Πανεπιστημιο Θεσσαλιας
Πολυτεχνικη Σχολη
Τμημα Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Μελέτη συμπεριφοράς εφαρμογών και μηχανισμών προστασίας κατά την εκτέλεση σε μη αξιόπιστους υπολογιστές

Διπλωματικη Εργασια

ΓΑΡΥΦΑΛΛΟΥ ΓΙΩΡΓΟΣ

Επιβλέποντες: Χρήστος Αντωνόπουλος, Επίκουρος Καθηγητής
Νικόλαος Μπέλλας, Αναπληρωτής Καθηγητής

Βόλος, Οκτώβριος 2016

Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Μελέτη συμπεριφοράς εφαρμογών και μηχανισμών προστασίας κατά την εκτέλεση σε μη αξιόπιστους υπολογιστές

Διπλωματικη Εργασια

ΓΑΡΥΦΑΛΛΟΥ ΓΙΩΡΓΟΣ

Επιβλέποντες: Χρήστος Αντωνόπουλος, Επίκουρος Καθηγητής
Νικόλαος Μπέλλας, Αναπληρωτής Καθηγητής

Εγκρίθηκε από τη διμελή εξεταστική επιτροπή την 18η Οκτωβρίου 2016.

(Υπογραφή)	*(Υπογραφή)*
.....	.....
Χρήστος Αντωνόπουλος	Νικόλαος Μπέλλας
Επίκουρος Καθηγητής	Αναπληρωτής Καθηγητής

Βόλος, Οκτώβριος 2016

Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική εργασία για την απόκτηση του Διπλώματος του Μηχανικού Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας, στο πλαίσιο του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Θεσσαλίας.

(Υπογραφή)

.....
Γαρυφάλλου Γιώργος

Στην οικογένεια και τους φίλους μου

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες καθηγητές μου κ. Χρήστο Αντωνόπουλο και κ. Νικόλαο Μπέλλα, καθώς και τους συναδέλφους Κωνσταντίνο Παρασύρη, Βασίλη Βασιλειάδη και Κουτσοβασίλη Πάνο για την συνεχή καθοδήγηση τους καθ΄ όλη τη διάρκεια αυτής της εργασίας. Θα ήθελα επίσης να ευχαριστήσω την οικογένεια μου και τους φίλους μου, που ήταν στο πλευρό μου καθ΄ όλη τη διάρκεια της φοίτησής μου.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κατάλογος Αλγορίθμων

Κεφάλαιο 1

Εισαγωγή

Η επεκτασιμότητα της διαδικασίας κατασκευής ημιαγωγών υπήρξε η κινητήρια δύναμη της εκθετικής αύξησης των δυνατοτήτων των συστημάτων ηλεκτρονικών υπολογιστών. Το μέγεθος των τρανζίστορ συνεχίζει να μειώνεται με την εξέλιξη της τεχνολογίας και, όπως παρατήρησε ο Γκόρντον Μουρ, ο αριθμός των τρανζίστορ ενός πυκνού ολοκληρωμένου κυκλώματος διπλασιάζεται κάθε δύο χρόνια. Αποτέλεσμα αυτού είναι η συνεχόμενη αύξηση της πυκνότητας των τρανζίστορ στους επεξεργαστές και κατ' επέκταση η βελτίωση στην απόδοσή τους. Ωστόσο, η συνεχόμενη βελτίωση της απόδοσης απαιτεί αύξηση στην κατανάλωση ενέργειας των επεξεργαστών, κάτι που με το σημερινό ρυθμό αύξησης των τρανζίστορ θα οδηγούσε σε τρομακτικές απαιτήσεις στην κατανάλωση.

Η κατανάλωση ενέργειας αποτελεί πρωταρχικό μέλημα κατά το σχεδιασμό των σύγχρονων υπολογιστικών συστημάτων. Από τη μία μεριά, τα κινητά υπολογιστικά συστήματα λειτουργούν σύμφωνα με αυστηρό προϋπολογισμό ενέργειας και πρέπει να μεγιστοποιήσουν την αυτονομία τους. Από την άλλη μεριά, ένα σημαντικό ποσοστό κόστους της λειτουργίας μεγάλης κλίμακας συστημάτων HPC και κέντρων δεδομένων, οφείλονται στην κατανάλωση ενέργειας. Με την αύξηση της πυκνότητας των τρανζίστορ στους επεξεργαστές και την αποτυχία της κλιμάκωσης του Dennard [4], [5], [6], [7], που προέβλεψε ότι η απόδοση ανά watt θα συνεχίσει την εκθετική αύξησή της με ρυθμό αντίστοιχο της πυκνότητας των τρανζίστορ, οι σχεδιαστές οδηγήθηκαν στη δημιουργία πολυπύρηνων επεξεργαστών. Ωστόσο, η κλιμάκωση των πολυπύρηνων επεξεργαστών περιορίζεται επίσης από την κατανάλωση ενέργειας, κάτι που αναγκαιοποιεί την προσέγγιση σε ενεργειακά αποδοτικά συστήματα.

Κλιμακώνοντας τα τρανζίστορ σε μικρότερες γεωμετρίες, τείνουν να ενισχύονται οι επιπτώσεις της κατασκευαστικής μεταβλητότητας, κάτι που οδηγεί σε λιγότερο ντετερμινιστικά ηλεκτρικά χαρακτηριστικά. Αυτό συνεπάγεται επιπτώσεις στην απόδοση και την κατανάλωση ενέργειας. Η στοχαστικότητα των χαρακτηριστικών του τρανζίστορ οδηγεί σε μειωμένη απόδοση και αυξημένες ζώνες προστασίας στην τάση και τη συχνότητα. Αυτές οι ζώνες προστασίας είναι υπερβολικά απαισιόδοξες, δεδομένου ότι πρέπει να αντισταθμίσουν τα χειρότερα σενάρια και τους συνδυασμούς του μη ντετερμινισμού, της θερμοκρασίας και τις επιδράσεις της γήρανσης κ.λπ. Το μέσο κόστος ηλεκτρικής ενέργειας που καταναλώνεται από τις ζώνες προστασίας είναι περίπου 35% [8]. Τέτοια σενάρια και συνδυασμοί θα εμφανιστούν πολύ

σπάνια, αν όχι ποτέ, καθ' όλη τη διάρκεια της ζωής κάθε συστατικού ενός τσιπ.

Ακόμη και με τη χρήση αυξημένων ζωνών προστασίας (guard bands) στην τάση τροφοδοσίας και τη συχνότητα ενός κυκλώματος, η μεγάλη ποσότητα στοιχείων που απαιτείται για τη δημιουργία συστημάτων μεγάλης κλίμακας (τάξης exa: συστήματα που είναι ικανά να κάνουν τουλάχιστον 10^6 υπολογισμούς το δευτερόλεπτο) αναπόφευκτα αφήνει τέτοια συστήματα ευάλωτα σε συχνές βλάβες. Πρόσφατες μελέτες δείχνουν ότι εάν δημιουργηθεί ένα σύστημα αυτής της κλίμακας με τα σημερινά συστατικά, το ποσοστό αποτυχίας του θα κυμαίνεται από μία βλάβη κάθε 37 λεπτά έως μία βλάβη κάθε 3 λεπτά [9], [10]. Αλλοιωμένα δεδομένα και λανθασμένα αποτελέσματα αναμένεται επίσης να εμφανίζονται πιο συχνά [11].

Μέρος του προβλήματος των απαισιόδοξων ζωνών προστασίας και της ενεργειακής ανεπάρκειας είναι ότι τα σύγχρονα υπολογιστικά συστήματα εκτελούν προγράμματα υπό αυστηρές απαιτήσεις ορθότητας. Σε αρκετά πεδία εφαρμογών, ωστόσο, ο χρήστης δεν ενδιαφέρεται για το ακριβές αποτέλεσμα, αλλά για μία προσέγγιση αυτού [12], [13]. Για παράδειγμα, σε αναλύσεις μεγάλου όγκου δεδομένων μπορεί κανείς να ενδιαφέρεται για μία πρόχειρη ταξινόμηση των δεδομένων σε ομάδες μεγάλης κλίμακας και όχι για την ακριβή τιμή αυτών των δεδομένων. Ομοίως, μέσα σε κάθε πρόγραμμα όλοι οι υπολογισμοί θεωρούνται εξίσου σημαντικοί για την ποιότητα του τελικού αποτελέσματος, παρόλο που στις περισσότερες περιπτώσεις αυτή η υπόθεση δεν αληθεύει.

Λαμβάνοντας υπόψη τις παραπάνω παρατηρήσεις, μελετάται το πεδίο της επιστήμης υπολογιστών που έχει να κάνει με την εκτέλεση εφαρμογών πάνω από αναξιόπιστο υλικό. Για να γίνει αυτό πρέπει να διερευνηθεί η συμπεριφορά της εφαρμογής, με στόχο την αναδόμηση των υπολογισμών ώστε να μπορεί να εκτελεστεί πάνω από αναξιόπιστο υλικό. Η μελέτη αυτή συνοψίζεται στην εύρεση της ανεκτικότητας της εφαρμογής σε λάθη και στη δημιουργία μηχανισμών προστασίας κατά την εκτέλεσή της πάνω από αναξιόπιστό υλικό, με στόχο τα αποτελέσματα που προκύπτουν να αποκλίνουν σε μικρό βαθμό από αυτά μίας αξιόπιστης εκτέλεσης.

Σε αυτή τη διπλωματική εργασία μελετάται η συμπεριφορά δύο εφαρμογών κατά την εκτέλεση τους πάνω από αναξιόπιστο υλικό καθώς και μηχανισμοί προστασίας των εφαρμογών από τυχόν λάθη που δύναται να προκύψουν. Η εισαγωγή σφαλμάτων αφορά σφάλματα χρονισμού και μοντελοποιείται από το λογισμικό αλλάζοντας τυχαία ένα υπολογισμένο αποτέλεσμα. Οι υπολογισμοί των εφαρμογών τμηματοποιούνται σε επιμέρους διακριτές διεργασίες οι οποίες ανάλογα με τη συνεισφορά τους στο τελικό αποτέλεσμα εκτελούνται είτε αξιόπιστα είτε αναξιόπιστα. Για την άνευ λαθών εκτέλεση των εφαρμογών δημιουργούνται μηχανισμοί ελέγχου λαθών και αποσφαλμάτωσης από λανθάνουσες συμπεριφορές που μπορεί να προκύψουν κατά της διάρκεια της εκτέλεσης, με σκοπό το αποτέλεσμα που παράγεται να είναι αποδεκτό.

Η εκτέλεση εφαρμογών πάνω από αναξιόπιστο υλικό στοχεύει στην μείωση της κατανάλωσης ισχύος αξιοποιώντας πυρήνες που λειτουργούν με χαμηλότερη παροχή τάσης. Η μείωση της κατανάλωσης ενέργειας είναι αρκετά σημαντική για συστήματα με περιορισμένο ενεργειακό προϋπολογισμό και η συνεχής επέκταση του τομέα των κινητών συστημάτων αναγκαιοποιεί αυτή τη δράση.

1.1 Οργάνωση της διπλωματικής

Η παρούσα εργασία είναι οργανωμένη σε πέντε κεφάλαια: στο Κεφάλαιο 2 δίνεται το θεωρητικό υπόβαθρο των σχετικών βασικών μοντέλων και τεχνολογιών. Αρχικά περιγράφεται το μοντέλο του συστήματος, στη συνέχεια οι μέθοδοι εισαγωγής σφαλμάτων και η υλοποίηση εισαγωγής τους και τέλος αναλύεται η τμηματοποίηση των υπολογισμών μιας εφαρμογής σε επιμέρους διεργασίες, η ανάθεση σημαντικότητας σε αυτές και ο έλεγχος λαθών των διεργασιών που εκτελέστηκαν αναξιόπιστα. Στο κεφάλαιο 3 αρχικά περιγράφεται η εφαρμογή MD, στη συνέχεια αναλύονται οι παραμετροποιήσεις που έγιναν ώστε να μπορεί να εκτελεστεί η εφαρμογή πάνω από αναξιόπιστο υλικό και τέλος περιγράφεται η πειραματική διαδικασία και η αξιολόγηση των αποτελεσμάτων. Στο κεφάλαιο 4 αρχικά περιγράφεται η εφαρμογή SmallPt και στη συνέχεια, αντίστοιχα με το κεφάλαιο 3, ακολουθεί η ανάλυση των παραμετροποιήσεων και τέλος η πειραματική διαδικασία και η αξιολόγηση των αποτελεσμάτων. Στο κεφάλαιο 5 περιγράφονται οι σχετικές με το θέμα εργασίες.

Κεφάλαιο 2

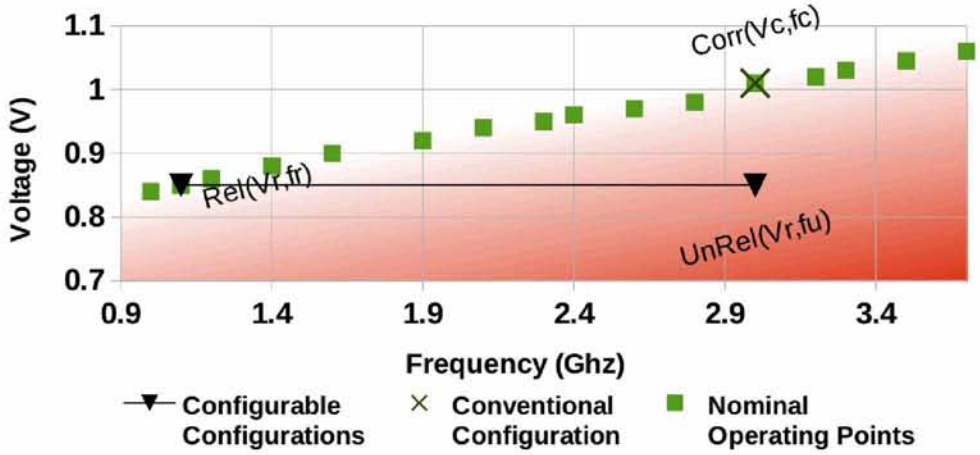
Θεωρητικό Υπόβαθρο

2.1 Μοντέλο συστήματος

Ως βάση για την εργασία θεωρείται ένα σύστημα με αρχιτεκτονική πολλαπλών πυρήνων που στελεχώνεται από δύο είδη πυρήνων, τον συμβατικό και τον παραμετροποιήσιμο (μη συμβατικός). Οι συμβατικοί πυρήνες εκτελούν κώδικα χωρίς σφάλματα σε κατάσταση $Corr = (V_c, f_c)$, όπου V_c είναι η τάση τροφοδοσίας και f_c είναι η αντίστοιχη συχνότητα λειτουργίας [1]. Οι παραμετροποιήσιμοι πυρήνες μπορούν να εναλλάσσουν τη λειτουργία τους ανάμεσα σε δύο διαμορφώσεις: την αξιόπιστη (Rel) και την αναξιόπιστη (UnRel). Θα αναφέρονται οι δύο αυτές διαμορφώσεις ως $Rel = (V_r, f_r)$ και $UnRel = (V_u, f_u)$. Και στις δύο περιπτώσεις ο επεξεργαστής λειτουργεί στην ίδια τάση τροφοδοσίας V_r, αλλά στην αναξιόπιστη έχει μεγαλύτερη συχνότητα $f_u = f_r + \alpha$. Όσο μεγαλύτερο είναι το α τόσα περισσότερα σφάλματα συμβαίνουν κατά τη διάρκεια της εκτέλεσης. Η μετάβαση αυτών των τρόπων λειτουργίας μπορεί να επιτευχθεί με βάση την προσέγγιση που περιγράφεται στο [14]. Έτσι ορίζουμε ως σφάλμα την απόκλιση στην απαιτούμενη λειτουργία του υλικού και το αποτέλεσμα ενός σφάλματος είναι ένα λάθος στους υπολογισμούς.

Ως παράδειγμα, το σχήμα 2.1 δείχνει τα διαφορετικά ονομαστικά σημεία λειτουργίας ενός επεξεργαστή Quad Core i7 (4 πυρήνες), συναρτήσει της συχνότητας (άξονας x) και της τάσης τροφοδοσίας (άξονας y). Κάθε ονομαστικό σημείο είναι υποψήφιο για Corr. Προχωρώντας κάτω από τα ονομαστικά σημεία, μπαίνουμε στη σφαίρα της αναξιόπιστης εκτέλεσης· όσο μεγαλύτερη είναι η απόσταση από το κοντινότερο σημείο Corr τόσο υψηλότερο είναι το ποσοστό σφάλματος. Στην περίπτωση που περιγράφεται, οι διαμορφώσεις Corr και UnRel έχουν επιλεχθεί να έχουν την ίδια συχνότητα. Ωστόσο, η διαμόρφωση UnRel έχει μικρότερη τάση τροφοδοσίας, κάτι που την καθιστά ενεργειακά πιο αποδοτική αλλά ταυτόχρονα ευάλωτη σε σφάλματα. Η διαμόρφωση Rel αντιστοιχεί στην ονομαστική διαμόρφωση που έχει την ίδια τάση τροφοδοσίας με την UnRel, αλλά χαμηλότερη συχνότητα. Αυτό εγγυάται λειτουργία χωρίς σφάλματα, αλλά οδηγεί σε μειωμένη αποδοτικότητα συγκριτικά με τη διαμόρφωση Corr.

Υποθέτουμε ένα σύστημα με τουλάχιστον έναν συμβατικό και έναν παραμετροποιήσιμο πυρήνα. Οι συμβατικοί πυρήνες χρησιμοποιούνται για αξιόπιστη εκτέλεση του λειτουργικού



Σχήμα 2.1: Ονομαστικά σημεία λειτουργίας για έναν επεξεργαστή Quad Core i7 και ενδεικτικές διαμορφώσεις Corr, UnRel και Rel [1]. Αυτά τα ονομαστικά σημεία έχουν εξαχθεί αναγκάζοντας τον πυρήνα να λειτουργήσει σε συγκεκριμένες συχνότητες (με χρήση του εργαλείου likwid) και παρακολουθώντας τις προκείπτουσες τάσεις (με χρήση του εργαλείου sensors).

συστήματος (OS), δηλαδή του κυρίως νήματος (thread) εκτέλεσης εφαρμογών και κάποιων διεργασιών. Οι παραμετροποιήσιμοι πυρήνες χρησιμοποιούνται για να εκτελέσουν κάποιες διεργασίες (tasks) εφαρμογών αναξιόπιστα, με σκοπό την εξοικονόμηση ενέργειας. Οι παραμετροποιήσιμοι πυρήνες εναλλάσσονται σε αξιόπιστη λειτουργία όταν εκτελούν διεργασίες ελέγχου λαθών ή διεργασίες επιπέδου συστήματος.

2.2 Εισαγωγή Σφαλμάτων

2.2.1 Εισαγωγή σφαλμάτων στο υλικό

Η εισαγωγή σφαλμάτων στο υλικό χρησιμοποιείται για να εξετάσουμε τις συνέπειες αυτών και τη συμπεριφορά του υλικού υπό μη αξιόπιστες συνθήκες λειτουργίας. Συνήθως αυτό γίνεται σε κυκλώματα VLSI σε επίπεδο τρανζίστορ, επειδή αυτά τα κυκλώματα είναι αρκετά πολύπλοκα για να δικαιολογήσουν τη μελέτη εισαγωγής σφαλμάτων και μάλιστα αυτή γίνεται καλύτερα κατανοητή σε τέτοιου είδους κυκλώματα. Τα τρανζίστορ συνήθως έχουν σφάλματα τύπου stuck-at (κολλημένα σε μία τιμή 0 ή 1), bridging (γεφύρωσης) [1] ή transient και τα αποτελέσματά τους εξετάζονται κατά τη λειτουργία του κυκλώματος. Τέτοια σφάλματα μπορούμε να εξετάσουμε με λογισμικά προσομοίωσης κυκλωμάτων ή σε κυκλώματα παραγωγής κομμένα από wafer.

[1]Τέτοια σφάλματα (stuck-at και bridging) μπορεί να προκύψουν λόγω ατέλειας του υλικού που χρησιμοποιείται για την παραγωγή ενός κυκλώματος ή λόγω σκόνης στο εργαστήριο παραγωγής, καθώς και λόγω γήρανσης του υλικού

Προσομοιώσεις υλικού γίνονται συνήθως με μία περιγραφή υψηλού επιπέδου του κυκλώματος. Αυτή η υψηλού επιπέδου περιγραφή μετατρέπεται σε περιγραφή επιπέδου τρανζίστορ και τα σφάλματα εισάγονται στο κύκλωμα. Συνήθως αυτά είναι σφάλματα τύπου stuck-at ή bridging, καθώς το λογισμικό προσομοίωσης χρησιμοποιείται συχνότερα για να ανιχνευτούν κατασκευαστικές ατέλειες. Το σύστημα, στη συνέχεια, προσομοιώνεται για να αξιολογηθεί η απόκριση του κυκλώματος σε συγκεκριμένα σφάλματα. Δεδομένου ότι είναι μία προσομοίωση, ένα νέο σφάλμα μπορεί εύκολα να εισαχθεί και στη συνέχεια να μετρηθεί η απόκριση του κυκλώματος στο νέο σφάλμα με επανεκτέλεση της προσομοίωσης. Παρόλο που καταναλώνεται χρόνος για την κατασκευή του μοντέλου, την εισαγωγή σφαλμάτων και την προσομοίωση, η προσομοίωση εξυπηρετεί στο ότι γίνονται ευκολότερα αλλαγές στο κύκλωμα σε σχέση με τη διαδικασία παραγωγής. Αυτές οι δοκιμές μπορούν να χρησιμοποιηθούν για τον έλεγχο του κυκλώματος σε πρώιμα στάδια της διαδικασίας σχεδίασης.

Εισαγωγή σφαλμάτων στο υλικό γίνεται και σε πραγματικά παραδείγματα του κυκλώματος μετά την κατασκευή. Το κύκλωμα υποβάλλεται σε κάποιο είδος παρεμβολής για την εμφάνιση του σφάλματος και στη συνέχεια εξετάζεται η συμπεριφορά του. Μέχρι στιγμής αυτό γίνεται με transient (παροδικά) σφάλματα λόγω της μεγάλης δυσκολίας και κόστους για την εισαγωγή stuck-at και bridging σφαλμάτων. Το κύκλωμα συνδέεται σε μία συσκευή ελέγχου, η οποία το θέτει σε λειτουργία, και εισάγοντας σφάλματα εξετάζει τη συμπεριφορά του. Αυτό καταναλώνει χρόνο για την προετοιμασία του κυκλώματος και τις δοκιμές, αλλά γενικά αυτές οι δοκιμές προχωρούν γρηγορότερα σε σύγκριση με τις προσομοιώσεις. Αυτού του είδους η εισαγωγή σφαλμάτων χρησιμοποιείται για τη δοκιμή κυκλωμάτων αμέσως πριν ή κατά την παραγωγή. Αυτοί οι έλεγχοι είναι μη παρεμβατικοί, δεδομένου ότι δεν μεταβάλλουν τη συμπεριφορά του κυκλώματος, εκτός από την εισαγωγή του σφάλματος. Πρέπει να συμπεριληφθούν ειδικά κυκλώματα για να προκαλέσουν ή να προσομοιώσουν σφάλματα στο τελικό κύκλωμα και αυτό πιθανότατα να επηρεάσει τον χρονισμό ή άλλα χαρακτηριστικά του κυκλώματος.

2.2.2 Εισαγωγή σφαλμάτων στο λογισμικό

Η εισαγωγή σφαλμάτων στο λογισμικό χρησιμοποιείται κυρίως για να εξεταστούν οι συνέπειες αυτών στο λογισμικό, η ανεκτικότητα και η συμπεριφορά του. Χρησιμοποιείται συνήθως σε κώδικα που έχει επικοινωνιακές ή συνεταιρικές συναρτήσεις/λειτουργικότητα, ώστε να υπάρχει αρκετή αλληλεπίδραση και να κάνει την εισαγωγή σφαλμάτων χρήσιμη. Μπορούν να εισαχθούν όλων των ειδών σφάλματα, από σφάλματα σε καταχωρητές και σφάλματα μνήμης, αναπαραγωγή ή χάσιμο πακέτων δικτύου, έως και λανθασμένες συνθήκες και flags. Αυτά τα σφάλματα μπορεί να εισάγονται σε προσομοιώσεις πολύπλοκων συστημάτων ή σε λειτουργικά συστήματα, με σκοπό να εξεταστούν τα αποτελέσματα.

