Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

The ParalOS Framework for Heterogeneous VPUs: Scheduling, Memory Management & Application Development

Διπλωματικη Εργασια

του

Ευάγγελου Πετρόγγονα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθήνα, Οκτώβριος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

The ParalOS Framework for Heterogeneous VPUs: Scheduling, Memory Management & Application Development

Διπλωματική Εργασία

του

Ευάγγελου Πετρόγγονα

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Νοεμβρίου 2020.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Δημήτριος Σούντρης	Διονύσιος Πνευματικάτος	Παναγιώτης Τσανάκας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π

Αθήνα, Οκτώβριος 2020

(Υπογραφή)

....

Ευάγγελος Πετρογγωνας

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

© 2020 – All rights reserved

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Copyright © –All rights reserved Ευάγγελος Πετρόγγονας, 2020.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή Δημήτριο Σούντρη τόσο για την καθοδήγηση του κατά τη διάρκεια της διπλωματικής άλλα κυρίως γιατί από το 3ο κιόλας έτος των σπουδών μου, αγκάλιασε με μεγάλη προθυμία την επιθυμία μου να ασχοληθώ με τα αντικείμενα του εργαστηρίου, ενώ ταυτόχρονα με έφερε από νωρίς σε επαφή με τον ¨πραγματικό κόσμο¨. Βεβαίως δεν γίνεται να παραλείψω τους δύο πιο στενούς συνεργάτες, και πλέον φίλους μου, το μεταδιδακτορικό ερευνητή Γέωργιο Λεντάρη και τον ΥΔ Βασίλειο Λέων, οι οποίοι με με μετέτρεψαν από έναν ανήσυχο φοιτητή σε έναν ικανό και συνειδητοποιημένο μηχανικό. Ένα μεγάλο ευχαριστώ οφείλω στους φίλους μου οι οποίοι ήταν συμπαραστάτες στις εύκολες και δύσκολες στιγμές αυτά τα 5, και κάτι... , χρόνια, Τέλος, το μεγαλύτερο ευχαριστώ το οφείλω στο σημαντικότερο αρωγό της ζωής μου, στη μητέρα μου, της οποίας η ανιδιοτέλεια και η μητρική αγάπη είναι βγαλμένες απο ποιητική συλλογή..., σε ευχαριστώ πολύ για αυτό και το να αποκαλούμε γιος σου είναι η μεγαλύτερη τιμή της ζωής μου.

Την συγκεκριμένη εργασία την αφιερώνω στον πατέρα μου, και συνάδελφο μου πλέον, ο οποίος εδώ και πολλά χρόνια πλέον με καθοδηγεί απο ψηλά ...

Περίληψη

Τα ετερογενή υπολογιστικά συστήματα έρχονται αντιμέτωπα με ένα συνεχώς μεταβαλλόμενο πεδίο εφαρμογών, οι οποίες διέπονται από ολοένα μεγαλύτερη υπολογιστική και προγραμματιστική πολυπλοκότητα. Καθώς ο νόμος του Moore, καταφθάνει σε ένα φυσικό τέλμα, τα ετερογενή υπολογιστικά συστήματα σε ψηφίδα SoC και συγκεκριμένα οι υπολογιστές όρασης Vision Processing Units (VPUs), αναδεικνύονται ως μια ελκυστική πρόταση σε διάφορους τομείς εφαρμογών. Αυτά τα συστήματα ωστόσο, εξακολουθούν να απαιτούν περίπλοκη και μονολιθική ανάπτυξη λογισμικού ώστε να επιτευχθούν αποδοτικές υλοποιήσεις.

Στο πλαίσιο αυτής της διπλωματικής, με γνώμονα τα παραπάνω χαρακτηριστικά, αναπτύχθηκε ένα προγραμματιστικό περιβάλλον ανάπτυξης εφαρμογών-Framework το οποίο αποσκοπεί στην επιτάχυνση της ανάπτυξης υπολογιστικά και προγραμματιστικά απαιτητικού λογισμικού σε VPUs, ενώ επιτρέπει την πλήρη εκμετάλλευση του παρεχόμενου υλικού μέσα από χαμηλού επιπέδου βελτιστοποιήσεις υπολογιστικών πυρήνων. Το προτεινόμενο framework στοχεύει ετερογενείς αρχιτεκτονικές και αποτελείται απο έναν δυναμικό δρομολογητή εργασιών (dynamic task scheduler), ένα καινοτόμο σύστημα διαχείρισης της Scratchpad μνήμης, την προτυποποίηση του συστήματος Εισόδου/Εξόδου, τεχνικές άμεσης και αποκεντρωμένης επικοινωνίας μεταξύ των εφαρμογών και τέλος έναν οπτικό Profiler

Για την αξιολόγηση της υλοποίησης επιλέχθηκε η οικογένειά των Myriad VPU επεξεργαστών της Intel/Movidius και αξιολογήθηκαν τόσο συνθετικά προγράμματα όσο και πραγματικές εφαρμογές όπως η υλοποίηση Συνελικτικών Νευρωνικών Δικτύων (ΣΝΔ) και αλγόριθμοι Visual Based Navigation (VBN) από τον τομέα της διαστημικής. Τα αποτελέσματα είναι ιδιαιτέρως ενθαρρυντικά, καθώς όσον αφορά την ταχύτητα εκτέλεσης παρατηρείται ένα περιορισμένο overhead της τάξης του ~8% έναντι βελτιστοποιημέων υλοποίησεων ΣΝΔ, ενώ βελτίωση έως και 4.2× εμφανίσθηκε σε αλγορίθους που εξαρτώνται απο το περιεχόμενο των δεδομένων. Όσον αφορά τη χρήση της Scratchpad μνήμης το προτεινόμενο σύστημα οδηγεί σε έως και 33% μειωμένες απαιτήσεις χώρου συγκριτικά με παραδοσιακές τεχνικές διαχείρισης μνήμης, ενώ τέλος το IPC υποσύστημα παρουσιάζει εως και 6× καλύτερη επίδοση σε σχέση με αυτό που παρέχεται από τον κατασκευαστή.

Λέξεις Κλειδιά

Ετερογενείς Αρχιτεκτονικές, Myriad, Framework, Profiling, IPC, Διαχείριση Μνήμης, Δρομολογητής Εργασιών, Ενσωματωμένα Συστήματα.

Abstract

Embedded systems are presented today with the challenge of a very rapidly evolving application diversity followed by increased programming and computational complexity. As the Moore's Law is reaching a, physics induced, cul-de-sac, customised heterogeneous System-on-Chip (SoC) and more specifically Vision Processing Units (VPUs) emerge as an attractive HW solution in various application domains. However, these platforms still require sophisticated monolithic SW development to provide efficient implementations

In this context, a framework for accelerating the SW development of computationally intensive applications on VPUs, while still enabling the exploitation of their full HW potential via low-level kernel optimisations is proposed in this thesis. This framework is tailored for heterogeneous architectures and integrates a dynamic task scheduler with a high level transparent API, a novel scratchpad memory management scheme, I/O standardisation, inter-process communication (IPC) techniques, and an insightful visual profiler.

The Intel Movidius Myriad family of VPUs is used as an evaluation platform employing both synthetic benchmarks and real-world applications, which vary from Convolutional Neural Networks (CNNs) to complex computer vision algorithms for Visual Based Navigation (VBN) targeting the space industry. The results are very promising, showcasing in terms of execution time, a limited ~8% performance overhead vs manually optimised CNN programs while achieving up to 4.2× performance gain in content-dependent applications. Regarding the Scratchpad Memory usage a reduction of up to 33% is recorded compared to well-established memory allocators and finally the IPC cost is decreased up to 6× vs the default vendor implementation.

Keywords

Heterogenous Architectures, Myriad, VPU, Framework, Scaratchpad Memory Management, Task Scheduling, Profiling, IPC, Embedded Systems

Contents

List of Figures

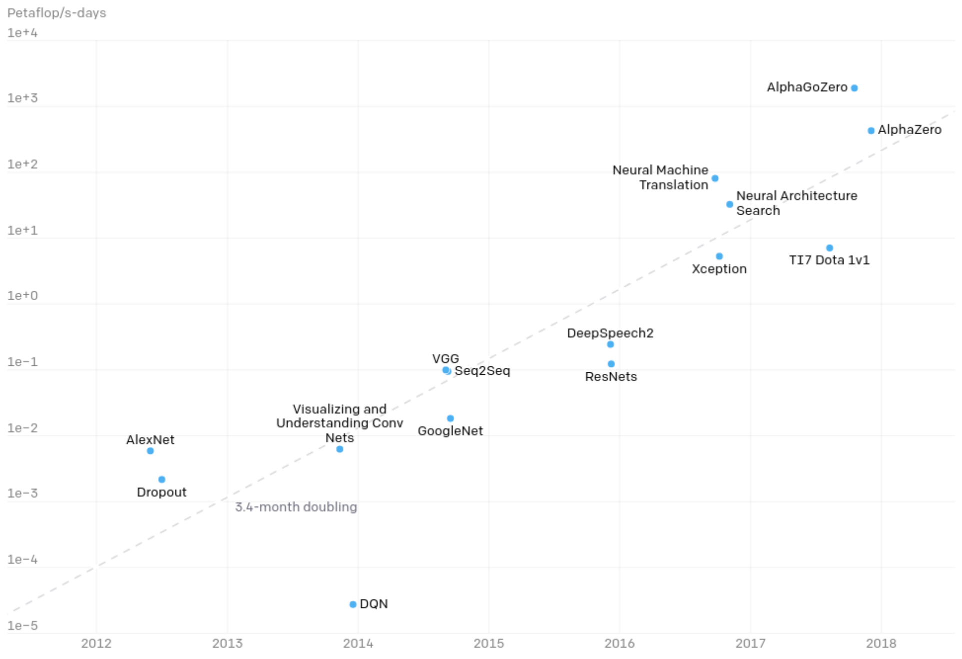
List of Tables

Listings

Εκτεταμένη Περίληψη

Εισαγωγή

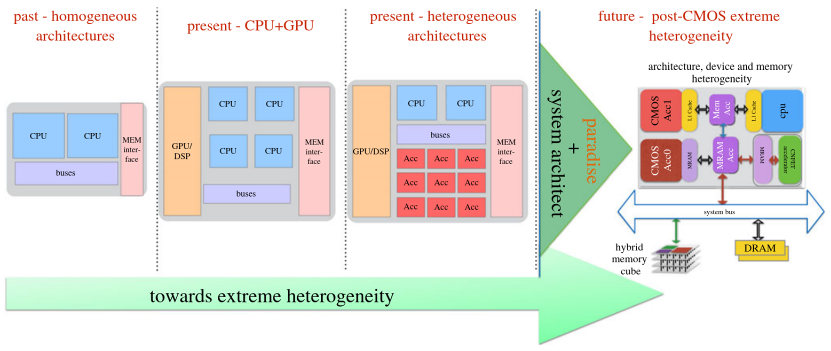
Η εισαγωγή εφαρμογών τεχνητής νοημοσύνης και όρασης υπολογιστών, έχει επιφέρει μια επανάσταση στον κόσμο της υπολογιστικής. Όπως υποστηρίζει ο Amodei [1] και παρουσιάζεται στο σχήμα 1, τόσο οι υπολογιστικές απαιτήσεις όσο και η φύση των αλγορίθμων μεταβάλλονται πιο γρήγορα από ποτέ.



Σχήμα 1: Συνολική Υπολογιστική απαίτηση σε petaflops-days, που χρειάστηκε για την εκπαίδευση του εκάστοτε νευρωνικού δικτύου [1]

Παραδοσιακές πλατφόρμες όπως η CPU, έχουν φθάσει στα όρια των δυνατοτήτων τους και έρχονται αντιμέτωπες με διάφορα προβλήματα όπως το Power & Memory Wall. Ταυτόχρονα η ¨δωρεάν' αύξηση επιδόσεων απόρροια, της κλιμάκωσης του Dennard έχει φθάσει σε φυσικά όρια. Οι Hennessy & Patterson [9] άλλα και ο Thompson [10] υποστηρίζουν ότι η λύση βρίσκεται στην ευρεία κλιμάκωση και υιοθέτηση της ετερογένειας (σχήμα 2), αφού οι άλλες εναλλακτικές απέχουν σημαντικά από την δυνατότητα χρήσης τους σε ευρεία κλίμακα.

Η ετερογένεια ωστόσο έχει το ενδογενές χαρακτηριστικό της δύσκολης προγραμματιση-


past - homogeneous architectures
present - CPU+GPU
present - heterogeneous architectures
future - post-CMOS extreme heterogeneity
CPU
CPU
MEM inter-face
buses
GPU/DSP
GPU/DSP
Acc
paradise + system architect
architecture, device and memory heterogeneity
CMOS Acc1
CMOS Acc0
Mem Acc
MRAM Acc
cpu
system bus
hybrid memory cube
DRAM
towards extreme heterogeneity

Σχήμα 2: Η εγκαθίδρυση της ετερογένειας ως η νέα υπολογιστική νόρμα [2]

μότητας. Δε θα είχε νόημα η υιοθέτηση περίπλοκων αρχιτεκτονικών αν δεν είναι εφικτή η εύκολη ανάπτυξη εφαρμογών σε αυτές. Η Intel διατύπωσε την άποψη ότι η προγραμματισημότητα είναι αντιστρόφως ανάλογη της ετερογένειας. Εάν αυτό ισχύει τότε ορθώνεται ένας νέος τοίχος αυτός της προγραμματισημότητας (Programmability Wall). Η επίλυση αυτού του προβλήματος όπως αναφέρει ο Leiserson [11] και ταυτίζεται και ο συγγραφέας, βρίσκεται στη δημιουργία Framework που στοχεύουν στην διευκόλυνση του προγραμματισμού.

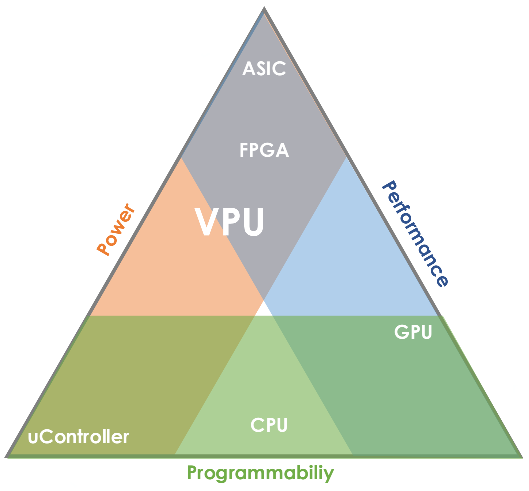
Σε αυτό το πλαίσιο γεννήθηκε και το *ParalOS* , ένα Framework που στοχεύει στην διευκόλυνση της χρήσης εξαιρετικά ετερογενών αρχιτεκτονικών όπως οι VPUs. Σε αυτή την προσπάθεια, ωστόσο έγινε προσπάθεια να διαφυλαχθεί η ταυτότητα αυτών των αρχιτεκτονικών και να μην θυσιαστούν στο βωμό της προγραμματισημότητας, οι ξεχωριστές αρετές της κάθε πλατφόρμας.

Υπόβαθρο

Vision Processing Units (VPUs)

Οι επεξεργαστές Vision Processing Units (VPUs) αποτελούν μια νεοαφιχθείσα κατηγορία ενσωματωμένων συσκευών που στοχεύουν να επαναπροσδιορίσουν την έννοια του edge computing. Διέπονται από εξαιρετικά υψηλή ετερογένεια, χαμηλή κατανάλωση και ο ρόλος τους είναι όχι απλά να επεξργάζονται την πληροφορία, αλλά να την κατανοούν, όπως υποστηρίζει η Intel Movidius

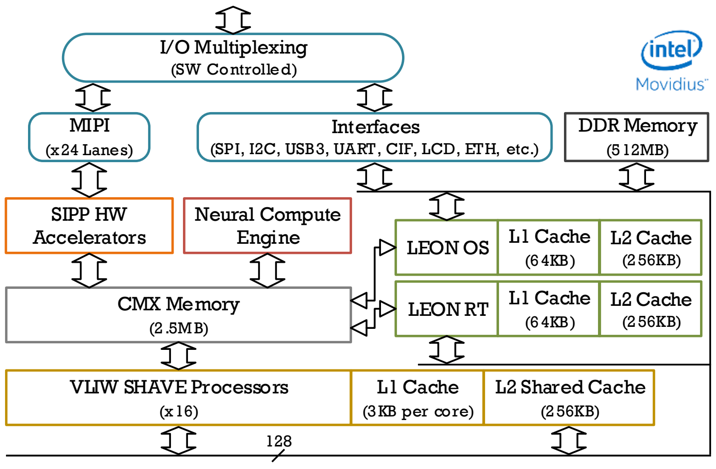
Στο σχήμα 3 παρουσιάζεται μια απλοποιημένη κατηγοριοποίηση των διάφορων συσκευών που απαντώνται στα ενσωματωμένα συστήματα. Όπως φαίνεται από άποψη καθαρών επιδόσεων υπολείπονται συγκριτικά με GPUs & FPGA, ενώ προσφέρουν αρκετά καλύτερες επιδόσεις από τις συμβατικές CPU. Στον τομέα της ενέργειας απαιτούν λιγότερη κατανάλωση ενέργειας από τις υπόλοιπες εναλλακτικές, παρουσιάζουν ωστόσο σημαντική προγραμματιστική δυσκολία. Τέλος όταν υπεισέρχεται η μετρική Performance per Watt, τότε αποτελούν ενδεχομένως την καλύτερη επιλογή αφού σε συγκεκριμένες κατηγορίες προβλημάτων, υπερβαίνουν και αυτή των FPGA.



