μεταφορά και επιτάχυνση του AVS προτύπου βίντεο
σε διπύρηνη ετερογενή αρχιτεκτονική

μεταπτυχιακός φοιτητής : δημήτρης ζαχαρής
υπεύθυνος καθηγητής : γεώργιος σταμούλης

πανεπιστήμιο θεσσαλίας 2007-2008

Ευχαριστίες

Ευχαριστώ τους καθηγητές μου, τους συνεργάτες μου, τους φίλους μου. Μία απλή αναφορά στο κείμενο αυτό εμένα θα τιμήσει περισσότερο εμένα παρά εκείνους. Ανξαρτήτως, ευχαριστώ τους γονείς μου για την πολυετή στήριξη.

Περιεχόμενα

Σ.Σ. : Η κατανόηση του παρόντος πνευματικού έργου έχει ως αναγκαία συνθήκη, γνώση βασικών εννοιών βίντεο , αρχιτεκτονικής υπολογιστών και ενσωματωμένων συστημάτων. Η ικανή - συνθήκη - έγκειται στην υπομονή του αναγνώστη.

Εισαγωγή

Τα τελευταία χρόνια, έχουμε δει την ανάδυση ενός αριθμού προτύπων που αφορούν ένα ευρύ φάσμα εφαρμογών βίντεο, από ασύρματες με χαμηλό ρυθμό μετάδοσης, έως high definition μεταδόσεις. Τα συστήματα αυτά έχουν υλοποιηθεί με τεχνολογίες ενός ή περισσότερων πυρήνων, γενικού σκοπού επεξεργαστές (GPPs) ή φιξαρισμένα ASICs. Οι απαιτήσεις της βιομηχανίας για υψηλής ποιότητας, υψηλής ευκρίνειας, αποκωδικοποίηση βίντεο σε πραγματικό χρόνο και συνήθως κάτω από περιορισμούς όπως αυτούς της χαμηλής κατανάλωσης ισχύος και του αριθμού πυλών, είναι μία πρόκληση που θέτει υπό δοκιμασίες την ικανότητα των πολυμεσικών αρχιτεκτονικών να φέρουν αποδοτικές λύσεις.

Οι επεξεργαστές υψηλής απόδοσης γενικού σκοπού (GPPs) προσφέρουν μία αρκετά ευέλικτη λύση υλοποιώντας πρότυπα βίντεο σε λογισμικό και κρύβοντας την οργάνωση του υλικού από τον developer που εργάζεται σε επίπεδο εφαρμογής. Αυτό επιτυγχάνεται με τη χρήση επεκτάσεων του ISA που υλοποιούν διανυσματικές λειτουργίες, οι οποίες είναι εξαιρετικά χρήσιμες σε τέτοιες εφαρμογές. Η χρήση όμως μίας τέτοιας λύσης δε συνίσταται για ενσωματωμένα συστήματα, για λόγους κατανάλωσης ισχύος και κόστους.

Τα *ASICs* από την άλλη πλευρά στοχεύουν σε ειδικές περιπτώσεις που απαιτούν πολύ υψηλό throughput ή πολύ χαμηλή κατανάλωση ισχύος. Όμως είτε ελάχιστες συνιστώσες τους μπορούν να προγραμματιστούν, είτε καμία, με αποτέλεσμα, εκτός του υψηλού κόστους ανάπτυξης να έχουν και πολύ χαμηλή ευελιξία ως συστήματα.

Για να συνδυαστούν τα πλεονεκτήματα των προαναφερθέντων τεχνολογιών, δηλαδή, ευελιξία των GPPs και απόδοση των ASICs, ASIPs όπως DSPs ή Multimedia Processors χρησιμοποιούνται για να λύσουν προβλήματα συνεχώς εξελισσόμενων προδιαγραφών όπως είναι ο τομέας του βίντεο. Η ανώτερη απόδοση των ASIPs κοστίζει όμως τη λύση του προβλήματος της υλοποίησης της εκάστοτε εφαρμογής στην εκάστοτε αρχιτεκτονική, άρα και την προσαρμογή σε διαφορετικό σύνολο εντολών. Το γεγονός αυτό μεγενθύνεται μιας και σε πολυπύρηνες αρχιτεκτονικές προσφέρονται ανεπαρκείς μεταγλωττιστές και προσομοιωτές αδύναμοι να παράγουν παραλληλισμό εργασιών διαβάζοντας την εφαρμογή.

Η εργασία αυτή περιγράφει τη λύση ενός τέτοιου προβλήματος. Συγκεκριμένα, τη μεταφορά και επιτάχυνση του AVS αποκωδικοποιητή βίντεο στον 388VDO Diamond dual core της Tensilica. Ξεκινώντας από μία υλοποίηση ανοιχτού κώδικα (τη μοναδική) για GPP αρχιτεκτονική -x86 Windows specific- με διαδοχικούς μετασχηματισμούς έγινε εφικτή η υλοποίηση σε ένα ASIP –Tensilica DC_388VDO.

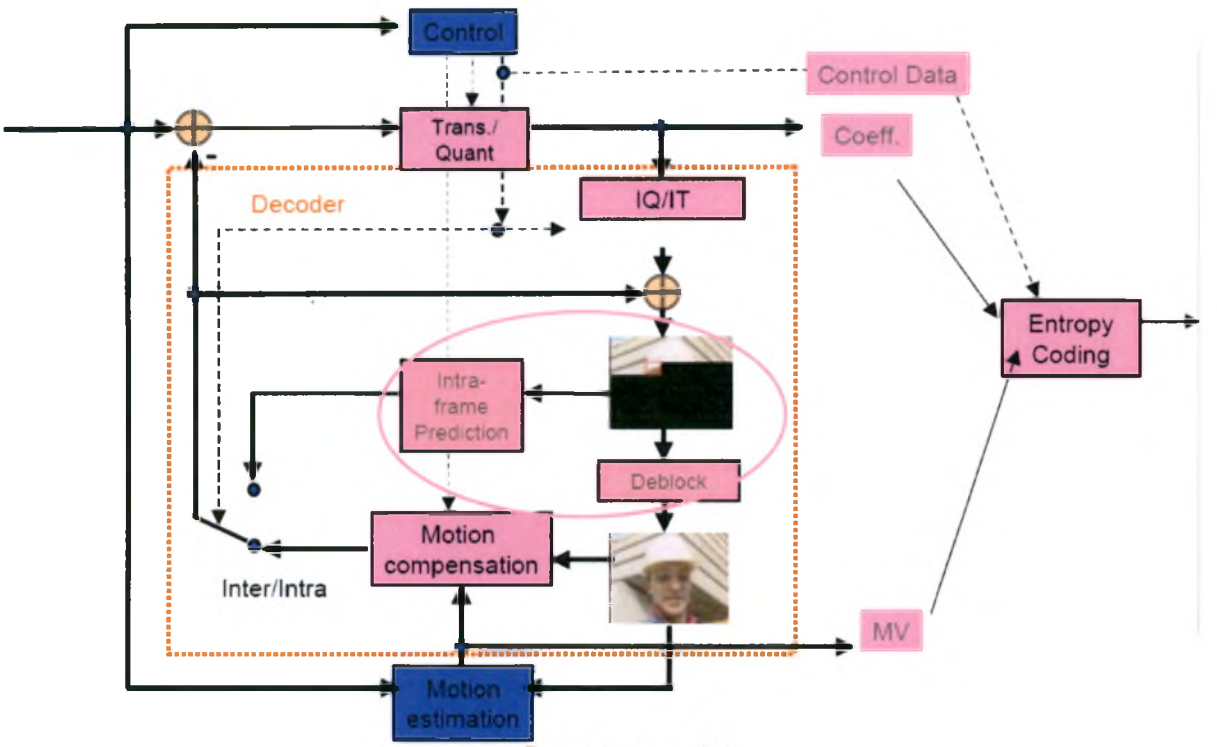
Ο Tensilica 388VDO Diamond περιλαμβάνει preconfigured εκδόσεις της 32μπιτης Xtensa configurable αρχιτεκτονικής, βελτιστοποιημένης για βίντεο κωδικοποίηση και αποκωδικοποίηση. Το συγκεκριμένο βελτιωμένο σύνολο εντολών υποστηρίζει όλα τα δημοφιλή video codecs όπως MPEG-4, H.264, VC-1 (όλα στο Main profile) για απόδοση μέχρι ανάλυση D1, δηλαδή 720x576x25 (PAL) ή 720x480x30 (NTSC) pixels/sec.

