.nGET && j==raidEC.eqFD[f]){
                        for(;l<vraid.chunksize;l++){
                            block_mult(sm[s][block_offset(l)],raidEC.BMatrix[i][j],bm);
                            block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                        }
                        s++;
                    }
```

```
                else{
                    for(;l<vraid.chunksize;l++,hbufindex++){
                        block_mult(&hbuf[block_offset(hbufindex)],raidEC.BMatrix[i][j],bm);
                        block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                    }
                }
            }

            if(f<raidEC.nGET && j==raidEC.eqFD[f])
                f++;
            else
                commands_tosend++;
            continue;
        }
        else{
            if(f<raidEC.nGET && j==raidEC.eqFD[f]){
                for(l=0;l<todo;l++){
                    block_mult(sm[s][block_offset(l)],raidEC.BMatrix[i][j],bm);
                    block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                }
                f++;
                s++;
            }
            else{
                for(l=0;l<todo;l++,hbufindex++){
                    block_mult(&hbuf[block_offset(hbufindex)],raidEC.BMatrix[i][j],bm);
                    block_block_add(&hbcp[block_offset(l)],bm,&hbcp[block_offset(l)]);
                }
            }
        }
    }
    command[commandsIndex].device = raidEC.device[(parity_disk+i)%raidEC.nRD];
    command[commandsIndex].request.sequence = commandsIndex;
    command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
    if(csfch==cslch)
        command[commandsIndex].request.offset = ofcs;
    else
        command[commandsIndex].request.offset = stripenum*vraid.chunksize;
    command[commandsIndex].request.length = todo;
    command[commandsIndex].data = hbcp;
    command[commandsIndex].success = VRAID_COMMAND_PENDING;
    commandsIndex++;
}
if(commandsIndex==commands_tosend)
    break;
command = (struct raidEC_command *)realloc(command,sizeof(struct raidEC_command)
                                            * commands_tosend);
if(csfch%raidEC.nSD>0)
    for(j=0;j<raidEC.nGET;j++){
        if(raidEC.eqFD[j]>(csfch-1)%raidEC.nSD)
            break;
    }
else
    j=0;
bufindex=bufSize;
```

```
        for(i=csfch%raidEC.nSD;i<=cslch%raidEC.nSD;i++,bufindex+=todo){
          if(csfch==cslch)
            todo=tdcs;
          else if(i==csfch%raidEC.nSD)
            todo=vraid.chunksize-ofcs%vraid.chunksize;
          else if(i==cslch%vraid.chunksize)
            todo=tdcs;
          else
            todo=vraid.chunksize;
          if(j<raidEC.nGET && i==raidEC.eqFD[j]){
            j++;
            continue;
          }
          command[commandsIndex].device = raidEC.device[(stripenum*raidEC.nSD+i)%raidEC.nRD];
          command[commandsIndex].request.sequence = commandsIndex;
          command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
          if(i==csfch%raidEC.nSD)
            command[commandsIndex].request.offset = ofcs;
          else
            command[commandsIndex].request.offset = stripenum*vraid.chunksize;
          command[commandsIndex].request.length = todo;
          command[commandsIndex].data = buf+block_offset(bufindex);
          command[commandsIndex].success = VRAID_COMMAND_PENDING;
          commandsIndex++;
        }
        break;
}
```

4.3.4 Υλοποίηση της διεργασίας degrade

Η διεργασία degrade αποτελεί και αυτή μία από τις βασικότερες διεργασίες του RAID EC συστήματος και είναι υπεύθυνη για τη διαχείριση των εσφαλμένων δίσκων, που εμφανίζονται κατά τη λειτουργία του συστήματος. Αναλυτικότερα, βάσει και της περιγραφής που δόθηκε στην ενότητα 3.2.3, η διεργασία degrade καλείται όποτε εντοπίζεται εσφαλμένος δίσκος και έχει σκοπό τον εντοπισμό του δίσκου αυτού και τη διερεύνηση του συστήματος για το κατά πόσο συνεχίζει να είναι βιώσιμο. Στη συνέχεια, και εφόσον το σύστημα παραμένει βιώσιμο, η διεργασία ενημερώνει το σύστημα με το κατάλληλα status και προετοιμάζει την αποκατάσταση του.

Η μεθοδολογία, την οποία ακολουθεί η παρούσα υλοποίηση της διεργασίας για την επίτευξη των παραπάνω, είναι η ακόλουθη. Η διεργασία αρχικά διατρέχει τους δίσκους του συστήματος και εντοπίζει αυτούς που είναι εσφαλμένοι. Στη συνέχεια, για κάθε δίσκο που εντοπίζει ως εσφαλμένο, ελέγχετε αρχικά το κατά πόσο μπορεί το σύστημα να παραμείνει λειτουργικό. Αυτό επιτυγχάνεται ελέγχοντας για το αν με αυτόν τον εσφαλμένο δίσκο ξεπερνιέται ο αριθμός των parity δίσκων, ο οποίος είναι και ο αριθμός

των εσφαλμένων δίσκων που υποστηρίζει το RAID EC συστήματος. Εφόσον το σύστημα παραμένει βιώσιμο ενημερώνει το status του σε "Degraded" και μαρκάρει κατάλληλα το συγκεκριμένο δίσκο ως εσφαλμένο, ώστε το σύστημα να συνεχίζει να είναι λειτουργικό. Τέλος, εντοπίζει ένα διαθέσιμο spare δίσκο και τον μαρκάρει κατάλληλα, ώστε να χρησιμοποιηθεί στην αποκατάσταση τόσο της πληροφορίας που περιείχε ο εσφαλμένος δίσκος, όσο και του ίδιου του δίσκου. Στην περίπτωση που δεν υπάρχουν στο σύστημα διαθέσιμοι spare δίσκοι η διεργασία επιστρέφει κατάλληλο μήνυμα. Στη συνέχεια, παρατίθεται ο σχετικός κώδικας που υλοποιεί τη διεργασία degrade.

```
int raidEC_degrade()
{
   int i,j,k,error,sperror = 0;
   int failure_risk = 0;
   int fdf = 0;
   error = pthread_mutex_lock(&raidEC.mutex);
   if(raidEC.nFD > raidEC.nCsD){
      vraid.raidstatus = VRAID_STATUS_FAILED;
      return -1;
   }
   vraid.raidstatus=VRAID_STATUS_DEGRADED;
   for(i=0;i<raidEC.nRD;i++){
      if(raidEC.nFD == raidEC.nCsD)
         failure_risk = 1;
      if(raidEC.device[i]->status == VRAID_DEVICE_ERROR){
         for(j=0,fdf=0;j<raidEC.nFD;j++){
            if(raidEC.failedDisks[j].disk==i){
               fdf=1;
               break;
            }
            if(raidEC.failedDisks[j].disk>i)
               break;
         }
         if(failure_risk == 1 && fdf == 0){
            vraid.raidstatus = VRAID_STATUS_FAILED;
            raidEC.nFD++;
            return -1;
         }
         for(k=raidEC.nFD;fdf==0 && k>j;k--){
            raidEC.failedDisks[k].disk=raidEC.failedDisks[k-1].disk;
            raidEC.failedDisks[k].stripe=raidEC.failedDisks[k-1].stripe;
         }
         raidEC.failedDisks[j].stripe =-1;
         if(fdf==0){
            raidEC.failedDisks[j].disk=i;
            raidEC.nFD++;
         }
         if(raidEC.spareleft<=0)
            sperror=VRAID_ERROR_NOSPARELEFT;
         else {
```

```
            for(j=0;j<vraid.sparenum;j++){
               if(vraid.spare[j].status == VRAID_DEVICE_OFFLINE){
                  raidEC.device[i]=&(vraid.spare[j]);
                  raidEC.device[i]->status = VRAID_DEVICE_REBUILDING;
                  raidEC.spareleft--;
                  break;
               }
            }
         }
      }
   }
   error=pthread_mutex_unlock(&raidEC.mutex);
   if(error==0)
      error=sperror;
   return error;
}
```

4.3.5 Υλοποίηση της διεργασίας recover

Η διεργασία recover ανήκει και αυτή στις τέσσερις σημαντικότερες διεργασίες του RAID EC συστήματος και είναι υπεύθυνη για την αποκατάσταση της πληροφορίας των χαμένων δίσκων του συστήματος σε νέους, με στόχο την επαναφορά τους RAID EC συστήματος σε normal mode λειτουργία. Η διαδικασία όμως αυτή πρέπει να γίνεται δυναμικά, ενώ το σύστημα συνεχίζει να λειτουργεί σε degraded mode. Ακολουθώντας τον αλγόριθμο που παρουσιάστηκε στην ενότητα 3.2.4, η υλοποίηση της διεργασίας γίνεται εφαρμόζοντας την παρακάτω μεθοδολογία.

