ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ
MPEG-2 ΑΠΟΚΩΔΙΚΟΠΟΙΗΤΗ ΣΕ ΥΠΟΛΟΓΙΣΤΙΚΟ ΠΕΡΙΒΑΛΛΟΝ ARM CORTEX A8

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΚΟΥΡΑΣ ΔΗΜΗΤΡΙΟΣ

Υπεύθυνος Καθηγητής **: Κατσαβουνίδης Ιωάννης**
Επιβλέπων Καθηγητής : **Μπέλλας Νικόλαος**

ΒΟΛΟΣ 2010

ΠΕΡΙΕΧΟΜΕΝΑ

1. Εισαγωγή 5
2. Θεωρητικό Υπόβαθρο 6
2.1 MPEG-2 Video Standard 7
2.1.1 Γενικά Χαρακτηριστικά του MPEG-2 Βίντεο 7
2.1.2 Η Διαδικασία Κωδικοποίησης του MPEG-2 10
2.1.3 Το Πρότυπο Αποκωδικοποίησης MPEG-2 11
2.1.4 Motion Estimation και Motion Compensation 12
2.1.5 Discrete Cosine Transform και Inverse Transform 13
2.1.6 Quantization και Inverse Quantization 15
2.1.7 ZigZag Scan και Inverse ZigZag Scan 17
2.1.8 Run Length Encoding - Decoding και Huffman 18
2.2 ARM Cortex A8 19
2.2.1 Αρχιτεκτονική ARM 19
2.2.2 Επεξεργαστής Cortex A8 20
3. Μέθοδοι Βελτιστοποίησης Λογισμικού 22
3.1Βελτιστοποίηση Πηγαίου Κώδικα C 23
3.4 Compiler Optimization Options 26
4. Υλοποίηση - Βελτιστοποίηση Λογισμικού 29
4.1 Real View Development Suite v4.0 (RVDS v4.0) 30
4.2 Δημιουργία Αρχείου Εισόδου (Input Video) 30
4.3 Profiling Κώδικα Αναφοράς 31
4.4 Saturate, Mismatch Control και Fast IDCT 32
4.4.1 Ανάλυση Ιδιοτήτων των Βίντεο για Βελτιστοποίηση 32
4.4.2 Saturate and Mismatch Control 34
4.4.3 Fast Inverse Discrete Cosine Transform (FIDCT) 36
4.4.3.1 Ελαχιστοποίηση της χρήσης FIDCT 36
4.4.3.2 Περιγραφή και Υλοποίηση Βελτιωμένου Αλγορίθμου FIDCT 37
4.4.3.3 Χρήση NEON instructions 40

4.4.4 Μέτρηση Απόδοσης - Νέο Profiling 44
4.5 Motion Compensation 47
4.5.1 Χρήση NEON instructions 47
4.5.2 Μέτρηση Απόδοσης – Νέο Profiling 49
4.6 Adding Prediction and Coefficient Data 51
4.6.1 Βελτιώσεις σε Επίπεδο Γλώσσας C 51
4.6.2 Χρήση NEON instructions 52
4.6.3 Μέτρηση Απόδοσης – Νέο Profiling 52
4.7 Clear Block 55
4.7.1 Χρήση NEON instructions 55
4.7.2 Μέτρηση Απόδοσης – Νέο Profiling 56
4.8 Buffer and Bit Functions 57
4.8.1 Δημιουργία Μεγαλύτερου Buffer 58
4.8.2 Ελαχιστοποίηση των Προσπελάσεων Μνήμης 59
4.8.3 Υπόλοιπες Βελτιστοποιήσεις 59
4.8.4 Μέτρηση Απόδοσης – Νέο Profiling 60
4.9 Επιπλέον Βελτιώσεις 63
4.9.1 Αλλαγές – Βελτιώσεις 63
4.9.2 Μέτρηση Απόδοσης – Νέο Profiling 65
4.10 Compiler Optimization Options 66
5. Συμπεράσματα και Προτάσεις για Μελλοντική Έρευνα 67
Αναφορές - Βιβλιογραφία 70

Ευχαριστίες,

Φτάνοντας στο τέλος της παρούσας πτυχιακής εργασίας, που σηματοδοτεί το τέλος των προπτυχιακών μου σπουδών αισθάνομαι την ανάγκη να ευχαριστήσω όλους τους ανθρώπους που ήταν κοντά μου όλα αυτά τα χρόνια.

Την Οικογένεια μου για την υλική και ψυχολογική ενίσχυση που μου προσέφερε , αλλά περισσότερο για την εμπιστοσύνη τους στο πρόσωπό μου.

Τον υπεύθυνο καθηγητή της πτυχιακής μου εργασίας Ιωάννη Κατσαβουνίδη που με βοήθησε με κάθε τρόπο να ανταπεξέλθω στις ανάγκες της παρούσας εργασίας και με μύησε στον κόσμο της ψηφιακής εικόνας και του βίντεο.

Τον επιβλέποντα καθηγητή Νικόλαο Μπέλλα, που ήταν ο πρώτος που μου ανέθεσε εργασία σε ενσωματωμένα συστήματα.

Τους φίλους και συμφοιτητές μου που ήταν κοντά μου όλα αυτά τα χρόνια.

Δημήτρης

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

Στα πλαίσια της παρούσας εργασίας πραγματοποιήθηκε η Βελτιστοποίηση του MPEG-2 αποκωδικοποιητή σε αρχιτεκτονική ARM Cortex A8. Ο στόχος που τέθηκε ήταν η αναπαραγωγή MPEG-2 βίντεο ανάλυσης SD (Standard Definition) σε πραγματικό χρόνο. Για την εργασία πάνω σε αυτό το πρόβλημα αποκτήθηκαν γνώσεις πάνω στις έννοιες της κωδικοποίησης και της αποκωδικοποίησης βίντεο και πιο συγκεκριμένα στο standard συμπίεσης MPEG-2. Επίσης ήταν απαραίτητη η μελέτη της αρχιτεκτονικής ARM Cortex A8 και η εξοικείωση πάνω στον προγραμματισμό στην συγκεκριμένη αρχιτεκτονική. Απαραίτητο στοιχείο επίσης για την ανάπτυξη Βελτιστοποιημένου κώδικα αποτελεί και η μελέτη μεθόδων Βελτιστοποίησης Λογισμικού. Όλα τα παραπάνω στοιχεία μαζί με της βελτιστοποιήσεις που πραγματοποιήθηκαν πάνω στον αποκωδικοποιητή, παρουσιάζονται στην παρούσα εργασία. Τέλος αναλύονται τα αποτελέσματα και γίνονται προτάσεις για μελλοντική έρευνα.

ΚΕΦΑΛΑΙΟ 2

Θεωρητικό Υπόβαθρο

Στο κεφάλαιο αυτό καταγράφεται το θεωρητικό υπόβαθρο που αποκτήθηκε για την υλοποίηση της παρούσας εργασίας. Αρχικά γίνεται μια σύντομη παρουσίαση του Standard αποκωδικοποίησης MPEG-2. Αναφέρονται οι εφαρμογές του και ιστορικά στοιχεία για την ανάπτυξη του. Στη συνέχεια αναλύονται οι διαδικασίες της κωδικοποίησης και της αποκωδικοποίησης ενώ περιγράφονται τα στάδια και των δύο διαδικασιών. Στην επόμενη ενότητα παρουσιάζεται η αρχιτεκτονική ARM και ειδικότερα ο επεξεργαστής Cortex A8. Καταγράφονται τα κύρια γνωρίσματα της αρχιτεκτονικής ARM και του Cortex A8. Επίσης αναφέρονται οι εφαρμογές του συγκεκριμένου επεξεργαστή και ορισμένες ενδεικτικές συσκευές που τον ενσωματώνουν.

2.1 MPEG-2 Video Standard

Τα αρχικά **MPEG** προέρχονται από τις λέξεις **Moving Picture Experts Group** (Ομάδα Ειδικών στην Κινούμενη Εικόνα) . Πρόκειται για μία επιτροπή που δρα στα πλαίσια του Διεθνούς Οργανισμού τυποποίησης. Επίσημα είναι γνωστή σαν ISO/IEC JTC1/SC29/WG11. Ιδρύθηκε το 1988 και είναι μέλος του **JTC1** (Joint ISO/IEC Technical Committee on Information Technology - Ενωμένη Τεχνική επιτροπή ISO/IEC στην Τεχνολογία της Πληροφορικής) .

Το MPEG-2 αποτελεί ένα standard για "*Generic coding of moving pictures and associated audio information*" και αναπτύχθηκε από την ISO/IEC JTC1/SC29/WG11 στις αρχές της δεκαετίας του 1990. Εγκρίθηκε σαν πρότυπο (standard) και δημοσιεύτηκε το 1995 με όνομα **ISO/IEC 13818**. Το MPEG2 δημοσιεύτηκε σε τέσσερα τμήματα (parts):

- Part I Systems: Περιγράφει το συγχρονισμό και τη σύνθεση του βίντεο και του ήχου.
- Part II Video: Περιγράφει την κωδικοποιημένη αναπαράσταση του βίντεο και τη διαδικασία αποκωδικοποίησης.
- Part III Audio: Περιγράφει την κωδικοποιημένη αναπαράσταση του ήχου.
- Part IV Conformance Testing: Περιγράφει διαδικασίες για την δοκιμή της συμβατότητας με το standard.

Αναπτύχθηκε για εφαρμογή στην ψηφιακή τηλεόραση. Υποστηρίζει μέγεθος εικόνας που ακολουθεί το τηλεοπτικό πρότυπο CCIR-601 (broadcast quality - ποιότητα εκπομπής) δηλαδή 704x480 pixels (NTSC) ή 704x576 pixels (PAL), HDTV (1920x1080) ποιότητα και εικόνα πλεκτής σάρωσης (interlaced). Ο ρυθμός μετάδοσης κυμαίνεται από 3 ως και πάνω από 15 Mbits/sec.

Οι εφαρμογές του είναι στην καλωδιακή τηλεόραση (**CableTV**), στη δορυφορική **(Direct Broadcasting Satellite TV**), όπως επίσης και στην ψηφιακή τηλεόραση (**DVB, DTV**). Σημαντική επίσης εφαρμογή του είναι και η ψηφιακή αποθήκευση βίντεο (**DVD - Digital Video Disk**).

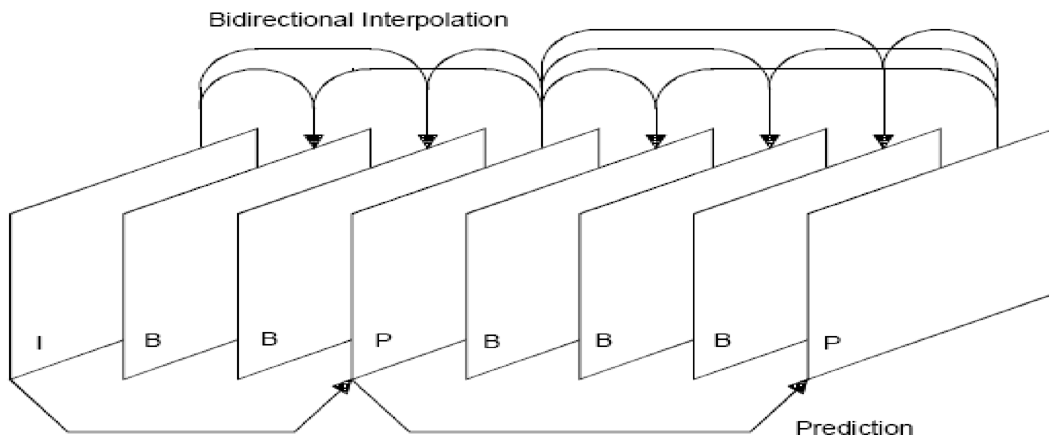
2.1.1 Γενικά Χαρακτηριστικά του MPEG-2 Βίντεο

Το αρχείο βίντεο (video bitstream) που ακολουθεί το πρότυπο MPEG-2 αποτελείται από πέντε επίπεδα : Video Sequence, Group Of Pictures (GOP), Picture, Slice, Macroblock, Block.

Video Sequence: Η υψηλότερη δομή αρχείου βίντεο. Αρχίζει με ένα sequence header, που προαιρετικά μπορεί να ακολουθείτε από ένα group of pictures header και έπειτα από ένα ή περισσότερα κωδικοποιημένα pictures. Η σειρά των κωδικοποιημένων pictures στο κωδικοποιημένο video bitstream είναι η σειρά με την οποία ο κωδικοποιητής τα επεξεργάζεται και όχι απαραίτητα η σειρά με την οποία αναπαράγονται. Το video sequence τερματίζεται με ένα sequence_end_code (ακολουθία bits που δηλώνει το τέλος του video sequence). Σε διάφορα σημεία του video sequence, μία επανάληψη του sequence header ή ένα group of picture header ή και τα δύο μπορούν να προηγούνται ενός κωδικοποιημένου picture.

Group Of Pictures (GOP): Αποτελείται από ένα group of pictures header και μια σειρά από ένα ή περισσότερα κωδικοποιημένα pictures που σκοπός του είναι να παρέχεται η δυνατότητα τυχαίας προσπέλασης τους στο video sequence.
Picture: Η κύρια μονάδα κωδικοποίησης ενός video sequence. Ένα picture αποτελείτε από τρεις ορθογώνιους πίνακες που αντιπροσωπεύουν τη φωτεινότητα (luminance - Y) και τα δύο χρώματα (Cb , Cr ή U , V). Για τύπο χρώματος (chroma format) 4:2:0 οι πίνακες U , V έχουν το μισό μέγεθος από τον πίνακα Y και στις δύο κατευθύνσεις (οριζόντια και κάθετα). Για τύπο χρώματος 4:2:2 οι πίνακες U και V έχουν το ίδιο οριζόντιο και το μισό κάθετο μέγεθος του πίνακα Y. Τέλος για τύπο χρώματος 4:4:4 οι πίνακες U, V έχουν το ίδιο μέγεθος με τον Y και στις δύο κατευθύνσεις. Το MPEG-2 καθορίζει τριών ειδών pictures:

- **Intra Pictures:** Τα Intra Pictures ή I-Pictures, κωδικοποιούνται χρησιμοποιώντας πληροφορίες από το Picture αυτό καθαυτά. Παρέχουν τα σημεία πρόσβασης στο video sequence όπου μπορεί να ξεκινήσει η αποκωδικοποίηση, αλλά κωδικοποιούνται με μέτρια κωδικοποίηση.
- **Predicted Pictures:** Τα Prediction Pictures ή P-Pictures κωδικοποιούνται με αναφορά στα κοντινότερα προηγούμενα I ή P Pictures. Αυτή η τεχνική ονομάζεται forward prediction και παρουσιάζεται στην εικόνα 2.1. Όπως τα I-Pictures έτσι και τα P-Pictures μπορούν να χρησιμοποιηθούν ως Pictures αναφοράς για B-Pictures και μελλοντικά P-Pictures. Επιπλέον, τα P-Pictures χρησιμοποιούν motion compensation κατά τη κωδικοποίηση τους, τεχνική που επιτυγχάνει μεγαλύτερη συμπίεση.
- **Bidirectional Pictures:** Τα Bidirectional Pictures ή B-Pictures είναι Pictures που χρησιμοποιούν και ένα προηγούμενο και ένα μελλοντικό Picture ως αναφορά (εικόνα 2.1). Αυτή η τεχνική καλείται bidirectional prediction. Τα B-Pictures παρέχουν τη μεγαλύτερη συμπίεση, ωστόσο ο χρόνος υπολογισμού τους είναι ο υψηλότερος.



Εικόνα 2.1: Παράδειγμα των τριών ειδών Picture

Slice: Ένα ή περισσότερα συνεχόμενα macroblock. Η σειρά των macroblock σε ένα slice είναι από αριστερά προς τα δεξιά και από πάνω προς τα κάτω. Το slice είναι σημαντικό στο χειρισμό

των λαθών (errors). Αν το video bitstream περιέχει κάποιο λάθος, τότε ο αποκωδικοποιητής μπορεί να αγνοήσει το λανθασμένο slice και να μεταβεί στο επόμενο. Η ύπαρξη περισσοτέρων slice βοηθάει στην απόκρυψη λαθών, ωστόσο τα bits που δεσμεύονται θα μπορούσαν να χρησιμοποιηθούν για να βελτιωθεί η ποιότητα των κωδικοποιημένων picture.

Macroblock: Η βασική μονάδα κωδικοποίησης στον αλγόριθμο MPEG-2. Είναι ένα 16x16 κομμάτι ενός picture. Σε αντιστοιχία με τον ορισμό του picture, το macroblock αποτελείτε από τέσσερα 8x8 luminance block (πίνακας Y) και δύο (για chrominance format 4:2:0), τέσσερα (για chrominance format 4:2:2), ή οχτώ (για chrominance format 4:4:4), που προέρχονται από τα αντίστοιχα τμήματα του chrominance (πίνακες U, V). Το macroblock χρησιμοποιείται για να αναφερθεί και στα δεδομένα του δείγματος (pixel) αλλά και στην κωδικοποιημένη αναπαράσταση τους (π.χ. macroblock DCT συντελεστών).

Block: Η μικρότερη μονάδα κωδικοποίησης στον αλγόριθμο MPEG-2. Έχει μέγεθος 8x8 και μπορεί να περιέχει δεδομένα του δείγματος (pixel) ή 64 DCT συντελεστές (συντελεστές που προέρχονται από το Διακριτό Μετασχηματισμό Συνημίτονου – Discrete Cosine Transform).

Μία επιπλέον τεχνική που παρέχεται από το MPEG-2 είναι το **scalability** (κλιμμάκωση). Το scalability παρέχει τη δυνατότητα να χωριστεί το MPEG-2 βίντεο σε διαφορετικά στρώματα. Το standard υποστηρίζει τέσσερις διαφορετικούς τρόπους για scalability.

Spatial Scalability: Το spatial scalability περιλαμβάνει τη δημιουργία δύο στρωμάτων βίντεο από ένα βίντεο πηγή, έτσι ώστε το χαμηλότερο στρώμα (base layer) να κωδικοποιείται από μόνο του παρέχοντας τη βασική χωρική ευκρίνεια του βίντεο, ενώ το «ενισχυτικό» στρώμα (enhancement layer) χρησιμοποιεί το χαμηλότερο στρώμα και φέρει την πλήρη χωρική ευκρίνεια του βίντεο πηγή.

SNR Scalability: Το SNR scalability περιλαμβάνει τη δημιουργία δύο στρωμάτων βίντεο από ένα βίντεο πηγή, με την ίδια χωρική ευκρίνεια αλλά με διαφορετική ποιότητα βίντεο, έτσι ώστε το χαμηλότερο στρώμα (base layer) να κωδικοποιείται από μόνο του παρέχοντας τη βασική ποιότητα βίντεο και το «ενισχυτικό» στρώμα (enhancement layer) κωδικοποιείται για να ενισχύσει το χαμηλότερο στρώμα.

Temporal Scalability: Το temporal scalability περιλαμβάνει το χωρισμό των frames σε στρώματα. Το χαμηλότερο στρώμα (layer) κωδικοποιείται από μόνο του, με χαμηλότερο frame rate, ενώ τα ενδιάμεσα frames κωδικοποιούνται από το "ενισχυτικό" στρώμα (enhancement layer) χρησιμοποιώντας τα ανακατασκευασμένα frames του base layer για πρόβλεψη.

Data Partitioning: Το data partitioning είναι μία μέθοδος στο πεδίο της συχνότητας που χωρίζει το block των 64 κβαντισμένων DCT συντελεστές σε δύο στρώματα. Το χαμηλότερο στρώμα περιέχει τους πιο κρίσιμους, χαμηλής συχνότητας συντελεστές και δευτερεύοντες πληροφορίες (όπως οι τιμές DC, τα motion vectors). Το ενισχυτικό στρώμα φέρει τις τιμές των υψηλών συχνοτήτων AC.

Μία ακόμη δυνατότητα του MPEG-2 είναι πως υποστηρίζει δύο μεθόδους σάρωσης, την προοδευτική (**progressive**) και την πλεκτή (**interlaced**). Η interlaced σάρωση σαρώνει τις

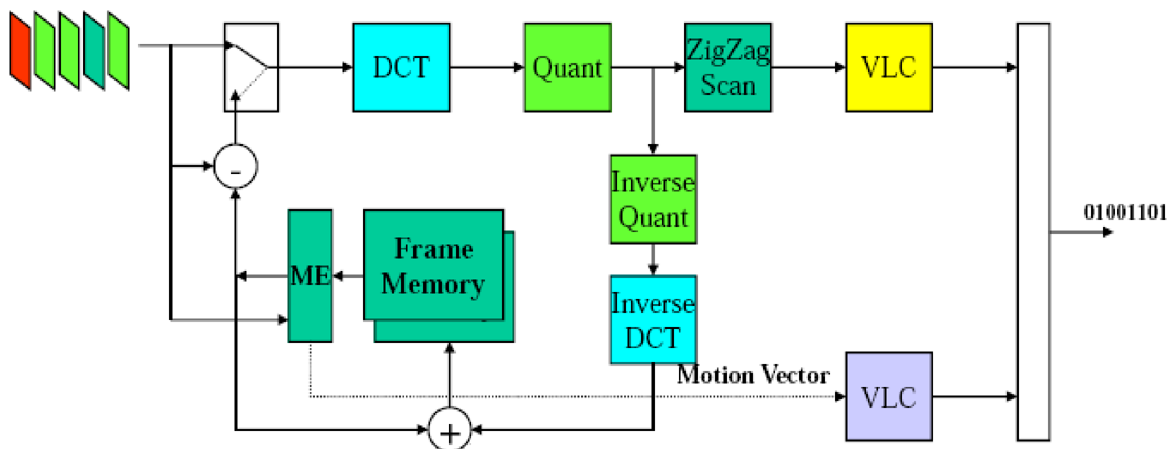
περιττές γραμμές ενός frame σαν ένα πεδίο (odd field) και τις άρτιες γραμμές σαν ένα άλλο πεδίο (even field). Η progressive σάρωση σαρώνει διαδοχικές γραμμές συνεχόμενα.

Ένα interlaced βίντεο χρησιμοποιεί δύο δομές picture: frame structure και field structure. Στη frame structure οι γραμμές των δύο field εναλλάσσονται και κωδικοποιούνται μαζί σαν frame. Ένα picture header χρησιμοποιείται και για τα δύο. Στη field structure τα δύο field ενός frame κωδικοποιούνται ανεξάρτητα το ένα από το άλλο και το odd filed ακολουθείται από το even field. Κάθε field έχει το δικό του picture header.

Από την άλλη κάθε picture στη progressive σάρωση αποτελεί frame picture.

2.1.2 Η Διαδικασία Κωδικοποίησης του MPEG-2

Η διαδικασία κωδικοποίησης του MPEG-2 περιγράφεται σχηματικά από την εικόνα 2.2.



Εικόνα 2.2: Διαδικασία Κωδικοποίησης MPEG-2

Η κωδικοποίηση των P και B picture διαφέρει από αυτή των I picture. Στα P και B picture το κάθε macroblock με τις τιμές pixel, περνάει στον αφαιρετή (-) και στη διαδικασία **motion estimation (ME)**. Το ME βρίσκει το macroblock του picture αναφοράς του ταιριάζει περισσότερο στο macroblock προς αποκωδικοποίηση και υπολογίζει το διάνυσμα κίνησης (motion vector) με μια διαδικασία που περιγράφεται στην παράγραφο 2.1.3 .Στη συνέχεια στέλνει το macroblock του reference picture στον αφαιρέτη ο οποίος υπολογίζει τη διαφορά και τη στέλνει για κωδικοποίηση.

Η διαφορά μετασχηματίζεται από το χωρικό πεδίο στο πεδίο συχνοτήτων μέσω Διακριτού Μετασχηματισμού Συνημίτονου (**DCT**) 2 διαστάσεων (παράγραφος 2.1.5).

Στη συνέχεια οι DCT συντελεστές της διαφοράς κβαντίζονται (**Quant**) ώστε να μειωθεί ο αριθμός των bits για την αναπαράσταση κάθε συντελεστή (παράγραφος 2.1.6). Συνήθως πολλοί συντελεστές κβαντίζονται στο 0.

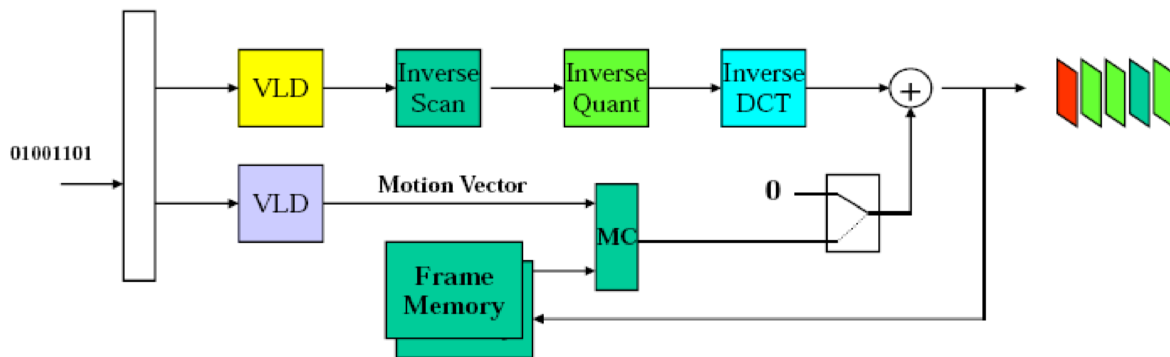
Οι κβαντισμένοι συντελεστές στη συνέχεια περνούν από το στάδιο του **zigzag scanning** (**ZigZag Scan –** παράγραφος 3.1.7), ώστε να προκύψουν ευνοϊκότερες ακολουθίες DCT συντελεστών για το **Run-Length Encoding.** Έπειτα κωδικοποιούνται μέσω **Run-Length Encoding** και **Huffman** (**VLC** - παράγραφος 2.1.7). Διαδικασία που μειώνει ακόμα περισσότερο των αριθμό των bits για κάθε συντελεστή. Σε αυτή τη διαδικασία οι DCT συντελεστές συνδυάζονται με τα motion vector και με άλλα απαραίτητα δεδομένα.

Στην περίπτωση των P-picture, επειδή αυτά χρησιμοποιούνται σαν picture αναφοράς από B και P picture, οι κβαντισμένοι DCT συντελεστές στέλνονται σε μια διαδικασία εσωτερικής αποκωδικοποίησης. Η διαφορά περνάει από τα στάδια αντίστροφης κβαντοποίησης (inverse quantize) και του αντίστροφου μετασχηματισμού (Inverse DCT). Το macroblock του reference picture διαβάζεται από τη μνήμη, προστίθεται με τη διαφορά (residual) και αποθηκεύεται πίσω στη μνήμη για να χρησιμοποιηθεί σαν αναφορά για επόμενα picture. Ο σκοπός είναι τα δεδομένα του reference picture που χρησιμοποιείται στον κωδικοποιητή να ταιριάζουν με τα δεδομένα του reference picture του αποκωδικοποιητή. Τα B-picture δεν χρησιμοποιούνται σαν picture αναφοράς.

Η κωδικοποίηση των I-picture ακολουθεί την ίδια διαδικασία, χωρίς το motion compensation ενώ ο αφαιρέτης (-) λαμβάνει την τιμή 0 αντί των διαφορών του macroblock. Σε αυτή την περίπτωση οι DCT συντελεστές αναπαριστούν μετασχηματισμένες τιμές pixel και όχι διαφορών. Όπως και στην περίπτωση των P-picture και τα αποκωδικοποιημένα I-picture αποθηκεύονται στη μνήμη για να χρησιμοποιηθούν σαν picture αναφοράς.

2.1.3 Το Πρότυπο Αποκωδικοποίησης MPEG-2

Η διαδικασία αποκωδικοποίησης του MPEG-2 παρουσιάζεται σχηματικά *από την εικόνα 2.3.*



Εικόνα 2.3: Διαδικασία Αποκωδικοποίησης MPEG-2

Η διαδικασία της αποκωδικοποίησης αποτελεί την αντίστροφη διαδικασία της κωδικοποίησης.