Οι προσομοιώσεις λογισμικού είναι συνήθως υψηλού επιπέδου περιγραφές ενός συστήματος, όπου είναι γνωστές οι αλληλεπιδράσεις και τα πρωτόκολλα αλλά όχι οι λεπτομέρειες της υλοποίησης. Αυτά τα σφάλματα τείνουν να δημιουργούν χαμένα μηνύματα, χαμένα χρονικά περιθώρια, επανεκπομπές και άλλα σφάλματα στην επικοινωνία ενός συστήματος. Η προσο-

μοίωση εκτελείται στη συνέχεια για να ανακαλυφθούν οι επιπτώσεις των σφαλμάτων. Λόγω του αφηρημένου χαρακτήρα αυτών των προσομοιώσεων, μπορεί να τρέχουν με μεγαλύτερη ταχύτητα από το πραγματικό σύστημα, αλλά δεν συλλαμβάνουν κατ' ανάγκη τις χρονικές πτυχές του τελικού συστήματος. Αυτού του είδους οι δοκιμές θα πρέπει να εκτελούνται για να ελεγχθεί ένα πρωτόκολλο ή για να εξεταστεί η αντίσταση μίας αλληλεπίδρασης σε σφάλματα. Τυπικά γίνονται νωρίς στον κύκλο σχεδίασης, έτσι ώστε να συμπληρώσουν τις υψηλού επιπέδου λεπτομέρειες προτού επιχειρηθεί η υλοποίηση. Αυτές οι προσομοιώσεις είναι μη παρεμβατικές, καθώς ως προσομοιώσεις δεν μπορούν να αποδώσουν την ακριβή συμπεριφορά ενός συστήματος.

Η εισαγωγή σφαλμάτων στο λογισμικό είναι περισσότερο προσανατολισμένη προς τις λεπτομέρειες υλοποίησης της εφαρμογής και μπορεί να αντιμετωπίσει τόσο διάφορες καταστάσεις ενός προγράμματος όσο και τις αλληλεπιδράσεις και την επικοινωνία του. Τα σφάλματα εδώ δημιουργούν χαμένα χρονικά περιθώρια, χαμένα μηνύματα, επανεκτελέσεις, κατεστραμμένα δεδομένα σε θέσεις μνήμης και καταχωρητές, λανθάνουσες αναγνώσεις από τον δίσκο και γενικά σφάλματα σε όλες τις καταστάσεις και τις ενέργειες που δίνεται πρόσβαση από το υλικό. Στη συνέχεια το σύστημα εκτελείται με το σφάλμα για να ελεγχθεί η συμπεριφορά του. Αυτές οι προσομοιώσεις τείνουν να διαρκούν περισσότερο, επειδή ενσωματώνουν όλες τις λειτουργίες και τις λεπτομέρειες του συστήματος, αλλά θα συλλάβουν με μεγαλύτερη ακρίβεια τις χρονικές πτυχές του συστήματος. Αυτές οι δοκιμές εισαγωγής σφαλμάτων γίνονται για να ελεγχθεί σε ποια σημεία παρουσιάζει ρήγματα το λογισμικό και να καταγραφούν τα σφάλματα τα οποία μπορεί να αντιμετωπίσει. Αυτό γίνεται αργότερα στον κύκλο σχεδιασμού για να δείξει την απόδοση του τελικού (ή κοντά στο τελικό) σχεδίου. Αυτές οι προσομοιώσεις μπορεί να είναι μη παρεμβατικές ειδικά αν τα χρονικά περιθώρια δεν ανησυχούν τους σχεδιαστές, αλλά στην περίπτωση που δεν ισχύει κάτι τέτοιο ο χρόνος που απαιτείται για να εισαχθούν σφάλματα από τον μηχανισμό εισαγωγής σφαλμάτων μπορεί να διαταράξει τη δραστηριότητα του συστήματος και να προκαλέσει αποτελέσματα τα οποία δεν αντιστοιχούν στα χρονικά περιθώρια του, όταν αυτό τρέχει χωρίς εισαγωγή σφαλμάτων. Αυτό συμβαίνει επειδή ο μηχανισμός εισαγωγής σφαλμάτων εκτελείται πάνω από το ίδιο σύστημα στο οποίο εκτελείται το λογισμικό που ελέγχεται.

Σε αυτή την εργασία εισάγονται σφάλματα στο λογισμικό προσομοιώνοντας την εκτέλεσή του πάνω από αναξιόπιστο υλικό. Στην περίπτωση αυτή το υλικό είναι η πηγή των σφαλμάτων που προκύπτουν στο λογισμικό και μελετάται η συμπεριφορά κάποιων εφαρμογών, ενώ εκτελούνται πάνω από αναξιόπιστο υλικό, με σκοπό τη δημιουργία μηχανισμών που οδηγούν στην ομαλή εκτέλεση της κάθε εφαρμογής.

2.2.3 Μοντελοποίηση σφαλμάτων

Στο παρόν πόνημα εξετάζονται μόνο σφάλματα χρονισμού [1]. Αυτά είναι στενά συνδεδεμένα με τον βαθμό στον οποίο το σύστημα λειτουργεί με μειωμένη τάση τροφοδοσίας (ή είναι υπερχρονισμένο), καθώς επίσης και με το μείγμα εντολών της κάθε εφαρμογής. Εντολές

που ενεργοποιούν μεγάλα μονοπάτια (στο pipeline) που είναι κοντά στο κρίσιμο μονοπάτι[2], τείνουν να αποτυγχάνουν πιο συχνά [15]. Η πιθανότητα αποτυχίας της κάθε εντολής είναι στενά συνδεδεμένη με τη μικροαρχιτεκτονική του επεξεργαστή, καθώς και με τη διαδικασία κατασκευής. Ακόμη και τσιπ με την ίδια αρχιτεκτονική, χρησιμοποιώντας βιβλιοθήκες της ίδιας τεχνολογίας, μπορεί να έχουν εντελώς διαφορετική συμπεριφορά [8]. Επιπλέον, στην περίπτωση που ένα σφάλμα οδηγήσει σε λάθος[3], αυτό δεν εξαρτάται μόνο από τα μονοπάτια που ενεργοποιούνται κατά τη διάρκεια των τρεχόντων κύκλων, αλλά και από τα μονοπάτια που έχουν ενεργοποιηθεί στο παρελθόν [16]. Η μοντελοποίηση τέτοιων μη ντετερμινιστικών φαινομένων είναι σχεδόν αδύνατη, καθώς τα συμπεράσματα είναι άμεσα συνδεδεμένα με το συγκεκριμένο σύστημα που χρησιμοποιήθηκε για τη δημιουργία του μοντέλου. Σύμφωνα με όσα είναι γνωστά την παρούσα χρονική στιγμή δεν υπάρχει κάποιο μοντέλο που να συνδυάζει όλες τις παρατηρήσεις σε μία ενιαία εφαρμόσιμη μέθοδο. Για το λόγο αυτό, δεν εξετάζεται το μείγμα εντολών της εφαρμογής και λαμβάνονται υπόψη μόνο οι επιπτώσεις της μειωμένης τάσης τροφοδοσίας. Όπως έχει ήδη αποδειχθεί [8], το ποσοστό σφάλματος κατά το οποίο το κύκλωμα αρχίζει να αποτυγχάνει στους χρονικούς περιορισμούς (Point of First Failure, PoFF) είναι εξαιρετικά χαμηλό: περίπου 1 σφάλμα κάθε 10 εκατομμύρια κύκλους μηχανής. Πέρα από αυτό το σημείο το ποσοστό σφάλματος αυξάνει εκθετικά μία τάξη μεγέθους για κάθε $10mV$ πτώσης της τάσης τροφοδοσίας [17], [8]. Όπως αναφέρθηκε νωρίτερα, για να εξασφαλιστεί η λειτουργική ορθότητα, οι σχεδιαστές συνήθως λογαριάζουν τις διακυμάνσεις των παραμέτρων επιβάλλοντας συντηρητικά περιθώρια που προφυλάσσουν από τις χειρότερες περιπτώσεις. Η έκταση των ορίων τάσης, που απαιτείται για να εξασφαλιστεί η άνευ σφαλμάτων λειτουργία υπό οποιεσδήποτε συνθήκες λειτουργίας του τσιπ, είναι συνήθως γύρω από το 15% [18]. Καθορίζεται το PoFF βάση της εξίσωσης (2.1), όπου α είναι το ποσοστό του πρόσθετου ορίου στην τάση τροφοδοσίας που εγγυάται την απρόσκοπτη λειτουργία του κυκλώματος και V_n είναι η ονομαστική τάση τροφοδοσίας. Επιλέγεται $\alpha = 15\%$ αφού φαίνεται να είναι συνεπής με αρκετές παρατηρήσεις που αναφέρονται στη βιβλιογραφία [18], [17], [8].

$$V_{PoFF} = \frac{100 - \alpha}{100} * V_n \tag{2.1}$$

Με βάση τις παραπάνω αναφορές, μοντελοποιείται το ποσοστό σφαλμάτων ως μία εκθετική συνάρτηση (2.2) λαμβάνοντας ως παραμέτρους την ονομαστική τάση τροφοδοσίας V_n του ζητούμενου σημείου λειτουργίας V_u. Στην περίπτωσή μας $V_n = V_c$ και V_u είναι η ρύθμιση της τάσης τροφοδοσίας για τον μη συμβατικό πυρήνα. Το μοντέλο αποκτά τις σταθερές β και γ μέσω των δεδομένων που παρέχονται από [17], [8].

$$Err(V_n, V_u) = \beta * e^{\gamma*(V_n - V_u)} \tag{2.2}$$

Τέλος, η συχνότητα λειτουργίας εξαρτάται γραμμικά από την τάση τροφοδοσίας σύμφωνα με την εξίσωση (2.3). Οι παράμετροι δ και ϵ εξαρτώνται από τη διαμόρφωση του συστήμα-

[2]Το κρίσιμο μονοπάτι είναι η μεγαλύτερη αλυσίδα τρανζίστορ μέσα σε ένα τσιπ τα οποία τροφοδοτούν το ένα το άλλο

[3]Λάθος: διαφορά ανάμεσα στο υπολογισμένο αποτέλεσμα και στο σωστό αποτέλεσμα, Σφάλμα: μία λανθασμένη λειτουργία στο υλικό

τος. Εξάγονται οι τιμές τους παρακολουθώντας την τάση τροφοδοσίας του επιλεγμένου επεξεργαστή, ενώ η συχνότητα λειτουργίας αλλάζει χρησιμοποιώντας Dynamic Frequency Scaling(DFS).

$$f(V) = \delta * V + \epsilon \tag{2.3}$$

2.2.4 Υλοποίηση εισαγωγής σφαλμάτων

Χρησιμοποιείται προσομοίωση εισαγωγής σφαλμάτων στο στάδιο εκτέλεσης για να χαρακτηριστεί η συμπεριφορά μίας εφαρμογής σε περιβάλλον που παρουσιάζει λάθη. Στο στάδιο εκτέλεσης τα σφάλματα αλλάζουν την τιμή ενός ψηφίου (bit) στο υπολογισμένο αποτέλεσμα.

2.2.4.1 Αναπαράσταση ακεραίων και αριθμών κινητής υποδιαστολής

Για να κατανοηθεί ο τρόπος που εισάγονται λάθη στα αποτελέσματα τα οποία είναι αποθηκευμένα στις μεταβλητές των διεργασιών θα περιγραφεί πρώτα εν συντομία πώς αναπαρίστανται οι αριθμοί στο δυαδικό σύστημα [19]:

Ακέραιοι αριθμοί: οι ακέραιοι (integers) αριθμοί αναπαριστώνται από τουλάχιστον 16 ψηφία (στην περίπτωσή αυτή είναι 32) με το αριστερότερο να αποτελεί το ψηφίο του προσήμου και τα υπόλοιπα 31 το σημαντικό.

Αριθμοί κινητής υποδιαστολής: οι αριθμοί κινητής υποδιαστολής απλής ακρίβειας αναπαριστώνται με 32 δυαδικά ψηφία και αποθηκεύονται σε 3 πεδία (Σχήμα 2.2) σύμφωνα με το πρότυπο IEEE 754: το πρόσημο (Sign - 1 bit), τον εκθέτη (Exponent - 8 bits) και το σημαντικό μέρος (Mantissa 23 bits). Η εύρεση του δεκαδικού αριθμού σύμφωνα με την αναπαράσταση που φαίνεται παρακάτω υπολογίζεται ως εξής:

$$number_{10} = (-1)^{Sign_{10}} * Mantissa_{10} * 2^{Exponent_{10} - bias} \tag{2.4}$$

Ο όρος bias αναφέρεται στην πόλωση που χρησιμοποιείται από το πρότυπο ώστε να είναι δυνατή και η αναπαράσταση αρνητικών εκθετών, όπου στην περίπτωση της απλής ακρίβειας έχει την τιμή 127 [20].

Σχήμα 2.2: Αναπαράσταση αριθμού κινητής υποδιαστολής [2]

Γνωρίζοντας πλέον τον τρόπο που αναπαριστώνται οι ακέραιοι αριθμοί και οι αριθμοί κινητής υποδιαστολής γίνεται εύκολα αντιληπτή η επίπτωση που θα έχει ένα bit flip στο αποτέλεσμα που είναι αποθηκευμένο σε μία μεταβλητή. Έστω για παράδειγμα ότι έχουμε

αποθηκεύσει σε μία μεταβλητή *var* την τιμή 0, 15. Η αναπαράσταση αυτού του αριθμού σύμφωνα με το πρότυπο IEEE 754 είναι η ακολουθία που φαίνεται στην γραμμή 1 του πίνακα 2.1. Το αριστερότερο ψηφίο αποτελεί το πιο σημαντικό ψηφίο (MSB - Most Significant Bit) της αναπαράστασης και αντιπροσωπεύει το πρόσημο, ενώ το δεξιότερο ψηφίο είναι το λιγότερο σημαντικό (LSB - Less Significant Bit) αυτής της αναπαράστασης και αποτελεί ένα τμήμα από το σημαντικό μέρος (mantissa). Μετρώντας λοιπόν από δεξιά προς τα αριστερά και από το 0 έως το 31, αν γίνει bit flip το ψηφίο στη θέση 30 και αλλάξει από 0 σε 1 (Πίνακας 2.1, γραμμή 2) τότε ο αριθμός που είναι αποθηκευμένος στη μεταβλητή *var* θα αλλάξει σε μεγάλο βαθμό (θα γίνει 5.104236ε+37) και κατ' επέκταση θα επηρεάσει όλους τους υπολογισμούς που ακολουθούν και εμπεριέχουν τη μεταβλητή *var*.

	Δυαδικό	Δεκαδικό
Αρχικός αριθμός	00111110000110011001100110011010	0,15
Μετά από εναλλαγή ψηφίου	01111110000110011001100110011010	5.104236ε+37

Πίνακας 2.1: Παράδειγμα εναλλαγής ψηφίου

Παρακάτω παρουσιάζονται τμήματα κώδικα για τον τρόπο με τον οποίο εισάγονται σφάλματα σε τυχαία ψηφία των μεταβλητών του προγράμματος, καθώς και για το πώς εξάγονται τα μέρη μίας μεταβλητής απλής ακρίβειας.

Κώδικας για την αναπαράσταση των τμημάτων μίας 32-bit float (απλής ακρίβειας) μεταβλητής (Αλγόριθμος 2.1). Η απλή ακρίβεια αποτελείται από 1 ψηφίο για το πρόσημο, 8 ψηφία για τον εκθέτη και 23 ψηφία για το σημαντικό μέρος σύμφωνα με το πρότυπο 754 της IEEE.

```
1 /*************************************************
2  * Get the sign, mantissa and exponent
3  * of a 32-bit floating point number
4  *************************************************/
5 typedef union {
6   float f;
7   struct {
8     unsigned int mantissa : 23;
9     unsigned int exponent : 8;
10    unsigned int sign : 1;
11  } parts;
12 } double_cast;
```

Αλγόριθμος 2.1: Σημαντικό μέρος, εκθέτης και πρόσημο αριθμού απλής ακρίβειας

Δηλώνοντας πλέον μία μεταβλητή *var* με τον τύπο που αναφέρεται παραπάνω - έστω *double_cast var* -, μπορεί αν γίνει ανάθεση τιμής στη μεταβλητή και να υπάρχει πρόσβαση στα 3 πεδία αυτής της μεταβλητής ως εξής:

- $var.f = x$ - Ανάθεση μίας τιμής x στη μεταβλητή *var*
- *var.parts.sign* - Η τιμή του προσήμου της μεταβλητής *var*

- *var.parts.exponent* - Η τιμή του εκθέτη της μεταβλητής *var*
- *var.parts.mantissa* - Η τιμή του σημαντικού της μεταβλητής *var*

2.2.4.2 Αποσπάσματα κώδικα

Κώδικας για την αλλαγή ενός ψηφίου με δύο πιθανές περιπτώσεις: αν το ψηφίο είναι 0 τότε εναλλάσσεται σε 1, ενώ αν είναι 1 τότε εναλλάσσεται σε 0. Το ψηφίο που αλλάζει είναι αυτό που βρίσκεται στη θέση n. Η αλλαγή αυτή γίνεται με τη χρήση μίας bitwise XOR ανάθεσης a ^= b ⇒ a = a ^ b.

```
1 /**************************************************
2  * Toglle the n-nth bit of a number
3  **************************************************/
4 number ^= 1<<n;
```

Αλγόριθμος 2.2: Εναλλαγή του ψηφίου ενός αριθμού στη θέση n

Στους κώδικες που ακολουθούν για την εισαγωγή σφαλμάτων στις μεταβλητές του προγράμματος χρησιμοποιείται η παραπάνω μέθοδος με την εξής αλλαγή: αντικαθίσταται το n με την έκφραση $rand()\%num\_of\_bits$, όπου num_of_bits είναι ο αριθμός των ψηφίων του εκάστοτε πεδίου. Η έκφραση $rand()\%num\_of\_bits$ επιστρέφει έναν τυχαίο ακέραιο στο πεδίο [0,num_of_bits] και σε εκείνο το σημείο θα γίνει το bit flip.

Κώδικας για την εισαγωγή λάθους σε μία μεταβλητή τύπου *integer/int* (Αλγόριθμος 2.3). Δίνεται μία αναφορά (*pointer*) της μεταβλητής που θα εισαχθεί σφάλμα στη συνάρτηση $fault\_in\_int()$ και η τιμή της μεταβλητής αλλάζει ως εξής: γίνεται τυχαία ένα *bit flip* σε ένα από τα 32 ψηφία της μεταβλητής (γραμμή 8). Η έκφραση $(8 * (int)sizeof(int))$ ισούται με την τιμή 32, επομένως η έκφραση $(rand()\%(8 * (int)sizeof(int)))$ επιστρέφει έναν τυχαίο ακέραιο αριθμό στο πεδίο [0,31].

```
1 /**************************************************
2  * Insert fault in integer by toggling a random bit.
3  * Same code for unsigned int by replacing
4  * sizeof(int) with sizeof(unsigned int)
5  * sizeof(int) returns size in bytes
6  **************************************************/
7 void fault_in_int(int *var){
8   *var ^= 1<<(rand()%(8*(int)sizeof(int)));
9 }
```

Αλγόριθμος 2.3: Εισαγωγή σφάλματος σε ακέραιο αριθμό

Κώδικας για την εισαγωγή λάθους σε μία μεταβλητή τύπου float (Αλγόριθμος 2.4). Δίνεται μία αναφορά (*pointer*) της μεταβλητής που θα εισαχθεί σφάλμα στη συνάρτηση $fault\_in\_float()$ και η τιμή της μεταβλητής αλλάζει ως εξής: γίνεται τυχαία ένα bit flip σε ένα από τα 3 πεδία (*sign*, *exponent*, *mantissa*) της μεταβλητής τύπου float το οποίο επιλέγεται επίσης τυχαία. Το πεδίο του προσήμου αποτελείται από ένα bit (ψηφίο), το πεδίο του

εκθέτη από 8 ψηφία και τέλος το πεδίο του σημαντικού αποτελείται από τα 23 εναπομείναντα ψηφία.

```
/************************************************
 * Insert fault in a 32-bit float number by toggling
 * one random bit of its parts
 * (mantissa,exponent,sign)
 ************************************************/
void fault_in_float(float *var){
  double_cast temp;
  temp.f = *var;
  switch( rand()% 3 ){
    case 0://Fault in Sign(1-bit)
      temp.parts.sign ^= 1<<0;
      *var = temp.f;
      break;
    case 1://Fault in Exponent(8-bits)
      temp.parts.exponent ^= 1<<(rand()%8);
      *var = temp.f;
      break;
    case 2://Fault in Mantissa(23-bits)
      temp.parts.mantissa ^= 1<<(rand()%23);
      *var = temp.f;
      break;
  }
}
```

Αλγόριθμος 2.4: Εισαγωγή σφάλματος σε αριθμό κινητής υποδιαστολής

2.3 Τμηματοποίηση υπολογισμών σε διεργασίες

Για την εκτέλεση των εφαρμογών πάνω από αναξιόπιστο υλικό είναι απαραίτητη η τμηματοποίηση των υπολογισμών της κάθε εφαρμογής σε ανεξάρτητα μεταξύ τους μέρη, τα οποία θα εκτελεστούν πάνω από τους διαθέσιμους πυρήνες του συστήματος. Η επιλογή του πυρήνα εκτέλεσης (Rel ή UnRel) του κάθε τμήματος γίνεται βάσει της σημαντικότητάς του και επιλέγεται κατά την εκτέλεση, δεδομένου ότι η σημαντικότητα μιας διεργασίας μπορεί να αλλάξει κατά τη διάρκεια των υπολογισμών. Για την ομαλή και με επιθυμητά αποτελέσματα εκτέλεση της εφαρμογής, ο προγραμματιστής πρέπει να ενσωματώσει ρουτίνες ελέγχου και αποσφαλμάτωσης στην εφαρμογή.

Αξιοποίηση σύγχρονων πλατφορμών: για να τρέξουν οι εφαρμογές πάνω από αναξιόπιστο υλικό χωρίς καταστροφικά σφάλματα ή ανεξέλεγκτο υποβιβασμό του τελικού αποτελέσματος είναι απαραίτητο να γίνει αναδόμηση αυτών, ώστε η εκτέλεση των υπολογισμών να μπορεί να γίνει πάνω από μοντέρνο πολυεπεξεργαστικό σύστημα με αποτελεσματικό τρόπο. Συγκεκριμένα χρησιμοποιούνται οι επεξεργαστές με καλύτερη ενεργειακή απόδοση, έχοντας όμως ως αντίκτυπο την αυξημένη αναξιοπιστία.

Προκειμένου να επιτευχθεί αυτό υιοθετείται η διεπαφή προγραμματισμού OpenMP [21]

ώστε να "σπάσουμε' τους υπολογισμούς σε μικρότερες διεργασίες (tasks) που μπορούν να εκτελεστούν αυτούσιες πάνω από έναν πυρήνα (Rel ή UnRel). Ο ορισμός των διεργασιών εκφράζεται με εντολές compiler #pragma (#pragma compiler directives). Από τη μία πλευρά, τα μοντέλα που βασίζονται σε διεργασίες προσφέρουν έναν απλό τρόπο για να εκφράσουν τον παραλληλισμό με τη μορφή διακριτών εργασιών, καθώς επίσης και στο να συλλάβουν τις εξαρτήσεις δεδομένων μεταξύ των εργασιών με σαφή τρόπο, αφήνοντας τον προγραμματισμό των εργασιών (τον τρόπο δηλαδή με τον οποίο ανατίθενται οι εργασίες στους διαθέσιμους πόρους ώστε να εκτελεστούν) στο σύστημα χρόνου εκτέλεσης (runtime system). Από την άλλη πλευρά, οι εντολές (#pragma) διευκολύνουν τους προοδευτικούς και μη επεμβατικούς μετασχηματισμούς κώδικα, χωρίς να απαιτείται να ξαναγραφτεί πλήρως ο κώδικας. Αυτό είναι ζωτικής σημασίας, αν κάποιος φιλοδοξεί να αξιοποιήσει την τεράστια ποσότητα κώδικα που είναι ήδη διαθέσιμος.

2.3.1 Παραδοχές του προγραμματιστή

Ο προγραμματιστής πρέπει να είναι εξοικειωμένος με την εφαρμογή και ως εκ τούτου να μπορεί να πάρει σοφές αποφάσεις ως προς το πώς να δομήσει το υπολογιστικό μέρος της εφαρμογής σε μορφή επιμέρους διεργασιών, ποιες διεργασίες να θεωρήσει περισσότερο σημαντικές για το τελικό αποτέλεσμα, καθώς και ποιες θα είναι οι αντίστοιχες συναρτήσεις ελέγχου του αποτελέσματος των διεργασιών. Πρέπει να τονιστεί ότι η σημαντικότητα των διεργασιών είναι άμεσα συνδεδεμένη με την κάθε εφαρμογή και δεν μπορεί να γίνει οποιαδήποτε αυθαίρετη γενίκευση. Ένα παρόμοιο επίπεδο εμπειρογνωσίας απαιτείται ώστε να παραλληλιστεί ένα κομμάτι υπολογισμού και να κατανεμηθεί αποτελεσματικά σε ένα πολυπύρηνο σύστημα. Η σημαντικότητα των διεργασιών είναι βασική πτυχή του σχεδιασμού για την καλύτερη δυνατή επίδοση και απαιτεί την πλήρη προσοχή του προγραμματιστή, όπως ακριβώς και η παραλληλοποίηση.

