Πανεπιστήμιο Κύπρου
Τμήμα Πληροφορικής

Ατομική Διπλωματική Εργασία:

ΑΝΑΛΥΣΗ ΓΙΑ ΑΠΟΔΟΤΙΚΗ ΑΝΙΧΝΕΥΣΗ ΛΑΘΩΝ ΣΕ ΚΡΥΦΗ ΜΝΗΜΗ ΕΝΤΟΛΩΝ

Επιβλέπων Καθηγητής: Σαζεϊδης Γιάννος

Φοιτήτρια: Ντρέου Λορένα (Α.Π.Τ Z0524477)

ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2010

Ευχαριστίες

Θα ήθελα να εκφράσω ένα μεγάλο ευχαριστώ στον επιβλέποντα καθηγητή μου, κύριο Γιάννο Σαζεϊδη για την απεριόριστη και πολύτιμη βοήθεια που μου προσέφερε, και για την υπομονή του. Χωρίς την βοήθεια του δεν θα μπορούσα να φτάσω στον στόχο μου και να πραγματοποιήσω τα επόμενα όνειρα μου. Θα ήθελα να τον ευχαριστήσω επίσης που μου έδωσε την ευκαιρία να είμαι μέλος της ερευνητικής του ομάδας. Τον ευχαριστώ μέσα από την καρδιά μου και του εύχομαι ότι καλύτερο στην ζωή και στην σταδιοδρομία του. Επίσης θα ήθελα να ευχαριστήσω τον Μάριο Κλεάνθους, το Νικόλα Λαδά, την Bushra Ahsan, και τον Ισίδωρο Σιδέρη για την πολύτιμη βοήθεια που μου προσφέρανε. Θα ήθελα επίσης να ευχαριστήσω το Πανεπιστήμιο Κύπρου για την υποστήριξη που μου έδωσε στα πρώτα χρόνια των σπουδών μου.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Η ανάγκη για αξιοπιστία και ανοχή σφαλμάτων στις μνήμες υπολογιστών έχει αυξηθεί δραματικά. Η ύπαρξη σφαλμάτων στις διάφορες μνήμες υπολογιστών οδήγησε τους ερευνητές να αναπτύξουν ιδιαίτερο ενδιαφέρον για αυτό το θέμα. Τα σφάλματα στις μνήμες έχουν επιπτώσεις στην απόδοση των προγραμμάτων, και στην εγκυρότητα των δεδομένων που αποθηκεύονται σε αυτές. Αυτά τα σφάλματα μπορούν να εντοπιστούν και να διορθωθούν, προσθέτοντας κυκλώματα για την ανίχνευση και διόρθωση σφαλμάτων, γνωστά ως κώδικες EDC/ECC. Από την άλλη η ανίχνευση και η διόρθωση σφαλμάτων έχει κάποιο κόστος σε ενέργεια και χώρο.

Οι κρυφές μνήμες είναι οι πιο γρήγορες αλλά και οι πιο ακριβές από άποψη κόστους, διότι βρίσκονται πολύ κοντά στον επεξεργαστή (on chip). Αυτό είναι σημαντικό για την απόδοση του υπολογιστή, διότι ο χρόνος που χρειάζεται ο επεξεργαστής για να έχει πρόσβαση στα δεδομένα που βρίσκονται στην κρυφή μνήμη είναι μικρότερος από τον χρόνο που χρειάζεται να φέρει τα δεδομένα από την κύρια μνήμη. Ο όγκος των δεδομένων που μπορεί να αποθηκεύσει μια κρυφή μνήμη έχει περιορισμούς, διότι η θέση της κοντά στον επεξεργαστή δεν της επιτρέπει να έχει μεγάλο μέγεθος. Από την μια θέλουμε τα δεδομένα μας να είναι έγκυρα, αλλά από την άλλη αποθηκεύοντας τους κώδικες προστασίας μειώνουμε τον χώρο των πραγματικών δεδομένων που μπορούν να αποθηκευτούν στην κρυφή μνήμη. Η ανίχνευση και διόρθωση λαθών είναι απαραίτητη, αλλά το επίπεδο προστασίας των δεδομένων εξαρτάται από την εφαρμογή (π.χ. σε εφαρμογές ζωτικής σημασίας όπως τα συστήματα πλοήγησης των αεροπλάνων η εγκυρότητα των δεδομένων είναι απαραίτητη).

1.2 Σκοπός ατομικής διπλωματικής εργασίας

Σε αυτή την εργασία ο στόχος είναι η εξοικονόμηση χώρου και ενέργειας που καταναλώνεται για ανίχνευση και διόρθωση σφαλμάτων σε κρυφές μνήμες εντολών, δεδομένων κτλ.

Θέλοντας πάντα τα δεδομένα που αποθηκεύουμε σε αυτές τις μνήμες να είναι έγκυρα, οδηγούμαστε στην ανάγκη να προσθέτουμε έξτρα κώδικες για την ανίχνευση και διόρθωση λαθών. Προσθέτοντας αυτές τις έξτρα πληροφορίες απαιτείται επιπρόσθετος χώρος στην κρυφή μνήμη που πρέπει να χρησιμοποιηθεί για την αποθήκευση τους. Οι επιπρόσθετες μονάδες που προσθέτονται στην κρυφή μνήμη για να υπολογίσουμε αυτούς τους κώδικες, έχουν ως αποτέλεσμα την μεγαλύτερη κατανάλωση ενέργειας μέσα στο chip. Επίσης ο έξτρα χρόνος που χρειάζεται για να υπολογίσουμε αυτούς τους κώδικες έχει επιπτώσεις στην απόδοση των συστημάτων.

Αυτή η μελέτη κάνει μια ανάλυση του κατά πόσον είναι εφικτό να αποφύγουμε κάποια από τα bits που χρησιμοποιούνται για να ανιχνεύσουμε λάθη στην κρυφή μνήμη εντολών RISC, χωρίς να διακινδυνεύουμε την ακεραιότητα και την προστασία των δεδομένων. Χρησιμοποιώντας την πληροφορία που περιέχεται στην κωδικοποίηση των RISC εντολών, ο σκοπός είναι να μειώσουμε τον αριθμό των bits που χρειάζεται για την ανίχνευση σφαλμάτων σε κρυφή μνήμη εντολών, παρέχοντας την ίδια προστασία των δεδομένων με λιγότερο overhead το οποίο σημαίνει λιγότερο χώρο και λιγότερη κατανάλωση ενέργειας. Έχει αποδειχτεί ότι το 12.5 % του χώρου στην κρυφή μνήμη χρησιμοποιείται για την αποθήκευση της έξτρα πληροφορίας που χρησιμοποιείται για την ανίχνευση και την διόρθωση λαθών [7].

1.3 Οργάνωση Παρουσίασης Μελέτης

Η οργάνωση της παρουσίασης της μελέτης ακολουθεί ως εξής:

Στο δεύτερο κεφάλαιο παρουσιάζονται επεξηγήσεις των βασικών εννοιών Faults, Errors και Failures. Αναφέρουμε σε συντομία τις διάφορες χρονικές φάσεις στις οποίες μπορούν να εισάγονται τα διάφορα σφάλματα. Στην συνέχεια κατατάσσουμε τα διάφορα σφάλματα σε διάφορες κατηγορίες ανάλογα με την χρονική διάρκεια τους, δηλαδή εάν τα σφάλματα είναι μόνιμα, εάν είναι περιστασιακά ή εάν είναι soft . Επίσης για την κάθε κατηγορία σφαλμάτων επεξηγούμε τους διάφορους παράγοντες που οδηγούν σε αυτά, και τις συνθήκες κάτω από τις οποίες αυτά τα σφάλματα επηρεάζουν την αξιοπιστία των συστημάτων. Στο τρίτο κεφάλαιο γίνεται μια εισαγωγή στην κρυφή μνήμη και στην συνέχεια δίνεται μια πιο λεπτομερής περιγραφή για τα διάφορα είδη κρυφών μνημών και για κάποιες από τις πολιτικές που ακολουθούμε όταν γραφούμε ή διαβάζουμε την κρυφή

μνήμη. Στο τέταρτο κεφάλαιο γίνεται μια εισαγωγή στην ανίχνευση και διόρθωση λαθών και έπειτα αναφέρουμε κάποιες τεχνικές που χρησιμοποιούνται για αυτό το σκοπό. Στο πέμπτο κεφάλαιο κάνουμε μια εισαγωγή σε σφάλματα, ανίχνευση και διόρθωση σφαλμάτων σε κρυφές μνήμες και επίσης αναφέρουμε κάποιες τεχνικές που χρησιμοποιούνται για την προστασία τους από τα μόνιμα και soft λάθη. Στο έκτο κεφάλαιο παρουσιάζεται το κίνητρο αυτής της μελέτη, κάποια χαρακτηριστικά των RISC ISA που είναι ενδιαφέρον για την μελέτη αυτή. Παρουσιάζεται επίσης το λογικό σχήμα που προτείνουμε για να παρέχουμε την ίδια προστασία με λιγότερα parity bits. Στο κεφάλαιο επτά παρουσιάζεται η μεθοδολογία και τα πειραματικά αποτελέσματα που προέκυψαν από την ανάλυση. Στο όγδοο κεφάλαιο παρουσιάζονται τα συμπεράσματα από αυτή την ανάλυση, και μελλοντική εργασία που μπορεί να γίνει στο θέμα αυτό. Τέλος παρουσιάζεται η βιβλιογραφία.

Κεφάλαιο 2

Σφάλματα: παράγοντες που οδηγούν σε αυτά.

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο γίνεται μια εισαγωγή στα διάφορα είδη σφαλμάτων που μπορούν να προκαλέσουν βλάβη στα κυκλώματα από τα οποία αποτελούνται οι υπολογιστές. Ένα σημαντικό σημείο είναι να αναγνωρίσουμε τις πηγές από τις οποίες προέρχονται τα διάφορα είδη σφαλμάτων, ώστε να δίνεται περισσότερη έμφαση στο είδος της προστασίας που πρέπει να παρέχεται έτσι ώστε τα κυκλώματα να είναι αξιόπιστα. Όμως, είναι αδύνατον να αναγνωριστούν όλες οι πηγές σφαλμάτων. Για να προστατεύονται στο μέγιστο δυνατό βαθμό από τις γνωστές πηγές σφαλμάτων, γίνονται διάφοροι έλεγχοι στα κυκλώματα κατά την κατασκευή τους, αλλά και αυτός ο έλεγχος δεν προσφέρει 100% προστασία. Η ανάπτυξη και συντήρηση προτύπων για εφόρου ζωής αξιοπιστία, αποτελεί μια σημαντική αποστολή για τους κατασκευαστές μικροεπεξεργαστών.

Η σμίκρυνση της τεχνολογίας κάνει τα κυκλώματα πιο ευαίσθητα στα σφάλματα και προκαλεί αλλαγές σε διάφορες παραμέτρους όπως η θερμοκρασία, η ισχύς κτλ. Οι αλλαγές σε αυτές τις παραμέτρους θέτουν σε κίνδυνο την απόδοση των επεξεργαστών. Επίσης υπάρχουν και διάφορα μοντέλα που κάνουν εικασίες για τη συχνότητα με την οποία συμβαίνουν αυτά τα λάθη στα κυκλώματα.

2.2 Τι είναι τα Faults, Errors, Failures και πότε μπορούν να εισάγονται στα συστήματα.

Για να μπορούμε να κατανοήσουμε καλύτερα τις πηγές λαθών πρέπει πρώτα να κατανοήσουμε κάποιες σημαντικές έννοιες όπως το Fault , το Error και τα Failures.

Τα σφάλματα (faults) είναι η ανακριβής κατάσταση του υλικού ή του λογισμικού ως αποτέλεσμα της φυσικής ατέλειας. Τα σφάλματα μπορεί να εισάγονται σε διάφορες χρονικές φάσεις, όπως για παράδειγμα τα σφάλματα που εισάγονται κατά την διάρκεια

σχεδίασης των συστημάτων, τα σφάλματα που προέρχονται κατά τη διάρκεια της κατασκευής και τα σφάλματα που εμφανίζονται κατά τη λειτουργία.

Τα λάθη (errors) είναι η εκδήλωση ενός σφάλματος ή ελαττώματος. Ορίζονται ως λάθη που προέρχονται ως αποτέλεσμα ενός σφάλματος. Είναι πολύ σημαντικό να αναφέρουμε ότι όλα τα σφάλματα δεν οδηγούν απαραίτητα σε λάθη. Αυτά τα αναγνωρίζουμε ως κρυμμένα λάθη (masked errors). Όπως για παράδειγμα έχουμε λογικά κρυμμένα λάθη εάν μια πύλη AND έχει ένα λάθος input και το άλλο input είναι 0 τότε το λάθος δεν μπορεί να διαδοθεί.

Οι αποτυχίες (failures) είναι το επίπεδο επίδρασης ενός συστήματος ως αποτέλεσμα κάποιου λάθους το οποίο είναι εμφανές στους χρήστες. Είναι σημαντικό να αναφέρουμε ότι όχι όλα τα λάθη οδηγούν σε αποτυχία του συστήματος. Αυτά τα λάθη τα αναγνωρίζουμε ως κρυμμένα λάθη, όπως για παράδειγμα η αλλαγή ενός bit από 0 σε 1 σε μια θέση μνήμης στην οποία δεν αναφερόμαστε ποτέ.

2.3 Κατηγορίες σφαλμάτων ανάλογα με την διάρκεια τους

Ανάλογα με την διάρκεια μπορούμε να κατατάσσουμε τα σφάλματα σε τρεις κατηγορίες:

Πρώτη κατηγορία είναι τα **soft** errors [1] τα οποία ορίζονται ως αυθόρμητα λάθη ή αλλαγές στις αποθηκευμένες πληροφορίες που δεν μπορούν να αναπαραχθούν. Υπάρχουν διάφοροι παράγοντες από τους οποίους μπορούν να προκύψουν τέτοια σφάλματα, οι οποίοι παρουσιάζονται πιο κάτω:

Πρώτος παράγοντας είναι τα φυσικά φαινόμενα (physical defects) όπως η κοσμική ραδιενέργεια, και η ραδιενέργεια που δημιουργείται από τα σωματίδια Alpha. Σε αυτή την περίπτωση η ακτινοβολία προέρχεται από την αποσύνθεση μετάλλων. Το πρόβλημα προέρχεται από το γεγονός ότι μειώνοντας το μέγεθος των transistors μειώνεται και η πιθανότητα μια ακτινοβολία να χτυπήσει ένα transistor, αλλά σε περίπτωση που συμβεί αυτό τότε έχει μεγαλύτερη πιθανότητα να καταστραφεί το transistor. Έχοντας υπόψη ότι ο αριθμός των transistors στο ίδιο χώρο διπλασιάζεται κάθε δυο χρόνια (Moore´s Law) τότε υπάρχει μεγαλύτερη πιθανότητα να καταστραφεί ένα chip από τέτοιους παράγοντες αν και αυτό μπορεί να συμβεί σπάνια.

Μια άλλη φυσική ατέλεια είναι το Electromagnetic Interference, το οποίο συμπεριλαμβάνει ηλεκτρομαγνητικά κύματα τα οποία μπορούν να δημιουργηθούν από το ίδιο το κύκλωμα, καθώς επίσης και ηλεκτρομαγνητικά κύματα από άλλες πηγές (π.χ. φούρνος μικροκυμάτων, ηλεκτροφόρα καλώδια, κτλ.) τα οποία μπορούν να προκαλέσουν τις παροδικές διασπάσεις,

Είναι σημαντικό να αναφέρουμε ότι η αποσύνθεση των ραδιενεργών ατόμων που υπάρχουν σε μεγάλο ποσοστό σε όλα τα υλικά, και οι εξωγήινες κοσμικές ακτίνες οι οποίες βομβαρδίζουν τη γη συνεχώς από τα μακρινά βάθη του γαλαξία, είναι δυο σημαντικές πηγές που προκαλούν soft λάθη στα κυκλώματα. Επίσης είναι πολύ σημαντικό να αναφέρουμε ότι τα soft λάθη σε διάφορα κυκλώματα μπορεί να μην έχουν επιπτώσεις στους χρήστες υπολογιστών. Όπως για παράδειγμα εάν ένα bit στην μνήμη έχει αλλάξει από 0 σε 1 και το σύστημα κλείνει πριν να χρησιμοποιηθεί η τιμή αυτής της θέσης τότε δεν υπάρχει κίνδυνος για σφάλμα. Υπάρχει επίσης και η περίπτωση που το ανακριβές bit να μπορεί να επικαλυφθεί προτού να χρησιμοποιηθεί. Για να είναι τα συστήματα αξιόπιστα η ανίχνευση και η διόρθωση λαθών υπολογιστών είναι πάρα πολύ σημαντική. Αυτό είναι δυνατό είτε προσθέτοντας έξτρα κυκλώματα για την ανίχνευση και διόρθωση λαθών, είτε επιλέγοντας πολύ αξιόπιστα συστατικά αναίσθητα στην ακτινοβολία. Όπως έχουμε αναφέρει και πιο πάνω η αξιοπιστία των κυκλωμάτων κινδυνεύει από την ραδιενέργεια των alpha σωματιδίων. Έχει αποδειχτεί σύμφωνα με το [1] ότι ένα alpha σωματίδιο μπορεί να προκαλέσει μια ξαφνική έκρηξη των 10^6 ηλεκτρόνιων σε semiconductor πέρα από ένα μήκος κλίμακας των microns. Το πρόβλημα αυτό αποκαλύφτηκε το 1978 από τους May and Wood μετά από λειτουργικά λάθη που υπήρχαν σε μια σειρά 2107 16Kb DRams που κατασκευάστηκαν από την Intel. Ανακάλυψαν ότι το πρόβλημα ήταν τα ίχνη ραδιενέργειας στα υλικά συσκευασίας μνήμης.

Μια άλλη σημαντική πηγή των soft errors είναι οι κοσμικές ακτίνες οι οποίες έρχονται από το βάθος του γαλαξία και είναι άγνωστη η πηγή τους. Αυτές οι ακτίνες, οι οποίες έχουν απέραντες ενέργειες και βομβαρδίζουν τη γη από όλες τις πλευρές, μπορούν να χτυπούν τον πυρήνα των πυριτίων (silicones) και να τον σπάζουν σε μικρά θραύσματα. Το κάθε ένα από αυτά τα θραύσματα παράγουν ένα ρεύμα ηλεκτρικών δαπανών που μπορούν να ανατρέψουν σχεδόν οποιοδήποτε κύκλωμα.

Το υλικό από το οποίο κατασκευάζονται τα κυκλώματα και επίσης το υλικό διαφορών εργαλείων που χρησιμοποιούνται κατά την κατασκευή των chips είναι αιτίες soft errors. Είναι σημαντικό να αναφέρουμε ότι ένα chip αποτελείται από διάφορες μονάδες οι οποίες έχουν κατασκευαστεί με διαφορετικό υλικό το οποίο αποτελείται από διάφορα σωματίδια. Αρκετές φορές αυτά τα σωματίδια αντιδρούν μεταξύ τους προκαλώντας soft errors. Τα πιο αξιόπιστα κυκλώματα και τα πιο ανθεκτικότατα στα soft errors είναι τα CMOS (Complementary metal–oxide–semiconductor) και n-MOS static arrays. Τα κυκλώματα τα οποία είναι περισσότερα ευαίσθητα στα soft errors είναι τα bipolar SRAMs. Για να έχουμε αξιόπιστα συστήματα πρέπει να λαμβάνουμε υπόψη τα πιο πάνω σφάλματα και την συχνότητα με την οποία συμβαίνουν.

Η δεύτερη κατηγορία είναι τα **περιστασιακά** λάθη (Intermittent) τα οποία εμφανίζονται όταν για παράδειγμα έχουμε μια χαλαρή σύνδεση μέσα στο σύστημα μπορεί να προκύψει βραχυκύκλωμα του συστήματος από ένα ανοικτό κύκλωμα [12].

Η τρίτη κατηγορία είναι τα **μόνιμα** λάθη (hard): Τα μόνιμα λάθη εμφανίζονται και παραμένουν στα συστήματα, όπως για παράδειγμα μια σπασμένη σύνδεση οδηγεί πάντα σε ανοιχτό κύκλωμα ή για παράδειγμα μία θέση της μνήμης η οποία δεν μπορεί να χρησιμοποιηθεί λόγω σφαλμάτων [2].

Οι παράγοντες από τους οποίους μπορεί να προκύψουν τέτοια σφάλματα είναι:

- Κατασκευαστικά ελαττώματα: Η κατασκευή δεν είναι μια τέλεια διαδικασία, ειδικά για τους μικροεπεξεργαστές. Είναι πολύ δύσκολο να κατασκευάζεις transistors με διαστάσεις σε κλίμακα των 45nm. Ένα άλλο κατασκευαστικό πρόβλημα είναι η κακή σύνδεση μεταξύ του chip και το board. Στην προσπάθεια τα καλώδια να είναι πολύ λεπτά (κάτω από κάποιο threshold) προκύπτει το πρόβλημα να αυξηθεί η αντίσταση έτσι το σήμα είναι πάρα πολύ αργό για το ρολόι. Η μεταβλητότητα αυτή μπορεί να οδηγήσει σε ανεπιθύμητες συμπεριφορές του συστήματος.
- Λειτουργικά ελαττώματα: Σε αυτή την περίπτωση οι ατέλειες μπορούν να εμφανιστούν κατά τη λειτουργία του συστήματος. Αυτά τα λάθη μπορεί να προκύψουν από το:
    a. Electromigration. Εάν η πυκνότητα ρεύματος είναι πάρα πολύ μεγάλη για το καλώδιο (wire), τα ηλεκτρόνια του μετάλλου από το οποίο

αποτελούνται οι συνδέσεις, «θα μεταναστεύσουν» μακριά και ενδεχομένως να καταλήξει σε μια σπασμένη σύνδεση.

b. Design Bugs: Αυτά τα bugs προέρχονται από προβλήματα όπως το ότι είναι πολύ δύσκολο να γίνει μια πλήρης επικύρωση ενός σύνθετου επεξεργαστή, από την πολυπλοκότητα σχεδιασμού των επεξεργαστών Οι έλεγχοι που γίνονται μετά την κατασκευή των κυκλωμάτων δεν μπορούν να καλύψουν 100% όλες τις περιπτώσεις. Υπάρχουν επίσης προβλήματα συγχρονισμού (Timing Bugs) όπως για παράδειγμα εάν η ταχύτητα ρολογιού ενός επεξεργαστή είναι 4GHz και υπάρχει μια διαδρομή στον αγωγό (pipeline) η οποία χρειάζεται 3.8 GHz, μπορεί να προκύψει κάτι το οποίο να κάνει αυτό το στάδιο να γίνει πιο αργό (πχ λόγο της αύξησης της θερμοκρασίας) και έτσι δεν θα προλαβαίνουν να ενημερωθούν με τις σωστές τιμές οι διάφορες μονάδες στο τέλος του κύκλου μηχανής.

2.4 Πως επηρεάζεται η αξιοπιστία των συστημάτων από την σμίκρυνση της τεχνολογίας.

Η σμίκρυνση της τεχνολογίας έχει ως αποτέλεσμα όλο και περισσότερο μικρότερο μέγεθος των transistors, το οποίο προκαλεί αυξανόμενη πυκνότητα ισχύος. Επίσης η αύξηση της πυκνότητας ρεύματος προκαλεί αύξηση της θερμοκρασίας. Ταυτόχρονα η τάση παροχής και το threshold voltage δεν μειώνεται ανάλογα με το μέγεθος των transistors. Για το λόγο αυτό μειώνεται η απόδοση και αυξάνεται η στατική ισχύος (leakage power). Κατασκευάζοντας κυκλώματα με μικρότερα transistors και λαμβάνοντας υπόψη τις πιο πάνω επιπτώσεις επηρεάζεται σε ένα μεγάλο βαθμό η θερμοκρασία και αξιοπιστία των επεξεργαστών, και αυξάνονται εκθετικά τα wearout failures.

Συστηματικές και τυχαίες παραλλαγές σε διάφορες παραμέτρους όπως η διαδικασία, η τάση παροχής και η θερμοκρασία (P,V,T) θέτουν μια σημαντική πρόκληση στο μελλοντικό σχεδιασμό μικροεπεξεργαστών υψηλής επίδοσης [3] . Οι διακυμάνσεις στις

τιμές τους έχουν επιπτώσεις στην επίδοση των κυκλωμάτων των μικροεπεξεργαστών. Μειώνοντας το μήκος των transistors πέρα των 90nm προκαλεί υψηλότερα επίπεδα διακυμάνσεων αυτών των παραμέτρων σε διάφορες συσκευές. Πιο κάτω επεξηγείται το τι προκαλούν οι διάφορες αλλαγές σε αυτές τις παραμέτρους:

- Process Variation: είναι πάντα μια κρίσιμη πτυχή της επεξεργασίας των semiconductors. Οι αλλαγές στις παραμέτρους των μεμονωμένων transinstors, αναγκάζουν τις ηλεκτρικές παραμέτρους να ποικίλουν, όπως η αντίσταση φύλλων(sheet resistance) και η τάση κατώτατων ορίων(threshold voltage). Επίσης οι αλλαγές σε αυτές τις παραμέτρους προκαλούν αλλαγές στη διανομή της συχνότητας και το leakage power. Η απαίτηση για χαμηλή ισχύ (Power) προκαλεί διακύμανση στην τάση (V). Επίσης η απαίτηση για την αύξηση της συχνότητας προκαλεί αυξημένη θερμοκρασία μεταξύ των συνδέσεων και επίσης διακυμάνσεις της θερμοκρασίας μέσα στα die. Προσέχουμε ότι τα chips με μέγιστη συχνότητα έχουν μια ευρεία διανομή του leakage και επίσης για μια δεδομένη διαρροή υπάρχει μια ευρεία διάταξη της συχνότητας. Σύμφωνα με το [3] οι διακυμάνσεις στο voltage προκαλούν παραλλαγές στην επίδοση και στο leakage των κυκλωμάτων. Ο ακριβής έλεγχος της Vth είναι πολύ σημαντικός. Επίσης οι αυξομειώσεις του leakage ρεύματος προκαλούνται και από τις αλλαγές στο μήκος των καναλιών στα κυκλώματα. Συμπεραίνουμε ότι αυξάνοντας το Vth, μειώνεται η συχνότητα και επίσης το leakage. Εάν μικραίνει το Vth τότε αυξάνεται το leakage και το frequency.
- Supply voltage variations: Εάν το Vdd>Vmax τότε έχουμε μεγαλύτερη κατανάλωση ενέργειας και αυξημένη θερμοκρασία. Εάν το Vdd<Vmin μειώνεται η κατανάλωση ενέργειας , μεγαλύτερο ρίσκο στην επίδοση . Σε αυτές τις περιπτώσεις το σύστημα δεν δουλεύει σωστά
- Temperature Variations: Όταν αυξάνεται η θερμοκρασία τότε έχουμε περισσότερο delay, και αυξάνεται το leakage power το οποίο προκαλεί επίσης και άλλη αύξηση της θερμοκρασίας. Είναι σημαντικό να αναφέρουμε ότι μέσα στο ίδιο chip υπάρχουν σημεία τα οποία έχουν μεγάλη αλλαγή στην θερμοκρασία μεταξύ τους (τα γνωστά hot spots). Αυτό σημαίνει ότι ανεξάρτητα αν οι αποστάσεις μεταξύ διαφόρων κυκλωμάτων μέσα στο ίδιο chip είναι πάρα πολύ

μικρές της τάξης των nm, η θερμοκρασία τους διαφέρει πάρα πολύ και αυτό έχει να κάνει με την συχνότητα χρήσης τους, δηλαδή τα κυκλώματα με αυξημένη θερμοκρασία χρησιμοποιούνται πιο συχνά και επίσης μπορεί να εκτελούν πιο σύνθετες λειτουργίες.