Όπως περιγράφηκε και στην ενότητα 3.2.4, η διαδικασία της αποκατάστασης της χαμένης πληροφορίας διεξάγεται ανά stripe. Έτσι, η διεργασία για κάθε stripe του συστήματος εντοπίζει, αρχικά, ποιοι δίσκοι του δεδομένου stripe είναι δίσκοι δεδομένων και ποιοι parity. Στη συνέχεια, εντοπίζει ποιοι δίσκοι είναι εσφαλμένοι και ποιοι όχι (για το συγκεκριμένο stripe) και υπολογίζει αντίστοιχα τον αριθμό των εσφαλμένων δίσκων δεδομένων και parity. Έχοντας την παραπάνω πληροφορία, η διεργασία διαβάζει και κλειδώνει τα τμήματα όλων των διαθέσιμων δίσκων δεδομένων για το αντίστοιχο stripe. Αντίστοιχα, διαβάζει και κλειδώνει τα τμήματα των nsf πρώτων διαθέσιμων δίσκων parity, όπου nsf ο αριθμός των εσφαλμένων δίσκων δεδομένων για το δεδομένο stripe. Για τους υπόλοιπους διαθέσιμους δίσκους parity, η διεργασία μόνο τους κλειδώνει. Η απαίτηση για κλείδωμα των διαθέσιμων δίσκων του stripe, όπως έχει αναφερθεί σε αρκετά σημεία της εργασίας αυτής, οφείλεται στην παραλληλία των διεργασιών και, στη συγκεκριμένη περίπτωση, στη δυνατότητα του συστήματος να διεξάγει παράλληλα τη

διαδικασία recover και την ικανοποίηση αιτημάτων εγγραφής, καθώς εξασφαλίζει τη μη αλλοίωση της σχετικής πληροφορίας.

Έπειτα, χρησιμοποιώντας τις μεθόδους **createGETable**, **calParity** και **GausElim** ανακτάται η πληροφορία των εσφαλμένων δίσκων δεδομένων του συγκεκριμένου stripe και χρησιμοποιώντας την πληροφορία αυτή και την πληροφορία των διαθέσιμων δίσκων δεδομένων, οι οποίοι αναγνώσθηκαν στο προηγούμενο βήμα της διεργασίας, υπολογίζεται η πληροφορία parity για τους αντίστοιχους εσφαλμένους δίσκους parity του stripe.

Τέλος, η διεργασία αποθηκεύει την ανακτώμενη πληροφορία στους νέους δίσκους που αντικαταστούν τους εσφαλμένους του συστήματος RAID EC. Επιπρόσθετα, ξεκλειδώνει όλους τους διαθέσιμους δίσκους του stripe που είχε κλειδώσει στο πρώτο της βήμα. Ακολουθεί ο κώδικας που υλοποιεί την παραπάνω μεθοδολογία.

```
for(stripenum=0;stripenum<vraid.mediasize/vraid.chunksize && vraid.raidstatus ==
                              VRAID_STATUS_RECOVERING;stripenum++){
  parity_disk=((stripenum+1)*raidEC.nSD)%raidEC.nRD;
  data_disk=(stripenum*raidEC.nSD)%raidEC.nRD;
  for(i=0,npf=0,nsf=0;i<raidEC.nRD;i++){
    j=(data_disk+i)%raidEC.nRD;
    check_disks[i]=sFD(j,j,stripenum);
    if(check_disks[i]==0){
      if(i<raidEC.nSD)
        nsf++;
      else
        npf++;
    }
  }
  for(i=0,j=0,commandsIndex=0,hbufindex=0;i<raidEC.nRD;i++){
    if(check_disks[i]==1){
      command[commandsIndex].device = raidEC.device[(data_disk+i)%raidEC.nRD];
      command[commandsIndex].request.sequence = commandsIndex;
      command[commandsIndex].request.offset = stripenum*vraid.chunksize;
      command[commandsIndex].request.length = vraid.chunksize;
      command[commandsIndex].success = VRAID_COMMAND_PENDING;
      if(i<raidEC.nSD){
        command[commandsIndex].request.command = VRAID_COMMAND_RLOCK;
        command[commandsIndex].data = hbuf+block_offset(hbufindex);
        hbufindex+=vraid.chunksize;
      }
      else if(j<nsf){
        command[commandsIndex].request.command = VRAID_COMMAND_RLOCK;
        command[commandsIndex].data = hbuf+block_offset(hbufindex);
        hbufindex+=vraid.chunksize;
        j++;
      }
      else {
        command[commandsIndex].request.command = VRAID_COMMAND_LOCK;
        command[commandsIndex].data = NULL;
```

```
        commandsIndex++;
    }
}

//send commands and verify that all the locks take place
do {
    locks_taken = 1;
    for(i=0;i<commandsIndex;i++){
        pthread_create(&th[i],NULL,&excCommand,&command[i]);
    }
    for(i=0;i<commandsIndex;i++)
        pthread_join(th[i],NULL);
    for(i=0;i<commandsIndex;i++){
        if(command[i].success == VRAID_ERROR_DISKIO){
            error = VRAID_ERROR_DISKIO;
            goto free_structures;
        }
        if(command[i].success == VRAID_ERROR_LOCK){
            locks_taken=0;
            break;
        }
    }
    if(!locks_taken){
        commands_pending = 0;
        for (i = 0; i < commandsIndex; i++){
            if(command[i].success==VRAID_ERROR_NOERROR && (command[i].request.command ==
                VRAID_COMMAND_LOCK || command[i].request.command == VRAID_COMMAND_RLOCK)){
               command[i].precommand=command[i].request.command;
               command[i].request.command = VRAID_COMMAND_UNLOCK;
               command[i].success = VRAID_COMMAND_PENDING;
               pthread_create(&th[commands_pending],NULL,&excCommand,&command[i]);
               commands_pending++;
            }
        }
        do{
            commands_pending=0;
            for(i=0;i<commandsIndex;i++){
                if(command[i].success == VRAID_COMMAND_PENDING){
                    commands_pending=1;
                    break;
                }
            }
        }while(commands_pending>0);

        for(i=0;i<commandsIndex;i++){
            if(command[i].request.command == VRAID_COMMAND_UNLOCK)
                command[i].request.command = command[i].precommand;
            command[i].success = VRAID_COMMAND_PENDING;
        }
    }
} while (!locks_taken);

//calculate the Failed Store Disks
create_eqFD(stripenum);
createGETable();
```

```
calParity(hbuf,0,vraid.chunksize,0,raidEC.nSD,sm);