Όχι όμως ακόμη το AVS. Εκ φύσεως, το τελευταίο είναι βασισμένο στα προηγούμενα, χρονικά, πρότυπα. Συνεπώς, ένας από τους αρχικούς στόχους είναι η εξακρίβωση πως όντως το AVS μπορεί να μεταφερθεί στο DC_388VDO. Το βήμα αυτό γίνεται αφού μελετηθεί η πολυπλοκότητα των modules που αποτελούν το συνολικό αλγόριθμο του AVS αποκωδικοποιητή. Μιας και η εταιρία αυτή λειτουργεί με business-to-business proprietary-source μοντέλο, οι χρήστες του framework πάνω στη συγκεκριμένη πλατφόρμα δεν έχουν σχεδιαστικές επιλογές (πρόσθήκη/αφαίρεση στο σύνολο εντολών) και επίσης ο αριθμός των χρηστών δεν έχει τον όγκο open-source εργαλείων. Συνεπώς, αφ' ενός οι μέθοδοι επιτάχυνσης και βελτιστοποίησης περιορίζονται, αφ' ετέρου η διαδικασία απασφαλμάτωσης εντός του συγκεκριμένου framework δυσχεραίνει εξαιρετικά, μιας και το τελευταίο δοκιμάζεται από μικρό κοινό. Έτσι ένας ακόμη στόχος είναι η καταγραφή ενός οδηγού ανάπτυξης για developers που επιθυμούν να εφαρμόσουν task-level παραλληλισμό οποιουδήποτε video-decoder σε ένα πολυ-πύρηνο σύστημα, προτείνοντας - προαιρετικά - προσθήκες στο, ειδάλλως, κλειστό σύνολο εντολών. Οι περισσότερες, πάντως, από τις τεχνικές επιτάχυνσης που περιγράφονται παρακάτω είναι εφαρμόσιμες σε οποιοδήποτε επεξεργαστή που υποστηρίζει, τουλάχιστον, SIMD εντολές.

Η πρόκληση, λοιπόν, που αντιμετωπίζεται κατά τη μεταφορά ενός νέου αποκωδικοποιητή βίντεο σε ένα ASIP σύστημα, είναι ο εντοπισμός και η εκμετάλλευση κάθε είδους παραλληλισμού, σε όλα τα επίπεδα ανάλυσης. Ειδικότερα, γίνεται χρήση παραλληλισμού σε επίπεδο pixel block (SIMD και instruction parallelism) αλλά και σε επίπεδο module (task και pipeline parallelism). Επίσης, αναλύεται η συνεισφορά κάθε μετασχηματισμού παραλληλισμού στη συνολική επιτάχυνση. Τελικός στόχος είναι να επιτύχουμε real-time D1 (720x576x25 συγκεκριμένα) AVS αποκωδικοποίηση.

Εν τέλει, παράγονται οφέλη τεχνολογικής αλλά και εργασιακής φύσης, ανοίγοντας νέες διαστάσεις βελτιστοποίησης της παραπάνω ιδέας, ίσως σε θέματα αρχιτεκτονικής και ισχύος, αλλά και ευρύτερα σχεδιαστικά σε επίπεδο engine node, όπως ο συνδυασμός πολλών συστημάτων DC_388VDO.

Το σύστημα : AVS Video Standard



AVS Block Diagram {1}

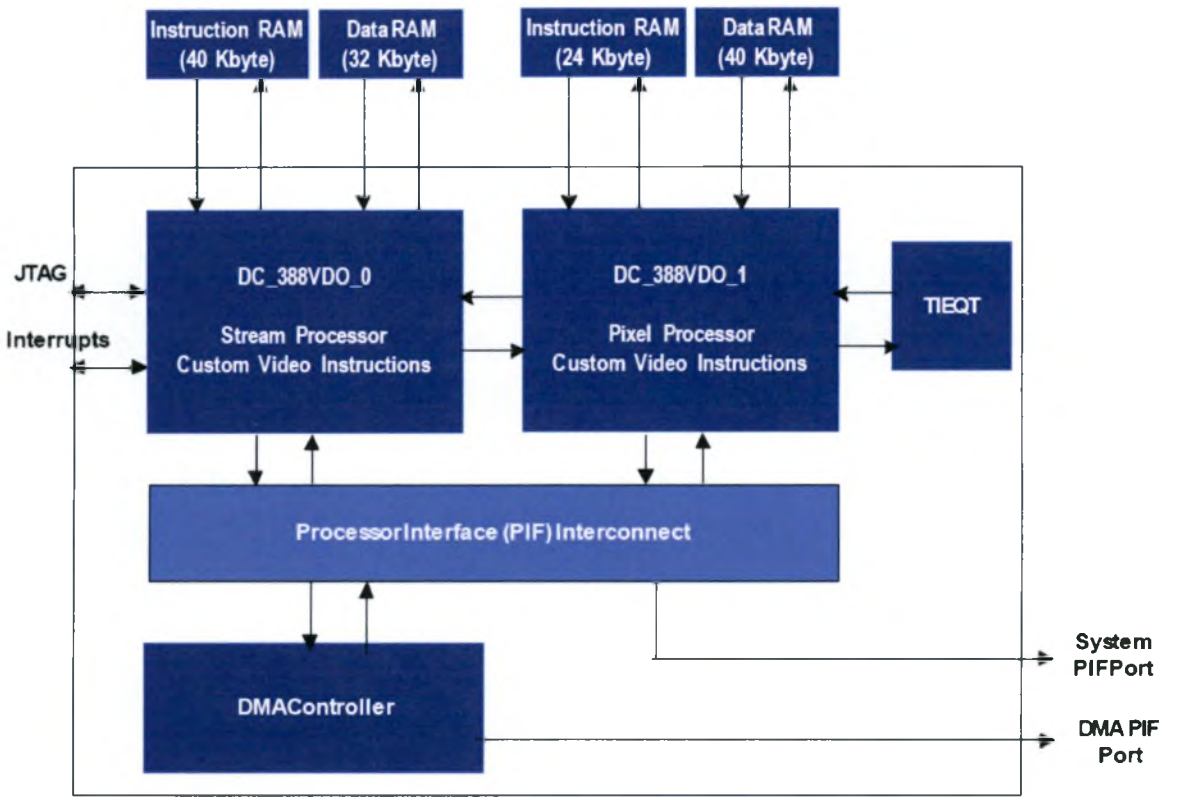
Το Audio Video Coding Standard (AVS) Κινεζικό γκρουπ εργασίας ανέπτυξε το ομώνυμο πρότυπο, το πρώτο οπτικοακουστικό πρότυπο που αναπτύχθηκε ανεξάρτητα και εξολοκλήρου στην Κίνα. Το πρώτο προσχέδιο, το οποίο ολοκληρώθηκε το 2003, αρχικά στόχευσε σε high definition, high quality βίντεο, είτε για εφαρμογές broadcast, είτε για αποθήκευση σε ψηφιακά μέσα. Ταυτόχρονα, όμως, επιτυγχάνει μεγαλύτερη αποδοτικότητα και μικρότερη πολυπλοκότητα συγκρινόμενο με άλλα πρότυπα κωδικοποίησης βίντεο όπως τα MPEG-2, MPEG-4 και H.264/AVC.

Αν και μοιάζει με το H.264 σε επίπεδο λειτουργικών μονάδων (functional modules point of view, {1}), χρησιμοποιεί διαφορετικές τεχνικές υλοποίησης. Μερικές από τις ουσιώδεις διαφορές παρουσιάζονται στην επόμενη σελίδα.

tools	AVS	H.264	MPEG-2
Intra interpolation	8x8, 5 modes for Y prediction, 4 modes for UV prediction	4x4, 9 modes for Y prediction, 4 modes for UV prediction	Prediction only to DC coefficients
Reference Frame	<=2	<=16	1
MC Block-size	16x16, 16x8, 8x16, 8x8	16x16, 16x8, 8x16, 8x8,8x4,4x8,4x4	16x16, 16x8(interlace)
¼ pixel MC	½ pixels 4-tap ¼ pixels 4-tap	½ pixels 6-tap ¼ pixels 4-tap	½ pixels 2-tap
Transform and Quantization	8x8 integer transform, encoding site normalization only	4x4 integer transform, both encoding and decoding sites need to normalize	8x8 float DCT
Entropy coding	Adaptive 2D VLC	CAVLC, CABAC	VLC
Loop filter	8x8 based Less boundaries Less BS-levels (0..2), Less pixels filtered (p0, p1, q0, q1)	4x4 based More boundaries More BS-levels (0..4), More pixels filtered (p0..p3,q0..q3)	N/A

AVS indicative differences from other standards {2}

Το σύστημα : Diamond 388VDO engine

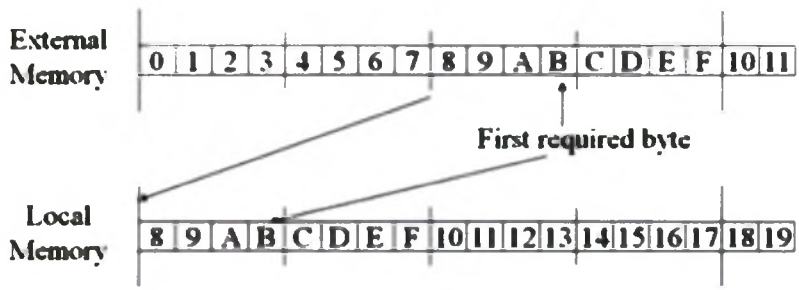


Diamond Video Engine 388VDO block diagram {3}

Ο 388VDO Diamond είναι μία ετερογενής διπύρηνη επεξεργαστική μηχανή που αποτελείται από δύο πυρήνες Xtensa LX με ενσωματωμένες στο σύνολό εντολών τους λειτουργίες στοχευμένες στην επεξεργασία βίντεο. Ο συγκεκριμένος επεξεργαστής, Xtensa LX2, έχει αρχιτεκτονική σχεδιασμένη για embedded εφαρμογές. Πιο συγκεκριμένα:

- Το Diamond Video Engine βασίζεται σε:
  - Δύο Xtensa LX επεξεργαστές. Ο πρώτος ονομάζεται Stream processor, ο δεύτερος Pixel Processor. Τα αρχιτεκτονικά χαρακτηριστικά περιλαμβάνουν:
    - 32-bit αρχιτεκτονική, big-endian byte διάταξη
    - 5-stage pipeline
    - 16/24/32-bit εντολές
    - 32-entry φάκελος καταχωρητών
    - Διαφορετικές αρχιτεκτονικές **επεκτάσεις** για κάθε πυρήνα χρησιμοποιώντας την επεκτάσιμη και configurable σχεδίαση των Xtensa LX. Συμπεριλαμβάνονται εξειδικευμένες εντολές βίντεο, φάκελοι καταχωρητών, ports, και ουρές.(από εδώ και στο εξής θα αναφερόμαστε σε όλα αυτά με τον όρο TIEs)
  - Δύο 32-bit Processor Interface (PIF) buses
    - System PIF: επιτρέπει άμεση πρόσβαση στην κύρια μνήμη από τους επεξεργαστές
    - DMA PIF: επιτρέπει στο DMA engine να μεταφέρει μεγάλα τμήματα μνήμης από και προς την κύρια μνήμη.
  - Τέσσερις τοπικές μνήμες που πρέπει να είναι προσαρτημένες στα τοπικά memory ports του 388VDO

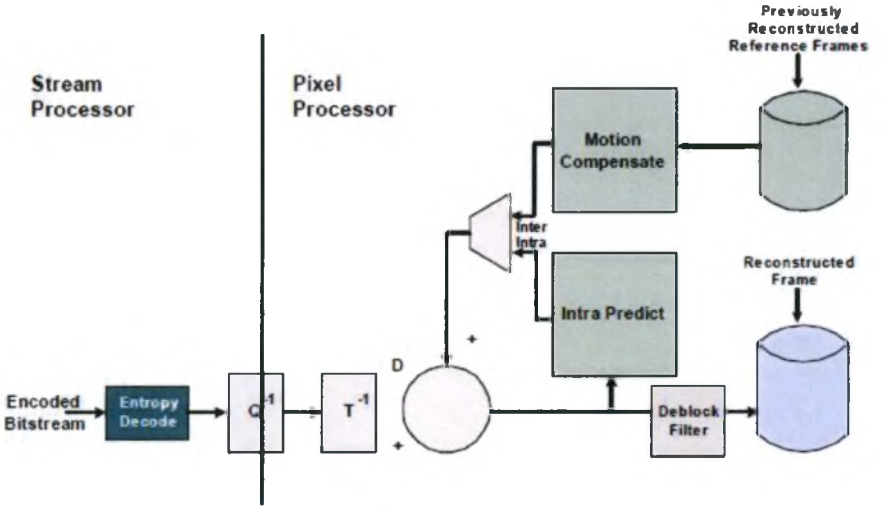
    - 388VDO_0 instruction RAM: 40Kbytes (32 bits wide)
    - 388VDO_0 data RAM: 32Kbytes (128 bits wide)
    - 388VDO_1 instruction RAM: 24Kbytes (32 bits wide)
    - 388VDO_1 data RAM: 40Kbytes (128 bits wide)
- 5 εισερχόμενα interrupts που μπορούν να χρησιμοποιηθούν από έναν System Controller
- Ένα εξερχόμενο interrupt request pin που μπορεί να χρησιμοποιήσει το 388VDO για να δημιουργήσει interrupts στο System Controller
- Low-power mode στο οποίο οι περισσότεροι καταχωρητές στο Diamond Video Engine έχουν το ρολόι τους switched off
- On-Chip debug (OCD) που χρησιμοποιεί ένα IEEE 1149.1 (JTAG) interface port.
- DMA
    - Η DMA μπορεί να φορτώσει N γραμμές των M 32 bit λέξεων
    - Υποστηρίζονται ξεχωριστά source και destination stride και άρα έχουμε υποστήριξη 2-D σε 1D και 1D σε 2D DMA λειτουργίες.
    - Υπάρχει μερικό alignment, δηλαδή οι unaligned προσβάσεις στη μνήμη περιορίζονται σε:



DMA alignment capability diagram {4}

- TIEQT
    - Είναι μία μονάδα που εκτελεί τη λειτουργία transpose. Ουσιαστικά είναι μία συστάδα καταχωρητών οργανωμένων σε ουρές, που επιτρέπει πρόσβαση σε μία ομάδα τιμών με συγκεκριμένη σειρά, η οποία μας βολεύει...
    - Υποστηρίζει transpose χωρίς να χρειάζεται πρόσβαση στη μνήμη.
    - Υποστηρίζει διάφορα addressing modes, όπως butterfly accesses και wrap around.
    - Η όλη λειτουργία του βασίζεται στους SIMD καταχωρητές του D.V.Engine

Τέλος, οι βίντεο λειτουργίες των επεξεργαστών του 388VDO engine έχουν προκατανεμηθεί με βάση τα ήδη υλοποιημένα πρότυπα ως εξής:



DV388 Partitioning Diagram {5}

Ροή Σχεδίασης

I. Απεγκλωβισμός OpenAVS source code από Windows specific C code
II. Μεταφερσιμότητα (Foreseeing Optimizations: τροποποίηση της software αρχιτεκτονικής, ώστε να ελαχιστοποιηθούν αργότερα οι μεταφορές δεδομένων μεταξύ των μνημών)
III. Υλοποίηση με TIEs (Foreseeing Optimizations ενεργές και εδώ για την εκμετάλλευση των πλεονεκτημάτων της συγκεκριμένης αρχιτεκτονικής αργότερα)
IV. Διαχωρισμός κώδικα, τοποθέτηση κώδικα-δεδομένων : SingleCore Model
V. Έλεγχος απόδοσης. Αν δεν ικανοποιείται ο ενδιάμεσος στόχος πήγαινε στο 3
VI. Οριστικός διαχωρισμός κώδικα σε δύο εκτελέσιμα
VII. Τοποθέτηση «σημαντικών» δεδομένων στις τοπικές μνήμες
VIII. Χρήση memcpy συναρτήσεων C και του Diamond Video Engine global addressing για μεταφορές δεδομένων μεταξύ των τοπικών μνημών, της εξωτερικής και μίας εκ των τοπικών και το αντίστροφο
IX. Εφαρμογή συγχρονισμού χρησιμοποιώντας spin-locks σε κοινή εξωτερική μνήμη
X. Μετατροπή memcpy σε DMA λειτουργίες
XI. Εισαγωγή buffering scheme και ανεξάρτητη εκτέλεση μοντέλων πυρήνων με spin-locks σε τοπικές μνήμες.
XII. Έλεγχος απόδοσης. Αν δεν ικανοποιείται ο τελικός στόχος, μελετούμε αρχικά τις μεταφορές δεδομένων μεταξύ των μνημών και τις ελαχιστοποιούμε. Αν αυτές είναι εντάξει, βελτιστοποιούμε περισσότερο τα επιμέρους modules με σειρά βαρύτητας και περιθωρίου κέρδους.

Μεταφερσιμότητα κώδικα

Ο αρχικός πηγαίος κώδικας OpenAVS είναι και η μόνη διαθέσιμη υλοποίηση στο ευρύ κοινό αυτή τη στιγμή. Ο κώδικας αυτός είναι γραμμένος για λειτουργικό Windows και αρχιτεκτονική x86, οπότε και χρησιμοποιεί συναρτήσεις που είναι εξειδικευμένες για το συγκεκριμένο σύστημα. Για να μπορεί ο κώδικας αυτός να είναι ανεξάρτητος αρχιτεκτονικής και λειτουργικού συστήματος, πρέπει να απεγκλωβιστεί από τις εξειδικευμένες κλήσεις.

Αρχικά, συγχωνεύθηκαν η βιβλιοθήκη και ο player της εφαρμογής σε ένα εκτελέσιμο βασικό video stream decoder σε αποσυμπιεσμένο .yuv αρχείο. Ενσωματώθηκε η επιλογή απενεργοποίησης εγγραφών δεδομένων σε αρχείο, η οποία υπάρχει για να μπορεί να γίνει έλεγχος ποιότητας της εξόδου του αποκωδικοποιητή, ώστε να μπορεί να γίνει μέτρηση της απόδοσης ανεξάρτητης από την εγγραφή σε αρχείο.

Το OpenAVS χρησιμοποιεί τη C99 stdint.h βιβλιοθήκη, η οποία αντικαταστάθηκε από μία open source έκδοση της stdint.h, ώστε να υποστηρίζεται από τους μεταγλωττιστές C σε Linux και Xtensa. Για τον ίδιο λόγο κάποιοι τύποι δεδομένων ορίστηκαν στο define.h, μιας και προηγουμένως ήταν Windows-specific. Πλέον, με μεταγλώττιση υπό συνθήκη, εκτός από το προαναφερθέν ζήτημα, αντιμετωπίζεται και η χρήση __inline__ (__inline στην MS-C) και το endianness (__BYTE_ORDER ,__LITTLE_ENDIAN σε x86, __BIG_ENDIAN σε XTENSA).

Unaligned double word προσβάσεις υποστηρίζονταν μέσω endianess-specific masking. Οι προσβάσεις αυτές άλλαξαν ώστε να είναι ανεξάρτητες του byte ordering.

Το αρχικό benchmark αρχείο, το stream_short.avs, απέκτησε EOS (End of Stream) ώστε να είναι απολύτως συμβατό με το πρότυπο.