Σχήμα 3: Τα VPUs στο φάσμα των Ενσωματωμένων Υπολογιστικών συστημάτων.

Myriad Family

Η οικογένεια των Myriad [4, 12], επεξεργαστών της Intel αποτελεί ενδεχομένως τον πιο αναγνωρίσιμο εκπρόσωπο της κατηγορίας των VPU. Αποτελείται από τις Myriad 2 & MyriadX των οποίων η αρχιτεκτονική παρουσιάζεται συνοπτικά στη συνέχεια.



Σχήμα 4: Η αρχιτεκτονική της MyriadX [4]

Περιλαμβάνουν δύο γενικού σκοπού LEON4 επεξεργαστές που βασίζονται στην 32-bit RISC SPARCv8 αρχιτεκτονική, αυτοί είναι ο LEON OS (LOS) και ο LEON RT (LRT). Ο πρώτος υποστηρίζει ένα λειτουργικό σύστημα πραγματικού χρόνου (RTEMS) και ο δεύτερος είναι υπεύθυνος για τη διαχείριση των IO και λοιπών περιφερειακών. Οι υπολογιστικές ικανότητες της συσκευής προσφέρονται κατά κύριο λόγο από τους Streaming Hybrid Architecture Vector Engines (SHAVEs) πυρήνες, οι οποιοι ελέγχονται άμεσα από τους προαναφερόμενους πυρήνες γενικού σκοπού. Οι SHAVES οφείλουν την ισχύ τους στην 128 bit VLIW SIMD αρχιτεκτονική τους. Οι Myriad διαθέτουν επιπλέον και μια σειρά από φίλτρα για επεξεργασία εικόνας σε υλικό που ονομάζονται Streaming Image Processing Pipeline

(SIPP).

Όσον αφορά την ιεραρχία μνήμης, το SoC συμπεριλαμβάνει DDR DRAM. Η κύρια μνήμη εργασίας ωστόσο είναι η Connection Matrix Memory (CMX) η οποία λειτουργεί ως NUMA ScratchPad. Ο κάθε SHAVE έχει προτιμητέα πρόσβαση σε ένα συγκεκριμένο κομμάτι της. Οι μεταφορές δεδομένων μεταξύ CMX-DDR, πραγματοποιούνται μέσω μιας Direct Memory Access μηχανής. Κάθε LEON πυρήνας περιλαμβάνει αποκλειστικές L1 (D+I) και L2 caches, ενώ οι SHAVEs έχουν αποκλειστική L1 (D+I) αλλά μοιράζονται μια κοινή L2$.

Η MyriadX, αποτελεί τον πιο πρόσφατο επεξεργαστή της οικογένειας και είναι η πρώτη συσκευή που περιέχει ένα αφιερωμένο επιταχυντή (NCE) για την εκτέλεση Συνελικτικών Νευρωνικών Δικτύων. Οι επιδόσεις του φθάνουν το $1TOPs$ με ρολόι 700Mhz, έχει 16 πυρήνες SHAVE, 2.5MB (CMX) και 512MB DRAM. Τέλος ο πρόγονος της, η Myriad 2, διαθέτει 12 πυρήνες SHAVE, 2MB (CMX) και ρολόι στα 600Mhz.

Frameworks

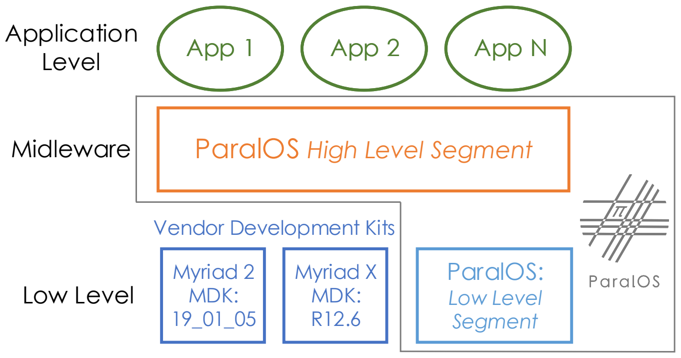
Στη αγορά υπάρχουν αρκετά διαθέσιμα Framework για την την παραλληλοποίηση εφαρμογών και την επιτάχυνσή τους σε μια ευρεία γκάμα συσκευών όπως τα OpenMp & MPI, [13, 14], ωστόσο δεν υποστηρίζουν VPUs. Το OpenVINO [15] που παρέχεται από την Intel έχει βελτιστοποιηθεί για την εκτέλεση νευρωνικών δικτύων στη MyriadX. Τέλος μια νέα προσπάθεια από την Intel, είναι η ανάπτυξη του OneAPI [6], ένα Framework που στηρίζεται στη γλώσσα προγραμματισμού DPC++ και υπόσχεται, την εκτέλεση του ίδιου κώδικα σε μια ευρεία γκάμα διαφορετικών ετερογενών συσκευών, αλλά βρίσκεται ακόμα σε αρχικό στάδιο και δεν υποστηρίζει VPUs.

ParalOS Framework

Ο στόχος του *ParalOS* είναι η δημιουργία ενός Framework που να επιτρέπει τον αποδοτικό προγραμματισμό και Design Space Exploration σε VPU. Ο στόχος είναι η αφαίρεση των ειδικών χαρακτηριστικών του συστήματος σε ένα ικανοποιητικό αλλά συντηρητικό βαθμό, ώστε να επιτρέπει τη γρήγορη ανάπτυξη προγραμμάτων σε αυτό, χωρίς ωστόσο να ναρκοθετεί τις βελτιστοποιήσεις σε χαμηλό επίπεδο.

Το Framework αποτελείται από δύο μέρη το *ParalOS: High Level Segment* και το *ParalOS: Low Level Segment* . Το πρώτο παρέχει στον προγραμματιστή ένα API για τη δημιουργία προγραμμάτων, δρομολογητή εργασιών, διαχείριση ΕΕ, και έναν visual profiler. Το δεύτερο αποτελείται από α) ένα καινοτόμο σύστημα διαχείρισης της Scratchpad Μνήμης ειδικά σχεδιασμένο για βέλτιστη χρήση των caches, το οποίο επιτρέπει τη δυναμική διαχείριση μνήμης με μηδενική χωρική επιβάρυνση και β) ενα αποκεντρωμένο σύστημα για επικοινωνία εντός του SoC.

Όπως φαίνεται στο σχήμα 5, το *ParalOS* δεν προσδοκά να αντικαταστήσει τις ήδη υπάρχουσες βιβλιοθήκες που παρέχονται από τον κατασκευαστή. Αντιθέτως, τις επεκτείνει όπου εντοπίστηκαν ελλείψεις και εισάγει ένα ενδιάμεσο επίπεδο αφαίρεσης πάνω στο οποίο προγραμματίζονται και εκτελούνται οι εφαρμογές.

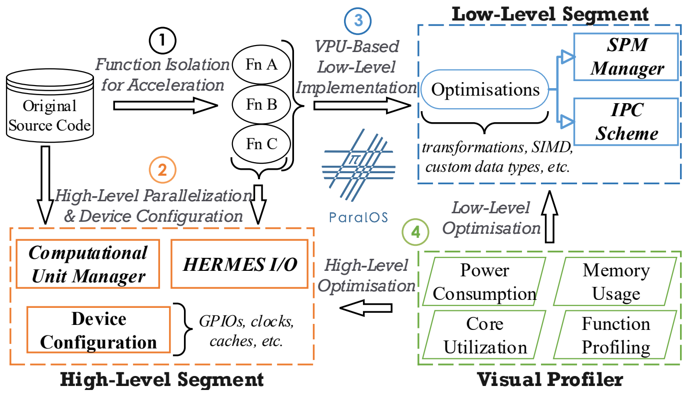


Σχήμα 5: Το software stack του *ParalOS*

Όσον αφορά το προγραμματιστικό παράδειγμα που παρουσιάζεται στο σχήμα 6, στηρίζεται στο διαχωρισμό της ανάπτυξης εφαρμογών σε δύο μέρη. Στο High Level ο προγραμματιστής χρησιμοποιεί το *ParalOS: High Level Segment* για να περιγράφει και να παραλληλοποιήσει τους αλγορίθμους, το οποίο εκτελείται στους γενικού σκοπού επεξεργαστές, που θα αποκαλούνται πυρήνες-διαχειριστές. Στη συνέχεια με τη βοήθεια του *ParalOS: Low Level Segment* για τη διαχείριση μνήμης και την ενδοεπικοινωνία, επιταχύνει την κάθε συνάρτηση/υπολογιστικό πυρήνα με βελτιστοποιήσεις χαμηλού επιπέδου. Αυτό το τμήμα εκτελείται στους SHAVEs που πλεόν θα αναφέρονται ως εργάτες.

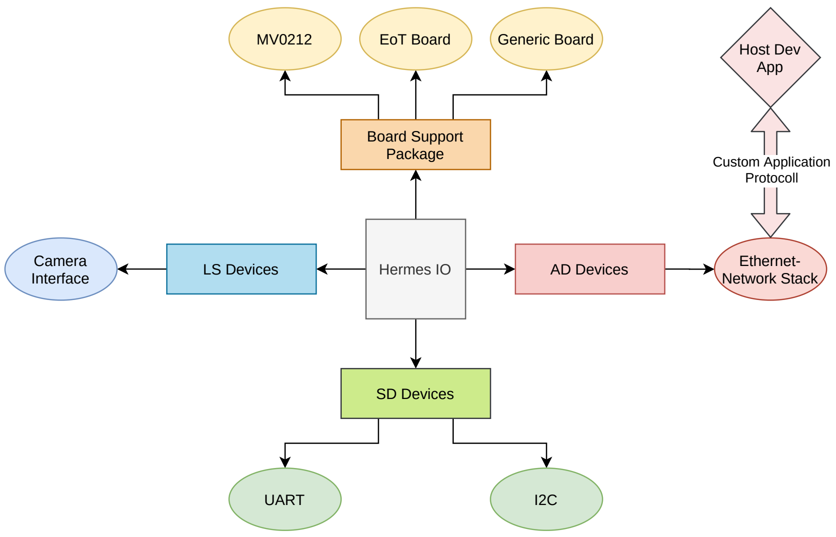
Hermes IO

Το συγκεκριμένο υποσύστημα, αποτελεί κομμάτι του *ParalOS: High Level Segment* και εκτελείται στους πυρήνες-διαχειριστές. Είναι υπεύθυνο για τη διαχείρισή των διεπαφών για την



Σχήμα 6: High-level Αρχιτεκτονική του *ParalOS* Framework

επικοινωνία με το περιβάλλον του, καθώς επίσης και για την αρχικοποίηση της πλακέτας του SoC. Οι υποχρεώσεις του περιλαμβάνουν την αποστολή και λήψη δεδομένων, την οργάνωση και την αποθήκευση των δεδομένων και είναι ειδικά διαμορφωμένο για την αποτελεσματικότερη διαχείρισή δεδομένων εικόνας. Πιο συγκεκριμένα, στην ουσία παρέχει ένα προτυποποιημένο API, παρόμοιο με αυτό του CMSIS [16], το οποίο περικλείει τις συναρτήσεις που είναι διαθέσιμες από τις βιβλιοθήκες του κατασκευαστή. Οι διεπαφές χωρίζονται σε τρείς κατηγορίες: α) μικρού μεγέθους διπλής κατεύθυνσης, όπως το UART & I2C, β) μεγάλου μεγέθους μονής κατεύθυνσης, όπως το LCD & CIF και γ) μεγάλου μεγέθους διπλής κατεύθυνσης που αναφέρεται στη δικτυακή διεπαφή. Αυτές παρουσιάζονται στο σχήμα 7. Ειδικά για το δίκτυο έχει αναπτυχθεί ένα ειδικό πρωτόκολλο με τον αντίστοιχο driver για τον υπολογιστή, το οποίο περικλείει τα δεδομένα σε μια επικεφαλίδα που περιέχει το μέγεθος και το είδος του τύπου δεδομένων, ενώ προαιρετικά υποστηρίζει και την προσθήκη ενός CRC πεδίου για το έλεγχο της εγκυρότητας του μηνύματος.



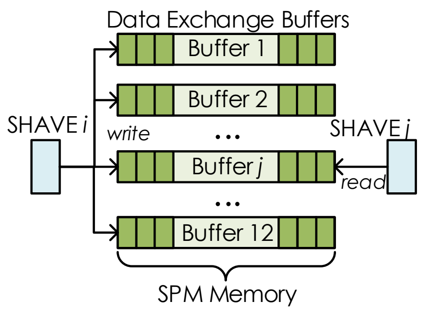
Σχήμα 7: Η αρχιτεκτονική του Hermes IO

SMPI: Επικοινωνία και συγχρονισμός μεταξύ των πυρήνων

Η ανταλλαγή δεδομένων μεταξύ των επεξεργαστικών μονάδων του VPU υποστηρίζεται με διάφορες μεθόδους, όπως το DMA Engine, ή μικρά register pipes. Ωστόσο αυτές οι μέθοδοι είναι σχεδιασμένες για παράλληλα προγράμματα, στα οποία ο παραλληλισμός τους δεν απαιτεί την ανταλλαγή μηνυμάτων. Ως αποτέλεσμα είναι ιδανικά για την μεταφορά μικρού όγκου δεδομένων (< 128B), ή πολύ μεγάλου ($> 2KB$). Για τους λόγους αυτούς υλοποιήθηκε ένα υποσύστημα επικοινωνίας και συγχρονισμού μεταξύ των πυρήνων του συστήματος, το οποίο προσφέρει ένα API παρόμοιο αυτού του MPI [14].

Όσον αφορά το συγχρονισμό, η πλατφόρμα παρέχει έναν μικρό αριθμό απο hardware mutexes. Αυτά ωστόσο χρησιμοποιούνται απο διάφορα υποσυστήματα και προγράμματα συνεπώς, δεν είναι εφικτή η εγκαθίδρυση ενός πιο ευέλικτου και fine-grained μηχανισμού κλειδώματος. Το πρόβλημα αυτό επιλύθηκε με την υλοποίηση ενός συστήματος εικονικών mutex, τα VMutex. Ο προγραμματιστής αντί να χρησιμοποεί απευθείας τα Hardware Mutex χρησιμοποιεί τα εικονικά. Όλα τα εικονικά αποτελούν μια ομάδα. Αυτή η ομάδα με τη χρήση ενός μηχανισμού του mapper, αντιστοιχίζεται στα πραγματικά. Ως αποτέλεσμα τεχνικές fine-grained κλειδώματος μπορούν να υλοποιηθούν, ενώ πραγματικά bottleneck εμφανίζονται μόνο όταν απαιτείται η ταυτόχρονη χρήση περισσότερων Hardware Mutex απο αυτά που διαθέτει το σύστημα.

Για το συγχρονισμό εκτός από τα VMutex, προσφέρεται και μια μέθοδος η `SMPIBarrier`. Αυτή υλοποιεί ένα φράγμα μεταξύ των διαφόρων εργατών διακόπτοντας την εκτέλεση τους, μέχρις ότου η εκτέλεση του προγράμματος να φτάσει σε αυτό το σημείο σε όλους. Όταν γίνει αυτό η εκτέλεση συνεχίζεται κανονικά. Η υλοποίηση του βασίζεται σε έναν τροποποιημένο αλγόριθμο που στηρίζεται στο [17].



Σχήμα 8: Η αρχιτεκτονική του SMPI

Η ανταλλαγή μηνυμάτων γίνεται μέσω των συναρτήσεων `SMPISend` και `SMPIReceive`. Για κάθε εργάτη όριζεται στην SPM ένας κυκλικός buffer αποδοχής. Όταν ένας εργάτης A, θέλει να στείλει ένα μήνυμα στον B, τότε ο A ελέγχει αν υπάρχει αρκετός χώρος στον buffer του. Εάν αυτός επαρκεί τότε γράφει τα δεδομένα στον συγκεκριμένο buffer ενημερώνοντας στην πορεία διάφορους μετρητές και σημαίες. Προαιρετικά μπορεί να χρησιμοποιηθεί ένα ειδικό πρωτόκολλο επικοινωνίας παρόμοιο με αυτό που περιεγράφη νωρίτερα στο υποσύστημα Hermes. Τα δεδομένα τότε περικλείονται σε ένα πακέτο το οποίο διαθέτει μια κεφαλίδα, όπου αναφέρεται το id του αποστολέα και το μέγεθος του μηνύματος.