k=kge=GausElim(sm,0,raidEC.nSD-1,vraid.chunksize,1);
for(j=0,i=0;i<raidEC.nSD;i++){
   if(j<raidEC.nGET && i==raidEC.eqFD[j]){
      for(l=0;l<vraid.chunksize;l++,hbufindex++)
         block_assign(hbuf+block_offset(hbufindex),sm[k][l]);
      k++;
      j++;
   }
}
//Calculate the Failed Parity Disks given that all Store Disks are available
for(j=0;j<vraid.chunksize;j++,hbufindex++){
   for(k=0,l=raidEC.nSD;l<raidEC.nRD;l++){
      if(check_disks[l]==0){
         block_init(&hbuf[hbufindex],0);
         for(i=0,m=0;(i<raidEC.nSD);i++){
            if(check_disks[i]==1){
               block_mult(&hbuf[block_offset(j+m*vraid.chunksize)],raidEC.BMatrix[l-raidEC.nSD][i],bm);
               m++;
            }
            else {
               block_mult(sm[kge+i-m][j],raidEC.BMatrix[l-raidEC.nSD][i],bm);
            }
            block_block_add(&hbuf[block_offset(hbufindex+k*vraid.chunksize)],
                              bm,&hbuf[block_offset(hbufindex+k*vraid.chunksize)]);
         }
         k++;
      }
   }
}
//Write the data and unlock
for(i=0,j=0,m=0,commandsIndex=0;i<raidEC.nRD;i++){
   command[commandsIndex].device = raidEC.device[(data_disk+i)%raidEC.nRD];
   command[commandsIndex].request.sequence = commandsIndex;
   command[commandsIndex].request.offset = stripenum*vraid.chunksize;
   command[commandsIndex].request.length = vraid.chunksize;
   command[commandsIndex].success = VRAID_COMMAND_PENDING;
   if(check_disks[i]==1){
      command[commandsIndex].request.command = VRAID_COMMAND_UNLOCK;
      command[commandsIndex].data = NULL;
   }
   else if(check_disks[i]==0){
      command[commandsIndex].request.command = VRAID_COMMAND_WUNLOCK;
      if(i<raidEC.nSD){
         for(t=0;t<vraid.chunksize;t++)
            block_assign(&smbuf[block_offset(t)],sm[kge+j][t]);
         command[commandsIndex].data = smbuf;
         j++;
      }
      else {
         command[commandsIndex].data = hbuf+block_offset((raidEC.nSD+m)*vraid.chunksize);
         m++;
      }
   }
   commandsIndex++;
```

```
    }
    /*Send commands*/
    th=(pthread_t *) malloc(sizeof(pthread_t)*commandsIndex);
    for(i=0;i<commandsIndex;i++){
        pthread_create(&th[i],NULL,&excCommand,&command[i]);
    }
    for(i=0;i<commandsIndex;i++){
        j=pthread_join(th[i],NULL);
    }
    for(i=0;i<commandsIndex;i++){
        if(command[i].success == VRAID_ERROR_DISKIO){
            error=VRAID_ERROR_DISKIO;
            goto free_structures;
        }
    }
    //Set as recovered the specific part
    merror = pthread_mutex_lock(&raidEC.mutex);
    for(i=0;i<raidEC.nFD;i++){
        raidEC.failedDisks[i].stripe=stripenum;
    }
    merror=pthread_mutex_unlock(&raidEC.mutex);
}
merror = pthread_mutex_lock(&raidEC.mutex);
raidEC.nFD=0;
vraid.raidstatus = VRAID_STATUS_NORMAL;
merror=pthread_mutex_unlock(&raidEC.mutex);
```

Κεφάλαιο 5

Αξιολόγηση και Εφαρμογές

Σκοπός του κεφαλαίου αυτού είναι η αξιολόγηση του συστήματος RAID EC, το οποίο σχεδιάστηκε και υλοποιήθηκε στην παρούσα εργασία, όπως και η εξέταση πιθανών βελτιώσεων και επεκτάσεων του συστήματος. Τέλος, στο κεφάλαιο αυτό θα λάβει χώρα και μια αναφορά στις πιθανές εφαρμογές του συστήματος RAID EC.

Επομένως, στην πρώτη ενότητα του κεφαλαίου παρουσιάζονται μετρήσεις της απόδοσης του RAID EC συστήματος και ενός RAID 5. Στο σημείο αύτο πρέπει να επισημανθεί ότι το RAID 5, το οποίο θα χρησιμοποιηθεί στις μετρήσεις, είναι υλοποιημένο σε επίπεδο λειτουργικού, ενώ το RAID EC, όπως έχει αναφερθεί, σε επίπεδο λογισμικού (user). Το γεγονός αυτό και μόνο δίνει πλεονέκτημα στο RAID 5 από άποψη επιδόσεων χρόνου απόκρισης σε αιτήματα ανάγνωσης και εγραφής. Βάσει των αποτελεσμάτων των μετρήσεων αυτών θα γίνει και η αξιολόγηση του RAID EC, αναγνωρίζοντας τα πλεονεκτήματα και τα μειονεκτήματα του συστήματος.

Στη δεύτερη ενότητα του κεφαλαίου λαμβάνει χώρα μια αναφορά στις δυνατές επεκτάσεις του συστήματος RAID EC και στα πλεονεκτήματα/μειονεκτήματα αυτών. Επιπρόσθετα, αναφορά γίνεται και στις πιθανές εφαρμογές του συστήματος που σχεδιάστηκε και αναπτύχθηκε στην παρούσα εργασία.

5.1 Αξιολόγηση του συστήματος RAID EC

Στα πλαίσια της αξιολόγησης της υλοποίησης και του σχεδιασμού του συστήματος RAID EC έλαβαν χώρα ένας αριθμός μετρήσεων αναφορικά με τους χρόνους απόδοσης των διεργασιών read, write και recover συναρτήσει διαφόρων παραμέτρων του συστήματος, όπως ο αριθμός των δίσκων δεδομένων (N) και των δίσκων parity (M), το μέγεθος των δίσκων και ο όγκος των δεδομένων προς ανάγνωση και εγγραφή.

Επιπρόσθετα, για την απόκτηση ενός μέτρου σύγκρισης στα πλαίσια της αξιολόγησης των διεργασιών read και write, πραγματοποιήθηκαν μετρήσεις της απόκρισης των αντίστοιχων διεργασιών του συστήματος RAID 5. Ο λόγος επιλογής του RAID 5 ως μέτρο σύγκρισης είναι το γεγονός ότι, αυτή τη στιγμή, αποτελεί ένα πολύ δημοφιλές και ευρέως χρησιμοποιούμενο σύστημα RAID. Στο σημείο αυτό πρέπει να επισημανθεί ότι το RAID 5, που χρησιμοποιήθηκε στις μετρήσεις αυτές, έχει υλοποιηθεί σε επίπεδο λειτουργικού, το οποίο εξασφαλίζει καλύτερη απόδοση στο σύστημα, έχοντας τη δυνατότητα καλύτερης διαχείρισης τόσο της μνήμης, όσο και της υπολογιστικής ισχύς του υπολογιστικού συστήματος πάνω στο οποίο λειτουργεί.