Τα κωδικοποιημένα δεδομένα που λαμβάνονται περνούν από το στάδιο **Run – Length Decoding** και **Huffman** (**VLD** – παράγραφος 2.1.8). Τη διαδικασία VLD ακολουθεί το στάδιο του **inverse zigzag scanning** (**inverse Scan -** παράγραφος 2.1.7). Στη συνέχεια οι κβαντισμένοι DCT συντελεστές περνούν από το στάδιο αντίστροφου κβαντισμού (**inverse quant**) και έπειτα στον αντίστροφο μετασχηματισμό DCT (**Inverse DCT** – παράγραφος 2.1.5) που μετασχηματίζει τα δεδομένα πίσω στο χωρικό πεδίο. Τα motion vector έρχονται από το VLD και περνούν στη διαδικασία **motion compensation** (**MC** - παράγραφος 2.1.4). Για τα B και P picture, τα δεδομένα των motion vector μεταφράζονται σε διεύθυνση μνήμης από το motion compensation, ώστε να διαβαστεί ένα συγκεκριμένο macroblock από προηγούμενο αποθηκευμένο reference picture. Ο αθροιστής (+) προσθέτει τις τιμές αυτού του macroblock με αυτές της διαφοράς, ώστε να επιτευχθεί η ανάκτηση των δεδομένων του picture. Για τα I picture, δεν υπάρχουν motion vector ή reference picture, έτσι το macroblock πρόβλεψης θεωρείται μηδέν. Για τα I και P picture, το αποτέλεσμα του αθροιστή αποθηκεύεται πίσω στη μνήμη ώστε να χρησιμοποιηθεί σαν picture πρόβλεψης από μελλοντικά picture.

2.1.4 Motion Estimation και Motion Compensation

Το **motion estimation** και το **motion compensation** αποτελούν ουσιαστικά δύο αντίστροφες διαδικασίες. Το motion estimation χρησιμοποιείται κατά την κωδικοποίηση ενός βίντεο MPEG-2 ενώ το motion compensation κατά την αποκωδικοποίηση ενός τέτοιου βίντεο.

Στον κωδικοποιητή ο motion estimator συγκρίνει κάθε ένα από τα macroblock προς κωδικοποίηση με macroblock σε προηγούμενα picture ή pictures αναφοράς (reference), τα

οποία είναι αποθηκευμένα στη μνήμη. Βρίσκει την περιοχή (διαστάσεων macroblock) του **reference picture** που ταιριάζει περισσότερο με το macroblock προς κωδικοποίηση. Για το ταίριασμα αυτό χρησιμοποιείται το Sum of Absolute Differences (**SAD**). Στη συνέχεια ο motion estimator υπολογίζει ένα διάνυσμα κίνησης (**motion vector**) που αντιπροσωπεύει την κάθετη και την οριζόντια μετατόπιση του macroblock προς κωδικοποίηση, ώστε να συμπίπτει με την περιοχή (διαστάσεων macroblock) του reference picture που του ταιριάζει περισσότερο. Δηλαδή υπολογίζεται η διαφορά των οριζοντίων και καθέτων συντεταγμένων. Πρέπει να σημειωθεί πως τα motion vector επιτυγχάνουν πρόβλεψη ½ pixel με γραμμική παρεμβολή των γειτονικών pixel. Ο motion estimator διαβάζει αυτό το matching macroblock (**predicted macroblock**) από το reference picture και το στέλνει στον αφαιρετή, ο οποίος το αφαιρεί pixel προς pixel από το macroblock προς κωδικοποίηση.

Σύμφωνα με τα είδη των picture που παρουσιάζονται στην παράγραφο 2.1.1, για B picture προκύπτουν δυο motion vector για κάθε macroblock ενώ για P picture ένα. Η διαδικασία αυτή πραγματοποιείται στο πρώτο στάδιο κωδικοποίησης των P και B picture.

Η αντίστροφη διαδικασία κατά την αποκωδικοποίηση είναι το **motion compensation**. Πριν από αυτή τη διαδικασία, τα motion vector λαμβάνονται από το video bitstream και αποκωδικοποιούνται με τη χρήση των Run – Length Decoding και Huffman. Στη συνέχεια περνούν στο στάδιο του motion compensation. Εκεί, από το reference picture που έχει ήδη αποκωδικοποιηθεί και βρίσκεται στη μνήμη, υπολογίζεται μέσω γραμμικής παρεμβολής (αν είναι απαραίτητο) η περιοχή διαστάσεων macroblock reference picture. Αυτή η περιοχή (που ονομάζεται **prediction macroblock**) στη συνέχεια θα περάσει στον αθροιστή, ώστε να προστεθεί με τα δεδομένα που προέκυψαν από την αποκωδικοποίηση των κωδικοποιημένων διαφορών (residual).

Ο αθροιστής αποτελεί το τελευταίο στάδιο της αποκωδικοποίησης. Σε αυτόν γίνεται η πρόσθεση των δεδομένων αποκωδικοποίησης με τα δεδομένα της πρόβλεψης (**adding prediction and coefficient data**). Σε αυτό το στάδιο πραγματοποιείται και το τελικό saturate ώστε οι τιμές των pixel που προκύπτουν να βρίσκονται στο πεδίο [0,255] (8bits). Στη συνέχεια τα αποκωδικοποιημένα picture είναι έτοιμα για αναπαραγωγή με τη σειρά ωστόσο αναπαραγωγής και όχι με τη σειρά επεξεργασίας τους.

Στις περιπτώσεις των I και P picture, τα picture αποθηκεύονται πίσω στη μνήμη ώστε να χρησιμοποιηθούν σαν picture πρόβλεψης από μελλοντικά P και B picture. Τα B picture δεν χρησιμοποιούνται σαν picture πρόβλεψης και δεν αποθηκεύονται στη μνήμη.

2.1.5 Discrete Cosine Transform και Inverse Transform

Ο Διακριτός Μετασχηματισμός Συνημίτονου (**Discrete Cosine Transform**) είναι μία μέθοδος που βρίσκει μεγάλη εφαρμογή στην ψηφιακή συμπίεση γενικά, αλλά και στο MPEG

ειδικότερα. Με το μετασχηματισμό DCT μπορούμε να μεταφέρουμε την πληροφορία που περικλείει η εικόνα από το πεδίο του χώρου στο πεδίο της συχνότητας (αφηρημένο πεδίο), όπου η περιγραφή της μπορεί να γίνει τελικά με σημαντικά μικρότερο πλήθος bits. Βασικό χαρακτηριστικό του DCT (και πολλών άλλων ορθογώνιων μετασχηματισμών) είναι η συγκέντρωση της ενέργειας του σήματος σε ένα σχετικά μικρό πλήθος συντελεστών του μετασχηματισμού (στις χαμηλές συχνότητες).

Στο MPEG-2 εφαρμόζεται 2 διαστάσεων DCT 8 σημείων σε κάθε block 8x8. Πρώτα εφαρμόζεται μονοδιάστατος DCT στις γραμμές του block και στη συνέχεια μονοδιάστατος DCT στις στήλες. Ο μονοδιάστατος Διακριτός Μετασχηματισμός Συνημίτονου 8 σημείων ορίζεται από το μαθηματικό τύπο:

$$t_u = \frac{C_u}{2}\sum_{x=0}^{7} s_x \cos\frac{(2x+1)\pi u}{16}, u = 0..7$$

$$C_0 = \frac{1}{\sqrt{2}},\ C_n = 1 \text{ for n} = 1..7$$

Στη διαδικασία της κωδικοποίησης το DCT είναι το πρώτο στάδιο κωδικοποίησης αν πρόκειται για κωδικοποίηση I picture ή το δεύτερο στάδιο αν πρόκειται για P ή B picture (προηγείται η διαδικασία motion estimation και ο υπολογισμός των διαφορών). Στα I picture μετασχηματίζονται οι τιμές των pixel κάθε block, ενώ στα P και B τα block των διαφορών που προκύπτουν από την αφαίρεση του macroblock προς κωδικοποίηση και του prediction macroblock.

Στην αποκωδικοποίηση, η αντίστροφη διαδικασία DCT είναι ο Inverse DCT. Ο **Inverse Discrete Cosine Transform** 8 σημείων ορίζεται με από τον μαθηματικό τύπο :

$$s_x = \frac{1}{2}\sum_{u=0}^{7} t_u C_u \cos\frac{(2x+1)\pi u}{16}, u = 0..7$$

$$C_0 = \frac{1}{\sqrt{2}},\ C_n = 1 \text{ for n} = 1..7$$

Εφαρμόζεται και αυτός σε block 8x8 κατά τη μία διάσταση αρχικά (γραμμές) και κατά τη δεύτερη διάσταση στη συνέχεια (στήλες). Στη διαδικασία της αποκωδικοποίησης το IDCT είναι το προτελευταίο στάδιο πριν τον αθροιστή (adding prediction and coefficient data).

Πραγματοποιεί ακριβώς την αντίστροφη διαδικασία από το DCT, δηλαδή μετασχηματίζει τους συντελεστές από το πεδίο της συχνότητας στο χωρικό πεδίο (pixel).

Η πολυπλοκότητα του παραπάνω μετασχηματισμού είναι αρκετά υψηλή ($O(N^2)$). Για αυτό το λόγο σπάνια χρησιμοποιείται ο ορισμός σε πραγματικές εφαρμογές, αλλά χρησιμοποιούνται πιο γρήγοροι και αποδοτικοί αλγόριθμοι όπως περιγράφεται στην παράγραφο 5.2.

2.1.6 Quantization και Inverse Quantization

Η μέθοδος που μας βοηθάει να απαλλαγούμε από σημαντικό μέρος της πληροφορίας είναι η κβαντοποίηση. Με τον όρο κβαντοποίηση γενικά εννοούμε τη μετατροπή ενός σήματος άπειρων (η πάρα πολλών) τιμών σε ένα σήμα ορισμένων διακριτών τιμών π.χ. η κβαντοποίηση μιας εικόνας που περιέχει εκατομμύρια χρώματα οδηγεί σε μία εικόνα που έχει 256 διαφορετικές τιμές για το χρώμα. Με άλλα λόγια κβαντοποίηση είναι ο περιορισμός των bits με τα οποία περιγράφουμε τα δείγματα του σήματος.

Στη διαδικασία της κωδικοποίησης, η κβαντοποίηση εφαρμόζεται στο 8x8 block των DCT συντελεστών που έχει προκύψει από τη διαδικασία DCT. Η περιοχή χαμηλότερης συχνότητας (DC και οι κοντινοί AC συντελεστές) παίρνει την ελάχιστη κβαντοποίηση ενώ η περιοχή υψηλής συχνότητας παίρνει τη μέγιστη κβαντοποίηση. Αυτό συμβαίνει επειδή το μάτι είναι πιο ευαίσθητο στις χαμηλές συχνότητες παρά στις υψηλές και αυτός είν. Δεδομένου ότι η ακρίβεια των DCT συντελεστών θα μειωθεί με την κβαντοποίηση , ο βαθμός και το σχήμα της είναι σχεδιασμένο διαφορετικά για κάθε τύπο picture, ώστε να μην προκαλέσει ορατή ζημιά στο βίντεο που θα προκύψει από την μετέπειτα αποκωδικοποίηση. Έτσι το MPEG-2 επιτρέπει να χρησιμοποιηθούν διαφορετικού βάρους πίνακες, ένας για Intra και ένας για NonIntra, για να ταιριάζουν με τα χαρακτηριστικά της συχνότητας του κάθε είδους picture. Η πράξη που εκτελείται κατά τη κβαντοποίηση είναι η διαίρεση του block με τον αντίστοιχο πίνακα. Μία επιπλέον δυνατότητα που παρέχει το MPEG-2 είναι πως μπορούν να οριστούν νέοι πίνακες κβαντοποίησης από το χρήστη, οι οποίοι χρησιμοποιούνται από τον κωδικοποιητή και περιλαμβάνονται στα headers του video bitstream ώστε να χρησιμοποιηθούν οι ίδιοι και από τον αποκωδικοποιητή.

Στη διαδικασία της αποκωδικοποίησης, το inverse quantization εφαρμόζεται αμέσως μετά το zigzag scanning, ώστε να έχει προκύψει το 8x8 DCT block. Η μέθοδος που ακολουθείται είναι η αντίστροφη της κβαντοποίησης. Δηλαδή, το 8x8 block πολλαπλασιάζεται με τον αντίστοιχο πίνακα. Η διαδικασία αυτή πραγματοποιείται πριν το IDCT. Στο στάδιο αυτό πραγματοποιούνται (πριν το IDCT) πραγματοποιούνται επίσης και δύο άλλες λειτουργίες της αποκωδικοποίησης. Η **Saturate** και η **Mismatch Control**.

Στη διαδικασία saturate οι DCT συντελεστές περιορίζονται στο διάστημα [-2048, 2047], σύμφωνα με τον μαθηματικό τύπο:

$$F'[v][u] = \begin{cases} 2047 & F''[v][u] > 2047 \\ F''[v][u] & -2048 \le F''[v][u] \le 2047 \\ -2048 & F''[v][u] < -2048 \end{cases}$$

όπου F΄΄[v][u] οι αντίστοιχες αρχικές τιμές των DCT συντελεστών στο 8x8 block και F΄[v][u], οι saturated τιμές των συντελεστών.

Η διαδικασία mismatch control ορίζεται από τη μαθηματική σχέση:

$$sum = \sum_{v=0}^{v<8} \sum_{u=0}^{u<8} F'[v][u]$$

$$F[v][u] = F'[v][u] \text{ for all u, v except } u = v = 7$$

$$F[7][7] = \begin{cases} F'[7][7] & \text{if sum is odd} \\ \begin{cases} F'[7][7] - 1 & \text{if } F'[7][7] \text{ is odd} \\ F'[7][7] + 1 & \text{if } F'[7][7] \text{ is even} \end{cases} & \text{if sum is even} \end{cases}$$

όπου F΄[v][u] οι κβαντικοποιημένες τιμές των συντελεστών του 8x8 block πριν το mismatch control και F[v][u] οι νέες τιμές των συντελεστών μετά το mismatch control.

Αρχικά υπολογίζεται το άθροισμα όλων των κβαντικοποιημένων συντελεστών DCT ενός block. Ανάλογα με την τιμή του αθροίσματος μεταβάλλεται η τιμή του τελευταίου AC συντελεστή ως εξής:

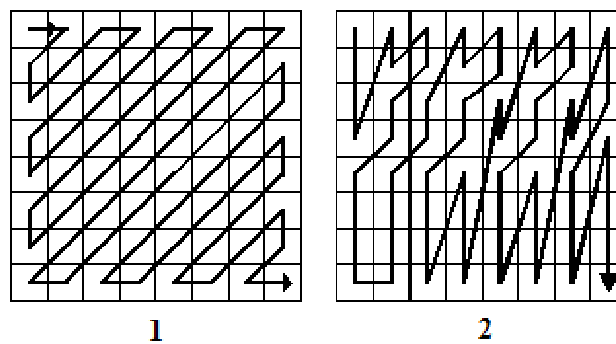
- Αν το άθροισμα είναι περιττός αριθμός τότε η τιμή δεν μεταβάλλεται.
- Αν το άθροισμα είναι άρτιος αριθμός και ο τελευταίος AC συντελεστής (F΄[7][7]) είναι περιττός τότε η τιμή αυτού μειώνεται κατά 1.
- Αν το άθροισμα είναι άρτιος αριθμός και ο τελευταίος AC συντελεστής (F΄[7][7]) είναι άρτιος τότε η τιμή αυτού αυξάνεται κατά 1.

Όπως αναφέρθηκε και στην παράγραφο 2.1.5 ο αλγόριθμος IDCT έχει αρκετά υψηλή πολυπλοκότητα. Αυτός είναι ο λόγος που χρησιμοποιούνται πιο γρήγοροι μέθοδοι για να επιτευχθεί η διαδικασία IDCT. Οι μέθοδοι αυτοί διαφέρουν σε ακρίβεια αποτελέσματος τόσο με τον ορισμό του IDCT, όσο και μεταξύ τους. Συνεπώς τα ανακατασκευασμένα picture των αποκωδικοποιητών θα αρχίσουν να αποκλίνουν με την πάροδο του χρόνου, καθώς δεδομένων των διαφορών στις IDCT μεθόδους, θα αποκωδικοποιούν περιστασιακά, ελαφρώς διαφορετικά block. Η διαδικασία mismatch control χρησιμοποιείται να μειωθεί αυτή η διαφορά μεταξύ των διαφορετικών αλγορίθμων Fast IDCT που χρησιμοποιούνται. O Η μέθοδος που χρησιμοποιείται

είναι η εξάλειψη των bit patterns που στατιστικά έχουν τη μέγιστη συμβολή για τις διαφορές ανάμεσα στις διαφορετικές μεθόδους IDCT.

2.1.7 ZigZag Scan και Inverse ZigZag Scan

Το **zigzag scan** (εικόνα 2.4) αποτελεί μία μέθοδο για τη μετατροπή ενός δυσδιάστατου πίνακα σε μία μονοδιάστατη ακολουθία για κωδικοποίηση run-length. Το zigzag scan εφαρμόζεται στα 8x8 block των κβαντισμένων συντελεστών DCT και ξεκινώντας από το DC σαρώνει το block με μία διαγώνια διαδρομή ώστε να προκύψει μία μονοδιάστατη ακολουθία. Με αυτή τη διαδικασία επιδιώκεται να εμφανιστούν όσο το δυνατόν περισσότερες συνεχόμενες μηδενικές τιμές στην ακολουθία, καθώς αυτές, κατά τη διαδικασία run-length coding κωδικοποιούνται με μηδενικό κόστος. Στο MPEG-2 υιοθετούνται δύο τρόποι για zigzag scan όπως φαίνεται στην εικόνα 2.4, με το δεύτερο τρόπο να είναι πιο αποδοτικός για interlaced video bitstream.



Εικόνα 2.4: (1) MPEG-2 zigzag block scan. (2) MPEG-2 alternative zigzag block scan.

Στον κωδικοποιητή το στάδιο zigzag scanning εφαρμόζεται πριν το στάδιο Variable Length Coding (VLD – Run Length Encoding και Huffman), ώστε να προκύψουν περισσότερα συνεχόμενα μηδενικά. Αποτελεί το προτελευταίο στάδιο της κωδικοποίησης.

Στον αποκωδικοποιητή, εφαρμόζεται η αντίστροφη διαδικασία. Το **Inverse zigzag scan** (εικόνα 2.3) παίρνει τη μονοδιάστατη ακολουθία των DCT συντελεστών και παράγει το 8x8 block. Το Inverse zigzag scan αποτελεί το δεύτερο στάδιο της αποκωδικοποίησης και πραγματοποιείται μετά το Variable Length Decoding και πριν από το Inverse Quantization.

2.1.8 Run Length Encoding - Decoding και Huffman

Οι δύο αυτές μέθοδοι δεν μεταβάλουν τις τιμές των δειγμάτων (όπως ο μετασχηματισμός DCT για παράδειγμα ή η κβαντοποίηση), αλλά χρησιμοποιούνται στο τελικό στάδιο της κωδικοποίησης, για να μειώσουν τον αριθμό bits που χρησιμοποιούνται για τη μετάδοση τους.

Η πρώτη (**Run-Length-Encoding**) στηρίζεται στο γεγονός ότι υπάρχουν πολλά μηδενικά σε διαδοχικές θέσεις και ανάμεσά τους κάποιες μη μηδενικές τιμές. Έτσι χρησιμοποιώντας κάποια σύμβολα (flags) που δείχνουν ότι αυτό που τα ακολουθεί δεν είναι διακριτή τιμή του σήματος αλλά ομάδα τιμών, ομαδοποιούν τα μηδενικά και τα μεταδίδουν σαν έναν αριθμό (πλήθος μηδενικών). Έτσι σχηματίζονται ζευγάρια τιμών που ο πρώτος δείχνει το πλάτος μιας μη μηδενικής συνιστώσας και ο δεύτερος των αριθμό μηδενικών που ακολουθεί μέχρι την επόμενη, γλυτώνοντας έτσι πολλά bit.

Η δεύτερη (**Huffman**) είναι μια μέθοδος που αντιστοιχίζει σε συχνότερα εμφανιζόμενες τιμές μία συμβολική τιμή που είναι μικρή (έχει όσο το δυνατόν λιγότερα bits). Ταυτόχρονα δημιουργεί και ένα «λεξικό» (έναν πίνακα δηλαδή) που δείχνει αυτή την αντιστοίχηση και προσθέτει και το λεξικό αυτό στο σήμα για να χρησιμοποιηθεί από τον αποκωδικοποιητή. Έτσι τιμές που εμφανίζονται συχνά και έχουν μεγάλο αριθμό bits περιγράφονται με άλλες που έχουν μικρότερο, άρα πάλι έχουμε οικονομία σε bits.

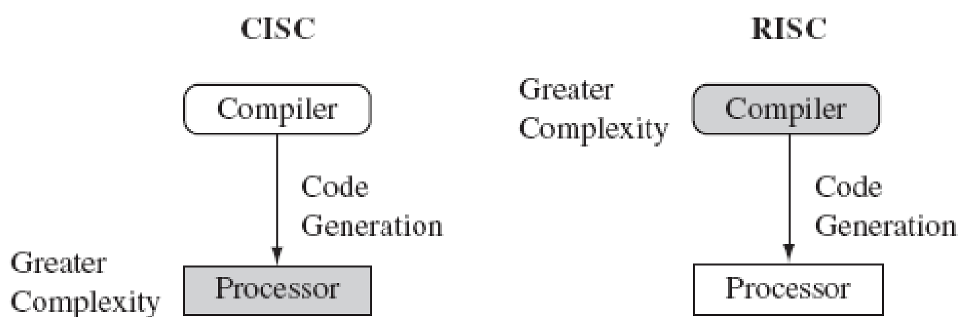
Και οι δύο αυτές μέθοδοι συνδυάζονται στο στάδιο **Variable Length Coding** (**VLC** εικόνα 2.2) του κωδικοποιητή. Σε αυτό το στάδιο ανιχνεύονται οι ακολουθίες συνεχόμενων μηδενικών DCT συντελεστών και προκύπτουν τα ζεύγη (run, length), όπου run το πλήθος των διαδοχικών μηδενικών τιμών και length η απόλυτη τιμή της επόμενης μη μηδενικής τιμής που ακολουθεί (το πρόσημο κωδικοποιείται με ξεχωριστό bit). Τα ζεύγη αυτά κωδικοποιούνται στη συνέχεια με πίνακες Huffman, ώστε για τα ζεύγη με μεγαλύτερη πιθανότητα εμφάνισης να δαπανούνται και λιγότερα bits. Μέσω τέτοιων πινάκων κωδικοποιούνται και άλλα χαρακτηριστικά του βίντεο (π.χ. το macroblock type).

Η αντίστροφη διαδικασία **Variable Length Decoding** (**VLD** εικόνα 2.3) αποτελεί το πρώτο στάδιο της αποκωδικοποίησης. Σε αυτό το στάδιο η ακολουθία των bits που λαμβάνεται από το video bitstream μέσω των πινάκων Huffman και της διαδικασίας Run Length Decoding, αντιστοιχίζεται στα χαρακτηριστικά και τα δεδομένα των βίντεο. Τα motion vector που προκύπτουν από το VLD χρησιμοποιούνται από το MC για την εύρεση των prediction macroblock στα reference picture. Ενώ οι DCT συντελεστές (ή οι διαφορές αυτών - P, B picture) σε μορφή 8x8 block χρησιμοποιούνται στα υπόλοιπα στάδια της αποκωδικοποίησης.

2.2 ARM Cortex A8

2.2.1 Αρχιτεκτονική ARM

Η αρχιτεκτονική ARM αποτελεί μία **32-bit reduced instruction set computer (RISC)** αρχιτεκτονική. Χαρακτηριστικό γνώρισμα των RISC αρχιτεκτονικών είναι πως βασίζονται σε μία λογική σχεδιασμού επεξεργαστών που υποστηρίζει πως απλοποιημένες εντολές μπορούν να παρέχουν μεγαλύτερη απόδοση, αν αυτή η απλοποίηση μπορεί να παρέχει πιο γρήγορη εκτέλεση των εντολών. Το εικόνα 2.5 φαίνεται σχηματικά η λογική της RISC αρχιτεκτονικής σε σχέση αυτή της CISC (Complex instruction set computing).



Εικόνα 2.5 : CISC vs RISC. Η CISC εστιάζει στην πολυπλοκότητα του επεξεργαστή ενώ η RISC στην πολυπλοκότητα του μεταγλωτιστή.

Βασικά χαρακτηριστικά της αρχιτεκτονικής RISC που ενσωματώνει η ARM είναι:

- Τη load/store αρχιτεκτονική.
- Τη μη υποστήριξη για misaligned προσπελάσεις μνήμης (ωστόσο τώρα υποστηρίζεται από τους ARMv6 πυρήνες και μετά).
- Συγκεκριμένο μέγεθος εντολών στα 32 bits για γρήγορη αποκωδικοποίηση και pipelining. Ωστόσο στη συνέχεια με την υποστήριξη των Thumb εντολών (16-bit) αυτό άλλαξε.
- Εκτέλεση σε single-cycle (κυρίως).

Οι επεξεργαστές ARM κατατάσσονται ανάλογα με την "οικογένεια" επεξεργαστών που ανήκουν, το version της αρχιτεκτονικής και τον πυρήνα τους Η τελευταία οικογένεια επεξεργαστών είναι η Cortex στην οποία ανήκει και ο πυρήνας Cortex A8 με version ARMv7-A.

Ένα σημαντικό χαρακτηριστικό των επεξεργαστών ARM από τους ARM7TDMI και έπειτα είναι η υποστήριξη των Thumb instructions. Όταν ο επεξεργαστής δουλεύει σε αυτό το mode, εκτελούνται 16-bit εντολές. Τις περισσότερες φορές οι εντολές αυτές αντιστοιχίζονται απευθείας σε κανονικές εντολές ARM. Αυτή η εξοικονόμηση χώρου σε bits επιτυγχάνεται από ορισμένες λειτουργίες μίας εντολής που υπονοούνται και από το όριο που τίθεται στο μέγεθος των τελεστών. Η τελευταία γενιά των Thumb εντολών είναι το Thumb2 που υποστηρίζει και ορισμένες 32bit εντολές.