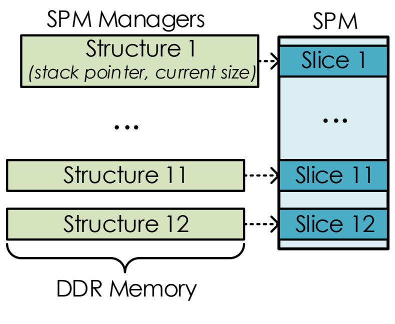
Όταν ο εργάτης B θέλει να διαβάσει τα δεδομένα από το buffer του, ελέγχει εάν έχει λάβει κάποιο μήνυμα συγκρίνοντας δύο μετρητές που αντιστοιχούν στα εγγεγραμμένα και αναγνωσμένα bytes του buffer. Στη συνέχεια διαβάζει τα ληφθέντα μηνύματα σειριακά. Τέλος αν έχει χρησιμοποιηθεί το πρωτόκολλο, ο B μπορεί να φιλτράρει τα εισερχόμενα μηνύματα, αναλόγως του αποστολέα.

Διαχειριστής SPM μνήμης

Ο διαχειριστής της SPM μνήμης λειτουργεί ως ένας χειροκίνητος allocator για την SPM. Παρέχεται σε δύο εκδόσεις, μία που υποστηρίζει μόνο στατικές δεσμεύσεις και μια που εκτελεί και δυναμικές. Η αποδοτική διαχείριση της συγκεκριμένης είναι καθοριστικής σημασίας για την ενεργειακή άλλα και χρονική επίδοση ολόκληρου του συστήματος. Για τον σκοπό αυτό αναπτύχθηκε το συγκεκριμένο υποσύστημα,

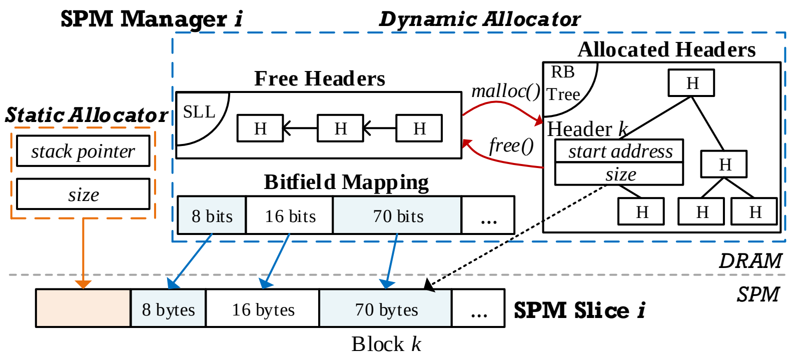
Αντί ενός κεντρικού διαχειριστή που ελέγχει το σύνολο της SPM, το συγκεκριμένο υποσύστημα αποτελείται από E διακριτούς διαχειριστές όπου E ο αριθμός των τμημάτων της SPM, με το κάθε τμήμα να ανήκει αποκλειστικά σε έναν εργάτη. Ως αποτέλεσμα οι χρονικές επιδόσεις βελτιώωνται, καθώς κατά την εκτέλεση, τα διάφορα αιτήματα επεξεργάζονται παράλληλα, χωρίς να απαιτείται ολικός συγχρονισμός. Ο κάθε διαχειριστής δεν είναι παρά μια συλλογή από δομές δεδομένων οι οποίες επιτηρούν και παρακολουθούν τη χρήση της SPM μνήμης. Αυτές οι δομές είναι τοποθετημένες στην DDR μνήμη και είναι προσβάσιμες τόσο από τους πυρήνες-διαχειριστές όσο και από τους εργάτες. Αυτός είναι ο ακρογωνιαίος λίθος της ερευνητικής καινοτομίας που εισάγεται στην παρούσα διπλωματική, δηλαδή ο φυσικός διαχωρισμός μεταξύ των headers (κεφαλίδες) και των ουσιαστικών δεδομένων: Τα πρώτα είναι τοποθετημένα στην DDR ενώ τα δεύτερα στην SPM. Ως αποτέλεσμα κάθε χωρική επιβάρυνση εξαφανίζεται και με αυτόν τον τρόπο επιτυγχάνεται 100% χρησιμοποίησή της κρίσιμης σημασίας SPM.

Όσον αφορά τη στατική δέσμευση μνήμης, η αρχιτεκτονική παρουσιάζεται στο σχήμα 9. Εξομοιώνει την λειτουργία της stack μνήμης και ως αποτέλεσμα επιτρέπεται μόνο η ανάθεση μνήμης και όχι η αποδέσμευση η επαναχρησιμοποίηση αυτής. Ειδικότερα χρησιμοποιείται μια τεχνική μετρητών για την παρόμοια με αυτήν του FreeRTOS level-1 heap allocator [18]. Σημειώνεται ότι η στατική δέσμευση προτείνεται ως διαδικασία και στην περίπτωση που απαιτείται δέσμευση στατικών δομών όπως buffers κλπ, ακόμα και εάν έχει ενεργοποιηθεί η λειτουργία της δυναμικής διαχείρισης που περιγράφεται στη συνέχεια.



Σχήμα 9: Αρχιτεκτονική του στατικού Διαχειριστή της SPM

Οι δεσμευμένες κεφαλίδες αποθηκεύονται σε ένα Red Black Tree (RBT) όπως και στον Jemalloc Allcoator [19], όπου το κλειδί για την προσπέλαση του δένδρου είναι η αρχική διεύθυνσή του δεσμευμένου μπλοκ. Συνεπώς δεδομένου N αριθμού δεσμευμένων μπλοκ, η διαδικασία εισαγωγής διαγραφής απαιτεί $O(log(N))$ χρόνου. Επιπλέον δεδομένου ότι το N είναι άνω φραγμένο, λόγω της εκ των προτέρων δέσμευσης, ο χείριστος δυνατός χρόνος εκτέλεσης (WCET) μπορεί να προϋπολογιστεί. Τα (RBT), παρουσιάζουν καλύτερες επιδόσεις συγκριτικά με άλλες, δομές που χρησιμοποιούνται σε διαχειριστές μνήμες όπως τα Splay Trees [20]. Τέλος σχετικά την υλοποίησή του δένδρου, αποφεύγεται η χρήση αναδρομής, η οποία είναι κατεξοχήν παράγοντας εμφάνισης προβλημάτων που σχετίζονται με την υπερχείλιση της στοίβας. Η συνολική αρχιτεκτονική του διαχειριστή παρουσιάζεται στο σχήμα 10



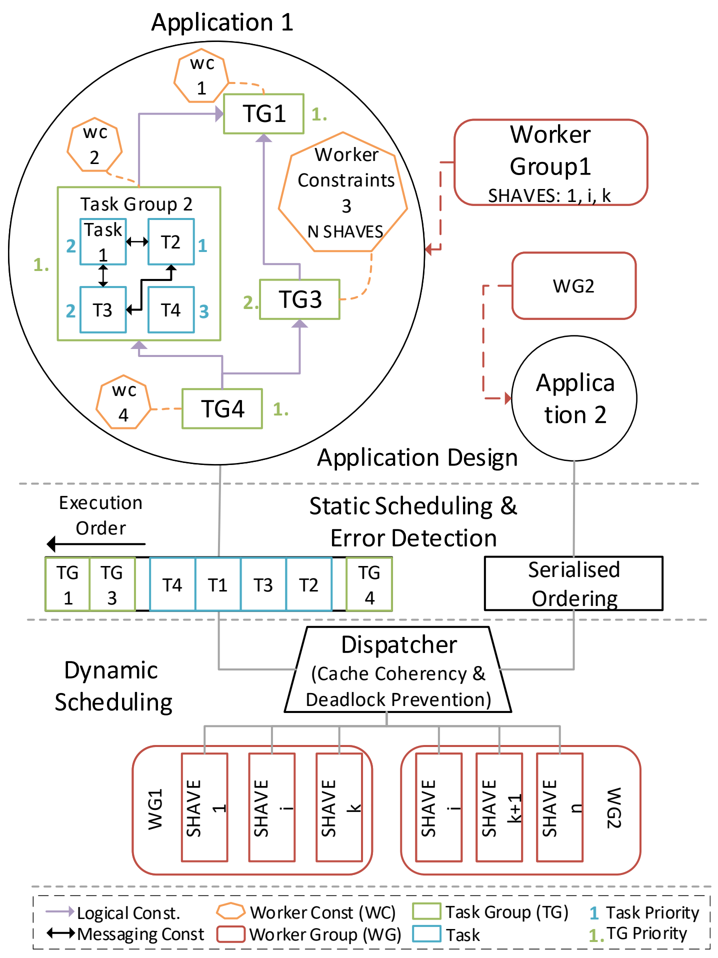
Σχήμα 10: Η συνολική αρχιτεκτονική του διαχειριστή της SPM

Όσων αφορά την πολιτική του allocation, εκτενής έρευνα έχει ήδη πραγματοποιηθεί [21], ωστόσο δεν έχει αναδειχθεί κάποια βέλτιστη τεχνική. Ο συγγραφέας είναι υπέρμαχος της άποψης που περιγράφεται στο [22] και με βάση την οποία μία first-fit-policy πολιτική επιλέχθηκε. Αυτή προσφέρει καλύτερο χρόνο εκτέλεσης με τον κίνδυνο ωστόσο του μεγαλύτερου κατακερματισμού της μνήμης.

Στη συνέχεια περιγράφονται οι διαδικασίες δέσμευσης και απελευθέρωσης της μνήμης. Όταν ο προγραμματιστής καλεί τη συνάρτηση malloc, τότε προσπελαύνεται το bitfield προκειμένου να αναζητηθεί το πρώτο μπλοκ μνήμης το οποίο τηρεί τις απαιτήσεις μεγέθους. Εάν ένα τέτοιο μπλοκ υπάρχει, τότε τα αντίστοιχα bit του bitfield σημαδεύονται ως δεσμευμένα, στη συνέχεια μια κεφαλίδα εξάγεται από τη στοίβα των ελευθέρων και ενημερώνεται με την αρχική διεύθυνση και το μέγεθος του μπλόκ. Έπειτα αυτή προστίθεται στο RBT, όπου βρίσκονται οι δεσμευμένες κεφαλίδες. Η απελευθέρωση είναι η αντίστροφη διαδικασία. Αρχικά αναζητείται το μπλόκ στο RBT και εξάγεται, ενώ ταυτόχρονα καθαρίζονται και τα περιεχόμενα του. Το bitfield ενημερώνεται ώστε να χαρακτηριστούν τα αντίστοιχα bit ως ελεύθερα και τέλος η κεφαλίδα εισάγεται στη στοίβα των ελευθέρων.

Διαχειριστής Υπολογιστικών Πόρων

Ο Διαχειριστής Υπολογιστικών πόρων διαδραματίζει έναν τριπλό ρόλο. α) Προσφέρει ένα API για την ανάπτυξη εφαρμογών, β) ρυθμίζει αυτόματα τα διάφορα μέρη του συστήματος και γ) αναλαμβάνει την δρομολόγηση και εκτέλεση των προγραμμάτων. Η αρχιτεκτονική του συστήματος παρουσιάζεται στο σχήμα 11 και αποτελείται και αυτό από τρία μέρη: α) την περιγραφή της εφαρμογής, β) τη στατική δρομολόγηση και έλεγχο σφαλμάτων και γ) τη δυναμική δρομολόγηση. Σημειώνεται ότι τα τελευταία δύο πραγματοποιούνται με αδιαφανή τρόπο απο το *ParalOS* , το μεν πρώτο κατά τη διάρκειά της φάσης της αρχικοποίησης το δεύτερο κατά τη διάρκεια της εκτέλεσης.



Σχήμα 11: Αρχιτεκτονική του διαχειριστή υπολογιστικών πόρων

API γράφου για την περιγραφή εφαρμογών

Για την περιγραφή των προγραμμάτων χρησιμοποιείται ένα απλό στη χρήση αλλά πολύ αναλυτικό API που βασίζεται στην απεικόνιση με χρήση γράφου. Ο προγραμματιστής δημιουργεί μία ή παραπάνω εφαρμογές, οι οποίες μπορούν να εκτελεστούν είτε σειριακά είτε παράλληλα και στη συνέχεια αναθέτει την εκτέλεσή τους σε ένα η παραπάνω Worker Groups (WGs). Οι εφαρμογές αποτελούνται από ένα η περισσότερα Task Groups (TGs), τα οποία αναπαριστούν μια μοναδική λογική λειτουργία, η οποία μπορεί να διαιρεθεί σε επιμέρους παράλληλα υποπροβλήματα το καθένα εκ των οποίων ονομάζεται Task. Τα TG χαρακτηρίζονται από έναν αριθμό από περιορισμούς (constraints). Οι α) λογικοί περιορισμοί, οι οποίοι ορίζουν εξαρτήσεις δεδομένων μεταξύ των TGs, β) προτεραιότητας (priority), που καθορίζουν τη σειρά εκτέλεσης και γ) εργατών (worker) τα οποία καθορίζουν το είδος τον ελάχιστο αριθμό και την προτεραιότητα των εργατών. Ομοίως και περιορισμοί προτεραιότητας και ανταλλαγής μηνυμάτων κατά την εκτέλεση ορίζονται για τα Tasks.

Δρομολόγηση

Όταν ο προγραμματιστής ολοκληρώσει την περιγραφή των εφαρμογών έχοντας δημιουργήσει όλα τα TG, Task και έχει ορίσει και όλους τους περιορισμούς, το Framework είναι έτοιμο να εφαρμόσει δρομολόγηση σε δύο στάδια. Το πρώτο αποτελεί την στατική δρομολόγηση, η οποία είναι υπεύθυνη για την εύρεση της βέλτιστης σειριακής εκτέλεσης των Task καθώς επίσης και για τον έλεγχο πιθανών σφαλμάτων. Τα σφάλματα μπορεί να είναι λ.χ. η ύπαρξη περισσότερων Task με περιορισμούς ανταλλαγής μηνυμάτων από ότι οι διαθέσιμοι εργάτες, κυκλικοί λογικοί περιορισμοί κ.ο.κ. Για τη σειρά των προγραμμάτων αρχικά τα TG ταξινομούνται με βάση την προτεραιότητά τους. Στη συνέχεια σειριακά ελέγχονται όλα για την ύπαρξη λογικών περιορισμών με κάποιο TG που βρίσκεται πιο μετά στη σειριακή ταξινόμηση. Σε αυτήν την περίπτωση το TG το οποίο βρίσκεται αργότερα τοποθετείται πριν το αρχικό και επανελέγχεται η εγκυρότητα της σειράς με βάση την προτεραιότητα. Η διαδικασία επαναλαμβάνεται μέχρις ότου να μη συμβούν άλλες αλλαγές. Τέλος τα Task μεταξύ των TG ταξινομούνται με βάση την προτεραιότητά τους. Τόσο τα Task όσο και τα TG τοποθετούνται σε FIFO δομές για την γρηγορότερη πρόσβαση σε αυτά κατά τη διάρκεια της εκτέλεσης. Η χρονική πολυπλοκότητα της στατικής δρομολόγησης είναι ίση με $O(N^2 M \log(M+N))$, μέ ένα amortised κόστος πιο κοντά σε $O(N \log N)$. Το συγκεκριμένο κόστος ωστόσο δεν είναι ιδιαίτερα σημαντικό αφού αυτή η φάση της δρομολόγησης γίνεται είτε κατά το στάδιο της αρχικοποίησης, είτε offline και εξάγεται σε ένα δυαδικό αρχείο το οποίο στη συνέχεια φορτώνεται κατά την εκτέλεση.

Στον αντίποδα, η δυναμική δρομολόγηση είναι καθοριστική για την επίδοση του συστήματος γενικότερα. Με σκοπό την ελαχιστοποίηση του άεργου χρόνου των εργατών καθώς επίσης και το φαινόμενο του Task Starvation ο αλγόριθμος υλοποιεί μια non-preemptive πολιτική λόγω της έλλειψής του κατάλληλου υλικού καθώς επίσης και του είδους των τυπικών εφαρμογών που εκτελούνται σε αυτή. Για κάθε εφαρμογή που εκτελείται, πραγματοποιείται συνεχώς polling προκειμένου να εντοπιστούν μέχρι E ελεύθεροι εργάτες, από τα WG που έχουν ανατεθεί στην εφαρμογή. Στη συνέχεια ο δρομολογητής, προσπαθεί να αναθέσει στους εργάτες E Task. Η τεχνική αυτή επαναλαμβάνεται σε όλες τις εφαρμογές μέχρι να ανατεθούν όλα τα Task. Σημειώνεται ότι κατά την ανάθεση ενός Task που έχει περιορισμούς ανταλλαγής μηνυμάτων, απαγορεύεται η εκτέλεση Task που ανήκουν σε άλλο TG. Αυτό πραγματοποιείται προκειμένου να αποφευχθούν πιθανά deadlock. Η πολυπλοκότητα αυτού του αλγορίθμου είναι $O(A\dot{M})$. όπου A ο αριθμός των εφαρμογών που εκτελούνται παράλληλα και M ο αριθμός των TG σε κάθε εφαρμογή.