Στο σημείο αυτό, θα πρέπει να επισημανθεί, ότι τα σχετικά πειράματα για την αξιολόγηση του RAID EC έγιναν σε περιβάλλον εικονικών μηχανών, εξαιτίας της ανάγκης για υποστήριξη μεγάλου αριθμού δίσκων, ευελιξία στην προσθήκη και αφαίρεση δίσκων και ελέγχου των κλήσεων για ανάγνωση και εγγραφή προς τη φυσική συσκευή. Επομένως, οι μετρήσεις που ακολουθούν αναφέρονται σε μία εικονική μηχανή με 24 εικονικούς δίσκους που χρησιμοποιούν ένα μόνο πραγματικό δίσκο. Αυτό το περιβάλλον δοκιμής επηρεάζει σημαντικά τους απόλυτους χρόνους πρόσβασης στο φυσικό μέσο, αλλά όχι τους σχετικούς. Ο χρόνος που χρειάζεται για να διαβαστεί ένα block είναι ο ίδιος για τις διάφορες περιπτώσεις των πειραμάτων. Έτσι, το εικονικό περιβάλλον που χρησιμοποιήθηκε δεν επηρεάζει τη σχετική συμπεριφορά του συστήματος, με αποτέλεσμα να μπορούν να εξαχθούν συμπεράσματα από τη σύγκριση των πειραμάτων με διαφορερικές παραμέτρους. Πιο συγκεκριμένα, χρησιμοποιήθηκε η πλατφόρμα Virtualization Xen έκδοση 3.4 σε λειτουργικό σύστημα Debian 5.0 GNU/Linux με πυρήνα έκδοσης 2.6.26.

Κατ' αυτόν τον τρόπο, αρχικά, έλαβαν χώρα οι παρακάτω μετρήσεις, εξετάζοντας την απόδοση των διεργασιών ανάγνωσης και εγγραφής του RAID EC συστήματος. Μετρήθηκε ο χρόνος απόκρισης σε seconds των read και write της υλοποίησης του Erasure – Code για αντίστοιχη ανάγνωση και εγγραφή αρχείων μεγέθους 50 MB και 100 MB για διάφορους συνδυασμούς των τιμών N και M. Τα αποτελέσματα των μετρήσεων αυτών παρουσιάζονται στον Πίνακα 2. Αντιστοίχως, στον Πίνακα 3 παρατίθενται οι χρόνοι των read και write του RAID-5 για τα αντίστοιχα αιτήματα. Το RAID-5 που

χρησιμοποιήθηκε για τις παρακάτω μετρήσεις αποτελείται από 10 δίσκους από τους οποίους, βάσει αρχιτεχτονικής, μόνο ο ένας χρησιμοποιείται για parity:

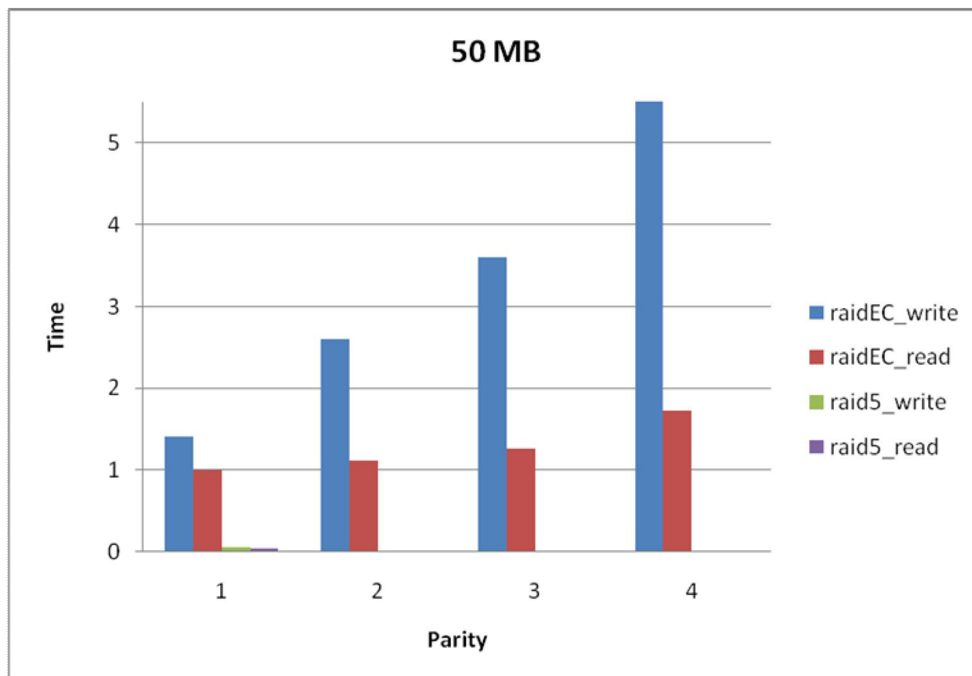
Πίνακας 2: Χρόνος απόκρισης των read και write για Erasure-Code

File Size	N+M Function	9+1	8+2	7+3	6+4
50M	Write	2.028567	3.732655	5.182489	8.093811
	Read	1.438703	1.605711	1.805328	2.485404
100M	Write	7.224851	11.227264	14.556846	21.48997
	Read	6.945352	6.577759	6.864224	9.611015

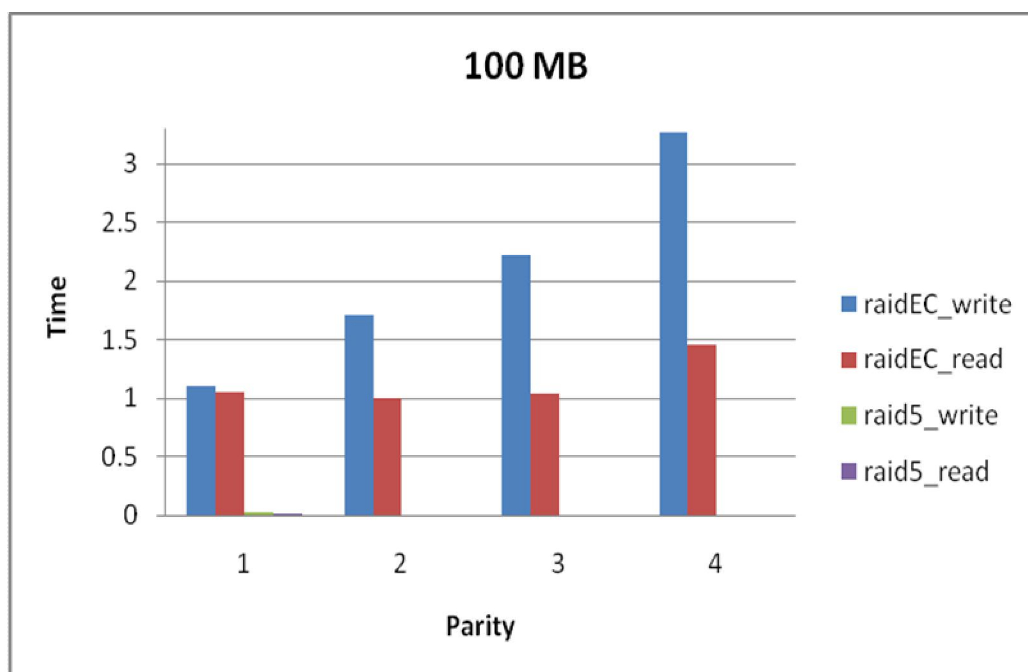
Πίνακας 3: Χρόνος απόκρισης των read και write για RAID-5

File Size	*N+M* *Function*	9+1
50M	write	0.084972
	read	0.058411
100M	write	0.210724
	read	0.063442

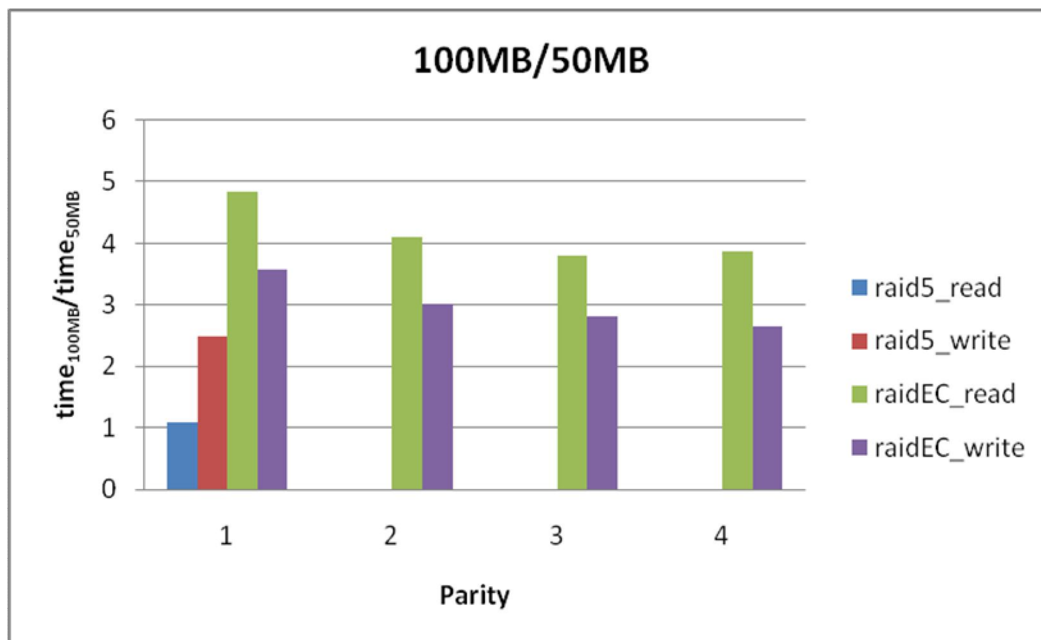