Διορθώθηκε ένα λάθος του αρχικού κώδικα, ο οποίος κρατούσε ένα reference frame σε έναν εσωτερικό buffer και έτσι, έβγαζε στην έξοδο το πρώτο (I) frame δύο φορές, γεγονός ασύμβατο με το πρότυπο. Αντί αυτού, το τελευταίο frame έμενε πάντα σε αυτό τον εσωτερικό buffer. Το API του κώδικα επεκτάθηκε ώστε να επηρεαστεί ελάχιστα το υπόλοιπο και να μπορεί να εξαχθεί αυτό το τελευταίο frame (μιας και η συνάρτηση που αποκωδικοποιεί ένα frame βρίσκεται πολύ ψηλά στο δέντρο κλήσεων συναρτήσεων).

Διορθώθηκε ένας αριθμός από λάθη στον αρχικό OpenAVS κώδικα που παρήγαγε λανθασμένη έξοδο YUV (μερικά από αυτά ορατα στην αναπαραγωγή των ακολουθιών εξόδου).

Φυσικά, αφαιρέθηκε ένας μεγάλος αριθμός σχολίων του αρχικού κώδικα που ήταν Unicode (Chinese) χαρακτήρες οι οποίοι προκαλούν warnings.

Foreseeing software optimizations

Προτού προσθέσουμε τη χρήση TIEs, έγινε προσπάθεια να βελτιωθεί η συμπεριφορά πρόσβασης στη μνήμη του αρχικού AVS αποκωδικοποιητή. Οι αλλαγές αυξάνουν το spatial locality των προσβάσεων στη μνήμη και είναι ιδιαίτερα χρήσιμες για το τελευταίο τμήμα της ροής σχεδίασης που απαιτεί ρητή, εκκινούμενη σε επίπεδο εφαρμογής, μετακίνηση των δεδομένων μεταξύ των μνημών των δύο πυρήνων και της εξωτερικής μνήμης. Οι περισσότερες από τις τροποποιήσεις αυτές είναι χρήσιμες για κάθε μεταφορά κώδικα από μία GPP υλοποίηση σε μία ενσωματωμένου συστήματος. Απαλείφουν υπολογισμούς που γίνονται κατ' επανάληψη χωρίς λόγο και βελτιστοποιούν την επαναχρησιμοποίηση δεδομένων στις τοπικές μνήμες. Εντοπίζονται τα δεδομένα που «καταναλώνονται» συχνά και αντιγράφονται στις τοπικές μνήμες. (Ο κύριος όγκος των αλλαγών αυτών αφορούν, φυσικά, το Motion Compensation module, το οποίο επιφέρει και τη μεγαλύτερη συσχέτιση μεταξύ όγκων δεδομένων οι οποίοι δε χωρούν αυτούσιοι στις τοπικές μνήμες, αλλά και το Deblocking Filter module το οποίο εφαρμόζεται σε γειτονικά blocks.)

Υλοποίηση με TIEs

Στη συνέχεια θα παρατεθούν μερικά παραδείγματα υλοποίησης με Tensilica Instruction Extensions ώστε αφεθεί στον αναγνώστη ένα επίστρωμα της λογικής που ακολούθησε η βελτιστοποίηση στο επίπεδο αυτό.

Bitstream parsing

Η συνάρτηση που χρησιμοποιείται περισσότερο στο κομμάτι αυτό είναι η read_bits(const AVS_BYTE*, AVS_DWORD*, AVS_INT). Χρησιμοποιείται κατά κόρον, μιας και όλα τα μπιτ του ρεύματος πρέπει να διαβαστούν. Η συνάρτηση αυτή βελτιστοποιήθηκε μέσω υλοποίησης με TIEs, τα οποία δε χρησιμοποιούν παρά ένα όρισμα στη συνάρτηση: xvd_bs_loadgetbits(numbits).

VLD

Το Variable Length Decoding στο AVS χαρακτηρίζεται ως Adaptive 2D VLC. Συγκεκριμένα χρησιμοποιούνται k-exp-Golomb κώδικες, 0<=k<=3 για όλα τα συντακτικά στοιχεία(syntax elements) με Run-Length ζεύγη. Χρησιμοποιούνται escape values για να απεικονίσουν σύμβολα μικρής πιθανότητας. Συνολικά, χρησιμοποιούνται 19 πίνακες απεικόνισης, ενώ κάποια από τα συντακτικά στοιχεία αποκωδικοποιούνται απευθείας, χωρίς απεικόνιση, με πράξεις που κοστίζουν τελικά περισσότερο (αλλά που λόγω σπανιότητας δε μας ενδιαφέρει να τα διατηρούμε στη μνήμη). Στο openAVS το inverse scan table είναι διδιάστατο, δηλαδή χρησιμοποιούνται δύο indexes για να δώσουν τη θέση ενός συντελεστή στο Macroblock. (το inverse scan γίνεται απευθείας μετά το VLD στον κώδικα, όπως και το inverse quantization).

Οι βελτιώσεις που έγιναν ήταν συνοπτικά οι εξής:

- Buffering των ανακτημένων τιμών από το bitstream ώστε να γίνει κατόπιν το Inverse SCAN
- Μετατροπή του αρχικού δυσδιάστατου πίνακα ISCAN που επιστρέφει τα indexes των coefficients σε μονοδιάστατο. Άμεση φόρτωση χωρίς την ανάγκη αύξησης του offset.(loads/2)
- «Πακετάρισμα» των ζευγών Run-Length στους 19 πίνακες(κατά τη δημιουργία τους) ώστε να μειωθεί το πλήθος των Loads από τη μνήμη έπειτα, αλλά και η χρήση τοπικής μνήμης(μιας και το εύρος τιμών τους επιτρέπει κάτι τετοιο). Χρήση των formats και των TIEs που παρέχει η αρχιτεκτονική ώστε να αποφύγουμε το manual shifting-masking αλλά και τον υπολογισμό των offsets κατά τη φόρτωση.(-2 πράξεις ανά συντελεστή, loads/2)
- Χρήση των FLIX εντολών(VLIW-like 2-slot parallel execution).Compiler dependent.

Αντίστροφος Μετασχηματισμός

Ο Αντίστροφος Μετασχηματισμός (Inverse Transformation) μετατρέπει έναν 8x8 πίνακα από συντελεστές μετασχηματισμού σε ένα 8x8 residual δείγμα δεδομένων, χρησιμοποιώντας την ακόλουθη διαδικασία, η οποία αποτελείται από δύο μονοδιάστατους μετασχηματισμούς:

a) Δεδομένου του πίνακα συντελεστών Coeff ο οριζόντιος αντίστροφος μετασχηματισμός παράγει τον πίνακα H'=Coeff x T_8^T όπου T_8^T είναι ο ανάστροφος του πίνακα μετασχηματισμού T_8.
b) Ένας νέος πίνακας H'' παράγεται από την πρόσθεση του 4 σε κάθε στοιχείο του H' και ολισθαίνοντας δεξιά κατά 3.

c) Ο πίνακας H βγαίνει ως εξής : $H = T_8 \times H''$

Ο πινακας μετασχηματισμού δίνεται από :

$$T_8 = \begin{Bmatrix} 8 & 10 & 10 & 9 & 8 & 6 & 4 & 2 \\ 8 & 9 & 4 & -2 & -8 & -10 & -10 & 6 \\ 8 & 6 & -4 & -10 & -8 & 2 & 10 & 9 \\ 8 & 2 & -10 & -6 & 8 & 9 & -4 & -10 \\ 8 & -2 & -10 & 6 & 8 & -9 & -4 & 10 \\ 8 & -6 & -4 & 10 & -8 & -2 & 10 & -9 \\ 8 & -9 & 4 & 2 & -8 & 10 & -10 & -6 \\ 8 & -10 & 10 & -9 & 8 & -6 & 4 & -2 \end{Bmatrix}$$

Τα στοιχεία του H πρέπει να περιορίζονται στο εύρος $[-2^{15}, 2^{15}-1]$. Οι τελικές τιμές των συντελεστών residual δίνονται από: $r_{ij} = (h_{ij}+2^6)>>7$

Η αρχική υλοποίηση σε C εκμεταλλεύεται τη συμμετρική μορφή του πίνακα μετασχηματισμού για να ελαττώσει τον αριθμό εντολών στο μισό και στις δύο κατευθύνσεις (Αλγόριθμος Loeffler). Οι απλοί αριθμητικοί συντελεστές επιτρέπουν την αντικατάσταση των εσωτερικών γινομένων με μία σειρά προσθέσεων και ολισθήσεων σε ένα σχήμα «πεταλούδας».

Η υλοποίηση με TIEs εξαφανίζει το for-loop που γίνεται ανά 8x8 μπλοκ, και το ξεδιπλώνει 8 φορές, χρησιμοποιώντας τους SIMD καταχωρητές. Έτσι, ο πίνακας Coeff διαβάζεται σε 8 κύκλους πλέον και τοποθετείται σε 8 SIMD καταχωρητές. Ο TIE κώδικας που υλοποιεί το μετασχηματισμό διαβάζει κάθε γραμμή σε ένα SIMD καταχωρητή και κάνει το reshuffling ταυτόχρονα, χρησιμοποιώντας την xvd_ld_8x24_xu TIE.