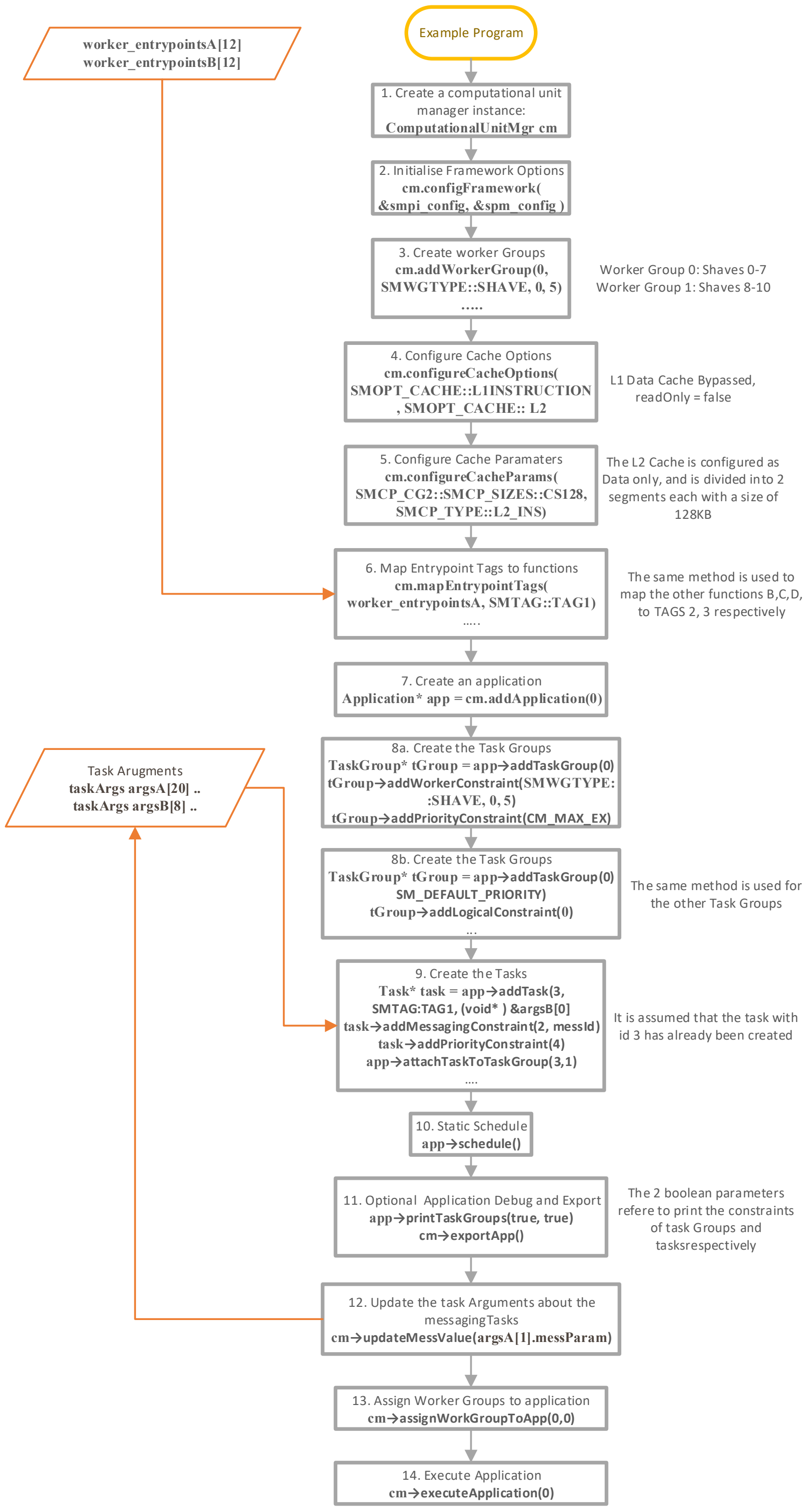
Τέλος ένα ακόμα καθοριστικό κομμάτι του συγκεκριμένου υποσυστήματος είναι ο dispatcher, ο οποίος είναι υπεύθυνος να αναθέτει τα Task στους Εργάτες. Όταν πραγματοποιείται η ανάθεση, αυτός αναλαμβάνει να διεκπεραιώσει όλες τις Low Level λειτουργίες που απαιτούνται για την ορθή λειτουργία του, όπως η εκκαθάριση των καταχωρητών, αρχικοποίηση της στοίβας. Επιπλέον ένας software μηχανισμός αναλαμβάνει να προσφέρει συνάφεια μεταξύ των caches όλου του συστήματος. Συγκεκριμένα όταν προσπελαύνονται δεδομένα που έχουν χαρακτηριστεί εγγράψιμα και αναγνώσιμα, τότε το υποσύστημα αυτομάτως ακυρώνει τις λανθάνουσες μνήμες που αντιστοιχούν στους σχετικούς workers, με βάση κάποιες παραμέτρους που δίδονται από τον προγραμματιστή στη φάση της αρχικοποίησης. Τέτοιες παράμετροι είναι, ο καθορισμός των caches ως μνήμες για εντολές ή δεδομένα, η διαίρεση τους σε επιμέρους τμήματα το μέγεθος του καθενός εξ΄ αυτών κ.ο.κ. Τέλος ο dispatcher είναι υπεύθυνος να απενεργοποιεί τους εργάτες, οι οποίοι δεν μπορούν πλέον να εκτελέσουν κάποιο Task.

Visual Profiler

Ο Visual Profiler στοχεύει στην υποστήριξη των προγραμματιστών κατά τη φάση της εξερεύνησης και βελτιστοποίησης του διαστημικού σχεδιασμού. Κώδικας συγκριτικής αξιολόγησης και δείκτες χρήσης SPM εισάγονται αυτόματα στο framework και τα αποτελέσματα των μετρήσεων είναι σειριακά χρησιμοποιώντας Google's Flatbuffers. Η έξοδος υπολογίζεται offline στο host PC και οι μετρήσεις που παρουσιάζονται περιλαμβάνουν χρήση πυρήνα, συνολικό χρόνο εκτέλεσης, framework με επιβάρυνση, προφίλ λειτουργίας και μέγιστη χρήση μνήμης. Σε επιλεγμένους πίνακες που διαθέτουν μονάδα μέτρησης ισχύος, ο profiler παρέχει επίσης μέση ισχύ και συνολικά αποτελέσματα κατανάλωσης ενέργειας.

Ροή Προγραμματισμού

Στο σχήμα 12 παρουσιάζεται ένα παράδειγμα της χρήσης του API για την περιγραφή και εκτέλεση ενός προγράμματος. Είναι ιδιαίτερά σημαντικό να τονιστεί ότι ο συνολικός αριθμός βημάτων που απαιτείται είναι 13. Αυτό σημαίνει αφενός ότι για τους έμπειρους προγραμματιστές, μειώνεται σε σημαντικό βαθμό ο χρόνος που απαιτείται για την συγγραφή ενός προγράμματος καθώς επίσης και η πιθανότητα λάθους. Επιπλέον για τους αρχάριους χρήστες, το συγκεκριμένο Framework εξομαλύνει αρκετά τη διαδικασία μάθησης και δύναται να αποκρύψει τα ιδιαίτερα χαρακτηριστικά της πλατφόρμας.


Example Program
worker_entrypointsA[12]
worker_entrypointsB[12]
1. Create a computational unit manager instance:
ComputationalUnitMgr cm
2. Initialise Framework Options
cm.configFramework(&smpi_config, &spm_config)
3. Create worker Groups
cm.addWorkerGroup(0, SMWGTYPE::SHAVE, 0, 5)
…..
Worker Group 0: Shaves 0-7
Worker Group 1: Shaves 8-10
4. Configure Cache Options
cm.configureCacheOptions(SMOPT_CACHE::L1INSTRUCTION , SMOPT_CACHE:: L2
L1 Data Cache Bypassed, readOnly = false
5. Configure Cache Paramaters
cm.configureCacheParams(SMCP_CG2::SMCP_SIZES::CS128, SMCP_TYPE::L2_INS)
The L2 Cache is configured as Data only, and is divided into 2 segments each with a size of 128KB
6. Map Entrypoint Tags to functions
cm.mapEntrypointTags(worker_entrypointsA, SMTAG::TAG1)
…..
The same method is used to map the other functions B,C,D, to TAGS 2, 3 respectively
7. Create an application
Application* app = cm.addApplication(0)
Task Arugments
taskArgs argsA[20] ..
taskArgs argsB[8] ..
8a. Create the Task Groups
TaskGroup* tGroup = app→addTaskGroup(0)
tGroup→addWorkerConstraint(SMWGTYPE::SHAVE, 0, 5)
tGroup→addPriorityConstraint(CM_MAX_EX)
8b. Create the Task Groups
TaskGroup* tGroup = app→addTaskGroup(0) SM_DEFAULT_PRIORITY)
tGroup→addLogicalConstraint(0)
...
The same method is used for the other Task Groups
9. Create the Tasks
Task* task = app→addTask(3, SMTAG:TAG1, (void*) &argsB[0]
task→addMessagingConstraint(2, messId)
task→addPriorityConstraint(4)
app→attachTaskToTaskGroup(3,1)
....
It is assumed that the task with id 3 has already been created
10. Static Schedule
app→schedule()
11. Optional Application Debug and Export
app→printTaskGroups(true, true)
cm→exportApp()
The 2 boolean parameters refere to print the constraints of task Groups and tasksrespectively
12. Update the task Arguments about the messagingTasks
cm→updateMessValue(argsA[1].messParam)
13. Assign Worker Groups to application
cm→assignWorkGroupToApp(0,0)
14. Execute Application
cm→executeApplication(0)

Σχήμα 12: Ροή Προγραμματιστικού Παραδείγματος

Table 0.1: Αξιολόγηση του προτεινώμενου Διαχειριστή της *SPM*

Test	**Memory Usage (KB)**			**Execution Time (ms)**		
($n \times B$)	TLSF[7]	Proposed	Diff (%)	TLSF[7]	Proposed	Diff (%)
5Kx4B	30	20	-33	1.342	1.457	+8.6
1Kx8B	10	8	-20	0.458	0.472	+3.2
500x16B	9	8	-11	0.343	0.362	+5.8
100x128B	13	12.8	-1.5	0.289	0.304	+5.3
10x1KB	10.02	10	-0.2	0.206	0.210	+2.1

Αξιολόγηση & Εφαρμογές

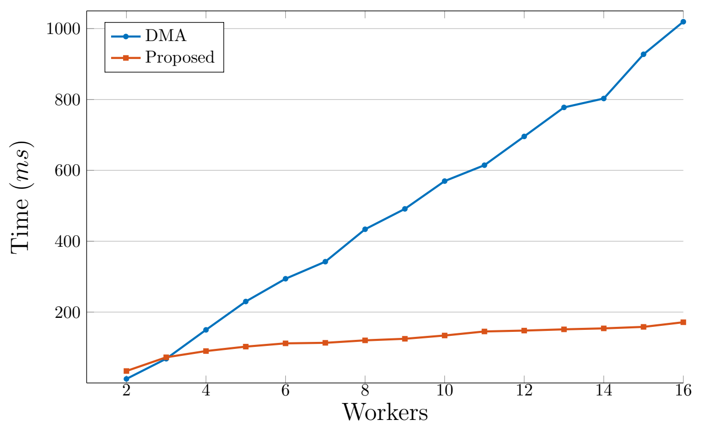
Δοκιμές με Συνθετικά Προγράμματα

Διαχειριστής SPM

Ο διαχειριστής της SPM του *ParalOS* συγκρίνεται με τον TLSF allocator[7], που είναι διάσημος για ενσωματωμένα συστήματα και σχεδιασμένος να ανταποκρίνεται σε περιορισμούς πραγματικού χρόνου. Αμφότεροι οι allocators εφαρμόζονται και αξιολογούνται στην MyriadX VPU. Για την αξιολόγηση, αξιοποιείται ένα πρόγραμμα δοκιμής μνήμης που αποτελείται από $3n$ allocations με μέγεθος B bytes. Το πρόγραμμα αξιολόγησης χωρίζεται σε δύο φάσεις: η πρώτη εκτελεί n διαδοχικά allocations, ακολουθούμενη από τον ίδιο αριθμό αποδεσμεύσεων, στοχεύοντας στον υπολογισμό της μέγιστης χρησιμοποιημένης μνήμης και της χωρικής απόδοσης, ενώ η δεύτερη υλοποιεί ακόμα $2n$ λειτουργίες σε τυχαίο μοτίβο, δηλαδή είναι είτε malloc είτε free, για να εξετάσουν την χρονική απόδοση. Τα αποτελέσματα της δοκιμής αξιολόγησης παρουσιάζονται στον πίνακα0.1

Σχετικά με την χρήση της μνήμης, τα αποτελέσματα δείχνουν πως η προτεινόμενη μέθοδος δεν προκαλεί οποιαδήποτε χωρική επιβάρυνση. Επιπρόσθετα, επιτυγχάνουμε έως και 33% μειωμένο αποτύπωμα μνήμης συγκριτικά με τον TLSF allocator. Σε ακραίες περιπτώσεις, π.χ., όταν ζητείται ένα μόνο byte, το προτεινόμενο framework επιτυγχάνει 93% καλύτερη χωρική απόδοση. Ωστόσο, αυτή η μείωση τείνει να μειωθεί όσο αυξάνεται το ζητούμενο μέγεθος block, παρέχοντας αμελητέα οφέλη σε μεγέθη block μεγαλύτερα του 1KB.

Σχετικά με την χρονική απόδοση, παρατηρούνται υψηλότεροι χρόνοι εκτέλεσης, το οποίο και δικαιολογείται από την διαφορετική φυσική τοποθεσία των μεταδεδομένων του διαχειριστή. Ωστόσο, αυτή η επιβάρυνση είναι σταθερά κάτω από το όριο του 10%, διατηρώντας έτσι τους χρόνους εκτέλεσης συγκρίσιμους. Το περιορισμένο κόστος επιβάρυνσης αποδίδεται στην εκμετάλλευση της ιεραρχίας της cache. Πιο συγκεκριμένα, οι δομές ελέγχου, που είναι αποθηκευμένες στη DDR, είναι προσβάσιμες από τις L1$ και L2$, παρέχοντας έτσι έναν μηχανισμό αντιστάθμισης.



Σχήμα 13: Κλιμακοσιμότητα του προτεινομένου IPC υποσυστήματος συναρτήσει του αριθμού των εργατών

Κλιμακοσιμότητα SMPI

Στη συνέχεια, αξιολογείται η απόδοση του *IPCScheme* στην MyriadX. Συγκρίνεται η κλιμάκωση του με την προεπιλεγμένη χρήση DMA συναλλαγών για επικοινωνίες χρόνου εκτέλεσης. Ως σημείο αναφοράς, ένα συγκεκριμένο σταθερό ποσό των συνολικών αρχικών πληροφοριών μεταδίδεται μεταξύ των εργατών, το οποίο χρησιμοποιεί συγκεκριμένου μεγέθους πακέτα για να μεταδώσει διαδοχικά το δικό τους μερίδιο δεδομένων σε κάθε εργάτη.

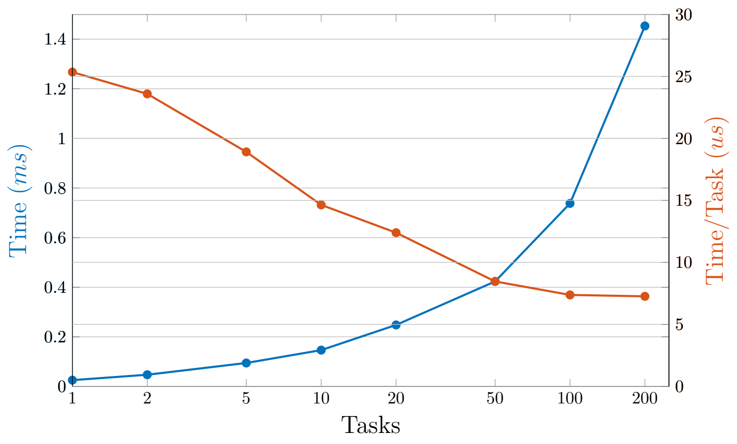
Τα αποτελέσματα που παρουσιάζονται στο σχήμα 13 υποδεικνύουν πως το προτεινόμενο IPC Scheme προσφέρει έως και 6 φορές καλύτερη κλιμάκωση. Αναλυτικότερα, όσο ο αριθμός των εργατών αυξάνεται, ο συνολικός αριθμός των μηνυμάτων αυξάνεται και αυτός πλημμυρίζοντας με αιτήματα τη μοναδική διαθέσιμη DMA μηχανή πάνω στο chip. Σε αντίθεση, η προτεινόμενη αποκεντρωμένη τεχνική buffering χρησιμοποιεί την SPM διασύνδεση υψηλού bandwidth για παράλληλες ανταλλαγές. Ως εκ τούτου, αντί να είναι σχεδόν ανάλογη του συνολικού αριθμού των μηνυμάτων (DMA προσέγγιση) ο IPC χρόνος εξαρτάται αναλογικά μόνο από τα μηνύματα ανά μεμονωμένο εργαζόμενο.