Σε συνέχεια των παραπάνω μετρήσεων δημιουργήθηκαν τα διαγράμματα που εμφανίζονται στα Σχήματα 26 και 27, με σκοπό να παρουσιαστεί ο τρόπος με τον οποίο μεταβάλλεται ο χρόνος απόκρισης του read και write σε συνάρτηση των τιμών του N και M για αρχεία διαφορετικού μεγέθους. Για την ακρίβεια, όλα τα διαγράμματα και οι μετρήσεις έχουν σταθερό αριθμό δίσκων που χρησιμοποιεί το RAID-EC και είναι ίσος με 10. Επομένως, μεταβάλοντας την τιμή του M (Parity), μεταβάλλεται αντίστοιχα και η τιμή του N. Το Σχήμα 28 κατασκευάστηκε για να παρουσιάσει με μεγαλύτερη ακρίβεια το ποσοστό με το οποίο αυξάνεται ο χρόνος απόκρισης των read και write για το RAID-EC και RAID-5, για αρχείο μεγέθους 100MB σε σχέση με ένα αρχείο των 50MB.



Σχήμα 26: Ο χρόνος απόκρισης (αναγόμενος στο μικρότερο χρόνο του Erasure-Code read) του read και write για Erasure-Code και RAID-5 σε αρχείο των 50 MB



Σχήμα 27: Ο χρόνος απόκρισης (αναγόμενος στο μικρότερο χρόνο του Erasure-Code read) του read και write για Erasure-Code και RAID-5 σε αρχείο των 100 MB



Σχήμα 28: Ο λόγος του χρόνου απόκρισης του read και write για Erasure-Code και RAID-5 για αρχείο 100MB προς αυτόν για αρχείο 50MB

Μελετώντας τα παραπάνω διαγράμματα, παρατηρούμε κατ' αρχάς ότι η διαδικασία του write είναι αρκετά πιο χρονοβόρα και επιπλέον, ότι επηρεάζεται περισσότερο από την αύξηση τόσο του αριθμού των δίσκων parity, όσο και από τον όγκο των μεταφερόμενων δεδομένων. Πιο συγκεκριμένα, παρατηρούμε ότι καθώς αυξάνεται ο αριθμός των δίσκων parity, ο χρόνος απόκρισης της write πολλαπλασιάζεται, ενώ ο αντίστοιχος χρόνος της read παραμένει σχεδόν σταθερός για δεδομένο όγκο πληροφορίας. Η συμπεριφορά αυτή εξηγείται από το γεγονός ότι κάθε φορά που εκτελείται η write πρέπει να ενημερώνει και τα αντίστοιχα parity του συστήματος. Αυτό έχει ως αποτέλεσμα, καθώς αυξάνεται ο αριθμός των δίσκων parity, να αυξάνεται τόσο ο όγκος των απαραίτητων υπολογισμών, όσο και ο όγκος της πληροφορίας που πρέπει να αποθηκεύεται στους δίσκους του συστήματος. Επιπρόσθετα, παρατηρείται ότι ο ρυθμός με τον οποίο αυξάνεται ο χρόνος απόκρισης των διεργασιών, καθώς αυξάνεται ο όγκος της μεταφερόμενης πληροφορίας, μειώνεται με την αύξηση του αριθμού των δίσκων parity. Η συμπεριφορά αυτή οφείλεται στην αρχιτεκτονική του RAID EC συστήματος και στην ανάγκη επιπρόσθετων αναγνώσεων στους δίσκους του συστήματος για τον υπολογισμό των parity, σε συνδυασμό με τη μεταβολή του αριθμού των δίσκων δεδομένων και δίσκων parity.

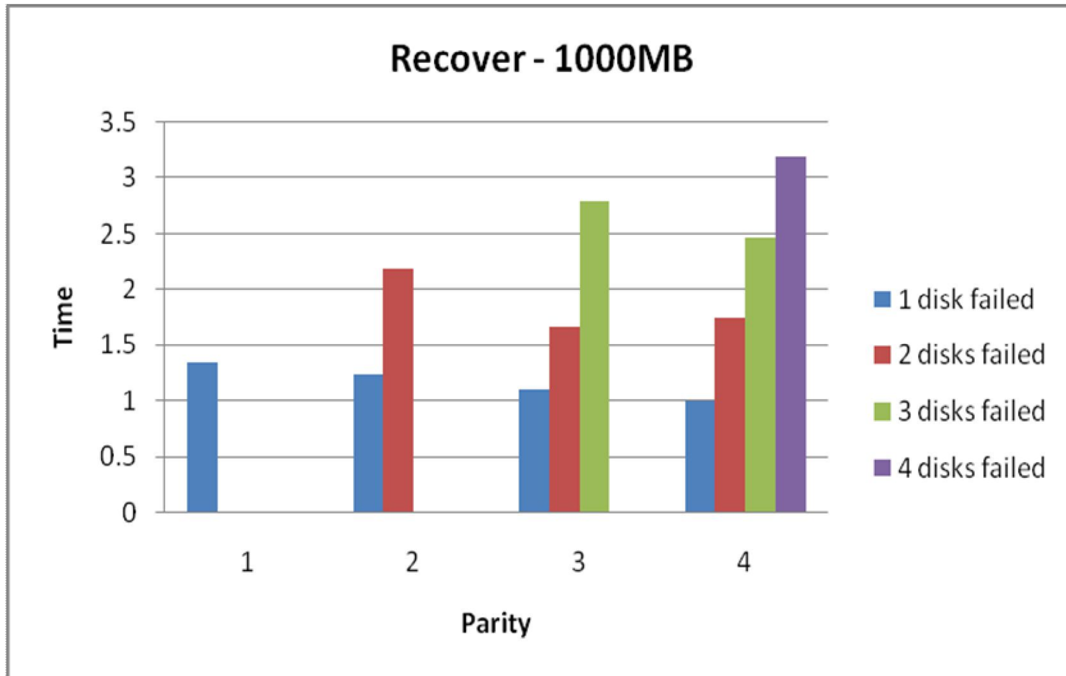
Στη συνέχεια, διεξάγεται ένας αριθμός μετρήσεων αναφορικά με το χρόνο απόκρισης της λειτουργίας του recover. Τα αποτελέσματα των μετρήσεων παρουσιάζονται στον Πίνακα

4. Οι μετρήσεις έγιναν σε συνάρτηση με τις τιμές των N και M, τον αριθμό των εσφαλμένων δίσκων και το μέγεθος των δίσκων, αφού το recover εκτελείται κάθε φορά για ολόκληρους τους δίσκους. Οι χρόνοι που εμφανίζονται στον Πίνακα 4 είναι μετρημένοι σε seconds.