Η επιλογή της συνάρτησης ελέγχου λαθών έχει ιδιαίτερη σημασία. Εάν η συνάρτηση ελέγχου λαθών είναι πολύπλοκη τότε αποτυγχάνουμε πρακτικά, εφόσον θα μπορούσε να επιτευχθεί το επιθυμητό αποτέλεσμα δηλώνοντας εξαρχής τη διεργασία ως σημαντική και εκτελώντας την αξιόπιστα, αποφεύγοντας έτσι το κόστος της συνάρτησης ελέγχου λαθών. Εάν είναι πολύ απλή, τότε ενδέχεται να καταστρέψει μία καλή έξοδο μιας διεργασίας και εν συνεχεία μπορεί να επιδεινωθεί το τελικό αποτέλεσμα του υπολογισμού.

2.3.2 Ορισμός διεργασιών

Οι διεργασίες καθορίζονται από την εντολή compiler #pragma omp task ακολουθούμενη από τη συνάρτηση ή το block κώδικα που αποτελεί τη διεργασία. Ανάλογα με τη σημαντικότητά τους, οι διεργασίες μπορεί να εκτελεστούν πάνω από αξιόπιστο ή αναξιόπιστο υλικό.

Οι εντολές #pragma επιτρέπουν να προσδιοριστούν τυχόν εξαρτήσεις δεδομένων, καθώς και θέματα συγχρονισμού μεταξύ των διεργασιών.

```
1 /*************************************************
2  * Task creation, execution and synchronization
3  *************************************************/
4 #pragma omp parallel
5 {
6   #pragma omp single nowait
7   {
8     // Create #num_of_tasks Tasks
9     for (i = 0; i < num_of_tasks; ++i){
10      #pragma omp task
11      task( ... );
12    }
13  }
14  #pragma omp taskwait
15 }
```

Αλγόριθμος 2.5: Παράδειγμα δημιουργίας, εκτέλεσης και συγχρονισμού διεργασιών

Στο παραπάνω παράδειγμα (Αλγόριθμος 2.5) φαίνεται ένα κομμάτι κώδικα που δημιουργεί τις διεργασίες, οι οποίες τοποθετούνται στην ουρά αναμονής μέχρι να επιλεχθούν προς εκτέλεση. Η εντολή *taskwait* λειτουργεί σαν εμπόδιο συγχρονισμού (synchronization barrier) για τις διεργασίες, διασφαλίζει δηλαδή ότι θα υπάρξει μία παύση στην τρέχουσα ροή εκτέλεσης μέχρι να εκτελεστούν όλες οι διεργασίες που βρίσκονται στην ουρά αναμονής. Είναι λοιπόν ένα σημείο συνάντησης το οποίο απαγορεύει στα νήματα να προχωρήσουν παρακάτω και ουσιαστικά τα καλεί να εκτελέσουν τις διεργασίες που περιμένουν στην ουρά. Η εντολή *single* είναι απαραίτητη ώστε κάθε διεργασία να δημιουργηθεί μόνο μία φορά. Αν δεν υπήρχε η εντολή *single*, τότε κάθε διεργασία θα είχε δημιουργηθεί τόσες φορές όσες και ο αριθμός των νημάτων που υποστηρίζονται από τον επεξεργαστή, το οποίο δεν είναι επιθυμητό. Τέλος, η εντολή *nowait* δίπλα από την εντολή *single* δίνει την εντολή στα υπόλοιπα νήματα (πέραν αυτού που δημιουργεί τις διεργασίες) να μην περιμένουν να τελειώσει το block που εκτελείται μόνο από ένα νήμα (αφαιρεί το υπονοούμενο φράγμα στο τέλος της εντολής *single*). Οπότε μόλις τα υπόλοιπα νήματα χτυπήσουν το φράγμα *taskwait*, αρχίζουν να εκτελούν όσες διεργασίες έχουν ήδη δημιουργηθεί, χωρίς να περιμένουν την ολοκλήρωση της διαδικασίας δημιουργίας των διεργασιών.

2.4 Σημαντικότητα διεργασιών και έλεγχος σφαλμάτων

Απόδοση Σημαντικότητας: ο προγραμματιστής καθορίζει τη σημαντικότητα των διάφορων τμημάτων του υπολογισμού με βάση το πόσο έντονα συμβάλλει κάθε μέρος στην ποιότητα ή την ορθότητα του τελικού αποτελέσματος. Σημαντικά τμήματα των υπολογισμών πρέπει να εκτελεστούν σωστά σε αξιόπιστους πυρήνες (δηλαδή σε πυρήνες που δεν προκαλούν σφάλματα), ενώ τα λιγότερο σημαντικά τμήματα μπορούν να εκτελεστούν σε δυνητικά αναξιόπιστους πυρήνες -και ενδεχομένως να παράγουν λάθος έξοδο- ή να μην εκτελεστούν καθόλου (αν

κρίνεται δυνατό) και να ελαφρύνουν ακόμα περισσότερο το συνολικό υπολογισμό.

Απομόνωση και έλεγχος σφαλμάτων: ο προγραμματιστής καθορίζει τον έλεγχο των εργασιών που έτρεξαν πάνω από αναξιόπιστο υλικό για αποσφαλμάτωση της εξόδου, έτσι ώστε να μη διαδοθούν σιωπηλά σφάλματα προς το υπόλοιπο του υπολογισμού με ανεξέλεγκτο τρόπο.

Κεφάλαιο 3

Molecular Dynamics - MD

3.1 Περιγραφή

Ένα από τα κύρια εργαλεία στη θεωρητική μελέτη συστημάτων σωματιδίων/ατόμων είναι η μέθοδος προσομοιώσεων μοριακής δυναμικής (Molecular Dynamics, γνωστή και ως MD). Αυτή η υπολογιστική μέθοδος υπολογίζει την εξαρτώμενη από τον χρόνο συμπεριφορά ενός μοριακού συστήματος και εφαρμόζεται σε συστήματα στερεών και υγρών, καθώς και σε συστήματα της κίνησης των αστεριών ή των γαλαξιών.

Για τις ανάγκες της αξιολόγησης, μοντελοποιείται ένα σύστημα από N άτομα υγρού Argon μέσα σε έναν τρισδιάστατο χώρο, τα οποία αλληλεπιδρούν κάτω από διαμοριακές δυνάμεις χωρίς να επηρεάζονται από εξωτερικούς παράγοντες (εξωτερικές δυνάμεις). Διαμοριακές είναι οι ελκτικές ή απωστικές δυνάμεις, οι οποίες δρουν μεταξύ γειτονικών σωματιδίων και είναι σχετικά ασθενείς σε σύγκριση με τις ενδομοριακές δυνάμεις που συγκρατούν τα άτομα σε συγκεκριμένες θέσεις μέσα στο μόριο.

Οι δυνάμεις αλληλεπίδρασης ανάμεσα στα μόρια προέρχονται από το δυναμικό Lennard-Jones [22], το οποίο είναι ένα απλό μαθηματικό μοντέλο που προσεγγίζει την αλληλεπίδραση μεταξύ ενός ζευγαριού ατόμων ή μορίων. Αν και δεν αποτελεί την πιο πιστή αναπαράσταση του δυναμικού ενέργειας επιφάνειας (μαθηματική συνάρτηση που δίνει την ενέργεια ενός σωματιδίου συναρτήσει της γεωμετρίας του), χρησιμοποιείται ευρέως λόγω της υπολογιστικής απλότητάς του. Το δυναμικό Lennard-Jones ενθυλακώνει μία ήπια ελκτική δύναμη για μεγάλες αποστάσεις και μία έντονη απωστική δύναμη για πολύ μικρές αποστάσεις. Η συνεχής κίνηση των σωματιδίων στα υγρά προκαλεί τη σύγκρουση των σωματιδίων τόσο με το πλαίσιο οριοθέτησης όσο και μεταξύ τους. Αυτό έχει ως αποτέλεσμα μία σύντομη παραμόρφωση στα σύννεφα ηλεκτρονίων των σωματιδίων, γνωστή ως διπολική ροπή [23], και τα σωματίδια έλκονται μεταξύ τους με τη μορφή διπόλων. Από την άλλη πλευρά, η απωθητική δύναμη προέρχεται από την αδυναμία του οποιουδήποτε ατόμου να περάσει μέσα από ένα άλλο. Η δύναμη αυτή, σύμφωνα με τον Lennard-Jones, περιγράφεται από την παρακάτω εξίσωση (3.1):

$$F(r) = -\frac{dV(r)}{dr} = \frac{24\epsilon}{\sigma}\left[2\left(\frac{\sigma}{r}\right)^{13} - \left(\frac{\sigma}{r}\right)^{7}\right] \tag{3.1}$$

όπου V είναι το διαμοριακό δυναμικό ανάμεσα σε δύο άτομα η σωματίδια, ϵ είναι το βάθος του δυναμικού (εξαρτώμενο από το υλικό), σ είναι η απόσταση στην οποία το διαμοριακό δυναμικό ανάμεσα σε 2 άτομα/σωματίδια είναι μηδέν, r είναι η απόσταση ανάμεσα στα δύο σωματίδια. Το δυναμικό Lennard-Jones προσομοιάζει τις ελκτικές αλληλεπιδράσεις ως α^7 και τις απωστικές ως α^{13}.

Εν απουσία εξωτερικών δυνάμεων το σύστημα είναι σε κατάσταση ισορροπίας και η ενέργειά του διατηρείται. Σε αυτή την κατάσταση, στατικές ιδιότητες όπως θερμοκρασία και πίεση μετρούνται ως μέσοι όροι με την πάροδο του χρόνου και η διατήρηση της ενέργειας παρακολουθείται μετρώντας τη συνολική ενέργεια του συστήματος περιοδικά κατά τη διάρκεια της προσομοίωσης.

Οι μεταβλητές εισόδου της εφαρμογής είναι: α) οι διαστάσεις του τρισδιάστατου κύβου, οι οποίες αφορούν την οριοθέτηση του χώρου μέσα στον οποίο αλληλεπιδρούν τα σωματίδια (bound), β) η μονάδα χρόνου που διαρκεί μία επανάληψη της προσομοίωσης (dt), γ) ο αριθμός των σωματιδίων (particles number) και δ) η συνολική διάρκεια της προσομοίωσης (duration). Ο τρισδιάστατος κύβος δημιουργείται σύμφωνα με τη μεταβλητή εισόδου bound και έχει διαστάσεις $x_{axis} = (2 * bound)$, $y_{axis} = (2 * bound)$, $z_{axis} = (2 * bound)$. Τα όρια μέσα στον χώρο από τα οποία οριοθετείται ένα σωματίδιο είναι $[-bound, +bound]$ προς κάθε έναν από τους 3 άξονες (x, y, z). Διαιρώντας τη διάρκεια της προσομοίωσης με τη μονάδα χρόνου που διαρκεί μία επανάληψη, προκύπτει ο αριθμός των συνολικών επαναλήψεων που θα τρέξει η εφαρμογή (Εξίσωση 3.2). Αν για παράδειγμα έχουμε $duration = 0.5$ και $dt = 0.001$ τότε θα έχουμε συνολικά 500 επαναλήψεις στην προσομοίωση.

$$total\_steps = \frac{duration}{dt} \tag{3.2}$$

Για κάθε σωματίδιο κατά τη διάρκεια εκτέλεσης της προσομοίωσης αποθηκεύεται: α) η θέση στην οποία βρίσκεται μέσα στο χώρο ως ένα διάνυσμα $(pos_x,\ pos_y,\ pos_z)$, β) η ταχύτητά του ως ένα διάνυσμα $(velocity_x,\ velocity_y,\ velocity_z)$, γ) η δύναμή του ως ένα διάνυσμα $(force_x,\ force_y,\ force_z)$, δ) το δυναμικό του $(potential)$ και ε) η ενέργειά του $(virial)$.

Οι έξοδοι της εφαρμογής είναι η συνολική ενέργεια του συστήματος η οποία προκύπτει από την εξίσωση (3.3) που υπολογίζεται και τυπώνεται ανά 100 επαναλήψεις, καθώς και η μέση πίεση της συνολικής προσομοίωσης της οποίας το άθροισμα ανανεώνεται επίσης ανά 100 επαναλήψεις και ο τελικός μέσος όρος υπολογίζεται μετά το τέλος των επαναλήψεων, διαιρώντας το συνολικό άθροισμα με το πλήθος των δειγματοληψιών που έγιναν για τον υπολογισμό του αθροίσματος (Εξίσωση 3.8).

$$Systems' \ Total \ Energy = Kinetic \ Energy + Potential \ Energy \tag{3.3}$$

όπου η κινητική και η δυναμική ενέργεια υπολογίζονται σύμφωνα με τις εξισώσεις (3.4) και (3.5) αντίστοιχα.

$$Kinetic \ Energy = \sum_{i=1}^{\#particles} \frac{vel_i.x^2 + vel_i.y^2 + vel_i.y^2}{2} \tag{3.4}$$

με $vel_i.x$, $vel_i.y$, $vel_i.z$ να συμβολίζουν την ταχύτητα του σωματιδίου i ως προς τον άξονα x, y, z αντίστοιχα.

$$Potential \ Energy = \sum_{i=1}^{\#particles} potential_i \tag{3.5}$$

Το άθροισμα της πίεσης ανανεώνεται ανά 100 επαναλήψεις σύμφωνα με την παρακάτω εξίσωση (Εξίσωση 3.6):

$$Pressure \ Sum+ = \frac{2 * (Kinetic \ Energy) + (Partial \ Virial)}{3 * \#particles} \tag{3.6}$$

Σημείωση: Ο τελεστής $+=$ συμβολίζει την επιπλέον ανάθεση. Αν έχουμε για παράδειγμα $a+=b$ τότε αυτό μεταφράζεται ως: $a = a + b$

όπου $Kinetic \ Energy$ είναι η κινητική ενέργεια που υπολογίζεται ανά 100 επαναλήψεις και $Partial \ Virial$ είναι το μερικό άθροισμα της ενέργειας των σωματιδίων που υπολογίζεται επίσης κάθε 100 επαναλήψεις ως εξής (Εξίσωση 3.7):

$$Partial \ Virial = \sum_{i=1}^{\#particles} virial_i \tag{3.7}$$

Η μέση πίεση της συνολικής προσομοίωσης υπολογίζεται στο τέλος των επαναλήψεων από την παρακάτω εξίσωση (Εξίσωση 3.8):

$$Simulation \ Average \ Pressure = \frac{Pressure \ Sum}{\#samples} \tag{3.8}$$

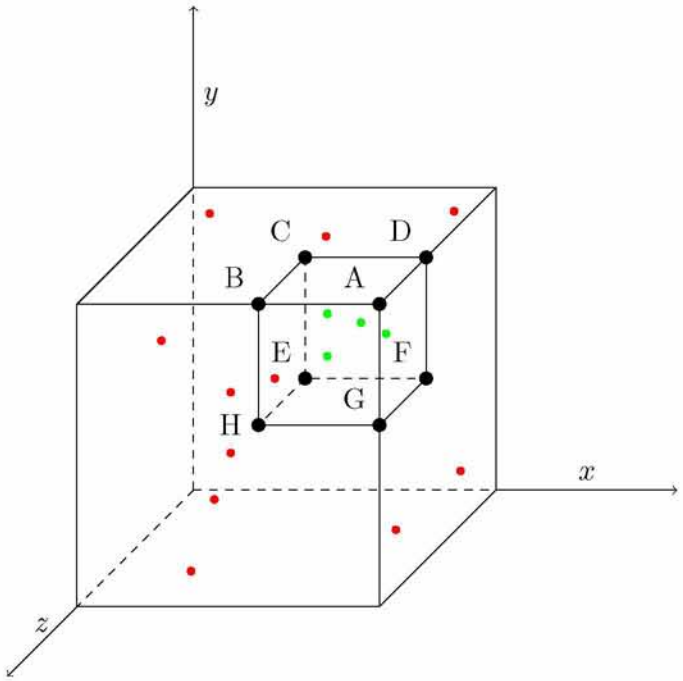
όπου $\#samples$ είναι ο αριθμός των δειγμάτων. Αν για παράδειγμα έχουμε 500 επαναλήψεις συνολικά και κάνουμε δειγματοληψία ανά 100 τότε, λαμβάνοντας δείγμα και κατά την πρώτη επανάληψη (η οποία αριθμείται ως 0), θα έχουμε συνολικά 6 δείγματα (0, 100, 200, 300, 400, 500).

3.2 Ορισμός διεργασιών

Η αναδόμηση των υπολογισμών της εφαρμογής MD έγινε σπάζοντας τον τρισδιάστατο κύβο σε μικρότερους (blocks) και κρατώντας πληροφορία για τα σωματίδια που βρίσκονται μέσα σε κάθε έναν υποκύβο στο εσωτερικό του μεγαλύτερου. Τα σωματίδια συνεχίζουν

να αλληλεπιδρούν με τα υπόλοιπα που βρίσκονται σε άλλα blocks. Η διαμέριση αυτή είναι πολύ χρήσιμη, εφόσον γνωρίζουμε ότι τα σωματίδια γειτονικών block αλληλοεπηρεάζονται σε μεγαλύτερο βαθμό σε σχέση με τα σωματίδια που τα block στα οποία ανήκουν είναι απομακρυσμένα. Αυτή η πληροφορία διευκολύνει την απόδοση σημαντικότητας στις διεργασίες, όπως αναλύεται παρακάτω. Βέβαια σημαντικό ρόλο έχει το μέγεθος του κάθε επιμέρους block σε σχέση με το συνολικό μέγεθος του κύβου, ώστε να εξομοιώνεται η σχετικότητα της κοντινής και μακρινής απόστασης μεταξύ δύο σωματιδίων που ανήκουν σε διαφορετικά block, με αποδεκτό και παράλληλα αξιοποιήσιμο τρόπο.

Παρακάτω (Σχήμα 3.1) φαίνεται ένας τρισδιάστατος κύβος και ένας υποκύβος (ABCDEFGH) στο εσωτερικό του. Οι πράσινες κουκίδες αναπαριστούν τα σωματίδια τα οποία εμπεριέχονται στον υποκύβο, ενώ τα κόκκινα αναπαριστούν τα υπόλοιπα σωματίδια τα οποία βρίσκονται εντός του κύβου και εκτός του υποκύβου.



Σχήμα 3.1: MD - Τρισδιάστατη αναπαράσταση του κύβου και ενός υποκύβου (block)

Ο αρχικός αλγόριθμος, που χρησιμοποιείται για την εύρεση των παραμέτρων που ανήκουν σε κάθε σωματίδιο, διατρέχει όλα τα σωματίδια ένα προς ένα και υπολογίζει την αλληλεπίδραση αυτού με κάθε ένα από όλα τα υπόλοιπα σωματίδια που υπάρχουν στον χώρο (Αλγόριθμος 3.1).

```
1 /*************************************************
2  * Particles Interaction Pseudocode
3  *************************************************/
4 for every particle P {
5     for every particle K {
6         if (P != K) {
7             update variables from interaction(P,K)
```

```
        }
    }
}
```

Αλγόριθμος 3.1: MD - Αλληλεπίδραση σωματιδίων

Ωστόσο, για να γίνει η ανανέωση των μεταβλητών, ο αλγόριθμος υπολογίζει την ταχύτητα και τη θέση ενός σωματιδίου προτού προβεί σε περαιτέρω υπολογισμούς. Προτού δηλαδή ελέγξει όλες τις αλληλεπιδράσεις με τα υπόλοιπα σωματίδια ανανεώνει αυτές τις δύο μεταβλητές του τρέχοντος σωματιδίου. Για τον λόγο αυτό, δημιουργείται η ανάγκη να τμηματοποιηθεί ο αρχικός αλγόριθμος σε δύο μέρη· το ένα αφορά την ανανέωση των δύο αυτών μεταβλητών που αποτελούν ουσιαστικά την προετοιμασία για το κυρίως μέρος του υπολογιστικού μέρους, και το άλλο αφορά το κυρίως μέρος που υπολογίζει την αλληλεπίδραση ενός σωματιδίου με τα υπόλοιπα που βρίσκονται στο χώρο. Για λόγους συμμετρίας το πλήθος των διεργασιών του πρώτου μέρους είναι ίσο με το πλήθος των διεργασιών του δεύτερου.

Μετά την αναδόμηση των υπολογισμών σε blocks, ο αλγόριθμος πλέον εκτελεί δύο λειτουργίες. Πρώτον, ανανεώνει τη θέση και την ταχύτητα των σωματιδίων και δεύτερον, διατρέχει τα block και υπολογίζει την αλληλεπίδραση του κάθε σωματιδίου που ανήκει σε ένα block $B1$ με αυτά ενός block $B2$ (Το $B1$ μπορεί να είναι το ίδιο με το $B2$ καθώς ένα block περιέχει περισσότερα από1 σωματίδια) (Αλγόριθμος 3.2)

```
/************************************************
 * Particles Interaction in a
 * block partitioned block
 ************************************************/
/* First part */
for every block B {
    update_pos_and_vel_for_every_particle_in_B()
}

/* Second part */
for every block B1 {
    for every particle P in B1 {
        for every block B2 {
            for every particle K in B2{
                if (P != K) {
                    update vars from interaction(P,K)
                }
            }
        }
    }
}
```

Αλγόριθμος 3.2: MD - Αλληλεπίδραση σωματιδίων με blocks

Παράδειγμα (Σχήμα 3.2): έστω ότι το μέγεθος του κύβου είναι $200 * 200 * 200$ και θέλουμε να το σπάσουμε σε block με μέγεθος $50 * 50 * 50$ το κάθε ένα, τότε θα προκύψουν 64 blocks και κάθε διεργασία θα αφορά τους υπολογισμούς για τα σωματίδια ενός block σε αλληλεπίδραση με τα σωματίδια ενός άλλου block. Εύκολα προκύπτει πως συνολικά υπάρχουν $64 * 64 = 4096$ διεργασίες που αφορούν τους υπολογισμούς αλληλεπίδρασης μεταξύ των σωματιδίων. Επιπλέον υπάρχουν ακόμη 4096 διεργασίες που αφορούν την ανανέωση της θέσης και της ταχύτητας των σωματιδίων, οπότε συνολικά προκύπτουν 8192 διεργασίες.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47

48	49	50	51
52	53	54	55
56	57	58	59
60	61	32	63

Σχήμα 3.2: MD - Τμηματοποίηση του χώρου

Παρακάτω (Αλγόριθμος 3.3) φαίνεται ο τρόπος με τον οποίο γίνεται η τμηματοποίηση του κύβου σε blocks. Ελέγχουμε αν η θέση ενός σωματιδίου είναι μέσα στο χώρο που οριοθετείται από το τρέχον block [block(x,y,z)] και αν η συνθήκη αυτή είναι αληθής τότε καταχωρούμε την πληροφορία και συνεχίζουμε. Όταν προσπελάσουμε όλα τα block μέσα στον κύβο, τότε έχουμε πληροφορία για όλα τα σωματίδια που εμπεριέχονται σε κάθε block του κύβου.

Μία παρατήρηση που πρέπει να σημειωθεί είναι πως η διαδικασία της τμηματοποίησης πρέπει να γίνεται σε κάθε επανάληψη του αλγορίθμου, δεδομένου ότι τα σωματίδια αλλάζουν θέση στο χώρο και μπορεί να περάσουν από ένα block σε ένα άλλο. Ωστόσο, θα μπορούσε να γίνει ελάφρυνση των υπολογισμών σε κάποιες από τις επαναλήψεις παραβλέποντας τη διαδικασία της τμηματοποίησης σε κάποιες από αυτές, αν κριθεί πως τα αποτελέσματα δε θα αποκλίνουν σημαντικά από το αναμενόμενο.

```
/*************************************************
 * MD partitioning psedo code
 *************************************************/
i = 0; //blocks counter
for z in (-bound, bound) with step=block_size{
  for y in (-bound, bound) with step=block_size{
    for x in (-bound, bound) with step=block_size{
      for every particle P {
        if (P.position inside block(x,y,z)) {
            block_i.add(P).
        }
      }
      i++;  //jump to next block
    }
  }
}
```

Αλγόριθμος 3.3: MD - Τμηματοποίηση του χώρου που αλληλεπιδρούν τα σωματίδια σε blocks

3.3 Σημαντικότητα διεργασιών

Σε ένα σωματίδιο P ασκείται μεγαλύτερη δύναμη από ένα σωματίδιο K το οποίο βρίσκεται σε απόσταση x σε σχέση με αυτή που ασκείται από ένα σωματίδιο L σε απόσταση $x + \alpha$ (όπου α είναι μία θετική σταθερά). Σύμφωνα με αυτήν την παρατήρηση, διαπιστώνεται ότι η αλληλεπίδραση των σωματιδίων είναι μεγαλύτερης σημασίας αν τα σωματίδια είναι κοντά μεταξύ τους, σε σχέση με το αν βρίσκονται μακριά. Κατ' επέκταση, οι διεργασίες που αφορούν δύο κοντινά block έχουν μεγαλύτερη σημαντικότητα σε σχέση με αυτά που απέχουν περισσότερο.