Κλιμακοσιμότητα του δρομολογητή του υποσυστήματος διαχείρισης υπολογιστικών πόρων

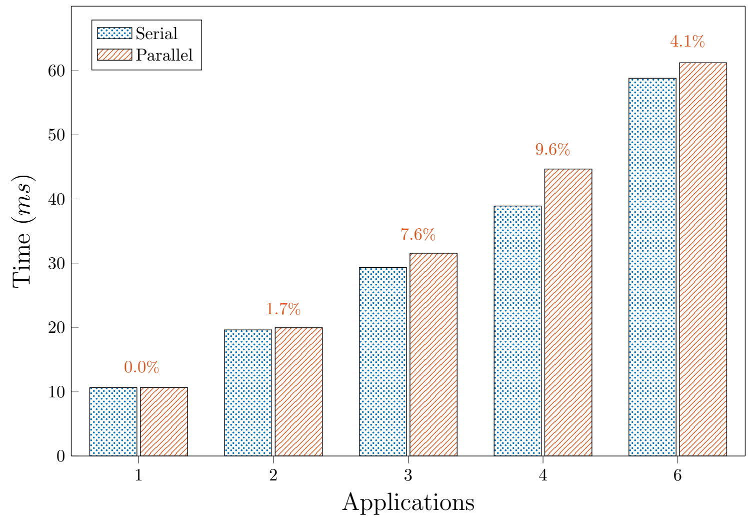
Το επόμενο συνθετικό πρόγραμμα εξετάζει τις δυνατότητες κλιμάκωσης του προτεινόμενου δυναμικού δρομολογητή στη MyriadX. Μία εφαρμογή αποτελούμενη από n Tasks χρησιμοποιήθηκε. Τα Tasks είναι ομαδοποιημένα σε τυχαία Task Groups τα οποία έχουν τεχνητούς λογικούς περιορισμούς (logical constraints) μεταξύ τους. Κάθε Task είναι μία συνάρτηση που επιστρέφει αμέσως χωρίς κανέναν υπολογισμό. Η cache έχει ρυθμιστεί να περιμένει, read-write δεδομένα, έτσι ώστε να χρησιμοποιηθεί το πρωτόκολλο συνοχής της.

Τα αποτελέσματα που παρουσιάζονται στο σχήμα 14 δείχνουν ότι ο δρομολογητής παρουσιάζει γραμμική κλιμάκωση συναρτήσει του αριθμού των tasks. Επιπλέον, το κόστος του δρομολογητή για κάθε task φαίνεται να είναι αντιστρόφως ανάλογο με τον αριθμό των tasks.

Συνήθως, η ταυτόχρονη εκτέλεση εφαρμογών πάσχει από εγγενή ζητήματα, όπως dead-



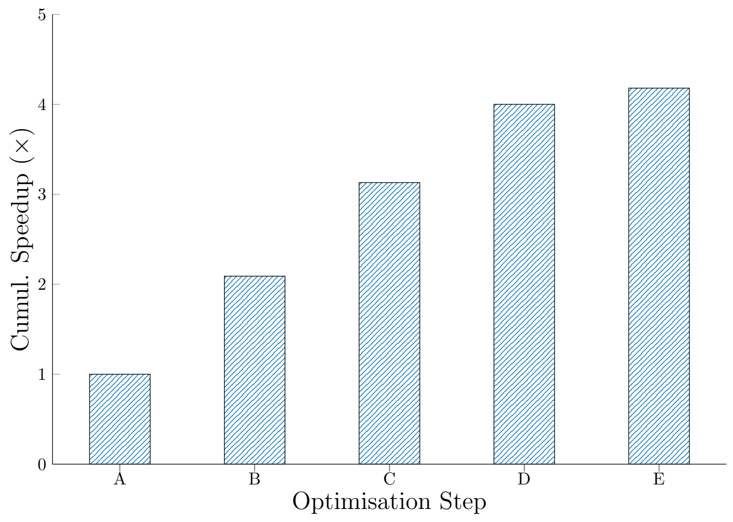
Σχήμα 14: Κλιμακωσιμότητα του δρομολογητή συναρτήσει του αριθμού των Tasks



Σχήμα 15: Σύγκριση παράλληλης και σειριακής εκτέλεσης προγραμμάτων (ΣΝΔ)

locks, συνοχή cache κ.λ.π. Για να ξεπεραστούν αυτά τα bottlenecks, το *ParalOS* χρησιμοποιεί μηχανισμούς οι οποίοι προκαλούν ωστόσο μεγαλύτερη επιβάρυνση. Αυτή η επιβάρυνση μετριέται υλοποιώντας πολλαπλές ξεχωριστές περιπτώσεις, και παράλληλα και διαδοχικά, της μηχανής εκτέλεσης Συνελικτικών Νευρωνικών Δικτύων (ΣΝΔ) που θα περιγραφεί σε παρακάτω ενότητα.

Τα αποτελέσματα παρουσιάζονται στο σχήμα 15, όπου η κυανή διακεκομμένη γραμμή απεικονίζει τον χρόνο εκτέλεσης συναρτήσει του αριθμό των εφαρμογών που εκτελούνται διαδοχικά. Σε αντίθεση, η πορτοκαλί γραμμή απεικονίζει τον χρόνο εκτέλεσης όταν ο ίδιος αριθμός εφαρμογών εκτελείται ταυτόχρονα. Όπως φαίνεται, η επιβάρυνση στην εκτέλεση του προγράμματος είναι μικρή, καθώς κυμαίνεται μεταξύ 1.7% και 9.6% ανάλογα με τον αριθμό εφαρμογών.



Σχήμα 16: Βελτίωση του Rendering με τη χρήστη των παρεχόμενων από το *ParalOS* βελτιστοποιήσεων

Εφαρμογές

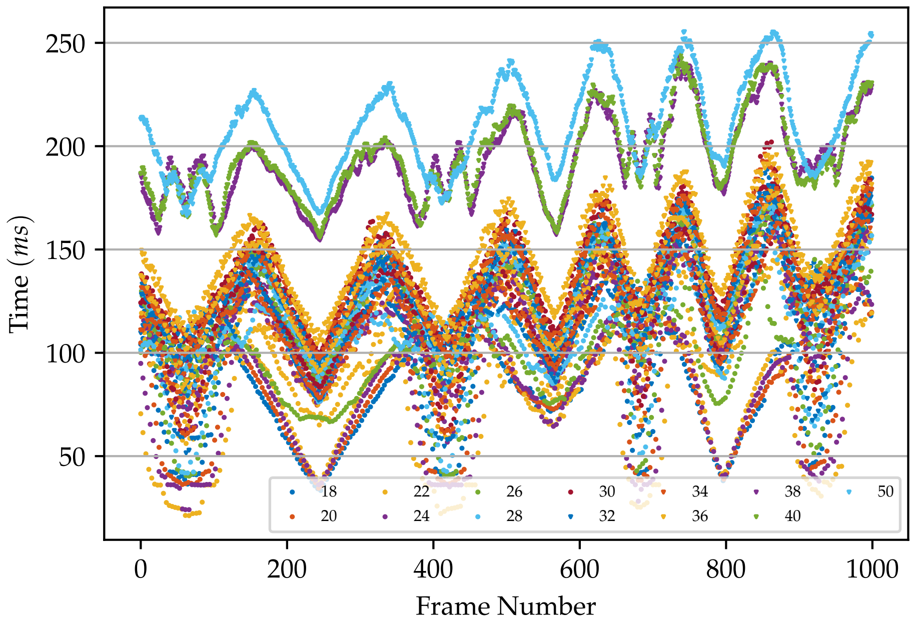
Visual Based Navigation (VBN)

Η συγκεκριμένη κατηγορία εφαρμογών, αφορά τη χρήση οπτικών αισθητήρων, για την αυτόνομη πλοήγηση μη επανδρωμένων οχημάτων. Ένας τέτοιος αλγόριθμος είναι ο HIPNOS [23], ο οποίος χρησιμοποιεί τη γεωμετρία ενός δορυφόρου για να τον κάνει track. Ο συγκεκριμένος αλγόριθμος εκτελέστηκε στην Myriad 2 με τη βοήθεια του *ParalOS* , αποτελείται από διάφορα στάδια ωστόσο το πιο ενδιαφέρον για την αξιολόγηση του Framework αποτελεί το rendering

Το Rendering αφορά την παραγωγή μιας εικόνας, το κάθε pixel της οποίας αναπαριστά μια τιμή βάθους, χρησιμοποιώντας ένα μοντέλο τριγώνων και τη θέση της κάμερας. Ένας αλγόριθμος rasterisation χρησιμοποιείται για να προβάλει τα τρίγωνα πάνω στο πλαίσιο της εικόνας. Χρησιμοποιεί bounding box για να προσδιορίσει τα pixel που βρίσκονται μέσα στην προβολή και στη συνέχεια για καθένα εξ΄ αυτών υπολογίζει την τιμή βάθους.

Αυτή η εφαρμογή είναι κατ΄ εξοχήν δυναμική αφού ο χρόνος εκτέλεσης εξαρτάται σε μεγάλο βαθμό, από τη γωνία της κάμερας. Στο σχήμα 16, παρουσιάζεται ο τρόπος με τον οποίο το *ParalOS* επιταχύνει την εκτέλεση αυτού του προβλήματος,

Το βήμα A, εκφράζει την αρχική υλοποίηση του αλγορίθμου δίχως τη χρήση του *ParalOS* . Στο βήμα B χρησιμοποιείται ο δυναμικός δρομολογητής επιφέροντας μια βελτίωση της τάξης του 2.09×. Στο βήμα C πραγματοποιείται DSE, για την εύρεση του βέλτιστου παραλληλισμού της εικόνας, αυξάνοντας το συνολικό όφελος σε 3.13×. Στο βήμα D, εξετάζονται και βελτιστοποιούνται οι διάφορες παράμετροι της cache, όπως ο αριθμός των τμημάτων της και ο καθορισμός των δεδομένων της γεωμετρίας του μοντέλου ως read-only. Το τελευταίο βήμα (E) αφορά τη χρήση συγκεκριμένων χαρακτηριστικών του προβλήματος, όπως π.χ. το γεγονός ότι τα κεντρικά τμήματα της εικόνας είναι περισσότερο πιθανά να περιέχουν κάποιο



Σχήμα 17: Design Space Exploration για την εύρεση του αριθμού των διαιρέσεων της εικόνας

κομμάτι του μοντέλου από τα ακριανά. Άρα στα Task που αφορούν τα κεντρικά τμήματα δίδεται μεγαλύτερη προτεραιότητα ώστε να εκτελεστούν πρώτα.

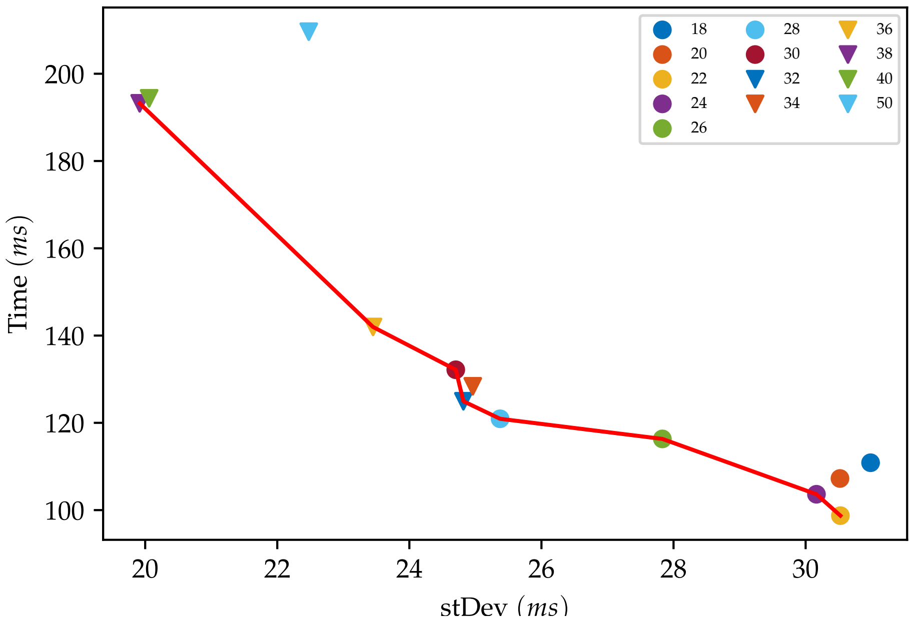
Νωρίτερα αναφέρθηκε η έννοια του βέλτιστου παραλληλισμού, αυτό αφορά τον αριθμό των παράλληλων Task στα οποία θα διαιρεθεί ο υπολογιστικός φόρτος του αλγορίθμου. Σε αυτό το σημείο επιδρούν δυο αντίρροπα χαρακτηριστικά. α) πολύ μεγάλος αριθμός σημαίνει ότι θα πρέπει όλη η γεωμετρία του δορυφόρου να ελεγχθεί πολλάκις, φθάνοντας σε σημείο να υπερκεράσει τους υπολογισμούς και β) πολύ μικρός αριθμός εγκυμονεί τον κίνδυνο σειριοποίησης του προβλήματος. Αυτό συμβαίνει διότι εάν το μοντέλο προβάλλεται σε ένα μικρό μέρος της εικόνας, τότε όλος ο φόρτος θα συγκεντρωθεί σε έναν πολύ μικρό αριθμό Task, προκαλώντας έτσι την μη αποδοτική χρήση όλων των υπολογιστικών πόρων.

Για αυτό το λόγο καταστρώθηκε το παρακάτω τεστ. 1000 διαφορετικές γωνίες θέασης χρησιμοποιήθηκαν, για να ελεγχθούν διαφορετικές τιμές παραλληλοποίησης. Στο σχήμα 17 αποτυπώνεται το σύνολο των αποτελεσμάτων συναρτήσει του χρόνου εκτέλεσης.

Αρκετές φορές σε εφαρμογές πραγματικού χρόνου, είναι ιδιαίτερα σημαντική εκτός από γρήγορα αποτελέσματα, η παράγωγή τους σε σταθερό χρόνο. Στο σχήμα 18 παρουσιάζεται το μέτωπο Paretto μεταξύ του μέσου χρόνου εκτέλεσης και της τυπικής απόκλισης. Τα αποτελέσματα δείχνουν ότι η ταχύτερη εκτέλεση παρουσιάζεται για διαίρεση σε 22 τμήματα, ενώ μια καλή επιλογή που συνδυάζει μικρότερη απόκλιση είναι τα 32 τμήματα.

Μηχανή Εκτέλεσης Συνελικτικών Νευρωνικών Δικτύων

Στην περιγραφή των VPU αναφέρθηκε ότι ένας σημαντικός τομέας εφαρμογών που χρησιμοποιούνται είναι η μηχανική μάθηση και η όραση υπολογιστών. Τα Συνελικτικά Νευρωνικά

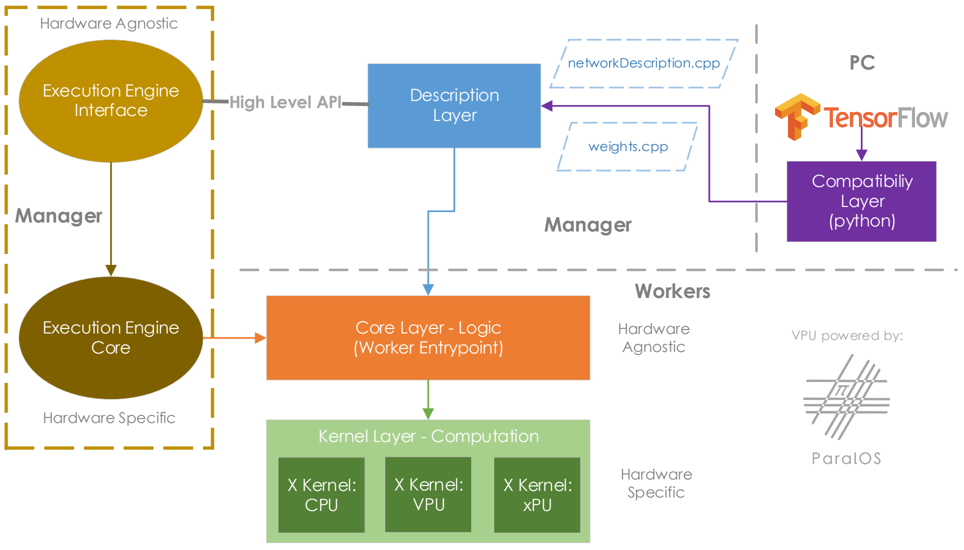


Σχήμα 18: Το μέτωπο Pareto για τον μέσο χρόνο εκτέλεσής και της τυπική απόκλισης για τα διάφορα μεγέθη διαίρεσης της εικόνας

Δίκτυα αποτελούν μια διαδεδομένη κατηγορία προβλημάτων μηχανικής μάθησης που σχετίζονται με εικόνες. Για το λόγο αυτό αναπτύχθηκε μια μηχανή εκτέλεσης ΣΝΔ, η οποία στηρίζεται στο *ParalOS* . Στο σχήμα 19 παρουσιάζεται η αρχιτεκτονική της μηχανής που αναπτύχθηκε.