2.5 Τεχνικές για την μέτρηση του ποσοστού soft errors

Όπως έχουμε τονίσει και πιο πριν είναι πολύ σημαντικό να καθορίσουμε τις πήγες σφαλμάτων και κάτω από ποιες συνθήκες αυτές είναι καταστροφικές για τα κυκλώματά μας. Για αυτό το λόγο διάφορες τεχνικές χρησιμοποιούνται για να μετρήσουμε το SER (soft error rate) έτσι ώστε να καθορίσουμε κάτω από ποιες συνθήκες τα κυκλώματα μας κινδυνεύουν από τις κοσμικές ή από την ραδιενέργεια των σωματιδίων alpha.

Μια τεχνική που χρησιμοποιείται για την μέτρηση του ποσοστού των soft errors είναι ο επιταχυνόμενος έλεγχος της ευαισθησίας των LSI chips κάτω από κοσμικές ακτίνες [1]. Τα chips βομβαρδίζονται από ακτίνες μορίων που θεωρείται ότι υπάρχουν στα cosmic rays. Η δυσκολία εδώ έχει να κάνει με το γεγονός ότι η ροη των σωματιδίων που χρησιμοποιούνται κατά τον έλεγχο μπορεί να μην είναι η ίδια με το μίγμα των σωματιδίων που αποτελούν τα cosmic rays και να μην έχουν την ίδια ενέργεια με τα cosmic rays που φτάνουν στα επίγεια ύψη. Η μέση ενέργεια η οποία θα δημιουργούσε ανατροπή στα κυκλώματα είναι 1GeV.

Μια άλλη τεχνική είναι το field testing. Η βασική ιδέα εδώ είναι να ελεγχθούν τα διάφορα συστατικά των chips σε διάφορα ύψη δηλαδή ύψη όπως στο επίπεδο της θάλασσας κτλ και να υπολογιστεί το SER(soft error rate) ανάλογα και με τα επίγεια ύψη. Αυτός ο έλεγχος είναι σημαντικός για το λόγο ότι ανάλογα με τα επίγεια ύψη αλλάζει η ενεργεία των ακτινών και επίσης τα σωματίδια που αποτελούν αυτές τις ακτίνες, τα οποία προκαλούν τα soft errors.

2.5 Wearout failure (μοντέλα)

Υπάρχουν διάφορα μοντέλα που επιτρέπουν lifetime-reliability-aware ανάλυση των εφαρμογών και των αρχιτεκτονικών[2]. Κάποια από αυτά τα μοντέλα για να

αξιολογήσουν την αξιόπιστα των επεξεργαστών μετρούν το MTTF(mean time to failure) το οποίο εξαρτάται από την χρήση και από την διάρκεια ζωής των επεξεργαστών. Μία άλλη τεχνική η οποία αφήνει την κατασκευή, για να πιστοποιήσει την αξιοπιστία στο χαμηλότερο αλλά πιθανότερο λειτουργικό σημείο είναι η DRM(dynamic reliability management) ή οποία προσπαθεί με αυτό τον τρόπο να μειώσει το κόστος της αξιόπιστης σχεδίασης των κυκλωμάτων. Αυτό το μοντέλο μας δίνει την δυνατότητα να σχεδιάσουμε κυκλώματα στα οποία η απόδοση και η αξιοπιστία δεν μπορεί να πέφτει κάτω από κάποιο όριο. Τα πιο κάτω μοντέλα (προσεγγίσεις) δείχνουν ποιες είναι οι πηγές λαθών και πότε μπορούν να προκαλέσουν wearout σφάλμα και εκφράζουν την αξιοπιστία των κυκλωμάτων από την άποψη του MTTF(mean time to failure). Τα μοντέλα αυτά δεν μοντελοποιούν μεταβατικές λειτουργίες αλλά μοντελοποιούν λειτουργίες σε καταστάσεις που υπάρχει ισορροπία.

- Μετανάστευση ηλεκτρονίων (Electromigration): Εάν αυξάνεται η πυκνότητα του ρεύματος μέσα στις συνδέσεις επεξεργαστών τότε τα άτομα μετάλλων που αποτελούν αυτές τις συνδέσεις έχουν μεγάλη πιθανότητα να "μεταναστεύσουν" από μια άκρη της σύνδεσης στην άλλη, προκαλώντας έτσι αύξηση της αντίστασης μέσα στις συνδέσεις και βραχυκύκλωμα**.** Σε αυτό το μοντέλο το MTTF εξαρτάται από την πυκνότητα ρεύματος στις συνδέσεις, από την ενέργεια ενεργοποίησης για την μετανάστευση ηλεκτρονίων (electromigration), την θερμοκρασία και επίσης από κάποιες σταθερές που εξαρτούνται από το μέταλλο που χρησιμοποιείται στις συνδέσεις.
- Stress Migration: Σε αυτή την περίπτωση η "μετανάστευση" των ατόμων στις συνδέσεις προκαλείται από θερμική πίεση. Αυτό έχει να κάνει με το γεγονός ότι οι συνδέσεις αποτελούνται από διάφορα υλικά τα οποία διαφέρουν στο ποσοστό θερμικής επέκτασης και αυτή η διαφορά προκαλεί θερμομηχανική πίεση. Σε αυτή την προσέγγιση το MTTF εξαρτάται από την θερμοκρασία του μετάλλου όταν αυτό είναι stress-free και επίσης από κάποιες σταθερές που εξαρτούνται από το μέταλλο.
- Time-dependent dielectric breakdown: Είναι γνωστό και ως gate oxide breakdown, και είναι αποτέλεσμα της gate dielectric's βαθμιαίας φθοράς το οποίο προκαλεί αποτυχία των transistor. Μειώνοντας τις διαστάσεις των transistors

κινδυνεύει η αξιοπιστία των Gate oxide, επίσης τα ηλεκτρικά πεδία μέσα στο gate oxide γίνονται μεγαλύτερα .Έχοντας περισσότερα transistors σε ένα chip είναι πιο μικρά και τα gate oxide και έχουν μεγαλύτερη πιθανότητα failures .

- Thermal cycling: Είναι σημαντικό να αναφέρουμε ότι απότομες αλλαγές της θερμοκρασίας στα chip προκαλούν φθορά όπως για παράδειγμα η αλλαγή θερμοκρασίας από αυτή του περιβάλλοντος σε αυτή του chip. Επίσης και μικρές αλλαγές στην θερμοκρασία μπορεί να προκαλέσουν φθορά.

Κεφάλαιο 3

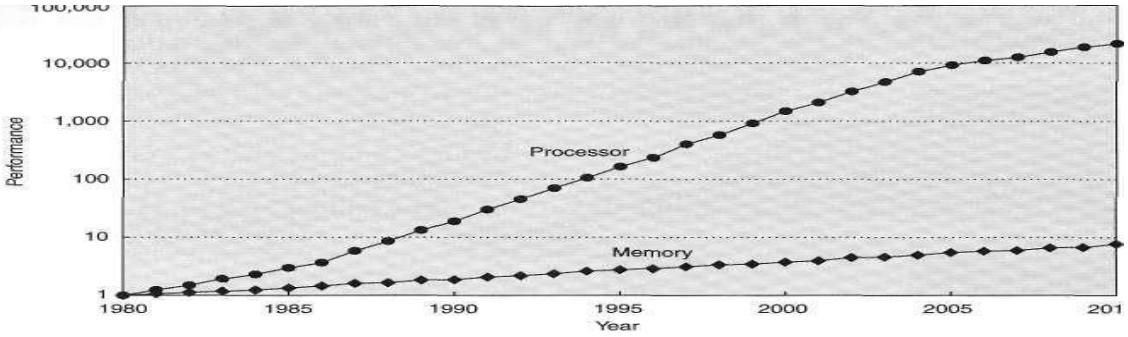
Κρυφή Μνήμη

3.1 Εισαγωγή στην κρυφή μνήμη

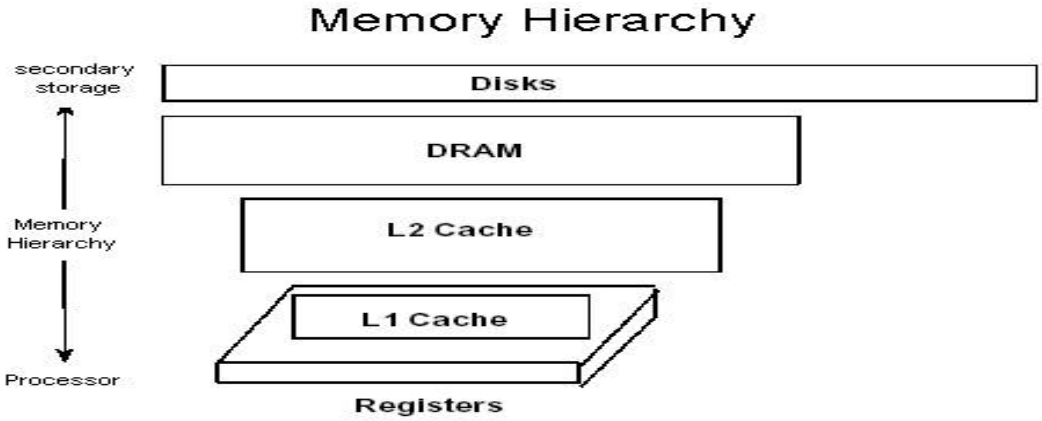
Η κρυφή μνήμη είναι μια μικρή και γρήγορη, συνήθως στατική, μνήμη SRAM η οποία περιέχει τα πιο πρόσφατα προσεγγισμένα δεδομένα της κύριας μνήμης. Η ανάγκη ύπαρξης της κρυφής μνήμης προέκυψε για το λόγο ότι η πρόσβαση στα δεδομένα στην κύρια μνήμη είναι πολύ χρονοβόρα. Για αυτό το λόγο υπάρχει ένα χάσμα μεταξύ της ταχύτητας των επεξεργαστών και τον χρόνο πρόσβασης στην μνήμη η οποία επηρεάζει την απόδοση των συστημάτων (Σχήμα 3.1).

Η αρχή της τοπικότητας, το χάσμα μεταξύ της ταχύτητας των επεξεργαστών, της ταχύτητας με την οποία έχουμε πρόσβαση στην μνήμη και το γεγονός ότι το μικρότερο υλικό είναι πιο γρήγορο οδήγησε στην ιεραρχία μνήμης (Σχήμα 3.2) .Για αυτό το λόγο σχεδιαστήκαν μικρά και γρήγορα κομμάτια μνήμης, τα οποία βοηθούν στην πιο γρήγορη μεταφορά των δεδομένων προς και από τον επεξεργαστή, βελτιώνοντας έτσι την απόδοση των συστημάτων. Η θεωρία που εξηγεί αυτό το γεγονός είναι γνωστή ως **“Locality of Reference”** [4]. Η αρχή της τοπικότητας λέει ότι τα περισσότερα προγράμματα δεν έχουν πρόσβαση σε όλο τον κώδικα ή στα δεδομένα ομοιόμορφα. Η τοπικότητα εμφανίζεται σε χρόνο (χρονική τοπικότητα) και σε χώρο (χωρική τοπικότητα). Σύμφωνα με την χρονική τοπικότητα, υπάρχει μεγάλη πιθανότητα εάν έχουμε πρόσβαση σε μια θέση μνήμης κάποια στιγμή, τότε αυτή η θέση μνήμης να ξαναχρησιμοποιηθεί στο πιο πρόσφατο μέλλον και είναι καλά αυτά τα δεδομένα να τοποθετηθούν στην κρυφή μνήμη. Σύμφωνα με την χωρική τοπικότητα εάν έχουμε πρόσβαση σε μια θέση μνήμης κάποια στιγμή, είναι πιθανό ότι στο κοντινό μέλλον θα έχουμε πρόσβαση σε γειτονικές θέσεις μνήμης με αυτή, για αυτό το λόγο αποθηκεύουμε στην κρυφή μνήμη και τα δεδομένα που βρίσκονται σε γειτονικές διευθύνσεις .

Εφόσον τα γρήγορα κομμάτια μνήμης κοστίζουν περισσότερα η ιεραρχία μνήμης οργανώνεται σε διάφορα επίπεδα, όπου το κάθε ένα είναι μικρότερο, γρηγορότερο, και ακριβότερο ανά byte από το επόμενο χαμηλότερο επίπεδο.



Σχήμα 3.1: Το χάσμα μεταξύ επεξεργαστή και κύριας μνήμης



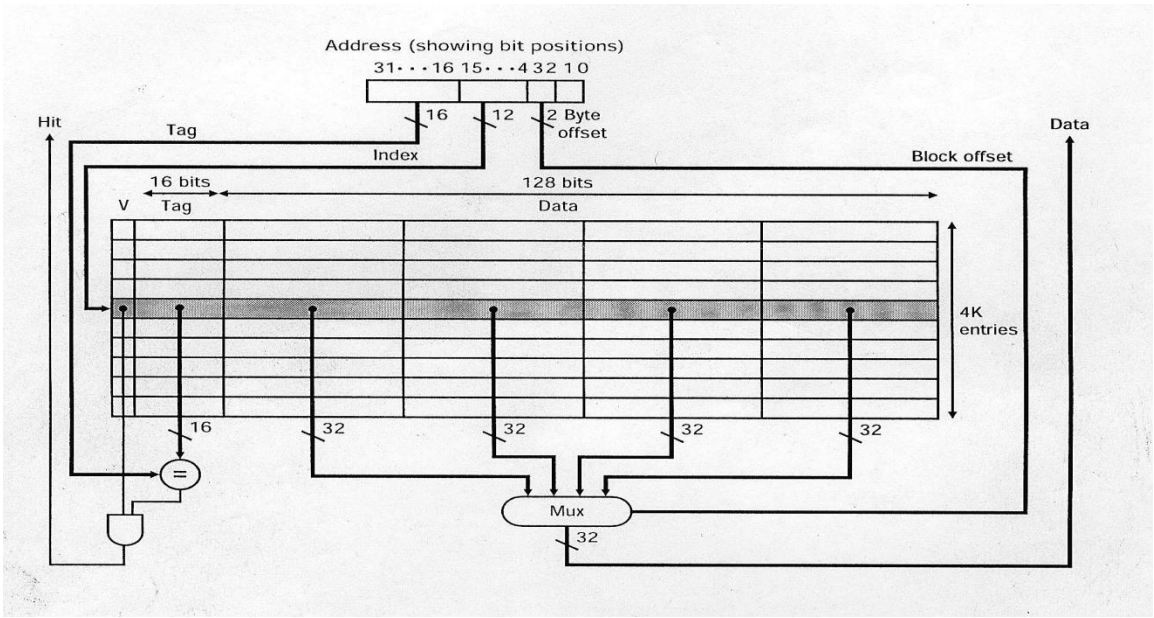
Σχήμα 3.2: Ιεραρχία Μνήμης

3.2 Οργάνωση δεδομένων σε κρυφές μνήμες, είδη τους και βασικές στρατηγικές χρήσης των κρυφών μνημών

Τα δεδομένα στην κρυφή μνήμη οργανώνονται σε blocks τα οποία περιέχουν πολλαπλά words. Ένα σύνολο από blocks οργανώνονται σε set. Είναι σημαντικό να καθοριστεί που μπορούμε να αποθηκεύσουμε τα δεδομένα που φέρνουμε από την μνήμη στην κρυφή μνήμη και επίσης από ποία θέση της κρυφής μνήμης μπορούμε να διαβάσουμε τα δεδομένα όταν έχουμε read hit. Το πιο γνωστό σχήμα είναι το set associative. Σύμφωνα με αυτό, όταν φέρνουμε blocks από την κύρια μνήμη πρέπει πρώτα να καθοριστεί σε ποιο set στην κρυφή μνήμη θα πρέπει να τοποθετηθεί, και επίσης σε ποιές θέσεις μνήμης μέσα στο set θα μπορέσει να τοποθετηθεί. Εάν το κάθε set

αποτελείται από n blocks τότε η κρυφή μνήμη λέγεται ότι είναι *n way set associative* , εάν έχουμε ένα block για κάθε set τότε λέμε ότι έχουμε *direct-mapped* κρυφή μνήμη και εάν έχουμε μόνο ένα σετ για όλη την κρυφή μνήμη τότε λέγεται *fully associative* κρυφή μνήμη. Στην πρώτη περίπτωση όταν φέρνουμε ή διαβάζουμε ένα block σε ένα *n way set associative* κρυφή μνήμη πρέπει στην αρχή να καθοριστεί σε ποιο set πρέπει να έχουμε πρόσβαση και μετά σε ποια από τις n θέσεις του set βρίσκεται ή πρέπει να τοποθετηθεί . Όταν η κρυφή μνήμη είναι *direct-mapped* τότε έχουμε μόνο ένα block ανά set. Όταν η κρυφή μνήμη είναι *fully associative* τότε το block μπορεί να τοποθετηθεί σε n διαφορετικές θέσεις. Κάθε γραμμή της κρυφής μνήμης περιλαμβάνει το Tag για το κάθε block, το οποίο χρησιμοποιείται για να ελέγξει όλα τα blocks σε ένα set. Σε περίπτωση που η διεύθυνση αντιστοιχεί με το tag το οποίο είναι αποθηκευμένο στην κρυφή μνήμη ξέρουμε ότι τα δεδομένα που θέλουμε βρίσκονται στην κρυφή μνήμη και μπορούμε να τα διαβάζουμε.

Το index χρησιμοποιείται για να επιλέξει το σωστό set στο οποίο θα πρέπει να τοποθετηθούν ή να διαβαστούν τα δεδομένα, και το block offset είναι η διεύθυνση των επιθυμητών δεδομένων μέσα στο block (Σχήμα 3.3). Το valid bit (v) δείχνει εάν αυτό το line φυλάσσει δεδομένα, το dirty bit (d) δείχνει εάν τα δεδομένα που βρίσκονται σε κάποιο σετ έχουν αλλάξει και δεν είναι τα ίδια με τα δεδομένα που βρίσκονται στην αντίστοιχη θέση μνήμης.



Σχήμα 3.3: Direct map κρυφή μνήμη

Είναι πολύ σημαντικό τα δεδομένα στην μνήμη και στην κρυφή μνήμη να είναι συνεπή. Για αυτό τον λόγο υπάρχουν δυο κύριες στρατηγικές που ακολουθούνται όταν γράφουμε στην κρυφή μνήμη.

1. Write through: Σε αυτή την περίπτωση ενημερώνεται το block και στην κρυφή μνήμη και στην κύρια μνήμη
2. Write back: Το block ενημερώνεται μόνο στην κρυφή μνήμη και πριν να αντικατασταθεί από ένα άλλο block τότε ενημερώνεται η κύρια μνήμη με τα δεδομένα του.

Όταν διαβάζουμε από την κρυφή μνήμη το block μπορεί να διαβαστεί συγχρόνως με τον έλεγχο του tag(το tag διαβάζεται και ελέγχεται έτσι ώστε να διαβάζουμε τα σωστά δεδομένα μόλις είναι έτοιμη η διεύθυνση). Εάν έχουμε hit τα δεδομένα μεταφέρονται στον επεξεργαστή, εάν έχουμε read miss τότε μπορούμε να ψάξουμε στα άλλα επίπεδα κρυφής μνήμης εάν υπάρχουν ή στην κύρια μνήμη και στην χειρότερη περίπτωση στο δίσκο.

Ένα μέτρο των οφελών των διαφορετικών οργανώσεων των κρυφών μνημών είναι το miss rate, το οποίο είναι το ποσοστό των προσβάσεων στην κρυφή μνήμη που οδηγούν σε αποτυχία (miss). Σε αυτή την περίπτωση τα δεδομένα που θέλουμε να έχουμε πρόσβαση δεν βρίσκονται στην κρυφή μνήμη και πρέπει να τα φέρουμε από την κύρια μνήμη. Άλλοι σημαντικοί παράμετροι που πρέπει να λαμβάνουμε υπόψη είναι το *average memory access time= Hit time + Miss rate X Miss penalty.*

Το average memory access time εξαρτάται από το Hit Rate το οποίο είναι ο χρόνος για να έχουμε πρόσβαση στην κρυφή μνήμη, και το miss penalty είναι ο χρόνος που χρειάζεται να αντικαταστήσουμε το block από τη μνήμη. Είναι σημαντικό να αναφέρουμε ότι έχοντας μεγαλύτερη κρυφή μνήμη υπάρχει μεγαλύτερο κόστος, αν και μπορούμε με αυτό τον τρόπο να μειώσουμε το miss rate. Για αυτό το λόγο στην ιεραρχία μνήμης μπορούμε να έχουμε και κρυφή μνήμη πολλαπλών επιπέδων. Η κρυφή μνήμη πρώτου επιπέδου μπορεί να είναι αρκετά μικρή και η πρόσβαση σε αυτή να είναι πιο γρήγορη, και τα άλλα επίπεδα κρυφής μνήμης πχ L2, L3 δεν είναι τόσο γρήγορα όσο η L1 αλλά μπορεί να είναι αρκετά μεγάλα να συλλάβει πολλές προσβάσεις που θα πήγαιναν στην κύρια μνήμη. Επίσης οι προσβάσεις σε αυτά τα επίπεδα είναι πιο γρήγορες από τις προσβάσεις στην κύρια μνήμη. Με αυτό τον τρόπο στις περισσότερες

περιπτώσεις αποφεύγουμε την πιο χρονοβόρα πρόσβαση στην κύρια μνήμη και προσπαθούμε να κρατήσουμε περισσότερα δεδομένα από το τι θα μπορούσαμε να κρατήσουμε μόνο στην L1 κρυφή μνήμη. Όταν θέλουμε να φέρουμε ένα block από την μνήμη και δεν υπάρχει άδειος χώρος στο set στο οποίο πρέπει να τοποθετηθεί τότε ακολουθούμε κάποιες στρατηγικές αντικατάστασης:

1. First-in first-out: Αντικαταστείτε το block το οποίο είναι στην κρυφή μνήμη για ένα μεγάλο χρονικό διάστημα.
2. Least recently used: Αντικαταστείτε το block το οποίο έχει χρησιμοποιηθεί λιγότερα.
3. Random replacement: Το block το οποίο θα πρέπει να αντικατασταθεί επιλέγεται τυχαία.

Κεφάλαιο 4

Τρόποι για ανίχνευση και διόρθωση λαθών

4.1 Εισαγωγή στην ανίχνευση και διόρθωση λαθών

Για να προστατέψουμε τα συστήματα μας από διάφορες πηγές λαθών υπάρχουν αρκετές τεχνικές οι οποίες χρησιμοποιούνται για την ανίχνευση και για την διόρθωση λαθών. Γενικά μπορούμε να προστατεύουμε τα δεδομένα προσθέτοντας έξτρα bits σε αυτή. Είναι γεγονός ότι προσθέτοντας έξτρα bits στην πληροφορία αυξάνεται το μέγεθος των δεδομένων που αποστέλλονται ή που αποθηκεύονται αλλά δίνει την δυνατότητα να ανιχνεύσουμε ή να διορθώσουμε τυχόν λάθη που μπορεί να έχουν προκύψει στα δεδομένα. Αυτή η έξτρα πληροφορία οδηγεί στη ανάγκη για επιπρόσθετες πράξεις για την κωδικοποίηση και για την αποκωδικοποίηση των δεδομένων έτσι ώστε στο σύνολο της πληροφορίας να διαχωρίζουμε ποια bit αντιστοιχούν στα πραγματικά δεδομένα και ποια bits είναι για προστασία των δεδομένων.

Οι πηγές λαθών που έχουμε αναφέρει πιο πάνω επηρεάζουν τα διάφορα ήδη κυκλωμάτων όπως τις μνήμες (διάφορα επίπεδα κρυφής μνήμης) και τα συνδυαστικά κυκλώματα (δεν έχουν μνήμη όπως για παράδειγμα η ALU) . Οι τεχνικές για την ανίχνευση ή/και για την διόρθωση λαθών που θα αναφέρουμε πιο κάτω χρησιμοποιούνται για την ανίχνευση και κάποιες φόρες για την διόρθωση λαθών σε μνήμες. Με αυτές τις τεχνικές μπορούμε να ανιχνεύσουμε εάν τα αποθηκευμένα δεδομένα έχουν αλλάξει από τέτοιες πηγές λαθών.