2.2.2 Επεξεργαστής Cortex A8

Ο επεξεργαστής ARM Cortex A8, όπως αναφέρεται και στην προηγούμενη παράγραφο κατατάσσεται στην οικογένεια επεξεργαστών Cortex με version ARMv7-A. Αποτελεί ένα σύγχρονο επεξεργαστή χαμηλής κατανάλωσης που στοχεύει να κατακτήσει τον κόσμο των smart devices αλλά και να μπει δυναμικά στο χώρο των netbook. Συσκευές που ενσωματώνουν τον επεξεργαστή Cortex A8 μεταξύ άλλων είναι το Apple iPhone 3GS, το Apple iPhone 4, to Nokia N900, το Sony Ericsson Satio, το Apple iPad. Μερικές από τις εφαρμογές του φαίνονται στον παρακάτω πίνακα :

Product Type	Application
Smartphone	Application processor running fully featured mobile OS
Netbook	Power-efficient main processor running desktop OS
Set-top Box	Main processor for managing Rich OS, Multi-format A/V and UI
Digital TV	Processor for managing Rich OS, UI, browser
Home Networking	Control processor for system management
Storage Networking (HDD, SSD)	Control processor to manage traffic flow
Printer	High-performance integrated processor

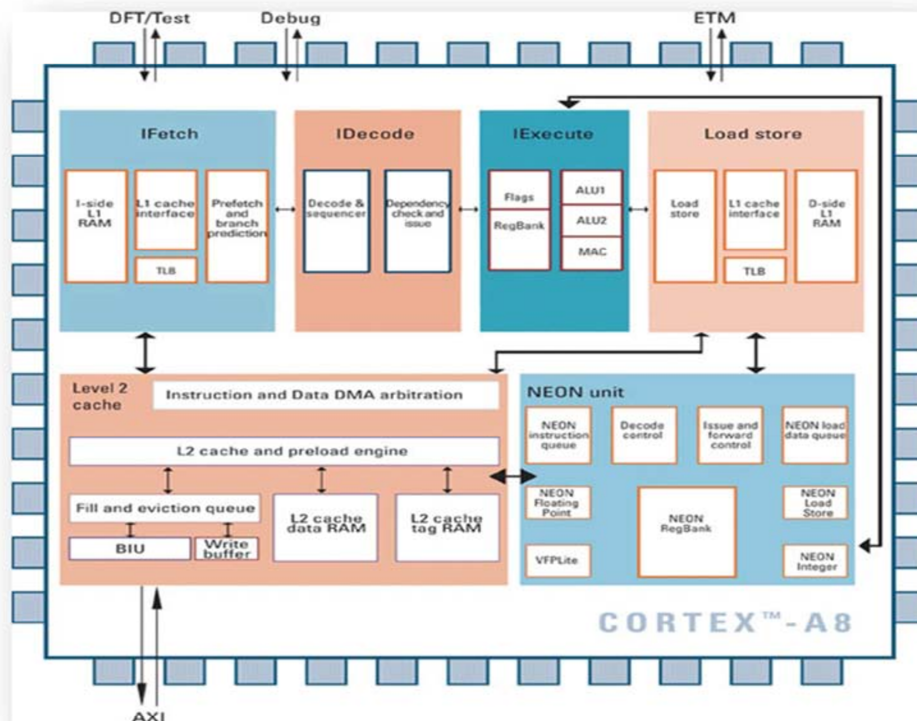
Πίνακας 2.1 : Εφαρμογές του επεξεργαστή ARM Cortex A8

Τα βασικά χαρακτηριστικά της αρχιτεκτονικής Cortex A8 είναι :

- Συχνότητα λειτουργίας από 600MHz έως και πάνω από 1GHz.
- Dhrystone Performance : 2.0 DMIPS / MHz (Dhrystone Millions Instructions Per Second) / MHz
- Single core
- Dual pipeline

- Instruction Set Architecture :
  - ✓ ARM.
  - ✓ Thumb2 – Thumb.
  - ✓ NEON (περιγράφονται στην παράγραφο 5.3)
  - ✓ VFPv3 Floating point: Υποστηρίζεται η δυνατότητα της χρήσης SIMD και στους floating point.
- Memory Management Unit (MMU)
- Optimized Level 1 cache: Το επίπεδο 1 της μνήμης cache είναι ενσωματωμένο στενά στον επεξεργαστή παρέχοντας πρόσβαση σε ένα κύκλο μηχανής και συνδυάζει το ελάχιστο latency με τη μέθοδο hash για να μεγιστοποιηθεί η απόδοση και να ελαχιστοποιηθεί η κατανάλωση ισχύος.
- Integrated Level 2 cache: Το επίπεδο 2 της μνήμης cache είναι ενσωματωμένο στον πυρήνα.
- Dynamic Branch Prediction : Για να ελαχιστοποιηθούν τα λάθη στο branch prediction, ο δυναμικός branch predictor πετυχαίνει 95% ακρίβεια σε ένα μεγάλο μέρος των συγκριτικών μετρήσεων επίδοσης.
- Σύστημα μνήμης: Βελτιστοποιημένο για χαμηλή κατανάλωση ενέργειας και υψηλή επίδοση.

Στην παρακάτω εικόνα (2.6) παρουσιάζεται το σχήμα του επεξεργαστή ARM Cortex A8.



Εικόνα 2.6: ARM Cortex A8 Schema

ΚΕΦΑΛΑΙΟ 3

Μέθοδοι Βελτιστοποίησης Λογισμικού

Στο κεφάλαιο αυτό καταγράφονται και αναλύονται οι τρόποι Βελτιστοποίησης Λογισμικού. Αρχικά περιγράφονται οι τεχνικές που μπορούν να χρησιμοποιηθούν σε επίπεδο Γλώσσας Προγραμματισμού ώστε ο πηγαίος κώδικας να γίνει πιο αποδοτικός. Στη συνέχεια παρουσιάζονται η βελτιστοποίηση αλγορίθμων και η χρήση ιδιοτήτων των αρχείων εισόδου, ως εργαλεία για τη βελτίωση της απόδοσης, ενώ γίνεται αναφορά και στον τρόπο που αυτά τα εργαλεία χρησιμοποιήθηκαν στην παρούσα εργασία. Η υποστήριξη που παρέχεται από την εκάστοτε αρχιτεκτονική, μέσω των SIMD εντολών, παρουσιάζεται στην επόμενη παράγραφο. Δίνεται ο ορισμός του SIMD και αναλύεται ο τρόπος λειτουργίας των εντολών αυτών. Παρουσιάζονται οι NEON εντολές που υποστηρίζονται από την αρχιτεκτονική ARM και περιγράφεται ο τρόπος που μπορούν αυτές να χρησιμοποιηθούν αποδοτικά. Τέλος, παρουσιάζονται οι διάφορες επιλογές βελτιστοποίησης του μεταγλωττιστή ARM και γίνεται ανάλυση της κάθε επιλογής ως προς το είδος της βελτιστοποίησης που επιτυγχάνει.

3.1 Βελτιστοποίηση Πηγαίου Κώδικα C

Σημαντικό ρόλο στην απόδοση μιας εφαρμογής παίζει ο τρόπος που χρησιμοποιείται η εκάστοτε Γλώσσα Προγραμματισμού για την ανάπτυξη του πηγαίου κώδικα και την υλοποίηση των απαραίτητων αλγορίθμων.

Σε πολλές περιπτώσεις, με χρήση απλών μεθόδων και τεχνικών, ο πηγαίος κώδικας μπορεί να τροποποιηθεί βελτιώνοντας την απόδοση. Συνήθως αυτές στοχεύουν στην ελαχιστοποίηση της χρήσης μνήμης, στην αντικατάσταση πράξεων με υψηλό υπολογιστικό κόστος και στη μείωση των branches.

Τέτοιες μέθοδοι χρησιμοποιούνται και στην παρούσα εργασία και αφορούν τη Γλώσσα Προγραμματισμού C, με την οποία έχει αναπτυχθεί ο κώδικας αναφοράς του Mpeg2 Decoder. Πιο συγκεκριμένα σε διάφορα σημεία του πηγαίου κώδικα χρησιμοποιούνται οι παρακάτω μέθοδοι :

- Ελαχιστοποίηση της χρήσης μνήμης και επαναχρησιμοποίηση ενδιάμεσων αποτελεσμάτων στους καταχωρητές.
- Μείωση των ελέγχων (if statements) με αντικατάσταση τους από απλές πράξεις.
- Loop Unrolling. Αποτελεί και αυτό ένα τρόπο για μείωση των ελέγχων (if statements). Επιτυγχάνεται μειώνοντας τον αριθμό των επαναλήψεων και ξεδιπλώνοντας τις εσωτερικές πράξεις του loop.
- Χρήση LookUp Table. Όταν το σύνολο των αποτελεσμάτων μίας υπολογιστικά ακριβής πράξης ή ενός ελέγχου είναι ήδη γνωστό, μπορεί να αποθηκευτεί στη μνήμη και να διαβαστεί από εκεί το κατάλληλο αποτέλεσμα χωρίς να εκτελεστεί η πράξη ή ο έλεγχος. Επιπλέον μέθοδος για μείωση των ελέγχων (if statements).
- Αντικατάσταση εκφράσεων από εκφράσεις που έχουν μικρότερο υπολογιστικό κόστος. Συνήθως αντικατάσταση αριθμητικών με λογικές πράξεις.
- Μεταφορά υπολογισμών που δεν εξαρτώνται από το loop, έξω από αυτό. Με αυτό τον τρόπο μειώνεται ο αριθμός των εκτελέσιμων εντολών.
- Μείωση των κλήσεων συναρτήσεων. Κάθε κλήση συνάρτησης επιφέρει επιπλέον κόστος λόγω της ανάγκης για αποθήκευση και αποκατάσταση των απαραίτητων καταχωρητών. Επιτυγχάνεται με χρήση inline συναρτήσεων, ένωση συναρτήσεων ακολουθούν η μία την κλήση της άλλης και έχουν τα ίδια ορίσματα κλήσης.

3.2 Βελτιστοποίηση Αλγορίθμων και Χρήση Ιδιοτήτων των Βίντεο

Οι αλγόριθμοι που υλοποιούνται στα πλαίσια μιας εφαρμογής αποτελούν ένα σημαντικό παράγοντα απόδοσης. Η βελτιστοποίηση αλγορίθμων μπορεί να επιτευχθεί είτε σε θεωρητικό είτε σε επίπεδο υλοποίησης σε κάποιο υπολογιστικό περιβάλλον.

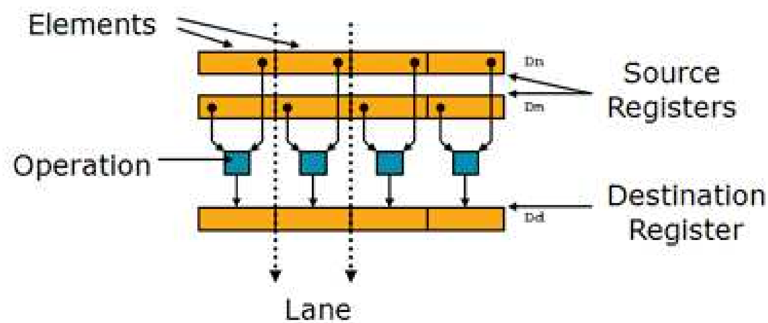
Κύριο χαρακτηριστικό τους είναι η πολυπλοκότητα. Η πολυπλοκότητα υπολογίζεται είτε συναρτήσει του μεγέθους της εισόδου σε υπολογιστικές πράξεις (time complexity) είτε σε ανάγκες αποθηκευτικού χώρου (storage complexity). Συνεπώς η βελτιστοποίηση αλγορίθμων ανάγεται σε μείωση της πολυπλοκότητας τους.

Οι τεχνικές που χρησιμοποιούνται ποικίλουν ανάλογα με το είδος του αλγορίθμου. Έτσι π.χ. για επαναληπτικούς αλγορίθμους χρησιμοποιούνται τεχνικές μείωσης των επαναλήψεων ή για αναδρομικούς μείωση της χρήσης στοίβας. Επίσης η χρήση ενδιάμεσων αποτελεσμάτων μπορεί να χρησιμοποιηθεί για τη μείωση του συνολικού αριθμού πράξεων που απαιτούνται (περίπτωση IDCT σε Fast IDCT).

Η μελέτη κοινών χαρακτηριστικών που εμφανίζονται στα δεδομένα εισόδου ενός αλγορίθμου και η ανάπτυξη ενός αποτελεσματικού τρόπου για την αξιοποίησή τους αποτελεί επίσης μία αποτελεσματική μέθοδο στη βελτιστοποίηση αλγορίθμων. Συνήθως, εξετάζεται η συχνότητα εμφάνισης ενός τέτοιου χαρακτηριστικού. Τις περισσότερες φορές τα χαρακτηριστικά αυτά αποτελούν ειδικές περιπτώσεις του αλγορίθμου για τις οποίες δεν απαιτείται η πλήρης διαδικασία. Οπότε, αν η πιθανότητα εμφάνισής τους είναι υψηλή και ο αλγόριθμος τροποποιηθεί αποδοτικά, τότε κατά την πρώτη επεξεργασία των δεδομένων εισόδου μπορεί να επιλεχθεί η βελτιωμένη έκδοση. Ωστόσο, για να επιτευχθεί ο παραπάνω στόχος δεν θα πρέπει το κόστος του αρχικού ελέγχου μαζί με το κόστος του βελτιωμένου αλγορίθμου να ξεπερνά το κόστος εκτέλεσης του αρχικού αλγορίθμου.

3.3 Χρήση SIMD - NEON Εντολών

Με τον όρο **SIMD** (Simple Instruction, Multiply Data) εννοούμε ειδικές εντολές που παρέχονται από τις αρχιτεκτονικές, οι οποίες με μία εντολή εκτελούν τη συγκεκριμένη πράξη σε πολλά δεδομένα ταυτόχρονα. Τα δεδομένα φορτώνονται σε ειδικούς registers μεγαλύτερου μεγέθους από αυτούς γενικής χρήσης του επεξεργαστή οι οποίοι συνήθως έχουν μέγεθος 64bit και 128bit. Στην αποκωδικοποίηση βίντεο και γενικότερα σε εφαρμογές multimedia αποτελούν σημαντικό εργαλείο για τη βελτιστοποίηση της απόδοση. Αυτό συμβαίνει επειδή σε τέτοιες εφαρμογές απαιτούνται συνήθως οι ίδιες πράξεις σε πολλά δεδομένα (π.χ σε όλα τα pixel ενός frame). Στην εικόνα 5.1 παρουσιάζεται ένα παράδειγμα εκτέλεσης μίας πράξης SIMD.



Εικόνα 5.1: Παράδειγμα εκτέλεσης SIMD εντολής

Η αρχιτεκτονική ARM Cortex A8 υποστηρίζει SIMD εντολές οι οποίες ονομάζονται NEON. Η μονάδα που τις υποστηρίζει είναι ανεξάρτητη από την κύρια μονάδα επεξεργασίας. Το **NEON Unit** παρέχει 32 64bit registers οι οποίοι συνδυαζόμενοι αποτελούν και τους 16 128bit registers. Επίσης προσφέρει ένα πλήρες Instructions Set για τη χρήση των registers ενώ όλες οι εντολές που υποστηρίζονται πακετάρονται σε μορφή συναρτήσεων C. Αυτή η μορφή των NEON εντολών ονομάζεται Intrinsic.

Για τη χρήση NEON εντολών μέσω Intrinsic υποστηρίζονται οι παρακάτω τύποι μεταβλητών :

64 bit vector	128 bit vector	Τύπος Μεταβλητής
int8x8_t	int8x16_t	Sign integer 8 bit
int16x4_t	int16x8_t	Sign integer 16 bit
int32x2_t	int32x4_t	Sign integer 32 bit
int64x1_t	int64x2_t	Sign integer 64 bit
uint8x8_t	uint8x16_t	Unsigned integer 8 bit
uint16x4_t	uint16x8_t	Unsigned integer 16 bit
uint32x2_t	uint32x4_t	Unsigned integer 32 bit
uint64x1_t	uint64x2_t	Unsigned integer 64 bit
float16x4_t	float16x8_t	Float 16 bit
float32x2_t	float32x 4_t	Float 32 bit

Πίνακας 3.1: Είδη NEON μεταβλητών

Επίσης υποστηρίζεται και ο τύπος της μορφής *struct int16x4x2_t{ int16x4_t val[2]; };* για όλους τους τύπους του πίνακα 3.1, ενώ οι τύποι μεταβλητών float ανήκουν στην κατηγορία VFPv3 Floating point instruction.

Το instruction set που παρέχεται από τον Cortex A8 περιλαμβάνει τις πράξεις για :

- Addition
- Multiplication
- Subtraction
- Comparison
- Absolute difference
- Max/Min
- Pairwise addition
- Folding minimum
- Folding minimum
- Reciprocal/Sqrt
- Shifts by signed variable
- Shifts by a constant
- Shifts with insert
- Loads and stores of a single vector
- Loads and stores of an N-element structure
- Extract lanes from a vector
- Set lanes within a vector
- Initialize a vector from bit pattern
- Set all lanes to same value
- Combining vectors
- Splitting vectors
- Converting vectors
- Table look up
- Extended table look up intrinsics
- Operations with a scalar value
- Vector extract
- Reverse vector elements (swap endianness)
- Other single operand arithmetic
- Logical operations
- Transposition operations

3.4 Compiler Optimization Options

Η χρήση των επιλογών βελτιστοποίησης που παρέχονται από το μεταγλωττιστή αποτελεί μία επιπλέον μέθοδο βελτιστοποίησης λογισμικού. Κάθε επιλογή στοχεύει σε συγκεκριμένο επίπεδο βελτιστοποίησης.

Ο μεταγλωττιστής ARM επιτυγχάνει υψηλή βελτιστοποίηση για την παραγωγή μικρότερου μεγέθους κώδικα και την αύξηση της απόδοσης. Εφαρμόζει βελτιστοποιήσεις κοινές με άλλους μεταγλωττιστές. Για παράδειγμα βελτιστοποιήσεις ροής πληροφοριών, όπως η απαλοιφή μιας κοινής υπό – έκφρασης και βελτιστοποιήσεις βρόγχου όπως ο συνδυασμός και η κατανομή.

Ο μεταγλωττιστής ARM παρέχει δύο επιλογές βελτιστοποίησης για το μέγεθος του κώδικα (-Ospace) και την απόδοση (-Otime).

- *-Ospace*: Η επιλογή αυτή επιβάλλει στο μεταγλωττιστή να εφαρμόσει βελτιστοποίηση για την μείωση του μεγέθους του παραγομένου image εις βάρος του συνολικού χρόνου εκτέλεσης. Η -Ospace επιλέγεται όταν το μέγεθος του κώδικα θεωρείται πιο κρίσιμο από την απόδοση. Για παράδειγμα όταν επιλέγεται η –Ospace, μεγάλα κομμάτια κώδικα εκτελούνται με κλήση συναρτήσεων αντί για inline κώδικα. Αν είναι απαραίτητο, τα τμήματα του κώδικα που είναι κρίσιμα ως προς το χρόνο μπορούν να μεταγλωττιστούν με την επιλογή –Otime και τα υπόλοιπα με την επιλογή –Ospace. Αν δεν επιλεχθεί καμία από τις –Otime και -Ospace, ο μεταγλωττιστής υποθέτει την –Ospace (Default choice).
- *-Otime:* Η επιλογή αυτή επιβάλλει στο μεταγλωττιστή να εφαρμόσει βελτιστοποιήσεις για τη μείωση του χρόνου εκτέλεσης, εις βάρος του μεγέθους του image. Η -Otime επιλέγεται όταν ο χρόνος εκτέλεσης θεωρείται πιο σημαντικός από το μέγεθος του image. Για παράδειγμα με όταν επιλέγεται η –Otime, ο μεταγλωττιστής μεταγλωττίζει τον κώδικα

```
while(expression)
{
        body;
}
```

σαν

```
if(expression)
{
        do {body;}
        while (expression);
}.
```

Οι παραπάνω επιλογές βελτιστοποίησης επιτυγχάνουν το στόχο τους στην πλειοψηφία των περιπτώσεων. Ωστόσο δεν είναι εγγυημένο πως πάντα η –Ospace θα παράγει μικρότερου μεγέθους image ή η –Otime θα επιτυγχάνει καλύτερο χρόνος εκτέλεσης.

Ο μεταγλωττιστής ARM δίνει και τη δυνατότητα επιλογής του επιπέδου βελτιστοποίησης μέσω της -Onum. Το τελικό αποτέλεσμα βελτιστοποίησης εξαρτάται από το επίπεδο βελτιστοποίησης που θα επιλεγεί και από την επιλογή ανάμεσα σε μείωση του μεγέθους του image και μείωση του χρόνου εκτέλεσης.

Οι τιμές που μπορεί να πάρει το num και τα αντίστοιχα επίπεδα βελτιστοποίησης είναι :

- -O0: Ελάχιστη βελτιστοποίηση. Οι περισσότερες από τις βελτιστοποιήσεις απενεργοποιούνται. Επιτυγχάνει το καλύτερο δυνατό αποτέλεσμα για debug και το χαμηλότερο επίπεδο βελτιστοποίησης.

- -O1: Ισορροπημένη βελτιστοποίηση. Αφαιρεί inline και static συναρτήσεις που δεν χρησιμοποιούνται. Απενεργοποιεί τις βελτιστοποιήσεις που βλάπτουν το debug mode.
- -O2: Υψηλή βελτιστοποίηση. Αποτελεί το default επίπεδο βελτιστοποίησης. Αν χρησιμοποιείται το --debug command line argument, η αντιστοίχιση του κώδικα αντικειμένου με τον πηγαίο κώδικα δεν είναι ξεκάθαρη.
- -O3: Μέγιστη βελτιστοποίηση. Η –O3 εφαρμόζει τις ίδιες βελτιστοποιήσεις με την –O2, ωστόσο ευνοεί ακόμη περισσότερο από την –O2 τη μείωση του μεγέθους του κώδικα ή τη βελτίωση της απόδοσης, ανάλογα με τη βελτιστοποίηση που έχει επιλεγεί. Δηλαδή :
    - ❖ Η –O3 –Otime αποσκοπεί στην δημιουργία ταχύτερου κώδικα από ότι η –O2 –Otime, με το ρίσκο να μεγαλώσει ακόμη περισσότερο το μέγεθος του κώδικα.
    - ❖ Η –O3 –Ospace αποσκοπεί στην δημιουργία ακόμη μικρότερου μεγέθους κώδικα από ότι η –O2 –Ospace, με το ρίσκο να μειωθεί ακόμη περισσότερο η απόδοση.

Επίσης, η –O3 εφαρμόζει και επιπλέον, πιο επιθετικές βελτιστοποιήσεις όπως είναι :

- High-level scalar optimizations, συμπεριλαμβανομένου του ξετύλιγμα βρόχου (loop unrolling), για -O3 -Otime. Αυτό μπορεί να επιτύχει σημαντική βελτίωση με μικρό κόστος στο μέγεθος του κώδικα, αλλά με το ρίσκο να αυξηθεί ο χρόνος μεταγλώττισης .
- Πιο επιθετικό inlining και automatic inlining για -O3 -Otime.

ΚΕΦΑΛΑΙΟ 4

Υλοποίηση – Βελτιστοποίηση Λογισμικού

Στο κεφάλαιο αυτό αρχικά παρουσιάζεται το εργαλείο RVDS v4.0 που χρησιμοποιήθηκε στη παρούσα εργασία. Στη συνέχεια αναφέρονται τα κριτήρια επιλογής των κατάλληλων αρχείων που χρησιμοποιούνται σαν είσοδοι σε ένα πρόβλημα Βελτιστοποίησης Λογισμικού. Έπειτα καταγράφονται τα χαρακτηριστικά του αρχείου εισόδου που χρησιμοποιήθηκε στην παρούσα εργασία, καθώς και η διαδικασία δημιουργίας του. Στη δεύτερη παράγραφο περιγράφεται η διαδικασία profiling και ο ρόλος της στη Βελτιστοποίηση Λογισμικού, ενώ παρουσιάζονται και τα αποτελέσματα του profiling του κώδικα αναφοράς. Αποτελέσματα που αποτελούν οδηγό για την επιλογή των σημείων του κώδικα που βελτιώθηκε. Στις επόμενες παραγράφους περιγράφονται οι αλλαγές - βελτιώσεις που πραγματοποιηθήκαν σε κάθε σημείο του κώδικα, ομαδοποιημένες σε υποπαραγράφους ανάλογα με το στάδιο της αποκωδικοποίησης που αυτές συμμετέχουν, καθώς και οι λόγοι που μας οδήγησαν στην κάθε αλλαγή - βελτίωση. Για κάθε μία από αυτές παρουσιάζονται οι σχετικές μετρήσεις απόδοσης. Πρέπει να σημειωθεί πως η ανάπτυξη του κώδικα σε περιβάλλον επεξεργαστή ARM Cortex A8, καθώς και τα profiling για τις μετρήσεις απόδοσης έγιναν με το Real View Development Suite v4.0 (RVDS v4.0). Οι λειτουργίες και οι δυνατότητες του RVDS περιγράφονται στην παράγραφο 4.1.

4.1 Real View Development Suite v4.0 (RVDS v4.0)

Το πρόγραμμα Real View development Suite αποτελεί ένα εργαλείο προσομοίωσης για επεξεργαστές αρχιτεκτονικής ARM. Το RVDS χρησιμοποιήθηκε στην παρούσα εργασία για την ανάπτυξη κώδικα σε περιβάλλον ARM Cortex A8.

Μια σημαντική δυνατότητα που παρέχεται από το RVDS είναι η δυνατότητα για profiling με τη χρήση του RVDS profiler. Το αποτέλεσμα του profiling μετά το πέρας της εκτέλεσης του προγράμματος είναι όπως φαίνεται παρακάτω:

Totals

Function Name	Coverage	Self Time	Total Time	Avg CPI	Delay	Eff	Called	Callers	Callees	Read
Decode_MPEG2_Non_Intra_Block	60.45%	3,841,882,080	6,217,785,131	1.46	787,136,591	79.51%	5,532,401	1	7	2,702,009
form_prediction	43.76%	3,529,595,929	3,529,595,929	1.23	197,759,122	94.40%	8,978,716	1	0	1,503,598
Full_Fast_IDCT	100.00%	2,243,842,125	2,243,842,125	1.08	117,638,325	94.76%	4,356,975	2	0	3,154,449
Decode_MPEG2_Intra_Block	63.37%	2,221,244,877	3,179,180,223	1.49	500,130,002	77.48%	2,398,554	1	8	1,527,586
decode_macroblock	41.26%	1,957,339,680	14,454,861,505	1.45	363,651,073	81.42%	3,426,422	1	5	1,565,164
motion_compensation	15.19%	1,900,159,589	5,998,663,486	1.12	97,065,376	94.89%	3,510,000	1	2	653,483
Get_motion_code	54.25%	738,384,644	738,384,644	1.61	127,328,656	82.76%	14,559,584	1	0	1,038,988
motion_vector	61.70%	564,507,180	1,723,538,250	1.48	65,437,547	88.41%	7,279,792	1	2	593,786
macroblock_modes	40.16%	551,355,893	860,480,413	1.60	120,596,129	78.13%	3,426,422	1	1	555,944
Fast_IDCT_4x4_MisMatch1	100.00%	513,840,600	513,840,600	1.14	57,501,210	88.81%	1,223,430	2	0	905,338
slice	64.65%	485,063,478	21,126,604,744	1.50	91,321,598	81.17%	78,000	1	5	399,726
decode_motion_vector	94.59%	420,646,426	420,646,426	1.42	7,579,308	98.20%	14,559,584	1	0	407,668
form_predictions	39.75%	418,962,835	3,988,991,897	1.48	37,432,591	91.07%	3,110,241	1	2	496,786
motion_vectors	50.93%	407,808,329	2,131,346,579	1.55	62,171,027	84.75%	4,161,709	1	1	618,815
Fast_IDCT_4x4_MisMatch0	100.00%	171,473,808	171,473,808	1.11	13,968,372	91.85%	481,668	2	0	250,467
Get_macroblock_address_increment	49.81%	147,978,663	147,978,663	1.61	13,766,467	90.70%	3,426,422	2	0	244,314
Fast_IDCT_2x2_MisMatch1	100.00%	144,071,481	144,071,481	1.12	5,009,010	96.52%	744,007	2	0	65,472
Get_macroblock_type	46.75%	142,051,153	309,124,520	1.76	23,984,954	83.12%	3,426,422	1	[illegible]	[illegible]

Functions	1	Size	8 (< 0.01%)	Coverage	50.00%	Read	0
Self Time	2 (< 0.01%)	Avg CPI	2.00	Delay	0 (0.00%)	Write	0

Εικόνα 4.1: Παράδειγμα αποτελέσματος profiling στο RVDS

Η προσομοίωση του κώδικα ARM πραγματοποιήθηκε με τη χρήση του RTMS Emulation Baseboard Cortex A8. Ωστόσο με αυτό το εργαλείο θα πρέπει να ορίσουμε κάποιες παραδοχές. Αυτό το συμπέρασμα προκύπτει από τις μετρήσεις που πραγματοποιήθηκαν στην παρούσα εργασία. Από τα αποτελέσματα προέκυψαν πως δεν προσομοιώνεται το dual pipelining του επεξεργαστή καθώς πάντα το CPI (Cycles Per Instruction είναι > =1). Επίσης δεν έγινε γνωστός ο τρόπος προσομοίωσης των cache.