Όπως φαίνεται αποτελείται από 4 επίπεδα αφαίρεσης καθώς και ένα υποσύστημα εκτέλεσης. Η μηχανή μπορεί να χρησιμοποιηθεί για την εκτέλεση δικτύων τόσο σε VPU όσο και σε CPU, οπότε τα διάφορα επίπεδα παρουσιάζουν διαφορετικό βαθμό εξάρτησης από το υλικό. Το πρώτο επίπεδο είναι αυτό της μετατροπής, η λειτουργία του οποίου είναι να μετατρέψει ένα μοντέλο από το δημοφιλές Framework Tensorflow σε μια ενδιάμεση περιγραφή κατανοητή από το επόμενο επίπεδο το Description. Το συγκεκριμένο προσφέρει μια βιβλιοθήκη για την περιγραφή δικτύων και είναι ανεξάρτητο της πλατφόρμας εκτέλεσης. Το επόμενο επίπεδο, αυτό του core, εμπεριέχει τη λογική για την εκτέλεση των διαφόρων Layer του δικτύου και αποτελεί το entrypoint των εργατών, κατά κύριο λόγο είναι κι αυτό ανεξάρτητο της πλατφόρμας εκτέλεσης. Το χαμηλότερο επίπεδο περιέχει τους υπολογιστικούς πυρήνες που εκτελούν τους υπολογισμούς και είναι διαφορετικοί για κάθε σύστημα εκτέλεσης. Τέλος, ο μηχανισμός εκτέλεσης προσφέρει ένα κοινό API για όλες τις πλατφόρμες, ενώ ταυτόχρονα αναθέτει την εκτέλεση του δικτύου στα ήδη υπάρχοντα Frameowork όπως το *ParalOS* , για τις VPU

Στον πίνακα 0.2, παρουσιάζεται η σύγκριση της συγκεκριμένης μηχανής με άλλες δημοσιευμένες εργασίες. Όπως φαίνεται το *ParalOS* , καταφέρνει να διατηρήσει την επιπλέον επιβάρυνση κάτω του 10%, συγκριτικά με ειδικά βελτιστοποιημένες υλοποιήσεις [24]. Η μόνη



Σχήμα 19: Αρχιτεκτονική του μηχανισμού εκτέλεσης ΣΝΔ.

εξαίρεση αποτελεί το παράδειγμα του MNIST όπου βέβαια λόγω του ιδιαίτερα μικρού μεγέθους του δικτύου έχει υιοθετηθεί μια εξαιρετικά αντισυμβατική υλοποίηση που στηρίζεται στην ύπαρξη όλου του κώδικα και των δεδομένων στην SPM [25]. Μέχρι και σε αυτήν την οριακή περίπτωση πάντως οι επιδόσεις κυμαίνονται σε αποδεκτά επίπεδα.

Περί της μείωσης του προγραμματιστικού φόρτου

Μια εξαιρετικά σημαντική συνεισφορά του *ParalOS* , είναι η μείωση του απαιτουμένου χρόνου για την ανάπτυξη εφαρμογών στις ιδιαίτερα ετερογενείς αρχιτεκτονικές των VPUs. Συγκεκριμένα, παρατηρείται μια μείωση της τάξης του $2-3\times$ στο χρόνο ανάπτυξης. Επιπλέον, λόγω του καλύτερα δομημένου προγραμματιστικού παραδείγματος, ο παραγόμενος κώδικας είναι ευκολότερος στη συντήρηση και λιγότερο πιθανόν να περιέχει λάθη. Τέλος σημειώνεται ότι η μεταφορά προγραμμάτων από πλατφόρμα σε πλατφόρμα γίνεται με ελάχιστες τροποποιήσεις, εν αντιθέσει με την τωρινή κατάσταση η οποία απαιτεί σημαντική ενασχόληση.

Table 0.2: Αξιολόγηση του *ParalOS* με τη χρήση ΣΝΔ

CNN Network	**Execution Time (ms)**			
	[24]	[25]	Proposed	Diff. (%)
CIFAR-10	7.24	-	7.80	+7.7
Ship-Detection	9.18	-	9.91	+7.9
MNIST	-	0.35	0.64	+82.9

Συμπεράσματα

Εν κατακλείδι, στη συγκεκριμένη διπλωματική παρουσιάστηκε το *ParalOS* , ένα Framework για τον αποδοτικό προγραμματισμό και DSE σε VPUs, το οποίο επιτρέπει την πλήρη αξιοποίηση των χαρακτηριστικών της εκάστοτε πλατφόρμας. Τα υποσυστήματα του είναι στοχευμένα και βελτιστοποιημένα για τον εξαιρετικά ετερογενή χαρακτήρα αυτών των συστημάτων και περιλαμβάνουν, ένα υψηλού επιπέδου API για περιγραφή εφαρμογών, δυναμικό δρομολογητή, ένα καινοτόμο διαχειριστή της Scratchpad μνήμης, ένα αποκεντρωμένο σύστημα ενδοεπικοινωνίας και τέλος έναν οπτικό profiler. Τα πειραματικά αποτελέσματα δείχνουν μια βελτίωση της τάξης του 4.2× σε εφαρμογές δυναμικού περιεχομένου, ενώ μια περιορισμένη επιδείνωση της απόδοσης συγκριτικά με χειροποίητες υλοποιήσεις, εκφράζει μια συντηρητική ανταλλαγή επίδοσης - προγραμματιστικής ευκολίας. Τέλος μεμονωμένα υποσυστήματα του *ParalOS* παρουσιάζουν σημαντικές βελτιώσεις έναντι των παρεχομένων βιβλιοθηκών από τον κατασκευαστή η άλλων καθιερωμένων λύσεων.

Chapter 1

Introduction: The State of the Industry

1.1 A New Era of Computing: AI & Vision Processing

The emergence of AI, lead to the dawn of a new era in the computing world [9]. Applications like Neural Networks, Computational Photography and overall the concept of *Smart Everything*, have revolutionised the industry. This revolution is materialised at the cost of ever increasing computational demands. Amodei etal [1] suggest that the AI complexity is increasing exponentially. Even more disturbing is the rate of increase, which by far surpasses the 18 month one imposed by Moore's Law; and is calculated to be between 3-4 months as shown in Fig. 1.1

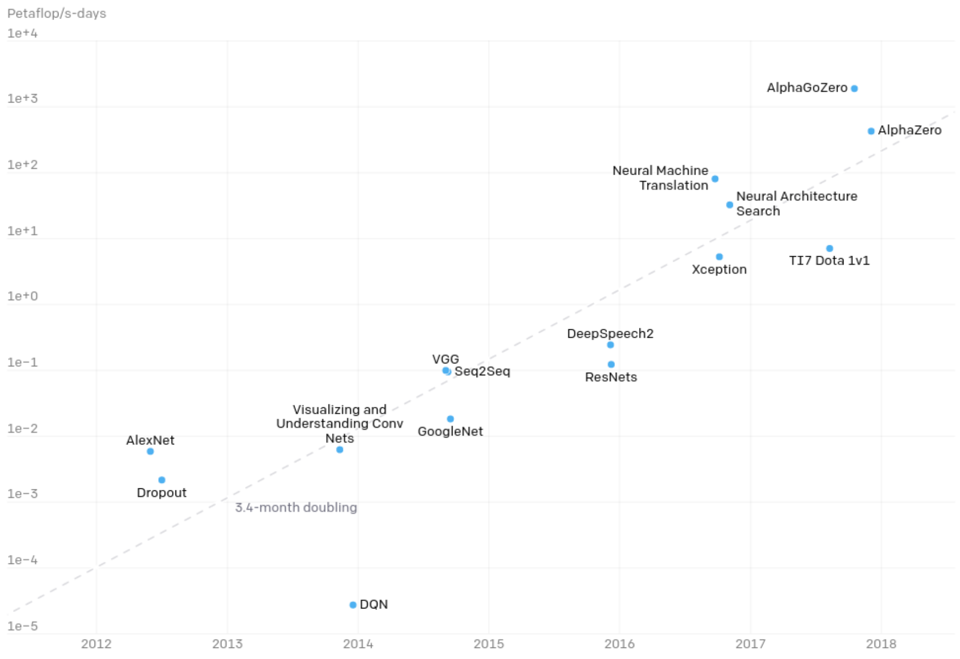


Figure 1.1: The total amount of compute, in petaflops-days, used to train selected network [1]

This figure upon closer examination from an algorithmic perspective, reveals one more truth. It is not just the rapid increase in complexity, but the applications exhibit great algorithmic diversity. These facts signal that a fundamentally different approach is required for the computing industry.

1.2 Nothing is free: The End of General Computing

Ted Hughes famously said “Nothing is free. Everything has to be paid for. For every profit in one thing, payment in some other thing. For every life, a death.”. The last decades *Dennard Scaling* has allowed a *free* exponential increase, that was the result of semiconductor fabrication innovations. Since 2005, the transistor scaling started to approach its physical limits, therefore the standardisation of multicore systems was adopted. The next decade is perhaps the time that the payment for all of those years is due and that payment is expressed via the the *Memory* & *Power* Wall.

Regarding the *Power Wall* [26], Esmaeilzadeh etal concluded, that due to the increased density of transistors power management will be a limiting factor. More specifically, the power-per-area has increased, in a way that power dissipation is not feasible. This means that up to 50% of the chip must be powered off at any time. As a result, nearly a 24-fold gap from a target of doubled performance per iteration, will be left by 2024.

Regarding the *Memory Wall*, concerns were expressed as early as 1996 in a work by Wulf [27]. He identified that the processing capabilities increased at a 50% faster rate than the memory. This is a trend that was not improved, therefore a huge gap between computational capabilities and memory performance exists. Sites [28], said the famous quote “It’s the memory Stupid!” and expressed that over the coming years, the memory subsystem design will be the only important design issue for processors. History though, did not validate this prediction and in its core the same architecture designs are still used today.

1.3 Heterogeneity to the Rescue

The solution to extend the exponential growth of computing capabilities, lies in the introduction of *heterogeneity.* Heterogeneous computing [29] refers to systems that use more than one kind of processors or cores or memory hierarchies. The performance gains or energy efficiency is achieved not just by adding the same type of processors, but using dissimilar coprocessors, that incorporate specialised processing capabilities, fine tuned for specific tasks.

Thompson & Spanuth [10] believe that the introduction of AI applications will reintroduce device fragmentation as specialised accelerators appear to be the only way forward for the time being. Hennessy [9] as well as Intel [30], claim that this is the golden era of computer architecture, as other more exotic alternatives [2] are not ready for widespread adoption.

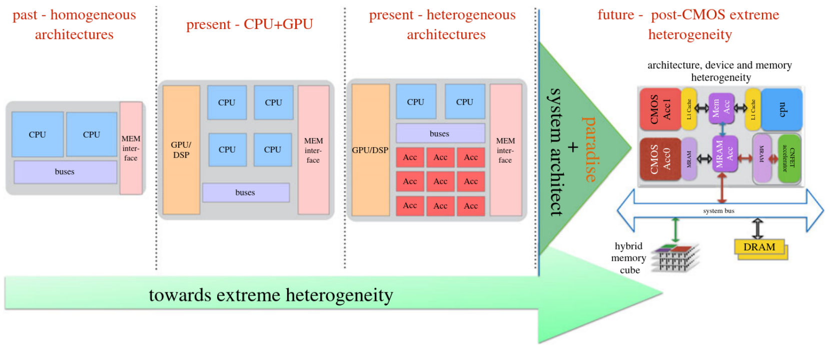

past - homogeneous architectures
present - CPU+GPU
present - heterogeneous architectures
future - post-CMOS extreme heterogeneity
architecture, device and memory heterogeneity
paradise + system architect
system bus
hybrid memory cube
DRAM
towards extreme heterogeneity

Figure 1.2: The establishment of Compute heterogeneity as the new standard [2]

This approach is better depicted in Figure 1.2, as the industry transitions from the introduction of GPUs to totally Heterogeneous Devices, consisting of General Purpose Cores, GPUs, an array of accelerators and a sophisticated memory hierarchy.

1.4 Programmability & Heterogeneity: Hitting an impassable Wall?

The most important question regarding the Heterogeneous Computing is the following *What about the Programmability and the Legacy Code?* Before answering this question, let's take a step back and answer this question instead *Why use Heterogeneity, in the first place ?*

The obvious answer is to compensate for the end of Moore's Law. On a second thought this is more of an intermediate step, than the true cause which actually drives the whole industry the last decades. This root cause is the need to keep delivering **exponentially better products** in the **same amount of time**. The term products refers to both the industry and research.

The first part of this sentence mentions *better products*, but what does a product consist of? A simplified approach is to divide it into two components i) the algorithm and ii) the computational platform. As established, the algorithms improve with high pace and the platforms, because of the heterogeneity, manage to keep up.

The second part of this sentence indirectly introduces the concept of programmability and more specifically the ability to create exponentially more capable software in the same amount of time. Is this feasible? Intel, at least at first glance, does not seem to believe so [30] and has actually stated that

$$\textbf{Generality} \propto \textbf{1/ArchitectureHeterogeneity}$$

Should this allegation be true, it will have ground breaking implications not limited to the boundaries of the computer society. This relation builds a new *Programmability Wall*, which signals the end of a decades old model of constant technological advancements. These constant advancements have additionally shaped the socioeconomic models that govern modern societies. The cause has become the goal at the same time, leading to a circular, endless pursuit of innovation. If this circle is disrupted an expanded conversation between all the stakeholders will need to be initiated, with undefined consequences for the modern way-of-living.

1.5 Possible Solutions & Motivation

Returning to the initial question about programmability, the majority of the computer society, including the author, believes that there is a solution. This solution is found in new *Software Frameworks.* Leiserson, paraphrasing Feyman's famous quote from the 60's "There is plenty of room at the bottom" [31]; suggests designing new programming language paradigms and frameworks that are grounded on heterogeneity [11]. These frameworks like the *Popcorn Framework* for heterogeneous devices[32] can not only provide the abstraction required for fluent programming but also mend the implication of *Wirth's Law* regarding the software bloating [33].

In this context, the idea for creating software for emerging embedded heterogeneous devices, namely Vision Processing Units (VPUs), was conceived. The framework's target is to abstract the Hardware Details, provide a foundation for fast and efficient application development, while enabling low level and platform specific optimisation, which led to the selection of the VPU in the first place. Taking all of that into consideration and with strong motivation and dedication *ParalOS* was born.

Figure 1.3: The *ParalOS* Logo

Chapter 2

Vision Processing Units (VPUs) & Frameworks

2.1 Vision Processing Units: Heterogeneity at its best

2.1.1 A gentle Introduction to VPUs

Vision Procesing Units (VPUs) is an emerging class of embedded devices. A VPU at it's very core is an inherently heterogeneous device. They employ a number of different compute units ranging from general purpose cores, specialised SIMD and or VLIW processors, to Hardware Filters. The heterogeneity is expanded in the Memory Subsystem as well. They include both caches and Scratchpad memories, a not very common approach, in addition to DDR. VPUs excel at Image Processing, Computational Photography and AI, so their most common usage is as a standalone microcontroller or a vision & AI accelerator.

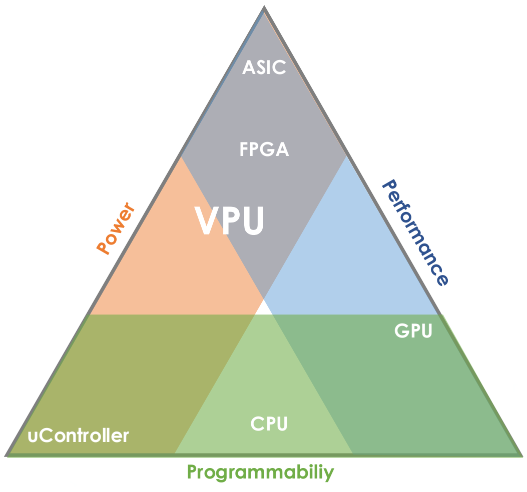


Figure 2.1: VPUs in the Spectrum of Embedded Compute Devices.[1]

[1]Disclaimer:The figure is for illustrative purposes only and it is focused on the VPUs. The author would like to raise a waver, should it spark a consist between enthusiastic supporters of the depicted platforms.