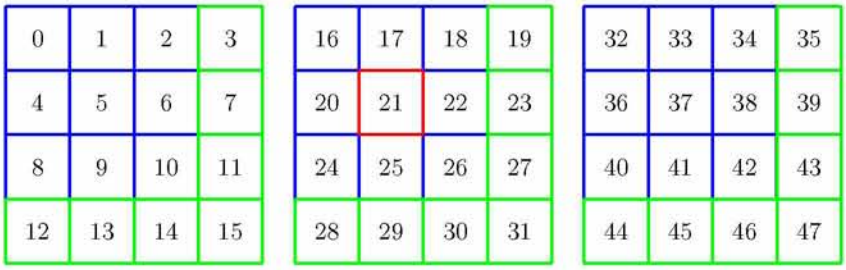
Για την απόδοση της σημαντικότητας στις διεργασίες παίζει ρόλο η απόσταση των block που αφορούν τη διεργασία. Τα δύο block που αφορούν την κάθε διεργασία είναι: α) το block -έστω ότι είναι το block x- το οποίο περιέχει τα σωματίδια για τα οποία κάνουμε του υπολογισμούς και ανανεώνουμε τις μεταβλητές τους στην παρούσα διεργασία και β) το block -έστω ότι είναι το block y- με το οποίο αλληλεπιδρούν τα σωματίδια του block x στην παρούσα διεργασία. Θα γίνεται αναφορά σε αυτή τη διεργασία ως $(x - y)$.

Στο παρακάτω σχήμα (Σχήμα 3.3) φαίνεται ο τρόπος με τον οποίο αποδίδεται σημαντικότητα σε κάθε διεργασία. Έστω ότι έχουμε τμηματοποιήσει τον κύβο σε 48 blocks και θέλουμε να υπολογίσουμε την αλληλεπίδραση των σωματιδίων που ανήκουν στο block 21 με τα σωματίδια όλων των υπολοίπων block του κύβου. Η διεργασία που είναι η σημαντικότερη από όλες είναι αυτή η οποία αφορά τον υπολογισμό για σωματίδια που βρίσκονται στο ίδιο block (Κόκκινο χρώμα: 21-21). Αμέσως μετά, ακολουθούν τα block γείτονες με απόσταση[1] 1 προς τις διαστάσεις x, y και z, τα οποία είναι χρωματισμένα με μπλε και αφορούν τις εξής διεργασίες: (21-0), (21-1), (21-2), (21-4), (21,5), (21-6), (21-8), (21-9), (21-10), (21-16), (21-17), (21-18), (21-19), (21-20), (21-22), (21-24), (21-25), (21-26), (21-32), (21-33), (21-34), (21-36), (21-37), (21-38), (21-40), (21-41), (21-42). Στη συνέχεια ακολουθούν τα block με απόσταση 2 τα οποία είναι χρωματισμένα με πράσινο και αφορούν τις διεργασίες: (21-3), (21-7), (21-11), (21-12), (21,13), (21-14), (21-15), (21-19), (21-23), (21-27), (21-28), (21-29), (21-30), (21-31), (21-35), (21-39), (21-43), (21-44), (21-45), (21-46), (21-47). Αντίστοιχα το μοντέλο επεκτείνεται για περιπτώσεις που υπάρχουν blocks σε μεγαλύτερες αποστάσεις.

3.4 Έλεγχος λαθών

Ο έλεγχος λαθών για την εφαρμογή MD γίνεται σε δύο σημεία, μία φορά μετά την εκτέλεση των πρώτων διεργασιών που υπολογίζουν τη θέση και την ταχύτητα των σωματιδίων και άλλη μία μετά την εκτέλεση των διεργασιών που υπολογίζουν τη δύναμη, το δυναμικό και την ενέργεια κάθε σωματιδίου. Ο έλεγχος λαθών βασίζεται στον υπολογισμό της συνολικής ενέργειας του συστήματος (Εξίσωση 3.3) μετά από την εκτέλεση των διεργασιών και στον

[1]Ορισμός απόστασης: η απόσταση μεταξύ δύο block a και b ορίζεται ως το πλήθος των block που υπάρχει ανάμεσα τους στον τρισδιάστατο χώρο. Η ελάχιστη απόσταση μεταξύ δύο διαφορετικών block είναι ίση με 1 $(a - b$, με $a \neq b)$. Μηδενική είναι η απόσταση ανάμεσα σε ένα block από τον εαυτό του $(a - a)$



Σχήμα 3.3: MD - Προσδιορισμός σημαντικότητας για το block 21. Από υψηλή σε χαμηλή σημαντικότητα: κόκκινο, μπλε, πράσινο

έλεγχο του αποτελέσματος με μία αναμενόμενη τιμή.

Ο αλγόριθμος που υπολογίζει τη συνολική ενέργεια του συστήματος περιγράφεται παρακάτω (Αλγόριθμος 3.4) και δέχεται ως ορίσματα των αριθμό των σωματιδίων, την ταχύτητα και το δυναμικό τους.

```
/***************************************************
 * Function that calculates the systems energy
 * (total energy = Kinetic + Potential)
 ***************************************************/
double calculate_system_energy(int particlesnumber, float4* vel, float* potential) {
  double E_kin = 0.0, E_V = 0.0;
  int i;
  for (i=0; i < particlesnumber; i++)
  {
    E_kin += 0.5 * ((double)vel[i].x*vel[i].x + (double)vel[i].y*vel[i].y
        + (double)vel[i].z*vel[i].z);
    E_V += potential[i];
  }
  return(E_kin + E_V);
}
```

Αλγόριθμος 3.4: MD - Συνάρτηση για τον υπολογισμό της συνολικής ενέργειας του συστήματος

Για τον έλεγχο λαθών μετά την εκτέλεση των πρώτων διεργασιών υπολογίζουμε την ενέργεια του συστήματος μετά την εκτέλεση των διεργασιών (*check_energy_after*), καθώς και το άθροισμα απόλυτων διαφορών ταχύτητας των σωματιδίων (*check_sum_vel*) του συστήματος (Αλγόριθμος 3.5). Η απόλυτη διαφορά είναι η απόλυτη τιμή από τη νέα ταχύτητα που υπολογίστηκε από την εκτέλεση των διεργασιών μείον την ταχύτητα που είχε το σωματίδιο πριν την εκτέλεση των διεργασιών. Ο τελικός έλεγχος αποτελείται από έλεγχο για: α) το αν το συνολικό άθροισμα έχει ξεπεράσει μία ορισμένη τιμή η οποία προκύπτει πειραματικά, β) το αν η ενέργεια μετά την εκτέλεση των διεργασιών έχει την τιμή NaN (Not a Number) λόγω κάποιας υπερχείλισης και γ) NaN στην θέση κάποιου σωματιδίου.

```
/************************************************
 * Result check function for position
 * and velocity update task of MD
 ************************************************/

/* Run tasks */
run_tasks(...);

/* Insert faults in tasks that were */
/* to run over unreliable hardware  */
inject_faults(...);

for (int k = 0; k < particlesnumber; ++k){
  check_sum_vel += fabs(new_vel[k].x - reference_vel[k].x) + fabs(new_vel[k].y - reference_vel[k].y) + fabs(new_vel[k].z - reference_vel[k].z);

  /* Check position for NaN */
  if(new_pos[k].x != new_pos[k].x || new_pos[k].y != new_pos[k].y || new_pos[k].z != new_pos[k].z){
      error_in_pos=1;
    }
}

/* Calculate the energy of the system  */
/* after the calculations and fault    */
/* fault injection                     */
check_energy_after=calculate_system_energy();

if( check_sum_vel > 10 || (check_energy_after != check_energy_after) || error_in_pos ){
    restore();
}
```

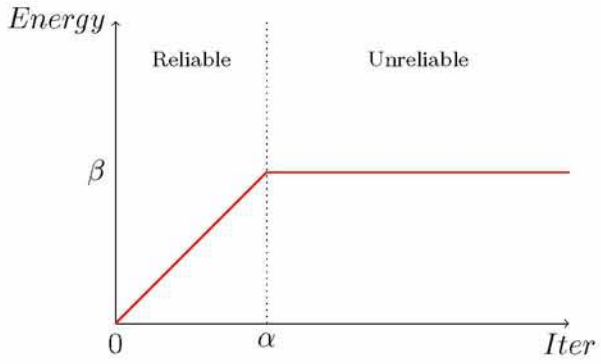
Αλγόριθμος 3.5: MD - Συνάρτηση ελέγχου λαθών για position και velocity των σωματιδίων

Για τον έλεγχο των υπόλοιπων διεργασιών (Αλγόριθμος 3.6) που υπολογίζουν τη δύναμη, το δυναμικό και την ενέργεια των σωματιδίων υπολογίζουμε την ενέργεια του συστήματος πριν την εκτέλεση των διεργασιών ($check\_energy\_before$), την ενέργεια του συστήματος μετά την εκτέλεση των διεργασιών ($check\_energy\_after$) καθώς και το άθροισμα απόλυτων διαφορών δύναμης των σωματιδίων ($check\_sum\_force$) το οποίο υπολογίζεται αντίστοιχα με το άθροισμα διαφορών ταχύτητας. Ο έλεγχος αποτελείται από έλεγχο για: α) το αν η ενέργεια έχει βγει εκτός των αναμενόμενων ορίων, β) το αν το άθροισμα απόλυτων διαφορών δύναμης έχει ξεπεράσει κάποιο όριο και γ) το αν το άθροισμα απόλυτων διαφορών δύναμης μετά την εκτέλεση των διεργασιών έχει την τιμή NaN (Not a Number) λόγω κάποιας υπερχείλισης. Τα όρια που φαίνονται στο παρακάτω τμήμα κώδικα έχουν προκύψει μετά από πειραματικές εκτελέσεις της εφαρμογής.

```
/************************************************
 * Result check function for force, potential and
 * virial update task of MD
 ************************************************/

/* Calculate the energy of the system  */
/* before the calculations             */
energy_before=calculate_system_energy();

/* Run tasks */
run_tasks(...);

/* Insert faults in tasks that were */
/* to run over unreliable hardware  */
inject_faults(...);

/* Calculate the energy of the system  */
/* after the calculations and fault    */
/* injection                          */
energy_after=calculate_system_energy();

for (k = 0; k < particlesnumber; ++k){
  check_sum_force += fabs(new_force[k].x - reference_force[k].x) + fabs(new_force[k].y - reference_force[k].y) + fabs(new_force[k].z - reference_force[k].z);
}

if( (fabs(check_energy_before-check_energy_after) > 2*log10(particlesnumber) ) || (check_sum_force > 7*log10(particlesnumber)) || (check_sum_force != check_sum_force ) )
{
    restore();
}
```

Αλγόριθμος 3.6: MD - Συνάρτηση ελέγχου λαθών για force, potential και virial των σωματιδίων

Η ενέργεια του συστήματος μεταβάλλεται σύμφωνα με τη γραφική παράσταση που φαίνεται παρακάτω (Σχήμα 3.4). Από την αρχή της προσομοίωσης μέχρι το σημείο α ($100^{η}$ επανάληψη) η ενέργεια του συστήματος αυξάνεται, ενώ στη συνέχεια μένει σταθερή γύρω από μία τιμή β έως το τέλος της προσομοίωσης. Μέχρι να σταθεροποιηθεί η ενέργεια του συστήματος εκτελούνται αξιόπιστα οι διεργασίες της εφαρμογής ενώ αφού σταθεροποιηθεί η ενέργεια ξεκινά η εισαγωγή σφαλμάτων. Εφόσον η ενέργεια σταθεροποιείται από ένα σημείο και έπειτα, είναι γνωστή η αναμενόμενη τιμή της μετά από κάθε εκτέλεση των διεργασιών και έτσι είναι εφικτό να αναγνωριστούν τυχόντα λάθη. Ο επιπλέον έλεγχος για το άθροισμα απόλυτων διαφορών δύναμης γίνεται ώστε να μη διαδοθούν σφάλματα τα οποία δεν μπορούν να αναγνωριστούν στην τρέχουσα επανάληψη και επηρεάσουν τους υπολογισμούς σε επόμενες επαναλήψεις.



Σχήμα 3.4: MD - Γραφική αναπαράσταση της ενέργειας του συστήματος [άξονας y: ενέργεια, άξονας x: επαναλήψεις (χρόνος)]

3.5 Μηχανισμοί Προστασίας

Παρακάτω (Αλγόριθμος 3.7) φαίνεται πώς επαναφέρεται η εφαρμογή από τυχόντα λάθη που μπορεί να προκύψουν κατά τη διάρκεια εκτέλεσης του πρώτου μέρους υπολογισμών (ανανέωση των position και velocity). Σε περίπτωση που κάποιο σωματίδιο έχει βγει εκτός ορίων λόγω σφάλματος στη θέση του (position), επαναφέρεται μέσα στο οριοθετημένο πλαίσιο όταν επανεκτελεστεί η αντίστοιχη διεργασία που το περιλαμβάνει.

```
/*************************************************
 * Protection Mechanism for position and velocity
 * update task of MD
 *************************************************/
/* REDO */
for (int i = 0; i < num_of_fault_tasks; ++i){
  task1(...);
}
```

Αλγόριθμος 3.7: MD - Μηχανισμός προστασίας για position και velocity με επανεκτέλεση των διεργασιών

Σε περίπτωση σφάλματος στη δύναμη, το δυναμικό ή την ενέργεια ενός σωματιδίου ο μηχανισμός προστασίας που αναλαμβάνει να επαναφέρει την εφαρμογή (Αλγόριθμος 3.8) εξετάζει τη σημαντικότητα των διεργασιών και τις εκτελεί με φθίνουσα σειρά σημαντικότητας. Ξεκινάει δηλαδή από τις περισσότερο σημαντικές διεργασίες και καταλήγει στις λιγότερο σημαντικές. Μετά την επανεκτέλεση μίας σειράς από διεργασίες σημαντικότητας x, ο μηχανισμός ελέγχει αν το λάθος παραμένει. Στην περίπτωση που το λάθος διορθώθηκε μετά από την επανεκτέλεση των διεργασιών τότε η διαδικασία διακόπτεται και ο αλγόριθμος συνεχίζει. Στην περίπτωση όμως που παραμένει το σφάλμα, τότε συνεχίζει εκτελώντας τις διεργασίες που ανήκουν σε χαμηλότερης σημαντικότητας κατηγορία $(x + 1)$ και αφού τις εκτελέσει ελέγχει ξανά για λάθη. Η διαδικασία αυτή συνεχίζεται μέχρι να διορθωθεί το λάθος ή να εκτελεστούν όλες οι διεργασίες (που επίσης αντιστοιχεί σε αποσφαλμάτωση της εφαρμογής).

Με αυτή τη μέθοδο επιδιώκεται να αποσφαλματωθούν τα αποτελέσματα σε πρώιμο στάδιο της διαδικασίας επανεκτέλεσης, καταναλώνοντας όσο το δυνατόν λιγότερο χρόνο υπολογισμού.

```
1  /************************************************
2   * Protection Mechanism for force, potential
3   * and virial update task of MD
4   ************************************************/
5  for (int priority = 1; priority < 3; ++priority){
6    for (int i = 0; i < num_of_fault_tasks; ++i){
7      //FT stands for Faulted task
8      if (FT[i].priority == priority){
9        /* Calculate current_block and  */
10       /* interactive_block indexes    */
11       int cur_block = ((int)((FT[i])/num_of_blocks));
12       int inter_block = ((FT[i])%num_of_blocks);
13
14       /* Redo Task */
15       task2(cur_block, inter_block, ...) ;
16     }
17   }
18
19   /* Check if error remains */
20   energy_after=calculate_system_energy();
21
22   check_sum_force = ...;
23
24   if( !contition ){
25       break;
26   }
27 }
```

Αλγόριθμος 3.8: MD - Μηχανισμός προστασίας για force, potential και virial με επανεκτέλεση των διεργασιών

Η συνθήκη *condition* και ο υπολογισμός του *check_sum_force* είναι ίδια με αυτά που παρουσιάστηκαν προηγουμένως (Αλγόριθμος 3.6, γραμμή 21 και γραμμή 25 αντίστοιχα).

3.6 Πειράματα και αποτελέσματα

Τα πειράματα εκτελέστηκαν σε μηχανήματα πανομοιότυπης τεχνολογίας με λειτουργικό σύστημα/kernel: Linux version 4.1.31-30-default (geeko@buildhost) (gcc version 4.8.5 (SUSE Linux)) #1 SMP PREEMPT Wed Aug 24 06:20:09 UTC 2016, επεξεργαστή: Intel Xeon E5606, 2.13 GHz, 8 MB cache, 1066 MHz memory, Quad-Core , RAM: 6GB. Το compile του κώδικα της εφαρμογής έγινε με τον compiler: gcc (SUSE Linux) 4.8.5 και για βελτιστοποιήσεις το flag: *O3*.

3.6.1 Περαματική διαδικασία

Για την αξιολόγηση των μηχανισμών ελέγχου λαθών και αποσφαλμάτωσης της εφαρμογής εκτελούνται 10.000 πειράματα για κάθε διαφορετική περίπτωση ποσοστού σφαλμάτων. Οι περιπτώσεις που επιλέγονται αναφέρονται στο α) 25% β) 50% γ) 75% και δ) 100% των συνολικών διεργασιών που έχουν επιλεχθεί για εισαγωγή σφαλμάτων. Ο αριθμός των σφαλμάτων ανέρχεται στα 10 σφάλματα ανά διεργασία.

Τα πειράματα που εκτελούνται αφορούν 4.096 σωματίδια σε κυβικό χώρο συνολικού όγκου $200 * 200 * 200$ και με όγκο ενός block $100 * 100 * 100$. Σύμφωνα με την ανάλυση της ενότητας 3.2 η εφαρμογή αποτελείται από 128 διεργασίες σε κάθε επανάληψη και ο αριθμός των συνολικών επαναλήψεων ανέρχεται στις 501 επαναλήψεις. Από τις 501 συνολικά επαναλήψεις οι 101 πρώτες περιέχουν διεργασίες που εκτελούνται αξιόπιστα και οι 400 επόμενες διεργασίες που εκτελούνται αναξιόπιστα. Στις 400 αυτές επαναλήψεις εισάγονται σφάλματα σε διεργασίες πλήθους αντίστοιχου με το ποσοστό σφαλμάτων και οι διεργασίες επιλέγονται τυχαία σε κάθε επανάληψη χρησιμοποιώντας τον αλγόριθμο του Knuth [24], αλγόριθμος ο οποίος πρόκειται για την επιλογή μίας σειράς από μοναδικούς τυχαίους αριθμούς.

Παρακάτω αναγράφονται οι τιμές που καταγράφονται από την εκτέλεση των πειραμάτων για την εξαγωγή στατιστικών δεδομένων:

- Αποτέλεσμα (pressure)
- Χρόνος αξιόπιστης εκτέλεσης
- Χρόνος αναξιόπιστης εκτέλεσης
- Χρόνος αναγνώρισης σφαλμάτων
- Χρόνος αποσφαλμάτωσης
- Χρόνος εισαγωγής σφαλμάτων
- Αριθμός λανθασμένων υποδείξεων παρουσίας λάθους (false positives errors)
- Αριθμός λανθασμένων υποδείξεων απουσίας λάθους (false negatives errors)
- Αριθμός σιωπηλά διαδοθέντων σφαλμάτων (missed errors)

Ο χρόνος αξιόπιστης εκτέλεσης αναφέρεται στις 101 πρώτες επαναλήψεις του αλγορίθμου ενώ ο χρόνος αναξιόπιστης εκτέλεσης αναφέρεται στις 400 επόμενες επαναλήψεις. Ανεξάρτητα από το ποσοστό των διεργασιών που επιλέγονται για αναξιόπιστη εκτέλεση, ο χρόνος αναξιόπιστης εκτέλεσης εμπεριέχει και τον χρόνο εκτέλεσης των διεργασιών που δεν επιλέχθηκαν και εκτελέστηκαν αξιόπιστα. Επιπλέον, στον χρόνο αναξιόπιστης εκτέλεσης δεν προσμετρώνται οι χρόνοι: α) εισαγωγής σφαλμάτων, β) ελέγχου λαθών, γ) διόρθωσης λαθών και δ) απόδοσης σημαντικότητας.

Με το τέλος της εκτέλεσης της εφαρμογής χαρακτηρίζεται η αξιοπιστία του μηχανισμού ελέγχου λαθών, η οποία μπορεί να ανήκει σε μία από τις τρεις παρακάτω περιπτώσεις:

α) Λανθασμένη υπόδειξη παρουσίας λάθους: η περίπτωση που με απενεργοποιημένο τον μηχανισμό αποσφαλμάτωσης και ενεργοποιημένο τον μηχανισμό αναγνώρισης λαθών, αναγνωρίστηκε ότι πρέπει να διορθωθούν λάθη και ενώ δεν έγινε η επιδιόρθωση, το τελικό αποτέλεσμα της εκτέλεσης ήταν αποδεκτό.

β) Λανθασμένη υπόδειξη απουσίας λάθους: η περίπτωση που με απενεργοποιημένο τον μηχανισμό αποσφαλμάτωσης και ενεργοποιημένο τον μηχανισμό αναγνώρισης λαθών, αναγνωρίστηκε ότι δεν πρέπει να διορθωθούν λάθη, το τελικό αποτέλεσμα της εκτέλεσης ήταν μη αποδεκτό.

γ) Σιωπηλά διαδοθέν σφάλμα: η περίπτωση που με ενεργοποιημένο τον μηχανισμό αποσφαλμάτωσης και ενεργοποιημένο τον μηχανισμό αναγνώρισης λαθών, αναγνωρίστηκε ότι πρέπει να διορθωθούν λάθη και το τελικό αποτέλεσμα της εκτέλεσης ήταν μη αποδεκτό.

Επιπλέον μπορούμε να εξάγουμε τα εξής:

$$\#UnRel\_tasks = \#faulted\_tasks\_per\_iter * \#UnRel\_iterations$$
$$\#Rel\_tasks = \#total\_tasks - \#UnRel\_tasks$$
$$\#Total\_faults = 10 * \#UnRel\_tasks$$

Όπου

- $\#UnRel\_tasks$ είναι ο αριθμός των διεργασιών που εκτελούνται αναξιόπιστα
- $\#Rel\_tasks$ είναι ο αριθμός των διεργασιών που εκτελούνται αξιόπιστα
- $\#faulted\_tasks\_per\_iter$ είναι ο αριθμός των διεργασιών που εκτελούνται αναξιόπιστα ανά επανάληψη
- $\#UnRel\_iterations$ είναι ο αριθμός των επαναλήψεων που εμπεριέχουν διεργασίες οι οποίες εκτελούνται αναξιόπιστα
- $\#total\_tasks$ είναι ο αριθμός των συνολικών διεργασιών της εφαρμογής
- $\#Total\_faults$ είναι ο αριθμός των συνολικών σφαλμάτων που εισήχθησαν στην εφαρμογή

Το πλήθος των συνολικών διεργασιών για το μέγεθος του block που δίνεται παραπάνω είναι ίσο με $\frac{128\ tasks}{iteration} * 501\ iterations = 64.128\ tasks$. Οι 400 από τις 501 επαναλήψεις εμπεριέχουν διεργασίες που μπορεί να εκτελεστούν σε αναξιόπιστο υλικό, επομένως για τις τέσσερις διαφορετικές περιπτώσεις ποσοστού λαθών προκύπτει το παρακάτω πλήθος αναξιόπιστων διεργασιών:

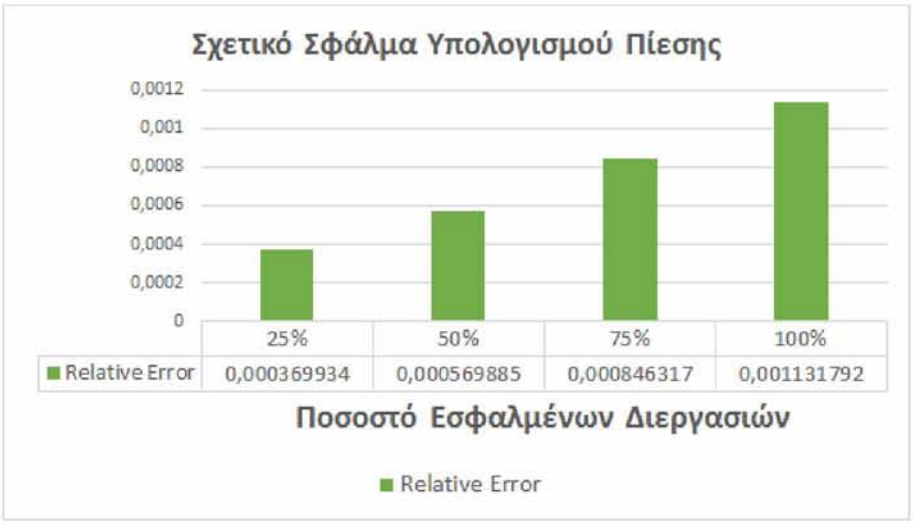
- $25\% : 25\% * 400 * 128 = 12.800$ αναξιόπιστες διεργασίες
- $50\% : 50\% * 400 * 128 = 25.600$ αναξιόπιστες διεργασίες
- $25\% : 75\% * 400 * 128 = 38.400$ αναξιόπιστες διεργασίες
- $100\% : 100\% * 400 * 128 = 51.200$ αναξιόπιστες διεργασίες

Σχήμα 3.5: MD - Ποσοστό διεργασιών που εκτελέστηκαν αναξιόπιστα επί του συνολικού αριθμού αναξιόπιστων διεργασιών διεργασιών (UnRel)

Σχήμα 3.6: MD - Ποσοστό διεργασιών που εκτελέστηκαν αναξιόπιστα επί του συνολικού αριθμού διεργασιών (UnRel + Rel)

3.6.2 Αποτελέσματα

Η πρώτη γραφική παράσταση (Σχήμα 3.7) δείχνει την ποιότητα του αποτελέσματος για διαφορετικές τιμές ποσοστού σφαλμάτων σύμφωνα με τη μετρική του σχετικού σφάλματος: $\delta x = \left|\frac{x-x0}{x0}\right|$, όπου x είναι το αποτέλεσμα της εκτέλεσης σε αναξιόπιστο υλικό, ενώ $x0$ είναι το αποτέλεσμα της αξιόπιστης εκτέλεσης.