Συνεπώς, το υπόλοιπο του υπολογισμού γίνεται απλό, αφού αποδειχθεί η ισοδυναμία των πράξεων στη C και σε TIEs. Ένα θέμα το οποίο εμφανίζεται είναι πως σε κάποια σημεία του υπολογισμού απαιτούνται αναγκαστικά περισσότεροι από 8 καταχωρητές, μιας και υπάρχουν περισσότερες από 8 διανυσματικές μεταβλητές. Αυτό υποχρεώνει τον προγραμματιστή - ή το μεταγλωττιστή τελικά - να τοποθετήσει κάποια μεταβλητή στη μνήμη για μετέπειτα χρήση.

Για λόγους ανάπτυξης που ελαχιστοποιούν την απασφαλμάτωση, ακολουθούμε την εξής ροή υλοποίησης:

1. Κάθετος μετασχηματισμός
2. Αναστροφή και αποθήκευση στη μνήμη
3. Οριζόντιος (αλλά πάλι λειτουργόντας σε κάθετη ροή λόγω TIEQT) μετασχηματισμός
4. Αναστροφή και αποθήκευση στη μνήμη

Φορτώνοντας από το TIEQT, τη μονάδα αναστροφής, πρέπει να προσέχουμε μιας και είναι απαραίτητη μία επέκταση προσήμου πριν από αριθμητικές πράξεις. Ένα πρόβλημα ακόμη ήταν πως τελικά ήταν αδύνατο να μην κάνουμε spill-over στη μνήμη. Έχοντας υπολογίσει και τα 8 αποτελέσματα, τα οποία βρίσκονται φυσικά σε καταχωρητές, στο AVS πρέπει να προσθέσουμε το 64. Αντίστοιχη πράξη add_i όμως δεν υπάρχει (υποστηρίζεται το εύρος [-8,7]). Συνεπώς πρέπει να χρησιμοποιήσουμε έναν ακόμη καταχωρητή SIMD για να αποθηκεύσουμε εκεί το ζητούμενο νούμερο.

Motion Compensation

Σε αντίθεση με το H.264 που υποστηρίζει μπλοκς με μέγεθος μέχρι και 4x4, το AVS υποστηρίζει μόνο 4 μεγέθη, 16x16, 8x16, 16x8, 8x8 αφού μικρότερα μπλοκ σπάνια χρησιμοποιούνται σε υψηλής ανάλυσης κωδικοποίηση βίντεο. Η ακρίβεια των motion vectors είναι quarter pixel για τις συνιστώσες luma και 1/8 pixel για τις chroma. Αφού δεν υπάρχουν δείγματα σε sub-sample θέσεις πρέπει αυτά να υπολογιστούν από τα γειτονικά τους. Έτσι, το 40% περίπου του MC module δαπανείται στο quarter pixel interpolation.

Ανάμεσα στα κύρια χαρακτηριστικά του interpolation είναι ο σημαντικός βαθμός επαναχρησιμοποίησης των ήδη υπολογισμένων δειγμάτων. Μία ουτοπική λύση θα ήταν να αποθηκεύουμε όλα τα υπολογισμένα δείγματα. Στοχεύθηκε, λοιπόν, η χρήση των εκάστοτε διαθέσιμων δεδομένων για τον υπολογισμό όσο το δυνατόν περισσότερων δειγμάτων. Σε αυτή τη λογική υλοποιήθηκε ένα software "pipeline" τέτοιο ώστε να μη χρειάζεται επαναφόρτωση του ίδιου pixel ακόμη κι αν χρειαστεί να υπολογίσουμε μία τιμή half pixel παραπάνω από μία φορά. Με τη μέθοδο αυτή ελαχιστοποιήθηκε ο αριθμός των φορτώσεων από τη μνήμη.

Μία ακόμη σημείωση αφορά την ευθυγράμμιση των δεδομένων(data alignment). Το MC χρειάζεται πολλαπλές φορτώσεις από θέσεις μνήμης που δεν είναι ευθυγραμμισμένες με τα vectors μας, το οποίο είναι εμπόδιο στην υλοποίηση με SIMD εντολές. Η υποστήριξη διάφορων μορφών μη ευθυγραμμισμένων φορτώσεων από την αρχιτεκτονική στη συγκεκριμένη περίπτωση έδωσε τη λύση. Χωρίς τη βοήθεια τέτοιων εντολών ο developer σε διαφορετικό σύστημα θα είχε σοβαρό πρόβλημα απόδοσης.

Deblocking Filter

Εδώ χρησιμοποιήθηκαν τεχνικές παρόμοιες με του Αντίστροφου Μετασχηματισμού για το ξεδίπλωμα βρόχων και την διανυσματοποίηση των υπολογισμών. Επιπλέον όμως, χρησιμοποιήθηκαν predicates για την υποθετική εκτέλεση των διαφόρων τμημάτων του κώδικα. Σημαντικό είναι να τονίσουμε τη διαφορά απόδοσης στο οριζόντιο από το κάθετο DF. Ενώ στο οριζόντιο η επιτάχυνση ήταν 6.5 σε σχέση με το αρχικό OpenAVS, στον κάθετο ήταν μόλις 2.5. Η διαφορά αυτή οφείλεται στην επιβάρυνση της πολύ συχνής αλλά σποραδικής χρήσης του TIEQT για την αναστροφή γραμμών.

(Ένα θέμα στη χρήση του TIEQT είναι η χρήση περιορισμών σε μερικές βελτιστοποιήσεις μεταγλωττιστών. Το TIEQT θεωρείται από τους προσομοιωτές εξωτερικός πυρήνας (τρίτος) και άρα η αναδιάταξη των εντολών φόρτωσης και αποθήκευσης ο κώδικας μπορεί να αποφέρει λογικό λάθος. Αυτό οφείλεται στο Release Memory Consistence μοντέλο που χρησιμοποιεί ο Diamond Core. Για την αντιμετώπιση αυτού, χρησιμοποιήσαμε #pragma no_reorder για να ενημερώνουμε το μεταγλωττιστή. Επίσης κάναμε flush το pipeline ώστε να εξαλείψουμε διάφορες εξαρτήσεις δεδομένων :#pragma flush.)

Αντίστροφη Κβαντοποίηση

Η αντίστροφη κβαντοποίηση χρησιμοποιεί έναν πίνακα από τιμές για να κάνει την πράξη Q_px$Quant(Q_p)$ που απαιτείται. Η μόνη TIE που πιθανά να μπορούσε να χρησιμοποιηθεί είναι η xvd_cav_calcquant. Αυτή χρησιμοποιεί την xvd_ca_quant κατάσταση η οποία είναι 8μπιτη και άρα δεν μπορεί να εκφράσει το πεδίο τιμών του πίνακα αντίστροφης κβαντοποίησης που έχει τιμές μέχρι και 60099. Εξαιτίας αυτού, η AVS Αντίστροφη Κβαντοποίηση δεν μπορεί να υλοποιηθεί με τις υπάρχουσες TIEs αποδοτικά.

Single Core σχήμα και αρχικές τοποθετήσεις κώδικα-δεδομένων

Στο σημείο αυτό, στόχος ήταν να βρούμε το μέγιστο δυνατό κέρδος που μπορούσαμε να έχουμε από τη χρήση των τοπικών μνημών. Έτσι, χρησιμοποιώντας τη διαδικασία που περιγράφεται στο Xtensa Linker Support Package manual, ανακατευθύναμε όλες τις εντολές του κώδικα στην εσωτερική SRAM ενώ τοποθετήσαμε τα πιο κρίσιμα δεδομένα στην εσωτερική μνήμη.

Παρατηρήθηκε, αρχικά, πως μεγάλο μέρος του χρόνου εκτέλεσης ήταν instruction fetches από την εξωτερική μνήμη. Αυτό οφειλόταν στη χρήση συναρτήσεων βιβλιοθηκών όπως η memcpy() αλλά και τελεστών όπως __mulsi3, __umodsi3, udivsi3. Προσωρινά τοποθετήθηκαν επίσης στην τοπική μνήμη και κατόπιν απαλείφθηκαν. Τέλος, στην εσωτερική μνήμη τοποθετήθηκε και η στοίβα.

Παρόλα αυτά, είναι προφανές πως από τον προκαθορισμένο διαχωρισμό των λειτουργιών ανάμεσα στους δύο πυρήνες, υπάρχει ανισορροπία στο φορτο, που γίνεται ακόμη μεγαλύτερη στην περίπτωση της αποκωδικοποίησης, μιας και δεν υπάρχει το Motion Estimation το οποίο θα τοποθετούσαμε στον Stream Processor. Οι συναρτήσεις που περιείχαν τμήματα κώδικα που στο dual-core μοντέλο θα τρέξουν σε διαφορετικούς πηρήνες αντικαταστάθηκαν.

Οριστικός διαχωρισμός κώδικα σε δύο εκτελέσιμα και τοποθέτηση στις τοπικές μνήμες. Strict Lockstep.