4.2 Δημιουργία Αρχείου Εισόδου (Input Video)

Η σωστή μέτρηση της απόδοσης μιας εφαρμογής προϋποθέτει τη χρήση ενός κατάλληλου και αντιπροσωπευτικού αρχείου εισόδου. Όσα περισσότερα χαρακτηριστικά

συγκεντρώνει ένα αρχείο εισόδου τόσο πιο ρεαλιστικά και αξιόπιστα θα είναι τα αποτελέσματα που θα προκύψουν.

Στην παρούσα εργασία σαν αρχείο εισόδου χρησιμοποιήθηκε με τα εξής βασικά χαρακτηριστικά :

- Συνολικός Αριθμός Καρέ : 2600
- Frame Rate : 30 fps
- Ανάλυση : 720x480
- Bit Rate : 4Mbps
- Chroma Format : 4:2:0

Το παραπάνω βίντεο προέκυψε από το συνδυασμό δέκα μικρότερων βίντεο των 260 καρέ. Τα αρχικά βίντεο ήταν σε ασυμπίεστη μορφή (αρχεία .yuv), με Chroma Format 4:2:0 και ανάλυση επίσης 720x480. Αρχικά, τα βίντεο κωδικοποιήθηκαν με τη χρήση του Mpeg2 Encoder v1.2 του MPEG Software Simulation Group. Για την κωδικοποίηση χρησιμοποιήθηκαν frame και field DCT coding, ενώ επιλέχθηκε Bit Rate ίσο με 4 Mbps. Στη συνέχεια τα βίντεο των 260 καρέ που προέκυψαν από την κωδικοποίηση, ενώθηκαν για να παραχθεί το Input_Inter.m2v που χρησιμοποιήθηκε τελικά σαν είσοδος στην εφαρμογή.

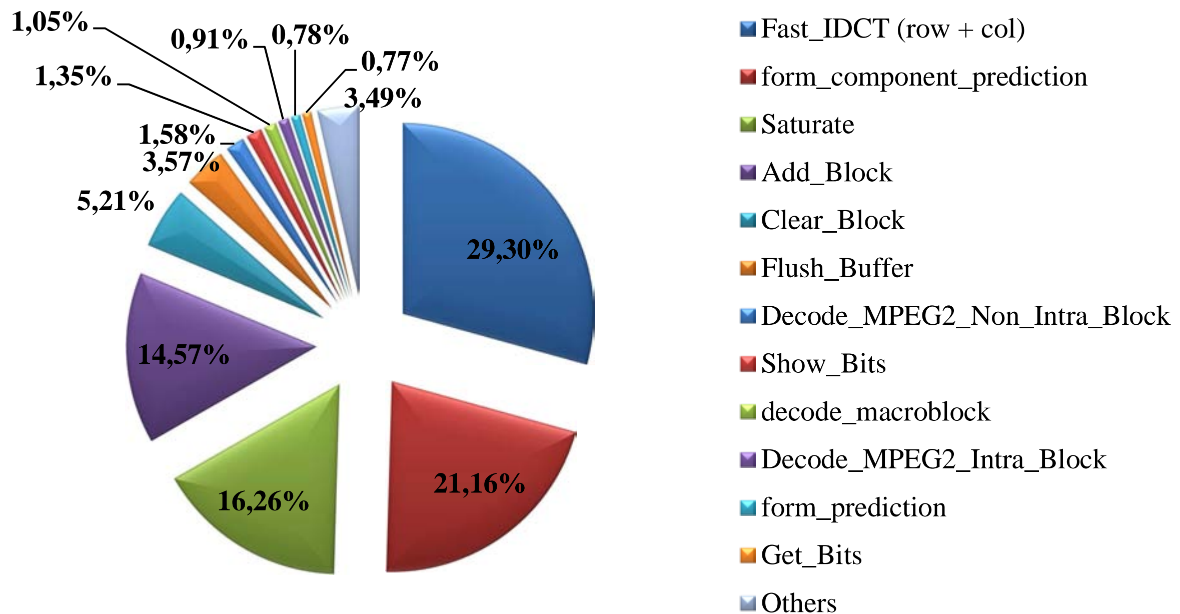
Τα παραπάνω χαρακτηριστικά αποτελούν τα χαρακτηριστικά ενός κοινού DVD – movie. Συνεπώς το αρχείο εισόδου ανταποκρίνεται στα χαρακτηριστικά ενός διαδεδομένου βίντεο MPEG-2.

4.3 Profiling Κώδικα Αναφοράς

Το profiling αποτελεί μία από τις σημαντικότερες διαδικασίες που πρέπει να γίνουν σε μια εργασία Βελτιστοποίησης Λογισμικού. Με τον όρο profiling ονομάζουμε τη διαδικασία συλλογής πειραματικών δεδομένων που θα αντικατοπτρίζουν την πολυπλοκότητα της εφαρμογής στο σύστημα το οποίο επιθυμούμε να τρέξουμε την εφαρμογή μας.

Σημαντικό ρόλο για ένα σωστό profiling παίζει το αρχείο που θα χρησιμοποιήσουμε σαν είσοδο. Για να λάβουμε ρεαλιστικά και αξιόπιστα αποτελέσματα θα πρέπει το αρχείο εισόδου να πληροί κάποιες προϋποθέσεις όπως αυτές περιγράφονται στην παράγραφο 4.2. Στο αρχικό profiling που πραγματοποιήθηκε στον κώδικα αναφοράς, όπως επίσης και στις μετρήσεις απόδοσης που ακολουθούν κάθε βελτιστοποίηση του κώδικα χρησιμοποιείται το βίντεο **Input_Video.m2v**.

Μετά το τέλος του profiling του κώδικα αναφοράς προέκυψαν τα επόμενα δεδομένα :



Διάγραμμα 1: Profiling Κώδικα Αναφοράς

4.4 Saturate, Mismatch Control και Fast IDCT

4.4.1 Ανάλυση Ιδιοτήτων των Βίντεο για Βελτιστοποίηση

Δύο σημαντικά εργαλεία για τη βελτίωση της απόδοση μιας εφαρμογής αποτελούν, η χρησιμοποίηση χαρακτηριστικών γνωρισμάτων που μπορεί να έχουν τα αρχεία εισόδου (παράγραφος 3.2), καθώς και η αξιοποίηση ορισμένων ειδικών περιπτώσεις των αλγορίθμων που υλοποιούνται στα πλαίσια της εφαρμογής. Με αυτά μπορούμε να βελτιώσουμε την απόδοση μιας εφαρμογής είτε με απλές μεθόδους είτε με πιο σύνθετους τρόπους.

Χαρακτηριστικό στοιχείο της συμπίεσης βίντεο αποτελεί το Variable Length Coding (για τον κωδικοποιητή). Με αυτή τη μέθοδο (όπως περιγράφεται και στην παράγραφο 2.1.8) επιτυγχάνεται σημαντικό ποσοστό συμπίεσης. Μεγάλο πλεονέκτημα του VLC είναι πως μπορεί να κωδικοποιεί συνεχόμενους μηδενικούς DCT συντελεστές με μηδενικό κόστος. Γίνεται λοιπόν

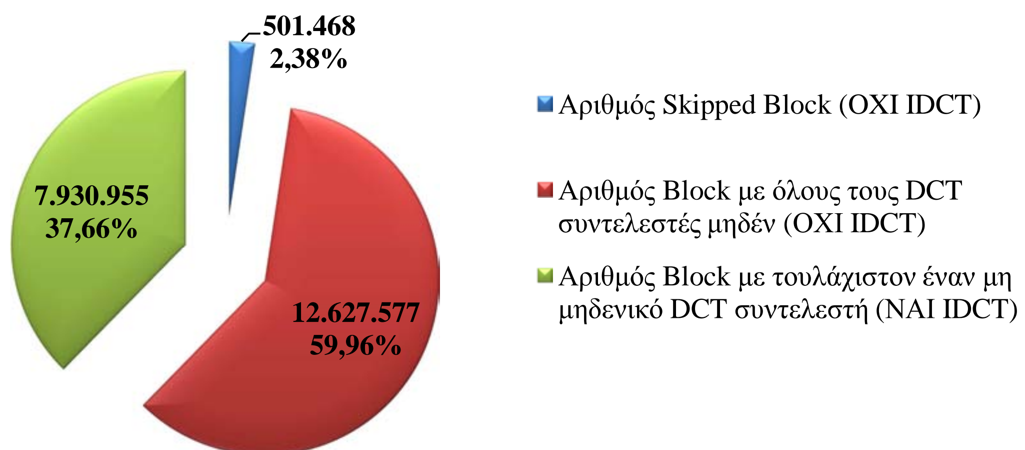
φανερό πως ο κωδικοποιητής προσπαθεί να παράγει όσο το δυνατόν περισσότερα συνεχόμενα μηδενικά στην κωδικοποίηση των blocks.

Ο αποκωδικοποιητής αντίστοιχα μπορεί μέσω της αντίστροφης διαδικασίας (Variable Length Decoding) να αποκωδικοποιεί με μηδενικό κόστος συνεχόμενους μηδενικούς DCT συντελεστές. Στη περίπτωση του αποκωδικοποιητή που χρησιμοποιείτε στην παρούσα εργασία (κώδικας αναφοράς), μετά από αυτό το στάδιο ακολουθείτε η συνήθη διαδικασία αποκωδικοποίησης χωρίς να λαμβάνονται υπόψη οι μηδενικοί συντελεστές DCT που προκύπτουν. Με αυτό τον τρόπο εκτελούνται τετριμμένες πράξεις και άσκοπες κλήσεις συναρτήσεων.

Οι παραπάνω παρατηρήσεις οδηγούν στα εξής συμπεράσματα :

- Σε block που υπάρχουν μόνο μηδενικοί DCT συντελεστές δεν είναι απαραίτητο να χρησιμοποιείται Inverse Discrete Cosine Transform καθώς το αποτέλεσμα θα είναι πάντα ένα block με μηδενικά.
- Αξιοποιώντας τους μηδενικούς DCT συντελεστές μπορούν να δημιουργηθούν ξεχωριστές συναρτήσεις IDCT για κάθε περίπτωση, που θα χρησιμοποιούν λιγότερες πράξεις.
- Δεν είναι απαραίτητο να εξετάζονται οι μηδενικοί συντελεστές κατά τη διαδικασία Saturating, καθώς δίνουν πάντα αποτέλεσμα μηδέν.
- Δεν είναι απαραίτητο να προστίθενται οι μηδενικοί συντελεστές στο τελικό άθροισμα των συντελεστών κατά τη διαδικασία Mismatch Control καθώς αποτελούν ουδέτερο στοιχείο και δεν αλλάζουν το αποτέλεσμα της πρόσθεσης.

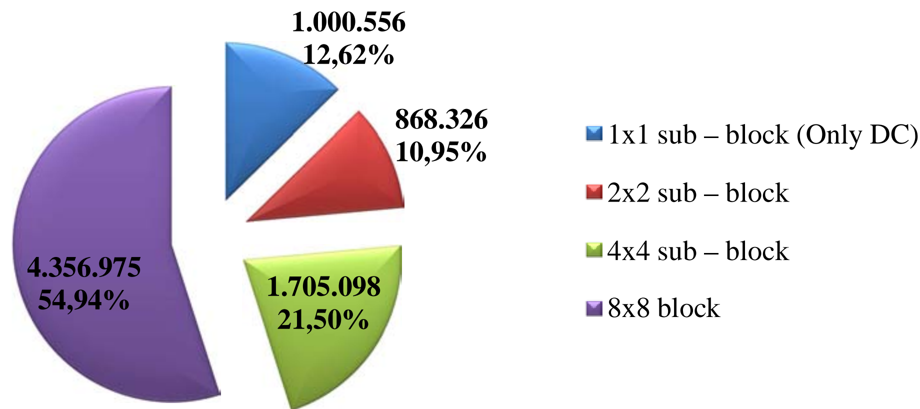
Χρησιμοποιώντας σαν είσοδο το αρχείο **Input_Inter.m2v** πραγματοποιήθηκαν οι παρακάτω μετρήσεις για τον αριθμό των block που απαιτούν IDCT. Τα αποτελέσματα που προέκυψαν είναι τα ακόλουθα :



Διάγραμμα 2: Είδη DCT block και χρήση IDCT

Ένα πολύ χρήσιμο χαρακτηριστικό των blocks που έχουν τουλάχιστον ένα μη μηδενικό συντελεστή είναι, το πώς αυτοί οι συντελεστές κατανέμονται στο block. Δηλαδή είναι σημαντικό να γνωρίζουμε αν αυτοί οι συντελεστές είναι συγκεντρωμένοι σε ένα μικρότερο sub – block διαστάσεων 1x1 , 2x2 ή 4x4. Θυμίζουμε πως μιλάμε για αρχικό block διαστάσεων 8x8.

Στο παρακάτω διάγραμμα φαίνεται πως τα **7,930,955 block** του **Input_Inter.m2v** που απαιτούν IDCT κατανέμονται ανάλογα με το sub – block στο οποίο είναι συγκεντρωμένοι οι μη μηδενικοί DCT συντελεστές:



Διάγραμμα 3: Κατανομή των block ανάλογα με τη συγκέντρωση των μη μηδενικών DCT συντελεστών στα sub – block

4.4.2 Saturate and Mismatch Control

Οι διαδικασίες Saturate και Mismatch Control περιγράφονται αναλυτικά στην παράγραφο 2.1.6. Στην παρούσα παράγραφο μελετώνται και βελτιώνονται μαζί, καθώς στον κώδικα αναφοράς υλοποιούνται στην ίδια συνάρτηση (Saturate). Επίσης, ο τρόπος βελτιστοποίησης που χρησιμοποιείται είναι ταυτόσημος και για τις δύο διαδικασίες, γεγονός που συνηγορεί στην παράλληλη βελτιστοποίηση τους.

Από το διάγραμμα 1 προκύπτει πως η συνάρτηση Saturate καταλαμβάνει την τρίτη θέση στο αρχικό profiling με ποσοστό 16.26%. Δηλαδή καταναλώνει 16.26% του συνολικού χρόνου εκτέλεσης. Το ποσοστό αυτό καθιστά αναγκαία την βελτιστοποίησής της.

Από την ανάλυση και τις μετρήσεις της παραγράφου 4.4.1 προκύπτει το συμπέρασμα πως εκτελούνται άσκοπες πράξεις κατά τη διάρκεια των Saturate και Mismatch Control. Αυτές

οι πράξεις αφορούν τους μηδενικούς DCT συντελεστές που εμφανίζονται σε μεγάλο ποσοστό στα DCT block, οι οποίοι αποτελούν ουδέτερα στοιχεία και στις δύο διαδικασίες.

Για την αποφυγή αυτών των πράξεων, οι διαδικασίες Saturate και Mismatch Control ενσωματώνονται στη διαδικασία αποκωδικοποίησης των DCT συντελεστών. Με τον τρόπο αυτό μόνο για τους μη μηδενικούς συντελεστές εκτελούνται οι πράξεις των δυο διαδικασιών.

Για τη διαδικασία Mismatch Control παρατηρούμε (όπως περιγράφεται στην παράγραφο 3.1) πως ελέγχεται αν το άθροισμα των DCT συντελεστών ενός block είναι περιττό ή άρτιο. Ο παραπάνω έλεγχος μπορεί να πραγματοποιηθεί εξετάζοντας μόνο το τελευταίο bit του αθροίσματος (0 για άρτιο και 1 για περιττό). Αυτή η πληροφορία μπορεί να δοθεί και από το τελευταίο bit της πράξης XOR όλων των συντελεστών. Με αυτό τον τρόπο η διαδικασία Mismatch Control βελτιώνεται ακόμη περισσότερο με την αντικατάσταση του τελικού if statement από λογικές πράξεις και του αθροίσματος από την πράξη XOR.

Για την εφαρμογή των βελτιστοποιήσεων, η συνάρτηση Saturate ενσωματώνεται στις συναρτήσεις Decode_MPEG2_Intra_Block και Decode_MPEG2_Non_Intra_Block του πηγαίου κώδικα. Επιπλέον αντικαθίσταται η εντολή *sum += val;* με την *xorval ^= val;* και η πράξη *if ((sum&1) == 0) { bp[63]^= 1; }* με την *bp[63] ^= (xorval&1) ^ 1;*.

Η προηγούμενη υλοποίηση μπορεί να εφαρμοστεί μόνο όταν δεν υπάρχει SNR scalable extension στο βίντεο προς αποκωδικοποίηση. Σε αντίθετη περίπτωση θα πρέπει να ολοκληρωθούν οι διαδικασίες αποκωδικοποίησης των DCT συντελεστών των σχετικών block και των δύο επιπέδων, να προστεθούν και έπειτα να συνεχιστεί η διαδικασία αποκωδικοποίησης του νέου πλέον block. Δηλαδή, δεν είναι εφικτό να εκτελεστούν οι Saturate και Mismatch Control κατά τη διάρκεια αποκωδικοποίησης των DCT συντελεστών.

Για να λυθεί αυτό το πρόβλημα δημιουργούνται δύο νέες συναρτήσεις, η Decode_MPEG2_Intra_Block_WithSNR και η Decode_MPEG2_Non_Intra_Block_WithSNR. Οι συναρτήσεις αυτές είναι ίδιες με αυτές που χρησιμοποιούνται στον κώδικα αναφοράς για την αποκωδικοποίηση των DCT συντελεστών και δεν ενσωματώνουν τη συνάρτηση Saturate. Αν υπάρχει SNR scalable extension καλούνται οι "_WithSNR" συναρτήσεις και η αποκωδικοποίηση συνεχίζεται σύμφωνα με την υλοποίηση του κώδικα αναφοράς.

Μετά την παραπάνω αλλαγή, ο έλεγχος για την ύπαρξη SNR scalable extension μεταφέρεται από το block στο macroblock. Δηλαδή, αντί να γίνεται ο έλεγχος αυτός για κάθε block του macroblock, γίνεται μία μόνο φορά για όλα τα blocks του macroblock. Με αυτό τον τρόπο ελαχιστοποιείτε η χρήση if statement στη συγκεκριμένη φάση της αποκωδικοποίησης, μειώνοντας έτσι τον αριθμό των εντολών.

Στον πηγαίο κώδικα αυτό μεταφράζεται ως εξής:

Στη συνάρτηση motion_compensation ο έλεγχος *if ((Two_Streams && enhan.scalable_mode == SC_SNR) || ld->MPEG2_Flag)* που βρίσκεται στο for loop το οποίο χρησιμοποιείται για την προσπέλαση των block κάθε macroblock τροποποιείται σε *if ((Two_Streams && enhan.scalable_mode == SC_SNR))* και μεταφέρεται έξω από το for loop. Ουσιαστικά με αυτό τον τρόπο ελέγχεται αν υπάρχει scalability και μάλιστα SNR scalability. Σε

αυτή την περίπτωση η διαδικασία της αποκωδικοποίησης συνεχίζεται χωρίς τις βελτιστοποιήσεις σε saturate και mismatch control (οι οποίες εκτελούνται με την κλήση της συνάρτησης saturate) και με τη χρήση των συναρτήσεων Decode_MPEG2_Intra_Block_WithSNR και Decode_MPEG2_Non_Intra_Block_WithSNR. Επίσης, ο αρχικός εσωτερικός έλεγχος στο for loop για την κλήση ή μη της συνάρτησης saturate θα έχει τη μορφή *if (ld->MPEG2_Flag).* Ο βελτιστοποιημένος κώδικας περιλαμβάνεται πλέον στο for loop του else statement.

4.4.3 Fast Inverse Discrete Cosine Transform (FIDCT)

Ο μετασχηματισμός **Inverse Discrete Cosine Transform** αποτελεί, όπως προκύπτει και από τον διάγραμμα 1 την πιο δαπανηρή διαδικασία της αποκωδικοποίησης. Ο ορισμός του IDCT περιγράφεται αναλυτικά στην παράγραφο 2.1.5. Στην πράξη, σπάνια χρησιμοποιούμε υλοποίηση του ορισμού σε πραγματικές εφαρμογές, λόγω της αυξημένης πολυπλοκότητας του αλγορίθμου. Η υλοποίηση του Mpeg2 Decoder που χρησιμοποιούμε σαν κώδικα αναφοράς, χρησιμοποιεί τον Chen-Wang algorithm (cf. IEEE ASSP-32, pp. 803-816, Aug. 1984) που με 11 πράξεις πολλαπλασιασμού και 29 πράξεις πρόσθεσης πετυχαίνει Inverse Discrete Cosine Transform 8 σημείων. Ωστόσο και αυτό το Fast IDCT, καταλαμβάνει την πρώτη θέση στο profiling ως η πιο δαπανηρή συνάρτηση.

4.4.3.1 Ελαχιστοποίηση της χρήσης FIDCT

Από τον διάγραμμα 2 προκύπτει το συμπέρασμα πως σχεδόν το 60% των block δεν χρειάζονται Inverse Discrete Cosine Transform. Στο κώδικα αναφοράς ωστόσο εκτελείτε IDCT για όλα τα blocks εκτός των skipped (blocks των skipped macroblock). Όπως περιγράφεται και στην παράγραφο 4.4.1 πρόκειται για block που περιέχουν μόνο μηδενικούς DCT συντελεστές.

Η άσκοπη χρήση του IDCT αποφεύγεται με την κλήση της κατάλληλης συνάρτησης αμέσως μετά το τέλος της αποκωδικοποίησης κάθε block. Με αυτό τον τρόπο μόνο τα block που χρειάζονται αποκωδικοποίηση των DCT συντελεστών θα χρησιμοποιήσουν IDCT. Δηλαδή, μόνο τα block που περιέχουν τουλάχιστον ένα μη μηδενικό συντελεστή.

Για την εφαρμογή της παραπάνω βελτιστοποίησης μεταφέρουμε την κλήση της IDCT από το for loop (προσοχή, το βελτιωμένο loop του else statement) της συνάρτησης motion_compensation στις συναρτήσεις Decode_MPEG2_Intra_Block και Decode_MPEG2_Non_Intra_Block. Πιο συγκεκριμένα στο κομμάτι του κώδικα των συναρτήσεων που σηματοδοτεί το τέλος της αποκωδικοποίησης των DCT συντελεστών ενός block. Δηλαδή, το

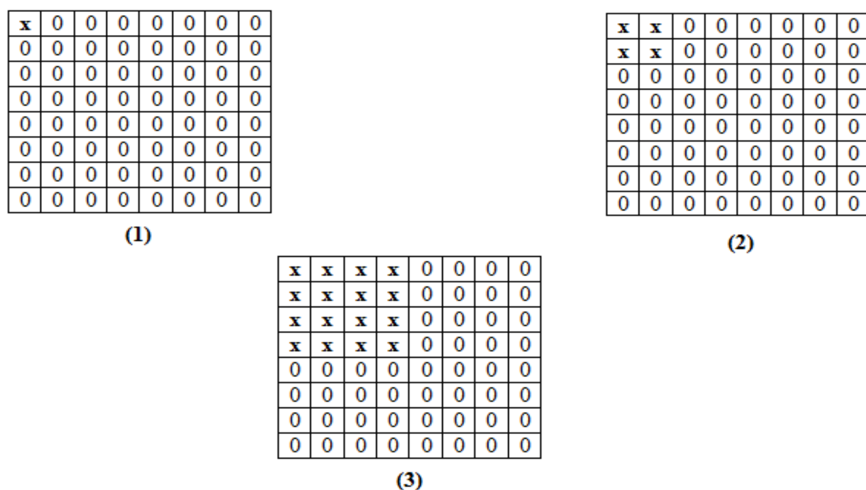
```
if (tab->run==64) /* end_of_block */
  {
        bp[63]^=(xorval&1)^1;
        return;
```

```
  }
γίνεται πλέον
if (tab->run==64) /* end_of_block */
  {
        bp[63]^=(xorval&1)^1;

      if (Reference_IDCT_Flag)
            Reference_IDCT(ld->block[comp]);
      else
            Fast_IDCT(ld->block[comp]);

      return;
  }
```

4.4.3.2 Περιγραφή και Υλοποίηση Βελτιωμένου Αλγορίθμου FIDCT

Από την παρατήρηση των αποτελεσμάτων του διάγραμμα 3 μπορούμε να συμπεράνουμε πως το 45.07% των blocks που απαιτούν Inverse Discrete Cosine Transform, έχουν τους μη μηδενικούς συντελεστές DCT συγκεντρωμένους σε ένα μικρότερο sub – block του αρχικού block. Αυτή η παρατήρηση μας οδηγεί στην δημιουργία μιας βελτιωμένης έκδοσης του αλγορίθμου των Chen-Wang που θα λαμβάνει υπόψη αυτή τη συγκέντρωση των συντελεστών σε μικρότερα blocks.

Για κάθε μία από τις περιπτώσεις του διαγράμματος 3, γνωρίζουμε εκ των προτέρων ποιοι DCT συντελεστές είναι σίγουρα μηδενικοί. Έτσι, αφαιρώντας ή απλοποιώντας εκείνες τις πράξεις που αυτοί οι συντελεστές εμφανίζονται ως τελεστές, επιτυγχάνεται η μείωση της πολυπλοκότητας του αλγορίθμου και η βελτίωση της απόδοσης του. Στην πραγματικότητα περιορίζουμε τον αλγόριθμο στο **1x1, 2x2 ή 4x4 sub-block** αντίστοιχα. Στην παρακάτω εικόνα (εικόνα 4.2) φαίνεται η μορφή του 8x8 block σε κάθε μία από τις παραπάνω περιπτώσεις.



Εικόνα 4.2: (1) Only DC (1x1) DCT block. (2) 2x2 DCT block. (3) 4x4 DCT block.

Εκτός από τη συγκέντρωση των μη μηδενικών συντελεστών σε sub – block υποχρεούμαστε να λάβουμε υπόψη και το αποτέλεσμα του mismatch control (περιγράφεται στην παράγραφο 2.1.6). Αν το άθροισμα των συντελεστών του block είναι άρτιο, ο τελευταίος DCT συντελεστής, που (όπως προκύπτει από την περιγραφή των παραπάνω περιπτώσεων) είναι πάντα 0, θα πάρει την τιμή 1. Αυτός ο παράγοντας εισάγει επιπλέον πολυπλοκότητα καθώς απαιτεί τον υπολογισμό του IDCT row για την τελευταία γραμμή, η οποία μετά το τέλος του υπολογισμού γεμίζει με μη μηδενικά ενδιάμεσα αποτελέσματα που πρέπει να χρησιμοποιηθούν στις πράξεις του IDCT column.

Από την παραπάνω διαδικασία προκύπτει πως για όλες τις περιπτώσεις συναρτήσεων IDCT mismatch1, η τελευταία γραμμή του block προς μετασχηματισμό θα είναι πάντα [0 0 0 0 0 0 0 1] με αποτέλεσμα IDCT row ίσο με [2 -6 9 -11 11 -9 6 -2]. Αποθηκεύοντας αυτό το αποτέλεσμα σε ένα πίνακα στη μνήμη, δεν χρειάζεται πλέον να εκτελείτε το IDCT row για την τελευταία γραμμή, καθώς το αποτέλεσμα μπορεί να φορτωθεί από τον πίνακα.

Στον πηγαίο κώδικα ο παραπάνω πίνακας ονομάζεται idct_table και ορίζεται σαν global μεταβλητή στο αρχείο idct.c ως *short idct_table[8];* Ενώ η αρχικοποίηση γίνεται στη συνάρτηση Initialize_Fast_IDCT με την εξής προσθήκη:

```
idct_table[0] = 2;
idct_table[1] = -6;
idct_table[2] = 9;
idct_table[3] = -11;
idct_table[4] = 11;
idct_table[5] = -9;
idct_table[6] = 6;
Idct_table[7] = -2;
```

Ειδικά για την περίπτωση που έχουμε 1x1 sub – block (Only DC), όταν ο μοναδικός μη μηδενικός συντελεστής είναι πολλαπλάσιος του 8, το block που προκύπτει μετά το IDCT είναι το ίδιο και για αποτέλεσμα mismatch 0 αλλά και για 1. Λαμβάνοντας υπόψη και αυτή την παρατήρηση μπορούμε να βελτιώσουμε την απόδοση για 1x1 IDCT.