When comparing this class of platforms, in terms of performance, it is placed between the CPUs and GPUs, while doing so in the smallest power envelope. Depending on the application nature, VPUs can outperform in performance/watt even the FPGA. When asking why VPUs are needed in the best sanwer was provided by *Movidius*, during the launch of *Myriad 2*. They explained their point by noting that *it's no longer sufficient to render a complex scene as a GPU does; the device must understand it.*

2.1.2 Myriad 2 VPU

The main target platform of this thesis is Intel's Movidius™ Myriad™ 2 VPU Vision Processing Unit (VPU) [12], but the concepts that are discussed can be genralised into other similar platforms. It is developed by Intel's Perceptual Computing Group to accelerate adoption of visually intelligent devices. Recently [34] Myriad 2 has passes the radiation tests making it suitable for Low Earth Orbit space missions.

Specifications Overview

The main characteristics of the SoC are the following, while a block diagram is shown in the figure 2.2:

- **Ultra Low Power Design**. For mobile and connected devices where battery life is critical,Intel's MyriadTM2 VPU provides a way to combine advanced vision applications in a low powerprofile. This enables new vision applications in small form factors that could not exist before. Moreover the 20 independent power islands, enable fine grained power management. As a result, this low power processor, allows the use of the device in space applications with extremely tight power envelope.
- **Heterogeneous, high throughput, multi-core architecture** based on
  - 2 x 32 bit LEON 4 SPARC-V8 RISC processors
  - 12 VLIW 128-bit vector SHAVE Processors optimised for machine vision
  - Configurable hardware accelerators for image and vision processing, with line-buffers enabling zero local memory access ISP mode
  - Homogeneous, centralised memory architecture; 2MB of on-chip memory with 400 GB/sec of sustained internal memory bandwidth
  - 512MB of LPDDR3 main memory
  - Multi level run-time configurable Cache Infrastructure
- **Small-area footprint**: To conserve space inside mobile, wearable, and embedded devices, as well as small satellites (ie. cubesats), Myriad 2 was designed with a very small footprint that can easily be integrated into existing products
- **Rich set of external communication peripherals**.

- 12 Lanes MIPI, 1.5 Gbps per lane configurable as CSI-2 or DSI.
- CIF, LCD Parallel Interfaces.
- I2C, SPI, UART for control and configuration.
- I2S for audio input.
- Bank of configurable GPIO, PWM.
- USB3 with integrated PHY.
- 2-slot SDIO.
- Debug interface.
- 1 Gbit Ethernet.

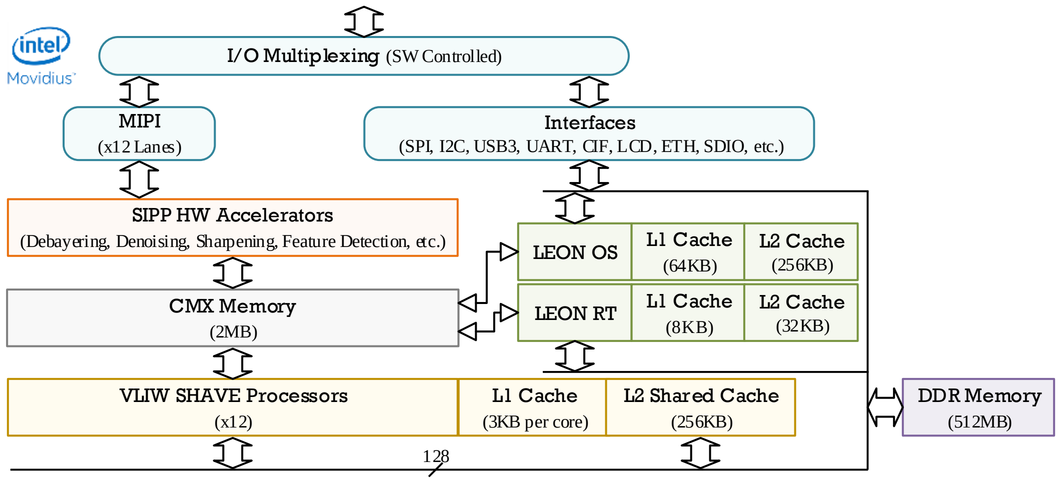

intel
Movidius
I/O Multiplexing (SW Controlled)
MIPI (x12 Lanes)
Interfaces (SPI, I2C, USB3, UART, CIF, LCD, ETH, SDIO, etc.)
SIPP HW Accelerators (Debayering, Denoising, Sharpening, Feature Detection, etc.)
CMX Memory (2MB)
LEON OS
L1 Cache (64KB)
L2 Cache (256KB)
LEON RT
L1 Cache (8KB)
L2 Cache (32KB)
VLIW SHAVE Processors (x12)
L1 Cache (3KB per core)
L2 Shared Cache (256KB)
DDR Memory (512MB)
128

Figure 2.2: Myriad 2 Block Diagram

Despite the fact that the Myriad 2 SoC has a number of subsystems, in the following subsections a more detailed analysis of the most important of them is presented.

General Purpose Leon Processors

Myriad 2 deploys two high performing LEON4 SPARCv8 General Purpose Processors each with distinct functionality. A block overview of the LEON 4 processor is provided below.

- **LeonOS** or LOS, is the main processor of the Platform as, most often, after booting the applications entry point is designed to LOS. It belongs to the CPU subsystem (CSS) and it is been destined as the main communication and control unit with the outside world via the external communication peripherals: I2C blocks, I2S blocks, SPI blocks, UART, GPIO, ETH and USB3.0. LOS is supported by relatively large Caches L1 (32 KB) and L2 (256 KB) caches, which allows booting Real Time Operating System on it, like RTEMS [35]. This block also offers an AHB DMA engine for more optimal data transfer via the external peripherals. Finally despite the control-oriented role of this CPU, it boasts significant compute capabilities, since it employs high performing ALU and FPU units [36] and currently it is considered one of the most high performing CPU's for space applications [8] Beside handling the external interfaces and communication Leon OS could also control SHAVE processors imaging algorithms.

- **LeonRT** or LRT Is the second of the SPARC CPUs. It belongs to the Media subsystem(MSS), an architectural unit designed for allowing external connections with imag-ing devices (camera sensors, LCDs, HDMI controllers etc.) as well as allowing use of the Hardware (HW) filters available in Myriad2. As such it is comprised by the MIPI, LCD, CIF interfaces, the SIPP HW filters and well as the AMC block which enables connections between these and CMX (SRAM) memory. Coordinating frame input and controlling the pipelines set in place usually require some effort. As such the Myriad2 platform offers the Leon RT RISC as part of the MSS. LRT as a co-processor is supported with smaller L2 cache memory (32 KB) than LOS. LRT is only one arbiter away from any Interface or HW filter register settings so it can efficiently change any required parameters of the MSS blocks with the minimum amount of delay due to bus arbitration.

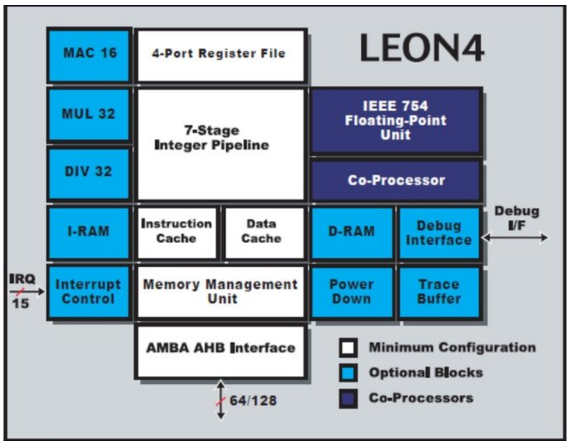


Figure 2.3: LEON 4 block diagram

SHAVES and Microprocessor Array UPA

The majority of the processing power of the Myriad 2 VPU orignates from the 12 proprietary, custom-designed SHAVE v3.0 processors. SHAVE stands for (Streaming Hybrid Architecture Vector Engine) and it contains wide and deep register-files coupled with a Variable-Length Long Instruction-Word (VLLIW) controlling multiple functional units including extensive SIMD capability for high parallelism and throughput at both a functional unit and processor level. The SHAVE processor is a hybrid stream processor architecture combining the best features of GPUs, DSPs and RISC with both 8/16/32 bit integer and

16/32 bit floating point arithmetic as well as unique features such as hardware support for sparse data structures. The architecture is designed to maximise performance-per-watt, while maintaining ease of programmability, especially in terms of computer vision and machine learning workloads. In conclusion they adopt a computation-heavy role, while control and more logic-heavy operations are left for the 2 RISK processors to handle. An architectural overview is presented on figure 2.4

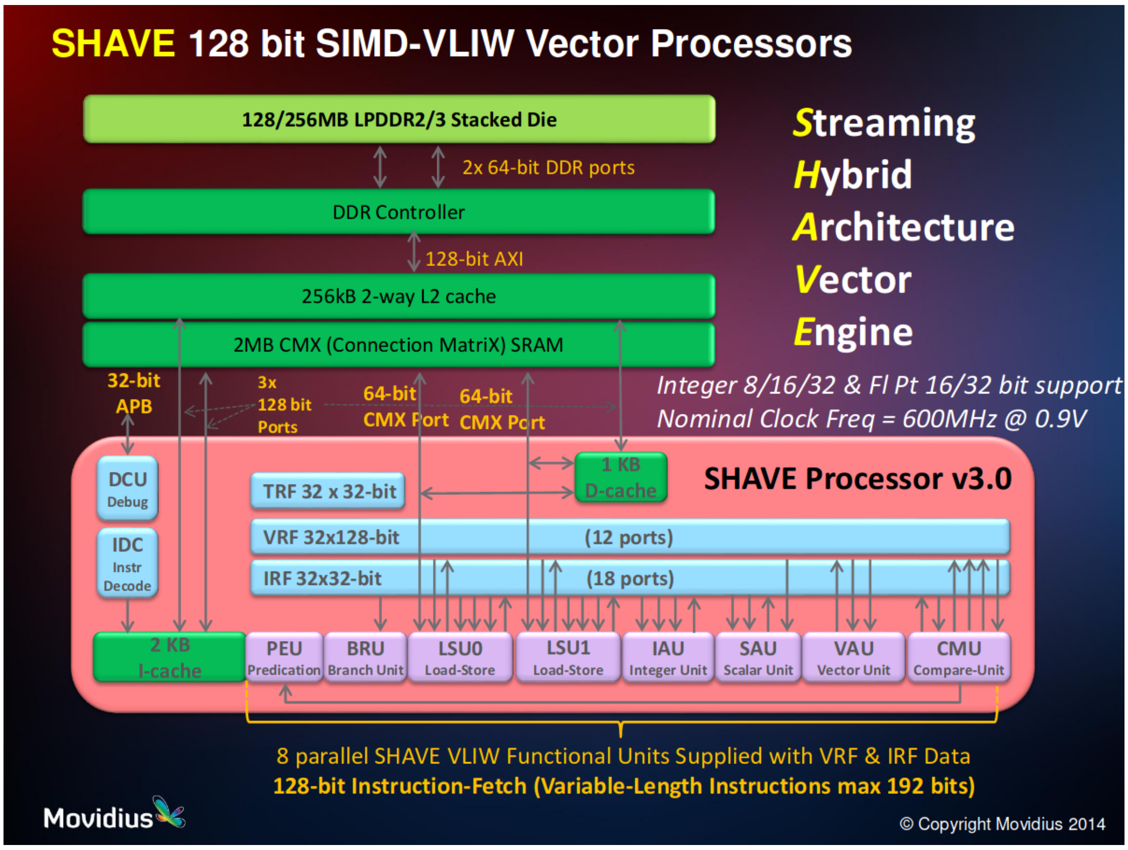


Figure 2.4: Shave v3.0 Block diagram [3]

Twelve (12) of these processors compose the Microprocessor Array (UPA) and they share a common 256 KB L2 (I+D) cache as well each shave has priority access to a CMX slice. UPA can be controlled from the LEON processors with minimal overhead.

Hardware Filters

The SoC employs more tha 20 programmable Hardware Filters to accelerate Imaging/Vision Kernels line Edge and Convolution Operator [3, 37]. Each accelerator has multiple memory ports to support the memory requirements and local decoupling buffers to minimize instantaneous bandwidth to and from the 2-Mbyte multicore memory subsystem. A local pipeline controller in each filter manages the read and writeback of results to the memory subsystem. The filters are connected to the multicore memory subsystem via a crossbar, and each filter can output one fully computed pixel per cycle for the input data, resulting in an aggregate throughput of600 Mpixels per second at 600 MHz.However, previous work suggest that these filters are designed for reducing power, instead of increasing performance [38]. They can be controlled via the SIPP framework that is described later.

Memory Subsystem Caches and CMX

- **CMX** is short for *Connection Matrix* and it is [37] a 2 MB SRAM user controlled memory, acting like a scratchpad area. The CMX block comprises 16 blocks (or slices) of 128 Kbytes, which in turn comprise four 32-Kbyte RAM instances organized as 4,096 words of 64 bits each, which are independently arbitrated, allowing each RAM block in the memory subsystem to be accessed independently. The 12 SHAVEs acting together can move (theoretical maximum) 12×128 bits of code and 24×64 bits of data, for an aggregate CMX memory bandwidth of 3,072 bits per cycle (1,536 bits of data). Finally it is noted that CMX is NUMA memory meaning that each SHAVE has higher bandwidth/lower power access to its "own" local slice, however it is also mentioned that Slice locality is a weak concept, each SHAVE can access any other slice in CMX at the same cost, but inter-slice routing resources are finite. In addition, a slave accessing data in its own slice is more energy-efficient [39].

- **Cache Infrastructure**. The Soc has a number of physically different caches and cache hierarchies. A map of theese caches is provided below:

PU	Type	Size	Associativity	line	Policy
SHAVE[N]	L1 I	2 KB	2-way	16 bytes	read-only cache
SHAVE[N]	L1 D	1 KB	Directly Mapped	16 bytes	write-back or write-through
SHAVES	L2	256 KB	2-way, 1-8 partitions	64 bytes	write-back
LeonOS	L1 I	32 KB	2-way	32 bytes	read-only
LeonOS	L1 D	32 KB	2-way	32 bytes	write-through
LeonOS	L2	256 KB	4-way	64 bytes	write-through or copy-back
LeonRT	L1 I	4 KB	2-way	32 bytes	read-only
LeonRT	L1 D	4 KB	2-way	32 bytes	write-through
LeonRT	L2	32 KB	4-way	64 bytes	write-through or copy-back

Table 2.1: Myriad 2 Cache system overview

- **DDR** is the main memory of the chip and has a size of 512MB. It is a volatile LPDDR3 and all 14 processors may execute code, as well as access data from the DDR via the cache infrastructure.

Direct Memory Access (DMA) controller

All these processors require cocurent memory access to both CMX and the DDR, as a result a dedicated DMA unit is required. The SoC's CMX DMA resides between the 128-bit MXI bus and CMX memory [39]. It provides high bandwidth data transfers between CMX and DDR in either direction. It also supports data transfers from DDR back to DDR or from CMX to CMX, allowing data to be relocated within the same physical location. shows a high level description of the DMA engine. The unit of work in the DMA engine is expressed though transaction tasks. Up to four linked lists of transactions are maintained in system memory, thus the DMA capability of serving transactions is not unlimited and can be easily flooded with requests if the programmer makes unregulated use of it.

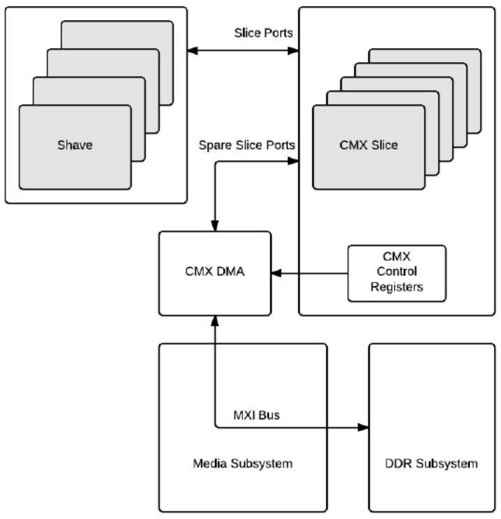


Figure 2.5: DMA engine Overview

Myriad Development Kit & Build System

The Myriad 2 Development KIT (MDK) comprises common code, which include driver and components,documentation support and toolchains that are required to develop application for the Myriad Family products. Part of the the toolchain is a quite extensive and complex build system based on *GNU Makefile*.

The build system is responsible for cross compiling the object code for the various heterogeneous processors. Afterwards the linker generates the memory map, as per the instructions of the programmer.