Φυσικά, αρχικά γίνεται ο διαχωρισμός σε δύο εκτελέσιμα με γνώμωνα την ελαχιστοποίηση των μεταφορών δεδομένων, την οποία και είχαμε προσχεδιάσει, αλλά και τον ισορροπημένο διαχωρισμό σε φόρτο εργασίας. Ο Pixel Processor που λειτουργεί ως σκλάβος, δεν έχει όμοια επεξεργαστική ισχύ και πρόκειται να εκτελεί διαφορετικές εργασίες από τον Stream Processor, έξου και η ετερογενής επεξεργαστική μηχανή. Περιμένει «δουλειά» από τον Stream Processor ή ένα σήμα τερματισμού. Ο Stream Processor αντίστοιχα επεξεργάζεται το bitstream και «πακετάρει» τα δεδομένα που χρειάζεται ο Stream Processor για να εκτελέσει τις λειτουργίες του.

Η αρχική μεταφορά των δεδομένων γίνεται με memcpy() στις κατάλληλες διευθύνσεις μνήμης. Αυτές είναι οργανωμένες ως εξής :

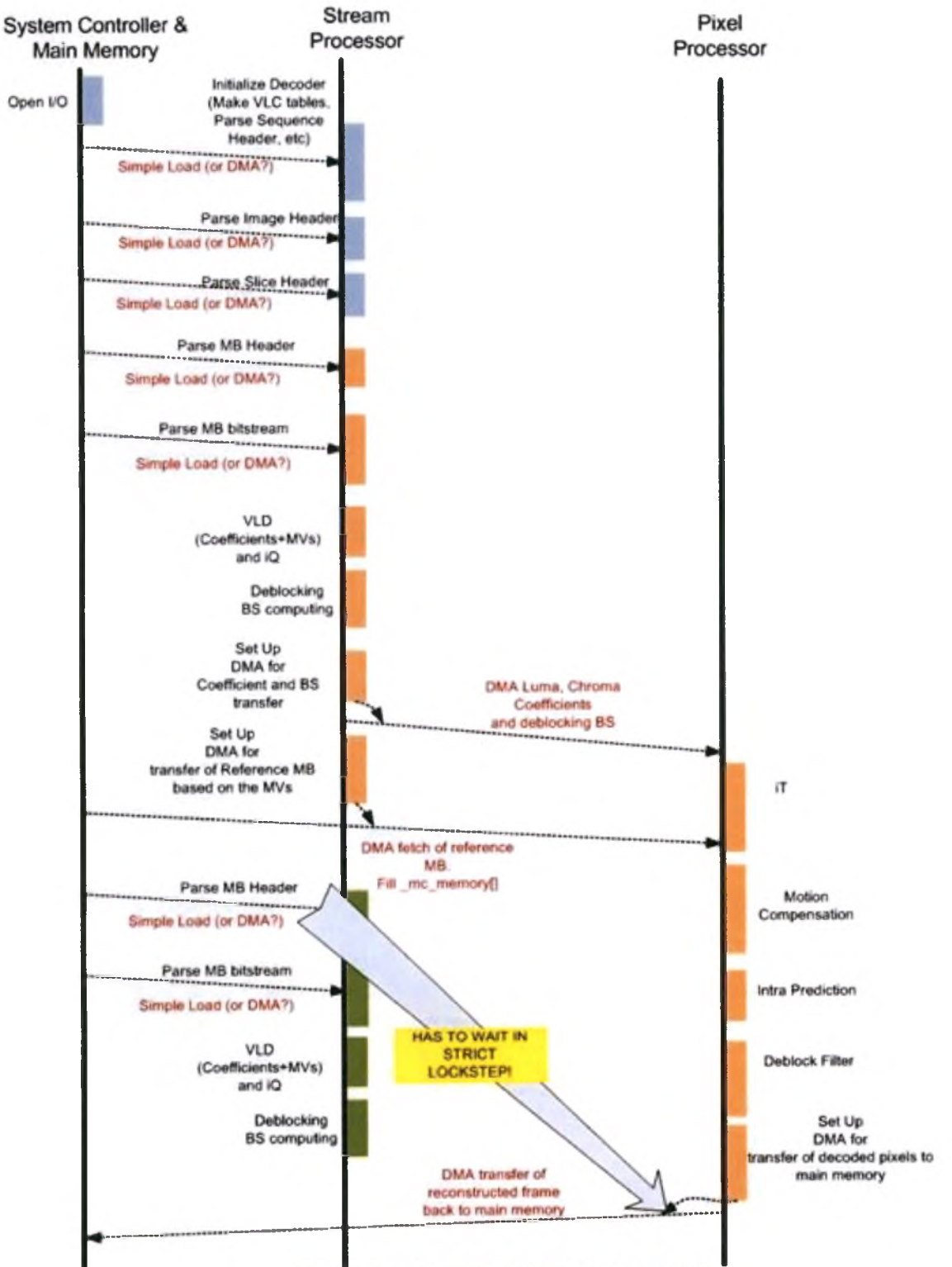
- System Rom: 0x4000_0000 – 0x4FFF_FFFF
- DMA Registers: 0x5000_0000 – 0x5000_011B
- Processor View:
  - Local Instruction Memory: 0x5008_0000
  - Local Data Memory: 0x5006_0000
- System View of DC_388VDO_0
  - Data Memory: 0x5012_0000
  - Instruction Memory: 0x5014_0000
- System View of DC_388VDO_1
  - Data Memory: 0x501a_0000
  - Instruction Memory: 0x501d_0000
- External Memory Address Space starts at 0x6000_0000
  - 0x6000_0000: 256 Bytes communication space

Μιας και οι διευθύνσεις των τοποθεσιών όπου μεταφέρονται κάποια δεδομένα core-to-core αποφασίζονται σε compile-time, οι επεξεργαστές ανταλάσσουν αυτές τις διευθύνσεις μέσω θυρίδων στην εξωτερική μνήμη πριν αρχίσουν την αποκωδικοποίηση. Κατόπιν οι συναλλαγές δεδομένων γίνονται με memcpy στην τοπική μνήμη του άλλου επεξεργαστή, ή στην εξωτερική μνήμη.

Για να υπάρξει συγχρονισμός χρησιμοποιούνται δύο ld/st εντολές συγχρονισμού. Οι εντολές αυτές, που απαλείφουν το loose memory ordering, οργανώθηκαν σε ένα API συγχρονισμού μέσω μίας μοναδικής εξωτερικής διεύθυνσης με πεπερασμένο σύνολο τιμών ανά frame:

- XT_L32AI: load 32 bits πριν από οποιαδήποτε μετέπειτα load/store και acquire/release που συμβαίνουν στο πρόγραμμα.
- XT_S32RI: Store 32 bits μετά από όλα τα προηγούμενα load/store και acquire/release έχουν ολοκληρωθεί.

Εν τέλει, σε αυτό το στάδιο λειτουργούμε σε πλήρες lockstep, θέτοντας κάθε επεξεργαστή να περιμένει έως ότου ολοκληρώσει τις λειτουργίες του αλλά και τις μεταφορές δεδομένων ο άλλος. Το μοντέλο αυτό παρουσιάζεται στην επόμενη σελίδα:



Strict Lockstep Timeline Diagram {6}

DMA Μεταφορές – Buffering scheme releasing strict lockstep

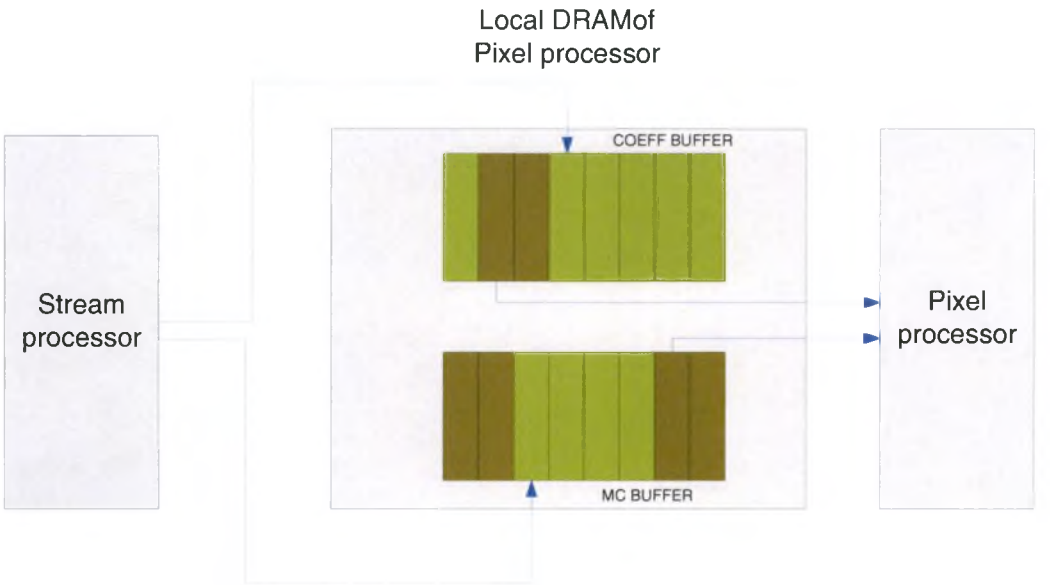
- Μία μεταφορά DMA προγραμματίζεται μέσω ενός descriptor, που αποτελείται από:
  - Destination Address (32 bits) Offset 0x0.
  - Source Address (32 bits) Offset 0x4.
  - Line Count [31:16], Line Size [15:0] Offset 0x8.
  - Destination Stride [31:16], Source Stride [15:0]. Offset 0xc
- Αφού αρχικοποιηθούν οι descriptors κατόπιν προγραμματίζουμε τους καταχωρητές ελέγχου του αντίστοιχου καναλιού DMA για να ξεκινήσει μία DMA μεταφορά είτε core-2-core είτε core-2-external και αντίστροφα.
- Τα κανάλια DMA είναι συνολικά 5 και οι καταχωρητές ελέγχου είναι:
  - FirstNxtPtr: Pointer to the first descriptor. Addr = BASE + 0x0 (RW)
  - LastNxtPtr: Pointer to the last descriptor. Addr = BASE + 0x4 (RW)
  - Control: 32 bit Control register. Addr = BASE + 0x8 (RW)
  - NumLinks: Number of DMA's that need to be performed. Addr = BASE + 0xC (RW)
  - AddNumLinks: Add X to the number of DMA's that need to be performed. Addr = BASE + 0x10 (RW)
  - DMACOUNT: Number of completed DMA's (FIFO order). Addr = Base + 0x14 (RW)
  - Status: Status register for this channel. Addr = Addr + 0x18 (RW)