Για την υλοποίηση του παραπάνω βελτιωμένου αλγορίθμου δημιουργούμε τις ακόλουθες συναρτήσεις :

- Full_Fast_IDCT : Είναι η συνάρτηση που χρησιμοποιεί ο κώδικας αναφοράς. Χρησιμοποιείται για 8x8 IDCT.
- Fast_IDCT_OnlyDC_MisMatch0 : Συνάρτηση για το 1x1 (OnlyDC) IDCT με αποτέλεσμα mismatch ίσο με 0 ή με αποτέλεσμα mismatch ίσο με 1 και DC πολλαπλάσιο του 8.
- Fast_IDCT_OnlyDC_MisMatch1 : Συνάρτηση για το 1x1 (OnlyDC) IDCT με αποτέλεσμα mismatch ίσο με 1.
- Fast_IDCT_2x2_MisMatch0 : Συνάρτηση για το 2x2 IDCT με αποτέλεσμα mismatch ίσο με 0.

- Fast_IDCT_2x2_MisMatch1 : Συνάρτηση για το 2x2 IDCT με αποτέλεσμα mismatch ίσο με 1.
- Fast_IDCT_4x4_MisMatch0 : Συνάρτηση για το 4x4 IDCT με αποτέλεσμα mismatch ίσο με 0.
- Fast_IDCT_4x4_MisMatch1 : Συνάρτηση για το 4x4 IDCT με αποτέλεσμα mismatch ίσο με 1.

Για την επιλογή της κατάλληλης συνάρτησης, κατά τη διάρκεια της αποκωδικοποίησης των DCT συντελεστών κάθε block, για όλους τους μη μηδενικούς συντελεστές, εκτελείτε η παρακάτω πράξη :

orval |= ((pos_in_block >> 3) | (pos_in_block & 7)) : Όπου pos_in_block είναι η θέση του συντελεστή στο 8x8 block (παίρνει τιμές από 0 έως 64).

Το αποτέλεσμα αυτής της πράξης είναι από 0 έως 7. Με βάση το αποτέλεσμα αυτό γίνεται η επιλογή του IDCT που θα χρησιμοποιηθεί. Για τις διάφορες τιμές της orval οι περιπτώσεις είναι :

1. orval = 0 : Επιλογή 1x1 IDCT.
2. orval = 1 : Επιλογή 2x2 IDCT.
3. orval = 2 και orval = 3: Επιλογή 4x4 IDCT.
4. Αλλιώς (orval = 4… 7) : Επιλογή 8x8 IDCT (Full Fast IDCT).

Για την επιλογή ανάμεσα στο mismatch 0 και 1 ελέγχουμε την τιμή του τελευταίου συντελεστή του block.

Τέλος, γίνεται και ο έλεγχος για το αν ο πρώτος συντελεστής (DC) είναι πολλαπλάσιος του 8. Όπως αναφέρθηκε παραπάνω, η παράμετρος αυτή χρησιμοποιείται μόνο στην περίπτωση του 1x1 IDCT. Για τον υπολογισμό της παραμέτρου αυτής εκτελούμε την :

DC_mod8 = DC_συντελεστής & 7: Η πράξη αυτή για θετικούς ακέραιους δίνει το υπόλοιπο της διαίρεσης του DC με το 8. Για αρνητικούς ακέραιους αριθμούς δίνει ένα αποτέλεσμα από 0 έως 7 χωρίς αναγκαία αυτό, να είναι και το υπόλοιπο της διαίρεσης του DC με το 8. Ωστόσο οποιοσδήποτε ακέραιος πολλαπλάσιος του 8 θα έχει Integer & 7 ίσο με 0.

Συνδυάζοντας τα αποτελέσματα των παραπάνω πράξεων επιλέγετε η κατάλληλη συνάρτηση. Για την επιλογή αυτή ορίζετε ένας global Function Pointer Table στον οποίο αποθηκεύονται, στις κατάλληλες θέσεις, οι δείκτες των συναρτήσεων IDCT. Με αυτό τον τρόπο αποφεύγονται τα πολλαπλά και δαπανηρά if statements.

Ο παραπάνω πίνακας ορίζεται σαν global στο αρχείο global.h με τον εξής τρόπο:

```
typedef void (*ptrfunc)(short *);
EXTERN ptrfunc FIDCT[8*2*8];
```

Για την επιλογή της κατάλληλης συνάρτησης χρησιμοποιούνται τρεις παράγοντες που απαιτούν ένα τρισδιάστατο πίνακα. Ωστόσο για να αποφύγουμε τις πολλές προσπελάσεις μνήμης και να περιορίσουμε το κόστος σε αριθμητικές και λογικές πράξεις , ο πίνακας FIDCT ορίζεται σαν μονοδιάστατος, μεγέθους 8*2*8. Για την επιλογή της κατάλληλης συνάρτησης

μέσω του πίνακα χρησιμοποιείται η παράσταση ***(orval<<4) + ((bp[63]&1)<<3) + val_mod8.*** Εκτελείται η πράξη ***bp[63]&1*** ώστε να μην ξεφύγουμε από τα όρια του πίνακα. Ο πίνακας FIDCT αρχικοποιείται στη συνάρτηση Initialize_Decoder πριν την κλήση της συνάρτησης Initialize_Fast_IDCT με το εξής κομμάτι κώδικα:

```
for(i=0; i<9; i++)
        FIDCT[i] = &Fast_IDCT_OnlyDC_MisMatch0; //(orval == 0, bp[63]&1 ==     0, val_mod8 E [0,7]) ||
(orval == 0, bp[63] == 1, val_mod8==0)
for(i=9; i<16; i++)
        FIDCT[i] = &Fast_IDCT_OnlyDC_MisMatch1; //orval == 0, bp[63]&1 == 1, val_mod8 != 0
for(i=16; i<24; i++)
        FIDCT[i] = &Fast_IDCT_2x2_MisMatch0; //orval == 1, bp[63]&1 == 0, val_mod8 E [0,7]
for(i=24; i<32; i++)
        FIDCT[i] = &Fast_IDCT_2x2_MisMatch1; //orval == 1, bp[63]&1 == 1, val_mod8 E [0,7]
for(i=32; i<40; i++)
        FIDCT[i] = &Fast_IDCT_4x4_MisMatch0; //orval == 2, bp[63]&1 == 0, val_mod8 E [0,7]
for(i=40; i<48; i++)
        FIDCT[i] = &Fast_IDCT_4x4_MisMatch1; //orval == 2, bp[63]&1 == 1, val_mod8 E [0,7]
for(i=48; i<56; i++)
        FIDCT[i] = &Fast_IDCT_4x4_MisMatch0; //orval == 3, bp[63]&1 == 0, val_mod8 E [0,7]
for(i=56; i<64; i++)
        FIDCT[i] = &Fast_IDCT_4x4_MisMatch1; //orval == 3, bp[63]&1 == 1, val_mod8 E [0,7]
for(i=64; i<8*2*8; i++)
        FIDCT[i] = &Full_Fast_IDCT; //orval > 3
```

Από τα σχόλια του παραπάνω κώδικα προκύπτει και ο τρόπος επιλογής της κατάλληλης συνάρτησης.

Περεταίρω βελτίωση στο IDCT επιτυγχάνεται ενοποιώντας τις συναρτήσεις για row και column. Με αυτό τον τρόπο δημιουργείτε μία συνάρτηση που στην αρχή εκτελεί το κατά γραμμές και στη συνέχεια το κατά στήλες IDCT. Έτσι ελαττώνονται οι περιττές κλήσεις συναρτήσεων (8 κλήσεις IDCT row και 8 κλήσεις IDCT column για κάθε block) καθώς με τη βελτίωση αυτή η Fast_IDCT καλείται μία φορά και εκτελεί 8x8 IDCT δύο διαστάσεων.

4.4.3.3 Χρήση NEON instructions

Οι SIMD εντολές αποτελούν ένα πολύ σημαντικό εργαλείο στη Βελτιστοποίηση Λογισμικού όπως περιγράφεται και στην παράγραφο 3.3. Η χρήση ωστόσο των εντολών αυτών προϋποθέτει την εύρεση των σημείων του πηγαίου κώδικα όπου αυτές μπορούν να χρησιμοποιηθούν αποδοτικά. Ένα τέτοιο σημείο στην περίπτωση μας αποτελεί το IDCT.

Χρησιμοποιώντας τις εντολές NEON που παρέχονται από την αρχιτεκτονική ARM Cortex A8 οι συναρτήσεις του Βελτιωμένου Αλγορίθμου IDCT υλοποιούνται ως εξής :

- Full_Fast_IDCT : Αποκλειστική χρήση NEON.
- Fast_IDCT_OnlyDC_MisMatch0 : Αποκλειστική χρήση NEON.
- Fast_IDCT_OnlyDC_MisMatch1 : Αποκλειστική χρήση NEON.
- Fast_IDCT_2x2_MisMatch0 : Χρήση κώδικα C για το IDCT row και χρήση NEON για το IDCT column.

- Fast_IDCT_2x2_MisMatch1 : Χρήση κώδικα C για το IDCT row και χρήση NEON για το IDCT column.
- Fast_IDCT_4x4_MisMatch0 : Αποκλειστική χρήση NEON.
- Fast_IDCT_4x4_MisMatch1 : Αποκλειστική χρήση NEON.

Ένα σημαντικό πρόβλημα που προέκυψε από τη χρήση NEON εντολών στο IDCT είναι η εύρεση ενός αποδοτικού τρόπου φόρτωσης των δεδομένων από τη μνήμη στα NEON διανύσματα.

Κάθε block DCT συντελεστών αποτελεί στην ουσία ένα πίνακα μεγέθους 64 = 8x8. Ο πίνακας αποθηκεύεται στη μνήμη κατά γραμμές. Ο IDCT 2 διαστάσεων που χρησιμοποιείται στον standard Mpeg2 εκτελεί πρώτα IDCT κατά γραμμές (row) και στη συνέχεια κατά στήλες (column). Ωστόσο, για να εκτελεστεί το IDCT row με τη χρήση διανυσμάτων (NEON) πρέπει το block να φορτωθεί στα διανύσματα κατά στήλες.

Μία αποδοτική λύση για το παραπάνω πρόβλημα είναι οι DCT συντελεστές, κατά την αποκωδικοποίησης, να αποθηκεύονται με τέτοιο τρόπο, ώστε κάθε γραμμή του block να περιέχει την αντίστοιχη στήλη.

Η παραπάνω λειτουργία επιτυγχάνεται με την εκτέλεση της παρακάτω εντολής για κάθε μη μηδενικό συντελεστή:

```
new_pos = ( (  prev_pos & 7) <<3  ) + ( prev_pos >> 3)  )
```

Όσον αφορά την υλοποίηση των IDCT συναρτήσεων με NEON εντολές, ακολουθήθηκε η εξής διαδικασία για κάθε μία από αυτές:

Στην συνάρτηση **Full_Fast_IDCT** αρχικά φορτώνεται το block από τη μνήμη κατά γραμμές (στις οποίες είναι αποθηκευμένες οι στήλες του block όπως περιγράφηκε παραπάνω) με τη χρήση δύο προσωρινών μεταβλητών τύπου int16x8_t . Για τη φόρτωση χρησιμοποιείται η εντολή vld1q_s16. Στη συνέχεια από το vector αυτό φορτώνονται τα κύρια vector (τύπου int32x4_t) που χρησιμοποιούνται στην υλοποίηση του αλγορίθμου. Η πράξη αυτή γίνεται μέσω των εντολών vget_low_s16 και vget_high_s16 οι οποίες σπάζουν το 128 bit vector σε δύο 64 bit vector, και της εντολής vmovl_s16 που τα μετασχηματίζει σε 128 bit vector. Αυτή η διαδικασία είναι απαραίτητη επειδή τα ενδιάμεσα αποτελέσματα του αλγορίθμου Fast IDCT απαιτούν ακρίβεια μεγαλύτερη από 16 bit. Ενδεικτικός κώδικας είναι ο παρακάτω :

```
va_s16x8 = vld1q_s16(block);
x01_s32x4 = vmovl_s16(vget_low_s16(va_s16x8));
x02_s32x4 = vmovl_s16(vget_high_s16(va_s16x8));
x01_s32x4 = vshlq_n_s32(x01_s32x4, 11);
x02_s32x4 = vshlq_n_s32(x02_s32x4, 11);
```

Μετά την εκτέλεση του αλγορίθμου ακολουθείται η αντίστροφη διαδικασία για την αποθήκευση των vector αποτελέσματος πίσω στο block. Οι εντολές που χρησιμοποιούνται σε αυτή την περίπτωση είναι η vqmovn_s32 που μετασχηματίζει το 128 bit vector σε 64 bit vector εκτελώντας ταυτόχρονα και saturate, η vcombine_s16 που συνδυάζει δύο 64 bit vector σε ένα

128 bit vector και η vst1q_s16 που αποθηκεύει τα περιεχόμενα του vector στη μνήμη. Το saturate στο διάστημα [-256, 255] των αποτελεσμάτων του IDCT που απαιτείται πριν την αποθήκευση τους πραγματοποιείται μέσω των εντολών vmaxq_s16 και vminq_s16 οι οποίες εκτελούν τις πράξεις VecResult[i] := (VecA[i] >= VecB[i]) ? VecA[i] : VecB[i] και VecResult[i] := (VecA[i] >= VecB[i]) ? VecB[i] : VecA[i]. Ο ενδεικτικός κώδικας είναι:

```
temp_s32x4 = vaddq_s32(x71_s32x4, x11_s32x4);
temp_s32x4 = vshrq_n_s32(temp_s32x4, 14);
x01_s16x4 = vqmovn_s32(temp_s32x4);

temp_s32x4 = vaddq_s32(x72_s32x4, x12_s32x4);
temp_s32x4 = vshrq_n_s32(temp_s32x4, 14);
x02_s16x4 = vqmovn_s32(temp_s32x4);

x_s16x8 = vcombine_s16(x01_s16x4, x02_s16x4);
x_s16x8 = vmaxq_s16(x_s16x8, const_s16x8);
x_s16x8 = vminq_s16(x_s16x8, const2_s16x8);
vst1q_s16(block, x_s16x8);
```

Για τις πράξεις που πραγματοποιούνται στα πλαίσια του Fast IDCT χρησιμοποιούνται κυρίως εντολές πρόσθεσης, αφαίρεσης, πολλαπλασιασμού και ολίσθησης. Ωστόσο χρησιμοποιούνται και εντολές που δεν κατατάσσονται στις εντολές απλής πράξεις όπως:

```
int32x4_t vmlaq_s32(int32x4_t a, int32x4_t b, int32x4_t c) : Vr[i] := Va[i] + Vb[i] * Vc[i]
int32x4_t vmlal_s16(int32x4_t a, int16x4_t b, int16x4_t c) : Vr[i] := Va[i] + Vb[i] * Vc[i]
```

Ένα σημαντικό πρόβλημα που προκύπτει από τη χρήση NEON vector είναι ο τρόπος με τον οποίο θα αντιστραφούν τα vector ώστε από το IDCT row να μεταβούμε στο IDCT column. Το πρόβλημα αυτό επιλύεται με τη χρήση των εντολών vzipq_s16. Οι εντολές αυτές συνδυάζουν δύο vector ίδιου τύπου, κάνοντας interleaving (εναλλάξ σάρωση των στοιχείων των δυο vector) τα στοιχεία τους και αποθηκεύουν το αποτέλεσμα σε μεταβλητές τύπου vector struct όπως π.χ.

```
struct int16x4x2_t
{
	int16x4_t val[2];
};
```

Ο κώδικας που υλοποιεί την παραπάνω λειτουργία τη συνάρτηση Full_Fast_IDCT είναι:

```
//Transpose block
v0_s16x8x2 = vzipq_s16(v0_s16x8x2.val[0], v0_s16x8x2.val[1]); // (0) 1 5 1 5 1 5 1 5  (1) 1 5 1 5 1 5 1 5
v1_s16x8x2 = vzipq_s16(v1_s16x8x2.val[0], v1_s16x8x2.val[1]); // (0) 3 7 3 7 3 7 3 7  (1) 3 7 3 7 3 7 3 7

v6_s16x8x2 = vzipq_s16(v6_s16x8x2.val[0], v6_s16x8x2.val[1]); // (0) 0 4 0 4 0 4 0 4  (1) 0 4 0 4 0 4 0 4
v7_s16x8x2 = vzipq_s16(v7_s16x8x2.val[0], v7_s16x8x2.val[1]); // (0) 2 6 2 6 2 6 2 6  (1) 2 6 2 6 2 6 2 6

v4_s16x8x2 = vzipq_s16(v0_s16x8x2.val[0], v1_s16x8x2.val[0]); // (0) 1 3 5 7 1 3 5 7  (1) 1 3 5 7 1 3 5 7

v5_s16x8x2 = vzipq_s16(v6_s16x8x2.val[0], v7_s16x8x2.val[0]); // (0) 0 2 4 6 0 2 4 6  (1) 0 2 4 6 0 2 4 6

v2_s16x8x2 = vzipq_s16(v5_s16x8x2.val[0], v4_s16x8x2.val[0]); // (0) 0 1 2 3 4 5 6 7  (1) 0 1 2 3 4 5 6 7
v3_s16x8x2 = vzipq_s16(v5_s16x8x2.val[1], v4_s16x8x2.val[1]); // (0) 0 1 2 3 4 5 6 7  (1) 0 1 2 3 4 5 6 7
```

```
v4_s16x8x2 = vzipq_s16(v0_s16x8x2.val[1], v1_s16x8x2.val[1]); // (0) 1 3 5 7 1 3 5 7  (1) 1 3 5 7 1 3 5 7

v5_s16x8x2 = vzipq_s16(v6_s16x8x2.val[1], v7_s16x8x2.val[1]); // (0) 0 2 4 6 0 2 4 6  (1) 0 2 4 6 0 2 4 6

v6_s16x8x2 = vzipq_s16(v5_s16x8x2.val[0], v4_s16x8x2.val[0]); // (0) 0 1 2 3 4 5 6 7  (1) 0 1 2 3 4 5 6 7
v7_s16x8x2 = vzipq_s16(v5_s16x8x2.val[1], v4_s16x8x2.val[1]); // (0) 0 1 2 3 4 5 6 7  (1) 0 1 2 3 4 5 6 7
```

Η συνάρτηση **Fast_IDCT_OnlyDC_MisMatch0** μετά την εφαρμογή του βελτιωμένου αλγορίθμου, καταλήγει στην αντιγραφή μιας συγκεκριμένης τιμής, που προκύπτει από πράξεις στον DC συντελεστή, σε όλες τις θέσεις του block. Η υλοποίησή της φαίνεται παρακάτω:

```
void Fast_IDCT_OnlyDC_MisMatch0(short *block)
{
	int val , i;

	val = iclp[(block[0]+4)>>3];
	for(i=0;i<64;i++)
	{
		block[i] = val;
	}
}
```

Αλλάζοντας την παραπάνω υλοποίηση ώστε η τιμή iclp[(block[0]+4)>>3] να φορτώνεται σε ένα vector το οποίο θα αποθηκεύεται σε όλες τις γραμμές του block προκύπτει η παρακάρω συνάρτηση:

```
void Fast_IDCT_OnlyDC_MisMatch0(short *block){

	int16x8_t va_s16x8;

	va_s16x8 = vdupq_n_s16(iclp[(block[0]+4)>>3]);

	vst1q_s16(block, va_s16x8);
	vst1q_s16(block+8, va_s16x8);
	vst1q_s16(block+16, va_s16x8);
	vst1q_s16(block+24, va_s16x8);
	vst1q_s16(block+32, va_s16x8);
	vst1q_s16(block+40, va_s16x8);
	vst1q_s16(block+48, va_s16x8);
	vst1q_s16(block+56, va_s16x8);
}
```

Στη συνάρτηση **Fast_IDCT_OnlyDC_MisMatch1** χρησιμοποιείται παρόμοια λογική με αυτή της συνάρτησης Full_Fast_IDCT όσον αφορά την αποθήκευση των vector. Η φόρτωση των δεδομένων για το IDCT row απαιτεί τις load vector πράξεις που φαίνονται παρακάτω (οι οποίες αποτελούν ταυτόχρονα και το αποτέλεσμα του IDCT row):

```
va_s16x8 = vdupq_n_s16(block[0]<<3);
vb_s16x8 = vld1q_s16(idct_table);
```

Στη συνέχεια, με τον ίδιο τρόπο που περιγράφηκε στην Full_Fast_IDCT τα αποτελέσματα του IDCT row από 16 bit μετασχηματίζονται σε 32 bit. Η διαδικασία συνεχίζεται με την εκτέλεση των πράξεων του IDCT column.

Στη συνάρτηση **Fast_IDCT_2x2_MisMatch0** το IDCT row εκτελείται με τη χρήση του πηγαίου κώδικα C, καθώς η χρήση NEON εντολών είναι ασύμφορη (οι μη μηδενικοί συντελεστές δεν είναι περισσότεροι από δύο σε κάθε γραμμή). Τα αποτελέσματα του IDCT column αποθηκεύονται απευθείας στην κατάλληλη θέση των vector που θα χρησιμοποιηθούν στη συνέχεια για την εκτέλεση του IDCT column μέσω της εντολής vsetq_lane_s32 η οποία αποθηκεύει την τιμή ενός 32 bit ακέραιου στην θέση vector. Το vector, η θέση του και η 32 bit τιμή αποτελούν τους τελεστές της πράξης.

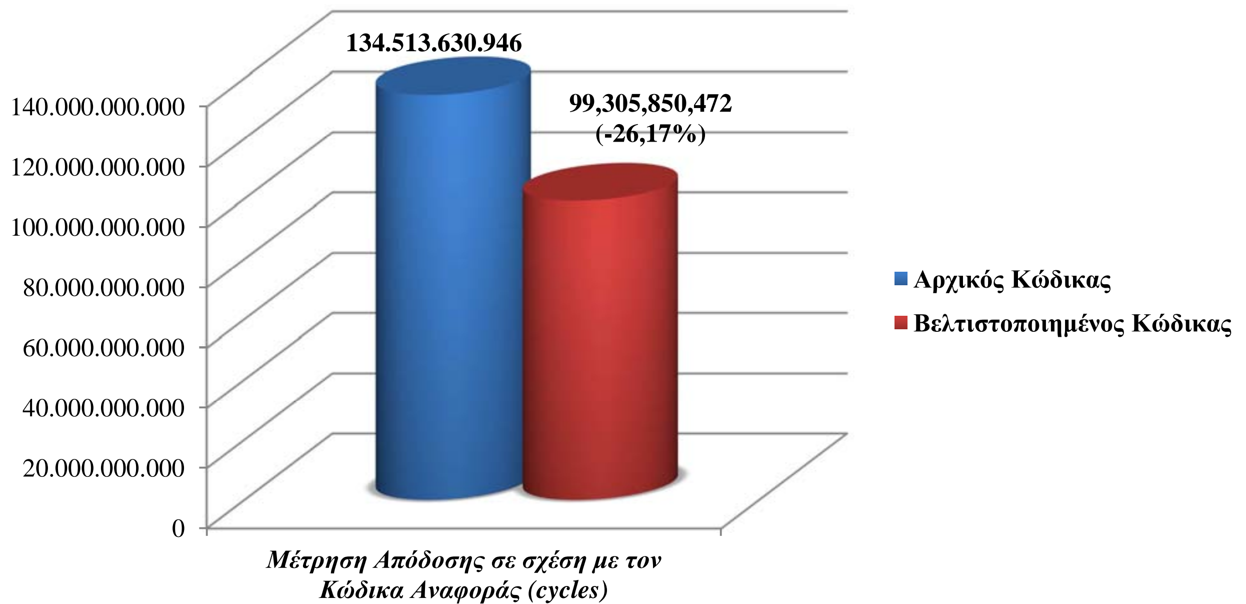
Η υλοποίηση της συνάρτησης **Fast_IDCT_2x2_MisMatch1** είναι ίδια με αυτή της Fast_IDCT_2x2_MisMatch0. Η διαφορά είναι πως μετά την IDCT row προκύπτουν μη μηδενικά αποτελέσματα και στην τελευταία γραμμή του block, τα οποία φορτώνονται σε vector για την IDCT (περίπτωση αποτελέσματος mismatch ίσο με 1 και φόρτωση αποτελεσμάτων από το πίνακ idct_table). Η IDCT column εκτελείται όπως και στη Fast_IDCT_2x2_MisMatch0 με τη διαφορά πως οι εκτελέσιμες πράξεις αυξάνονται και μερικές τροποποιούνται.

Η συνάρτηση **Fast_IDCT_4x4_MisMatch0** υλοποιείται όμοια με την **Full_Fast_IDCT** με τη διαφορά πως χρησιμοποιείται ένα vector (τύπου int32x4_t) για κάθε γραμμή στο οποίο φορτώνονται οι τέσσερις πρώτη συντελεστές των τεσσάρων πρώτων γραμμών, καθώς οι υπόλοιποι DCT συντελεστές είναι μηδενικοί. Επίσης, λαμβάνοντας υπόψη τους μηδενικούς συντελεστές γίνονται και οι κατάλληλες τροποποιήσεις – αφαιρέσεις των εκτελέσιμων πράξεων. Και σε αυτή την περίπτωση είναι απαραίτητο το transpose των vector για την έναρξη της IDCT column.

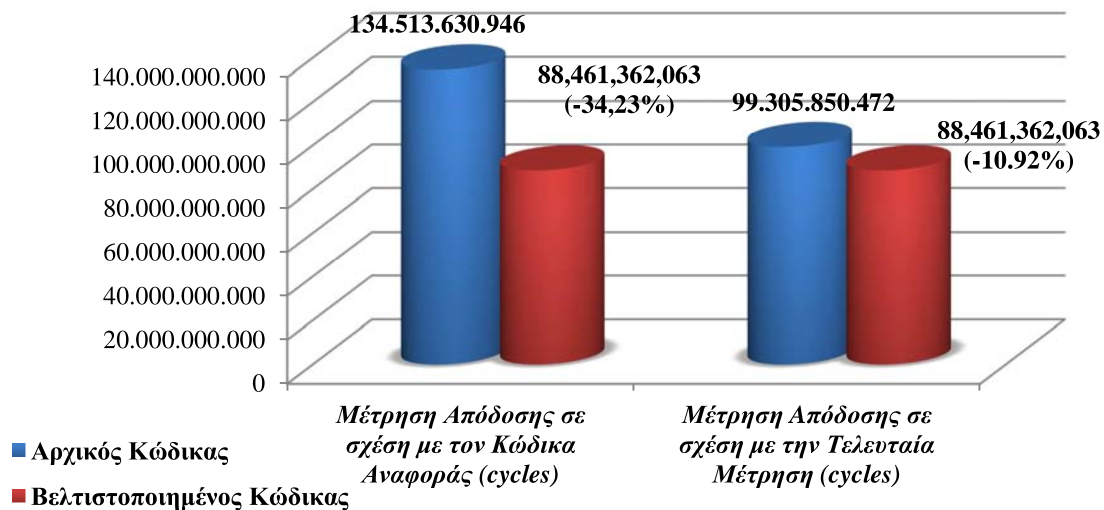
Τέλος η συνάρτηση **Fast_IDCT_4x4_MisMatch1** υλοποιείται με τον τρόπο υλοποίησης της **Fast_IDCT_4x4_MisMatch0,** λαμβάνοντας υπόψη και την τελευταία γραμμή του block στην υλοποίηση της IDCT column.