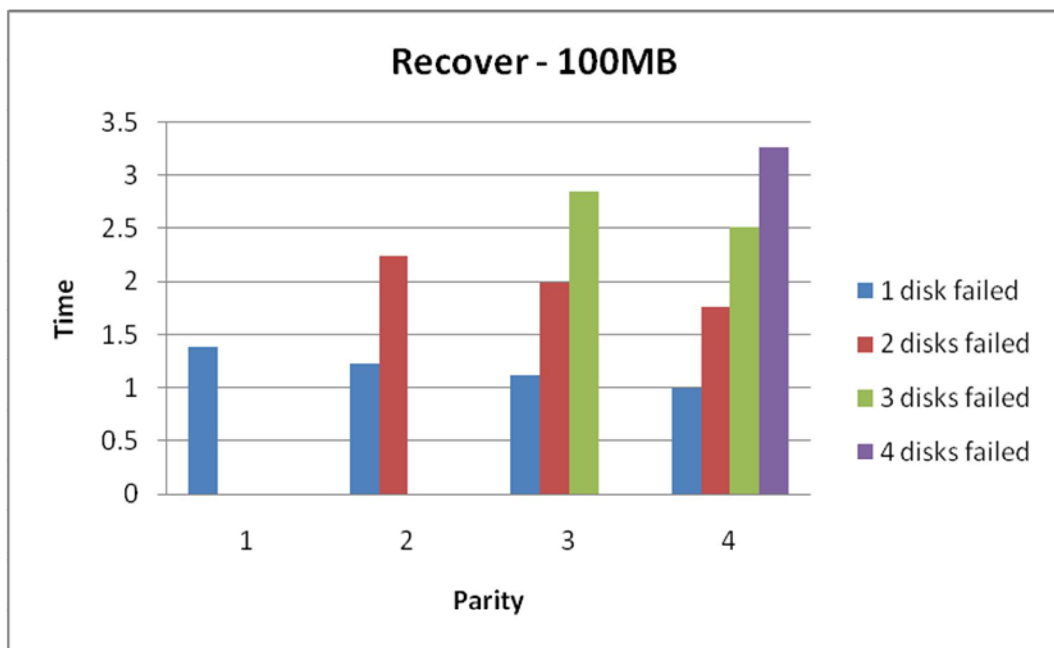
Πίνακας 4: Χρόνος recover για Erasure-Code

N+M		9+1	8+2		7+3			6+4			
Disk(s) Failed		1	1	2	1	2	3	1	2	3	4
Device Size	1000MB	273.9028	250.478	445.5381	224.1914	339.7707	567.69556	203.42599	353.9913	502.7439	646.9989
	100MB	27.25534	24.1531	44.43337	22.13203	39.37449	56.116575	19.745256	34.7865	49.58365	64.4809

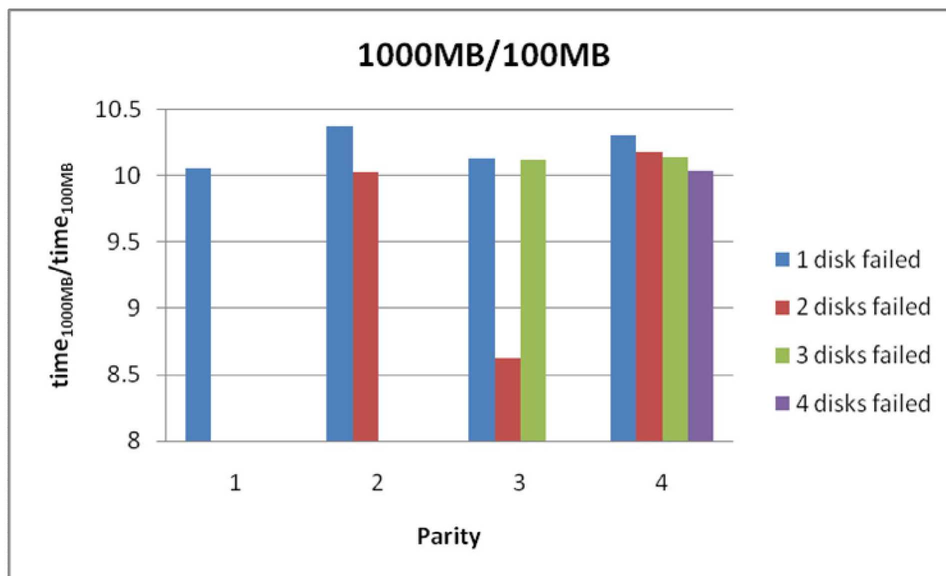
Χρησιμοποιώντας τις μετρήσεις που εμφανίζονται στον Πίνακα 4, κατασκευάστηκαν τα διαγράμματα των Σχημάτων 29, 30 και 31. Τα Σχήματα 29 και 30 απεικονίζουν τους χρόνους απόκρισης του recover για διάφορους συνδυασμούς των τιμών N και M και τους εσφαλμένους δίσκους για διαφορετικά μεγέθη των δίσκων, αναγόμενους στην μικρότερη τιμή των μετρήσεων του χρόνου απόκρισης για το κάθε μέγεθος δίσκου, ώστε να είναι περισσότερο ευδιάκριτη η διακύμανση των χρόνων αυτών. Στο Σχήμα 31 εμφανίζεται ο λόγος του χρόνου απόκρισης του recover για μέγεθος δίσκων 1000MB ανά περίπτωση, προς τους αντίστοιχους χρόνους για μέγεθος δίσκων 100MB.



Σχήμα 29: Ο χρόνος απόκρισης (αναγόμενος στο μικρότερο χρόνο) του recover για το Erasure-Code και δίσκους μεγέθους 1000 MB



Σχήμα 30: Ο χρόνος απόκρισης (αναγόμενος στο μικρότερο χρόνο) του recover για το Erasure-Code και δίσκους μεγέθους 100 MB



Σχήμα 31: Ο λόγος του χρόνου απόκρισης του recover για το Erasure-Code για μέγεθος δίσκων 1000MB προς αυτόν για δίσκους μεγέθους 100MB

Εξετάζοντας τις παραπάνω μετρήσεις και διαγράμματα, παρατηρούμε αρχικά, όπως είναι αναμενόμενο, να αυξάνεται ο χρόνος αποκατάστασης του συστήματος καθώς αυξάνεται ο αριθμός των εσφαλμένων δίσκων. Επιπρόσθετα, όμως, παρατηρείται ότι ο χρόνος αποκατάστασης για ένα δεδομένο αριθμό εσφαλμένων δίσκων μειώνεται, καθώς αυξάνεται ο αριθμός των parity δίσκων. Αυτό εξηγείται από το γεγονός ότι αυξάνοντας των αριθμό των parity δίσκων υπάρχει μεγαλύτερη πλεονάζουσα πληροφορία parity και ο υπολογισμός της χαμένης πληροφορίας γίνεται χρησιμοποιώντας πληροφορία που είναι αποθηκευμένη σε λιγότερους δίσκους, με αποτέλεσμα και τη διεξαγωγή λιγότερων κλήσεων στους δίσκους του συστήματος. Τέλος παρατηρείται ότι ο ρυθμός αύξησης του χρόνου απόκρισης της διαδικασίας recover, καθώς αυξάνεται το μέγεθος των δίσκων του συστήματος, δεν επηρεάζεται σημαντικά ούτε από το αριθμό των parity δίσκων, ούτε από τον αριθμό των εσφαλμένων δίσκων.