Χρησιμοποιούνται 3 κανάλια DMA operations:

- CH_COEFF (Channel 0) : Μεταφέρει 8x8 blocks transform coefficients μετά το VLD από τοπική σε τοπική μνήμη, από το Stream Processor στον Pixel Processor. Επίσης παραμέτρους από τον υπολογισμό του Boundary Strength για το Deblocking Filter, και άλλες τιμές ελέγχου συγκεντρωμένες σε μία μοναδική δομή. Το DMA setup γίνεται από τον Stream Processor και αποτελείται πάντα από έναν descriptor
- CH_MC (Channel 1) : Μεταφέρει block δεδομένων στον Pixel Processor από την εξωτερική μνήμη για να μπορέσει να γίνει το Motion Compensation. Το DMA setup το κάνει ο Stream Processor μιας και αυτός έχει πρόσβαση στα Motion Vectors αλλά και για λόγους εξισορρόπησης φόρτου(πιο κοντά στο 50-50).
- CH_FINAL_FRAME (Channel 2): Γράφει πίσω στη μνήμη τα αποκωδικοποιημένα pixels από τον Pixel Processor στην εξωτερική μνήμη. Αυτή η μεταφορά γίνεται setup από τον Pixel Processor.

Να σημειώσουμε πως η DMA χρησιμοποιείται αποδοτικά όταν μεταφέρει μεγάλους όγκους δεδομένων, διαφορετικά δεν είναι αποδοτική η κλήση της.

Σκοπός μας είναι να δημιουργήσουμε, αφού έχουμε ελέγξει τη λειτουργία της DMA, ένα buffering μοντέλο μεταφοράς δεδομένων, που θα κρατά απασχολημένους και τους δύο επεξεργαστές ταυτόχρονα, δεδομένου του balancing του φόρτου εργασίας που έχουμε ήδη κάνει. Αυτό θα είναι κάπως έτσι :



Buffering Diagram {7}

Έχοντας εξαλείψει τους κύκλους εξάρτησης δεδομένων ανάμεσα στους δύο cores και εφόσον εξασφαλίσαμε πως υπάρχει αρκετός χώρος στην τοπική μνήμη του Pixel Processor προχωράμε σε buffering των δεδομένων ώστε «εξαφανιστεί» η σειριοποίηση στην εκτέλεση, δηλαδή να αφήσουμε τους επεξεργαστές να τρέχουν σχεδόν ανεξάρτητα σε μοντέλο Producer-Consumer.

Οι έλεγχοι που πρέπει να γίνουνε είναι για buffer overflow, buffer underflow και πως μία μεταφορά δεδομένων έχει ολοκληρωθεί. (Και φυσικά πως η κατανάλωση γίνεται με την ίδια σειρά που γίνεται η παραγωγή δεδομένων)

Το μοντέλο Producer-Consumer που εφαρμόζεται είναι το εξής:

```
Stream Processor
1.    handshaking;
2.    WAIT for core1;
3.    write local communication addresses
       to an external shared location;
4.    RELEASE core1;
5.    synchronize communication values;
6.    WAIT for core1;
7.    while there is work to do
      {
8.          produce;
9.          if there is no empty place
             in the buffer(size==N)
10.               WAIT;
11.         transfer;
12.         inform pp that a new
             transfer has happened;
      }
13.   Send a termination signal to core1;
14.   exit;
```

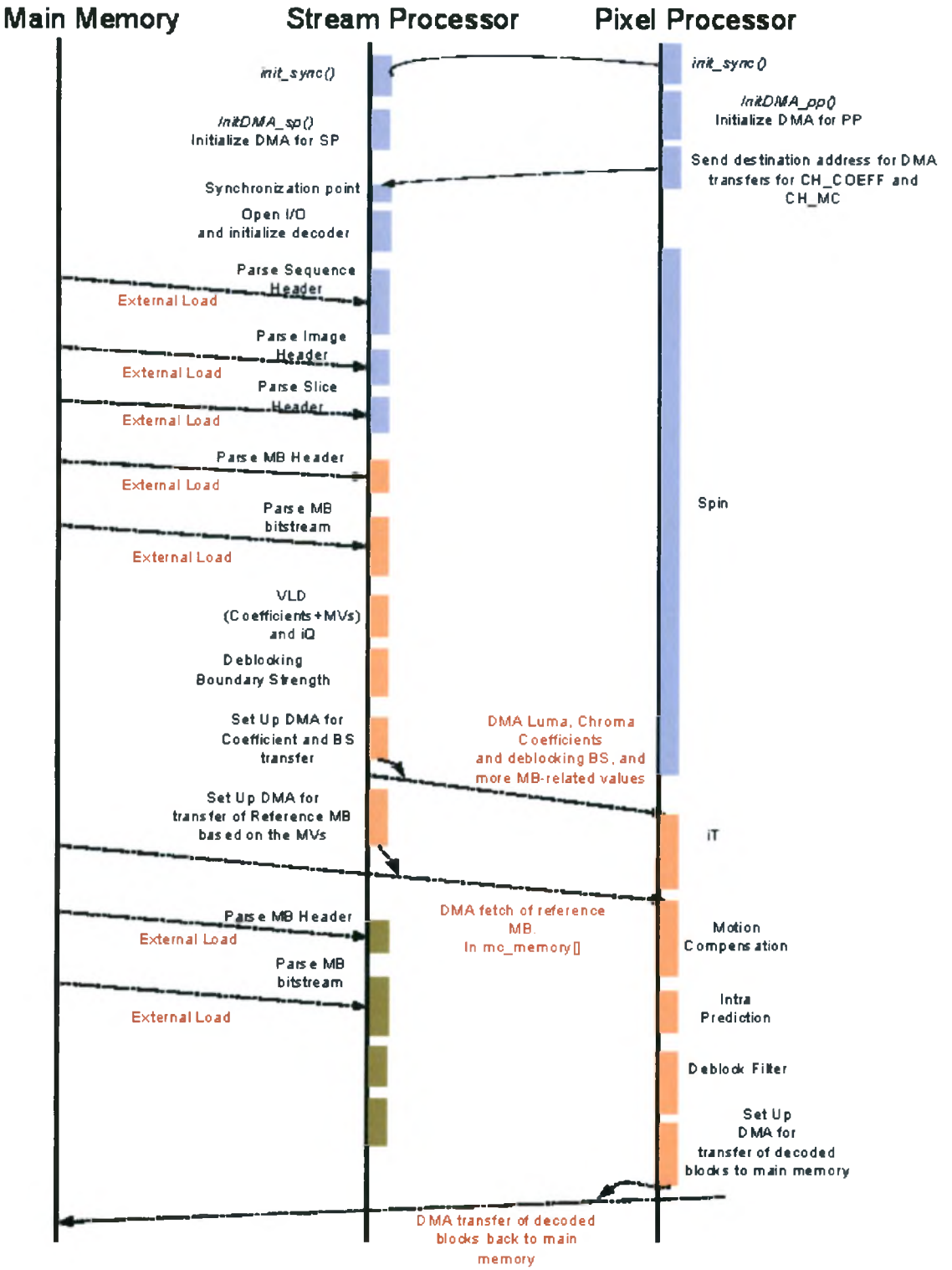
```
Pixel Processor
1.    handshaking;
2.    write local communication addresses
       to an external shared location;
3.    RELEASE core0;
4.    WAIT for core0;
5.    synchronize communication values;
6.    RELEASE core0;
7.    while there is no termination
       signal
      {
8.          if there is work to do in
             the buffer
            {
9.                consume;
10.               transfer to result
                    location;
11.               inform pp that there
                    is a new empty place
                  in the buffer(size==N);
            }
      }
12.   exit;
```

Ένα πρόβλημα έγκειται στο ποιος επεξεργαστής είναι πιο αποδοτικό να επιβεβαιώσει την παραγωγή δεδομένων.Αν τοποθετήσουμε τον PP να το κάνει αυτό, αυξάνουμε το critical path, δηλαδή κάνουμε τον «αργό» processor ακόμη πιο αργό, λόγω του spinning αναμονής για ολοκλήρωση της μεταφοράς. Απο την άλλη πλευρά, ο SP είναι ελεύθερος να προσωρησει στο επόμενο item-production χωρίς να περιμένει να ολοκληρωθούν όλα τα DMA transfers. Επίσης,

το spinning μέχρι να φτάσει το DMACOUNT μία τιμή πιθανότατα θα είναι ελάχιστο μιας και ο Producer λόγω μικρότερου φόρτου θα έχει προχωρήσει στο επόμενο item production.