4.4.4 Μέτρηση Απόδοσης – Νέο Profiling

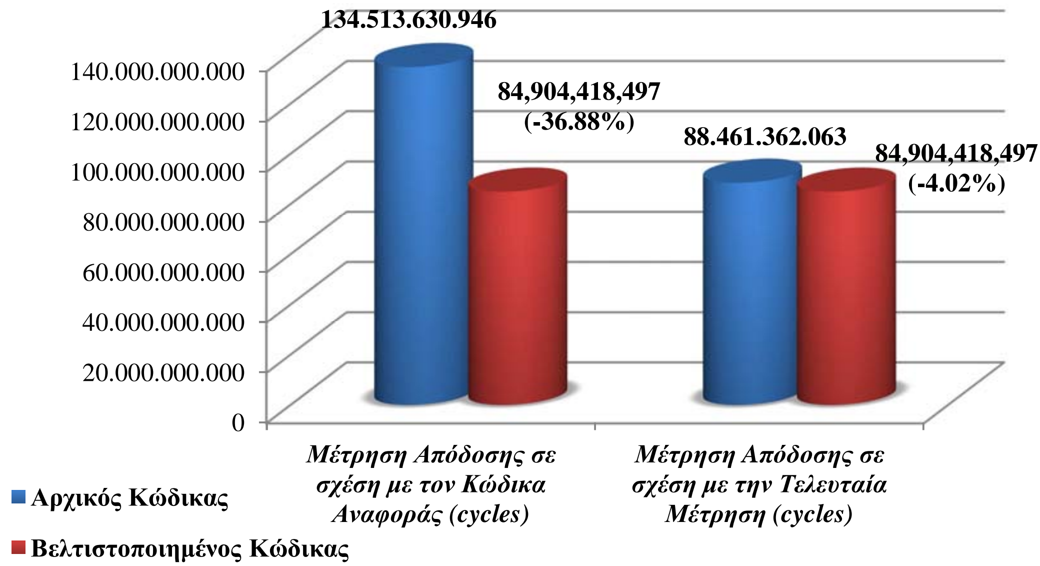
Τα αποτελέσματα που προκύπτουν όσον αφορά τη βελτίωση της απόδοσης μετά από τις βελτιστοποιήσεις της παρούσας παραγράφου παρουσιάζονται στα παρακάτω διαγράμματα :



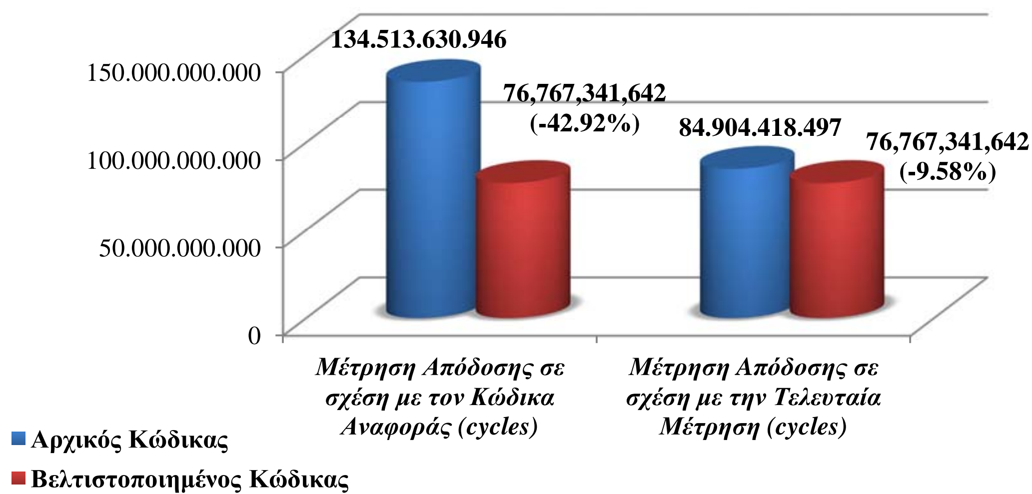
Διάγραμμα 4 : Μέτρηση Απόδοσης για τη Βελτιστοποίηση των Saturate και Mismatch Control



Διάγραμμα 5 : Μέτρηση Απόδοσης Μετά την Ελαχιστοποίηση Χρήσης IDCT

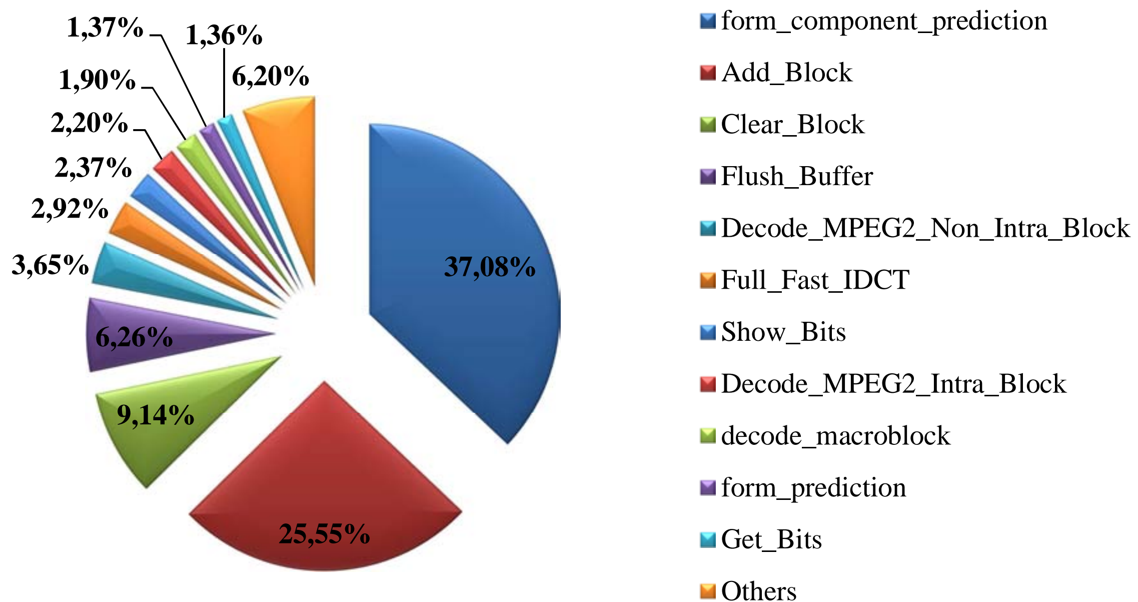


Διάγραμμα 6 : Μέτρηση Απόδοσης Μετά την Υλοποίηση του Βελτιωμένου Αλγορίθμου IDCT



Διάγραμμα 7 : Μέτρηση Απόδοσης Μετά τη Χρήση Εντολών NEON στη διαδικασία IDCT

Μετά τις βελτιστοποιήσεις σε Saturate, Mismatch Control και Fast IDCT τα αποτελέσματα του profiling φαίνονται στο παρακάτω διάγραμμα :



Διάγραμμα 8 : Profiling with Saturate, Mismatch Control and FIDCT optimized

4.5 Motion Compensation

Η διαδικασία Motion Compensation περιγράφεται αναλυτικά στην παράγραφο 2.1.4. Το Motion Compensation υλοποιείτε στον κώδικα αναφοράς από τη συνάρτηση form_component_prediction. Από τα διαγράμματα 1 και 8 προκύπτει το συμπέρασμα πως η διαδικασία αυτή αποτελεί μία πολύ δαπανηρή διαδικασία, καθώς στο αρχικό profiling καταλαμβάνει τη δεύτερη θέση με ποσοστό 21.16% ενώ στο τελευταίο profiling καταλαμβάνει την πρώτη θέση με ποσοστό 37.08%.

4.5.1 Χρήση NEON instructions

Το Motion Compensation αποτελεί μία διαδικασία που μπορούν να χρησιμοποιηθούν αποδοτικά NEON εντολές. Οι πράξεις που απαιτεί το Motion Compensation εκτελούνται για

όλα τα στοιχεία ενός macroblock 16x16 ή ενός block 16x8 του reference frame (ή field), ανάλογα με τον τύπο του motion vector που χρησιμοποιείται. Οι παραπάνω τύποι motion vector αναφέρονται στο luminance. Για το chrominance τα motion vectors αντιστοιχούν σε 16x16 block για chroma format 4:4:4 και σε 8x8 block για chroma format 4:2:2 και 4:2:0.

Για να υλοποιηθεί το Motion Compensation με NEON εντολές απαιτείται να γνωρίζουμε εκ των προτέρων το width του block. Αυτό συμβαίνει επειδή πρέπει να είναι γνωστό τι μεγέθους NEON vectors θα χρησιμοποιηθούν. Υπενθυμίζεται πως στη NEON unit υπάρχουν 64-bit και 128-bit registers και πως στο standard MPEG-2 το κάθε στοιχείο του block αναπαριστάται με 8 bits.

Συνεπώς, για block με width 16 χρησιμοποιείται 128-bit vector, στο οποίο φορτώνεται κάθε γραμμή του block, ενώ αντίστοιχα για block με width 8 χρησιμοποιείται 64-bit vector με τον ίδιο τρόπο. Τελικά, για να υποστηριχθεί η παραπάνω λειτουργικότητα συνάρτηση form_component_prediction αντικαθίσταται από δύο νέες συναρτήσεις (για width 8 και για width 16), τις form_component_predictionWidth16 και form_component_predictionWidth8. Επίσης μειώνουμε το πλήθος των arguments που αυτές δέχονται αφαιρώντας το width argument, καθώς προκύπτει απευθείας από την κλήση της αντίστοιχης συνάρτησης.

Η υλοποίηση των συναρτήσεων form_component_predictionWidth16 και form_component_predictionWidth8 ακολουθέι την ίδια λογική με τη μόνη διαφοράς στο μέγεθος των vector που χρησιμοποιούνται. Στη Width16 χρησιμοποιούνται vectors τύπου uint8x16_t ενώ στη Width8 vectors τύπου uint8x8_t. Η κύριες εντολές που χρησιμοποιούνται πέρα από της εντολές load, store και τις εντολές απλών πράξεων είναι οι :

vhadd_u8(uint8x8_t a, uint8x8_t b);
vhaddq_u8(uint8x16_t a, uint8x16_t b);

οι οποίες εκτελούν την πράξη *(Va[i]+Vb[i])>>1* και αποτελούν σημαντικό εργαλείο στις περιπτώσεις παρεμβολής μισού pixel.

Ιδιαίτερες περιπτώσεις στις δυο αυτές συναρτήσεις αποτελούν οι ταυτόχρονες παρεμβολές οριζοντίως και καθέτως. Στην περίπτωση της form_component_predictionWidth16 συνάρτησης υπάρχει περίπτωση υπερχείλισης των αποτελεσμάτων καθώς η πράξη που απαιτείται είναι η *(a + b + c + d + 2)>>2* όπου a, b , c , d οι τιμές των γειτονικών pixel του σημείου που θέλουμε να παρεμβάλουμε. Το πρόβλημα αυτό επιλύεται με την εξής μέθοδο :

1. Φορτώνουμε όλα τα απαραίτητα δεδομένα από τη μνήμη στους NEON registers.
2. Με τη χρήση της εντολής vandq_u8 (λογικό AND) και της mask 0x3 αποθηκεύονται σε διαφορετικά vector στη μνήμη τα less significant bits, κάθε τιμής των vector. Παράδειγμα αποτελεί η εντολή :
   lsb1_u8x16 = vandq_u8(s1_u8x16, mask_u8x16);
3. Με τα vector αυτά υπολογίζουμε τα κρατούμενα της πρόσθεσης των αρχικών vector μέσω απλών πράξεων πρόσθεσης.
4. Στη συνέχεια αφού τα αρχικά vector έχουν απαλλαγεί από τα 2 less significant bits, τα ολισθαίνουμε κατά 2 bits δεξιά με την εντολή vshrq_n_u8 που ολισθαίνει όλα τα στοιχεία ενός vector κατά δυο bits δεξιά.

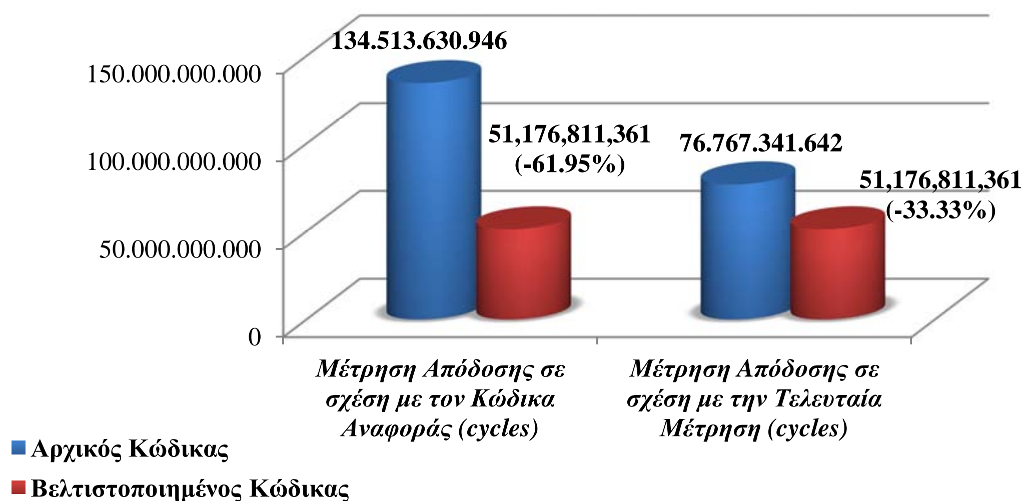
5. Οι πράξεις πρόσθεσης εκτελούνται κανονικά πλέον χωρίς το φόβο υπερχειλίσεις και στη συνέχεια στο αποτέλεσμα προστίθενται και τα κρατούμενα της πρόσθεσης από το βήμα 3.

Στη συνάρτηση form_component_predictionWidth16 το πρόβλημα επιλύεται με πιο απλό τρόπο καθώς οι οκτώ 8bit αριθμοί του κάθε vector μετασχηματίζονται σε 16bit και αποθηκεύονται σε 128bit vector. Με αυτό τον τρόπο επιλύεται το πρόβλημα της υπερχείλισης. Στο τέλος της διαδικασίας πραγματοποιείται η αντίστροφη διαδικασία. Δηλαδή οι 8 bit αριθμοί μετασχηματίζονται πάλι σε 8 bit (δεν υπάρχει πλέον πρόβλημα υπερχείλισης). Για τις παραπάνω λειτουργίες χρησιμοποιούνται οι πράξεις vmovl_u8 (μετασχηματισμός 8bit αριθμών σε 16bit αριθμούς) και vmovn_u16 (μετασχηματισμός 16bit αριθμών σε 8bit αριθμούς).

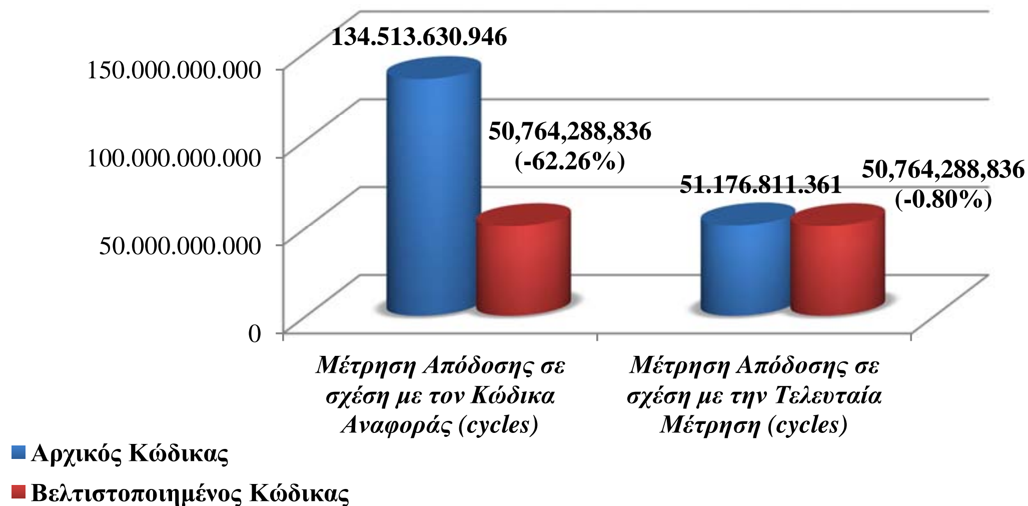
Μία επιπλέον βελτίωση στην απόδοση μπορεί να επιτευχθεί κάνοντας χρήση των inline functions. Στην περίπτωση του Motion Compensation ως inline functions ορίζονται οι συναρτήσεις form_component_predictionWidth16 και form_component_predictionWidth8.

4.5.2 Μέτρηση Απόδοσης – Νέο Profiling

Ολοκληρώνοντας της βελτιώσεις στη διαδικασία Motion Compensation η απόδοση μεταβάλλεται σε κάθε βελτιστοποίηση σύμφωνα με τα παρακάτω διαγράμματα :

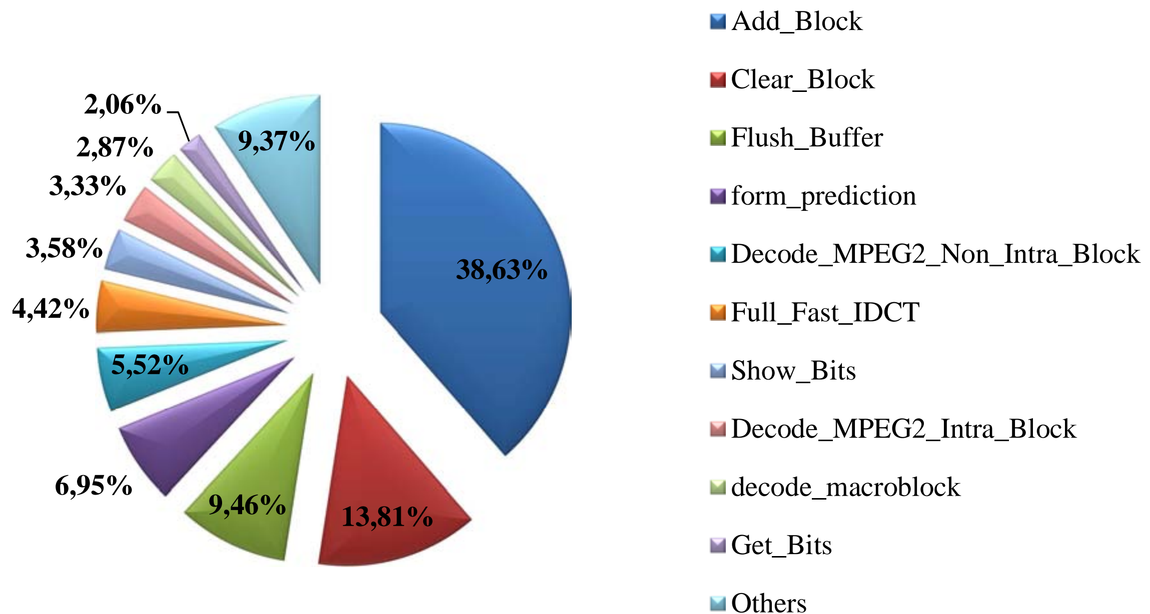


Διάγραμμα 9 : Μέτρηση Απόδοσης Μετά τη Χρήση Εντολών NEON στη διαδικασία Motion Compensation



Διάγραμμα 10 : Μέτρηση Απόδοσης Μετά τη Χρήση Inline στις Συναρτήσεις form_component_predictionWidth16 και form_component_predictionWidth8

Μετά τις βελτιστοποιήσεις στη διαδικασία Motion Compensation τα αποτελέσματα του profiling φαίνονται στο παρακάτω διάγραμμα :



Διάγραμμα 11 : Profiling with Motion Compensation optimized

4.6 Adding Prediction and Coefficient Data

Η διαδικασία Adding Prediction and Coefficient Data μαζί με το τελικό Saturation αποτελούν το τελευταίο στάδιο της αποκωδικοποίησης ενός picture (frame ή field). Μετά από αυτό το στάδιο προκύπτουν οι τελικές τιμές των pixel για το luminance και το chrominance του picture. Ουσιαστικά αυτό που γίνεται σε αυτό το στάδιο είναι να αθροίζονται τα δεδομένα που προέκυψαν από τη διαδικασία αποκωδικοποίησης ενός macroblock με τα δεδομένα που προέκυψαν από το motion compensation κάνοντας ταυτόχρονα και το αντίστοιχο saturate στο διάστημα [0 , 255].

Η παραπάνω λειτουργικότητα υλοποιείται στο πηγαίο κώδικα αναφοράς από τη συνάρτηση Add_block. Στο profiling του κώδικα αναφοράς (διάγραμμα 1) η συνάρτηση Add_block καταλαμβάνει την τέταρτη θέση με ποσοστό 14.57% ενώ στο τελευταίο profiling (διάγραμμα 11) καταλαμβάνει την πρώτη θέση με ποσοστό 38.63%.

4.6.1 Βελτιώσεις σε Επίπεδο Γλώσσας C

Μία πρώτη βελτίωση της συνάρτησης Add_block αποτελεί η μεταφορά των if statements από την Add_block στην καλούσα συνάρτηση motion_compensation. Παράλληλα με αυτή την αλλαγή μειώνονται τα call arguments της Add_block και ο έλεγχος για το . Με τα παραπάνω επιτυγχάνετε :

1. Μείωση της χρήσης if statement. Οι έλεγχοι για το chroma format, για τον είδος του picture (field – frame) και για το είδος DCT Coding (field based – frame based) εκτελούνται μόνο μία φορά για κάθε macroblock και όχι ξεχωριστά για κάθε block. Πρόκειται για μία έγκυρη αλλαγή καθώς η αποκωδικοποίηση ενός picture γίνεται ανά macroblock, γεγονός που σημαίνει πως όλα τα blocks ενός macroblock θα έχουν τα ίδια χαρακτηριστικά.
2. Ελάττωση των εκτελέσιμων εντολών.

   Η αντιστοίχηση των blocks στα reference και current pictures και η εύρεση των αντίστοιχων διευθύνσεων πραγματοποιείται σε επίπεδο macroblock, μία φορά για το luminance και μία φορά για κάθε chrominance (ουσιαστικά υπολογίζεται η αντίστοιχη διεύθυνση για το πρώτο block). Οι διευθύνσεις των υπολοίπων block υπολογίζονται προσθέτοντας το κατάλληλο διάστημα (ανάλογα με τα χαρακτηριστικά των picture και του macroblock) στην προηγούμενη διεύθυνση.

   Παρόμοια υπολογίζεται και το διάστημα που πρέπει να προστεθεί για να αντιστοιχίζεται η κάθε γραμμή του block με την κατάλληλη θέση στο picture. Και αυτός ο υπολογισμός μεταφέρετε στη motion_compensation και εκτελείται σε επίπεδο macroblock, μία φορά για το luminance και μία φορά για κάθε chrominance.

   Η συνάρτηση Add_Block αντικαθίσταται από τις Add_Block_addflag1 και Add_Block_addflag1. Πλέον ο έλεγχος

ιf ((macroblock_type & MACROBLOCK_INTRA) == 0)
μεταφέρεται στη συνάρτηση motion_compensation και εκτελείτε μία φορά για κάθε macroblock. Συνεπώς με έναν ελέγχο επιλέγεται η κατάλληλη συναρτήση add_block που καλείται για κάθε block.

Υλοποιώντας τις παραπάνω αλλαγές στον πηγαίο κώδικα προκύπτουν τα παρακάτω αποτελέσματα όσον αφορά τη μέτρηση της απόδοσης :

4.6.2 Χρήση NEON instructions

Η διαδικασία Adding Prediction and Coefficient Data αποτελεί ένα ακόμη σημείο της αποκωδικοποίησης που μπορούν να χρησιμοποιηθούν αποδοτικά NEON εντολές. Σε αυτό το στάδιο αθροίζονται τα 8x8 block δεδομένων ενός macroblock που προέκυψαν μετά το IDCT, με τα αντίστοιχα 8x8 blocks που προέκυψαν από την αποκωδικοποίηση των motion vector και του motion compensation. Ταυτόχρονα, γίνεται και το saturate.

Το άθροισμα των στοιχείων των block υλοποιείτε στον κώδικα αναφοράς με δύο for loop, ενώ το saturate υλοποιείτε με τη χρήση ενός πίνακα μεγέθους 1024 με διάστημα αποθήκευσης στοιχείων [-384 , 640], ο οποίος αρχικοποιείται ως εξής :

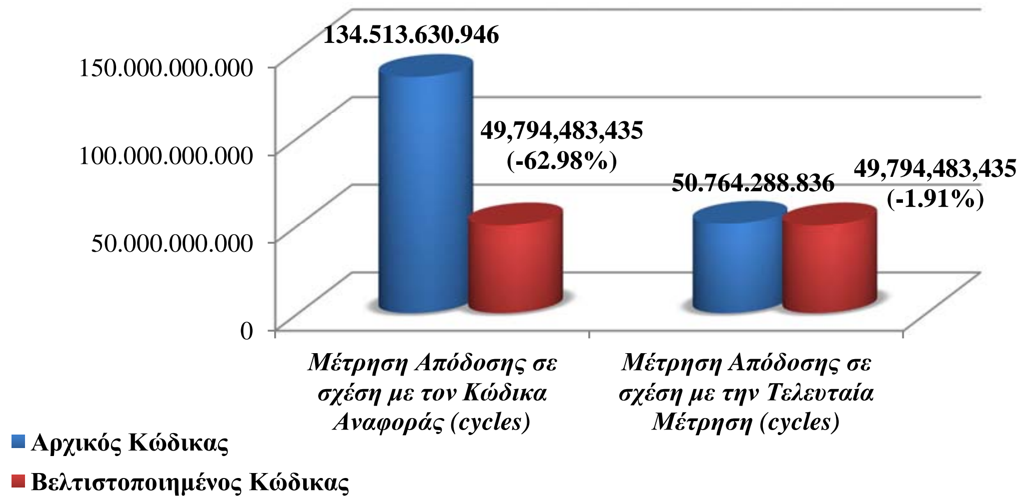
```
for (i=-384; i<640; i++)
        Clip[i] = (i<0) ? 0 : ((i>255) ? 255 : i);
```

Με τη χρήση διανυσμάτων χρησιμοποιείται μόνο ένα for loop για το άθροισμα ενώ το δεύτερο αντικαθίσταται από NEON πράξεις. Το saturate επιτυγχάνεται μέσω των διανυσματικών εντολών για μετατροπή των 16-bit αποτελεσμάτων σε 8-bit που απαιτούνται για την αναπαράσταση ενός pixel. Η παραπάνω μετατροπή πραγματοποιείται με τη χρήση της εντολής vqmovun_s16.

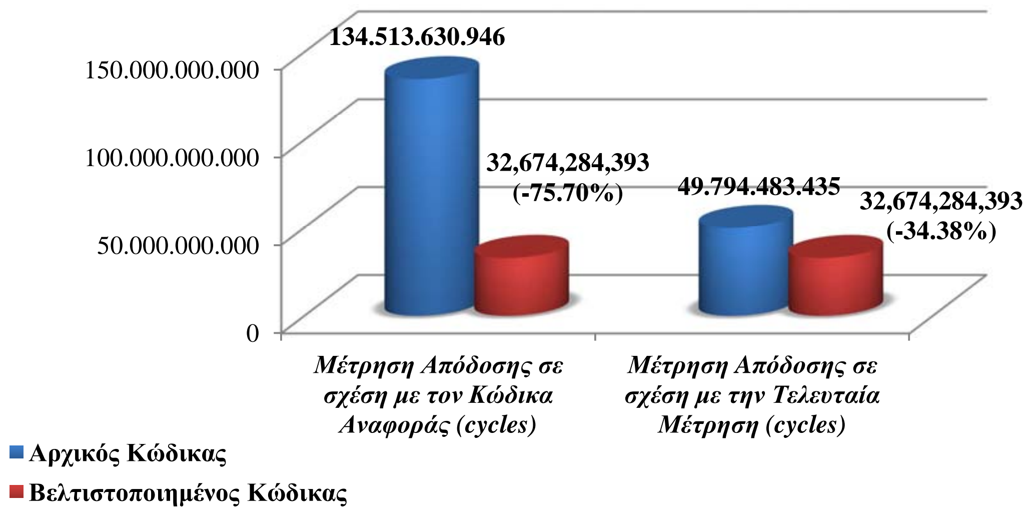
Περαιτέρω βελτίωση στη διαδικασία Adding Prediction and Coefficient Data επιτυγχάνεται ορίζοντας τις Add_Block_addflag1 και Add_Block_addflag0 ως inline.