Γενικότερο συμπέρασμα των παραπάνω μετρήσεων που έλαβαν χώρα τόσο για το RAID EC, όσο και για το σύστημα RAID 5, είναι κατ' αρχάς ότι το RAID 5 είναι σαφώς πιο αποδοτικό από το RAID EC. Αυτό δικαιολογείται αρχικά από το γεγονός ότι το RAID 5 είναι υλοποιημένο σε επίπεδο λειτουργικού, ενώ το RAID EC σε επίπεδο user. Έπειτα, η παρούσα υλοποίηση του RAID EC εμφανίζει αρκετά προβλήματα με τη διαχείριση της μνήμης του συστήματος, γεγονός το οποίο απαιτεί μια επιπρόσθετη προσπάθεια βελτιστοποίησης της συμπεριφοράς αυτής. Εντούτοις, ένας από τους λόγους ύπαρξης του

παραπάνω προβλήματος είναι και αυτή καθ'αυτή η υλοποίηση του συστήματος σε επίπεδο λειτουργικού.

Ένα επιπρόσθετος λόγος της ύπαρξης διαφοράς απόδοσης των δύο συστημάτων είναι και η αρχιτεκτονική του RAID EC. Η τελευταία, όμως, αποτελεί και το μεγαλύτερο πλεονέκτημα του RAID EC απέναντι στις άλλες αρχιτεκτονικές RAID. Το RAID EC έχει τη δυνατότητα να δημιουργήσει συστήματα RAID με μεταβλητό αριθμό δίσκων δεδομένων και δίσκων parity, εξασφαλίζοντας μεγαλύτερη προστασία στην αποθηκευόμενη πληροφορία του συστήματος. Η ιδιότητα αυτή του RAID EC του χαρίζει την ικανότητα να σχηματίζει συστήματα από σχετικά μεγάλο αριθμό δίσκων, εξασφαλίζοντας παράλληλα υψηλού επιπέδου προστασία στην αποθηκευόμενη πληροφορία, αλλά και ικανοποιητική απόδοση.

5.2 Επεκτάσεις και Εφαρμογές

Βάσει της ανάλυσης που έλαβε χώρα στην προηγούμενη ενότητα, το κύριο αδύναμο σημείο της παρούσας υλοποίησης του συστήματος RAID EC αποτελούν οι χρόνοι εκτέλεσης των διαφόρων λειτουργιών του. Ο κυριότερος λόγος των επιδόσεων αυτών, όπως παρατηρήθηκε, είναι η διαχείριση της μνήμης του συστήματος και γενικότερα των πόρων αυτού.

Για την αντιμετώπιση του θέματος αυτού, αρχικά, θα μπορούσε να γίνει μια βελτιστοποίηση στο σχεδιασμό της υλοποίησης του συστήματος RAID EC ως προς τον τρόπο διαχείρισης της μνήμης και των threads της εφαρμογής. Μια όμως πιο ουσιαστική αντιμετώπιση της χαμηλής απόδοσης του συστήματος θα μπορούσε να επέλθει με μία από τις δύο παρακάτω επεκτάσεις. Η πρώτη από αυτές είναι ο σχεδιασμός και η υλοποίηση του RAID EC σε επίπεδο λειτουργικού. Εφαρμογές υλοποιημένες σε επίπεδο λειτουργικού έχουν αμεσότερο έλεγχο της μνήμης και της υπολογιστικής ισχύς, όπως και των υπόλοιπων πόρων του υπολογιστικού συστήματος στο οποίο δραστηριοποιούνται, σε σχέση με τις εφαρμογές επιπέδου χρήστη. Αυτό έχει ως αποτέλεσμα και τη μικρότερη σπατάλη μνήμης, όπως και την καλύτερη επικοινωνία με τα μέσα αποθήκευσης, επιτυγχάνοντας πολύ καλύτερους χρόνους από μία υλοποίηση σε επίπεδο χρήστη. Τα αρνητικά της υλοποίησης σε επίπεδο λειτουργικού είναι η αυξημένη περιπλοκότητα που πρέπει να έχει μια τέτοια υλοποίηση ώστε να μπορεί να χειριστεί τους πόρους του υπολογιστικού συστήματος και οι περιορισμένες δυνατότητες αποσφαλμάτωσης της εφαρμογής.

Μια δεύτερη επέκταση, η οποία θα μπορούσε να γίνει στο RAID EC, είναι η υλοποίηση του ως embedded σύστημα. Πιο συγκεκριμένα, θα μπορούσε να κατασκευαστεί μία

κάρτα, η οποία θα προγραμματιζόταν να υλοποιεί την αρχιτεκτονική του RAID EC, η οποία περιγράφηκε στην παρούσα εργασία, έχοντας τη δυνατότητα να εκτελέσει όλες τις απαιτούμενες λειτουργίες του συστήματος. Η υλοποίηση ως embedded σύστημα έχει όλα τα πλεονεκτήματα της εφαρμογής επιπέδου λειτουργικού (και μάλιστα επαυξημένα) και επιπρόσθετα διαθέτει δικούς της αφιερωμένους υπολογιστικούς πόρους, όπως μνήμη και υπολογιστική ισχύ. Τα χαρακτηριστικά αυτά εξασφαλίζουν εκπληκτικές επιδόσεις για το σύστημα RAID EC, οι οποίες θα φτάνουν και ίσως να ξεπερνούν τις επιδόσεις του RAID 5. Από την άλλη πλευρά, η κατασκευή ενός embedded συστήματος είναι αρκετά πιο περίπλοκη από μια εφαρμογή επιπέδου λειτουργικού, πόσο μάλλον από μία εφαρμογή επιπέδου χρήστη.