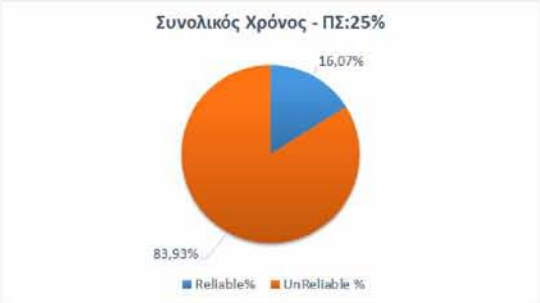
Σχήμα 3.7: MD - Σχετικό σφάλμα

Είναι εμφανές ότι και στις τέσσερις περιπτώσεις το τελικό αποτέλεσμα είναι αρκετά καλό, με το μεγαλύτερο σχετικό σφάλμα $\delta x \approx 0.0011$ να εμφανίζεται στην περίπτωση που εισάγονται τα περισσότερα σφάλματα.

Στην συνέχεια φαίνεται ο συνολικός χρόνος εκτέλεσης για κάθε περίπτωση ποσοστού σφαλμάτων όπως και της αξιόπιστης εκτέλεσης (Σχήμα 3.8).

Σχήμα 3.8: MD - Συνολικός χρόνος εκτέλεσης

Όπως φαίνεται από το παραπάνω σχήμα, με την αύξηση του ποσοστού σφαλμάτων παρατηρείται γραμμική αύξηση στον χρόνο εκτέλεσης λόγω της επανεκτέλεσης των διεργασιών σε περιπτώσεις λάθους.



(α΄) Ποσοστό εσφαλμένων διεργασιών 25%

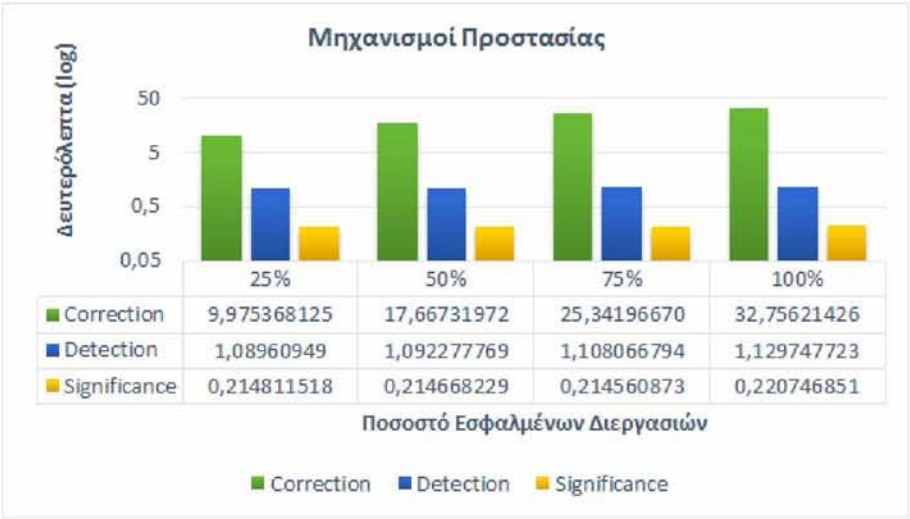
(β΄) Ποσοστό εσφαλμένων διεργασιών 50%

(γ΄) Ποσοστό εσφαλμένων διεργασιών 75%

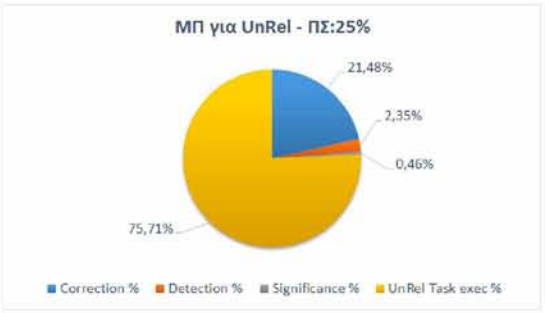
(δ΄) Ποσοστό εσφαλμένων διεργασιών 100%

Σχήμα 3.9: MD - Χρόνοι αξιόπιστης και αναξιόπιστης εκτέλεσης για κάθε ποσοστό σφαλμάτων

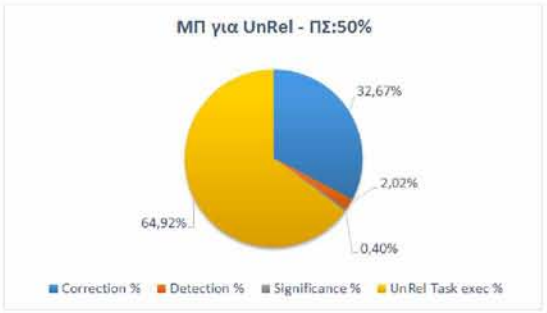
Στα παραπάνω σχήματα (Σχήμα 3.9) φαίνονται τα ποσοστά που καταλαμβάνουν η αξιόπιστη και η αναξιόπιστη εκτέλεση επί του συνολικού χρόνου κάθε περίπτωσης. Η μερίδα του λέοντος ανήκει σε όλες τις περιπτώσεις στον χρόνο αναξιόπιστης εκτέλεσης.



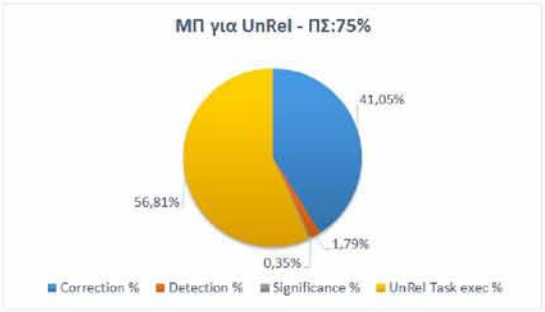
Σχήμα 3.10: MD - Χρόνοι μηχανισμών προστασίας



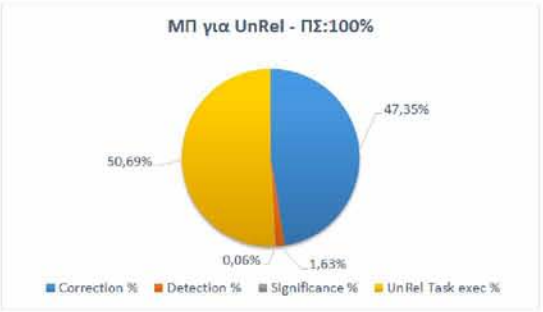
(α′) Ποσοστό εσφαλμένων διεργασιών 25%



(β′) Ποσοστό εσφαλμένων διεργασιών 50%



(γ′) Ποσοστό εσφαλμένων διεργασιών 75%



(δ′) Ποσοστό εσφαλμένων διεργασιών 100%

Σχήμα 3.11: MD - Ανάλυση χρόνου εκτέλεσης για τις τέσσερις περιπτώσεις ποσοστού σφαλμάτων σχετικά με τον χρόνο αναξιόπιστης εκτέλεσης

Στη γραφική παράσταση (Σχήμα 3.10) φαίνεται ο χρόνος που καταλαμβάνει κάθε ρουτίνα του μηχανισμού προστασίας σε λογαριθμική κλίμακα. Συγκεκριμένα αναγράφονται οι χρόνοι α) αναγνώρισης λαθών , β) επιδιόρθωσης λαθών και γ) απόδοσης σημαντικότητας, για κάθε περίπτωση ποσοστού σφαλμάτων.

Αντίστοιχα, στις γραφικές παραστάσεις (Σχήμα 3.11) φαίνονται τα ποσοστά που καταλαμβάνουν οι μηχανισμοί προστασίας και η εκτέλεση των αναξιόπιστων διεργασιών, σε σχέση με τον συνολικό χρόνο αναξιόπιστης εκτέλεσης. Ο συνολικός χρόνος αναξιόπιστης εκτέλεσης είναι το άθροισμα του χρόνου εκτέλεσης αναξιόπιστων διεργασιών, του χρόνου ελέγχου λαθών, του χρόνου διόρθωσης λαθών και του χρόνου απόδοσης σημαντικότητας.

Όπως φαίνεται από τα σχήματα που προηγούνται, μεγάλος μέρος της συνολικής εκτέλεσης καταλαμβάνει η λειτουργία της επιδιόρθωσης των σφαλμάτων και αυτό οφείλεται στην επανεκτέλεση των διεργασιών σε περιπτώσεις λάθους. Πιο συγκεκριμένα, για μεγάλο ποσοστό σφαλμάτων, ο χρόνος επιδιόρθωσης λαθών είναι περίπου ίσος με τον χρόνο εκτέλεσης των αναξιόπιστων διεργασιών, κάτι που σημαίνει πως επανεκτελούνται σχεδόν όλες οι διεργασίες.

Συνοψίζοντας τις μετρήσεις που αφορούν την ανάλυση των χρόνων εκτέλεσης, οι γραφικές παραστάσεις (Σχήμα 3.12) δείχνουν το ποσοστό του συνολικού χρόνου εκτέλεσης που καταλαμβάνει κάθε επιμέρους λειτουργία του αλγορίθμου της εφαρμογής για κάθε ποσοστό σφαλμάτων.

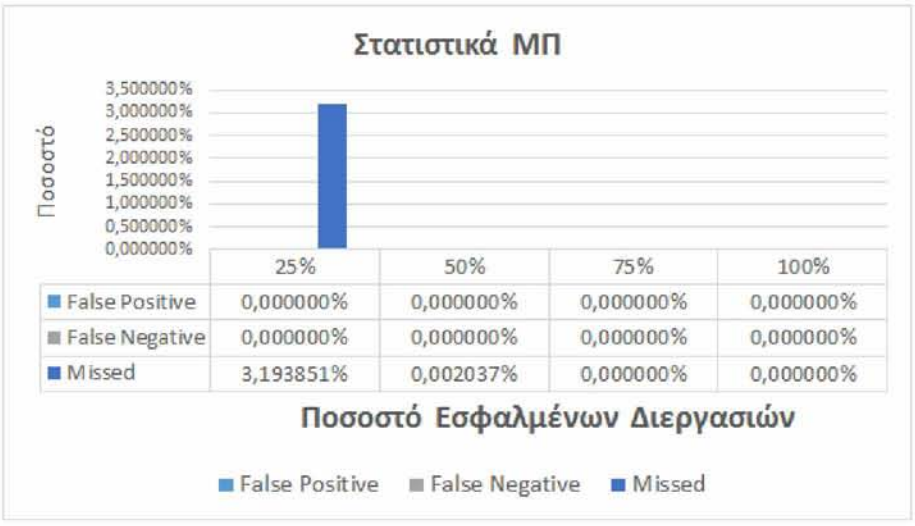
(α′) Ποσοστό εσφαλμένων διεργασιών 25%

(β′) Ποσοστό εσφαλμένων διεργασιών 50%

(γ′) Ποσοστό εσφαλμένων διεργασιών 75%

(δ′) Ποσοστό εσφαλμένων διεργασιών 100%

Σχήμα 3.12: MD - Ανάλυση χρόνου εκτέλεσης για τις τέσσερις περιπτώσεις ποσοστού σφαλμάτων σχετικά με τον συνολικό χρόνο εκτέλεσης



Σχήμα 3.13: MD - Λανθασμένη υπόδειξη παρουσίας λάθους (False Positive), λανθασμένη υπόδειξη απουσίας λάθους (False Negative), Αριθμός σιωπηλά διαδοθέντων σφαλμάτων (Missed)

Στο παραπάνω σχήμα (Σχήμα: 3.13) φαίνεται η αποτελεσματικότητα των μηχανισμών προστασίας για κάθε περίπτωση ποσοστού σφαλμάτων. Αναγράφονται τα ποσοστά για: α) λανθασμένη υπόδειξη παρουσίας λάθους, β) λανθασμένη υπόδειξη απουσίας λάθους γ) σιω-

πηλά διαδοθέν σφάλματα.

Στις παραπάνω γραφικές παραστάσεις οι χρόνοι αναξιόπιστης εκτέλεσης δεν εμπεριέχουν τον χρόνο εισαγωγής σφαλμάτων, διότι αφορά αποκλειστικά την προσομοίωση. Για λόγους πληρότητας στην επόμενη γραφική παράσταση (Σχήμα: 3.14) αναγράφονται οι χρόνοι εισαγωγής σφαλμάτων για κάθε περίπτωση ποσοστού σφαλμάτων.

Σχήμα 3.14: MD - Χρόνοι εισαγωγής σφαλμάτων

Όπως φαίνεται από τις παραπάνω γραφικές παραστάσεις, η αναξιόπιστη εκτέλεση αποτελεί το μεγαλύτερο μέρος της συνολικής εκτέλεσης της εφαρμογής. Οι διεργασίες που επιλέγονται για αναξιόπιστη εκτέλεση ξεπερνούν σε αριθμό τις διεργασίες που επιλέγονται για αξιόπιστη εκτέλεση και αυτό δικαιολογεί το ποσοστό του συνολικού χρόνου εκτέλεσης που καταλαμβάνεται από την αναξιόπιστη εκτέλεση.

Το επιπλέον υπολογιστικό τμήμα της απόδοσης σημαντικότητας καταλαμβάνει ελάχιστο χρόνο σε σχέση με τον συνολικό χρόνο εκτέλεσης και ουσιαστικά δεν επιβαρύνει την εκτέλεση της εφαρμογής. Το ίδιο ισχύει και για το τμήμα ελέγχου λαθών, όπου επίσης δεν έχει μεγάλο υπολογιστικό κόστος. Αντίθετα, το επιπλέον υπολογιστικό τμήμα της διόρθωσης λαθών καταλαμβάνει ένα μεγάλο μέρος του συνολικού χρόνου εκτέλεσης και αυτό οφείλεται στην επανεκτέλεση των διεργασιών.

3.6.3 Μελλοντικές επεκτάσεις

Η διαδικασία αποσφαλμάτωσης απαιτεί μεγάλο υπολογιστικό κόστος και αποτελεί κομβικό σημείο για τη μείωση του χρόνου εκτέλεσης σε μεγάλα ποσοστά σφαλμάτων. Ο λόγος για τον οποίο είναι χρονοβόρο αυτό το τμήμα της εφαρμογής είναι η επανεκτέλεση των διεργασιών, που σε ορισμένες περιπτώσεις είναι πιθανό να εκτελεστούν όλες οι διεργασίες μίας επανάληψης για δεύτερη φορά. Σε αυτό το τμήμα μπορούν να μελετηθούν ρουτίνες αποσφαλμάτωσης που, αντί για επανεκτέλεση των διεργασιών, κάνουν οπισθοδρόμηση στα δεδομένα σε μία προηγούμενη κατάσταση η οποία είναι αναμφισβήτητα ορθή. Με αυτόν τον τρόπο, δύναται να

μειωθεί ο χρόνος εκτέλεσης της διαδικασίας αποσφαλμάτωσης σε μεγάλο βαθμό, θα πληρωθεί όμως το αντίστοιχο κόστος στην ακρίβεια των αποτελεσμάτων.

Τέλος, μπορούν να γίνουν δοκιμές με διαφορετικά μεγέθη block όπου θα αυξήσουν μεν τη συνολική εκτέλεση του αξιόπιστου και του αναξιόπιστου τμήματος, δύναται να μειώσουν δε το χρόνο αποσφαλμάτωσης με επανακτέλεση των διεργασιών.

Κεφάλαιο 4

Small Path Tracer - SmallPt

4.1 Περιγραφή

Η εφαρμογή SmallPt (Small Path Tracer) ανήκει στην κατηγορία της δημιουργίας ψηφιακής φωτορεαλιστικής εικόνας με τη χρήση Η/Υ (rendering). Δεδομένης της περιγραφής μίας σκηνής που καθορίζει την τοποθεσία των επιφανειών σε μία σκηνή, τη θέση των φωτεινών πηγών και την τοποθέτηση της κάμερας, δημιουργείται μία ψηφιακή εικόνα αυτής της σκηνής. Η δημιουργία της σκηνής γίνεται με σφαίρες διαφορετικού μεγέθους, χρώματος και υλικού. Τα υλικά που υποστηρίζονται είναι τα εξής: διάχυτο (δηλαδή όχι λαμπερό), καθρέφτης και γυαλί. Η εφαρμογή είναι υλοποίηση του David Bucciarelli [25], η οποία είναι βασισμένη στην υλοποίηση του Kevin Beason [26]

Η δημιουργία της εικόνας μπορεί να γίνει με τη χρήση είτε της μεθόδου Path Tracing (έμμεσος φωτισμός) είτε της μεθόδου Direct Lighting (άμεσος φωτισμός). Και οι δύο είναι αμερόληπτες[1] Monte Carlo μέθοδοι, με τη διαφορά ότι έχουν διαφορετικό αποτέλεσμα, διαφορετική πολυπλοκότητα και χρόνο υπολογισμού οπότε και αναφέρονται σε διαφορετικές εφαρμογές. Οι περιγραφές των μεθόδων δίνονται παρακάτω.

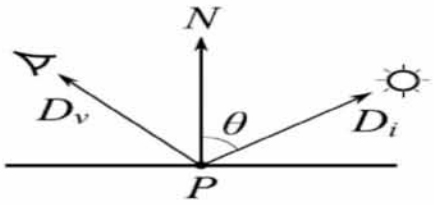
Path Tracing: για κάθε pixel της εικόνας στέλνεται μία ακτίνα προς τη σκηνή. Όταν χτυπήσει η ακτίνα σε μία επιφάνεια, τότε σημειώνεται το χρώμα της επιφάνειας και η κατεύθυνση της ακτίνας αφού αναπηδήσει στην επιφάνεια (Σχήμα 4.1β′). Στη συνέχεια η ακτίνα αναπηδά και συνεχίζει μέχρι να φτάσει σε φωτεινή πηγή (υπάρχει όριο στις αναπηδήσεις σε περίπτωση που δε φτάσει ποτέ σε φωτεινή πηγή). Στο τέλος όλες οι πληροφορίες που συνέλεξε η ακτίνα επιστρέφονται προς τα πίσω, βήμα προς βήμα. Σε κάθε βήμα κλιμακώνεται το φως από ορισμένους παράγοντες μέχρι μία τελική τιμή να φτάσει στο pixel. Επαναλαμβάνοντας την όλη διαδικασία αρκετές φορές με την προσθήκη τυχαιότητας και βρίσκοντας το μέσο όρο από όλα τα αποτελέσματα, σιγά σιγά αρχίζει να αναδύεται μία εικόνα όπου με κάθε επανάληψη που περνάει αρχίζει να βελτιώνεται. Η στρατηγική αυτή είναι ουσιαστικά μία τυφλή αναζήτηση

[1]Η Monte Carlo μέθοδος χαρακτηρίζεται ως αμερόληπτη δηλώνοντας την έλλειψη συνεπών σφαλμάτων (το μόνο σφάλμα πηγάζει από την επίδραση της τυχαιότητας). Αυτή η λεπτότητα είναι κατάλληλη για τα μαθηματικά αλλά στην πράξη όλες οι υλοποιήσεις είναι μεροληπτικές και σκοπός είναι να δομηθούν έτσι ώστε να είναι αποδεκτές

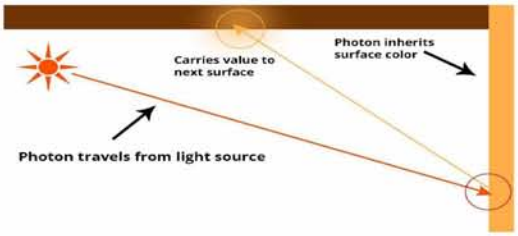
φωτός. Αν μία ακτίνα δε χτυπήσει μία πηγή φωτός τότε θα επιστρέψει μαύρο χρώμα. Για εξωτερικές σκηνές οι περισσότερες ακτίνες θα αναπηδήσουν προς τον ουρανό και θα βρουν μία φωτεινή πηγή, αλλά για εσωτερικούς χώρους οι πηγές φωτός είναι μικρές και χρειάζονται πολλές ακτίνες μέχρι να βρεθούν. Αυτή η μέθοδος είναι ιδανική για τις περιπτώσεις όπου θέλουμε η τελική εικόνα να είναι πιο κοντά στην πραγματικότητα, προσδίδοντας αρκετά ρεαλιστικό φωτισμό χωρίς να μας ενδιαφέρει ο χρόνος που θα πάρει για να δημιουργηθεί.

Direct Lighting: για κάθε pixel της εικόνας στέλνεται μία ακτίνα προς τη σκηνή. Αν η ακτίνα χτυπήσει σε μία επιφάνεια, τότε σημειώνεται το χρώμα της επιφάνειας και η κατεύθυνση της ακτίνας αφού αναπηδήσει στην επιφάνεια. Στη συνέχεια, η ακτίνα αναπηδά και συνεχίζει μέχρι να φτάσει σε φωτεινή πηγή χωρίς να κάνει άλλη αναπήδηση σε κάποια επιφάνεια. Άρα περιορίζεται σε δύο περιπτώσεις: α) η ακτίνα χτυπά πρώτα σε φωτεινή πηγή και επιστρέφει, β) η ακτίνα χτυπά μία φορά σε κάποια επιφάνεια και αν η επόμενη επιφάνεια είναι φωτεινή πηγή τότε επιστρέφει με την πληροφορία που συνέλεξε, διαφορετικά δε λαμβάνεται υπόψη. Επαναλαμβάνοντας την όλη διαδικασία αρκετές φορές με την προσθήκη τυχαιότητας και βρίσκοντας τον μέσο όρο από όλα τα αποτελέσματα, σιγά σιγά αρχίζει να αναδύεται μία εικόνα όπου με κάθε επανάληψη που περνάει αρχίζει να βελτιώνεται. Αυτή η μέθοδος είναι ιδανική για εφαρμογές που απαιτούν γρήγορα αποτελέσματα (video games), αλλά δεν προσφέρει τη βέλτιστη εικόνα.

Η μέθοδος που επιλέγεται σε αυτήν την εργασία είναι η Path Tracing και η εξίσωση που πρέπει να λυθεί σύμφωνα με αυτή τη μέθοδο για τη δημιουργία της εικόνας περιγράφεται και αναλύεται παρακάτω.



(α′) Ανάκλαση ακτίνας



(β′) Ακτίνα που κληρονομεί το χρώμα της επιφάνειας από την οποία ανακλάται [27]

Σχήμα 4.1: SmallPt - Συμπεριφορά και ιδιότητες ακτίνας

$$L(P \to D_v) = L_e(P \to D_v) + \int_\Omega F_s(D_v, D_i)|cos\theta|L(Y_i \to -D_i)dDi \tag{4.1}$$

$L(P \to D_v)$: είναι η ακτινοβολία (ένταση του φωτός) που προέρχεται από το σημείο P με κατεύθυνση προς το D_v. Αυτός είναι ο όρος που πρέπει να υπολογιστεί.

$L_e(P \to D_v)$: είναι η ακτινοβολία που εκπέμπει το ίδιο το σημείο P (ισούται 0, εκτός εάν το σημείο P είναι πηγή φωτός). Αυτό μπορεί να ελεγχθεί από την περιγραφή της σκηνής.

$\int_\Omega F_s(D_v, D_i)|cos\theta|L(Y_i \to -D_i)dDi$: αυτός ο όρος αναφέρεται στο φως που αντανακλάται από το σημείο P με κατεύθυνση προς το σημείο D_v. Εδώ προστίθεται όλο το φως που μπαίνει στο σημείο P από όλες τις κατευθύνσεις, διαμορφωμένο από την πιθανότητα να

διαχέεται προς την κατεύθυνση D_v (βασισμένο στη BRDF συνάρτηση F_s [28]).