4.2 Τεχνικές για την ανίχνευση ή/και για την διόρθωση λαθών

Οι τεχνικές που θα εξετάζουμε διαχωρίζονται σε δυο ομάδες. Στην πρώτη ομάδα θα μελετήσουμε κάποιες τεχνικές οι οποίες μπορούν να ανιχνεύσουν τα λάθη αλλά δεν μπορούν να τα διορθώσουν, και στην δεύτερη ομάδα είναι οι τεχνικές οι οποίες μπορούν ταυτόχρονα να ανιχνεύσουν και να διορθώσουν σφάλματα που μπορούν να προκύψουν στα σύστημα λόγο των πηγών λαθών. Η έξτρα πληροφορία που προσθέτουμε είναι γνωστή ως κώδικες ανίχνευσης λαθών (EDC) ή κώδικες διόρθωσης λαθών (ECC)

4.2.1 Parity error detection

Η πιο απλή μορφή για την ανίχνευση λαθών είναι το Parity Bit. Το Parity Bit είναι ένα έξτρα bit το οποίο προσθέτουμε στα δεδομένα (ένα σύνολο από bits) για να εξασφαλίσει ότι ο συνολικός αριθμός των άσσων σε ένα σύνολο από bits (δεδομένα) είναι ζυγός ή περιττός. Υπάρχουν δύο παραλλαγές των parity bits, το even parity bit και το odd parity bit. Στο Even parity bit το έξτρα bit που προσθέτουμε έχει την τιμή 1 εάν ο συνολικός αριθμός των άσσων στα δεδομένα είναι περιττός και έχει την τιμή 0 εάν ο συνολικός αριθμός των άσσων στα δεδομένα είναι ζυγός. Στο odd parity bit το έξτρα parity bit που προσθέτουμε στο σύνολο των bits που αποτελούν τα δεδομένα έχει την τιμή 0 εάν ο αριθμός των άσσων είναι περιττός και έχει την τιμή 1 εάν ο συνολικός αριθμός των άσσων είναι ζυγός. Πιο κάτω δείχνουμε ένα παράδειγμα με τις τιμές που μπορεί να πάρει το parity bit ανάλογα με τα δεδομένα που ελέγχει και ανάλογα με τον τύπο του parity bit που χρησιμοποιείται για έλεγχο(even, odd)

7 bits of data (number of 1s)	Even parity	Odd parity
0000000 (0)	**0**0000000	**1**0000000
1010001 (3)	**1**1010001	**0**1010001
1101001 (4)	**0**1101001	**1**1101001

Παράδειγμα 4.1: Οι τιμές του parity bit

Η ανίχνευση λαθών χρησιμοποιώντας το parity bit έχει το πλεονέκτημα ότι χρησιμοποιεί μόνο ένα έξτρα bit για έλεγχο σε ένα σύνολο από bits και απαιτεί μόνο πύλες XOR για τον υπολογισμό του. Στο παράδειγμα 4.2 παρουσιάζεται το πως μπορούμε να υπολογίσουμε το parity bit:

Αρχικά δεδομένα: 1010001

Υπολογισμός του parity bit: P=1 ^ 0 ^ 1 ^ 0 ^ 0 ^ 0 ^ 1 = 1

Even Parity = 1

Odd Parity =0

Αρχικά δεδομένα με even parity bit = 10100011

Αρχικά δεδομένα με odd parity bit = 10100010

Παράδειγμα 4.2: Υπολογισμός του parity bit

Είναι σημαντικό να αναφέρουμε ότι χρησιμοποιώντας το parity bit ως τρόπο για ανίχνευση λαθών μπορούμε να ανιχνεύσουμε μόνο περιττό αριθμό bits που μπορούν να αλλάξουν και επίσης δεν μπορούμε να κάνουμε διόρθωση λαθών. Ο λόγος είναι ότι δεν μπορούμε να καθορίσουμε την θέση του bit που έχει αλλάξει από τα αρχικά δεδομένα. Πιο κάτω στο παράδειγμα 4.3 δείχνουμε πως μπορούμε να ανιχνεύσουμε ότι υπάρχει σφάλμα στα δεδομένα που αποστέλλονται ή που αποθηκεύονται.

Αρχικά δεδομένα = 1010001

Υπολογισμός του parity bit P=1 ^ 0 ^ 1 ^ 0 ^ 0 ^ 0 ^ 1 = 1

Τα δεδομένα μετά που έχει προκύψει κάποιο σφάλμα στην $3^{τη}$ θέση = 1000001

Υπολογισμός του parity bit στα λανθασμένα δεδομένα P=1 ^ 0 ^ 0 ^ 0 ^ 0 ^ 0 ^ 1 = 0

Παράδειγμα 4.3: Έλεγχος του parity bit για ανίχνευση λαθών

Εφόσον η τιμή του even parity bit από 1 έχει αλλάξει σε 0 τότε μπορούμε να συμπεράνουμε ότι τα δεδομένα αυτά δεν είναι σωστά

Πιο κάτω στο Παράδειγμα 4.4 δείχνουμε μια περίπτωση οπού δεν μπορούμε να ανιχνεύουμε ζυγό αριθμό σφαλμάτων χρησιμοποιώντας parity bit

.

Αρχικά δεδομένα = 1010001

Υπολογισμός του parity bit P=1 ^ 0 ^ 1 ^ 0 ^ 0 ^ 0 ^ 1 = 1

Τα δεδομένα μετά που έχει προκύψει κάποιο σφάλμα στην $3^{τη}$ και στην $7^{μη}$ θέση = 1000000

Υπολογισμός του parity bit στα λανθασμένα δεδομένα P=1 ^ 0 ^ 0 ^ 0 ^ 0 ^ 0 ^ 1 = 1

Παράδειγμα 4.4: Περίπτωση όπου δεν μπορούμε να ανιχνεύουμε ζυγό αριθμό σφαλμάτων χρησιμοποιώντας parity bit

Εφόσον η τιμή του parity bit στα λανθασμένα δεδομένα είναι η ίδια με την τιμή του parity bit πριν να συμβεί το σφάλμα θα μπορούσαμε να συμπεράνουμε ότι δεν έχει προκύψει κάποιο σφάλμα στα αρχικά δεδομένα. Όπως βλέπουμε όμως έχουν αλλάξει δυο bits από τα αρχικά δεδομένα έτσι καταλήγουμε στο συμπέρασμα ότι η ανίχνευση λαθών χρησιμοποιώντας το parity bit δουλεύει καλά εάν ο αριθμός των bits που έχει αλλάξει είναι περιττός.

4.2.2 Hamming Code (ECC)

Ο πιο κοινός τύπος για την ανίχνευση και την διόρθωση λαθών που χρησιμοποιούνται στα RAM είναι βασισμένος στους κώδικες που επινοούνται από το P.W. Hamming [5]. Στους κώδικες Hamming, προσθέτουμε στα n bits δεδομένα ένα σύνολο από k parity bits, δημιουργώντας έτσι μια νέα λέξη μεγέθους n + k bits. Οι θέσεις των bits είναι αριθμημένες ακολουθιακά από το 1 ως το n + k. Οι θέσεις οι οποίες είναι δύναμη του 2 διατηρούνται για τα parity bits και τα υπόλοιπα bits αποτελούν τα δεδομένα. Χρησιμοποιώντας το Hamming code μπορούμε να κάνουμε ανίχνευση λαθών μέχρι 2 bits και διόρθωση λαθών μόνο σε ένα bit. Επίσης το Hamming distance μας δίνει πόσα bits έχουν αλλάξει ως αποτέλεσμα κάποιου σφάλματος. Για να μπορούμε να καθορίσουμε πόσα k parity bits χρειάζονται για n data bits πρέπει να ισχύει η σχέση $n+k \leq 2^k - 1 - k$ [5]. Όπως για παράδειγμα χρησιμοποιώντας 4 parity bits μπορούμε να ανιχνεύσουμε τα λάθη σε δεδομένα μεγέθους το πολύ 11 bits. Πιο κάτω εξηγούμε με ένα παράδειγμα το πώς δουλεύει αυτή η προσέγγιση.

Έστω ότι έχουμε 8 bit δεδομένα 11000100. Για να μπορούμε να ανιχνεύσουμε την θέση στην οποία έχει προκύψει κάποιο λάθος χρειάζονται 4 parity bits.

Αυτά τα 4 parity bits μπαίνουν σε θέσεις οι οποίες είναι δύναμη του 2 όπως φαίνεται στο πιο κάτω πίνακα.

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
	P1	P2	1	P4	1	0	0	P8	0	1	0	0

Πίνακας 4.1: Οι θέσεις που τοποθετούνται τα parity bits

Στο πιο κάτω πίνακα 4.2 παρουσιάζονται ποίες θέσεις ελέγχονται από το κάθε parity bit.

Το P1 ελέγχει τις θέσεις που στην δυαδική τους μορφή έχουν 1 στο least significant bit.
Το P2 ελέγχει τις θέσεις που στην δυαδική τους μορφή έχουν 1 στο δεύτερο least significant bit.
Το P4 ελέγχει τις θέσεις που στην δυαδική τους μορφή έχουν 1 στο τρίτο least significant bit.
Το P8 ελέγχει τις θέσεις που στην δυαδική τους μορφή έχουν 1 στο δεύτερο τέταρτο significant bit.

Πίνακας 4.2: Οι θέσεις στα δεδομένα που ελέγχονται από το κάθε parity bit

Λαμβάνοντας υπόψη τις δυαδικές θέσεις που ελέγχει το κάθε parity bit υπολογίσουμε τα parity bits όπως φαίνεται στον πίνακα 4.3.

P1= XOR of bits (3, 5, 7, 9, 11) = 1^1^0^0^0=0
P2=XOR of bits (3, 6, 7, 10, 11) = 1^0^0^1^0=0
P4=XOR of bits (5, 6, 7, 12) = 1^0^0^0=1
P8=XOR of bits (9, 10, 11, 12) = 0^1^0^0=1

Πίνακας 4.3: Υπολογισμός του parity bit ανάλογα με την δυαδική θέση που ελέγχει

Ξέροντας τώρα τις τιμές των parity bits και τις θέσεις που ελέγχουν, το σύνολο των δεδομένων μαζί με τα parity bits είναι της μορφής

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
Δεδομένα + parity bits	0	0	1	1	1	0	0	1	0	1	0	0

Πίνακας 4.4: Αρχικά δεδομένα μαζί με τα parity bits

Τα δεδομένα αποθηκεύονται στην μνήμη μαζί με τα parity bits τους. Όταν διαβάζουμε από την μνήμη, τα δεδομένα ελέγχονται εάν έχει προκύψει κάποιο σφάλματα. Το parity ελέγχεται στο ίδιο σύνολο των bits που έχουμε αναφέρει πιο πάνω αλλά σε αυτή την περίπτωση συμπεριλαμβάνει και το parity bit. Το σύνολο των check bits είναι όπως φαίνεται στον πίνακα 4.5.

C1= XOR of bits (1, 3, 5, 7, 9, 11) = 0^1^1^0^0^0=0
C2=XOR of bits (2, 3, 6, 7, 10, 11) = 0^1^0^0^1^0=0
C4=XOR of bits (4, 5, 6, 7, 12) = 1^1^0^0^0=0
C8=XOR of bits (8, 9, 10, 11, 12) = 1^0^1^0^0=0

Πίνακας 4.5: Το σύνολο των check bits

Αυτός ο έλεγχος υποδεικνύει ένα even parity και όταν το C = C8C4C2C1≠0 ο δεκαδικός αριθμός του C μας δίνει την θέση στην οποία έχει προκύψει κάποιο λάθος

Πιο κάτω παρουσιάζουμε ένα παράδειγμα όπου τα δεδομένα που διαβάζουμε από την μνήμη είναι λανθασμένα

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
Αρχικά δεδομένα + parity	0	0	1	1	1	0	0	1	0	1	0	0
Λανθασμένα δεδομένα	0	0	1	1	0	0	0	1	0	1	0	0

Πίνακας 4.6: Αρχικά και λανθασμένα δεδομένα και τα ανάλογα parity bits τους

Όπως βλέπουμε πιο πάνω το bit στην 5^{τη} θέση έχει αλλάξει, για να ανιχνεύσουμε αυτό το λάθος υπολογίσουμε ξανά τα check bits όπως φαίνεται στον πίνακα 4.6 και παίρνουμε το C = C8C4C2C1=0101=5 το οποίο μας δίνει την θέση στην οποία έχει προκύψει κάποιο σφάλμα.

C1= XOR of bits (1, 3, 5, 7, 9, 11) = 0^1^0^0^0^0=1
C2=XOR of bits (2, 3, 6, 7, 10, 11) = 0^1^0^0^1^0=0
C4=XOR of bits (4, 5, 6, 7, 12) = 1^0^0^0^0=1
C8=XOR of bits (8, 9, 10, 11, 12) = 1^0^1^0^0=0

Πίνακας 4.7: Το σύνολο των check bits

Προσθέτοντας ένα έξτρα parity bit στα δεδομένα μπορούμε να ανιχνεύσουμε μέχρι 2 λάθη αλλά μπορούμε να διορθώσουμε μόνο ένα. Αυτή η παραλλαγή του Hamming Code είναι γνωστή ως SECDED.

Στο παράδειγμα μας προσθέτουμε ένα parity bit στο least significant bit των δεδομένων μαζί με τα parities τους όπως φαίνεται στον πίνακα 4.8.

Bit position	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	0	1	1	1	0	0	1	0	1	0	0	P

Πίνακας 4.8: Το σύνολο των δεδομένων μαζί μα τα parities bits τους (SECDED)

Η τιμή του P υπολογίζεται ως XOR των άλλων 12 bits . Επειδή υπολογίσουμε το even parity bit στο παράδειγμα μας το P = 1.

Έτσι το σύνολο των δεδομένων μαζί με τα parity bits είναι 0011100101001.

Όταν διαβάζουμε αυτά τα δεδομένα από την μνήμη και υπολογίζουμε τα Check bits για όλα τα Parity bits πρέπει η τιμή όλων το parity bits να είναι 0 για το λόγο ότι πιο πριν έχουμε υπολογίσει το even parity. Κάνοντας αυτούς τους ελέγχους διακρίνουμε τις πιο κάτω περιπτώσεις

Εάν το C = 0 και το P=0 τότε δεν υπάρχει σφάλμα

Εάν το C ≠ 0 και το P=1 τότε υπάρχει ένα σφάλμα το οποίο μπορεί να διορθωθεί από την στιγμή που μπορούμε να καθορίσουμε την θέση του

Εάν το C ≠ 0 και το P=0 τότε μπορούμε μόνο να ανιχνεύσουμε δυο σφάλματα.

Εάν το C = 0 και το P=1 τότε υπάρχει σφάλμα στο P13 bit

Το Hamming Code μας δίνει την δυνατότητα να ανιχνεύσουμε και ζυγό αριθμό λαθών σε αντίθεση με το parity bit για το λόγο ότι η ίδια θέση δεδομένων προστατεύεται τουλάχιστον από 2 parity bits. Επίσης εάν συμβεί σφάλμα και στα parity bits μπορούμε να το ανιχνεύσουμε επειδή για να υπολογίσουμε τα check bits συμπεριλαμβάνονται και οι θέσεις των parity bits.

Κεφάλαιο 5

Προσεγγίσεις για ανίχνευση και διόρθωση λαθών σε Κρυφές Μνήμες

5.1 Σφάλματα, ανίχνευση και διόρθωση τους στην κρυφή μνήμη

Όπως έχουμε αναφέρει και στο δεύτερο κεφάλαιο δεδομένου ότι το μέγεθος των διαφόρων συσκευών και καλωδίων συνεχίζει να μειώνεται, αυξάνεται η πιθανότητα να συμβούν μόνιμα σφάλματα στα κυκλώματα. Στους υπάρχοντες επεξεργαστές τα μόνιμα σφάλματα εμφανίζονται λόγω των ατελειών κατά την κατασκευή, λόγω των process variation, λόγω της γήρανσης αυτών των κυκλωμάτων και λόγω διαφόρων παραγόντων που έχουμε αναφέρει στο κεφάλαιο 2. Αυτά τα σφάλματα εμφανίζονται ως λειτουργικά σφάλματα στο επίπεδο της μικροαρχιτεκτονικής. Είναι δύσκολο να ανιχνεύουμε όλα τα σφάλματα κατά την κατασκευή των κυκλωμάτων. Τέτοιου είδους σφάλματα συμβαίνουν και στο επίπεδο των κρυφών μνημών. Επίσης οι κρυφές μνήμες κινδυνεύουν και από άλλες πήγες σφαλμάτων τα οποία προκαλούν soft και μόνιμα λάθη. Όπως ξέρουμε ο σκοπός ύπαρξης της κρυφής μνήμης είναι να βοηθήσει στην βελτίωση της επίδοσης των διαφόρων προγραμμάτων. Τα διάφορα είδη σφαλμάτων που εμφανίζονται σε κρυφές μνήμες οδηγούν στην αλλοίωση των δεδομένων που αποθηκεύονται σε αυτές, θέτοντας έτσι σε κίνδυνο την εγκυρότητα της πληροφορίας. Επίσης τα μόνιμα σφάλματα που συμβαίνουν θέτουν εκτός λειτουργίας διάφορα κελιά της κρυφής μνήμης, τα οποία δεν μπορούν πια να χρησιμοποιηθούν για αποθήκευση των δεδομένων. Αυτό το γεγονός οδηγεί στην μείωση του χώρου που χρησιμοποιείται για να αποθηκευτούν τα δεδομένα έχοντας επιπτώσεις στην απόδοση διαφόρων εφαρμογών. Πιο κάτω περιγράφονται με μεγαλύτερη λεπτομέρεια κάποιες από τις τεχνικές που χρησιμοποιούνται σε κρυφές μνήμες για να προστατευτούν από τα πιο πάνω είδη σφαλμάτων. Αυτές οι τεχνικές διαχωρίζονται σε δυο ομάδες. Στην πρώτη περιγράφουμε κάποιες τεχνικές που χρησιμοποιούνται για να καλύψουμε τα μόνιμα σφάλματα σε κρυφές μνήμες. Στην δεύτερη ομάδα περιγράφονται τεχνικές για την ανίχνευση και την διόρθωση των soft errors σε διαφορά επίπεδα κρυφών μνημών. Ο σκοπός τους είναι να προστατεύουν τις κρυφές μνήμες πληρώνοντας όμως κάποιο κόστος σε απόδοση, χώρο, ισχύος κτλ.

5.2 Τεχνικές προστασίας των κρυφών μνημών από διάφορα σφάλματα

5.2.1 Τεχνικές που χρησιμοποιούνται για να καλύψουν τα μόνιμα σφάλματα σε κρυφές μνήμες

Ένας τρόπος που χρησιμοποιείται για να καλύπτει τα μόνιμα σε κρυφές μνήμες είναι επιτρέποντας την κρυφή μνήμη να έχει επιπλέον χωρητικότητα. Μια προσέγγιση που χρησιμοποιείται για τα σφάλματα στην κρυφή μνήμη είναι το Graceful Degradation [6] . Αυτή η προσέγγιση χρησιμοποιείται για να καλύψει τα ελαττώματα θέτοντας εκτός λειτουργίας (διαγράφοντας) τα ελαττωματικά κομμάτια της κρυφής μνήμης. Όμως θέτοντας εκτός λειτουργίας διευθύνσεις της κρυφής μνήμης στις οποίες γίνεται συνεχής πρόσβαση έχει επιπτώσεις στην απόδοση. Τα ελαττώματα στις κρυφές μνήμες εκδηλώνονται με πολλούς και διαφορετικούς τρόπους . Μπορεί μια γραμμή στην κρυφή μνήμη να είναι ασταθής και δεν μπορεί να διατηρήσει την τιμή της, σε αυτή την περίπτωση λέμε ότι έχουμε ελαττωματική γραμμή στην κρυφή μνήμη. Υπάρχει περίπτωση μια ομάδα από cells της κρυφής μνήμης να είναι ελαττωματικά. Σε αυτή την περίπτωση λέμε ότι τα cells της κρυφής μνήμης είναι αδύνατα και μπορεί η τιμή τους για παράδειγμα να είναι πάντα 0 ή 1 και δεν μπορούν ποτέ να ενημερωθούν με την σωστή τιμή δηλαδή έχουμε ελαττωματικό set της κρυφής μνήμης. Άλλοι τρόποι με τους οποίους εκδηλώνονται τα σφάλματα στην κρυφή μνήμη είναι να έχουμε ελαττωματικό way όπως για παράδειγμα σε ένα 4 way set associative κρυφή μνήμη υπάρχει περίπτωση να μην μπορούμε ποτέ να ενημερώσουμε ή να διαβάζουμε την τιμή του 4 block για το λόγο ότι αυτή η θέση μνήμης στο set είναι ελαττωματική.

Υπάρχει περίπτωση να έχουμε ελαττωματικές πόρτες μέσω των οποίων έχουμε πρόσβαση στην κρυφή μνήμη, το οποίο έχει επιπτώσεις στο γεγονός ότι δεν θα μπορούμε να έχουμε πρόσβαση μέσω αυτής της πόρτας στην κρυφή μνήμη. Εάν όλα τα sets που αποτελούν την κρυφή μνήμη είναι ελαττωματικά τότε λέμε ότι έχουμε ελαττωματική κρυφή μνήμη. Διάφορες στρατηγικές graceful υποβάθμισης μπορούν να υπερνικήσουν την επίδραση των διάφορων εκδηλώσεων ελαττωμάτων στη κρυφή μνήμη. Οι τεχνικές αυτές αναφέρονται πιο κάτω

- Line Delete. Όταν μια συγκεκριμένη γραμμή της κρυφής μνήμης είναι ελαττωματική τότε μπορεί να χαρακτηριστεί και να αποκλειστεί από τις γραμμές

της κρυφής μνήμης οι οποίες είναι λειτουργήσιμες. Για να είναι γνωστές οι ελαττωματικές γραμμές δημιουργείται ένας χάρτης σφαλμάτων ο οποίος προγραμματίζεται έτσι ώστε να καταγράψει τις γραμμές της κρυφής μνήμης οι οποίες έχουν σημαδευτεί για το λόγο ότι είναι ελαττωματικά. Η υλοποίηση του χάρτη σφαλμάτων γίνεται με διάφορους τρόπους, ένας απλός τρόπος που χρησιμοποιείται είναι να προσθέσουμε ακόμα ένα valid bit στην κρυφή μνήμη για να μας δείξει εάν η γραμμή στην οποία θέλουμε να έχουμε πρόσβαση δεν είναι ελαττωματική.

- Set delete. Όταν ένα set της κρυφής μνήμης γίνεται ελαττωματικό τότε μπορεί να σημαδευτεί ως ένα ελαττωματικό set. Σε αυτή την περίπτωση είναι ανεπαρκής να θέσεις εκτός λειτουργίας ολόκληρο το set για το λόγο ότι αφήνει μια "τρύπα" στο χώρο διευθύνσεων της κρυφής μνήμης, το οποίο οδηγεί σε μια σημαντική υποβάθμιση της απόδοσης. Για να μην έχουμε μεγάλες επιπτώσεις στην απόδοση χρησιμοποιείται ένα σχέδιο remapping το οποίο μπορεί να κατευθύνει τις διευθύνσεις από ένα ελαττωματικό set σε ένα λειτουργήσιμο.
- Way delete. Σε περίπτωση που τουλάχιστον ένα way είναι μη διαθέσιμο λόγο σφαλμάτων τότε μπορεί να κλίσει το way της κρυφής μνήμης. Σε ένα n-way set associative κρυφή μνήμη ένα n bit χάρτης σφαλμάτων μπορεί να μας καθοδηγήσει στο να αναγνωρίσουμε ποια ways είναι μη διαθέσιμα. Εάν για κάθε way υπάρχει ένα bit το οποίο να παρουσιάζει την διαθεσιμότητα τότε για να κλίσουμε αυτό το way σε όλα τα set μπορούμε να αλλάξουμε την τιμή αυτού του bit σε όλα τα sets της κρυφής μνήμης.
- Cache resizing. Σε περίπτωση που έχουμε πολλά σφάλματα τα οποία είναι μαζί, συνεχόμενα, τότε λέμε ότι αυτά τα σφάλματα δημιουργούν ένα cluster. Για να τα σημαδέψουμε αυτά όλα μαζί μπορεί να μετατρέψουμε ένα προγραμματιζόμενο αποκωδικοποιητή για να αποκλείσει γραμμές ή set ανάλογα με την υλοποίηση και να χρησιμοποιήσει μονό τις διαθέσιμες πηγές.
- Port delete. Σε αυτή την περίπτωση χρειάζεται να αλλάξουμε την εξωτερική εικόνα μιας κρυφής μνήμης. Όταν σβήνουμε μια πόρτα της κρυφής μνήμης τότε αυτό μπορεί να επηρεάζει την ροή των εντολών στο pipeline για το λόγο ότι δεν

θα πρέπει αυτές οι εντολές να έχουν πρόσβαση στην κρυφή μνήμη μέσω της ελαττωματικής πόρτας.

- Row remapping. Όταν οι γραμμές της κύριας μνήμης και οι γραμμές της κρυφής μνήμης χάνονται τότε οι προσβάσεις σε ελαττωματικές γραμμές μπορεί να καθοδηγηθούν σε άλλες γραμμές που δεν είναι ελαττωματικές .

5.2.2 Τεχνικές που χρησιμοποιούνται για να καλύψουν τα soft λάθη σε κρυφές μνήμες

Διάφορες προσεγγίσεις ακολουθούντα επίσης για την ανίχνευση και την διόρθωση των soft λαθών σε κρυφές μνήμες [7]. Πιο κάτω θα αναφερθούμε σε κάποιες από αυτές τις τεχνικές έχοντας υπόψη ότι η ανίχνευση και διόρθωση λαθών σε κρυφές μνήμες έχει κάποιο overhead στην κατανάλωση ισχύος, στην απόδοση, στο χώρο, και στο bandwidth της κρυφής μνήμης.