Αν τοποθετήσουμε τον SP να περιμένει να ολοκληρωθούν τα DMA transfers πριν ενημερώσει τον PP πως έχει παραχθεί ένα νέο item, δεν αυξάνουμε το critical path, δηλαδή παραμένει ο PP όσο αργός είναι. Εγκυμονεί όμως μεγαλύτερος κίνδυνος να παραμένει stale όσο δεν υπάρχει ένα item προς κατανάλωση.

Μετά από μοντελοποίηση των πιθανών σχημάτων τα αποτελέσματα έδειξαν πως ο συνολικός χρόνος εκτέλεσης είναι μικρότερος στην πρώτη από τις δύο περιπτώσεις. Η διαφορά αυτή μεγαλώνει όσο μεγαλώνει ο αριθμός των items που πρέπει να παραχθούν. Όμως αν το μέγεθος του buffer μεγαλώσει τότε ο χρόνος εκτέλεσης είναι μικρότερος στη δεύτερη περίπτωση.



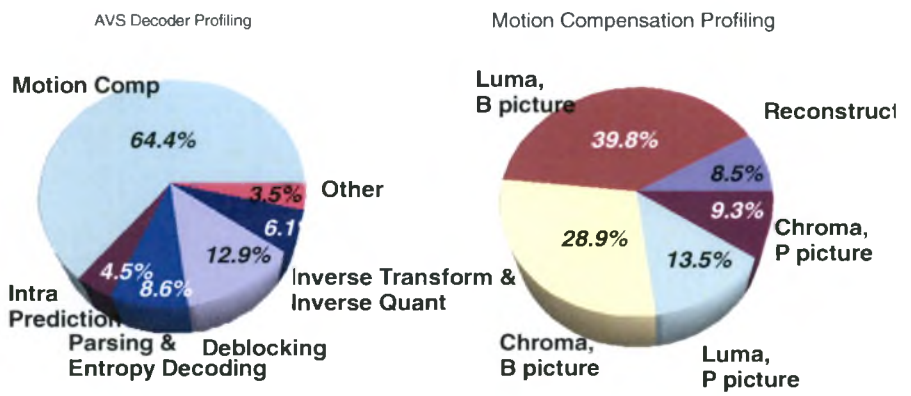
Buffering Scheme Timeline Diagram {8}

Αποτελέσματα

OpenAVS baseline code profiling

Το profiling έγινε χρησιμοποιώντας ως είσοδο ένα bitstream D1 720x576 και ως έξοδο YUV 4:2:0 μορφή. Η ακολουθία βίντεο αποτελείται από 52 frames, έχει μέγεθος 1,02 MB και τα αποκωδικοποιημένα frames ακολουθούν το format : $I\{PBB\}_4I\{PBB\}_4$ κλπ. Μηδενικό latency penalty, ένα εξαιρετικά αισιόδοξο σενάριο που υποθέτει κάθε memory latency να είναι πάντα 1, ή αλλιώς να έχουμε μία τέλεια μνήμη. Ο αποκωδικοποιητής χρειάστηκε 3,2 δις εντολές για 51 frames, που αντιστοιχεί σε περίπου 3,6 δις κύκλους (IPC = 0.89). Με άλλα λόγια ο απαιτούμενος επεξεργαστής θα πρέπει να είναι χρονισμένος τουλάχιστον στα 1,73GHz για να αποκωδικοποιήσει.

Αναλυτικότερα τα σχετικά βάρη των modules:



Module Relative Complexity Diagram {9}

Για να αναδειχθεί η χρήση των τοπικών μνημών, εκτελέστηκε ένα δεύτερο profiling με όλες τις προσβάσεις να γίνονται σε μία εξωτερική μνήμη. Η μνήμη έχει καθυστέρηση 64 κύκλους για την πρώτη ανάγνωση και 32 για την πρώτη εγγραφή, αλλά κανένα penalty μετά από αυτό. Σε αυτή την περίπτωση, ο αποκωδικοποιητής χρειάστηκε 221 δις κύκλους για τις ίδιες 3,2 δις εντολές (IPC=0,014). Δηλαδή, απαιτείται ένας Xtensa Processor χρονισμένος στα 110,5GHz σε αυτό το worst-case scenario.

Pure C software profiling – No internal SRAM memory placement

Αφού προστέθηκαν οι βελτιστοποιήσεις software και τροποποιήθηκε ο κώδικας για να υλοποιηθεί κατόπιν με TIEs είχαμε:
403,32 εκατομμύρια κύκλους (972 cycles/pixel). Αυτό αντιστοιχεί σε :

Motion Compensation: 53.4%
Deblocking: 15.9%
Intra Prediction: 3%
Parsing & VLD: 11.1%
Inverse Transform and Inverse Quant: 7.6%
Data Movement: 2.8%
Other: 6.2%

TIE profiling – No internal SRAM placement

Απαιτήθηκαν 211,7 εκατομμύρια κύκλοι (510 cycles/pixel). Αντιστοιχούν :

Motion Compensation: 40%
Deblocking: 22.8%
Intra Prediction: 5.1%
Parsing & VLD: 13.2%
Inverse Transform and Inverse Quant: 3.6%
Data Movement: 11%
Other: 4.3%

TIE profiling – Instruction και Data SRAM placement

Απαιτήθηκαν 31,7 εκατομμύρια κύκλοι (76 cycles/pixel). Αντιστοιχούν σε:

Motion Compensation: 33.3%
Deblocking: 19.8%
Intra Prediction: 4.8%
Parsing & VLD: 14.5%
Inverse Transform and Inverse Quant: 3.2%
Data Movement: 15.3%
Other: 9.1%

Η τοποθέτηση των συναρτήσεων βιβλιοθηκών που προαναφέρθηκαν στην εσωτερική μνήμη εντολών, επέφερε μείωση του χρόνου εκτέλεσης από 119 cycles/pixel σε 76 cycles/pixel.

Συνολική απόδοση και τελικό speedup στο διπύρηνο μοντέλο:

Optimization		Equivalent F_{clk} (MHz)	Speedup Factor
Baseline OpenAVS code		1730	1
Software Optimizations		1461	1.18
SIMD parallelization (TIEs)	1. Parsing and VLD only	1354	1.28
	2. (1) plus MC. Intra Prediction. inverse Transform	748	2.31
	3. (2) plus Deblocking	649	2.67
Dual Core		359	4.8

Μελλοντικά Θέματα

Η πρόκληση, λοιπόν, που αντιμετωπίζεται κατά τη μεταφορά ενός νέου αποκωδικοποιητή βίντεο σε ένα ASIP σύστημα, είναι ο εντοπισμός και η εκμετάλλευση κάθε είδους παραλληλισμού, σε όλα τα επίπεδα ανάλυσης. Ειδικότερα, γίνεται χρήση παραλληλισμού σε επίπεδο pixel block (SIMD και instruction parallelism) αλλά και σε επίπεδο module (task και pipeline parallelism).

Εν τέλει, παράγονται οφέλη τεχνολογικής αλλά και εργασιακής φύσης, ανοίγοντας νέες διαστάσεις βελτιστοποίησης της παραπάνω ιδέας, ίσως σε θέματα αρχιτεκτονικής και ισχύος, αλλά και ευρύτερα σχεδιαστικά σε επίπεδο engine node, όπως ο συνδυασμός πολλών συστημάτων DC_388VDO. Εκτός όμως από την έξοδο του συγκεκριμένου codec στην παραγωγή, μπορεί να επιτευχθεί και η εκπαίδευση νέων μηχανικών μέσω αυτής της μεθοδολογίας στο σχεδιασμό και συγγραφή embedded εφαρμογών και κυρίως βίντεο.

Παράρτημα

[1] R. B. Lee, “Multimedia extensions for general-purpose processors,” *Proc. IEEE Workshop on Signal Processing Systems*, pp. 9-23, Nov. 1997.

[2] Jose Lau, “MPEG-4, AVS deliver better video compression more flexible format,” *Electronic Times Asia,* June 1st, 2006.

[3] “Diamond Standard Core Processor Architecture,” *Tensilica White Paper,* July 2007.

[4] OpenAvs Source code. [Online] Available:

http://sourceforge.net/projects/openavs

[5] ISO/IEC IS 13818, General Coding of Moving Picture and Associated Audio Information, 1994.

[6] Information technology – Coding of audio-visual objects – Part 2: Visual (ISO/IEC FCD 14496), July 2001.

[7] Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC), May 2003.

[8] Liang Fan, Siwei Ma, Feng Wu, “Overview of AVS Video Standard,” *2004 IEEE International Conference on Multimedia and Expo (ICME '04).*

[9] Tensilica’s Xtensa LX2 Architecture [online] Available:

http://www.tensilica.com/products/xtensa/xtensalx/arch/index.htm

[10] “VDO Instruction Set Architecture (ISA) Extensions Reference Manual,” July 2007.

[11] J.L. Hennessy, D.A. Patterson, “*Computer Architecture. A Quantitative Approach*,” Morgan Kaufmann, 4th Edition, 2006.

[12] Iain E.G. Richardson, “H.264 and MPEG-4 video compression,” 2003

[13] “H264 Deblock Filter on 388VDO Diamond Core,” Application Note, March 2008