Η προσέγγιση που γίνεται με τη χρήση της μεθόδου Monte Carlo μετατρέπει την εξίσωση ως εξής:

$$L(P \rightarrow D_v) = L_e(P \rightarrow D_v) + \frac{F_s(D_v, D_i)|cos\theta|\hat{L}(Y_i \rightarrow -D_i)}{p^{tot}_{angle}(D_i)} \tag{4.2}$$

Στην παραπάνω εξίσωση έχει αντικατασταθεί το ολοκλήρωμα με ένα Monte Carlo δείγμα, το οποίο έχει την ίδια μέση τιμή και είναι διαιρεμένο με τον αριθμό των δειγμάτων για κάθε pixel ώστε να δημιουργηθεί μία ομαλή εικόνα.

4.2 Ορισμός διεργασιών

Αρχικά, ο σειριακός υπολογισμός των χρωμάτων κάθε pixel γίνεται προσπελάζοντας ένα προς ένα τα pixel της εικόνας από αριστερά προς τα δεξιά και από πάνω προς τα κάτω, κάνοντας τους απαραίτητους υπολογισμούς για να υπολογιστεί ο τελικός χρωματισμός του καθενός (Αλγόριθμος 4.1).

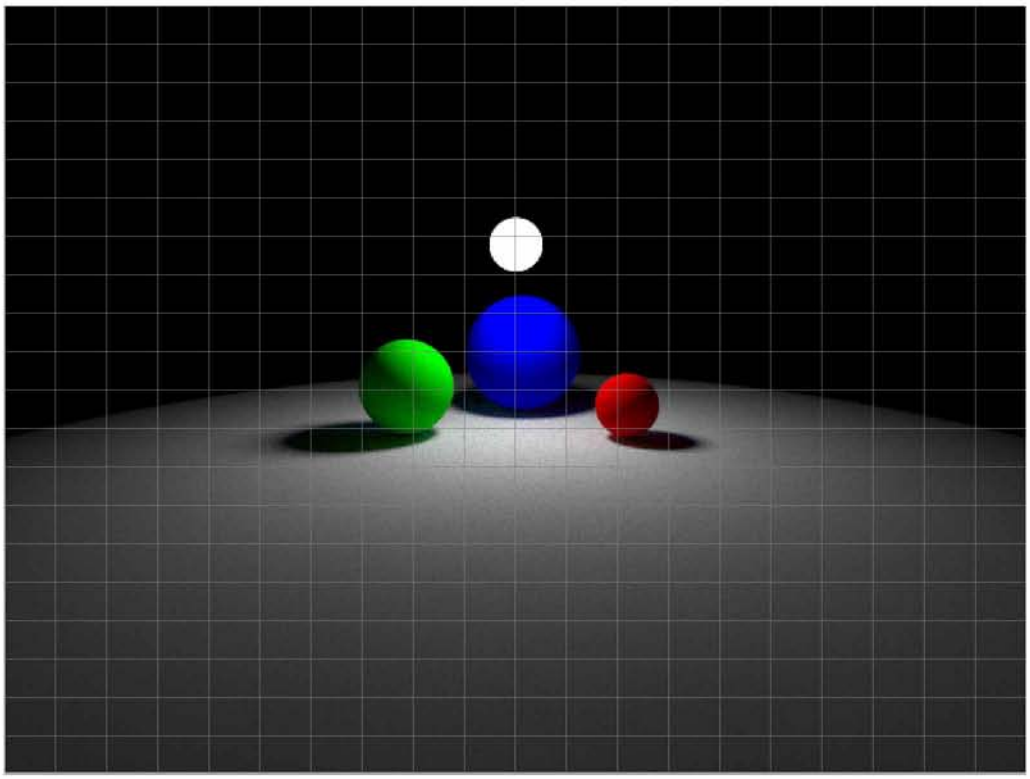
```
1 /************************************************
2  * Calculation of the colors (RGB) of each pixel
3  ************************************************/
4 for (y = 0; y < height; y++) { // Loop over image rows
5   for (x = 0; x < width; x++) { // Loop cols
6         int i = y * width + x;  //index of pixel
7         calculate_colors_of_pixel(i);
8   }
9 }
```

Αλγόριθμος 4.1: SmallPt - Υπολογισμών των χρωμάτων RGB κάθε pixel

Για την ανάθεση των υπολογισμών της εφαρμογής SmallPt σε διεργασίες και την αντιστοίχιση αυτών με μία κλίμακα σημαντικότητας ως προς το τελικό αποτέλεσμα, ο αλγόριθμος αναδομήθηκε ώστε οι υπολογισμοί για κάθε pixel της εικόνας να γίνονται σε block. Ο τρόπος υπολογισμού δεν αλλάζει σε κανένα σημείο, παρά μόνο στη σειρά που υπολογίζονται οι χρωματισμοί των pixel. Πλέον υπολογίζονται τα χρώματα των pixel που ανήκουν σε ένα block μαζί, άσχετα με την τοποθεσία τους στην εικόνα. Αυτός ο υπολογισμός για τα χρώματα των pixel ενός block αποτελεί πλέον μία διεργασία. Η διαμέριση αυτή έγινε με στόχο να μπορεί να μελετηθεί η ομοιογένεια των χρωμάτων μέσα σε ένα block για την απόδοση σημαντικότητας στις διεργασίες. Αν ένα block δεν είναι ομοιογενές (τα όρια και ο τρόπος υπολογισμού της ομοιογένειας στα χρώματα ενός block αναλύονται παρακάτω), τότε συνεπάγεται ότι εμπεριέχει αρκετά διαφορετικά χρώματα που πιθανόν να αποτελούν κρίσιμα σημεία στην τελική εικόνα και κατά συνέπεια επιλέγεται να γίνει αξιόπιστα η εκτέλεσή του. Όπως και στην εφαρμογή MD, έτσι και εδώ παίζει ρόλο το μέγεθος των block. Πιο συγκεκριμένα, αν είναι πολύ μεγάλο τότε μειώνεται η πιθανότητα να τρέξει η διεργασία αναξιόπιστα μιας και είναι περισσότερο πιθανό να ξεπερνάει το όριο της ομοιογένειας χρωμάτων που τίθεται. Αντίθετα, στην περίπτωση που είναι πολύ μικρό αυξάνεται η πιθανότητα εκτέλεσής του αναξιόπιστα,

αλλά ταυτόχρονα αυξάνεται και ο χρόνος εκτέλεσης του αλγορίθμου, καθώς και μειώνεται το PSNR. Η μείωση αυτή του PSNR συμβαίνει διότι υπάρχουν περισσότερα τμήματα της εικόνας που έχουν εκτελεστεί αναξιόπιστα.

Στο σχήμα (Σχήμα 4.2) φαίνεται μία τυπική τμηματοποίηση της εικόνας σε blocks. Η εικόνα έχει χωριστεί σε μικρότερα τμήματα και η κάθε διεργασία αφορά τον υπολογισμό χρωμάτων των pixel που αντιστοιχούν σε ένα block. Η απόδοση σημαντικότητας γίνεται σε κάθε block ανάλογα με τον χρωματισμό των pixel στο εσωτερικό του.



Σχήμα 4.2: SmallPt - Τμηματοποίηση εικόνας σε blocks

Παρακάτω φαίνεται ο ψευδοκώδικας (Αλγόριθμος 4.2) για τον υπολογισμό του χρωματισμού των pixel σύμφωνα με την τμηματοποίηση που αναλύεται παραπάνω.

```
/************************************************
 * Calculation of the colors (RGB) of each pixel
 * in a blocked partitioned image
 ************************************************/
for (k = 0; k < blocks_num; ++k){   //Loop blocks
    //task
    for (l = 0; l < blockPixelCount; ++l){  //Loop pixels
        i = index(block[k],pixel[l]); //index of pixel
        calculate_colors_of_pixel(i);
    }
}
```

Αλγόριθμος 4.2: SmallPt - Υπολογισμών των χρωμάτων RGB κάθε pixel με blocks

4.3 Σημαντικότητα διεργασιών

Ο προσδιορισμός της σημαντικότητας των διεργασιών (κάθε διεργασία είναι ένα block εικόνας) της εφαρμογής SmallPt γίνεται εξετάζοντας τα pixel του κάθε block. Συγκεκριμένα, από τις τιμές των χρωμάτων (σε αναπαράσταση RGB έχουμε 3 τιμές για κάθε χρώμα) του κάθε pixel εξάγουμε τη διακύμανση των χρωμάτων από τα pixel που ανήκουν σε ένα block, ώστε να έχουμε πληροφορία για την ομοιογένεια του μίγματος χρωμάτων εντός του block. Όσο μικρότερη είναι η διακύμανση, τόσο περισσότερο ομοιογενές είναι το μίγμα χρωμάτων εντός του block και αν δεν ξεπερνάει το όριο που τίθεται παρακάτω, επιτρέπει την εκτέλεση της διεργασίας αυτού του block πάνω από αναξιόπιστο υλικό. Η απόδοση σημαντικότητας στις διεργασίες γίνεται περιοδικά ανά 50 επαναλήψεις, επειδή η ομοιογένεια των χρωμάτων εντός των block δεν αλλάζει σημαντικά από επανάληψη σε επανάληψη, αλλά η αλλαγή παρατηρείται σε βάθος επαναλήψεων κατά της διάρκεια εκτέλεσης.

Ομοιογένεια χρωμάτων εντός του block: η αναπαράσταση των χρωμάτων ενός pixel γίνεται χρησιμοποιώντας το οκταψήφιο μοντέλο RGB [29] όπου κάθε χρώμα δημιουργείται από ένα μίγμα αποχρώσεων των χρωμάτων κόκκινο (R), πράσινο (G) και μπλε (B). Κάθε απόχρωση αυτών των τριών χρωμάτων αναπαριστάται από έναν ακέραιο αριθμό στο πεδίο $[0, 255]$ και συνδυάζοντας τις αποχρώσεις μεταξύ τους μπορούν να δημιουργηθούν $16,777,216$ πιθανά χρώματα.

Για να υπολογίσουμε την ομοιογένεια των χρωμάτων εντός του block πρέπει να αντιστοιχίσουμε τις τρεις διαστάσεις του χρωματικού μοντέλου RGB που αντιπροσωπεύουν ένα χρώμα σε μία. Αυτό το υλοποιούμε χρησιμοποιώντας την ευκλείδεια νόρμα ενός χρώματος (Εξίσωση 4.3). Στη συνέχεια, συλλέγουμε τις τιμές των χρωμάτων που ανήκουν σε ένα block και υπολογίζουμε τη διακύμανσή τους. Αν η διακύμανση βρίσκεται ανάμεσα σε κάποια όρια που προκύπτουν μετά από πειραματικές εκτελέσεις της εφαρμογής, τότε επιλέγουμε μία από τις πιθανές περιπτώσεις: α) την εκτέλεση της διεργασίας πάνω σε αξιόπιστο υλικό, β) την εκτέλεση της διεργασίας πάνω σε αναξιόπιστο υλικό και γ) την παράλειψη της εκτέλεσης της διεργασίας και τη διατήρηση των αποτελεσμάτων από προηγούμενη επανάληψη. Η τελευταία περίπτωση παρουσιάζει ενδιαφέρον, καθώς παραλείπονται οι υπολογισμοί για τον χρωματισμό κάποιων pixel και γεννάται η απορία για τον τρόπο με τον οποίο η επιλογή αυτή επηρεάζει το τελικό αποτέλεσμα. Όπως φαίνεται και σε παρακάτω παράδειγμα, υπάρχουν σημεία (ολόκληρα block) σε κάποιες εικόνες που από τις πρώτες επαναλήψεις είναι πολύ φωτεινά (φωτεινή πηγή) ή πολύ σκοτεινά (κενό). Αυτά τα σημεία αλλάζουν την απόχρωσή τους ελάχιστα κατά τη διάρκεια δημιουργίας της τελικής εικόνας, με αποτέλεσμα να είναι δυνατόν να παραλειφθούν οι υπολογισμοί που τα αφορούν και να ελαφρυνθεί έτσι ο συνολικός φόρτος εργασίας. Επιπλέον, εισάγοντας σφάλματα το αποτέλεσμα μπορεί να αποκλίνει σε μικρό βαθμό, τόσο ώστε να μη γίνει αντιληπτό από τον μηχανισμό ελέγχου σφαλμάτων. Με την παράλειψη κάποιων υπολογισμών που δεν εξάγουν διαφορετικά (δηλαδή με μικρή διαφορά) αποτελέσματα από προηγούμενες επαναλήψεις διατηρούνται κάποια μέρη της εικόνας υγιή από σφάλματα, και έτσι το τελικό αποτέλεσμα βελτιώνεται.

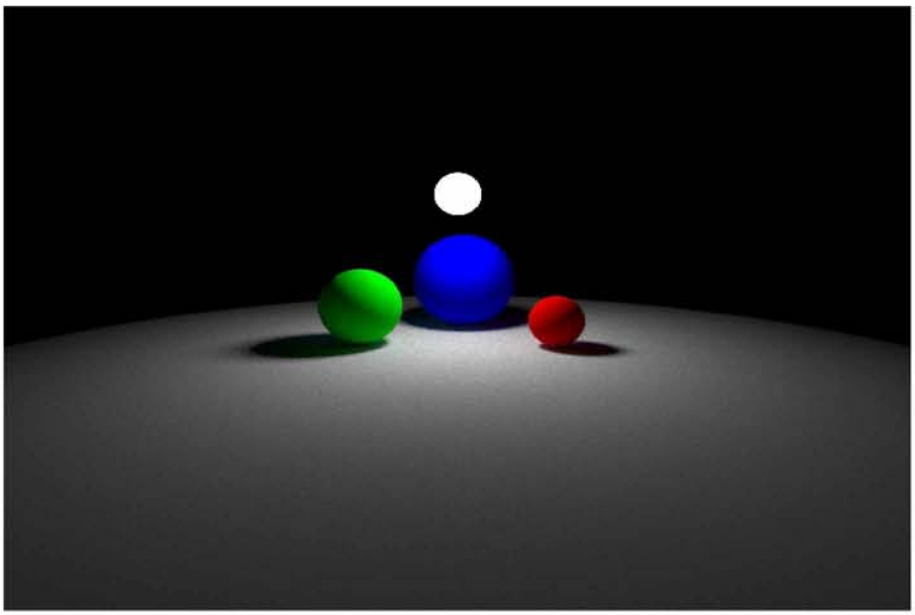
$$\|color\_norm\| = \sqrt{R^2 + G^2 + B^2} \tag{4.3}$$

```
/************************************************
 * Significance assignment based on RGB values
 ************************************************/
if ( currentSample % 50 ==0){
  float data[blockPixelCount];
  for (k=0; k<blocks_num; ++k){
    float var=0.f;
    for (l = 0; l < blockPixelCount; ++l){
      int y= block[k][l]/width;
      int x = block[k][l]%width;
      const int i = (height - y - 1) * width + x;
      data[l] = sqrt(pow(toInt(colors[i].x),2)
      + pow(toInt(colors[i].y),2)  +
      pow(toInt(colors[i].z),2) );
    }
    calculate_variance(&var, data, blockPixelCount);
    significance[k] = (int)var;

    //reliable blocks(Paint them Red)
    if (var> significance_threshold){
      for (l = 0; l < blockPixelCount; ++l){
        int y= block[k][l]/width;
        int x = block[k][l]%width;
        const int i = (height - y - 1) * width + x;
        pixels[block[k][l]] = 255 |
                  (toInt(colors[i].y) << 8) |
                  (toInt(colors[i].z) << 16);
      }
    }

    //inject fault blocks(Paint them Green)
    if (var < significance_threshold && var>= drop_threshold){
      for (l = 0; l < blockPixelCount; ++l){
        int y= block[k][l]/width;
        int x = block[k][l]%width;
        const int i = (height - y - 1) * width + x;
        pixels[block[k][l]] = toInt(colors[i].x) |
                  (255 << 8) |
                  (toInt(colors[i].z) << 16);
      }
    }

    //drop blocks(Paint them Yellow)
    if (var < drop_threshold && drop_blocks){
      for (l = 0; l < blockPixelCount; ++l){
        int y= block[k][l]/width;
        int x = block[k][l]%width;
```

```
48          const int i = (height - y - 1) * width + x;
49          pixels[block[k][l]] = 255 |
50                      (255 << 8) |
51                      (toInt(colors[i].z) << 16);
52        }
53      }
54    }
55 }
```

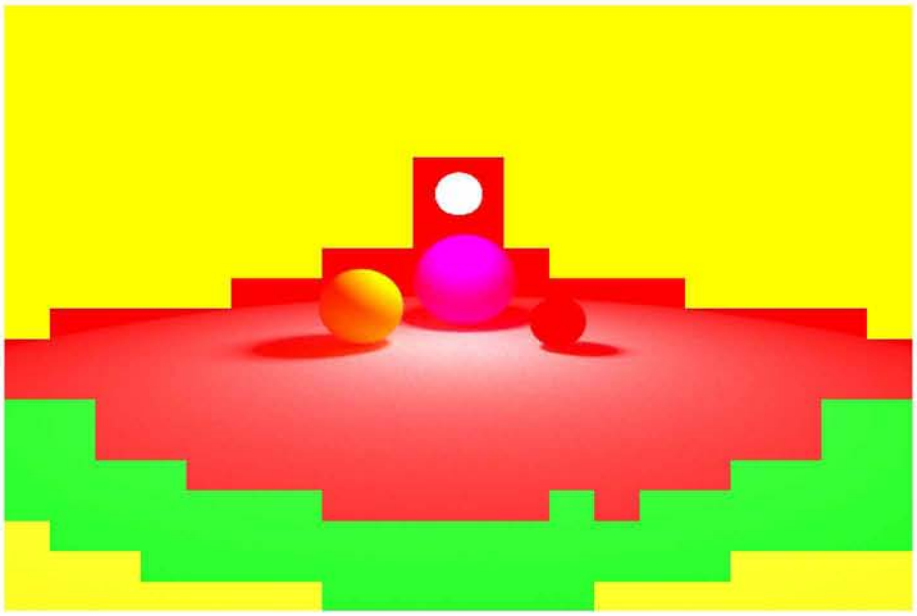
Αλγόριθμος 4.3: SmallPt - Προσδιορισμός σημαντικότητας στις διεργασίες βάσει των τιμών RGB των pixel

Στο σχήμα (Σχήμα 4.3) φαίνεται ο τρόπος με τον οποίο γίνεται η απόδοση της σημαντικότητας των block σε μία εικόνα που δημιουργείται από την εφαρμογή SmallPt. Στο σχήμα (Σχήμα 4.3α′) φαίνεται η αρχική εικόνα, ενώ στο σχήμα (Σχήμα 4.3β′) φαίνεται η εικόνα με χρωματισμένα τα block που: α) θα εκτελεστούν αξιόπιστα (κόκκινο), β) θα εκτελεστούν αναξιόπιστα (πράσινο) και γ) δε θα εκτελεστούν καθόλου (κίτρινο). Ο αλγόριθμος (Αλγόριθμος 4.3) περιγράφει τη μέθοδο απόδοσης σημαντικότητας. Οι γραμμές 23 εώς 57 έχουν προστεθεί απλά για απεικόνιση του τρόπου χρωματισμού των block της εικόνας, ανάλογα με την σημαντικότητά τους, και δεν αποτελούν μέρος της συνάρτησης απόδοσης σημαντικότητας.



(α′) Εικόνα αναφορά

Σχήμα 4.3: SmallPt - Τμηματοποίηση σε blocks και σημαντικότητα



(β′) Εικόνα με χρωματισμένα τα block σύμφωνα με τη σημαντικότητά τους

Σχήμα 4.3: SmallPt - Τμηματοποίηση σε blocks και σημαντικότητα

4.4 Έλεγχος Λαθών

Ο αλγόριθμος ελέγχου λαθών για την εφαρμογή SmallPt (Αλγόριθμος 4.4) διατρέχει όλα τα blocks και ελέγχει τη σημαντικότητά τους. Αν η σημαντικότητα ενός block είναι υψηλή, αυτό σημαίνει πως η διεργασία αυτού του block εκτελέστηκε πάνω από αξιόπιστο υλικό και επομένως δεν περιέχει λάθη (γραμμή 11). Το ίδιο ισχύει και για τα block που η σημαντικότητά τους είναι αρκετά χαμηλή, σε βαθμό που δεν εκτελέστηκαν καθόλου, οπότε δεν ελέγχονται για λάθη ούτε αυτά τα block (γραμμή 16). Σε διαφορετική περίπτωση (στην περίπτωση δηλαδή όπου εισάγονται σφάλματα) ελέγχεται το block για λάθη προσπελάζοντας τα pixel που εμπεριέχει, ελέγχοντάς τα ένα προς ένα (γραμμές 19-25).

```
1  /************************************************
2   *  Result check function for SmallPt
3   ************************************************/
4  for (k = 0; k < blocks_num; ++k){
5    /* Do not check for errors     */
6    /* if no faults were injected */
7    /* in the task */
8    if (significance[k]>=significance_threshold)
9      continue;
10
11   /* Do not check for errors  */
12   /* if the task was dropped */
13   if (significance[k]<=drop_threshold)
14     continue;
15
```

```
16   for (l = 0; l < blockPixelCount; ++l){
17     i = index_of_pixel(block[k][l]);
18     if (diff(pixel(i),reference_pixel(i))>diff_threshold)
19     {
20       restore_pixel(i);
21     }
22   }
23 }
```

Αλγόριθμος 4.4: SmallPt - Συνάρτηση ελέγχου λαθών

Παρακάτω (Αλγόριθμος 4.5) περιγράφεται ο έλεγχος που γίνεται σε κάθε pixel, ώστε να αποφασιστεί αν το χρώμα του pixel είναι λανθασμένο. Αυτό γίνεται βρίσκοντας την απόσταση των χρωμάτων του pixel με το αντίστοιχο pixel αναφοράς (*reference_pixel*) στον χρωματικό χώρο CIELAB [30]. Στη συνέχεια αναλύεται περισσότερο η διαδικασία εύρεσης της απόστασης μεταξύ δύο χρωμάτων. Αν η συνθήκη είναι αληθής σημαίνει πως κρίθηκε ότι το pixel έχει λανθασμένη τιμή και πρέπει να διορθωθεί (γραμμή 9).

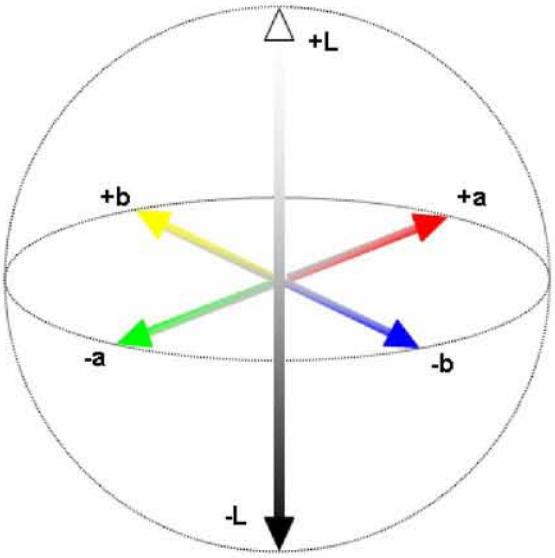
```
1 /************************************************
2  * diff(pixel(i), reference_pixel(i) ) > diff_threshold
3  * i is the index of a pixel in the image
4  ************************************************/
5 if ( RGB_color_Lab_difference_CIE94(pixel(i).red,
6   pixel(i).green, pixel(i).blue, reference_pixel(i).red,
7   reference_pixel(i).green, reference_pixel(i).blue)>0.5)
8 {
9     restore_pixel(i);
10 }
```

Αλγόριθμος 4.5: SmallPt - Συνθήκη ελέγχου λαθών

Η συνάρτηση *RGB_color_Lab_difference_CIE94* παίρνει ως ορίσματα τα χρώματα (R,G,B) από δύο pixel και επιστρέφει τη διαφορά τους στον χρωματικό χώρο CIELab (Σχήμα 4.4). Η πρόθεση του μοντέλου CIELab (ή L*a*b* ή Lab) είναι να παράγει έναν χρωματικό χώρο που είναι περισσότερο αντιληπτικά γραμμικός από άλλους χρωματικούς χώρους. Αντιληπτικά γραμμικός σημαίνει πως μία αλλαγή στην τιμή ενός χρώματος, θα πρέπει να παράγει μία αλλαγή περίπου ίδιας οπτικής σημασίας. Σε αντίθεση με το χρωματικό μοντέλο RGB, ο χρωματικός χώρος CIELab έχει σχεδιαστεί για να προσεγγίζει την ανθρώπινη όραση. Φιλοδοξεί σε αντιληπτική ομοιομορφία και η L συνιστώσα του χώρου ταιριάζει πολύ με την ανθρώπινη αντίληψη φωτεινότητας.

Για να γίνει η μετατροπή από το χρωματικό μοντέλο RGB στον χρωματικό χώρο CIELab πρέπει πρώτα να γίνει μετατροπή στον χρωματικό χώρο XYZ και μετά στον CIELab. Η διαφορά δύο χρωμάτων στον χρωματικό χώρο CIELab (σύμφωνα με τη φόρμουλα του 1994) δίνεται από την εξίσωση 4.4.