Η μια τεχνική που χρησιμοποιείται για την ανίχνευση και διόρθωση λαθών σε L1 κρυφή μνήμη δεδομένων (L1D) χρησιμοποιεί EDC κώδικα για την ανίχνευση λαθών στο πρώτο επίπεδο κρυφής μνήμης L1, το οποίο στις περισσότερες περιπτώσεις είναι απλό parity κώδικα. Σε περίπτωση που κάποιο λάθος ανιχνεύετε στην L1 τότε τα δεδομένα ανακτούνται από το δεύτερο επίπεδο της κρυφής μνήμης L2, στην οποία κάθε μπλοκ δεδομένων προστατευτεί με ECC κώδικα για την διόρθωση λαθών. Αυτή η τεχνική είναι γνωστή ως EDC/ECC και απαιτεί write through L1 κρυφή μνήμη η οποία αυξάνει σημαντικά το bandwidth που απαιτείτε για τις προσβάσεις στην L2 κρυφή μνήμη. Λαμβάνοντας υπόψη ότι το L1 είναι write through τότε κάθε ενημέρωση στα δεδομένα της L1 ακολουθείται με ενημέρωση στα δεδομένα της L2, επίσης σε περίπτωση που ανιχνεύεται κάποιο σφάλμα στα δεδομένα της L1 τότε τα δεδομένα ανακτώνται από την L2 . Αυτό το γεγονός αυξάνει την κατανάλωση ισχύος στην κρυφή μνήμη L2. Από την άλλη έχοντας μόνο κώδικες για ανίχνευση λαθών στην L1 έχει ως πλεονέκτημα να έχουμε μικρότερη και γρηγορότερη κρυφή μνήμη L1.

Μια άλλη προσέγγιση είναι να έχουμε κώδικες διόρθωσης λαθών και στην L1 και στην L2 [7]. Αυτή η προσέγγιση είναι γνωστή ως ECC/ECC σχήμα προστασίας της κρυφής μνήμης. Συνήθως ο κώδικας που χρησιμοποιείται μπορεί να ανιχνεύσει μέχρι και 2 σφάλματα και να διορθώσει ένα σφάλμα. Αυτή η προσέγγιση απαιτεί write back κρυφή

μνήμη L1 όπου τα δεδομένα στην κρυφή μνήμη L2 και στα πιο κάτω επίπεδα μνήμης ενημερώνονται μονό όταν το block των δεδομένων στην L1 αντικαθίσταται από κάποιον άλλο block. Όμως έχοντας τους κώδικες ECC στο πρώτο επίπεδο της κρυφής μνήμης, L1, αυξάνει το μέγεθος του κάθε block δεδομένων υποβιβάζοντας τον χρόνο πρόσβασης στην L1. Επίσης αυξάνεται το latency στην L1 το οποίο οφείλεται στην καθυστέρηση που χρειάζεται για τον υπολογισμό του ECC κώδικα. Σε περίπτωση που ο κώδικας ECC χρησιμοποιείται για την διόρθωση λαθών σε block granularity τότε εάν σε ένα write στην κρυφή μνήμη L1 χρειάζεται να αντικατασταθεί μονό ένα word από το block απαιτεί Read Modify Write στην L1 το όποιο είναι πιο χρονοβόρο από το γράψιμο. Το πιο πάνω πρόβλημα μπορούσε να λυθεί έχοντας ECC κώδικες σε word granularity αλλά και αυτό έχει το μειονέκτημα ότι χρειάζεται περισσότερα bits για την ανίχνευση και την διόρθωση λαθών ανά μπλοκ το όποιο οδηγεί σε μεγαλύτερη και πιο αργή κρυφή μνήμη L1 και μεγαλύτερη κατανάλωση ισχύος στην L1. Διάφορες παραλλαγές των πιο πάνω προσεγγίσεων έχουν μελετηθεί έτσι ώστε να λυθούν κάποια από τα προβλήματα που υπάρχουν.

Μια παραλλαγή των πιο πάνω προσεγγίσεων (Punctured ECC Recovery Code, PERC) βασίζεται σε μια επιπρόσθετη δομή on chip η οποία αποθηκεύει τους κώδικες για την διόρθωση λαθών. Η δομή αυτή εκτός από τους κώδικές διόρθωσης λαθών (ECCp) για τα δεδομένα και τα tags, έχει και ένα bit το οποίο δείχνει το μπλοκ στο πρώτο επίπεδο της κρυφής μνήμης, L1, στο οποίο αντιστοιχεί ο κώδικας. Με αυτό τον τρόπο στην L1 αποθηκεύονται μόνο οι κώδικες για την ανίχνευση λαθών (EDC) και οι κώδικες για την διόρθωση λαθών αποθηκεύονται σε μια δομή λεγομένη PERC [7] . Ο κώδικας ελέγχου χωρίζεται σε rd bits για ανίχνευση και rp bits για διόρθωση σφαλμάτων. Με αυτό τον τρόπο μπορούν να ανιχνεύσουν μέχρι d σφάλματα και να διορθώσουν μέχρι p σφάλματα. Επίσης αυτός ο μηχανισμός έχει το πλεονέκτημα ότι μπορεί να χρησιμοποιηθεί καλύτερο ECC κώδικας. Επίσης για το λόγο ότι οι ECC κώδικες αποθηκεύονται σε άλλη δομή δεν αυξάνεται το latency στο L1D (L1 data cache). Επίσης αυτή η παραλλαγή δεν αυξάνει το bandwidth στο δεύτερο επίπεδο της κρυφής μνήμης, L2, για το λόγο ότι εάν ανιχνεύεται κάποιο σφάλμα στην LlD τότε χρησιμοποιούνται οι ECCp κώδικες που είναι στην PERC. Όμως προσθέτει έξτρα πίεση στο data bus που φέρνει τα αποθηκευμένα δεδομένα στην L1D. Αποσυνδέοντας τα EDC/ECC bits μειώνει

την κατανάλωση ενέργειας για το λόγο ότι σε μια πράξη ανάγνωσης έχουμε πρόσβαση μονό στα bits των δεδομένων και στα bits που χρησιμοποιούνται για τον κώδικα EDC και δεν χρειάζεται να υπολογίσουμε το ECC για κάθε ανάγνωση. Όμως κατά την εγράφη των δεδομένων σε κρυφή μνήμη χρειάζεται να υπολογίσουμε και τους ECC κώδικες.

Μια άλλη προσέγγιση για την ανίχνευση και την διόρθωση των soft errors σε κρυφές μνήμες τελευταίου επιπέδου (LLC) είναι η Memory Mapped ECC [8]. Και σε αυτή την περίπτωση ο κώδικας για την ανίχνευση λαθών είναι αποσυνδεδεμένος από τον κώδικα για την διόρθωση λαθών. Η διάφορα αυτής της προσέγγισης με τις προηγούμενες είναι ότι ο κώδικας για την διόρθωση λαθών αποθηκεύεται σαν προσπελάσιμα δεδομένα στο DRAM(off-chip) αντί να αποθηκευτούν στην κρυφή μνήμη. Για την ανίχνευση και την διόρθωση λαθών, ο κώδικας που εφαρμόζεται και είναι αποθηκευμένος στην LLC είναι απλός. Αυτός μπορεί να ανιχνεύει σφάλματα και να διορθώνει μικρό αριθμό σφαλμάτων. Σε περίπτωση που χρειάζεται περεταίρω διόρθωση για τα σφάλματα χρησιμοποιείται ο κώδικας ο όποιος είναι αποθηκευμένος στο DRAM για το λόγο ότι έχει περισσότερες ικανότητες διόρθωσης περισσότερων σφαλμάτων. Ο σκοπός αυτής της προσέγγισης είναι να μειώσει το on-chip ECC overhead γράφοντας μονό τις “μολυσμένες” γραμμές της κρυφής μνήμης (dirty cache lines) στο DRAM. Έτσι κρατούμε στο DRAM το πιο σύνθετο ECC κώδικα μονό για τα δεδομένα που είναι αποθηκευμένα σε αυτές τις γραμμές και όχι για όλα τα δεδομένα. Αυτή η προσέγγιση χρησιμοποίει two-tiered μηχανισμό για να εξασφαλίζει την ακεραιότητα δεδομένων . Το tier1 error code (T1EC) είναι ένας κώδικας ο όποιος μπορεί να αποκωδικοποιείται σε χαμηλό latency και σε χαμηλή ενέργεια. Η πολυπλοκότητα υπολογισμού του κώδικα αυτού δεν είναι μεγάλη. Το tier2 error code (T2EC) το οποίο είναι αποθηκευμένο στο DRAM υπολογίζεται μονό σε dirty write- back στο LLC και διαβάζεται μονό σε περιπτώσεις όπου το T1EC δεν μπορεί να διορθώσει τα σφάλματα.

Για την αποθήκευση της μονάδας T2ER απαιτείται να είναι διαθέσιμη μια περιοχή από το memory name space στο DRAM . Η δομή με τους ECC κώδικες γίνεται mapped σε φυσικές διευθύνσεις DRAM και είναι cacheable στο τελευταίο επίπεδο της κρυφής μνήμης (LLC). Με αυτό τον τρόπο η μονάδα T2ER αποθηκεύεται ως δεδομένα στο LLC. Έτσι σε περίπτωση ανίχνευσης κάποιον σφαλμάτων σε δεδομένα που είναι αποθηκευμένα στο LLC υπάρχει μεγάλη πιθανότητα ο κώδικας για την διόρθωση λαθών

να βρίσκεται στο LLC μειώνοντας έτσι το χρόνο που χρειάζεται για τις προσβάσεις στο DRAM. Σε περίπτωση που ο κώδικας δεν βρίσκεται στην κρυφή μνήμη μπορεί να διαβαστεί από το DRAM με το ίδιο τρόπο όπως τα δεδομένα και να γραφτεί στις διευθύνσεις που αντιστοιχούν στο LLC. Με αυτό το τρόπο το LLC διαχωρίζεται δυναμικά σε δεδομένα και σε T2EC που αντιστοιχεί σε αυτά. Κάποια πλεονεκτήματα αυτής της προσέγγισης είναι ότι δεν χρειάζεται αφιερωμένη μονάδα αποθήκευσης on chip για τους κώδικες ECC μειώνοντας έτσι την διαρροή και το overhead σε χώρο. Επίσης δίνει την δυνατότητα οι κώδικες ECC που αποθηκεύονται στο DRAM να προφέρουν καλύτερο διόρθωση λαθών. Ένα μειονέκτημα αυτής της προσέγγισης είναι ότι αποθηκεύοντας τα ECC σε διευθύνσεις της κρυφής μνήμης ως δεδομένα μειώνει τον αριθμό των γραμμών στην κρυφή μνήμη που μπορούν να αποθηκευτούν πραγματικά δεδομένα δηλαδή μειώνεται η χωρητικότητα του LLC για τα πραγματικά δεδομένα.

Κεφάλαιο 6

Προτεινόμενο λογικό σχήμα της κρυφής μνήμης

6.1 Το κίνητρο για αυτή την μελέτη

Όπως αναφέρθηκε και στο πρώτο εισαγωγικό κεφάλαιο η ανάγκη για προστασία των κρυφών μνημών από τα σφάλματα οδηγεί σε επιπρόσθετα bits τα οποία αποθηκεύονται σε αυτά. Η προσθήκη αυτής της έξτρα πληροφορίας για ανίχνευση και διόρθωση λαθών οδηγεί σε overhead σε χώρο και σε ενέργεια. Διάφορες έρευνες που έχουν γίνει για την αξιολόγηση του overhead σε χώρο έχουν συμπεράνει ότι περίπου 12.5% του χώρου στην κρυφή μνήμη χρησιμοποιείται για την αποθήκευση των bits προστασίας [7]. Επίσης ο υπολογισμός και η αποκωδικοποίηση τους, αυξάνει την συνολική κατανάλωση ενέργειας και το latency στην κρυφή μνήμη. Από την άλλη για να προστατεύουμε τα δεδομένα που αποθηκεύονται στην κρυφή μνήμη είναι απαραίτητο να προσθέσουμε κώδικες για την ανίχνευση και διόρθωση λαθών. Το επίπεδο προστασίας που παρέχουμε στα δεδομένα εξαρτάται από τον αριθμό των bits δεδομένων που προστατεύονται από ένα parity bit. Αυτό το στοιχείο είναι γνωστό ως granularity στο οποίο ανιχνεύουμε τα σφάλματα.

Γενικότερα ισχύει ότι, όσο πιο μικρό είναι το granularity στο οποίο ανιχνεύουμε τα σφάλματα, τόσο περισσότερο αυξάνεται ο αριθμός των bits προστασίας. Το μέγεθος του parity κώδικα αυξάνεται ανάλογα με το επίπεδο προστασίας που θέλουμε να παρέχουμε, αυξάνοντας με αυτό τον τρόπο το overhead σε χώρο. Είναι σημαντικό να αναφέρουμε ότι για να υπολογίσουμε τους parity κώδικες χρησιμοποιούμε XOR λογικές πύλες, οι οποίες είναι απλές πράξεις που καταναλώνουν ελάχιστη ενέργεια. Αλλά και πάλι μεγαλύτερο parity κώδικα οδηγεί σε μεγαλύτερη κατανάλωση ενέργειας.

Για σκοπούς κατανόησης πιο κάτω παρουσιάζεται ένα παράδειγμα όπου πρώτα προστατεύουμε τα δεδομένα μιας εντολής χρησιμοποιώντας 4 bits even parity κώδικα σε granularity ενός byte (κάθε parity bit προστατεύει 8 bit δεδομένα που βρίσκονται στο ίδιο byte). Έπειτα προστατεύουμε τα ίδια δεδομένα χρησιμοποιώντας 1 bit even parity κώδικα σε granularity 32 bits. Η μεθοδολογία για τον υπολογισμό του even parity κώδικα επεξηγείται με λεπτομέρεια στο τέταρτο κεφάλαιο. Υποθέτουμε ότι λόγω κάποιου σφάλματος έχουν αλλάξει 2 bits στην εντολή.

Παρατηρώντας τις τιμές των parity bits πριν (Σχήμα 6.1.1) και μετά τα σφάλματα (Σχήμα 6.1.2) βλέπουμε ότι μετά την αλλαγή των δυο bits στην εντολή λόγο σφαλμάτων, χρησιμοποιώντας μόνο 1 bit parity κώδικα δεν μπορούσαμε να ανιχνεύσουμε τα σφάλματα για το λόγο ότι η τιμή του Parity bit είναι η ίδια με πριν. Χρησιμοποιώντας 4 bit parity κώδικα σε byte granularity τα σφάλματα μπορούσαμε να τα ανιχνεύουμε για το λόγο ότι έχουν αλλάξει οι τιμές των δυο parity bits που προστατεύουν αυτά τα bytes.

Εντολή (32 bits)	0 1 0 0 0 0 0 0	1 1 1 1 0 0 0 0	1 0 1 0 0 0 0 0	0 1 0 0 1 1 0 0
Even Parity (4 bits)	1	0	0	1
Even Parity (1 bit)	0			

Σχήμα 6.1.1: Τα αρχικά δεδομένα και τα parity bits πριν να προκύψει κάποιο σφάλμα

Εντολή (32 bits)	0 1 0 0 0 0 0 **1**	1 1 1 1 0 0 0 0	1 0 1 0 0 **1** 0 0	0 1 0 0 1 1 0 0
Even Parity (4 bits)	**0**	0	**1**	1
Even Parity (1 bit)	0			

Σχήμα 6.1.2: Τα λανθασμένα δεδομένα και τα parity bits μετά που έχει προκύψει κάποιο σφάλμα

Είναι σημαντικό να αναφέρουμε ότι, επειδή τα σφάλματα συνέβηκαν σε διαφορετικά bytes της εντολής (1 byte = 8 bits) όπου το κάθε byte προστατεύεται από διαφορετικό parity κώδικα, είναι εφικτό να τα ανιχνεύουμε. Διαφορετικά εάν τα σφάλματα συνέβαιναν στο ίδιο byte της εντολής δεν θα μπορούσαμε να τα ανιχνεύουμε με 1 bit parity που προστατεύει αυτό το byte. Για ανίχνευση σφαλμάτων στο ίδιο byte θα χρειαζόταν το granularity προστασίας από ένα parity bit να είναι μικρότερο του ενός byte. Αυτό το γεγονός έχει να κάνει με την ιδιότητα ότι 1 bit parity μπορεί να ανιχνεύει μόνο περιττό αριθμό σφαλμάτων στα δεδομένα που προστατεύει.

6.2 Χαρακτηριστικά των RISC ISA

Σε αυτή την μελέτη το σύνολο των δεδομένων που εξετάσαμε είναι εντολές RISC μεγέθους 32 bits. Ένα κίνητρο που αποφασίσαμε να μελετήσουμε τις κρυφές μνήμες εντολών είναι η κανονική μορφή των εντολών RISC και κάποιες ιδιότητές τους. Πιο

κάτω παρουσιάζονται οι μορφές των εντολών για την Mips [9] (Σχήμα 6.2.1) και την Alpha [10] (Σχήμα 6.2.2) αρχιτεκτονική οπού το μέγεθος των εντολών είναι 32 bits.

Bit position	[31..26]	[25..21]	[20..16]	[15..11]	[10..6]	[5..0]
R-type	Opcode	Rs(1st source register)	Rt(2st source register)	Rd (dest register)	sa	funct
I-type	Opcode	Rs (source register)	Rt (dest register)	Immediate		
J-type	Opcode	Address				

Σχήμα 6.2.1: Μορφή εντολών για Mips Αρχιτεκτονική

Bit position	[31..26]	[25..21]	[20..16]	[15..5]	[4..0]
Operate-type	Opcode	Rs(1st source register)	Rs(2nd source register)	Funct	Rd(dest register)
Memory-type	Opcode	Rd(dest register)	Rs(source register)	Immediate	
Branch, Jump/Call tyepe	Opcode	Rs(source register)	Constant		

Σχήμα 6.2.2: Μορφή εντολών για Alpha Αρχιτεκτονική

Παρατηρώντας τα πιο πάνω σχήματα βλέπουμε ότι οι εντολές έχουν μια συγκεκριμένη μορφή και είναι ομαδοποιημένες σε τρεις κατηγορίες και στις δυο αρχιτεκτονικές. Υπάρχουν κάποιες διαφορές μεταξύ των δυο αρχιτεκτονικών στην ομαδοποίηση των εντολών και στις θέσεις μέσα στην εντολή που χρησιμοποιείται για να αποθηκεύουμε τους καταχωρητές. Σημαντικό είναι να παρατηρήσουμε ότι το μέγεθος των καταχωρητών και το μέγεθος του immediate πεδίου σε bits, είναι το ίδιο και στις δυο αρχιτεκτονικές. Κάποιες ιδιότητες που μας οδήγησαν να μελετήσουμε την προστασία των δεδομένων σε κρυφές μνήμες εντολών είναι το γεγονός ότι το πεδίο που αποθηκεύει μια άμεση τιμή είναι αρκετό μεγάλο και σπάνια έχουμε τόσες μεγάλες άμεσες τιμές που να χρησιμοποιήσουν και τα 16 bits αυτού του πεδίου. Πολύ συχνά μια άμεση τιμή χωρεί στα πρώτα 8 bits αυτού του πεδίου και το ανώτερο byte του είναι 0. Μια άλλη σημαντική

παρατήρηση είναι ότι πολύ συχνά στις εντολές ο καταχωρητής πηγής είναι ο ίδιος με τον καταχωρητή προορισμού.

Ο σκοπός μας σε αυτή την μελέτη είναι να έχουμε την ίδια προστασία των δεδομένων σε κρυφές μνήμες εντολών με λιγότερα parity bits έτσι ώστε να μειώσουμε το overhead σε χώρο και ενέργεια. Είναι σημαντικό να αναφέρουμε ότι παρόμοια ανάλυση μπορεί να γίνει και στην κρυφή μνήμη δεδομένων. Οι ιδιότητες των εντολών που έχουμε αναφέρει πιο πάνω βοηθάνε σε αυτή την ανάλυση. Εάν το ανώτερο byte του immediate πεδίου είναι πάντα 0 τότε το parity bit που αντιστοιχεί σε αυτό το byte θα είναι πάντα 0 δηλαδή αυτό το parity bit δεν θα χρειαζόταν για το λόγο ότι η τιμή του είναι γνωστή. Στην μελέτη αυτή δεν εξετάζαμε περαιτέρω πόσο συχνά ισχύουν οι πιο πάνω ιδιότητες αλλά όπως έχω αναφέρει και στο πρώτο εισαγωγικό κεφάλαιο ήταν ένα κίνητρο για να επικεντρωθούμε στην κρυφή μνήμη εντολών.

6.3 Το λογικό σχήμα της Κρυφής Μνήμης με έξτρα bits προστασίας

Πιο κάτω επεξηγούμε σχηματικά πως αλλάζει η λογική δομή της κρυφής μνήμης εντολών όταν προσθέτουμε σε αυτή parity κώδικες για προστασία των δεδομένων. Για σκοπούς κατανόησης χρησιμοποιούμε το πιο κάτω παράδειγμα:

Έστω ότι έχουμε μια direct map κρυφή μνήμη εντολών όπου το κάθε block της αποθηκεύει 16 εντολές μεγέθους 32 bits. Το μέγεθος του μπλοκ είναι 16 * 32 = 512 bits. Αν υποθέσουμε ότι για κάθε εντολή στην κρυφή μνήμη χρησιμοποιούμε 4 bit parity κώδικα (1 bit parity σε byte granularity) τότε ο συνολικός αριθμός των parity bits που χρειάζεται να προστατέψουμε ένα μπλοκ εντολών είναι 16 * 4 = 64 bits. Από αυτό το παράδειγμα βλέπουμε ότι το overhead σε χώρο που πληρώνουμε για τους έξτρα parity κώδικες ανά μπλοκ είναι 64/512 = 0.125 το οποίο είναι 12.5 %.

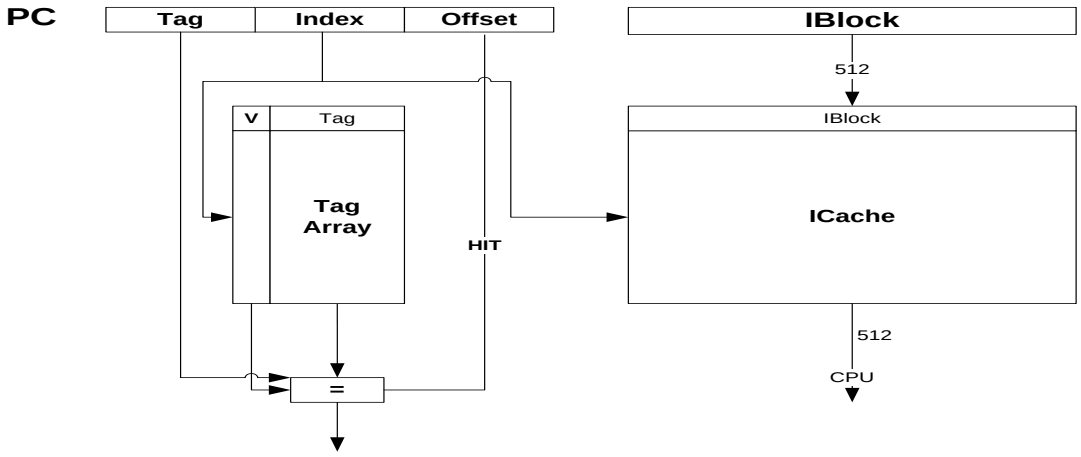
Στο Πίνακα 6.3 παρουσιάζουμε συνοπτικά τους παραμέτρους της baseline direct map κρυφής μνήμης εντολών.

Περιγραφή Παραμέτρων	**Το μέγεθος τους σε bits**
Parity κώδικα ανά εντολή	4 bits
Εντολές	32 bits

Block	16 εντολές * 32 bits = 512 bits
Parity κώδικα ανά block	4 * 16 = 64 bits

Πίνακας 6.3: Παράμετροι της direct map ICache

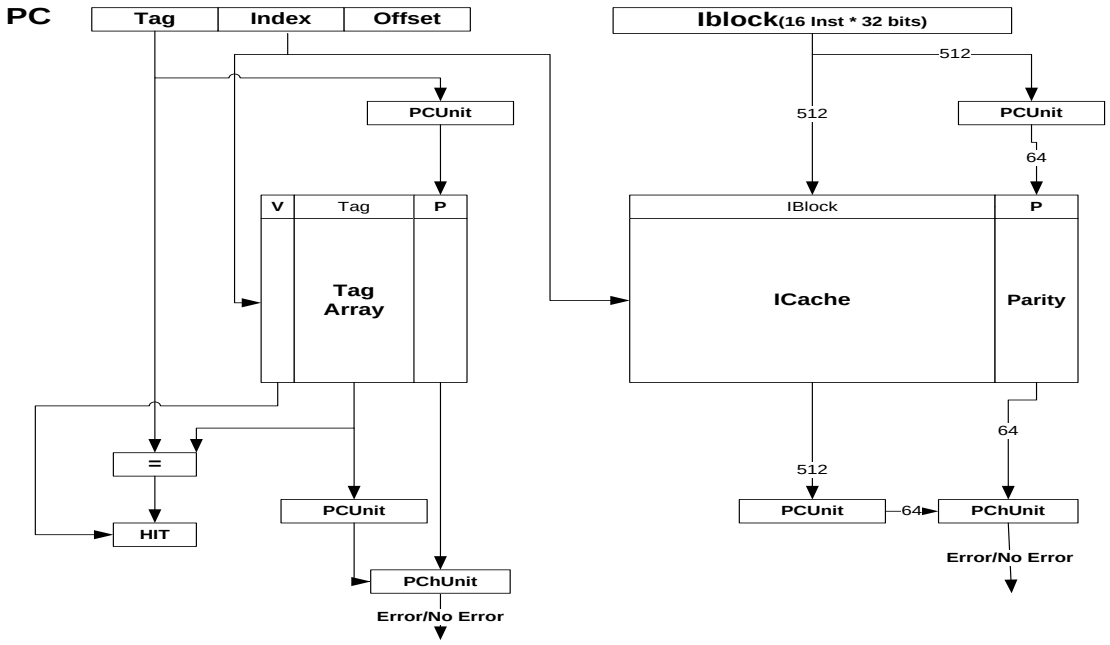
Στο σχήμα 6.3.1 βλέπουμε πως είναι η απλοποιημένη λογική δομή μιας κρυφής μνήμης direct map με τις πιο πάνω διαμορφώσεις χωρίς τα bits προστασίας. Το program counter (PC) είναι ένας καταχώρησης ο οποίος κρατά τη διεύθυνση της θέσης μνήμης της επόμενης εντολής που πρέπει να εκτελεστεί, όταν εκτελείται η τρέχουσα εντολή από τον επεξεργαστή. Αρχικά γίνεται έλεγχος αν η εντολή βρίσκεται στην κρυφή μνήμη. Πρώτα με το index επιλέγουμε το set στο οποίο βρίσκεται το block που περιέχει την εντολή. Στο Tag Array αποθηκεύουμε το tag του κάθε block εντολών που βρίσκονται στην κρυφή μνήμη. Σε περίπτωση που το tag του PC αντιστοιχεί με το tag το οποίο είναι αποθηκευμένο στο tag array και τα δεδομένα είναι έγκυρα (valid bit δείχνει την εγκυρότητα τους) τότε έχουμε Hit και μπορούμε να διαβάζουμε την εντολή από την κρυφή μνήμη, η οποία μπαίνει για εκτελέσει στον επεξεργαστή. Περισσότερες λεπτομερές το πως βρίσκουμε τα δεδομένα που θέλουμε να εκτελεστούν παρουσιάζονται στο τέταρτο κεφάλαιο.