4.6.3 Μέτρηση Απόδοσης – Νέο Profiling

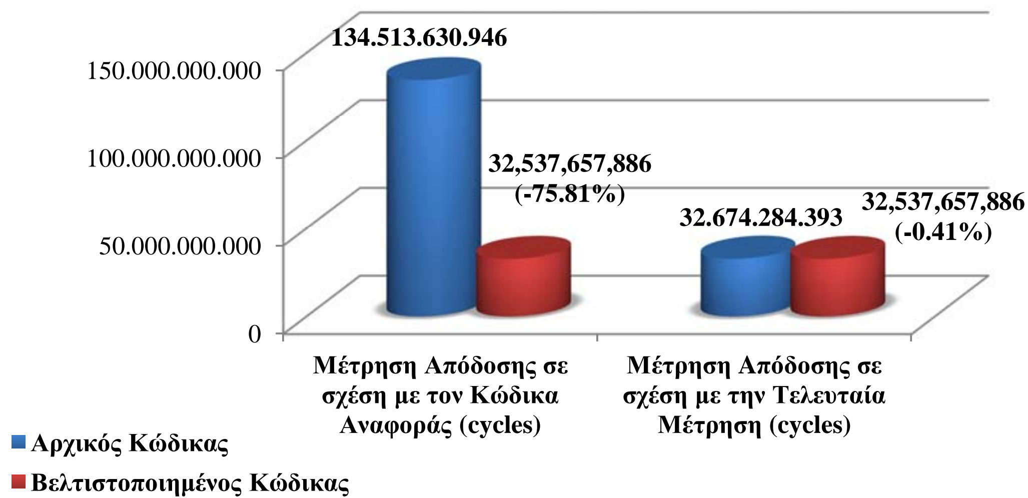
Ολοκληρώνοντας τις βελτιώσεις στη διαδικασία Adding Prediction and Coefficient Data η απόδοση μεταβάλλεται σύμφωνα με τα επόμενα διαγράμματα:



Διάγραμμα 12 : Μέτρηση Απόδοσης Μετά τις Βελτιώσεις στην Add_Block σε Επίπεδο Γλώσσας C

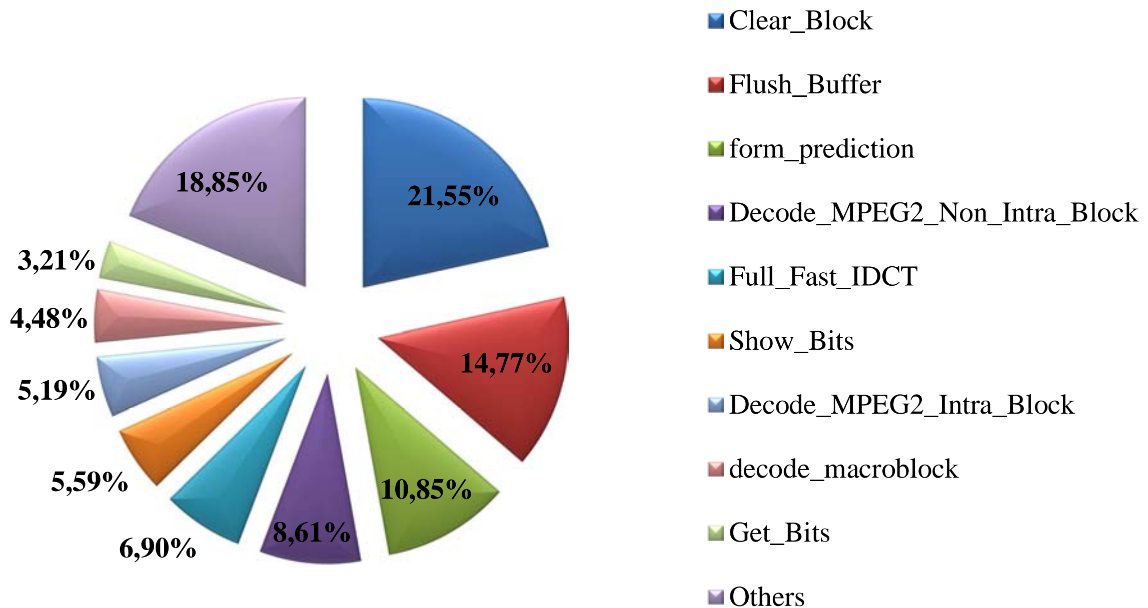


Διάγραμμα 13 : Μέτρηση Απόδοσης Μετά τη Χρήση Εντολών NEON στη διαδικασία Adding Prediction and Coefficient Data.



Διάγραμμα 15 : Μέτρηση Απόδοσης Μετά τη Χρήση Inline στις Συναρτήσεις Add_Block_addflag1 και Add_Block_addflag0

Το νέο profiling παρουσιάζεται στον παρακάτω διάγραμμα :



Διάγραμμα 16 : Profiling with Adding Prediction and Coefficient Data optimized

4.7 Clear Block

Η συνάρτηση Clear_Block υλοποιεί το μηδενισμό των πινάκων που αποθηκεύονται τα αποκωδικοποιημένα block και καλείται πάντα πριν την έναρξη της αποκωδικοποίησης ενός DCT block, ώστε να μην υπάρχουν δεδομένα από προηγούμενα αποκωδικοποιημένα block.

Η Clear_Block στο αρχικό profiling (διάγραμμα 1) καταλαμβάνει την Πέμπτη θέση με ποσοστό 5.41% ενώ στο τελευταίο profiling (διάγραμμα 16) καταλαμβάνει την πρώτη θέση με ποσοστό 21.55%.

4.7.1 Χρήση NEON instructions

Η συνάρτηση Clear_Block υλοποιείται στον κώδικα αναφοράς με τη χρήση ενός for loop και λειτουργεί θέτοντας την τιμή μηδέν σε κάθε στοιχείο του block.

```
For (i=0; i<64; i++)
        *Block_Ptr++ = 0;
```

Η Clear_Block βελτιώνεται με τη χρήση NEON εντολών ως εξής :

1. Ορίζεται ένα vector το οποίο αρχικοποιείται με την τιμή μηδέν.
2. Το vector αποθηκεύεται σε κάθε γραμμή του block.

Με τον παραπάνω τρόπο ελαχιστοποιούμε της προσπελάσεις μνήμης δημιουργώντας στην ουσιαστικά μία γρήγορη memset.

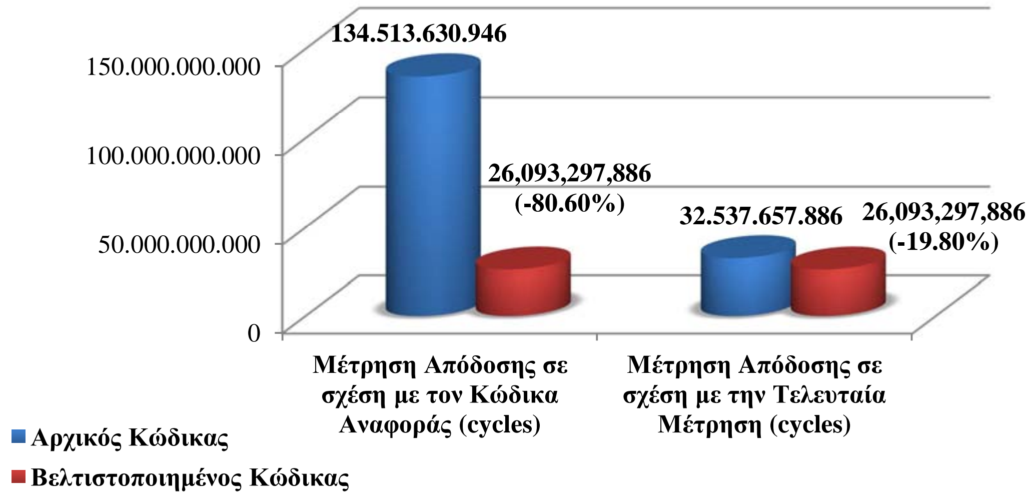
Ο πηγαίος κώδικας της συνάρτησης Clear_Block με χρήση NEON vector είναι:

```
static void Clear_Block (int comp)
{
        short *Block_Ptr;
        int16x8_t blk_s16x8;

        Block_Ptr = ld->block[comp];

        //fortwnetai to 0 se vector
        blk_s16x8 = vdupq_n_s16(0);

        //apothikeueati se kthe grammi tou block
        vst1q_s16(Block_Ptr, blk_s16x8);
        vst1q_s16(Block_Ptr+8, blk_s16x8);
        vst1q_s16(Block_Ptr+16, blk_s16x8);
        vst1q_s16(Block_Ptr+24, blk_s16x8);
        vst1q_s16(Block_Ptr+32, blk_s16x8);
        vst1q_s16(Block_Ptr+40, blk_s16x8);
        vst1q_s16(Block_Ptr+48, blk_s16x8);
        vst1q_s16(Block_Ptr+56, blk_s16x8);
}
```

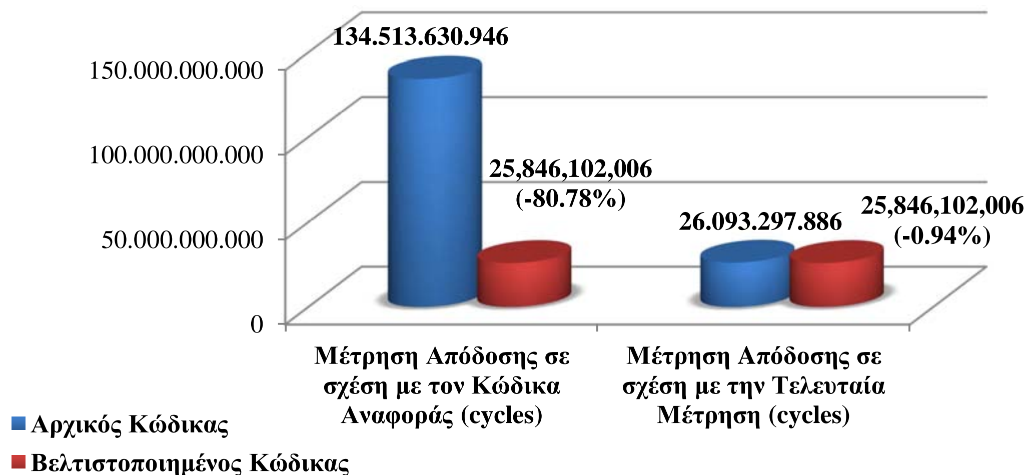
Η απόδοση της Clear_Block βελτιώνεται ακόμη περισσότερο με τον ορισμό της συνάρτησης ως inline function.

4.7.2 Μέτρηση Απόδοσης – Νέο Profiling

Ολοκληρώνοντας τις βελτιώσεις στη συνάρτηση Clear_Block η απόδοση μεταβάλλεται σύμφωνα με τα παρακάτω διαγράμματα:

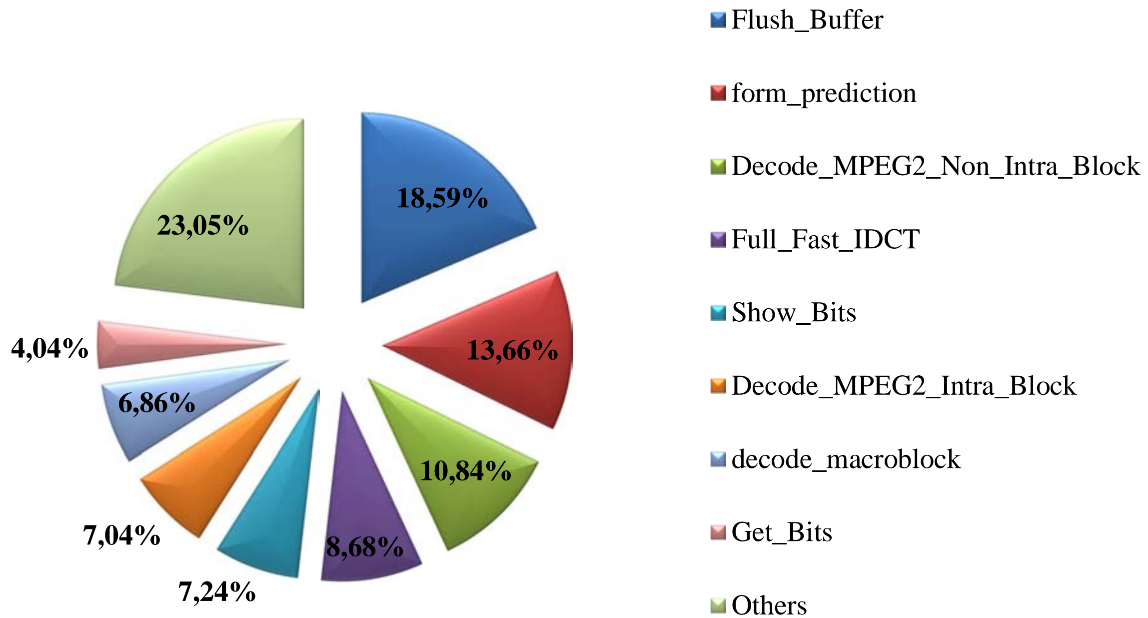


Διάγραμμα 17 : Μέτρηση Απόδοσης Μετά τη Χρήση NEON εντολών στην Clear_Block.



Διάγραμμα 18 : Μέτρηση Απόδοσης Μετά το Inline της Clear_Block.

Το νέο profiling παρουσιάζεται στο επόμενο διάγραμμα :



Διάγραμμα 19 : Profiling with Clear_Block optimized.

4.8 Buffer and Bit Functions

Στην παράγραφο αυτή μελετώνται και βελτιώνονται οι λειτουργίες που πραγματοποιούνται πάνω στον buffer.

Στον κώδικα αναφοράς η λειτουργία του buffer οργανώνεται σε δύο επίπεδα. Υπάρχει ένας εσωτερικός μεγέθους 32-bit, με τον οποίο επικοινωνεί το κυρίως πρόγραμμα και ένας εξωτερικός μεγέθους 2048 bytes, στον οποίο φορτώνονται τα δεδομένα από το bitstream (φορτώνονται δεδομένα από το δίσκο σε bytes). Στον εσωτερικό buffer τα δεδομένα φορτώνονται από τον εξωτερικό (σε bits).

Οι συναρτήσεις που χρησιμοποιούνται για προσπέλαση των δεδομένων του εσωτερικού buffer είναι :

1. int Show_Bits(int N) : Επιστρέφει την αριθμητική τιμή n bits από τον εσωτερικό buffer του συστήματος χωρίς να απομακρύνει bits.

2. void Flush_Buffer(int N) : Απομακρύνει n bits από τον εσωτερικό buffer του συστήματος.
3. int Get_Bits(int N) : Απομακρύνει n bits από τον εσωτερικό buffer του συστήματος και επιστρέφει την αριθμητική τιμή των n bits. Υλοποιείται ως : int Get_Bits(int N) {r = Show_Bits(N); Flush_Buffer(N); return r;}
4. void Flush_Buffer32() : Όμοια με τη Flush_Buffer για N = 32.
5. int Get_Bits32() : Όμοια με τη Get_Bits για N = 32.
6. int Get_Bits1() : Όμοια με τη Get_Bits για N = 1.

Ενώ για το γέμισμα του εξωτερικού buffer χρησιμοποιείτε η συνάρτηση :

void Fill_Buffer() : Μεταφέρει δεδομένα από το bitstream στον εξωτερικό buffer.

Από το τελευταίο profiling (διάγραμμα 19) προκύπτει το συμπέρασμα πως οι παρπάνω συναρτήσεις απαιτούν συνολικά περίπου το 30% του συνολικού χρόνου εκτέλεσης. Μάλιστα η Flush_Buffer καταλαμβάνει την πρώτη θέση.

4.8.1 Δημιουργία Μεγαλύτερου Buffer

Από την παρατήρηση της υλοποίησης της Flush_Buffer και της Flush_Buffer32 προκύπτει το συμπέρασμα πως για κάθε κλήση τους, που απαιτεί φόρτωση δεδομένων από τον εξωτερικό στον εσωτερικό buffer, εκτελείται έλεγχος για την ανάγκη φόρτωσης δεδομένων στον εξωτερικό buffer.

Αυτός ο έλεγχος μπορεί να αποφευχθεί με τη δημιουργία ενός μεγαλύτερου εξωτερικού buffer και η μεταφορά του συγκεκριμένου ελέγχου σε διαφορετικό σημείο του κώδικα όπου και θα πραγματοποιείται λιγότερες φορές.

Για την εφαρμογή των παραπάνω στον πηγαίο κώδικα κάνουμε τις παρακάτω αλλαγές :

4 Αυξάνεται το μέγεθος του εξωτερικού buffer από 2048 bytes σε 500 Kbytes.

5 Μεταφέρεται ο έλεγχος από τις συνάρτησης Flush_Buffer και Flush_Buffer32 στη picture_data. Ο έλεγχος πλέον πραγματοποιείται μετά το τέλος της αποκωδικοποίησης κάθε slice και είναι της μορφής :

if (UnReadBytes <100Kbytes) Fill_Buffer( );

6 Αλλάζει η υλοποίηση της συνάρτησης Fill_Buffer. Κάθε φορά που καλείτε η συνάρτηση υπολογίζεται ο αριθμός των bytes που είναι ακόμη αχρησιμοποίητα (από τον προηγούμενο έλεγχο ο αριθμός αυτός θα είναι < 102400). Τα συγκεκριμένα bytes θα είναι αποθηκευμένα στις τελευταίες θέσεις του buffer. Με τη χρήση μιας γρήγορης memcpy που υλοποιείτε με τη βοήθεια NEON vector τα αχρησιμοποίητα bytes μεταφέρονται στην αρχή του buffer, ενώ στις υπόλοιπες θέσεις φορτώνονται νέα δεδομένα.

7 Πραγματοποιούνται κατάλληλες αλλαγές και στη συνάρτηση Initialize_Buffer για να υποστηριχθούν οι νέες βελτιώσεις.

4.8.2 Ελαχιστοποίηση των Προσπελάσεων Μνήμης

Παρατηρώντας τις συναρτήσεις που αφορούν την προσπέλαση δεδομένων των buffer προκύπτει το συμπέρασμα πως γίνονται περιττές προσπελάσεις μνήμης. Οι περισσότερες από τις εντολές των συναρτήσεων αυτών χρησιμοποιούν τελεστές μέσω δείκτη σε δομή struct. Με αυτό τον τρόπο γίνεται μία προσπέλαση μνήμης για να ανακτηθεί η τελική διεύθυνση από το δείκτη και μία για να ανακτηθεί η τιμή του τελεστή. Η χρήση των δεικτών σε struct είναι απαραίτητη ώστε να υποστηριχθεί το scalability που περιγράφεται στο standard Mpeg2. Κάθε layer έχει τη δική του μεταβλητή struct με τις δικές του τιμές μεταβλητών.

Για να ελαττωθούν οι προσπελάσεις μνήμης στις συγκεκριμένες συναρτήσεις ο πηγαίος κώδικας τροποποιείται ως εξής :

1. Δημιουργούνται global μεταβλητές, αντίστοιχες με αυτές που βρίσκονται στη δομή struct και αφορούν τους buffer, στις οποίες ανατίθενται οι τιμές των αντίστοιχων μεταβλητών του struct.
2. Δημιουργείται η συνάρτηση switch_layer_context η οποία καλείται κάθε φορά που υπάρχει η ανάγκη να μεταβούμε σε διαφορετικό layer. Δέχεται σαν όρισμα το δείκτη struct του επόμενου layer, αντιγράφει τις τιμές των global μεταβλητών πίσω στο struct του ενεργού layer, θέτει ως ενεργό το layer που δίνεται σαν όρισμα στη συνάρτηση και τελικά αντιγράφει τις τιμές των μεταβλητών που αφορούν τους buffer από το struct του ενεργού layer στις global μεταβλητές.
3. Αντικαθίστανται όπου εμφανίζονται, οι μεταβλητές των struct από τις global μεταβλητές.

4.8.3 Υπόλοιπες Βελτιστοποιήσεις

Περεταίρω βελτίωση των συναρτήσεων που χρησιμοποιούνται για την προσπέλαση των buffer, επιτυγχάνεται με τη μείωση των εκτελέσιμων εντολών.

Στον κώδικα αναφοράς η μεταβλητή ld->Incnt (η οποία γίνεται Incnt μετά τη χρήση global μεταβλητών) χρησιμοποιείται για να μετράει τα διαθέσιμα bits του εσωτερικού 32-bit buffer. Αλλάζοντας τη λειτουργία της Incnt ώστε πλέον να μετράει τα bits του buffer που έχουν ήδη χρησιμοποιηθεί, καταργούνται οι εντολές αφαίρεσης που χρησιμοποιούνταν για την υπολογισμό των bits ολίσθησης κατά τη διαδικασία φόρτωσης δεδομένων στον εσωτερικό buffer.

Μια επιπλέον βελτίωση μπορεί να επιτευχθεί μειώνοντας τις κλήσεις της Flush_Buffer. Αυτή η βελτίωση πραγματοποιείται μέσω της συνάρτησης Get_Bits που όπως περιγράφηκε παραπάνω υλοποιείτε με μια κλήση της Show_Bits και μία κλήση της Flush_Buffer. Από την παρατήρηση του πηγαίου κώδικα προκύπτει το γεγονός πως για πολλές από τις κλήσεις της Get_Bits για N=1 ή της Get_Bits1, δεν είναι απαραίτητη η κλήση της Flush_Buffer καθώς

ακολουθούν ή ακολουθούνται από κλήσεις της Get_Bits με N>1 ή της Flush_Buffer που πραγματοποιούν τον απαιτούμενο έλεγχο για την ανάγκη φόρτωσης δεδομένων στον εσωτερικό buffer.

Συνεπώς, αλλάζοντας την υλοποίηση της Get_Bits1 (ώστε να μην γίνεται κλήση της Flush_Buffer) και αντικαθιστώντας της προαναφερθείσες κλήσεις της Get_Bits με την Get_Bits1 επιτυγχάνεται η ζητούμενη βελτίωση.

Μία ακόμη συνάρτηση του αφορά προσπελάσεις του εσωτερικού buffer και δύναται να βελτιωθεί είναι η Show_Bits. Στον κώδικα αναφοράς η Show_Bits υλοποιείται ως εξής :

unsigned int Show_Bits(int N) { return bfr >> (32-N); }

Στην παραπάνω υλοποίηση η πράξη της αφαίρεσης μπορεί να αποφευχθεί για πολλές από της κλήσεις της Show_Bits. Αυτό είναι εφικτό αλλάζοντας τη λογική της Show_Bits ώστε να καλείται με παράμετρο όχι το N, αλλά το 32-N. Δηλαδή να καλείται όχι με τα bits του buffer που πρέπει να επιστραφούν, αλλά με τα bits του buffer που πρέπει να ολισθήσουν. Η παραπάνω βελτίωση δεν είναι αποδοτική όταν η κλήση της Show_Bits πραγματοποιείται με παράμετρο μεταβλητή και όχι αριθμό (δηλαδή όταν καλείται μέσω της Get_Bits).

Για την εφαρμογή της παραπάνω βελτίωσης δημιουργούμε τη συνάρτηση :

unsigned int Show_Bits_Direct(int N) { return bfr >> N; }

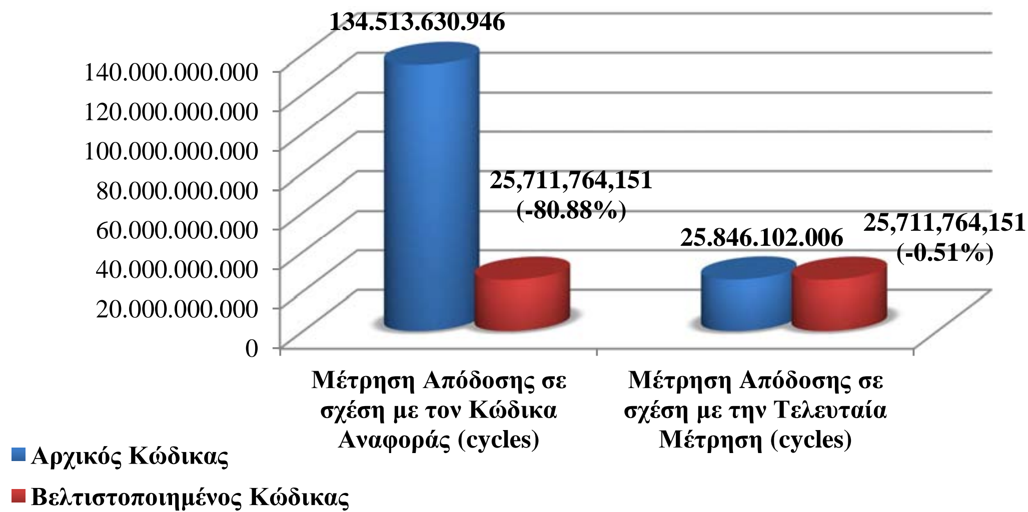
η οποία αντικαθιστά τις κλήσεις της Show_Bits στον πηγαίο κώδικα,όταν αυτή καλείται με παράμετρο αριθμό. Η Show_Bits_Direct καλείται με παράμετρο τον αριθμό των bits του buffer που πρέπει να ολισθήσουν ώστε να επιστραφεί το ζητούμενο αποτέλεσμα.

Μία τελευταία βελτίωση των συναρτήσεων που αφορούν τις προσπελάσεις δεδομένων των buffer αποτελεί η χρήση των inline functions.

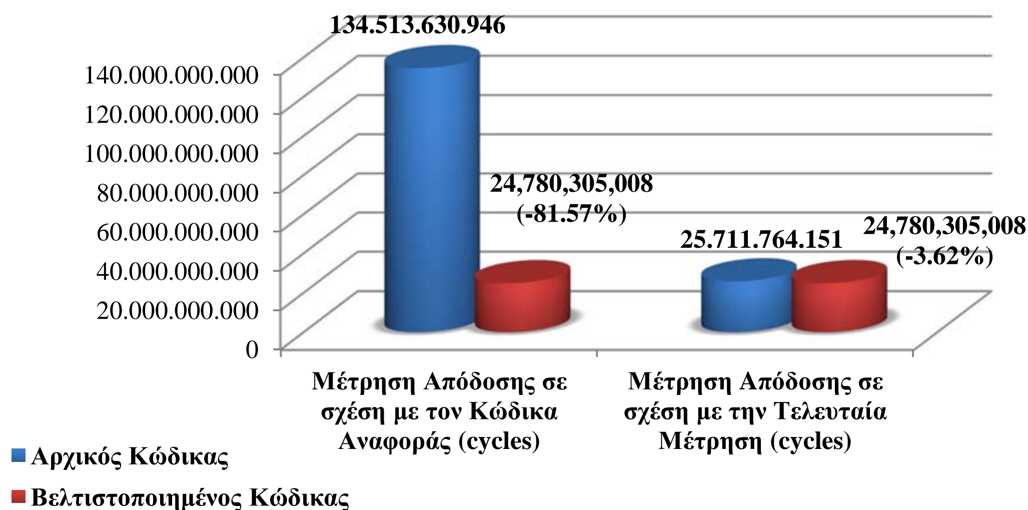
Για το σκοπό αυτό δημιουργείται το αρχείο getbits.h στο οποίο μεταφέρουμε τις παραπάνω συναρτήσεις, τις οποίες ορίζουμε σαν inline. Η getbits.h γίνεται include από τα αντίστοιχα αρχεία.