Η διεθνής επιτροπή χρωματικών χώρων και χρωμάτων (International Commission on Illumination - CIE) [31] αναφέρεται στη μετρική απόσταση χρωμάτων ως ΔE^*_{ab} (ονομάζεται επίσης και ΔE^*, dE^*, dE, ή "$DeltaE$") [32], όπου το ελληνικό γράμμα Δ χρησιμοποιείται



Σχήμα 4.4: SmallPt - Χρωματικός χώρος CIELab [3]

για να υποδηλώσει τη διαφορά και το E προέρχεται από τη λέξη 'Empfindung', που σημαίνει αίσθηση στα Γερμανικά.

Διαφορετικές μελέτες έχουν προτείνει διαφορετικές τιμές ΔE που αντιπροσωπεύουν την μόλις αισθητή διαφορά ανάμεσα σε δύο χρώματα (JND - Just Noticeable Difference). Μη εμπειρικά, η τιμή 1 χρησιμοποιείται συχνά, αλλά σε μία πρόσφατη μελέτη [33], εκτιμήθηκε πως κατάλληλη είναι η τιμή $\Delta E = 2,3$ ως JND. Ωστόσο, οι αντιληπτικές ανομοιομορφίες στον χρωματικό χώρο CIELAB εμποδίζουν την εύρεση της χρυσής τομής και έχουν οδηγήσει την CIE στην συνεχόμενη αναθεώρηση του ορισμού τους για το ΔE, οδηγώντας στις τελικές μέχρι στιγμής φόρμουλες του 1994 και 2000. Αυτές οι ανομοιομορφίες είναι σημαντικές επειδή το ανθρώπινο μάτι είναι πιο ευαίσθητο σε ορισμένα χρώματα απ' ότι σε άλλα. Μία καλή μετρική θα πρέπει να το λαμβάνει υπόψη, προκειμένου ο όρος 'μόλις αισθητή διαφορά - JND' να έχει νόημα. Σε αντίθετη περίπτωση, ένα συγκεκριμένο ΔE μπορεί να είναι ασήμαντο μεταξύ δύο χρωμάτων για τα οποία το μάτι δεν έχει την απαραίτητη αντιληπτική ευαισθησία, αλλά σε άλλο μέρος του φάσματος μπορεί να είναι ευδιάκριτη η διαφορά τους.

Το ΔE (1994) ορίζεται στον χρωματικό χώρο $L^*C^*h^*$ με διαφορές στη φωτεινότητα (lightness - L^*), καθαρότητα ή ένταση χρώματος (chroma - C^*) και απόχρωση (hue - h^*) που υπολογίζονται από τις συντεταγμένες $L^*a^*b^*$. Δεδομένων δύο χρωμάτων στον χώρο CIELAB (L_1^*, a_1^*, b_1^*) και (L_2^*, a_2^*, b_2^*), η διαφορά τους είναι:

$$\Delta E_{94}^* = \sqrt{\left(\frac{\Delta L^*}{k_L S_L}\right)^2 + \left(\frac{\Delta C_{ab}^*}{k_c S_C}\right)^2 + \left(\frac{\Delta H_{ab}^*}{k_H S_H}\right)^2} \tag{4.4}$$

όπου:

$$\Delta L^* = {L_1}^* - {L_2}^*$$
$${C_1}^* = \sqrt{{a_1}^{*2} + {b_1}^{*2}}$$
$${C_2}^* = \sqrt{{a_2}^{*2} + {b_2}^{v2}}$$
$$\Delta C^*_{ab} = C^*_2 - C^*_1$$
$$\Delta H^*_{ab} = \sqrt{{\Delta E^*_{ab}}^2 - \Delta L^{*2} - {\Delta C^*_{ab}}^2} = \sqrt{\Delta a^{*2} + \Delta b^{*2} - {\Delta C^*_{ab}}^2}$$
$$\Delta a^* = a^*_1 - a^*_2$$
$$\Delta b^* = b^*_1 - b^*_2$$
$$S_L = 1$$
$$S_C = 1 + K_1 C^*_1$$
$$S_H = 1 + K_2 C^*_1$$
$$k_L = 1$$
$$K_1 = 0.045$$
$$K_2 = 0.015$$

Γεωμετρικά η ποσότητα ΔH^*_{ab} αντιστοιχεί στον αριθμητικό μέσο όρο των μηκών χορδής, των ίδιων χρωματικών κύκλων από τα δύο χρώματα [34].

4.5 Μηχανισμοί προστασίας

Παρακάτω (Αλγόριθμος 4.6) φαίνεται ο τρόπος με τον οποίο επαναφέρεται η εφαρμογή από τυχόν λάθη που μπορεί να προκύψουν κατά τη διάρκεια εκτέλεσης των διεργασιών. Σε περίπτωση που η διαφορά κάποιου pixel από το αντίστοιχο pixel αναφοράς διαφέρει περισσότερο από $diff\_threshold$ τότε το pixel παίρνει την τιμή του pixel αναφοράς. Ο έλεγχος της απόστασης γίνεται σύμφωνα με την εξίσωση (4.4). Η επαναφορά των στοιχείων του πίνακα pixels αφορά την εμφάνιση της εικόνας κατά της διάρκεια δημιουργίας της (δηλαδή στις ενδιάμεσες επαναλήψεις) και δεν είναι απαραίτητη για την αποσφαλμάτωση των αποτελεσμάτων.

```
1  /************************************************
2   * restore_pixel(i)
3   * i is the index of a pixel in the image
4   ************************************************/
5  if ( diff(pixel(i), reference_pixel(i) ) > diff_threshold){
6    colors[i] = colors_buffer[i];
7    pixels[i] = toInt(colors[i].x) |
8                (toInt(colors[i].y) << 8) |
9                (toInt(colors[i].z) << 16);
10 }
```

Αλγόριθμος 4.6: SmallPt - Μηχανισμός προστασίας με επαναφορά χρωμάτων

4.6 Πειράματα και αποτελέσματα

Τα πειράματα εκτελέστηκαν σε μηχανήματα πανομοιότυπης τεχνολογίας με λειτουργικό σύστημα/kernel: Linux version 4.1.31-30-default (geeko@buildhost) (gcc version 4.8.5 (SUSE Linux)) #1 SMP PREEMPT Wed Aug 24 06:20:09 UTC 2016, επεξεργαστή: Intel Xeon E5606, 2.13 GHz, 8 MB cache, 1066 MHz memory, Quad-Core , RAM: 6GB. Το compile του κώδικα της εφαρμογής έγινε με τον compiler: gcc (SUSE Linux) 4.8.5 και για βελτιστοποιήσεις τα flags: *O3, msse2, mfpmath=sse, ftree-vectorize, funroll-loops.*

4.6.1 Περαματική διαδικασία

Για την αξιολόγηση των μηχανισμών ελέγχου λαθών και αποσφαλμάτωσης της εφαρμογής εκτελούνται 10.000 πειράματα για κάθε διαφορετική περίπτωση ποσοστού σφαλμάτων. Οι περιπτώσεις που επιλέγονται αναφέρονται στο α) 25% β) 50% γ) 75% και δ) 100% των συνολικών διεργασιών που έχουν επιλεχθεί για εισαγωγή σφαλμάτων. Ο αριθμός των σφαλμάτων ανέρχεται στα 10 σφάλματα ανά διεργασία.

Τα πειράματα που εκτελούνται αφορούν τη σκηνή που φαίνεται και στην εικόνα (4.3α') με μέγεθος 640∗480 = 307.200 pixel. Το μέγεθος ενός block είναι 32∗24 = 768 pixel, επομένως η εικόνα αποτελείται από $\frac{307.200}{768} = 400$ block, άρα και από 400 διεργασίες, ανά επανάληψη. Για την αξιολόγηση ο αλγόριθμος θα εκτελεστεί για 5.001 επαναλήψεις εκ των οποίων οι πρώτες 501 θα εκτελεστούν αυστηρά αξιόπιστα και στη συνέχεια δύναται να εκτελεστούν διεργασίες αναξιόπιστα, ανεξάρτητα από τον τελικό αριθμό επαναλήψεων. Σύμφωνα με τις παραπάνω παρατηρήσεις, το συνολικό πλήθος διεργασιών της εφαρμογής για 5.001 επαναλήψεις είναι $400\frac{task}{iter}$∗5.001 $iter$ = 2.000.400 διεργασίες. Από την επανάληψη 501 και ύστερα, οι διεργασίες επιλέγονται είτε για αξιόπιστη εκτέλεση, είτε για αναξιόπιστη εκτέλεση, είτε για παράλειψη της εκτέλεσής τους, σύμφωνα με την ανάλυση της ενότητας 4.3. Στις επαναλήψεις μετά την 501^{η} εισάγονται σφάλματα σε διεργασίες που επιλέχθηκαν για αναξιόπιστη εκτέλεση, σε πλήθος διεργασιών αντίστοιχο με το ποσοστό σφαλμάτων. Οι διεργασίες επιλέγονται τυχαία σε κάθε επανάληψη χρησιμοποιώντας τον αλγόριθμο του Knuth [24], αλγόριθμος ο οποίος αξιοποιείται για την επιλογή μίας σειράς από μοναδικούς τυχαίους αριθμούς.

Παρακάτω αναγράφονται οι τιμές που καταγράφονται από την εκτέλεση των πειραμάτων για την εξαγωγή στατιστικών δεδομένων:

- (PSNR) σε σχέση με την αξιόπιστη εκτέλεση
- Χρόνος αξιόπιστης εκτέλεσης
- Χρόνος αναξιόπιστης εκτέλεσης
- Χρόνος αναγνώρισης σφάλματων
- Χρόνος αποσφαλμάτωσης
- Χρόνος εισαγωγής σφαλμάτων

- Αριθμός αναξιόπιστων διεργασιών
- Αριθμός αξιόπιστων διεργασιών
- Αριθμός διεργασιών που δεν εκτελέστηκαν

Στην εφαρμογή SmallPt ακολουθείται μία διαφορετική προσέγγιση για τις μετρήσεις χρόνου εκτέλεσης από αυτήν της εφαρμογής MD. Ο χρόνος αξιόπιστης εκτέλεσης αναφέρεται σε όλες τις διεργασίες που εκτελέστηκαν αξιόπιστα. Αυτό σημαίνει ότι προσμετράται ο χρόνος εκτέλεσης των διεργασιών που ήταν δυνατό να εκτελεστούν αναξιόπιστα, αλλά για ποσοστό σφαλμάτων μικρότερο του 100%, δεν επιλέχθηκαν για αναξιόπιστη εκτέλεση. Αυτές οι διεργασίες μεταφέρονται στην αξιόπιστη εκτέλεση. Έτσι ο χρόνος αξιόπιστης εκτέλεσης αναφέρεται στις διεργασίες που επιλέχθηκαν για αξιόπιστη εκτέλεση συν τις διεργασίες που δεν επιλέχθηκαν για αναξιόπιστη εκτέλεση. Επιπλέον, στον χρόνο αναξιόπιστης εκτέλεσης δεν προσμετρούνται οι χρόνοι: α) εισαγωγής σφαλμάτων, β) ελέγχου σφαλμάτων, γ) διόρθωσης σφαλμάτων και δ) απόδοσης σημαντικότητας.

Το πλήθος των α) αναξιόπιστων διεργασιών, β) αξιόπιστων διεργασιών και γ) διεργασιών που δεν εκτελέστηκαν, εξάγεται από κάθε εκτέλεση της εφαρμογής και δεν μπορεί να υπολογιστεί εκ των προτέρων, διότι διαφέρει από εκτέλεση σε εκτέλεση της εφαρμογής. Επίσης είναι σημαντικό να αναφερθεί ότι στα πειράματα με ποσοστό σφαλμάτων μικρότερο του 100%, η αντιστοιχία στο πλήθος των διεργασιών που επιλέγονται για εισαγωγή σφαλμάτων δεν είναι ένα προς ένα με το αντίστοιχο πλήθος των πειραμάτων με ποσοστό σφαλμάτων 100%. Για παράδειγμα, αν στην περίπτωση που το ποσοστό σφαλμάτων είναι 100% επιλέγονται 100 διεργασίες για αναξιόπιστη εκτέλεση σε μία επανάληψη, τότε όταν το ποσοστό σφαλμάτων είναι 50%, δεν συνεπάγεται ότι ο αλγόριθμος θα επιλέξει 50 διεργασίες για εισαγωγή σφαλμάτων στην ίδια επανάληψη. Αυτό συμβαίνει διότι εισάγοντας λιγότερα σφάλματα (λόγω του μικρότερου ποσοστού) επιτυγχάνεται καλύτερο αποτέλεσμα σε κάποιες διεργασίες και κατ' επέκταση μειώνεται η σημαντικότητά τους. Σύμφωνα με αυτήν την παρατήρηση και την ανάλυση της ενότητας 4.3 προκύπτει ότι ο αλγόριθμος θα επιλέξει περισσότερες διεργασίες, την εκτέλεση των οποίων θα παραλείψει, μειώνοντας έτσι τον αριθμό των διεργασιών που θα εκτελεστούν αναξιόπιστα.

4.6.2 Αποτελέσματα

Η πρώτη γραφική παράσταση δείχνει την ποιότητα της παραγόμενης εικόνας για διαφορετικές τιμές ποσοστού σφαλμάτων σύμφωνα με τη μετρική PSNR (Peak Signal-to-Noise Ratio)

Σχήμα 4.5: SmallPt - PSNR

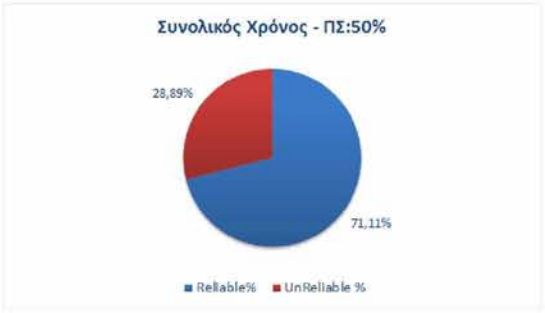
Είναι εμφανές ότι και στις τέσσερις περιπτώσεις το τελικό αποτέλεσμα είναι αρκετά καλό με το μικρότερο $PSNR \approx 54,3dB$ να εμφανίζεται στην περίπτωση που εισάγονται τα περισσότερα σφάλματα.

Στην συνέχεια φαίνεται ο συνολικός χρόνος εκτέλεσης για κάθε περίπτωση ποσοστού σφαλμάτων, καθώς και της αξιόπιστης εκτέλεσης (Σχήμα 4.6).

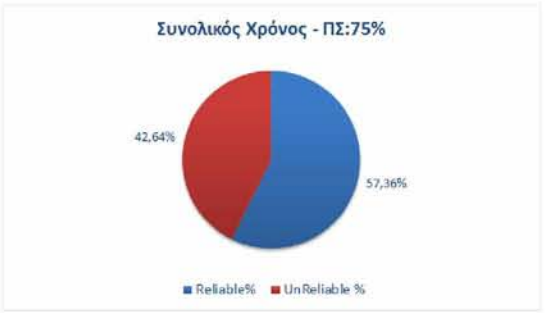
Σχήμα 4.6: SmallPt - Συνολικός χρόνος εκτέλεσης

Από την παραπάνω γραφική παράσταση προκύπτει ότι για τις περιπτώσεις ποσοστού σφαλμάτων: α) 25%, β) 50% και γ) 75% ο συνολικός χρόνος εκτέλεσης είναι μικρότερος από την ακριβή εκτέλεση της εφαρμογής. Αυτό συμβαίνει διότι το ποσοστό των διεργασιών που δεν εκτελούνται καθόλου είναι αρκετά μεγάλο, με αποτέλεσμα να μειώνεται αισθητά το υπολογιστικό κόστος.

(α′) Ποσοστό εσφαλμένων διεργασιών 25%



(β′) Ποσοστό εσφαλμένων διεργασιών 50%

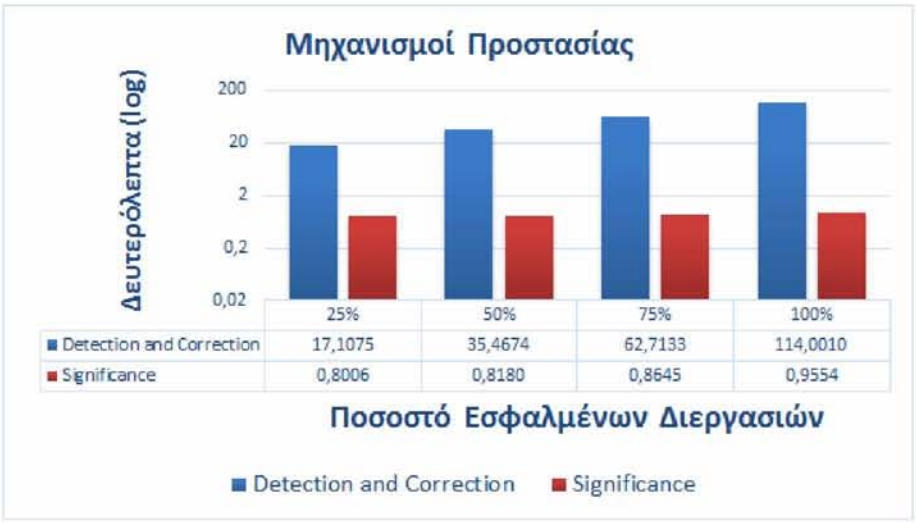


(γ′) Ποσοστό εσφαλμένων διεργασιών 75%

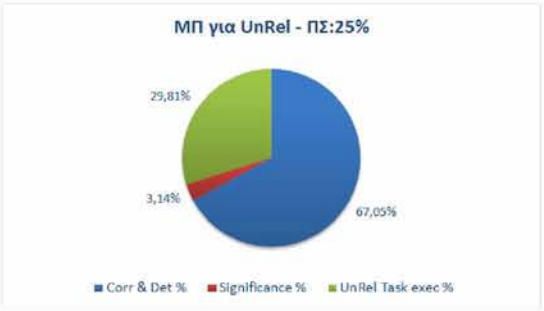
(δ′) Ποσοστό εσφαλμένων διεργασιών 100%

Σχήμα 4.7: SmallPt - Χρόνοι αξιόπιστης και αναξιόπιστης εκτέλεσης για κάθε ποσοστό σφαλμάτων

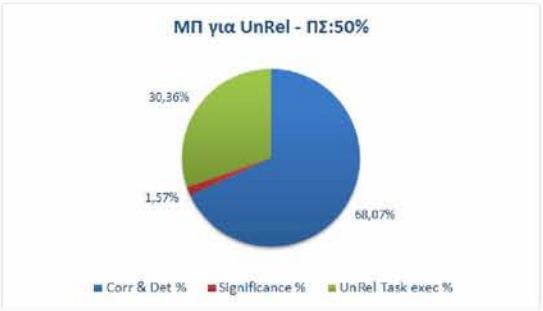
Στις παραπάνω γραφικές παραστάσεις (Σχήμα 4.7) φαίνονται τα ποσοστά αξιόπιστης και αναξιόπιστης εκτέλεσης επί του συνολικού χρόνου. Ο χρόνος αναξιόπιστης εκτέλεσης είναι ανάλογος του ποσοστού των σφαλμάτων που εισάγονται στην εφαρμογή.



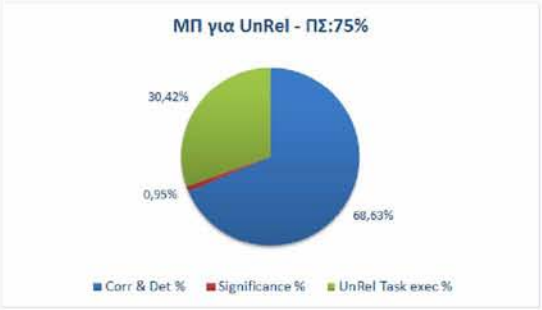
Σχήμα 4.8: SmallPt - Χρόνοι μηχανισμών προστασίας



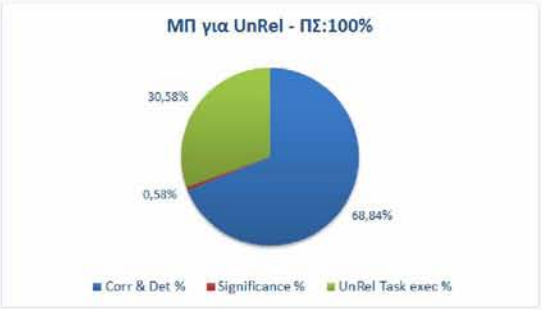
(α′) Ποσοστό εσφαλμένων διεργασιών 25%



(β′) Ποσοστό εσφαλμένων διεργασιών 50%



(γ′) Ποσοστό εσφαλμένων διεργασιών 75%



(δ′) Ποσοστό εσφαλμένων διεργασιών 100%

Σχήμα 4.9: SmallPt - Ανάλυση χρόνου εκτέλεσης για τις τέσσερις περιπτώσεις ποσοστού σφαλμάτων σχετικά με τον χρόνο αναξιόπιστης εκτέλεσης

Σχήμα 4.10: SmallPt - Κατανομή διεργασιών

(α′) Ποσοστό εσφαλμένων διεργασιών 25%

(β′) Ποσοστό εσφαλμένων διεργασιών 50%

(γ′) Ποσοστό εσφαλμένων διεργασιών 75%

(δ′) Ποσοστό εσφαλμένων διεργασιών 100%

Σχήμα 4.11: SmallPt - Ανάλυση χρόνου εκτέλεσης για τις τέσσερις περιπτώσεις ποσοστού σφαλμάτων σχετικά με τον συνολικό χρόνο εκτέλεσης

Στη γραφική παράσταση (Σχήμα 4.8) φαίνονται οι χρόνοι που καταλαμβάνουν οι ρουτίνες του μηχανισμού προστασίας σε λογαριθμική κλίμακα. Αναγράφονται οι χρόνοι: α) αναγνώρισης και επιδιόρθωσης λαθών και β) απόδοσης σημαντικότητας, για κάθε περίπτωση ποσοστού σφαλμάτων. Όπως φαίνεται και από την γραφική παράσταση ο χρόνος απόδοσης σημαντικότητας είναι σχεδόν αμελητέος ενώ ο χρόνος ελέγχου λαθών αποτελεί ένα μεγάλο μέρος του συνολικού χρόνου εκτέλεσης. Ο χρόνος επιδιόρθωσης λαθών είναι ελάχιστος (η ρουτίνα αποτελείται από μία εντολή) και για το λόγο αυτό προσμετράται στον χρόνο εύρεσης λαθών.

Στις γραφικές παραστάσεις (Σχήμα 4.9) φαίνονται τα ποσοστά που καταλαμβάνουν οι μηχανισμοί προστασίας και η εκτέλεση των αναξιόπιστων διεργασιών, σε σχέση με τον συνολικό χρόνο αναξιόπιστης εκτέλεσης. Ο συνολικός χρόνος αναξιόπιστης εκτέλεσης είναι το άθροισμα του χρόνου εκτέλεσης αναξιόπιστων διεργασιών, του χρόνου ελέγχου λαθών, του χρόνου διόρθωσης λαθών και του χρόνου απόδοσης σημαντικότητας.

Στη γραφική παράσταση (Σχήμα 4.10) φαίνεται το ποσοστό των συνολικών διεργασιών που α) εκτελέστηκαν αξιόπιστα, β) εκτελέστηκαν αναξιόπιστα και γ) δεν εκτελέστηκαν καθόλου.

Συνοψίζοντας τις μετρήσεις που αφορούν την ανάλυση των χρόνων εκτέλεσης, οι γραφικές παραστάσεις (Σχήμα 4.11) δείχνουν το ποσοστό του συνολικού χρόνου εκτέλεσης που καταλαμβάνει κάθε επιμέρους λειτουργία του αλγορίθμου της εφαρμογής για κάθε ποσοστό σφαλμάτων.

Σχήμα 4.12: SmallPt - Χρόνοι εισαγωγής σφαλμάτων

Οι διεργασίες που επιλέγονται για αξιόπιστη εκτέλεση ξεπερνούν σε αριθμό τις διεργασίες που επιλέγονται για αναξιόπιστη εκτέλεση και αυτό δικαιολογεί το ποσοστό του συνολικού χρόνου εκτέλεσης που καταλαμβάνεται από την αξιόπιστη εκτέλεση. Ωστόσο, το πλήθος των διεργασιών που δεν εκτελούνται καθόλου ξεπερνάει κατά πολύ το πλήθος των διεργασιών των άλλων δύο περιπτώσεων, με αποτέλεσμα να ελαφρύνει σε μεγάλο βαθμό τους συνολικούς υπολογισμούς. Επιπλέον, για τις περιπτώσεις ποσοστού σφαλμάτων 25%, 50% και 75%, οδηγεί σε κέρδος στον συνολικό χρόνο εκτέλεσης σε σχέση με την αποκλειστικά αξιόπιστη εκτέλεση της εφαρμογής.