Σχήμα 6.3.1: baseline Direct map κρυφή μνήμη εντολών(ICache)

Στο Σχήμα 6.3.2 δείχνουμε πως αλλάζει η λογική δομή της baseline κρυφής μνήμης όταν προσθέτουμε έξτρα parity κώδικες για προστασία δεδομένων στο Tag Array και

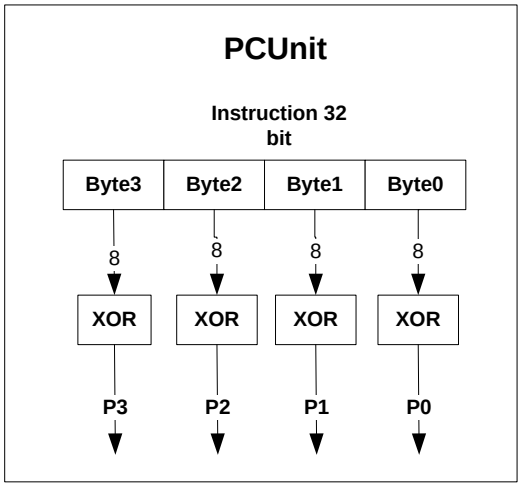
στην κρυφή μνήμη εντολών. Εμείς θα επικεντρωθούμε στους parity κώδικες για προστασία των δεδομένων στην κρυφή μνήμη αλλά είναι σημαντικό να αναφέρουμε ότι προστασία χρειάζεται και το tag στο Tag Array. Αρχικά πριν να αποθηκεύσουμε τις εντολές ενός block στην κρυφή μνήμη υπάρχει μια έξτρα δομή (PCUnit) η οποία υπολογίζει τους κώδικες parity. Από το πιο κάτω σχήμα βλέπουμε ότι μαζί με το block των εντολών αποθηκεύουμε στην κρυφή μνήμη και τους αντιστοίχους parity κώδικες. Όταν διαβάζουμε ένα μπλοκ εντολών από την κρυφή μνήμη επαναϋπολογίσουμε τους parity κώδικες τους, και συγκρίνουμε αν αυτά τα parity bits είναι τα ίδια με τα parity bits του block στην κρυφή μνήμη. Αν αυτά τα bits είναι τα ίδια τότε ξέρουμε ότι δεν έχει προκύψει κάποιο σφάλμα στα δεδομένα, διαφορετικά υπάρχει σφάλμα στα δεδομένα που διαβάζουμε από την κρυφή μνήμη.



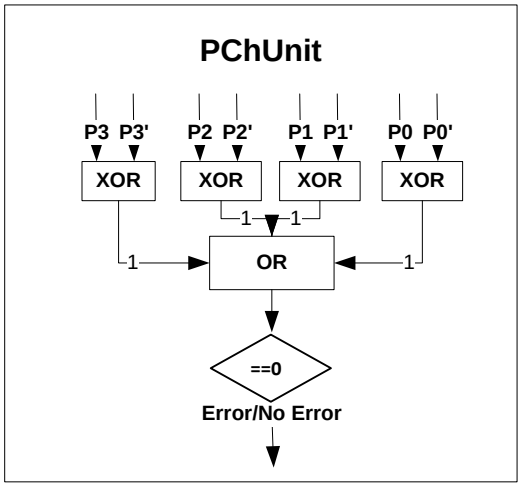
Σχήμα 6.3.2: baseline Direct map ICache με έξτρα bit προστασίας των δεδομένων

Στο Σχήμα 6.3.3.a και 6.3.3.b παρουσιάζονται οι μονάδες που χρησιμοποιούνται για να υπολογίσουμε και να ελέγχουμε τους κώδικες parity. Το PCUnit υπολογίζει για κάθε byte της εντολής (1 byte = 8 bits) ένα parity bit, χρησιμοποιώντας XOR λογικές πύλες. Στο τέλος για κάθε εντολή θα έχουμε 4 bit κώδικα parity (περισσότερη λεπτομέρεια το πως υπολογίσουμε τους parity κώδικες παρουσιάζονται στο τέταρτο κεφάλαιο). Το PChUnit ελέγχει τους τα parity bits που είναι αποθηκευμένα στην κρυφή μνήμη εντολών

με τα parity bits που επαναϋπολογίσουμε πριν μια εντολή που διαβάζουμε από την κρυφή μνήμη θα πρέπει να σταλεί για εκτέλεση. Η έξοδος από την κάθε πύλη XOR είναι 0 εάν οι τιμές εισόδου είναι οι ιδέες δηλαδή εάν τα parity bits που έχουμε επαναϋπολογίσει είναι τα ίδια με αυτά που είναι αποθηκευμένα στην κρυφή μνήμη. Χρησιμοποιώντας OR λογικές πύλες ελέγχουμε αν αυτές η τιμές είναι 0 το οποίο σημαίνει ότι είναι οι ίδιες, διαφορετικά εάν η έξοδος από την OR είναι 1 σημαίνει ότι έχει προκύψει κάποιο σφάλμα .



Σχήμα 6.3.3.a: PCUnit μονάδα για υπολογισμό του parity κώδικα ανά εντολή



Σχήμα 6.3.3.b: PChUnit μονάδα για έλεγχο του parity κώδικα ανά εντολή

6.4 Το προτεινόμενο λογικό σχήμα της κρυφής μνήμης εντολών

Όπως έχουμε αναφέρει ο σκοπός σε αυτή την μελέτη είναι να μειώσουμε το χώρο που χρειάζεται στην κρυφή μνήμη εντολών για την αποθήκευση των bits προστασίας χωρίς να μειώσουμε το βαθμό προστασίας που παρέχεται σε αυτήν.

Στο σχήμα 6.4 παρουσιάζεται η λογική μορφή της κρυφής μνήμης εντολών βασισμένη στην προσέγγιση μας, οπού η κρυφή μνήμη εντολών έχει τα ίδια χαρακτηριστικά (τις ίδιες διαμορφώσεις) με την baseline κρυφή μνήμη εντολών (σχήμα 6.3.2). Η διαφορά του λογικού σχήματος που προτείνουμε εμείς, από το baseline σχήμα της κρυφής μνήμης εντολών με έξτρα bit προστασίας (σχήμα 6.3.2), είναι ότι, πριν να αποθηκευτεί ένα

block εντολών στην κρυφή μνήμη θα πρέπει να γίνει κάποιο μετασχηματισμό στα δεδομένα των εντολών.

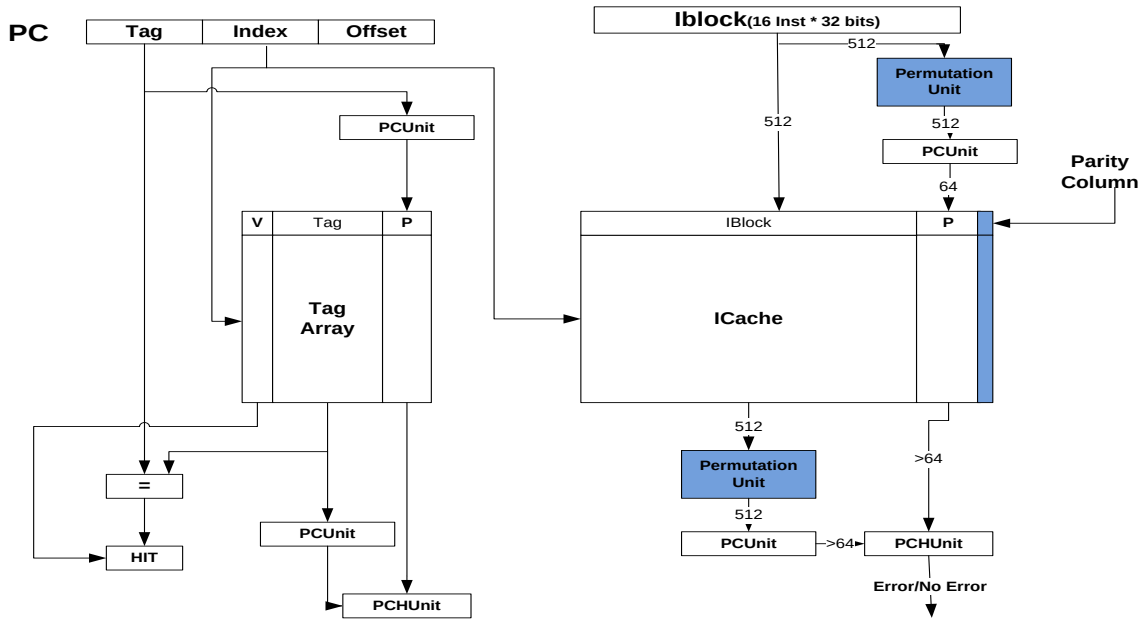
Ο μετασχηματισμός γίνεται σε επίπεδο της εντολής αλλάζοντας τις θέσεις των bits μέσα στην ίδια την εντολή (Permutation Unit).

Παρατηρώντας το σχήμα 6.4 βλέπουμε ότι το block των εντολών αποθηκευτεί στην κρυφή μνήμη ως είναι, χωρίς καμία αλλαγή στις εντολές. Τα parity bits που αποθηκεύονται στην κρυφή μνήμη για την κάθε εντολή του block, έχουν υπολογιστεί πάνω στις εντολές μετά το μετασχηματισμό.

Ο σκοπός της καινούργιας έννοιας που προσθέτουμε, του μετασχηματισμού, είναι να βοηθήσει να έχουμε τις ίδιες τιμές σε τουλάχιστον μια στήλη που αντιστοιχεί σε κάποιο parity bit position για όλα τα blocks. Ξέροντας, για παράδειγμα, ότι τουλάχιστον ένα από τα 64 parity bits που αποθηκεύονται στην κρυφή μνήμη έχει πάντα την τιμή 0 για όλα τα blocks εντολών, τότε δεν είναι ανάγκη να το αποθηκεύουμε. Με αυτό τον τρόπο μειώνουμε τον χώρο που καταναλώνεται στην κρυφή μνήμη για τα bits προστασίας. Εναλλακτικά για να μειώσουμε την ενέργεια που καταναλώνεται στην κρυφή μνήμη, αυτό το parity bit μπορεί να παραμένει εκεί αλλά να είναι σβηστό έτσι ώστε να μην καταναλώνει ενέργεια.

Στο σχήμα 6.4 αυτό αντιστοιχεί στο Parity Column, το οποίο έχει την ίδια τιμή για όλα τα blocks εντολών.

Όταν διαβάζουμε ένα μπλοκ εντολών από την κρυφή μνήμη και θέλουμε να ελέγξουμε αν έχει προκύψει κάποιο σφάλμα, ξαναχρησιμοποιείται ο μηχανισμός που εκτελεί το ίδιο μετασχηματισμό στις εντολές, που χρησιμοποιήθηκε για να υπολογίσουμε τα parity bits πριν να αποθηκευτούν. Η μονάδες για υπολογισμό και έλεγχο των parity bits είναι οι ίδιες με αυτές στα σχήματα 6.3.3.a και 6.3.3.b.



Σχήμα 6. 4: Λογικό σχήμα της κρυφής μνήμης εντολών για να παρέχουμε την ίδια προστασία με πιο λίγα parity bits (μέγεθος parity κώδικα αποθηκευμένο στην κρυφή μνήμη < 64 bits)

Κεφάλαιο 7:

Πειραματική Αξιολόγηση

7.1 Μεθοδολογία

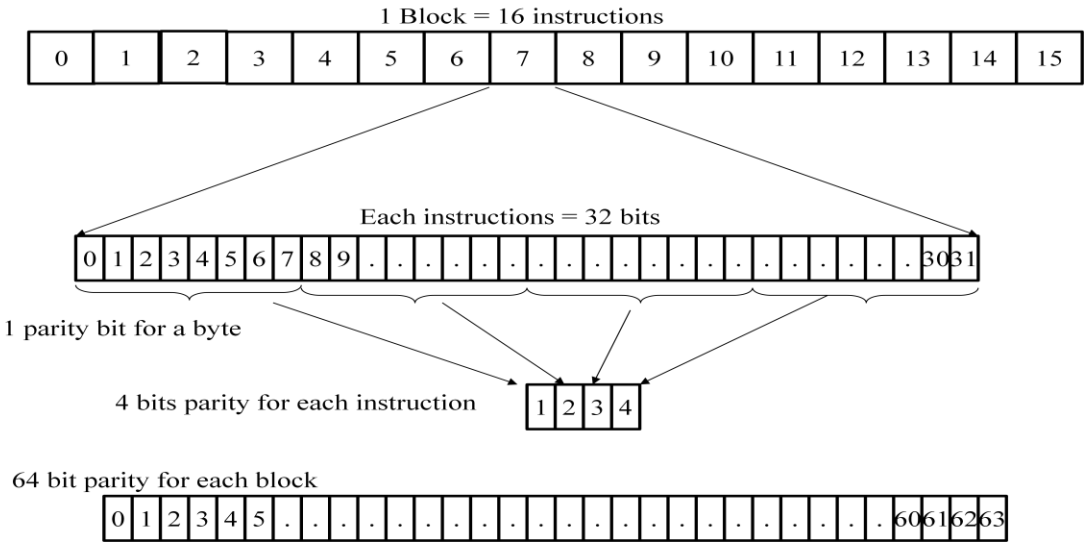
7.1.1 Offline ανάλυση διαφόρων Spec2000 benchmarks

Ως αρχικό βήμα σε αυτή την μελέτη χρησιμοποιήθηκε μια ανάλυση στις εντολές του offline κώδικα των 26 Spec2000 benchmarks (πίνακας 7.1) [11] τα οποία είχαν μεταγλωττιστεί αρχικά σε RISC αρχιτεκτονική Alpha. Υποθέτουμε ότι οι εντολές στην κρυφή μνήμη είναι λογικά ομαδοποιημένα σε blocks όπου το κάθε block περιέχει 16 εντολές μεγέθους 32 bits. Κάθε εντολή προστατεύεται από 4 parity bits , οπού το κάθε parity bit ανιχνεύει σφάλματα σε byte granularity (σχήμα 7.1.1).

Spec2000 Benchmarks

ammp00	gzip00b
applu00	lucas00
apsi00	mcf00b
art00	mesa00
bzip200	mgrid00
crafty00	parser00
eon00b	perlbmk00
equake00	sixtrack00
facerec00	swim00
fma3d00	Twolfoo
galgel00	vortex00
gap00	vpr00
gcc00	wupwise00

Πίνακας 7.1: Spec2000 benchmarks



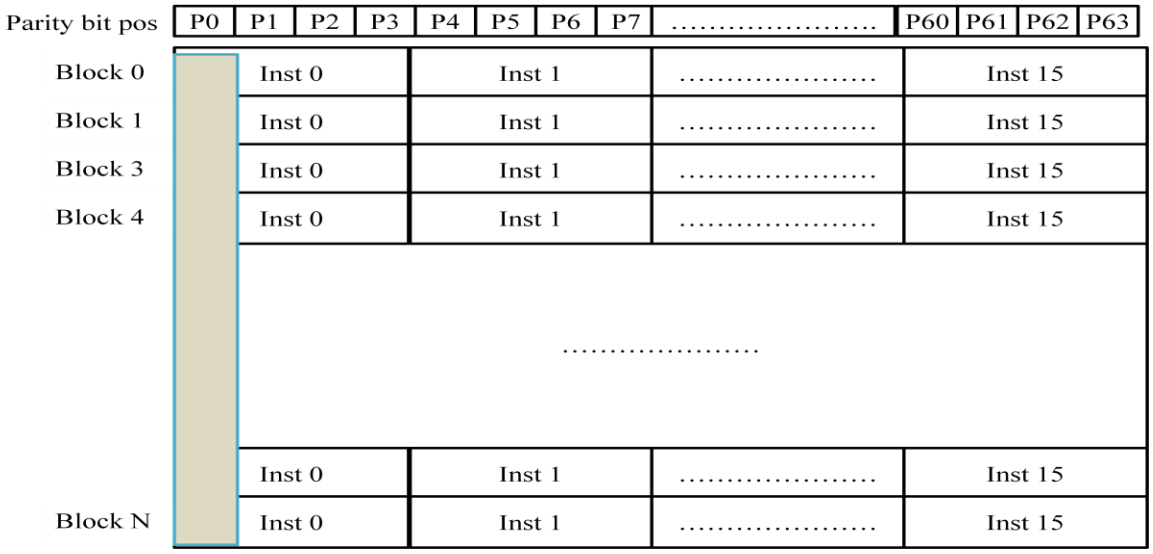
Σχήμα 7.1.1: Λογική ομαδοποίηση των στατικών εντολών

Είναι σημαντικό να αναφέρουμε ότι δεν αναλύθηκαν οι εντολές που διαβάζουμε από την κρυφή μνήμη εντολών και ούτε οι δυναμικές εντολές αλλά οι στατικές εντολές που υπάρχουν στο offline κώδικα των διαφόρων benchmarks. Η πιο πάνω οργάνωση εντολών και ο αριθμός τον parity bits για την κάθε εντολή είναι μια υπόθεση που κάναμε σε αυτή την μελέτη.

Όπως φαίνεται και στο πιο πάνω σχήμα 7.1.1 για να προστατεύουμε ένα block εντολών χρησιμοποιούμε 64 bits even parity. Η κάθε μια από τις εντολές ανάλογα με την διεύθυνση της μπορεί να αποθηκευτεί σε μια από τις 16 θέσεις στο block από το 0 έως το 15. Για να καθορίσουμε την θέση της εντολής μέσα στο block χρειαζόμαστε 4 bits τα οποία αποθηκεύονται στις θέσεις 6 έως 2 του 32 bit καταχωρητή program counter (PC). Αυτά τα bits είναι το offset το οποίο έχουμε εξηγήσει πιο πάνω στο τρίτο κεφάλαιο, που δείχνει πιο δεδομένα από το block θέλουμε, στην μελέτη αυτή είναι 4 bits έτσι ώστε να επιλέξουμε μια εντολή που μπορεί να βρίσκεται σε διαφορετικές διευθύνσεις μέσα στο block από το 0 έως 15 . Τα 2 τελευταία least significant bit του PC είναι 0.
Εφόσον το μέγεθος των εντολών είναι 32 bits και το κάθε block περιέχει 16 εντολές τότε το μέγεθος του block είναι 16 * 32 = 512 bits.

Εάν ο συνολικός αριθμός των εντολών που διαβάζουμε από τον offline κώδικα των benchmarks είναι X τότε ο συνολικός αριθμός των blocks στα οποία θα αποθηκευτούν οι εντολές μας είναι N = X/16. Στο πιο κάτω σχήμα 7.1.2 παρουσιάζεται μια λογική

ομαδοποίηση των εντολών σε blocks, ο αριθμός των block που υπάρχουν ανάλογα με το μέγεθος του offline κώδικα και τα 64 parity bits από το 0 έως 63 ανάλογα με τις εντολές που προστατεύουν. Για παράδειγμα το parity bits που βρίσκονται στις θέσεις από το 0 έως το 3 προστατεύουν σε byte granularity την πρώτη εντολή που βρίσκετε στην διεύθυνση 0 σε όλα τα blocks.



Σχήμα 7.1.2: τα blocks των εντολών και οι θέσεις των parity bits

Ως πρώτο βήμα σε αυτή την μελέτη, μετρήσαμε τον αριθμό των ίδιων τιμών που βρίσκονται σε κάθε στήλη που αντιστοιχεί στο κάθε ένα από τα 64 parity bit position (πχ. ο αριθμός των ίδιων τιμών που αντιστοιχεί στην στήλη για το parity bit position P0 όπως φαίνεται στο σχήμα 7.1.2).

Για να αναλύσουμε τις τιμές που υπάρχουν στο κάθε ένα από τα parity bit positions, χρησιμοποιήσαμε δυο σημαντικούς παραμέτρους **Pratio** και **Benefit Function**. Το Pratio είναι το ποσοστό των πιο συχνών τιμών που βρίσκεται σε κάθε στήλη αντίστοιχα με το Parity bit position. Για παράδειγμα εάν μια στήλη που αντιστοιχεί σε κάποιο parity bit position έχει για όλα τα blocks 60 % μηδενικά και 40 % άσσους τότε το ποσοστό των ίδιων πιο συχνών τιμών είναι 60% . Είναι σημαντικό να τονίσουμε ότι οι πιο συχνές τιμές μπορεί να είναι μηδενικά ή άσσοι . Για το κάθε ένα parity bit position ορίζουμε το Pratio βάση της πιο κάτω συνάρτησης:

$$\text{For each i=0 to 63 } \{\textbf{Pratio[i]} = \sum_{j=0}^{n-1} \text{Pi[j]}\}$$

οπού το i είναι ένα από τα 64 parity bit positions από το 0 έως 63, το j είναι ο αριθμός των γραμμών που υπάρχουν για το κάθε parity bit position (ο αριθμός των γραμμών = με των αριθμό των blocks), το P[j] είναι η τιμή του parity bit που αντιστοιχεί σε κάποια στήλη και το n είναι ο συνολικός αριθμός των γραμμών (το οποίο είναι το ίδιο με τον αριθμό των blocks).

Για παράδειγμα παρατηρώντας το σχήμα 7.1.2 η στήλη που αντιστοιχεί στο P0 έχει n διαφορετικές τιμές ανάλογα με τον αριθμό του block. Π.χ. για το block 0 έχουμε P0[0] για το block 1 έχουμε P0[1] και συνεχίζουμε μέχρι στο τελευταίο block P0[n-1]. Για αυτή την στήλη το Pratio[0]=P0[0]+P0[1]+…+P0[n-1]

Το Benefit Function ορίζεται από την ακόλουθη συνάρτηση:

$$\textbf{Benefit Function} = \sum_{i=0}^{63} \text{Pratio[i]}$$

ως άθροισμα των 64 τιμών PRatios που έχουμε για κάθε στήλη του parity bit position.

Στο τέλος της πρώτης ανάλυσης είχαμε για το κάθε ένα από τα 26 Spec2000 benchmarks το μεγαλύτερο ποσοστό των ίδιων τιμών για το κάθε parity bit position. Το **Benefit Function** που χρησιμοποιήθηκε είναι ένα μετρό σύγκρισης των αποτελεσμάτων πριν και μετά την χρήση της μεθόδου μας.

7.1.2 Offline ανάλυση διαφόρων Spec2000 benchmarks μετά τις μεταθέσεις (permutations).

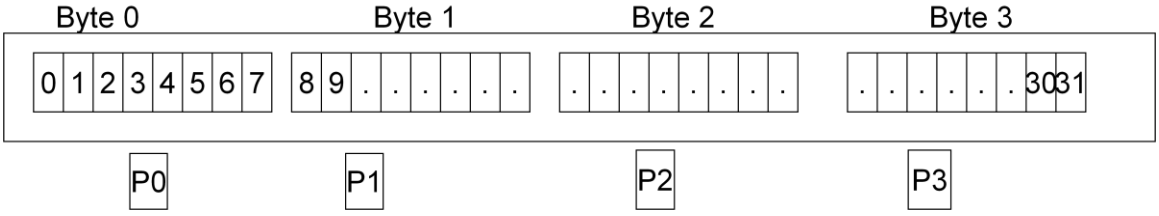
Όπως έχουμε αναφέρει ο αρχικός στόχος αυτής της μελέτης ήταν να έχουμε στήλες που αντιστοιχούν σε κάποιο ή κάποια parity bit positions οπού ο συνολικός αριθμός των ίδιων τιμών είναι 100% έτσι ώστε να τα αποφύγουμε. Η μέθοδος που χρησιμοποιήθηκε για να ατυχούμε τον στόχο μας είναι οι μεταθέσεις (Permutations).

7.1.2.1 Τι είναι οι μεταθέσεις (Permutations) και σε τι ωφελούν

Οι μεταθέσεις (Permutations) είναι η ανταλλαγή των τιμών μεταξύ των 32 θέσεων μέσα στην εντολή, έτσι ώστε να πετύχουμε μεγαλύτερο ποσοστό των ίδιων τιμών σε τουλάχιστον μια στήλη που αντιστοιχεί σε κάποιο parity bit position. Ανταλλάζουμε τις τιμές μεταξύ θέσεων της ίδιας εντολής που βρίσκονται σε διαφορετικά bytes.