Το RAID EC σύστημα, παρόλα τα προβλήματα απόδοσης που παρατηρήθηκαν και τα οποία βελτιώνονται με τις επεκτάσεις που περιγράφηκαν στις παραπάνω παραγράφους, διαθέτει αρκετά πλεονεκτήματα ως αρχιτεκτονική. Όπως έχει ήδη περιγραφεί, η αρχιτεκτονική του RAID EC αποτελεί μια αρκετά ευέλικτη λύση για τα αποθηκευτικά συστήματα, καθώς είναι σε θέση να παραμετροποιηθεί ανάλογα με τις εκάστοτε απαιτήσεις ενός συστήματος για μεγαλύτερη ασφάλεια στα δεδομένα του ή για καλύτερες επιδόσεις. Σε κάθε περίπτωση, το RAID EC μπορεί να αποτελέσει τη χρυσή τομή μεταξύ της διασφάλισης των δεδομένων και των επιδόσεων, ιδιαιτέρως σε αποθηκευτικά συστήματα με αυξημένες απαιτήσεις σε αποθηκευτικό χώρο ή συστήματα που δέχονται μεγάλο αριθμό από διαδοχικά αιτήματα αναγνώσεων και εγγραφών. Κατά αυτόν τον τρόπο η αρχιτεκτονική του RAID EC θα μπορούσε να εφαρμοστεί τόσο σε ένα απλό υπολογιστικό σύστημα, όσο και για την κάλυψη των απαιτήσεων ενός Server που εξυπηρετεί ένα κατανεμημένο δίκτυο από προσωπικούς υπολογιστές. Σε αυτό συνηγορεί και η δυνατότητα της εφαρμογής του συστήματος RAID EC να υλοποιεί τις διεπαφές του με τα αποθηκευτικά μέσα μέσω δικτυού από network block devices, οι οποίοι είναι και ένα από τα αντικείμενα με τα οποία ασχολήθηκε και το σύστημα vRAID, το οποίο αναπτύχθηκε στο εργαστήριο του CSLab του ΕΜΠ και στο οποίο στηρίχθηκε και η παρούσα εργασία. Επιπρόσθετα, μια ακόμη εφαρμογή του RAID EC θα μπορούσε να είναι ένας Sever Web εφαρμογών, ο οποίος έχει απαιτήσεις τόσο για αυξημένο όγκο δεδομένων, όσο και για πολλές σειριακές κλήσεις ανάγνωσης και εγγραφής στα δεδομένα αυτού. Τέλος, μια ακόμη εφαρμογή για την αρχιτεκτονική του RAID EC, η οποία ξεφεύγει από τα πλαίσια των αποθηκευτικών συστημάτων, αποτελούν τα δίκτυα και η μεταφορά δεδομένων μέσω αυτών. Αναλυτικότερα, θα μπορούσε να χρησιμοποιηθεί η αρχιτεκτονική του RAID EC για τη διάσπαση ενός όγκου πληροφορίας σε n+m μικρότερα τμήματα (όπου m τα τμήματα που αναφέρονται σε parity πληροφορία), με σκοπό την αποστολή τους μέσω ενός αναξιόπιστου δικτύου χωρίς να απαιτείται απάντηση για την παραλαβή τους και τυχόν επανάληψη της αποστολής κάποιων από τα τμήματα αυτά. Στη συνέχεια ο παραλήπτης, λαμβάνοντας οποιαδήποτε n από τα n+m των υποτμημάτων της αρχικής πληροφορίας, θα μπορεί με ασφάλεια να ανακτήσει την αποστελλόμενη πληροφορία. Κατά αυτόν τον τρόπο η αρχιτεκτονική του

RAID EC κάνει εφικτή τη χρήση ενός αναξιόπιστου δικτύου για τη μεταφορά ενός όγκου πληροφορίας.

Βιβλιογραφία

1. Anvin H. P., 2004, «The mathematics of RAID-6»
2. Beachy J. A., et al., 2000, «ABSTRACT ALGEBRA: REVIEW PROBLEMS ON GROUPS AND GALOIS THEORY», Northern Illinois University, Waveland Press, Inc., Illinois
3. Carlet C., 1998, «More Correlation-Immune and Resilient Functions over Galois Fields and Galois Rings», GREYC, University de Caen and INRIA Project Codes, Domaine de Voluceau, BP 105, France
4. Chen P. M., et al., 1994, «RAID: High-Performance, Reliable Secondary Storage», ACM Computing Surveys, Vol 26, No. 2
5. Courtright W. V., et al., 1996, «A Structured Approach to Redundant Disk Array Implementation», Appears in the Proceedings of the International Computer Performance and Dependability Symposium (IPDS), Sept. 4-6, 1996, School of Computer Science Carnegie Mellon University Pittsburg, PA 15213
6. Δούδαλη Ι. Δ., 2006, «Αποδοτική Μεταφορά Δεδομένων σε Κατανεμημένα Μέσα Αποθήκευσης Υψηλών Επιδόσεων αξιοποιώντας Δικτυακές Τεχνολογίες ATA-Over-Ethernet και Myrinet», Διπλωματική εργασία για το τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου, τομέας Τεχνολογίας, Πληροφορικής και Υπολογιστών, Εργαστήριο Υπολογιστικών Συστημάτων, Αθήνα
7. ΕΠΙΣΕΥ-ΕΜΠ & ALTEC A.B.E.E., 2005, «Τεύχος 4 Εγχειρίδιο Υλοποίησης και Εγκατάστασης Λογισμικού Δημιουργίας Μεγάλου Αποθηκευτικού Χώρου», στα πλαίσια του έργου «Ανάπτυξη Συστοιχίας Υπολογιστών Υψηλών Επιδόσεων για Προηγμένους Εξυπηρετητές Ιστοσελίδων Διαδικτύου», Πρόγραμμα Τεχνολογικών Επιδεικτικών Έργων, Παραδοτέο ΕΕ4, Αθήνα
8. ΕΠΙΣΕΥ-ΕΜΠ & ALTEC A.B.E.E., 2005, «Τεύχος 5 Εγχειρίδιο Χρήσης», στα πλαίσια του έργου «Ανάπτυξη Συστοιχίας Υπολογιστών Υψηλών Επιδόσεων για Προηγμένους Εξυπηρετητές Ιστοσελίδων Διαδικτύου», Πρόγραμμα Τεχνολογικών Επιδεικτικών Έργων, Παραδοτέο ΕΕ4, Αθήνα
9. Καραμανλή Μ. Σ., 2008, «Μελέτη και Προσομοίωση Τεχνικών Κωδικοποίησης Διαύλου για Σύγχρονα Συστήματα Ασυρμάτων Επικοινωνιών», Διπλωματική εργασία για το τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

του Εθνικού Μετσοβίου Πολυτεχνείου, τομέας Συστημάτων Μετάδοσης Πληροφορίας και Τεχνολογίας Υλικών, Αθήνα

10. Manasse M. S., Thekkath C. A. και Silverberg A., «A Reed-Solomon Code for Disk Storage, and Efficient Recovery Computations for Erasure-Coded Disk Storage», συνεργασία των Microsoft Research, Mountain View και Mathematics Department, University of California at Irvine, California, USA

11. Mattman T. W., 1992, «The computation of Galois groups over function fields», Department of Mathematics and Statistics, McGill University, Montreal

12. Μπακόπουλος Α. και Χρυσοβέργης Ι., 1999, «ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΑΡΙΘΜΗΤΙΚΗ ΑΝΑΛΥΣΗ Με Βιβλιοθήκη Προγραμμάτων και Δισκέτα», Εκδόσεις: Συμεών, Αθήνα

13. Plank J.S., 1996, «A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems», Technical Report CS-96-332, Department of Computer Science, University of Tennessee, Knoxville

14. Plank J. S. & Ding Y., 2003, «Note: Correction to the 1997 Tutorial on Reed-Solomon Coding», Technical Report UT-CS-03-504 Department of Computer Science University of Tennessee, Knoxville

15. Rizzo L., «Effective Erasure Codes for Reliable Computer Communication Protocols», Dip. di Ingegneria dell' Informazione, Universita di Pisa, Pisa (Italy)

16. Silberschatz A., Korth H. F. και Sudarshan, 2004, «Συστήματα Βάσεων Δεδομένων. Τέταρτη Έκδοση», απόδοση Μαίρη Γκλαβά, Εκδόσεις: Μ. Γκιούρδας, Αθήνα

17. Wilkins D. R., 2006, «Part IV: Introduction to Galois Theory», Course 311: Hilary Term 2006, Copyright @ David R. Wilkins 1997 – 2006

18. Χατζηκωνσταντίνος Γ. Κ., 2008, «Αποδοτική και Αξιόπιστη Επικοινωνία με Κατανεμημένα Συστήματα Αποθήκευσης Δεδομένων μέσω Δικτυακής Συσκευής Αποθήκευσης», Διπλωματική εργασία για το τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου, τομέας Τεχνολογίας, Πληροφορικής και Υπολογιστών, Εργαστήριο Υπολογιστικών Συστημάτων, Αθήνα