Το επιπλέον υπολογιστικό τμήμα της απόδοσης σημαντικότητας καταλαμβάνει ελάχιστο χρόνο σε σχέση με τον συνολικό χρόνο εκτέλεσης και ουσιαστικά δεν επιβαρύνει την εκτέλεση της εφαρμογής. Αντίθετα, το επιπλέον υπολογιστικό τμήμα του ελέγχου λαθών και διόρθωσης αυτών καταλαμβάνει ένα μεγάλο μέρος του συνολικού χρόνου εκτέλεσης και αυτό οφείλεται στη σχετικά πολύπλοκη λειτουργία ελέγχου λαθών.

4.6.3 Μελλοντικές επεκτάσεις

Η διαδικασία εύρεσης λαθών απαιτεί μεγάλο υπολογιστικό κόστος και αποτελεί κομβικό σημείο για τη μείωση του χρόνου εκτέλεσης σε μεγάλα ποσοστά σφαλμάτων. Ο λόγος για τον οποίο είναι χρονοβόρο αυτό το τμήμα της εφαρμογής είναι η μετατροπή των χρωμάτων στον χρωματικό χώρο CIELab για την εύρεση της διαφοράς ανάμεσα σε δύο χρώματα (Εξίσωση 4.4). Σε αυτό το τμήμα μπορεί να μελετηθούν άλλες μετρικές απόστασης χρωμάτων με λιγότερο υπολογιστικό κόστος που να μειώνουν το χρόνο εύρεσης λαθών.

Η εικόνα που παράγεται ως έξοδος στην αναξιόπιστη εκτέλεση της εφαρμογής έχει ελάχιστο $PSNR \approx 54,3dB$ για την περίπτωση που εισάγονται τα περισσότερα σφάλματα. Ωστόσο, φαίνεται ότι αυτή η τιμή του $PSNR$ δύναται να βελτιωθεί αν βρεθεί καλύτερη συνάρτηση αποσφαλμάτωσης. Η συνάρτηση που χρησιμοποιείται σε αυτή την εργασία κάνει ουσιαστικά οπισθοδρόμηση των χρωμάτων, όταν βρεθεί λάθος, σε τιμές που ανήκουν σε

παλαιότερες επαναλήψεις αλλά είναι εγγυημένα σωστές. Με αυτήν την παρατήρηση φαίνεται πως με μία συνάρτηση αποσφαλμάτωσης που κάνει μικρότερη οπισθοδρόμηση σε βάθος επαναλήψεων είναι δυνατόν να επιτευχθεί καλύτερη τιμή $PSNR$.

Τέλος, μπορούν να γίνουν δοκιμές με επανεκτέλεση του αλγορίθμου σε περιπτώσεις όπου αναγνωρίζεται λάθος, καθώς και δοκιμές με διαφορετικά μεγέθη block. Και στις δύο περιπτώσεις πρέπει να μελετηθεί η χρονική απόκριση της εφαρμογής με σχετική πειραματική διαδικασία.

Κεφάλαιο 5

Σχετική Δουλειά

Στην ενότητα αυτή παρουσιάζονται μερικές από τις ερευνητικές προσπάθειες σε τεχνικές λογισμικού και υλικού για την εισαγωγή και ανίχνευση σφαλμάτων, καθώς και την ανακούφιση των δυσμενών επιπτώσεων των σφαλμάτων στη λειτουργικότητα και την επίδοση των συστημάτων.

- Από το πρόγραμμα SCoRPiO - Significance-Based Computing for Reliability and Power Optimization έχει αναπτυχθεί ένα framework [1] που χαλαρώνει τις ζώνες προστασίας του υλικού[1] για τη μείωση του ενεργειακού αποτυπώματος, με κόστος την υποβαθμισμένη ποιότητα εξόδου. Το προγραμματιστικό μοντέλο του SCoRPiO επιτρέπει στον προγραμματιστή να καθορίσει τη σημαντικότητα των διεργασιών μιας εφαρμογής σύμφωνα με την συνεισφορά τους στο τελικό αποτέλεσμα. Επιπλέον, παρέχει ρουτίνες ελέγχου λαθών και αποσφαλμάτωσης για τον χειρισμό μιας εκτέλεσης που έχει διακοπεί ή δεδομένων που έχουν καταστραφεί. Βασισμένο στον βαθμό αναξιόπιστης εκτέλεσης που παρέχεται από τον χρήστη και στη σημαντικότητα των διεργασιών, το σύστημα χρόνου εκτέλεσης εκτελεί τις λιγότερο σημαντικές διεργασίες πάνω από αναξιόπιστους πυρήνες, οι οποίοι έχουν χαμηλότερη κατανάλωση ενέργειας από τους συμβατικούς.

- Topaz [35]: είναι ένα framework για πλατφόρμες που μπορούν να χρησιμοποιηθούν για προσέγγιση υπολογισμών εκτελώντας τους αναξιόπιστα. Χρησιμοποιεί έναν μηχανισμό ανίχνευσης ανωμαλιών για την ανίχνευση και αποσφαλμάτωση μη αποδεκτών αποτελεσμάτων, μέσω επανεκτέλεσης σε αξιόπιστο υλικό.

- Chisel [36]: είναι ένα framework το οποίο δεδομένου ενός συνδυασμού αξιοπιστίας ή/και ακρίβειας προδιαγραφών υλικού, αυτόματα επιλέγει προσεγγιστικά τμήματα προς εκτέλεση. Σκοπός του είναι να συνθέσει έναν προσεγγιστικό υπολογισμό που ελαχιστοποιεί την κατανάλωση ενέργειας, ενώ παράλληλα ικανοποιεί την αξιοπιστία και ακρίβεια των προδιαγραφών.

[1] Οι ζώνες προστασίας στο υλικό οδηγούν σε περιττή κατανάλωση ενέργειας για εργασίες που μπορούν να ανεχθούν ανακρίβεια στα δεδομένα και στα σφάλματα

- Rinard: σε μία από τις πρώιμες χρονολογικά προσπάθειες για υπολογισμούς τμηματοποιημένους σε διεργασίες, προτείνει έναν μηχανισμό λογισμικού που επιτρέπει στον προγραμματιστή να προσδιορίσει block διεργασιών και στη συνέχεια να δημιουργήσει ένα πιθανολογικό μοντέλο σφάλματος για κάθε διεργασία [37]. Αυτό επιτυγχάνεται με την εισαγωγή σφαλμάτων στην εκτέλεση των διεργασιών και έπειτα παρατηρώντας την προκύπτουσα παραμόρφωση εξόδου, καθώς και τα ποσοστά αποτυχίας. Ένα τέτοιο μοντέλο μπορεί να χρησιμοποιηθεί για να εκτιμηθεί το πώς οι αποτυχίες στην εκτέλεση ενός προγράμματος επηρεάζουν την ακρίβεια του αποτελέσματος, καθώς και για την εξαγωγή ορίων σφάλματος υπό διαφορετικά σενάρια εκτέλεσης.

- Relax: πρόκειται για ένα αρχιτεκτονικό framework που επιτρέπει στους προγραμματιστές να σημειώνουν περιοχές κώδικα για τις οποίες μπορεί να εμφανιστούν σφάλματα υλικού και, παράλληλα, μπορεί να απενεργοποιήσει τους μηχανισμούς αποσφαλμάτωσης [38]. Το υλικό υποστηρίζει αναγνώριση σφαλμάτων και μηχανισμοί προστασίας σε C/C++ παρέχουν επαναφορά από σφάλματα υλικού σε διαφορετικά επίπεδα διακριτότητας κώδικα.

- EnerJ: προτείνει προγραμματισμό προσεγγίσεων δηλώνοντας ρητά δομές δεδομένων που ενδέχεται να υπόκεινται σε προσεγγιστικούς υπολογισμούς με αντάλλαγμα την αυξημένη απόδοση ή την ανοχή σε βλάβες [39]. Το EnerJ επιτρέπει να γίνονται οι υπολογισμοί σε επιθετικά κλιμακούμενης τάσης επεξεργαστές και οι δομές δεδομένων να αποθηκεύονται στην DRAM με χαμηλό ρυθμό ανανέωσης, καθώς και στην SRAM με χαμηλή παροχή τάσης. Εκθέτοντας τους προσεγγιστικούς υπολογισμούς στον προγραμματιστή, απαιτείται επέκταση του συνόλου εντολών του επεξεργαστή (ISA) με εντολές που προσφέρουν μόνο μία προσδοκία, αντί για εντολές που εγγυώνται ότι μία λειτουργία θα εκτελεστεί σωστά [40].

- GEMFi [41]: είναι ένα εργαλείο εισαγωγής σφαλμάτων βασισμένο στον προσομοιωτή Gem5. Το GEMFi παρέχει μεθόδους εισαγωγής σφαλμάτων και επεκτείνεται εύκολα για μελλοντικά μοντέλα σφαλμάτων. Ακόμη υποστηρίζει πολλαπλά μοντέλα επεξεργαστών και σύνολα εντολών, καθώς και επιτρέπει την εισαγωγή σφαλμάτων τόσο σε λειτουργικές μοντελοποιήσεις όσο και σε μοντελοποιήσεις με ακρίβεια στους κύκλους μηχανής. Το GEMFi προσφέρει γρήγορη προώθηση από εκστρατείες προσομοιώσεων με χρήση σημείων ελέγχου και επιπλέον διευκολύνει την παράλληλη εκτέλεση πολλών πειραμάτων σε ένα δίκτυο από σταθμούς εργασίας.

- ApproxIt: αποτελεί ένα framework για προσεγγιστικούς υπολογισμούς που απευθύνεται σε επαναληπτικές μεθόδους και βασίζεται σε έναν ελαφρύ μηχανισμό ελέγχου ποιότητας [42]. Τα ελαστικά και ευαίσθητα σε σφάλματα μέρη κάθε εφαρμογής προσδιορίζονται κάνοντας εκ των προτέρων profiling. Στη συνέχεια, κατά τον χρόνο εκτέλεσης γίνονται αναδιαμορφώσεις των τρόπων προσέγγισης σε συγκεκριμένες επαναλήψεις, λαμβάνοντας υπόψη τη χρονομεταβαλλόμενη ελαστικότητα της κάθε εφαρμογής και την απαίτηση για την ποιότητα της εξόδου που έχει θέσει ο χρήστης.

- Razor: για να υποστηρίζει το υλικό ανοχή σε λάθη και προσεγγιστικούς υπολογισμούς δημιουργούνται σχέδια που μπορεί είτε να επεκτείνουν ένα υπάρχον σχέδιο με ένα επιπλέον κόστος, είτε να γεννούν ριζικά νέες αρχιτεκτονικές. Το Razor είναι ένα σχέδιο επεξεργαστή που βασίζεται στη δυναμική αναγνώριση και διόρθωση σφαλμάτων χρονισμού των κρίσιμων μονοπατιών, λόγω της μειωμένης παροχής τάσης [43]. Η βασική ιδέα είναι η ρύθμιση της τάσης τροφοδοσίας παρακολουθώντας το ποσοστό σφαλμάτων κατά τη λειτουργία. Αυτό επιτυγχάνεται χρησιμοποιώντας αυτόματα κλείστρα τα οποία ελέγχονται από ρολόγια που έχουν καθυστέρηση.

- Η ERSA είναι μία πολυπύρηνη αρχιτεκτονική, οι πυρήνες της οποίας είτε είναι εντελώς αξιόπιστοι είτε έχουν ελαστική αξιοπιστία [44]. Συγκεκριμένα, χρησιμοποιεί μία σαφή και βασισμένη στην εφαρμογή ανάθεση του κώδικα στους επεξεργαστές με διαφορετικά επίπεδα αξιοπιστίας.

Κεφάλαιο 6

Συμπεράσματα

Από την πειραματική διαδικασία παρατηρείται:

- Για την εκτέλεση εφαρμογών πάνω από αξιόπιστο υλικό απαιτείται αποδοτική ανίχνευση λαθών και αποσφαλμάτωση ώστε να προκύπτει αποδεκτό αποτέλεσμα.

- Για να έχει νόημα η εκτέλεση σε αναξιόπιστο υλικό, οι ρουτίνες ελέγχου λαθών και επιδιόρθωσης των σφαλμάτων πρέπει να είναι πολύπλευρα μελετημένες ώστε να έχουν μικρό υπολογιστικό κόστος και να μην επιτρέπουν την σιωπηλή διάδοση λαθών.

- MD: Όπως παρατηρείται για τον έλεγχο των σφαλμάτων, η μετρική της συνολικής ενέργειας του συστήματος κρίνεται σημαντική και ωφέλιμη καθώς βοηθάει στην διόρθωση των σφαλμάτων προσδίδοντας ελάχιστα επιπλέον υπολογιστικό κόστος.

- MD: Η επανεκτέλεση τον διεργασιών σε περιπτώσεις λάθους δεν αποτελεί την βέλτιστη λύση μηχανισμού προστασίας καθώς αυξάνει αισθητά την συνολική εκτέλεση της εφαρμογής. Σε αυτό το σημείο είναι δυνατόν να μελετηθούν διαφορετικοί μηχανισμοί προστασίας είτε με προσεγγιστικές μεθόδους είτε με επαναφορά τον δεδομένων σε τιμές προηγούμενων επαναλήψεων.

- MD: Από την αξιολόγηση των μηχανισμών προστασίας προκύπτει πως για μικρό ποσοστό εσφαλμένων διεργασιών, εμφανίζονται σιωπηλά διαδοθέντα σφάλματα. Αυτό συμβαίνει λόγω του ότι ο μηχανισμός ελέγχου λαθών αδυνατεί να εντοπίσει λάθη που δεν δημιουργούν μεγάλη διαφορά στο αποτέλεσμα κάθε επανάληψης και κατ΄ επέκταση δεν γίνεται επιδιόρθωση αυτών.

- SmallPt: Σε αυτήν την εφαρμογή γίνεται εκμετάλλευση της δυναμικής απόδοσης σημαντικότητας για τον προσεγγιστικό υπολογισμό του τελικού αποτελέσματος.

- SmallPt: Παρά το σημαντικό υπολογιστικό κόστος του ελέγχου σφαλμάτων προκύπτει κέρδος χρόνου εκτέλεσης λόγω της προσεγγιστικής μεθόδου που ακολουθείται κατά την οποία μεγάλο πλήθος διεργασιών δεν εκτελείται.

- Σε αυτή τη διπλωματική, οι μηχανισμοί ελέγχου λαθών περιορίζονται σε ρουτίνες λογισμικού, αυξάνοντας έτσι την πολυπλοκότητα και κατ' επέκταση την συνολική εκτέλεση των εφαρμογών. Αυτό μπορεί να ελαφρυνθεί σε περιπτώσεις όπου υπάρχουν μηχανισμοί υλικού οι οποίοι παρουσιάζουν τα λάθη(όπως στην cache ή στην RAM), με αποτέλεσμα να επιτυγχάνεται υπολογιστικό κέρδος στην εύρεση των λαθών.

6.1 Περίληψη

Σε αυτή την εργασία μελετώνται δύο εφαρμογές με στόχο την εκτέλεση τους σε αναξιόπιστο υλικό. Αρχικά, αναλύεται το βασικό μαθηματικό υπόβαθρο κάθε εφαρμογής που απαιτείται για την ανάλυση και τη μετατροπή της δομής της κάθε εφαρμογής, ώστε να εκτελείται σε αναξιόπιστο υλικό. Επιπρόσθετα, εξετάζεται η διαδικασία τμηματοποίησης της κάθε εφαρμογής σε επιμέρους διεργασίες για την εκτέλεσή της σε πολυπύρηνο σύστημα και, στη συνέχεια, η μέθοδος με την οποία προσδίδεται σημαντικότητα σε κάθε διεργασία. Μάλιστα για κάθε εφαρμογή αναλύεται η απαραίτητη επέκταση με μηχανισμούς αναγνώρισης λαθών και αποσφαλμάτωσης, ώστε να παράγονται επιθυμητά αποτελέσματα κατά την αναξιόπιστη εκτέλεση.

Όπως φαίνεται από την πειραματική διαδικασία που εκτελέστηκε για κάθε εφαρμογή, είναι δυνατή η εκτέλεση εφαρμογών πάνω από αναξιόπιστο υλικό με μικρό κόστος στην ακρίβεια των αποτελεσμάτων αλλά με κέρδος στην κατανάλωση ισχύος. Κάνοντας τις απαραίτητες παραμετροποιήσεις οι εφαρμογές εκτελούνται πάνω από αναξιόπιστο υλικό και τα λάθη που προκύπτουν επιλύονται στον χρόνο εκτέλεσης με ένα επιπλέον υπολογιστικό κόστος. Σύμφωνα με αυτά, η παραμετροποίηση μίας εφαρμογής συνοψίζεται σε: α) αναλυτική μελέτη του μαθηματικού υπόβαθρου και της λειτουργίας της εφαρμογής, β) τμηματοποίηση της εφαρμογής σε επιμέρους διεργασίες με τρόπο ώστε να μπορεί να αποδοθεί σημαντικότητα σε αυτές, γ) απόδοση σημαντικότητας στις διεργασίες και δ) δημιουργία μηχανισμών προστασίας για τον έλεγχο των αποτελεσμάτων των διεργασιών που εκτελέστηκαν αναξιόπιστα.

6.2 Μελλοντική δουλειά

Για την επέκταση αυτή της εργασίας προτείνεται η βελτιστοποίηση των μεθόδων που προστέθηκαν στις δύο αυτές εφαρμογές, τόσο προγραμματιστικά όσο και αλγοριθμικά, για την επίτευξη καλύτερων αποτελεσμάτων. Επιπλέον, προτείνεται η αξιολόγηση της συμπεριφοράς των εφαρμογών καθώς και των μηχανισμών που προτάθηκαν σε αυτήν την εργασία με κάποιο framework εισαγωγής σφαλμάτων. Με τη χρήση ενός framework εισαγωγής σφαλμάτων δύναται να εισαχθούν σφάλματα σε διάφορα σημεία του pipeline ενός επεξεργαστή. Αυτό συνεπάγεται πως είναι δυνατόν να υπάρξουν σφάλματα σε όλα τα τμήματα μιας εντολής (Fetch, Execute, WB κ.λ.π). Αυτό σημαίνει ουσιαστικά ότι η εισαγωγή σφαλμάτων με τη χρήση ενός σχετικού framework έρχεται πιο κοντά στο υλικό και χάνει την αφηρημένη έννοια που προστίθεται από μία γλώσσα προγραμματισμού υψηλού επιπέδου. Τέλος, είναι θεμιτός ο εμπλουτισμός αυτής της εργασίας με τη μελέτη επιπλέον εφαρμογών, για τη δημιουργία μίας

πολύπλευρα ορισμένης εικόνας για την εκτέλεση εφαρμογών σε αναξιόπιστο υλικό.

Βιβλιογραφία

[1] Konstantinos Parasyris, Vassilis Vassiliadis, Christos D. Antonopoulos, Spyros Lalis και Nikolaos Bellas. "*A Significance-Aware Software Stack for Computing on Unreliable Hardware*". 2015.

[2] Tushar B. Kute. "*C Program to find the floating point IEEE 754 representation*", 2013. Επίσης διαθέσιμο στη σελίδα http://itsitrc.blogspot.gr/2013/09/c-program-to-find-floating-point-ieee.html.

[3] Username:gggustafson. "*CThe Known Colors Palette Tool*", 2015. Επίσης διαθέσιμο στη σελίδα http://www.codeproject.com/Articles/243610/The-Known-Colors-Palette-Tool-Final-Revision-Hopef.

[4] Robert H. Dennard, Fritz Gaensslen, Hwa Nien Yu, Leo Rideout, Ernest Bassous και Andre LeBlanc. "*Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions*". SC-9(5), 1974.

[5] Adrian McMenamin. "*The end of Dennard scaling*", 2013.

[6] Koomey J. G. "*Outperforming Moore's Law*". 2010.

[7] Jonathan Koomey, Stephen Berard, Marla Sanchez και Henry Wong. "*Implications of Historical Trends in the Electrical Efficiency of Computing*". 33(3):46–54, 2010.

[8] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K.Flautner, και T. Mudge. "*A self-tuning dvs processor using delay-error detection and correction*". 41(4):792–804, 2006.

[9] J. Shalf, S. Dosanjh, και J. Morrison. "*Exascale computing technology challenges*". 6449 of Lecture Notes in Computer Science:1–25, 2011.

[10] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill και others. "*Exascale software study: Software challenges in extreme scale systems*". 2009.

[11] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer και M. Snir. "*To-ward exascale resilience*". 2009.

[12] A. Doucet, S. Godsill και C. Andrieu. "*On Sequential Monte Carlo Sampling Methods for Bayesian Filtering*". 10(3):197–208, 2000.

[13] Í. Goiri, R. Bianchini, S. Nagarakatte και T. D. Nguyen. "*ApproxHadoop: Bringing Approximations to MapReduce Frameworks*". σελίδες 383–397, 2015.

[14] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay και A. Burg. "*Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment*". σελίδες 381–386, 2015.

[15] A. Rahimi, L. Benini και R. K. Gupta. "*Analysis of instruction-level vulnerability to dynamic voltage and temperature variations*". σελίδες 1102–1105, 2012.

[16] G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. Ogrenci Memik και S. Parthasarathy. "*b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units*". σελίδες 105:1–105:6, 2015.

[17] D. Blaauw, S. Kalaiselvan, K. Lai, W. Ma, S. Pant, C. Tokunaga, S.Das και D. M. Bull. "*Razor II: in situ error detection and correction for PVT and SER tolerance,*". σελίδες 400–401, 2008.

[18] M. S. Gupta, J. A. Rivers, P. Bose, G. Y. Wei και D. Brooks. "*Tribeca: Design for pvt variations with local recovery and fine-grained adaptation*". σελίδες 435–446, 2009.

[19] D.A. Patterson και J.L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2004.

[20] IEEE Computer Society (1985). *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[21] OpenMP Architecture Review Board. *OpenMP Application Program Interface (version 4.0)*, 2013.

[22] L. Verlet. "*Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules*". 159:98–103, 1967.

[23] L. Onsager. "*Electric moments of molecules in liquids*". 58(8):1486–1493, 1936.

[24] D.E. Knuth. *The Art of Computer Programming: Seminumerical algorithms*. The Art of Computer Programming. Addison-Wesley Pub. Co., 1981.

[25] David Bucciarelli. "*SmallptCPU vs SmallptGPU*", 2009. Επίσης διαθέσιμο στη σελίδα davibu.interfree.it/opencl/smallptgpu/smallptGPU.html.

[26] Kevin Beason. "*SmallPt. Global Illumination*", 2012. Επίσης διαθέσιμο στη σελίδα www.kevinbeason.com/smallpt.

[27] Digital Tutors Team. "*Understanding Global Illumination*", 2013. Επίσης διαθέσιμο στη σελίδα http://blog.digitaltutors.com/understanding-global-illumination/.

[28] Chris Wynn. "*An Introduction to BRDF-Based Lighting*", 2014.

[29] H.R. Kang. *Computational Color Technology.* SPIE Press monograph. Society of Photo Optical, 2006.

[30] J. Schanda. *Colorimetry: Understanding the CIE System.* Wiley, 2007.

[31] CIE. "*International Commission on Illumination*", 1913.

[32] W. Backhaus, R. Kliegl και J.S. Werner. *Color Vision: Perspectives from Different Disciplines.* Walter de Gruyter, 1998.

[33] M. Mahy, L. VanEycken και A. Oosterlinck. "*Evaluation of uniform color spaces developed after the adoption of CIELAB and CIELUV Color Research and Application*". 19:105–121, 1994.

[34] G.A. Klein. *Industrial Color Physics.* Springer Series in Optical Sciences. Springer New York, 2010.

[35] S. Achour και M. C. Rinard. "*Approximate computation with outlier detection in topaz*". σελίδες 711–730, 2015.

[36] S. Misailovic, M. Carbin, Z. Qi S. Achour και M. C. Rinard. "*Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels*". 49:309–328, 2014.

[37] M. Rinard. "*Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks*". σελίδες 324–334, 2006.

[38] M.de Kruijf, S. Nomura και K. Sankaralingam. "*Relax: An Archi- tectural Framework for Software Recovery of Hardware Faults*". σελίδες 497–508, 2010.

[39] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze και D. Grossman. "*EnerJ: Approximate Data Types for Safe and General Low-power Computation*". σελίδες 164–174, 2011.

[40] H. Esmaeilzadeh, A. Sampson, L. Ceze και D. Burger. "*Architecture Support for Disciplined Approximate Programming*". σελίδες 301–312, 2012.

[41] K. Parasyris, G. Tziantzoulis, C. Antonopoulos και N. Bellas. "*Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates*". σελίδες 622–629, 2014.

[42] Q. Zhang, F. Yuan, R. Ye και Q. Xu. "*ApproxIt: An Approximate Computing Framework for Iterative Methods,*". σελίδες 1–6, 2014.

[43] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner και T. Mudge. "*Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation*". σελίδες 7–, 2003.

[44] L. Leem, H. Cho, J. Bau, Q. A. Jacobson και S. Mitra. "*ERSA: Error Resilient System Architecture for Probabilistic Applications*". σελίδες 1560–1565, 2010.