4.8.4 Μέτρηση Απόδοσης – Νέο Profiling

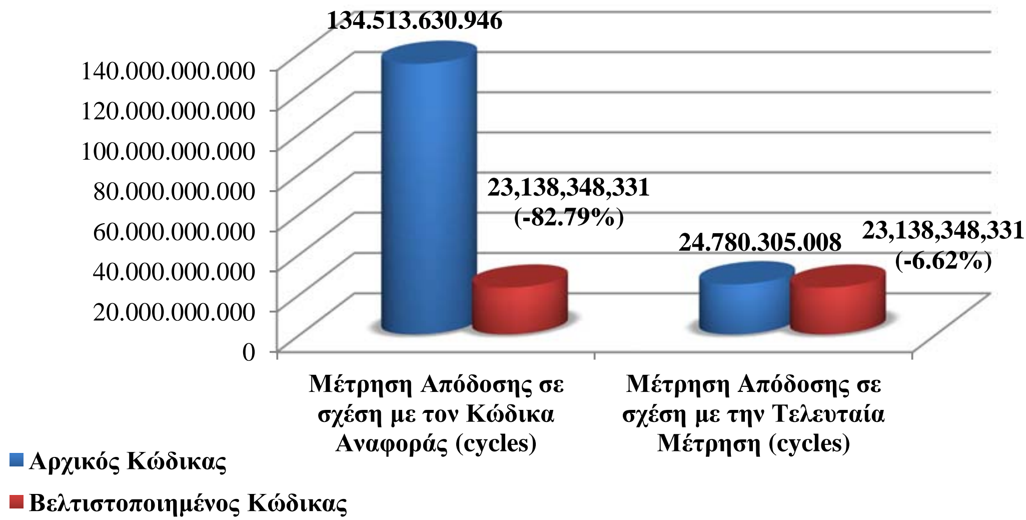
Ολοκληρώνοντας τις βελτιώσεις στις συναρτήσεις που αφορούν τους Buffer, η απόδοση μεταβάλλεται σύμφωνα με τα επόμενα διαγράμματα:



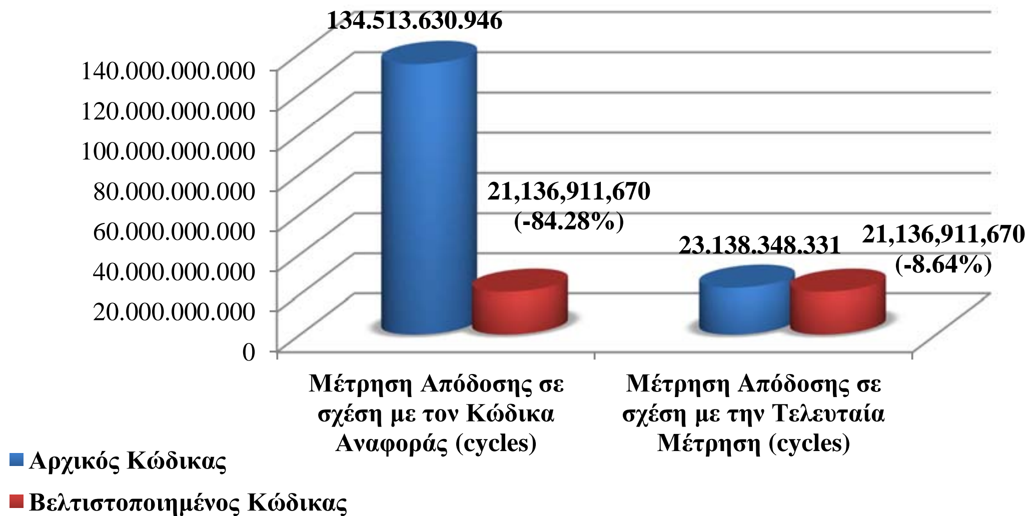
Διάγραμμα 20 : Μέτρηση Απόδοσης Μετά τη Δημιουργία Μεγαλύτερου Buffer.



Διάγραμμα 21 : Μέτρηση Απόδοσης Μετά την Ελαχιστοποίηση των Προσπελάσεων Μνήμης

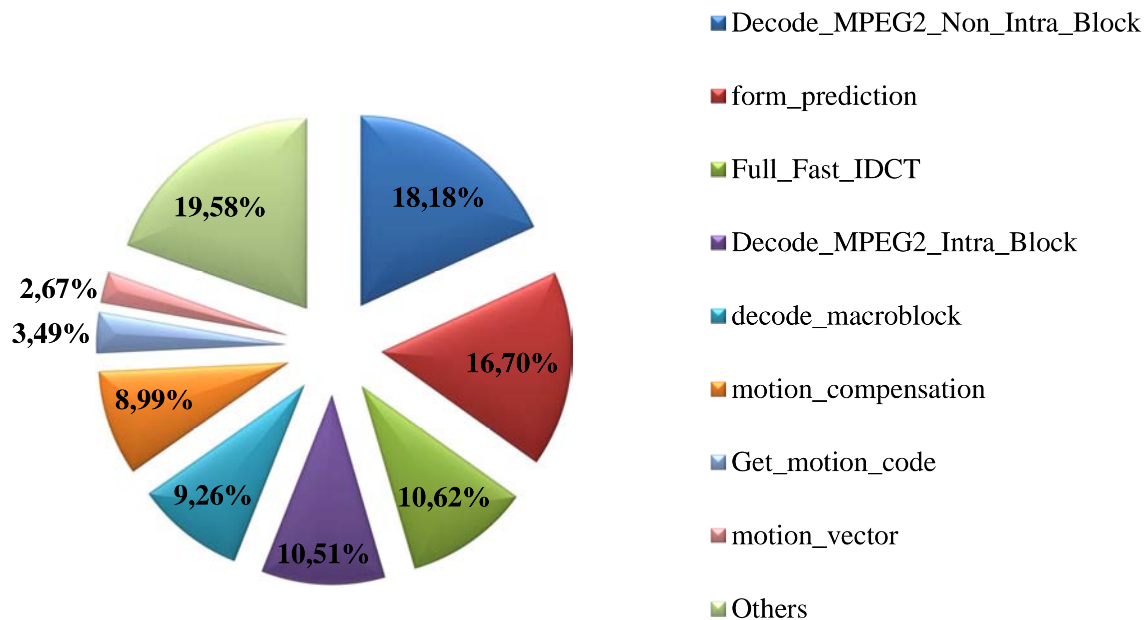


Διάγραμμα 22 : Μέτρηση Απόδοσης Μετά τη Μείωση των Εκτελέσιμων Εντολών, την Ελαχιστοποίηση των Κλήσεων Flush_Buffer και τη δημιουργία της Show_Bits_Direct.



Διάγραμμα 23 : Μέτρηση Απόδοσης Μετά τον Ορισμό των Συναρτήσεων Προσπέλασης Δεδομένων των Buffer ως inline.

Το νέο profiling παρουσιάζεται στο επόμενο διάγραμμα :



Διάγραμμα 24 : Profiling with Buffer Functions optimized.

4.9 Επιπλέον Βελτιώσεις

Επιπλέον βελτιώσεις είναι εφικτό να πραγματοποιηθούν στον πηγαίο κώδικα. Αυτές αφορούν κυρίως βελτιστοποιήσεις σε επίπεδο γλώσσας C. Στην παρούσα παράγραφο παρουσιάζονται οι επιπλέον βελτιώσεις και η μέτρηση της αντίστοιχης απόδοσης.

4.9.1 Αλλαγές – Βελτιώσεις

Από την ανάλυση της υλοποίησης της συνάρτησης decode_macroblock προκύπτει το συμπέρασμα πως πραγματοποιούνται έλεγχοι και πράξεις που μπορούν αποφευχθούν. Για να επιτευχθεί αυτό η decode_macroblock τροποποιείται ως εξής :

1. Πριν από κάθε κλήση των συναρτήσεων για αποκωδικοποίηση του κάθε block του macroblock, γίνεται έλεγχος για το είδος του macroblock (Mpeg1 ή Mpeg2), ώστε να επιλεγεί η κατάλληλη συνάρτηση. Αυτός ο έλεγχος μεταφέρεται, ώστε να πραγματοποιείται μία φορά για κάθε macroblock και όχι για κάθε block.

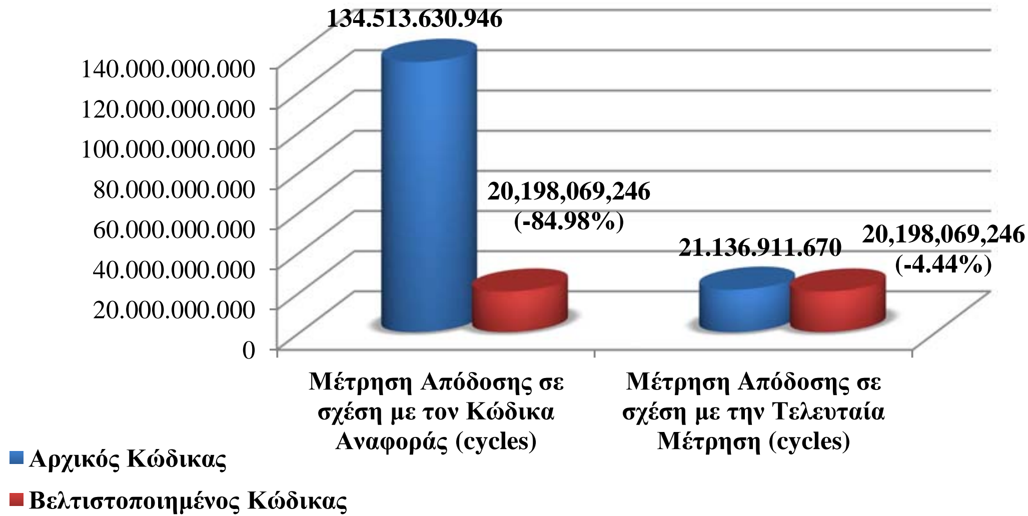
2. Για την προσπέλαση του κάθε block του macroblock χρησιμοποιείται ένα for loop, με διάστημα από 0 έως τον αριθμό των block (μεταβλητή block_count). Ο αριθμός αυτός μπορεί να υπολογιστεί ενός ελέγχου (if statement) για το chroma format. Με αυτό τον τρόπο επιτυγχάνεται το unroll του for loop και αποφεύγονται περιττοί έλεγχοι.
3. Για κάθε block πραγματοποιείτε έλεγχος για την ανάγκη αποκωδικοποίησης ή μη του block. Έλεγχος έχει ως εξής :
   if (coded_block_pattern & (1<<(block_count-1-comp))
   Με την προηγούμενη αλλαγή (unroll for loop) γνωρίζουμε ακριβώς το block_count (συνολικός αριθμός block) και το comp (ο δείκτης του block). Συνεπώς, οι πράξεις *(1<<(block_count-1-comp)* αντικαθίστανται από τα προϋπολογισμένα αποτελέσματα.

Οι συναρτήσεις αποκωδικοποίησης των DCT block (Decode_MPEG2_Intra_Block και Decode_MPEG2_Non_Intra_Block) μπορούν βελτιωθούν περισσότερο με τους εξής τρόπους :

1. Ο υπολογισμός της τελικής τιμής του κάθε μη μηδενικό DCT συντελεστή που αποκωδικοποιείται γίνεται με την εξής εντολή :
   val = sign ? –val : val;
   Όπου val η ακέραια τιμή του συντελεστή και sign (παίρνει τιμές 0 για θετικό και 1 για αρνητικό αριθμό) το πρόσημο που προέκυψε από την αποκωδικοποίηση. Η παραπάνω εντολή αποτελεί ουσιαστικά ένα if statement. Μπορεί ωστόσο να αντικατασταθεί από απλές πράξεις ως εξής :
   val = (val XOR (- sign)) + sign;
   Η εντολή αυτή λειτουργεί ως εξής :
   Για sign = 0 : val = (val XOR (0)) + 0 = val + 0 = val.
   Για sign = 1 : val = (val XOR ([1 1 1 … 1 1 1] (σε αναπαράσταση bits))) + 1 =
   = (NOT) val + 1 = - val.
   Που καταλήγει στο ζητούμενο.
2. Μετά την αποκωδικοποίηση κάθε DCT συντελεστή, πραγματοποιείται ο έλεγχος :
   if (base.scalable_mode == SC_DP && nc == base.priority_breakpoint – 63)
   switch_layer_context (&enhan);
   για την ύπαρξη Data Partitioning. Για την αποφύγει των περιττών ελέγχων ο έλεγχος *if (base.scalable_mode == SC_DP)* μεταφέρεται πριν την έναρξη της αποκωδικοποίησης των AC συντελεστών. Αν είναι αληθείς συνεχίζεται η η ίδια διαδικασία αποκωδικοποίησης με τον έλεγχο *if (nc == base.priority_breakpoint – 63) { switch_layer_context (&enhan); }*, ενώ σε περίπτωση που είναι ψευδείς δεν πραγματοποιείται κανένας έλεγχος και επιπλέον καταργείται η πράξη nc++ .

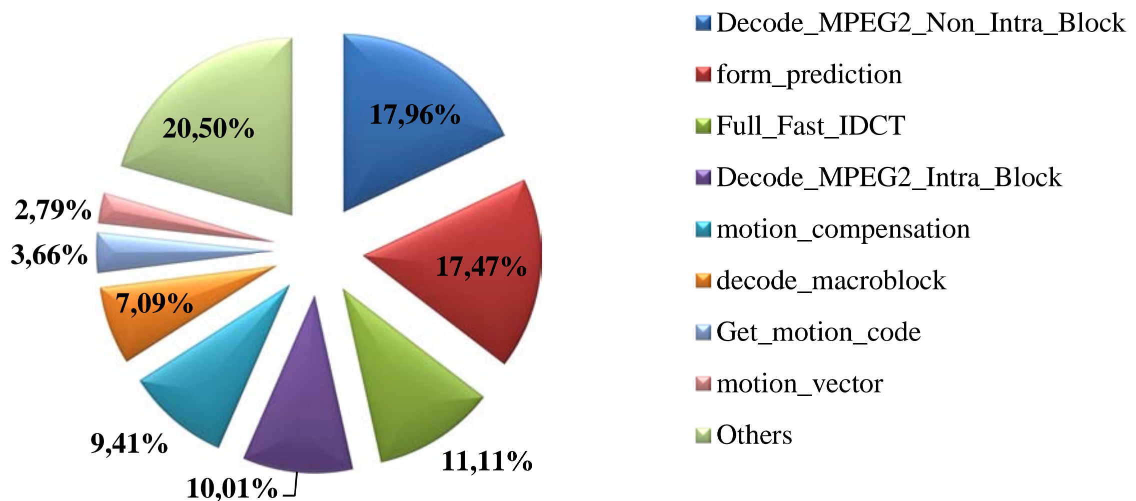
4.9.2 Μέτρηση Απόδοσης – Νέο Profiling

Ολοκληρώνοντας τις βελτιώσεις σε επίπεδο γλώσσας C στα διάφορα σημεία του πηγαίου κώδικα, η απόδοση μεταβάλλεται σύμφωνα με τα παρακάτω διαγράμματα:



Διάγραμμα 25 : Μέτρηση Απόδοσης Μετά τις Υπόλοιπες Βελτιώσεις σε Επίπεδο Γλώσσας C

Το νέο profiling παρουσιάζεται στο επόμενο διάγραμμα :

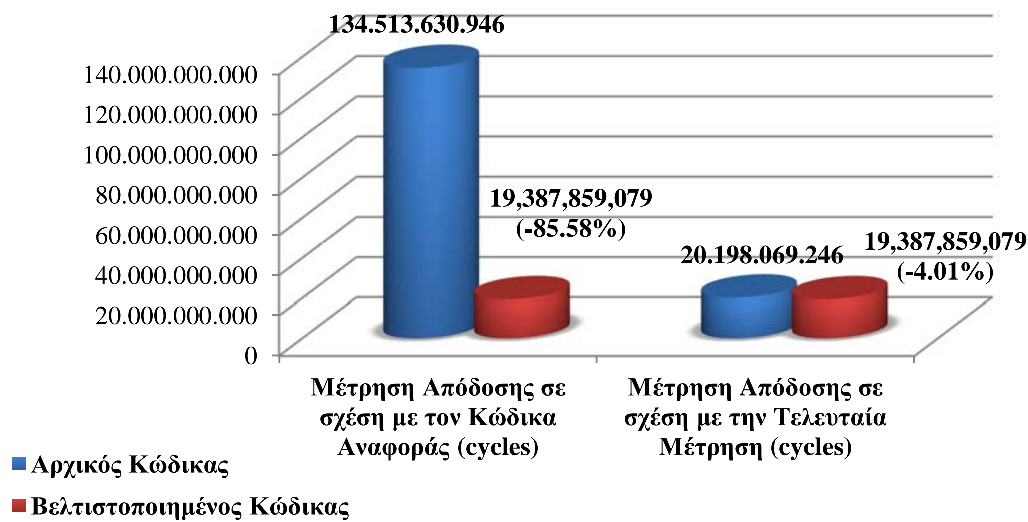


Διάγραμμα 26 : Profiling With Several Optimizations on C Language.

4.10 Compiler Optimization Options

Μία μέθοδος Βελτιστοποίησης Λογισμικού αποτελεί (όπως περιγράφεται στην παράγραφο 3.4) η χρήση των επιλογών βελτιστοποίησης που παρέχονται από τον μεταγλωττιστή.

Από τις επιλογές βελτιστοποίησης που παρέχει ο armcc, η ιδανικότερος συνδυασμός επιλογών για μείωση του συνολικού χρόνου εκτέλεσης είναι ο –O3 –Otime. Μεταγλωττίζοντας τον πηγαίο κώδικα μετά την υλοποίηση όλων των βελτιστοποιήσεων, με τις επιλογές –O3 –Otime και μετρώντας την απόδοση προκύπτουν τα αποτελέσματα :



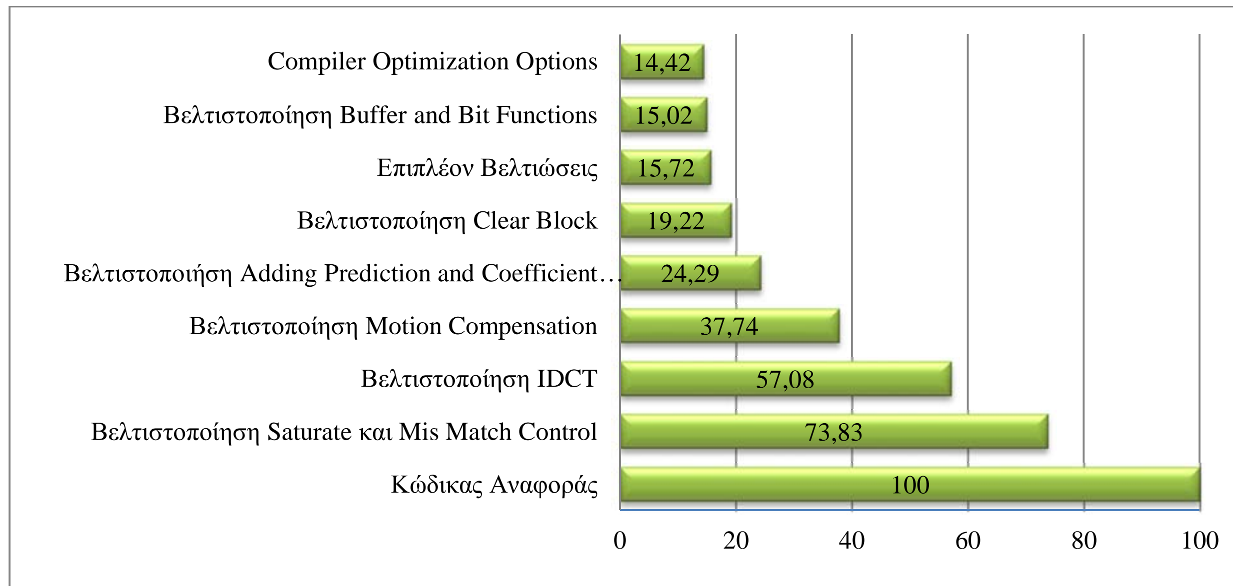
Διάγραμμα 27 : Profiling With Compiler's –O3 –Otime Options Used.

ΚΕΦΑΛΑΙΟ 5

Συμπεράσματα και Προτάσεις για Μελλοντική Έρευνα

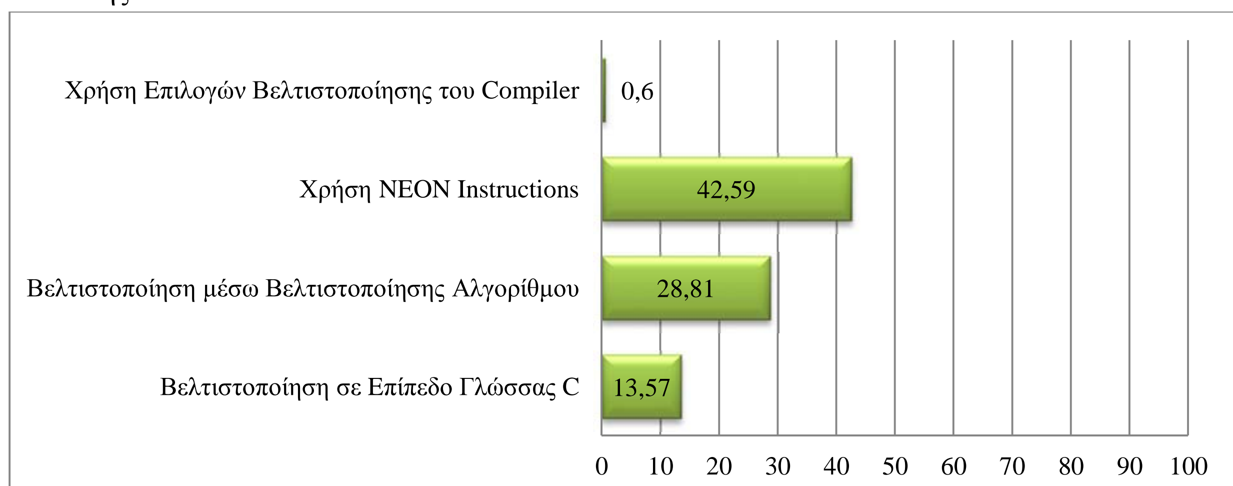
Στο τελευταίο κεφάλαιο της εργασίας καταγράφονται συγκεντρωτικά τα αποτελέσματα των μετρήσεων που προέκυψαν από κάθε στάδιο βελτιστοποίησης. Αναλύονται και παρουσιάζεται το τελικό αποτέλεσμα της όλης διαδικασίας. Τέλος, αναφέρονται ορισμένες προτάσεις εργασίας για μελλοντική έρευνα που προκύπτουν από την παρούσα εργασία.

Ολοκληρώνοντας την υλοποίηση των Βελτιστοποιήσεων στον πηγαίο κώδικα και συγκεντρώνοντας όλα τα αποτελέσματα που προέκυψαν από της μετρήσεις απόδοσης κατασκευάζεται το επόμενο διάγραμμα στο οποίο φαίνεται πως μεταβάλλεται η απόδοση σε κάθε στάδιο βελτιστοποίησης.



Διάγραμμα 5.1: Συγκεντρωτικός Πίνακας Απόδοσης (%)

Στο παρακάτω διάγραμμα παρουσιάζεται η συμβολή κάθε τεχνικής βελτιστοποίησης που χρησιμοποιήθηκε στην παρούσα εργασία ως ποσοστό επί της συνολικής βελτίωσης της απόδοσης.



Διάγραμμα 5.2: Ποσοστά Συμμετοχής της κάθε Τεχνικής στη Συνολική Βελτίωση

Από το διάγραμμα 5.2 προκύπτει το συμπέρασμα πως η μεγαλύτερη βελτίωση επιτεύχθηκε με τη **χρήση NEON εντολών** που αποτελούν κομμάτι της εκάστοτε αρχιτεκτονικής. Αυτό υποδεικνύει το πόσο σημαντική είναι η γνώση της αρχιτεκτονικής και των δυνατοτήτων που αυτή παρέχει για την ανάπτυξη μιας εφαρμογής. **Η Βελτιστοποίηση μέσω Βελτιστοποίησης Αλγορίθμου** αποτελεί τη δεύτερη καλύτερη μέθοδο που χρησιμοποιήθηκε και υποδεικνύει το σημαντικό ρόλο της επιλογής κατάλληλου αλγορίθμου για μια εφαρμογή. Τρίτη είναι **η Βελτιστοποίηση μέσω Βελτιστοποίησης του Πηγαίου Κώδικα** που οδηγεί στο συμπέρασμα πως ένας "καλογραμμένος" Κώδικας, με τον οποίο αποφεύγονται περιττές πράξεις και προσπελάσεις μνήμης αποτελεί σημαντικό εργαλείο. Η τεχνική με τη μικρότερη συνεισφορά είναι η **χρήση των επιλογών Βελτιστοποίησης του Μεταγλωττιστή (Compiler),** η οποία ωστόσο αποτέλεσε και την πιο "εύκολη", από πλευράς χρόνου ενασχόλησης, τεχνική και αποτελεί έναν απλό τρόπο για τη βελτίωση της απόδοσης μιας εφαρμογής.

Μετά την ολοκλήρωση όλων των βελτιστοποιήσεων ο συνολικός χρόνος εκτέλεσης μειώνεται κατά **85.58%.** Ή διαφορετικά η εκτέλεση επιταχύνεται κατά **6.93 φορές.** Ωστόσο για την επίτευξη του στόχου της παρούσας εργασίας, που όπως αναφέρθηκε και στο εισαγωγικό κεφάλαιο (κεφάλαιο 1) είναι η αναπαραγωγή MPEG-2 βίντεο ανάλυσης SD (Standard Definition) σε πραγματικό χρόνο, πρέπει να είναι γνωστή και η συχνότητα λειτουργίας του επεξεργαστή. Η δυνατότητα αυτή δεν παρέχεται από το RVDS. Ωστόσο στην παράγραφο 2.2.2 αναφέρεται πως η συχνότητα λειτουργίας των Cortex A8 κυμαίνεται από 600MHz έως και πάνω από 1GHz. Από τον συνολικό αριθμό κύκλων μηχανής (**19,387,859,079 cycles**) που χρειάστηκε ο προσομοιωτής για να αποκωδικοποιήσει το βίντεο εισόδου των **2600 frame** (Input_Video.m2v), με χρήση του βελτιστοποιημένου κώδικα, προκύπτει το συμπέρασμα πως για να επιτευχθεί αναπαραγωγή MPEG-2 βίντεο ανάλυσης SD σε πραγματικό χρόνο, απαιτείται επεξεργαστής Cortex A8 με συχνότητα λειτουργίας τουλάχιστον (19,387,859,079 cycles / 2600 frames) * 30 frames/s = **223,706,066 cycles / s** ή περίπου **224MHz.** Συνεπώς ο στόχος επιτεύχθηκε. Μάλιστα είναι δυνατή η αναπαραγωγή βίντεο και **υψηλότερης ανάλυσης** σε πραγματικό χρόνο.

Σαν συνέχεια της παρούσας εργασίας, θα μπορούσε ο βελτιστοποιημένος MPEG-2 αποκωδικοποιητής να δοκιμαστεί σε ένα πραγματικό περιβάλλον αρχιτεκτονικής ARM Cortex A8. Στα πλαίσια της παρούσας εργασίας αυτό δεν ήταν εφικτό.

Η δουλειά ωστόσο που έγινε μπορεί να χρησιμοποιηθεί για περεταίρω έρευνα στη Βελτιστοποίηση βίντεο εφαρμογών στον Cortex αλλά και σε άλλες αρχιτεκτονικές. **Ο βελτιωμένος αλγόριθμος Fast IDCT** μπορεί να ενσωματωθεί σε εφαρμογές άλλων αποκωδικοποιητών που κάνουν χρήση IDCT. Επίσης η δουλεία που έγινε πάνω στην ελαχιστοποίηση της χρήσης if μπορεί να αποτελέσει οδηγό για ανάλογες περιπτώσεις. Ο τρόπος που χρησιμοποιήθηκαν **οι NEON εντολές** και η λύση που δόθηκε στα διάφορα προβλήματα υλοποίησης που προέκυψαν κατά τη χρήση τους μπορούν να χρησιμοποιηθούν και σε άλλες αρχιτεκτονικής που υποστηρίζουν SIMD.

Αναφορές - Βιβλιογραφία

[1] ***ITU-T Recommendation H.262***, *SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services – Coding of moving video*

[2] *Pyter Symes*, ***Digital Video Comprassion (2001)***

[3] *Andrew N. Sloss, Dominic Symes, Chris Wright, With a contribution by John Rayfield,* ***ARM System Developer's Guide Designing and Optimizing System Software***

[4] *http://infocenter.arm.com/help/index.jsp*