Βλέποντας το σχήμα 7.2.1 ο αλγόριθμος ξεκινά ως έξης:

Αλλάζει την τιμή του bit που βρίσκεται στην θέση 0 με την τιμή του bit που βρίσκεται στην θέση 8. Ελέγχουμε εάν μετά την αλλαγή το parity bit που αντιστοιχεί σε αυτό το byte έχει αλλάξει στην επιθυμητή τιμή, έτσι ώστε να πετύχουμε να κάνουμε το ποσοστό τον ίδιων τιμών που αντιστοιχεί σε αυτό το parity bit position μεγαλύτερο. Εάν για παράδειγμα στην στήλη που αντιστοιχεί στο P0 έχουμε 60% μηδενικά και το P0 που αντιστοιχεί στο πρώτο byte μιας εντολής είναι έχει την τιμή 1, τότε εάν μετά την ανταλλαγή της τιμής της θέσης 0 με την τιμή που βρίσκεται στην θέση 8 έχουμε καταφέρει το P0 να είναι 0 οδηγεί σε αύξηση των πιο συχνών τιμών. Εάν όμως αυτή η αλλαγή δεν έχει κάποιο όφελος τότε συνεχίζουμε αλλάζοντας τη τιμή που βρίσκεται στην θέση 0 με την τιμή που βρίσκεται στην θέση 9 μέσα στην ίδια την εντολή . Συνεχίζεται αυτός ο έλεγχος με τα υπόλοιπα bits του byte 0. Η ίδια διαδικασία ακολουθείται και για τα υπόλοιπα bytes της εντολής αλλά για παράδειγμα την τιμή στην θέση 8 της εντολής η οποία βρίσκεται στο δεύτερο byte μπορούμε να την αλλάξουμε με τις τιμές που βρίσκονται στις θέσεις από 16 μέχρι 31 της εντολής. Οι ίδιες αλλαγές γίνονται και όταν θέλουμε να αλλάξουμε την θέση 9 της εντολής που βρίσκεται στο byte 2. Τις θέσεις που βρίσκονται στο byte 3 της εντολής δηλαδή τις θέσεις από 16 έως 23 μπορούμε να τις ανταλλάξουμε μόνο με τις θέσεις που βρίσκονται στο τέταρτο byte από 24 έως 31. Πιο περιληπτικά οι ανταλλαγές που γίνονται μεταξύ των θέσεων σε μια εντολή παρουσιάζονται στον πίνακα 7.2.1



Σχήμα 7.2.1: Επιτρεπόμενες αλλαγές στις θέσεις των bits μέσα στην ίδια την εντολή

Κάθε bit που ανήκει στο Byte0 pos[0 έως 7] γίνεται permute με τις θέσεις 8 έως 31
Κάθε bit που ανήκει στο Byte1 pos[8 εως15] γίνεται permute με τις θέσεις 16 έως 31

Κάθε bit που ανήκει στο Byte2 pos[16 έως 23] γίνεται permute με τις θέσεις 24 έως 31

Πίνακας 7.2.1: Επιτρεπόμενες αλλαγές στις θέσεις των bits μέσα στην ίδια την εντολή

7.1.2.2 Ανάλυση του offline κώδικα μετά τις μεταθέσεις (permutations)

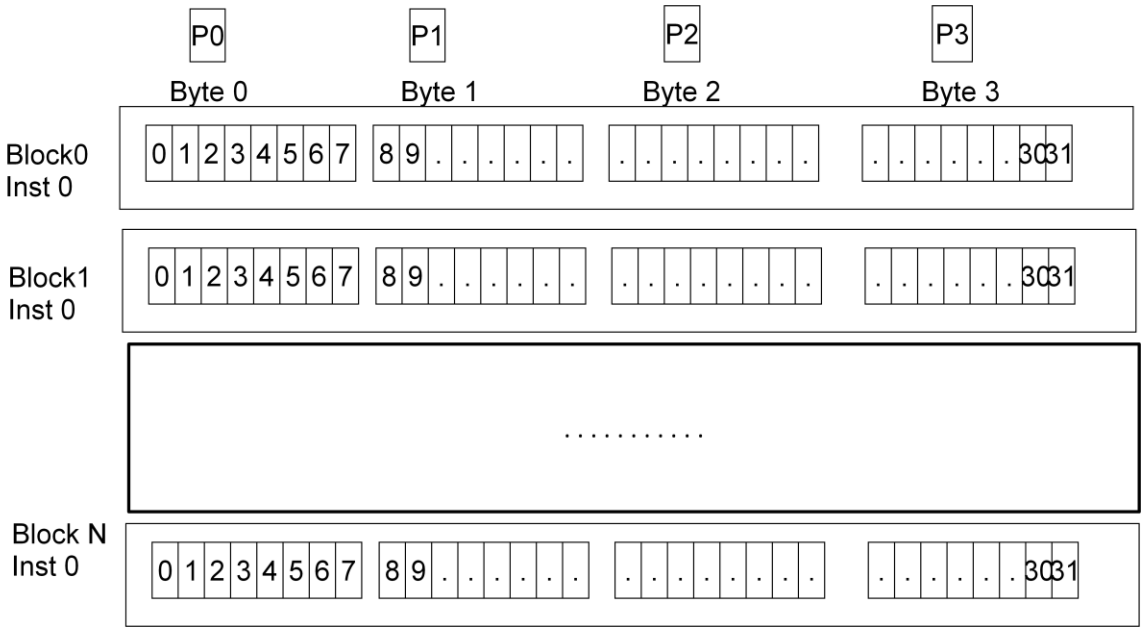
Όπως έχουμε αναφέρει και πιο πριν οι εντολές ομαδοποιούνται σε blocks. Η κάθε μια από αυτές έχει μια διεύθυνση στο block από 0 έως 15. Αναλύσαμε τις εντολές ανάλογα με την διεύθυνση τους στο block, δηλαδή για όλες τις εντολές που βρίσκονται στην ίδια θέση μέσα στο block είχαμε υπολογίσει αρχικά τα **Pratios** και το **Benefit Function** το οποίο το ορίζαμε ανάλογα με την θέση της εντολής στο block ως

$$Benefit\ Function = \sum_{i=0}^{3} \text{Pratio}[4 * \text{x} + \text{i}]$$

Το x είναι μια θέση από 0 έως 15 της εντολής στο block και το i είναι το parity bit position για την κάθε εντολή. Για παράδειγμα στο σχήμα 7.2.2 παρουσιάζουμε την ομάδα των εντολών που ανήκουν στην θέση 0 μέσα στο block. Με τον ίδιο τρόπο ομαδοποιήσαμε και τις εντολές που ανήκουν στις άλλες θέσεις μέσα στο block. Η κάθε ομάδα ξεχωριστά αποτελείται από 4 parity bit position. Για την κάθε ομάδα ξεχωριστά είχαμε υπολογίσει τα Pratioς τους και το benefit function το οποίο ήταν το άθροισμα των τεσσάρων Pratios.

Benefit Function ανά ομάδα = Pratio[0] + Pratio[1] + Pratio[2] + Pratio[3]

Μετά την κάθε μετάθεση (permutation) ανά εντολή επαναϋπολογίσουμε τα Pratios και το benefit function τους έτσι ώστε να συγκρίνουμε τα καινούργια αποτελέσματα με τα παλιά και να βρούμε τις καλύτερες μεταθέσεις (permutations) οι οποίες οδηγούν σε ένα μεγαλύτερο ποσοστό ίδιων τιμών σε κάθε στήλη. Το benefit function το χρησιμοποιούμε για να συγκρίνουμε και να αναλύσουμε εάν φτάσαμε σε μια καλύτερη λύση. Εάν μετά από κάθε μετάθεση (permutation) έχω μεγαλύτερο Benefit Function σημαίνει ότι έχω πλησιάζει σε στήλες με μεγαλύτερο ποσοστό των ίδιων τιμών. Στο τέλος κρατούμε τις μεταθέσεις (permutations) οι οποίες οδηγούν στην πιο μεγάλη τιμή του Benefit Function



Σχήμα 7.2.2: Λογική ομαδοποίηση εντολών για όλα τα blocks ανάλογα με την θέση τους στο block

Είναι σημαντικό να αναφέρουμε ότι οι μεταθέσεις (permutations) δεν γίνονται σε επίπεδο του byte για το λόγο ότι υπολογίσουμε το Parity σε byte granularity. Οι ανταλλαγές μεταξύ των θέσεων που βρίσκονται στο ίδιο byte δεν αλλάζουν την τιμή του Parity bit που αντιστοιχεί σε αυτό το byte. Επίσης οι μεταθέσεις (permutations) γίνονται μόνο forward για το λόγο ότι αλλάζοντας ξανά για παράδειγμα το bit στην θέση 8 με το bit στην θέση 1 το parity bit θα γίνει όπως πριν.

7.1.2.3 Παράδειγμα

Ο σκοπός αυτού του παραδείγματος είναι να γίνει μια καλύτερη κατανόηση της μεθοδολογίας που χρησιμοποιήθηκε για να πλησιάζουμε σε στήλες των parity bit positions που έχουν μεγάλο ποσοστό των ίδιων τιμών.

Υποθέτουμε ότι έχουμε δύο εντολές που βρίσκονται στην ίδια θέση σε δυο διαφορετικά blocks. Πιο κάτω παρουσιάζονται τα βήματα που ακολουθήσαμε στην ανάλυση μας για να φτάσουμε σε στήλες ανάλογα με το parity bit position που έχουν μεγάλο ποσοστό των ίδιων τιμών

Βήμα 1: Υπολογίσουμε τις τιμές των parity bits ανά byte granularity

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	P3	P2	P1	P0
1 0 0 1 0 0 0 0	0 1 1 1 0 0 0 1	0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	0	0	0	1
1 0 1 0 0 0 0 1	1 0 0 1 1 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0	1	1	1	0

Σχήμα 7.2.3: Αρχικά δεδομένα των εντολών και τα parity που αντιστοιχούν σε αυτά

Βήμα 2: Υπολογίσουμε τα Pratios για την κάθε parity στήλη των parity bits

Pratio3 = 50, Pratio2 = 50, Pratio1 = 50, Pratio0 =5 0, Benefit Function= Pratio3 + Pratio2 + Pratio1 + Pratio0 = 200

Βήμα 3: Μετάθεση στα bits των εντολών έτσι ώστε να πετύχουμε να έχουμε την ίδια τιμή σε τουλάχιστον μια στήλη που αντιστοιχεί σε κάποιο parity position

Βήμα 3.1: Μετά την πρώτη μετάθεση (permutation) ανταλλάχτηκαν οι τιμές μεταξύ τις θέσεις 0 και 9 στην εντολή έτσι ώστε να πετύχουμε μεγαλύτερη τιμή του Benefit Function

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	P3	P2	P1	P0
1 0 0 1 0 0 0 0	0 1 1 1 0 0 0 1	0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	0	0	0	1
1 0 1 0 0 0 0 1	1 0 0 1 1 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	1	1	0	1

Σχήμα 7.2.4: Τα δεδομένα των εντολών και τα parity που αντιστοιχούν σε αυτά μετά το permutation που αυξάνει το benefit function κάνοντας τις στήλες P0 και P1 να έχουν τις ίδιες τιμές

Βήμα 3.2: Υπολογίσουμε τα Patios για την κάθε parity στήλη των parity bits μετά την μετάθεση (permutation)

Pratio3 = 50, Pratio2 = 50, Pratio1 = 100, Pratio0 = 100, Benefit Function= Pratio3 + Pratio2 + Pratio1 + Pratio0 = 300. Αφού το benefit function έχει μεγαλύτερη τιμή από την προηγούμενη κρατούμε αυτή ως μια καλύτερη λύση

Βήμα 3.3: Μετά την δεύτερη καλύτερη μετάθεση (permutation) ανταλλάχτηκαν οι τιμές μεταξύ τις θέσεις 16 και 25 στην εντολή έτσι ώστε να πετύχουμε μεγαλύτερη τιμή του Benefit Function από την προηγούμενη.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	P3	P2	P1	P0
1 0 0 1 0 0 **0** 0	0 1 1 1 0 0 0 **0**	0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	0	1	0	1
1 0 1 0 0 0 **1** 1	1 0 0 1 1 0 0 **0**	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0	1	0	1

Σχήμα 7.2.5: Τα δεδομένα των εντολών και τα parity που αντιστοιχούν σε αυτά μετά το permutation που αυξάνει το benefit function κάνοντας τις στήλες P3 και P2 να έχουν τις ίδιες τιμές

Βήμα 3.4: Υπολογίσουμε τα Pratios για την κάθε parity στήλη των parity bits μετά την μετάθεση

Pratio3 = 100, Pratio2 = 100, Pratio1 = 100, Pratio0 = 100, Benefit Function= Pratio3 + Pratio2 + Pratio1 + Pratio0 = 400. Αφού το benefit function έχει την μεγαλύτερη τιμή και έχουμε πετύχει να έχουμε 100% ίδιες τιμές σε όλα από τα τέσσερα parity bit position αυτή είναι η καλύτερη λύση.

Γενικότερα αν καταλήγουμε σε τέτοιες στήλες που έχουμε σε όλη την στήλη την ίδια τιμή δεν χρειάζεται να αποθηκευτεί στην κρυφή μνήμη αφού ξέρουμε για παράδειγμα ότι η τιμή για την στήλη P0 είναι πάντα 0 μετά την σειρά των πιο πάνω μεταθέσεων (permutations).

7.1.2.4 Πολυπλοκότητα των Μεταθέσεων (Permutations)

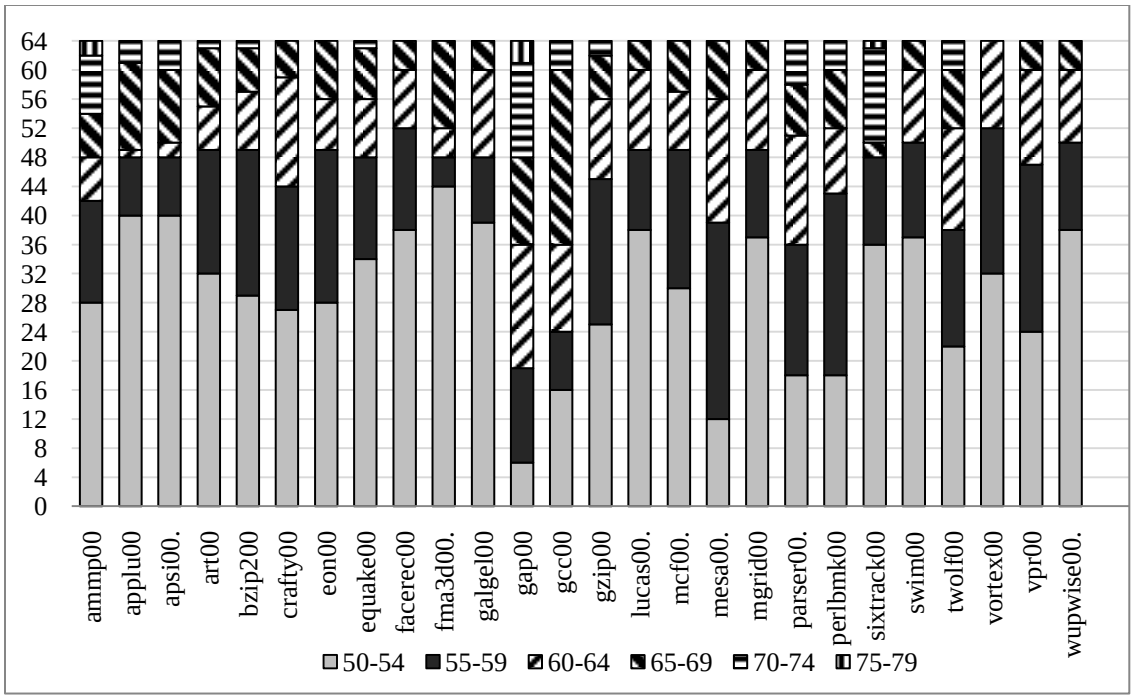
Είναι σημαντικό να αναφέρουμε την χρονική πολυπλοκότητα που χρειάζεται για να πραγματοποιηθούν οι μεταθέσεις. Γενικά σε δεδομένα μεγέθους 32 bits μπορούμε να αλλάξουμε την τιμή του κάθε bit με τις υπόλοιπες 31 θέσεις. Οι συνδυασμοί που μπορούμε να έχουμε με n bits δεδομένα έχει χρονική πολυπλοκότητα της τάξης O(n!) για το λόγο ότι κάθε τιμή μπορεί να συνδυαστεί με τις υπόλοιπες n-1 τιμές. Για να πετύχουμε όλους τους συνδυασμούς που μπορούμε να κάνουμε με n bits όσο αυξάνεται το n αυξάνεται υπερβολικά και ο χρόνος που χρειάζεται για τον υπολογισμό τους Αυτό το γεγονός δεν είναι χρονικά αποδοτικό. Ο αλγόριθμος που χρησιμοποιούμε σε αυτή την μελέτη βρίσκει ένα σύνολο από όλους τους συνδυασμούς που αναφέραμε πιο πάνω. Όπως είδαμε συνδυάζουμε θέσεις των bits στην εντολή οι οποίες δεν βρίσκονται στο ίδιο byte και επίσης δεν αλλάζουμε ξανά τις θέσεις που έχουν ήδη αλλαχτεί για παράδειγμα αν ανταλλάζουμε την θέση 0 με την θέση 8 δεν πάμε προς τα πίσω να αλλάξουμε την

θέση 8 με την θέση 0. Αυτό ισχύει για όλες τις θέσεις μέσα στην εντολή . Η χρονική πολυπλοκότητα του αλγορίθμου μας είναι της τάξης $O(n^2)$ οπού το n είναι το μέγεθος των εντολών.

Επίσης είναι σημαντικό να αναφέρουμε ότι το Permutation Unit που προσθέτουμε στο baseline σχήμα της κρυφής μνήμης με bit προστασίας δεδομένων (Κεφάλαιο 6) μπορεί να είναι προγραμματιζόμενο ή σταθερό. Είναι προγραμματιζόμενο όταν για κάθε μια από τις εφαρμογές που αναλύουμε ο αριθμός των permutations και οι θέσεις που ανταλλάζονται στην ίδια εντολή έτσι ώστε να πετύχουμε να έχουμε 100 % ίδιες τιμές σε όλα τα parity bit positions, είναι διαφορετικές από μια εφαρμογή στην άλλη. Αν όμως για όλες τις εφαρμογές, τις μεταθέσεις (permutations) που γίνονται για να έχουμε τουλάχιστον μια στήλη με 100 % ίδιες τιμές, είναι οι ίδιες για όλες τις εφαρμογές τότε το permutation unit είναι σταθερό.

7.2 Αξιολόγηση αποτελεσμάτων

Σε αυτό το υποκεφάλαιο θα επικεντρωθούμε στην παρουσίαση, ανάλυση και αξιολόγηση αποτελεσμάτων της μελέτης αυτής. Σε αυτό το σημείο θα παρουσιαστούν τα ποσοστά το πιο συχνών τιμών που υπάρχουν για κάθε benchmark έχοντας υπόψη ότι έχουμε 64 στήλες parity bit οπού η κάθε μια αντιστοιχεί σε ένα parity bit position. Αρχικά παρουσιάζονται τα αποτελέσματα χωρίς μεταθέσεις (permutations) και έπειτα τα αποτελέσματα με τις μεταθέσεις (permutations).

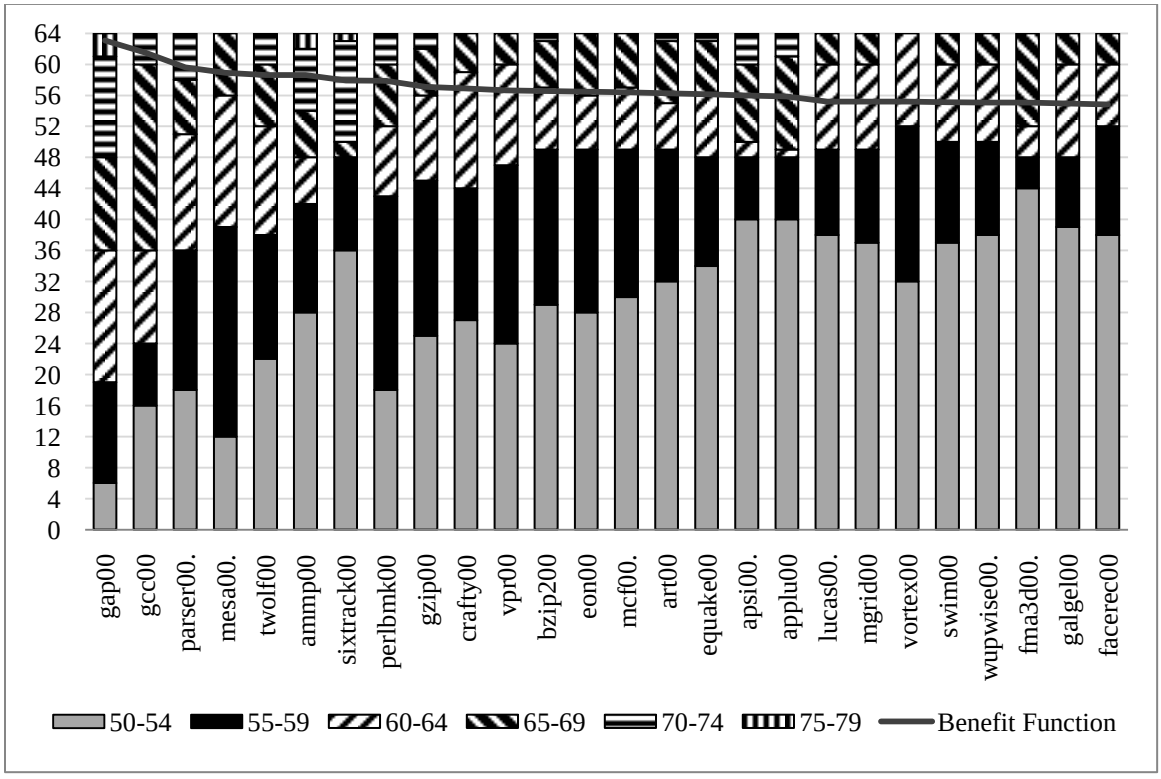


Σχήμα 7.3: Parity Ratios Before Permutation

Από το πιο πάνω σχήμα βλέπουμε το ποσοστό των ίδιων τιμών που αντιστοιχούν σε κάθε Parity bit position. Οι μπάρες αντιπροσωπεύουν ποσοστά των ίδιων τιμών και στον άξονα τον y είναι ο αριθμός των στηλών που έχουν αυτά τα ποσοστά. Στον άξονα τον x είναι τα 26 benchmarks που αναλύσαμε[11]. Παρατηρούμε ότι αρχικά σχεδόν σε όλα τα benchmarks έχουμε αρκετές στήλες με ποσοστό των ίδιων τιμών μεταξύ 50% έως 54%. Παρατηρούμε ότι το ποσοστό των ίδιων τιμών αρχικά δεν είναι μεγάλος. Αν παρατηρούμε το benchmark fma3d00 υπάρχουν 40 στήλες από τις 64 που έχουν 50%-54% ίδιες τιμές. Το ποσοστό των ίδιων τιμών σχεδόν σε όλα τα benchmarks διακυμαίνεται μεταξύ 50% έως 70%. Παρατηρούμε επίσης ότι στις 9 από τα 26 benchmarks παρουσιάζονται στήλες οι όποιες έχουν ποσοστό των ίδιων τιμών μεταξύ 70% έως 74% το οποίο είναι ένα θετικό στοιχειό. Ο μεγαλύτερος αριθμός των στηλών με αυτό το ποσοστό παρουσιάζεται στο gap00 οπού έχουμε 11 από τις 64 στήλες. Το μεγαλύτερο ποσοστό τον ίδιων τιμών (στο ίδιο parity position) είναι από 80%-84% και παρουσιάζεται μόνο σε 2 benchmarks. Στην μελέτη αυτή σημαντικό στοιχείο είναι να έχουμε αρχικά στήλες με μεγάλο ποσοστό των ίδιο τιμών και δεν είναι τόσο σημαντικό αν ο αριθμός των στηλών είναι μεγάλος. Θα ήταν ένα θετικό στοιχείο εάν οι

περισσότερες στήλες έχουν μεγάλο ποσοστό αλλά όπως βλέπουμε από το πιο πάνω σχήμα αυτό δεν ισχύει.

Στο πιο κάτω σχήμα βλέπουμε ένα άλλο μέτρο σύγκρισης ανάλογα με το μέγεθος του εκτελέσιμου των διαφόρων benchmarks την κατανομή των parity στηλών σε αντιστοιχία με το normalized Benefit Function. Έχει γίνει μετασχηματισμό της μεγαλύτερης τιμής του Benefit Function στην τιμή 64. Εάν το μεγαλύτερο Benefit Function έχει την τιμή X τότε η τιμή που χρησιμοποιήθηκε για το normalization των Benefit Functions σε όλα τα benchmarks είναι y=X/64. Το normalized Benefit Function = Benefit Function of each benchmark/y

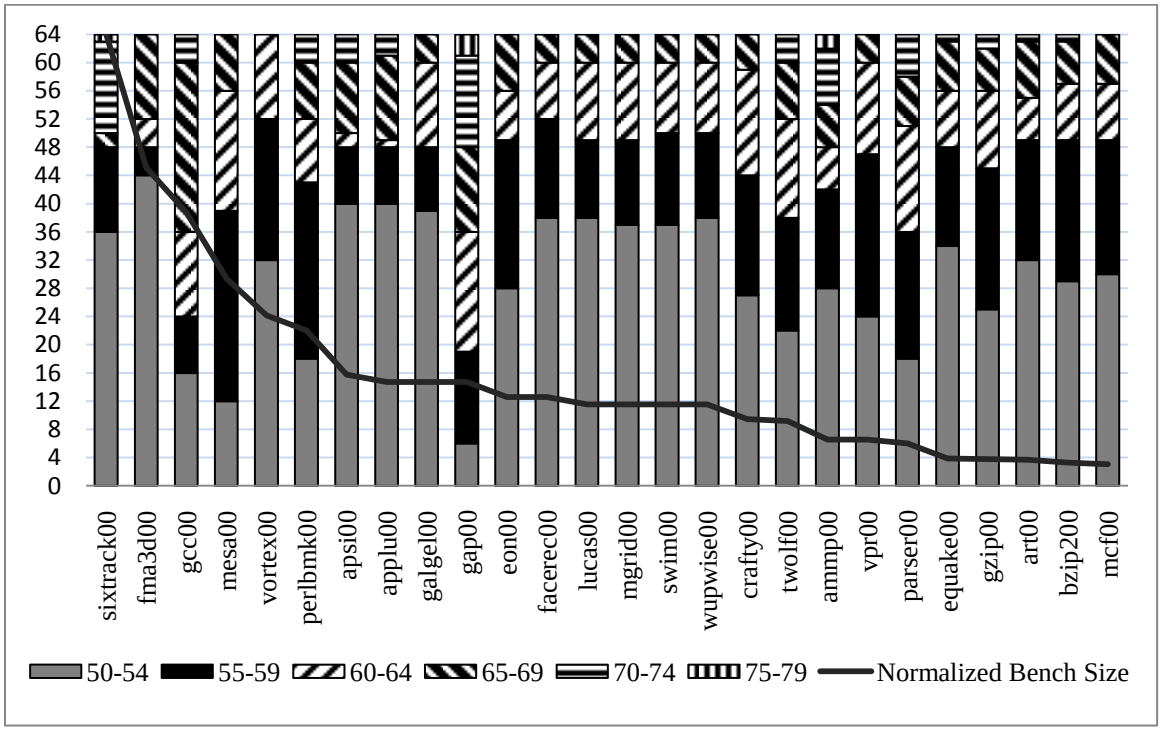


Σχήμα 7.4: Parity Ratios & Normalized Benefit Function before permutations

Παρατηρούμε ότι η μεγαλύτερη τιμή του Benefit Function (το οποίο είναι ένα μέτρο σύγκρισης για μας) είναι στο πρώτο benchmark gap00 το οποίο έχει τον μικρότερο αριθμό στηλών με ποσοστό των ίδιων τιμών από 50%-54%. Παρατηρούμε επίσης ότι η πιο μικρή τιμή του benefit function είναι στα τελευταία benchmarks, στα οποία το ποσοστό των ίδιων τιμών σε μια στήλη διακυμαίνεται μεταξύ 50%-70%. Σημαντική

παρατήρηση είναι ότι το benefit function έχει την καλύτερη τιμή στο πρώτο benchmark οπού η διαφορά με τα άλλα benchmarks είναι ότι εμφανίζονται κάποιες στήλες με ποσοστό των ίδιων τιμών μεταξύ 75%-79% . Από τις πιο πάνω παρατηρήσεις το benefit function έχει την καλύτερη τιμή όπου έχουμε στήλες με μεγαλύτερο ποσοστό των ίδιων τιμών.

Στο πιο κάτω σχήμα παρουσιάζουμε ένα άλλο μέτρο σύγκρισης ανάλογα με το μέγεθος του εκτελέσιμου του κάθε benchmark. Έχει γίνει μετασχηματισμός του μεγαλύτερου benchmark size στην τιμή 64KB. Εάν το μεγαλύτερο benchmark έχει μέγεθος X KB τότε η τιμή που χρησιμοποιήθηκε για το normalization σε όλα τα benchmarks είναι y=X/64. Το normalized Benchmark Size = Size of each benchmark(KB) / y

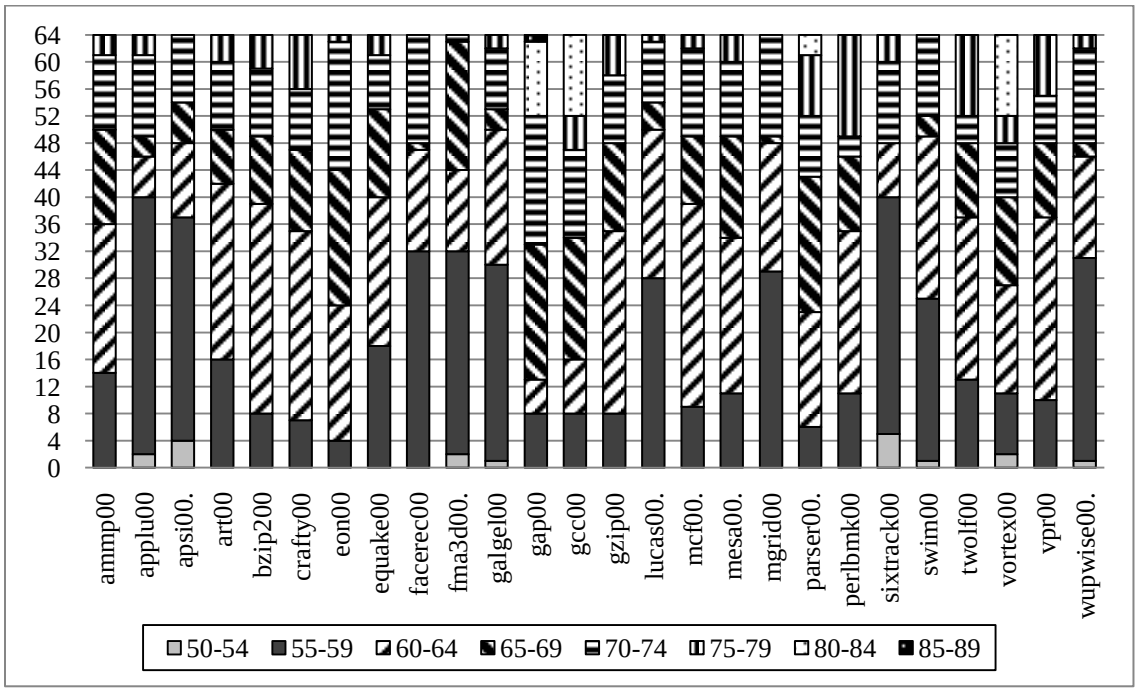


Σχήμα 7.5: Parity Ratios & Normalized Benchmark size before permutations

Βλέποντας το πιο πάνω σχήμα θα αναμέναμε να παρατηρήσουμε ότι το μεγαλύτερο benchmark θα είχε τον πιο μεγάλο αριθμό στηλών που έχουν το μικρότερο ποσοστό των

ίδιων τιμών δηλαδή θα είχε περισσότερες στήλες με ποσοστό 50% - 54% των ίδιο τιμών, από όλα τα benchmarks. Η υπόθεση αυτή έγινε για το λόγο ότι σε μεγαλύτερα benchmarks θα αναμέναμε να έχουμε περισσότερες εντολές οι όποιες θα ήταν διαφορετικές. Όπως φαίνεται από το πιο πάνω σχήμα το πρώτο benchmark το sixtrack00 ο οποίος έχει το μεγαλύτερο μέγεθος δεν έχει το μεγαλύτερο αριθμό στηλών με ποσοστό 50% - 54% ίδιων τιμών. Επίσης εάν το συγκρίνουμε με το τελευταίο benchmark το οποίο έχει το μικρότερο μέγεθος, στο πρώτο benchmark εμφανίζονται και στήλες οι οποίες έχουν 70% - 74% ίδιες τιμές. Στο τελευταίο benchmark to mcf00 παρατηρούμε ότι υπάρχουν στήλες με ποσοστό μικρότερο από 70% - 74% ίδιων τιμών. Μια εικασία που κάνουμε για να εξηγήσουμε αυτό το γεγονός είναι ότι στα μεγαλύτερα προγράμματα μπορεί να υπάρχει duplicate κώδικας λόγο compiler optimization το οποίο οδηγεί να έχουμε duplicate εντολές. Με αυτό τον τρόπο έχει μεγάλη πιθανότητα στις ίδιες διευθύνσεις σε διαφορετικά blocks να έχουμε ίδιες εντολές οδηγώντας έτσι τα parity bits στην ίδια στήλη να είναι οι ίδιες.

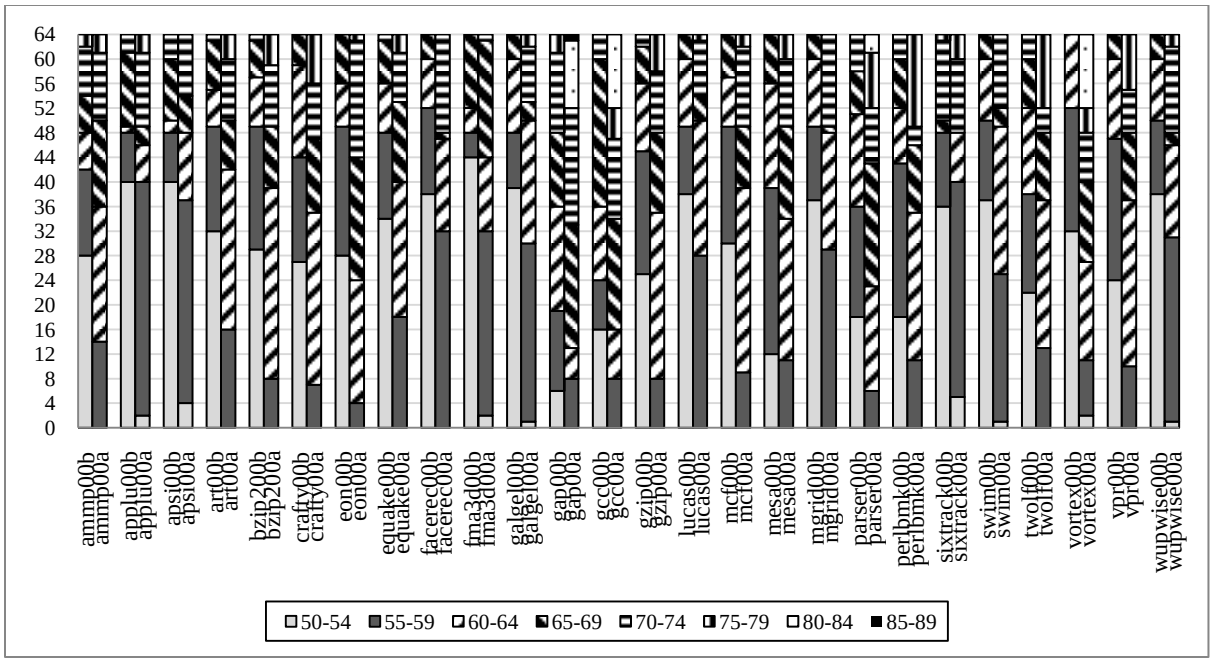
Στο πιο κάτω σχήμα παρουσιάζεται ο αριθμός των στηλών (y άξονας) και αντίστοιχα το ποσοστό των ίδιων τιμών για την κάθε μια στήλη μετά τα permutations.



Σχήμα 7.6: Parity ratios after permutations

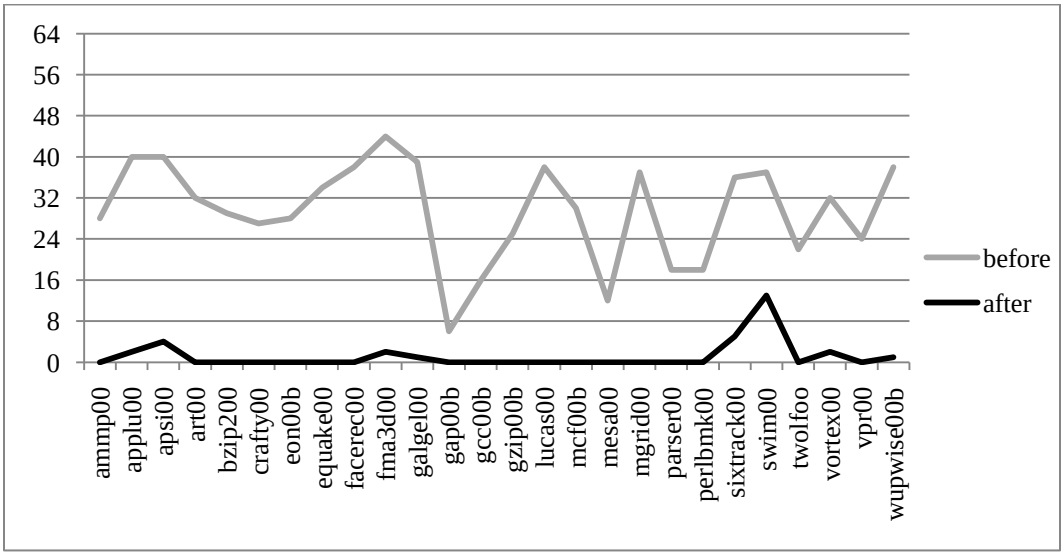
Όπως βλέπουμε από το πιο πάνω σχήμα μετά τα permutations έχει μειωθεί ο αριθμός των στηλών που έχουν 50%-54% ίδιες τιμές και έχει αυξηθεί ο αριθμός των στηλών με μεγαλύτερα ποσοστά των ίδιων τιμών. Σημαντικό να παρατηρήσουμε είναι ότι σε κάποια από τα benchmarks όπως gap00, gcc00, parser00,vortex00 έχουμε στήλες με 80%-84% ίδιες τιμές. Επίσης στο gap00 έχουμε στήλες με 85%-89% ίδιες τιμές. Όπως βλέπουμε μετά τις μεταθέσεις (τα permutations) δεν πετύχαμε να έχουμε στήλες με 100 % ίδιες τιμές το οποίο ήταν το επιθυμητό μας αποτέλεσμα αλλά πετύχαμε να αυξάνουμε το ποσοστό των ίδιων τιμών και να μειώσουμε σημαντικά τον αριθμό των στηλών με μικρό ποσοστό από 50%-54%. Μετά τις μεταθέσεις (permutations) έχει αυξηθεί ο αριθμός των στηλών με ποσοστό από 60% έως 79% και έχουν παρουσιαστεί καινούργια ποσοστά από 80% έως 89% σε κάποια από τα benchmarks

Στο πιο κάτω σχήμα συγκρίνουμε το ποσοστό των ίδιων τιμών που υπάρχουν σε κάθε parity bit position , σε κάθε στήλη πριν και μετά τις μεταθέσεις (permutations). Παρατηρούμε ότι σε όλα τα benchmarks έχει μειωθεί σημαντικά ο αριθμός των στηλών που είχαν ποσοστό των ίδιων τιμών από 50%-54%. Γενικότερα στα περισσότερα benchmarks έχει αυξηθεί ο αριθμός των στηλών με ποσοστό των ίδιων τιμών πάνω από 55%. Σημαντικό είναι να αναφέρουμε εδώ ότι στα περισσότερα benchmarks παρουσιαστήκαν στήλες με ποσοστά των ίδιων τιμών μεταξύ των 75%-79%, το οποίο δεν υπήρχε πριν να εφαρμόζουμε την μέθοδο των μεταθέσεων (permutations) . Αυτό που παρατηρούμε γενικότερα είναι ότι ανάλογα με το μεγαλύτερο ποσοστό των ίδιων τιμών το οποίο υπήρχε πριν τις μεταθέσεις (permutations) για κάθε benchmark (MAXbefore), μετά τις μεταθέσεις (permutation) είτε ο αριθμός των στηλών με αυτό το ποσοστό έχει αυξηθεί σημαντικά είτε έχουν παρουσιαστεί στήλες με μεγαλύτερο ποσοστό από αυτό που υπήρχε πριν όπως για παράδειγμα έχουν παρουσιαστεί στήλες με ποσοστό των ίδιων τιμών από 80%-89% το οποίο δεν υπήρχε πριν τις μεταθέσεις (permutations) (MAXafter>MAXbefore)



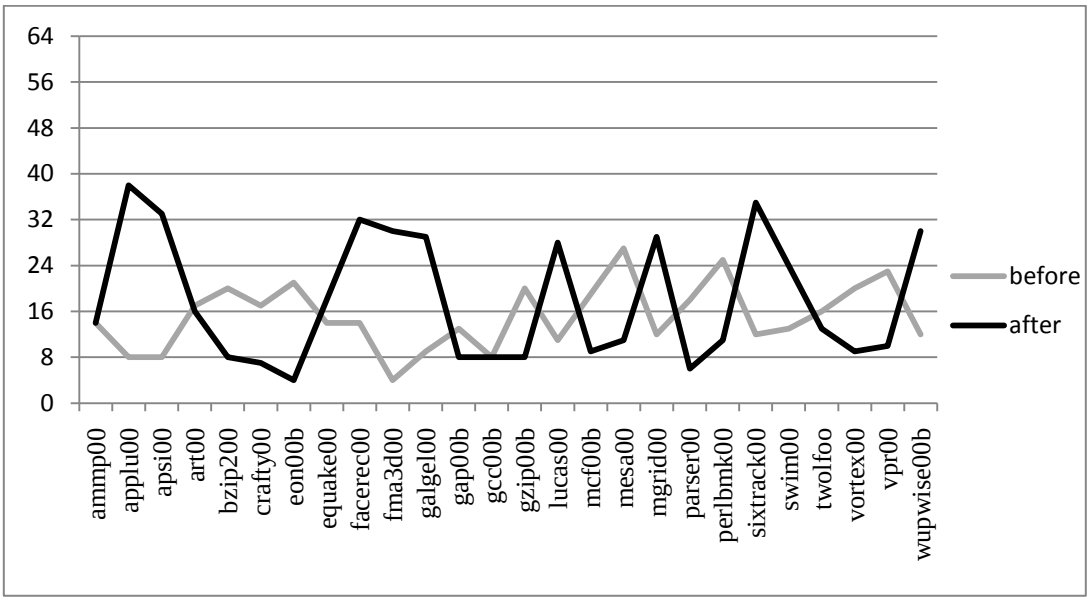
Σχήμα 7.7: Parity Ratios Before & After Permutations

Στα πιο κάτω σχήματα στον άξονα του x παρουσιάζονται όλα τα benchmarks, και στον άξονα του y παρουσιάζεται ο αριθμός των στηλών που έχουν το αντίστοιχο ποσοστό των ίδιων τιμών το οποίο ανήκει σε μια από τις ομάδες: 50%-54%, 55%-59%, 60%-64%, 65%-69%, 70%-74%, 75%-79%, 80%-84%, 85%-89%.



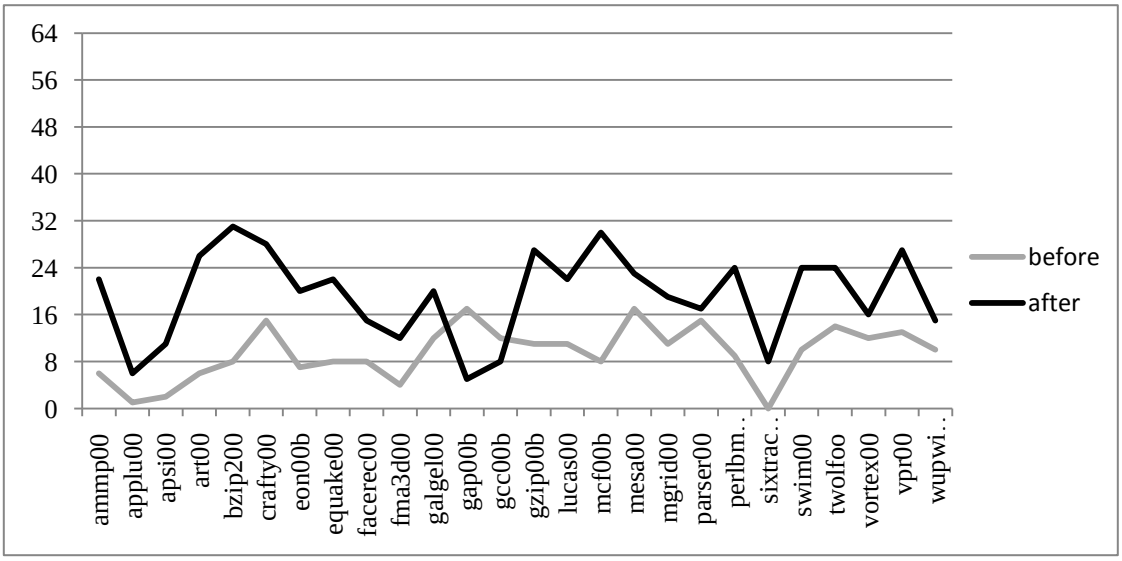
Σχήμα 7.8: Pratio = 50%-54% before and after permutation

Πιο πάνω (Σχήμα 7.8) βλέπουμε ότι ο αριθμός των στηλών που έχουν 50%-54% ποσοστό των ίδιων τιμών μετά τις μεταθέσεις (permutations) έχει μειωθεί σημαντικά σε όλα τα benchmarks.



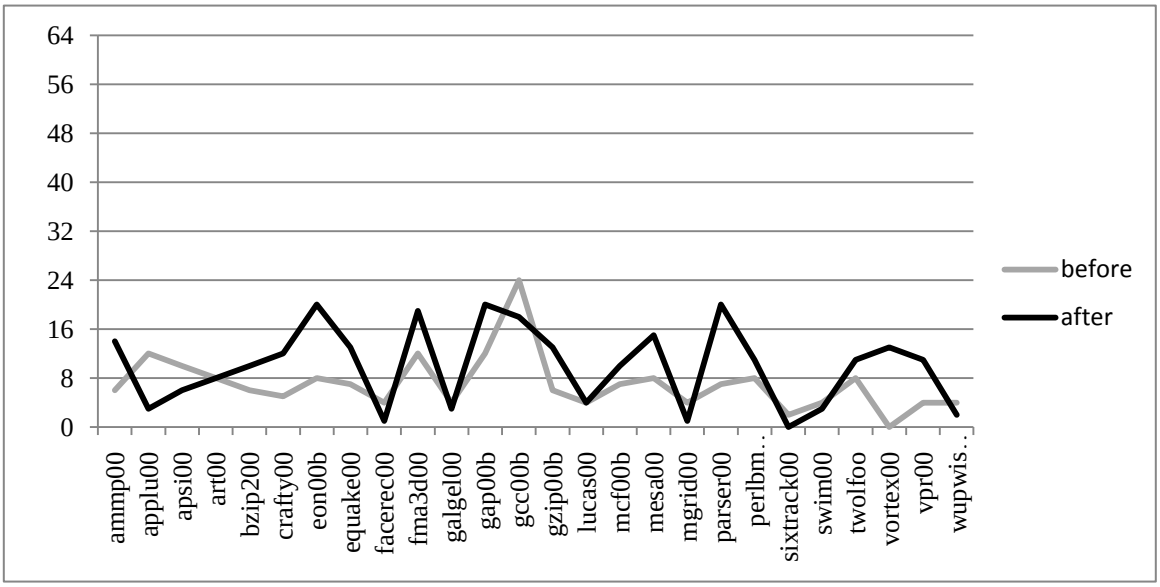
Σχήμα 7.9: Pratio = 55%-59% before and after permutation

Πιο πάνω (Σχήμα 7.9) βλέπουμε ότι ο αριθμός των στηλών που έχουν 55%-59% ίδιες τιμές μετά τις μεταθέσεις (permutations) σε κάποια από τα benchmarks έχει μειωθεί και σε κάποια άλλα έχει αυξηθεί.



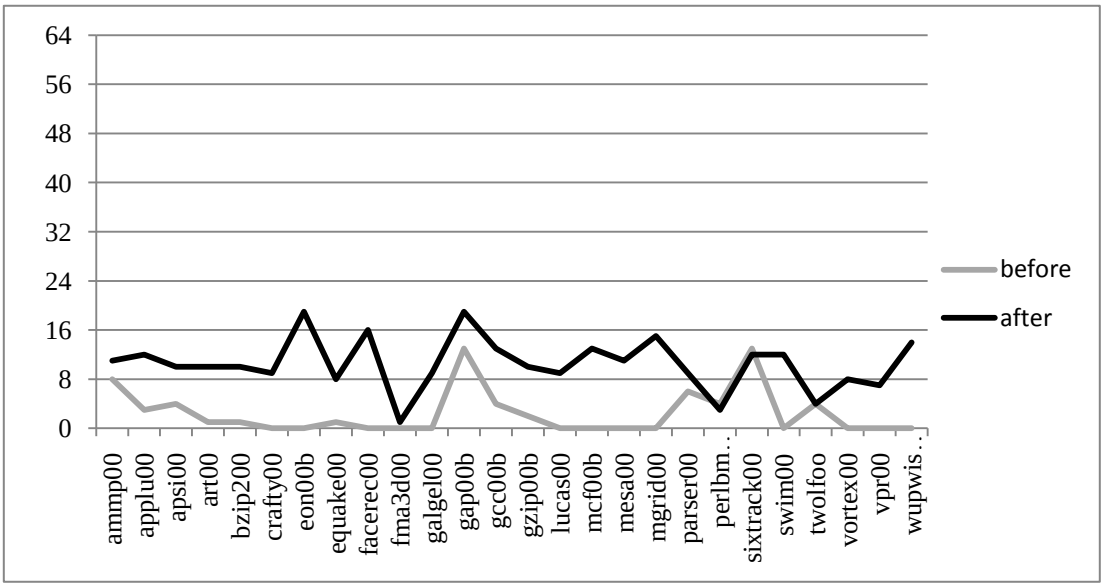
Σχήμα 7.10: Pratio = 60%-64% before and after permutation

Πιο πάνω (Σχήμα 7.10) βλέπουμε ότι ο αριθμός των στηλών που έχουν 60%-64% ίδιες τιμές μετά τις μεταθέσεις (permutations) έχει αυξηθεί στα περισσότερα benchmarks



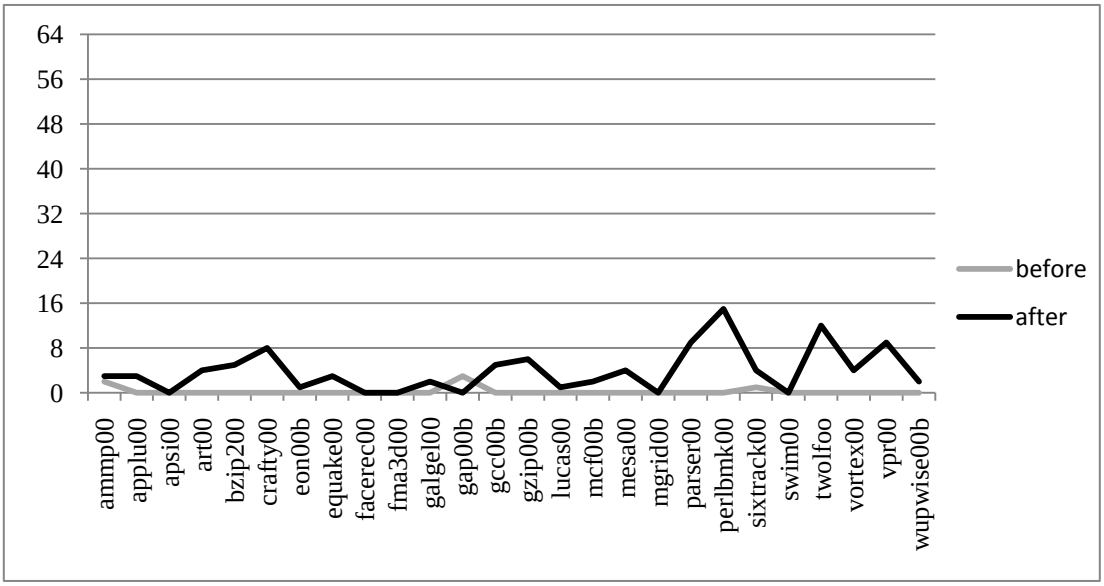
Σχήμα 7.11: Pratio = 65%-69% before and after permutation

Στο πιο πάνω (Σχήμα 7.11) βλέπουμε ότι ο αριθμός των στηλών που έχουν 65%-69% ίδιες τιμές μετά τις μεταθέσεις (permutations) σε κάποια από τα benchmarks έχει αυξηθεί αλλά σε κάποια άλλα έχει μειωθεί.



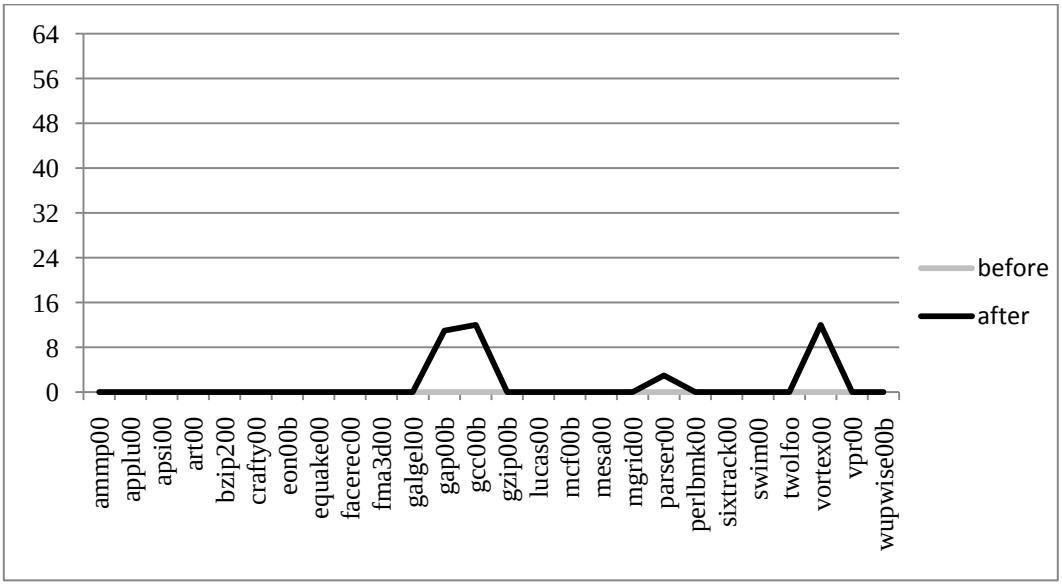
Σχήμα 7.12: Pratio = 70%-74% before and after permutation

Πιο πάνω (Σχήμα 7.12) βλέπουμε ότι ο αριθμός των στηλών που έχουν 70%-74% ίδιες τιμές μετά τις μεταθέσεις (permutations) στα περισσότερα benchmarks έχει αυξηθεί.



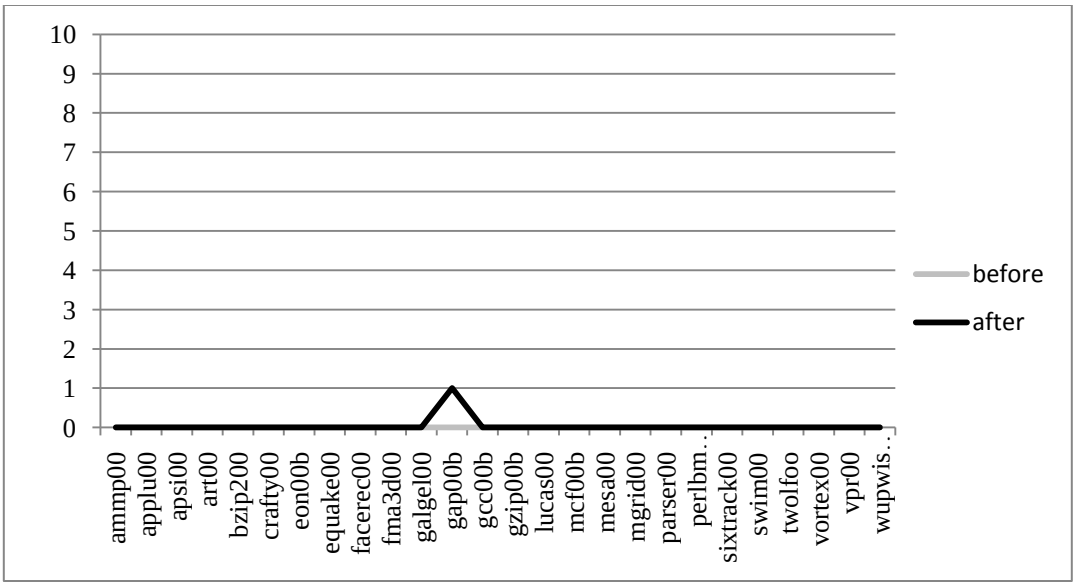
Σχήμα 7.13: Pratio = 75%-79% before and after permutation

Πιο πάνω (Σχήμα 7.13) βλέπουμε ότι ο αριθμός των στηλών που έχουν 75%-79% των ίδιων τιμών μετά τις μεταθέσεις (permutations) στα περισσότερα benchmarks έχει αυξηθεί. Επίσης παρατηρούμε ότι πριν αυτά τα ποσοστά δεν υπήρχαν στα περισσότερα benchmarks. Μετά τις μεταθέσεις (permutations) βλέπουμε ότι σχεδόν σε όλα τα benchmarks έχουμε στήλες με αυτό το ποσοστό των ίδιων τιμών.



Σχήμα 7.14: Pratio = 80%-84% before and after permutation

Πιο πάνω (Σχήμα 7.14) βλέπουμε ότι μετά τις μεταθέσεις (permutations) έχουν εμφανιστεί στήλες σε κάποια από τα benchmarks με ποσοστό 80%-84%. Αρχικά δεν υπήρχαν στήλες με τέτοιο ποσοστό.

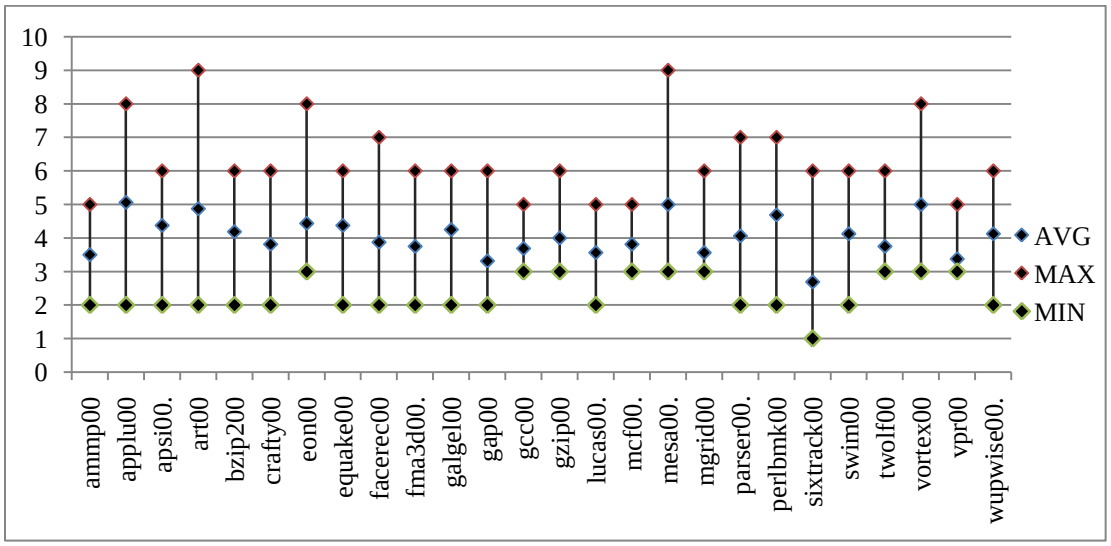


. Σχήμα 7.15: Pratio = 80%-84% before and after permutation

Πιο πάνω (Σχήμα 7.15) βλέπουμε ότι μετά τις μεταθέσεις (permutations) έχει εμφανιστεί μια στήλη με ποσοστό 85%-89% οπού αρχικά δεν υπήρχε μόνο σε ένα από τα 26 benchmarks στο gap00.

Στο πιο κάτω σχήμα παρουσιάζεται ο average αριθμός των μεταθέσεων (permutations) που έγινε σε κάθε benchmark και επίσης ο μεγαλύτερος και ο μικρότερος αριθμός των μεταθέσεων (permutations). Όπως έχουμε αναφέρει και στην μεθοδολογία μας έχουμε ομαδοποίηση λογικά τις εντολές που βρίσκονται στην ίδια θέση μέσα στο block αλλά σε διαφορετικά blocks (πχ οι εντολές που βρίσκονται στην θέση 0 του block αλλά σε διαφορετικά blocks αποτελούν την πρώτη ομάδα). Για την κάθε ομάδα εντολών ξέρουμε τις μεταθέσεις (permutations) που εκτελούνται σε επίπεδο εντολών και στο τέλος ξέρουμε των αριθμό των μεταθέσεων (permutations) που χρειάστηκε να γίνει έτσι ώστε να φτάσουμε σε μία καλύτερη λύση δηλαδή έτσι ώστε να έχουμε το μεγαλύτερο benefit function για αυτή την ομάδα. Το MAX καθορίζεται από την ομάδα εντολών που κάνει το μεγαλύτερο αριθμό των μεταθέσεων (permutations) για να φτάσει στην καλύτερη λύση. Το MIN καθορίζεται από την ομάδα εντολών που κάνει το μικρότερο αριθμό των μεταθέσεων (permutations) για να φτάσει στην καλύτερη λύση, και το average είναι το άθροισμα των μεταθέσεων (permutations) που χρειάστηκε για την κάθε

ομάδα εντολών / 16. Ο αριθμός των ομάδων είναι 16 για το λόγο ότι σε κάθε block έχουμε 16 θέσεις που μπορούν να τοποθετηθούν οι εντολές .

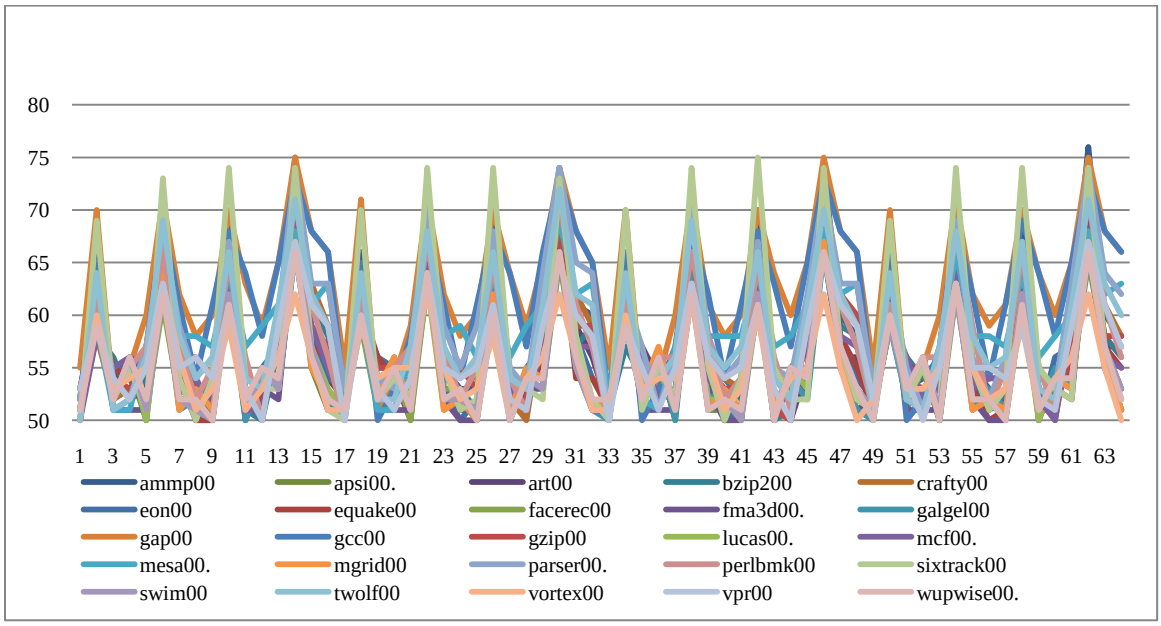


Σχήμα 7.16: AVG. MAX. MIN. Parity Ratios After Permutations

Από πιο πάνω (Σχήμα 7.16) παρατηρούμε ότι το average αριθμός των μεταθέσεων (permutations) που χρειάστηκε να γίνει για να έχουμε το καλύτερο PRatio βάση του benefit Function (μεγαλύτερο ποσοστό των ίδιων τιμών σε parity bit position) διακυμαίνεται μεταξύ 2.8 και 5 για όλα τα benchmark. Παρατηρούμε επίσης ότι ο μεγαλύτερος αριθμός των μεταθέσεων (permutations) που χρειάστηκε να γίνει για να έχουμε την καλύτερη λύση (στήλες με μεγάλο ποσοστό των ίδιων τιμών) σε τουλάχιστον μια από τις 16 ομάδες εντολών, είναι 9 (στο art, mesa) . Ο μικρότερος αριθμός των μεταθέσεων (permutations) για να βρούμε την καλύτερη λύση είναι 1(sixtrack). Αυτά τα στοιχειά είναι σημαντικό να αξιολογηθούν για το λόγο ότι μας δίνουν κάποια σημαντικά στοιχεία για την πολυπλοκότητα του Permutation Unit. Όπως έχουμε αναφέρει πιο πριν το permutation unit μπορεί να είναι προγραμματιζόμενο ή σταθερό. Από πιο πάνω βλέπουμε ότι ο συνολικός αριθμός των μεταθέσεων (permutations) για το κάθε benchmark δεν είναι ο ίδιος και επίσης για την κάθε ομάδα εντολών, ανάλογα με την θέση που έχουν στο block ,οι μεταθέσεις (permutations) που γίνονται σε διαφορετικά

benchmarks παρατηρήσαμε ότι δεν είναι οι ίδιες. Από αυτό συμπεραίνουμε ότι το Permutation Unit μπορεί να είναι προγραμματιζόμενο ανάλογα με την εφαρμογή.

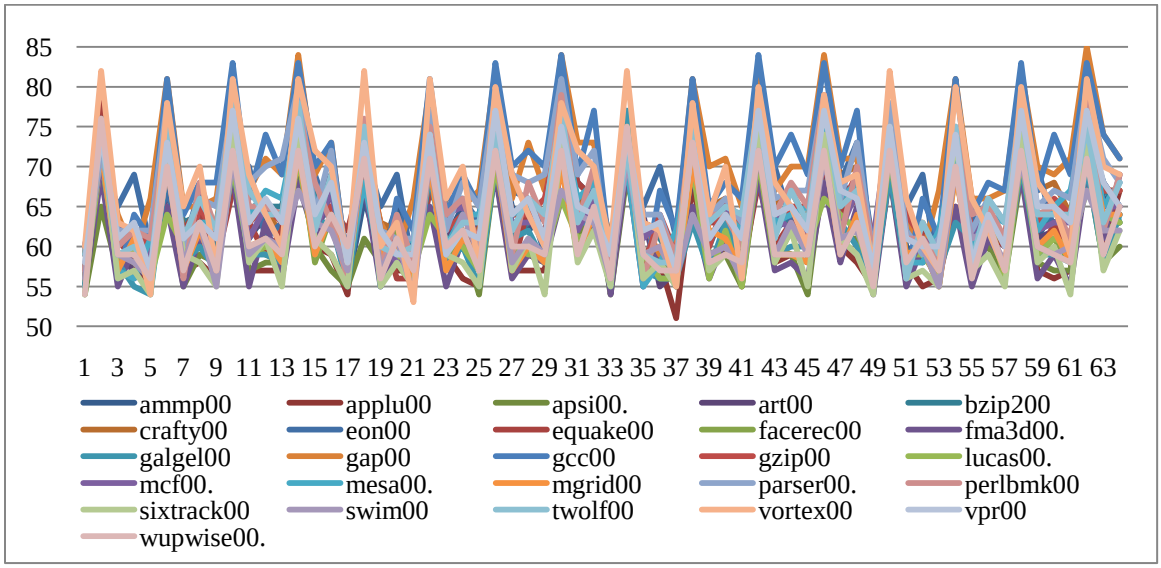
Στο πιο κάτω σχήμα παρουσιάζεται για κάθε μια από τις 64 θέσεις (x άξονας) των parity bits το ποσοστό των ίδιων τιμών (y άξονας) για το κάθε benchmark πριν τις μεταθέσεις (permutations).



Σχήμα 7.17: Pratios for each parity position Before Permutations

Σημαντική παρατήρηση στο Σχήμα 7.17 είναι ότι σχεδόν σε όλα τα benchmarks οι αυξομειώσεις των ποσοστών των ίδιων τιμών γίνονται στα ίδια parity bit position. Παρατηρούμε επίσης ότι στις θέσεις 2, 6, 10,14 ..... 62 έχουμε τα μεγαλύτερα ποσοστά. Στο έκτο κεφάλαιο είχαμε δείξει πως υπάρχει μεγάλη πιθανότητα το ανώτερο byte των εντολών που έχουν immediate πεδίο να είναι 0. Τα πιο πάνω parity bit positions ελέγχουν το δεύτερο byte των εντολών ανάλογα με την θέση τους μέσα στο block. Για παράδειγμα το parity bit στην θέση 2 ελέγχει το δεύτερο byte των εντολών που βρίσκονται στην θέση 0 του block. Το parity bit στην θέση 6 ελέγχει το δεύτερο byte των εντολών που βρίσκονται στην θέση 1 του block. Το parity bit στην θέση 10 ελέγχει το δεύτερο byte των εντολών που βρίσκονται στην θέση 2 του block. Έχοντας υπόψη το baseline σχήμα που έχουμε παρουσιάζει στο έκτο κεφάλαιο η θέση του δεύτερου byte ανάλογα με την

θέση της εντολής μέσα στο byte είναι: θέση δευτέρου = 4x + 2 οπού το x είναι η διεύθυνση της εντολής μέσα στο block από 0 έως 15. Αν αντικατασταθεί η τιμή του x με τις θέσεις 0 έως 15 παίρνουμε τα parity bit positions που στο πιο πάνω σχήμα έχουν τις μεγαλύτερες συχνότητες. Συμπεραίνουμε ότι ένας πιθανός λόγος που σε αυτά τα parity bit positions έχουμε μεγάλη πιθανότητα συχνών τιμών είναι ότι το δεύτερο byte της εντολής είναι 0 και το parity που ελέγχει αυτό το byte είναι 0. Επίσης όσο πιο συχνές είναι αυτές οι εντολές τόσα περισσότερα parity bit με τιμή 0 έχουμε, το οποίο αυξάνει το ποσοστό των ίδιων τιμών για αυτές τις θέσεις



Σχήμα 7.18: Pratios for each parity position After Permutations

Πιο πάνω (Σχήμα 7.18) βλέπουμε τα ίδια χαρακτηρισητικα που παρατηρήσαμε και στο προηγουμενο σχήμα. Η διαφορά εδώ είναι ότι το μεγαλύτερο ποσοστό το πιο συχνών τιμών μετά τις μεταθέσεις (permutations) έχει αυξηθεί απο 76% που ήταν πριν στα 85% .

Κεφάλαιο 8

Συμπεράσματα και Μελλοντική Εργασία

8.1 Συμπεράσματα

Αυτό το κεφάλαιο ολοκληρώνει την παρουσίαση της ΑΔΕ και αναφέρει τα συμπεράσματα στα οποία έχουμε καταλήξει στην μελέτη αυτή. Στην συνέχεια αναφέρεται μελλοντική εργασία που μπορεί να γίνει για να πετύχουμε να έχουμε 100 % ίδιες τιμές σε τουλάχιστον μια στήλη που ελέγχεται από κάποιο parity bit.

Σε αυτή την μελέτη προτείναμε μια ανάλυση των τιμών στα parity bit positions και μία μέθοδο η οποία μας βοηθάει να αυξήσουμε το ποσοστό των ίδιων τιμών. Κίνητρο για αυτή την ανάλυση ήταν το γεγονός ότι ένα μεγάλο ποσοστό του χώρου στην κρυφή μνήμη και της ενέργειας καταναλώνεται από τα bit προστασίας που χρησιμοποιούμε να ανιχνεύουμε soft errors. Ο σκοπός μας είναι να παρέχουμε την ίδια προστασία των δεδομένων αλλά με λιγότερα bits. Για να πετύχουμε αυτό έπρεπε να βρούμε τουλάχιστον μια στήλη που αντιστοιχεί σε κάποιο parity bit position η οποία έχει 100 % ίδιες τιμές. Με αυτό τον τρόπο αυτό το parity bit θα ήταν περιττό να αποθηκευτεί στην κρυφή μνήμη αφού θα ξέραμε ποία ήταν η τιμή του. Σε αυτή την μελέτη προτάθηκε η μέθοδος των μεταθέσεων, Permutations, η οποία αλλάζοντας τα bits μέσα στην ίδια την εντολή και χρησιμοποιώντας κάποια συνάρτηση για να ωφελήσει τις στήλες με μεγαλύτερο ποσοστό των ίδιων τιμών, έχει ως σκοπό να βοηθήσει να πετύχουμε parity στήλες με ποσοστό των ίδιων τιμών 100%. Κατά την ανάλυση αυτή έγιναν offline μετρήσεις στο κώδικα των 26 Spec2000 benchmarks[11], για τις τιμές που είχε το κάθε parity bit position. Μετά την χρήση της μεθόδου μας παρατηρήσαμε ότι αυξήθηκε περίπου 10% το πιο μεγάλο ποσοστό των ίδιων τιμών που υπήρχε πριν για το κάθε parity bit position, αλλά δεν έφτασε στα 100%. Όμως από τις παρατηρήσεις που έγιναν στα αποτελέσματα είδαμε κάποια στοιχειά τα οποία ήταν ενδιαφέρον όπως για παράδειγμα ότι ο αριθμός των permutations που γίνονται είναι διαφορετικά ανά benchmark και δεν είναι πολύ μεγάλος το οποίο οδηγεί το permutation unit που προσθέσαμε στο baseline σχήμα της κρυφής μνήμης να είναι προγραμματιζόμενο. Επίσης παρατηρήσαμε ότι μεγάλα ποσοστά των ίδιων τιμών είχαν τα parity bit position τα οποία υπολογίζονται στο δεύτερο byte της

εντολής. Αυτό το γεγονός έχει να κάνει με ιδιότητες των RISC εντολών οπού συχνά το ανώτερο byte μιας εντολής τύπο immediate είναι 0. Επίσης ένα ενδιαφέρον στοιχειό ήταν ότι αναμέναμε τα benchmarks με μεγαλύτερο μέγεθος να έχουμε περισσότερες parity στήλες με χαμηλό ποσοστό των ίδιων. Όμως από τις παρατηρήσεις αυτό δεν ίσχυε και μπορεί ένας λόγος να είναι ότι υπάρχει duplicate κώδικας το οποίο οδηγεί να έχουμε τις ίδιες εντολές περισσότερα από μια φόρα στον κώδικα.

8.2 Μελλοντική εργασία

Μελλοντική εργασία που θα μπορούσε να γίνει στο θέμα αυτό είναι η ανάλυση σε κρυφές μνήμες δεδομένων για να βρούμε parity στήλες που να έχουν την ίδια τιμή.
Θα μπορούσε επίσης να αξιολογηθεί το ποσοστό των ίδιο τιμών στην ίδια parity στήλη στην υπάρχουσα μελέτη χρησιμοποιώντας μια άλλη συνάρτηση αξιολόγησης που θα οφείλεται σε parity στήλες με μεγάλο ποσοστό των ίδιο τιμών και ίσως σε στήλες που να έχουν 100% ίδιες τιμές , χρησιμοποιώντας την ίδια μέθοδο των μεταθέσεων (Permutations).
Μια άλλη ανάλυση που θα μπορούσε να γίνει είναι να μελετηθούν τα ποσοστά των ίδιων τιμών σε parity στήλη κάνοντας τις μεταθέσεις όχι μόνο μέσα στην ίδια εντολή αλλά και μεταξύ των εντολών που βρίσκονται στο ίδιο block. Επίσης θα μπορούσε να μετρήσουμε τα bit ratios σε εντολές που είναι στο ίδιο block και να ομαδοποιήσουμε τα bits που έχουν την μεγαλύτερη συχνότητα των ίδιων τιμών . Αυτό θα ωφελήσει το parity bit να έχει μεγάλο ποσοστό των ίδιων τιμών.

Βιβλιογραφία

[1] Ziegler, J. F., Curtis, H. W., Muhlfeld, H. P., Montrose, C. J., and Chin, B. 1996. IBM experiments in soft fails in computer electronics (1978–1994). IBM J. Res. Dev. 40, 1 (Jan. 1996), 3-18.

[2] Borkar, S., Karnik, T., Narendra, S., Tschanz, J., Keshavarzi, A., and De, V. 2003. Parameter variations and impact on circuits and micro architecture. In Proceedings of the 40th Annual Design Automation Conference (Anaheim, CA, USA, June 02 - 06, 2003). DAC '03. ACM, New York, NY, 338-342.

[3] Srinivasan, J., Adve, S. V., Bose, P., and Rivers, J. A. 2005. Lifetime Reliability: Toward an Architectural Solution. IEEE Micro 25, 3 (May. 2005), 70-80.

[4] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann, May 2002

[5] Ramming K **W.** Error detecting and error correcting codes.Bell Syst. Tech. J. 29:147-60, 1950. Bell Telephone Laboratories, Murray Hill, NJ

[6] Lee, H., Cho, S., and Childers, B. R. 2007. Performance of Graceful Degradation for Cache Faults. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (March 09 - 11, 2007). ISVLSI. IEEE Computer Society, Washington, DC, 409-415.

[7] NN Sadler, DJ Sorin 2006. Choosing an error protection scheme for a microprocessor's L1 data cache. ICCD October 2006

[8] Yoon, D. and Erez, M. 2009. Memory mapped ECC: low-cost error protection for last level caches. SIGARCH Comput. Archit. News 37, 3 (Jun. 2009), 116-127.

Ηλεκτρονικές Πηγές

[9] Mips instruction format. Web Resource at
http://www.d.umn.edu/~gshute/spimsal/talref.html
[10].Alpha architecture instruction format. Web resource at
http://serdis.dis.ulpgc.es/~ii-ac/Alpha.pdf
[11]. http://www.spec.org/cpu2000/CINT2000/index.html
[12] http://people.ee.duke.edu/~sorin/prior-courses/ece254-fall2008/lectures/2.1-faults.pdf