Streaming Image Processing Pipeline Framework (SIPP)

The model used by many image processing libraries, such as OpenCV, consists of performing whole frame operations in series. This leads to high usage of DDR memory since frames need to be read from and written to the main memory between operations. Even though platforms with large CPU cache sizes can support this model, it is not suitable for embedded system where memory size as well as power usage are limiting factors. Therefore, a different approach is followed by Myriad2 which aims to maximise the usage of available resources.The model used by SIPP framework consists of a graph of connected filters. Image data is read from the DDR to the CMX memory via DMA filters and after the processing the result is written back to the DDR. The processing is achieved by streaming data from one filter to the other in a scanline-by-scanline basis. The buffers used to hold the processed lines are located in the low-latency CMX memory, thus avoiding the need for DDR accesses except for those in the first and the last stage of the graph. Hence, benefits are gained by the SIPP framework regarding the performance as well as the power drain of the developed applications.

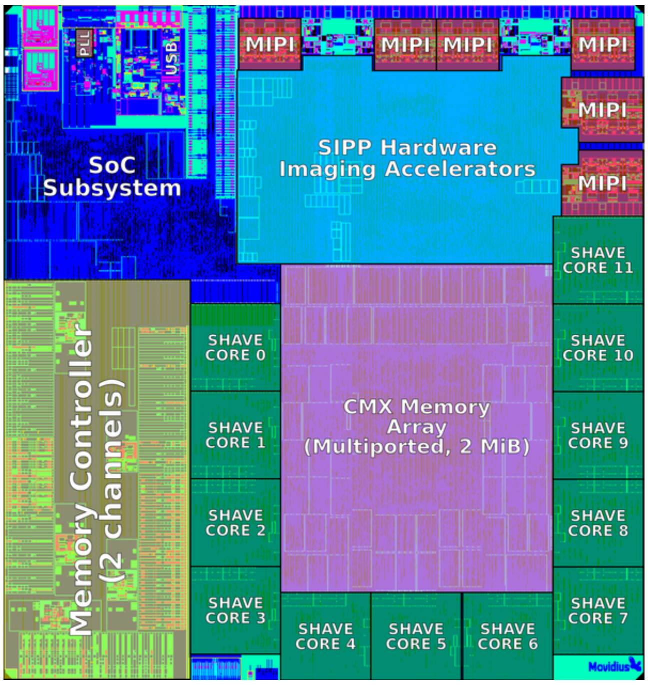


Figure 2.6: ma2450 die [3]

2.1.3 Myriad X

Myriad X was introduced in 2017 and is the successor to the Myriad 2[4]. It is the first VPU that was introduced under Intel's brand and it is more of an evolution step than a radical redesign. All the changes that were introduced is the result of transitioning to TSMC's 16nm FFC process. The extra space was utilised to add more functional units and increase the SoC's Frequency by 100Mhz to 700Mhz. More specifically, it is the first Intel's VPU to feature a dedicated hardware accelerator for deep neural network inference, i.e., the Neural Compute Engine(NCE). The chip's performance as a dedicated neural network accelerator is 1 TOPS for real-world applications. The SoC integrates 2 LEON4s and 16 SHAVEs and provides 512MB LPDDR4 DDR and 2.5MB CMX (Scratchpad) memories.

A comparisson of the two VPUs of the Myriad Family is presented on table 2.2

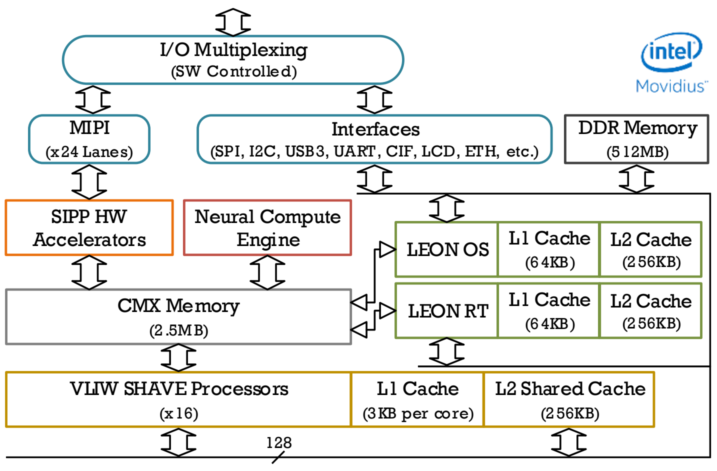


Figure 2.7: Myriad X block architecture [4]

Table 2.2: Comparisson between the Myriad Platforms.

Movidius Myriad Family VPUs		
	Myriad 2	**Myriad X**
Compute Capacity	>1 TOPS	>4 TOPS
Vector Processors	12x SHAVE Processors	16x SHAVE Processors
CPUs	2x LEON4 cores (RISC; SPARC V8)	2x LEON4 cores (RISC; SPARC V8)
On-chip Accelerators	∼20 image/vision processing accelerators	20+ image/vision processing accelerators Neural Compute Engine (DNN accelerator)
Neural Network Capability	1st Gen DNN Support (Up to 100 GFLOPS)	Neural Compute Engine (Up to 1 TOPS)
On-chip Memory and Bandwidth	2 MB (400GB/sec)	2.5 MB (450GB/sec)
DRAM Support	Max: 8Gb LPDDR2 (533MHz, 32-bit) LPDDR3 (933MHz, 32-bit)	Max: 16Gb LPDDR4 (1600MHz, 32-bit)
DRAM Configurations	1Gbit LPDDR2 (MA215X) 4Gbit LPDDR3 (MA245X)	No in-package memory (MA2085) 4Gbit LPDDR4 (MA2485)
Encoder/Codec	VGA, 720p, 1080p, H.264 (software encoder)	M/JPEG 4K at 60Hz encoder H.264/H.265 4K at 30Hz encoder
Key Interfaces	12x MIPI lanes (DPHY 1.1) USB 3 SPI I2S SD 1GbE	16x MIPI lanes (PHY 1.2) USB 3.1 Quad SPI I2S 2x SD 10GbE PCIe 3.0
Process	28nm HPC+/HPC/HPM (TSMC)	16nm FFC (TSMC)

2.1.4 GAP application processors: GAP8

Intel Movidius is not the sole provider of VPU's, GreenWave Technologies, a spin-off of from the PULP (Parallel Ultra-Low-Power Processing Platform) project [40] has introduced the *GAP8* AI accelerator [5]. It is an IoT application processor that enables massive deployment of low-cost, battery operated intelligent devices that capture, analyse, classify and act on fusion of rich data sources such images, sounds, radar signatures and vibrations.

The architecture is similar to that of the Myriad Family as presented on figure 2.8. The most important aspects of the SoC are:

- A compute cluster of 8 cores.
- A Convolutional Neural Network accelerator (HWCE)
- A fabric controller (FC) core for control, communications and security functions
- A series of highly autonomous smart I/O peripherals for connection to cameras, microphones and other capture and control devices.

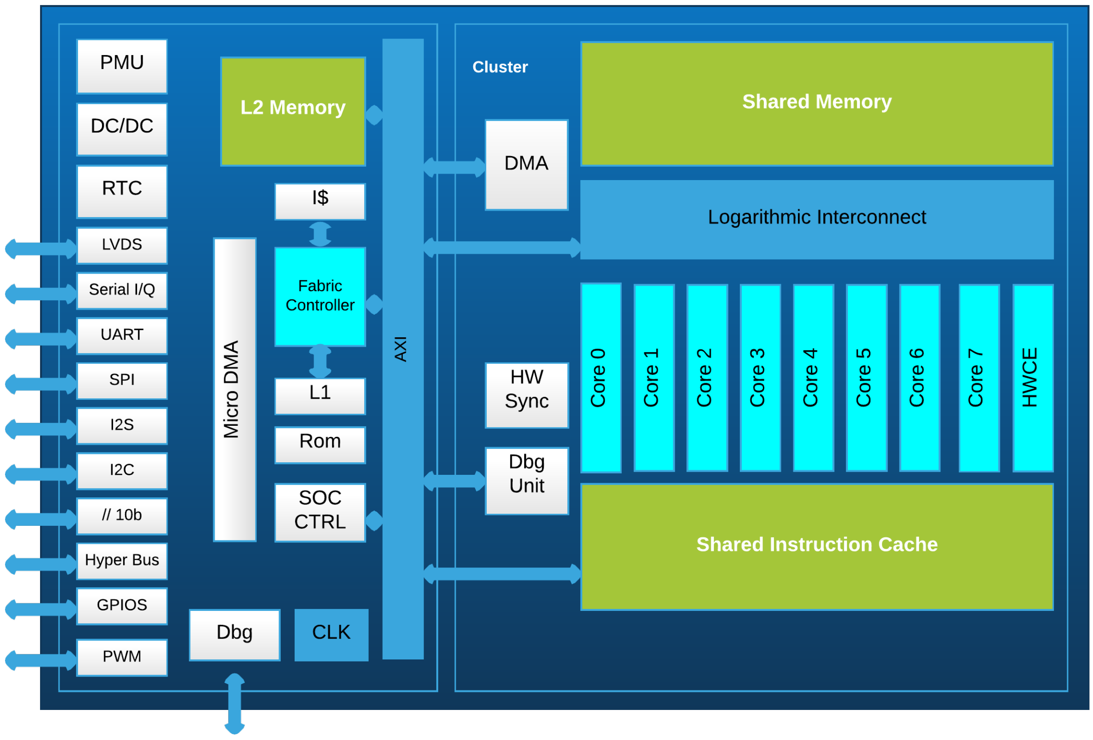


Figure 2.8: GAP8 architecture [5]

All 9 cores support the same rich extension of the RISC-V Instruction Set Architecture (ISA). The PULP open-source platform, provides the foundation for GAP8. This gives GAP8 a solid heritage based on several generations of test chips, a vibrant community and a full tool chain to support software development for devices, which enables fast time to market for integrators. The heterogeneity this time, does not orginate from different ISA & architectures, but from different extensions and implementation of the same RISC-V ISA.

All cores and peripherals are power switchable and voltage and frequency adjustable on demand. DC/DC regulators and clock generators with ultra fast reconfiguration times are integrated. This allows GAP8 to adapt extremely quickly to the processing/ energy requirements of a running application. All elements share access to a L2 memory area. The cluster cores and HWCE share access to a L1 memory area and instruction cache. Multiple DMA units allow autonomous, fast, low power transfers between memory areas in parallel with computation. A memory protection unit is included to allow secured execution of applications on the fabric controller.

Regarding general performance characteristics, GAP8 promises:

- Up to 250 MHz (FC) 175 MHz (Cluster) internal clock
- 8 GOPS at a few tens of mWs
- 5x5 convolution 16 bit-fixed point in one cycle
- FC delivers 200 MOPS at 10mW @1.2V/250MHz and 4mW @1.0/150MHz.

In terms of of AI and vision performance, the tiny darknet is executed at 0.8fps at 85mW, QVGA Face Detection at 0.4mW avg per fps and for autonomous drone navigation (DroNet [41]) 15fps using just $84mW$ is achieved.

Overal when comparing GAP8, to the Myriad Family of processors, it offers an order of magnitude less performance but it does in a tenth of the Myriad's total power evelop, so the total performance per watt is comparable. GAP8, however is interesting, due to the multi-ISA heterogeinety. This paves the way for future architectures that deploy customised cores with targeted ISA extensions depending the application domain, maximising the efficiency, while requiring a small power budget [42, 43]

2.2 SWFrameworks: There is plenty of room at the top

2.2.1 Frameworks

OpenMP

OpenMP [13] is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran on many systems, instruction-set architectures and operating systems, It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

In its essence it is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

A significant limitation though for VPUs is the requirement for direct compiler support, which as of the time of this thesis is not available. However, OpenMP has been used for other embedded systems [44], FPGAs and GPGPUs, thus in future VPUs might be supported.

OmpSS

OmpSS [45] is a programming model that aims to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity (devices like GPUs, FPGAs). However, it can also be understood as new directives extending other accelerator-based APIs like CUDA or OpenCL.

Asynchronous parallelism is enabled in OmpSs by the use of data dependencies between the different tasks of the program. To support heterogeneity, a new construct is introduced: the target construct. The goal of this construct is to specify that a given element can be run in a set of devices. The target construct can be applied to either a task construct, which means that the task can be executed on a device, or a function definition, which means that this function has to be present in the device code.

Many heterogeneous architectures are supported including x86, Nvidia GPUs, ARM and Mali as well as FPGAs [46]. Despite not currently supported, the Mercurium compiler in which OmpSS is based upon, could be an attractive option for integration with the VPUs.

MPI

The Message Passing Interface (MPI) is a standardised and portable message-passing standard that function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.

The MPI interface is meant to provide essential virtual topology, synchronisation, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. MPI programs always work with processes.

MPI is not very popular, in heterogeneous devices, due to incompatibilities of data representation and interoperability of differing implementations of the message passing layer.

OpenCL

OpenCL [47] is a standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms. OpenCL greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories including professional creative tools, scientific and medical software, vision processing, and neural network training and inferencing.

The Myriad Family of VPUs does not directly support OpenCL, some functionality though is supported via customised vendor libraries, that offer similar API. In addition OpenCL support has been recently added for custom layers in OpenVINO, as described later.

OpenVINO

OpenVINO™ toolkit [15] is a comprehensive toolkit developed by Intel, for quickly developing applications and solutions on Intel hardware, ranging from CPUs to FPGAs and VPUs.It accelerates applications with high-performance, AI and deep learning inference deployed from edge to cloud.

It uses a model Optimiser and compiler that inputs a network from popular frameworks like TensorFlow [48] and Pytorch [49] and then produces an intermediate representation of it. This implementation is then optimised depending on the target platform and is executed via the respective plugin.

The main limitation of this solution, is that only AI and Vision application are supported, despite the fact that in recent version primitive development of OpenCL based kernels is added. Perhaps by exploiting this functionality, more complex applications could be programmed using this framework.

OneAPI

OneAPI [6] is a very promising solution which aims to offer a Unified, Standards-Based Programming Model that will eventually support all kind of heterogeneous architectures including CPUs, GPUs, FPGAs, VPUs and other specialised accelerators.

Intel oneAPI products will deliver the tools needed to deploy applications and solutions across these architectures. Its set of complementary toolkits—a base kit and specialty add-ons simplify programming and help developers improve efficiency and innovation. It is based on the DPC++ (data parallel C++) language.

In the current beta state, it ony supports CPUs and some GPUs but, when completed it will create a common developer experience across compute accelerator architectures, eliminating the need for developers to maintain separate code bases, multiple programming languages, and different tools and workflows for each architecture.

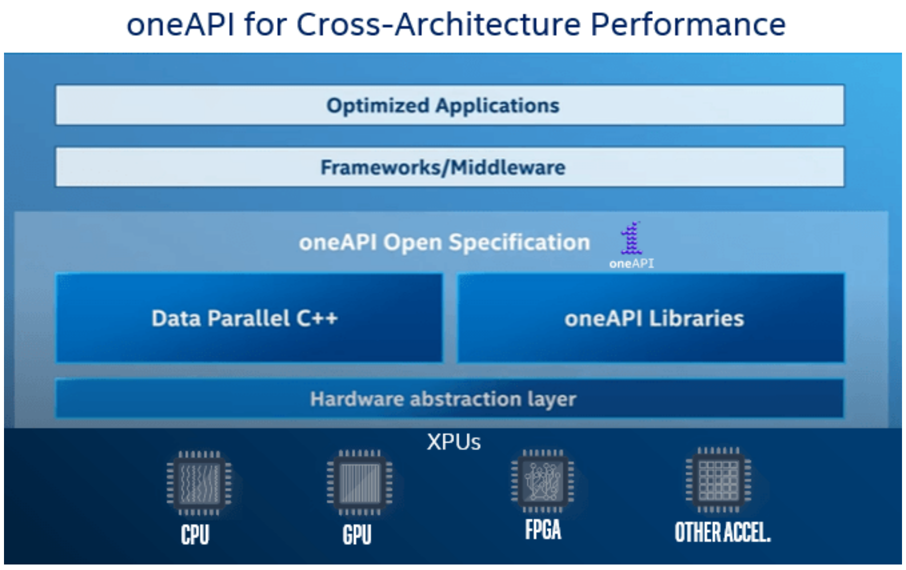


Figure 2.9: OneAPI architecture [6]

2.2.2 Operating Systems

Yocto Project

The Yocto Project [50] is a Linux Foundation collaborative open source project whose goal is to produce tools and processes that enable the creation of Linux distributions for embedded and IoT software that are independent of the underlying architecture of the embedded hardware.

The project offers different sized targets from "tiny" to fully featured images which are configurable and customisable by the end user. The project encourages interaction with upstream projects and has contributed heavily to OpenEmbedded-Core and BitBake as well as to numerous other projects, including the Linux kernel. The resulting images are typically useful in systems where embedded Linux would be used, these being single-use focused systems or systems without the usual screens/input devices associated with desktop Linux systems. As well as building Linux systems, there is also an ability to generate a toolchain for cross compilation and a software development kit (SDK) tailored to their own distribution, also referred to as the Application Developer Toolkit (ADT).

The Yocto project does not currently support VPU's but it will be added in future versions of VPUs.

Chapter 3

The ParalOS Framework

3.1 Introduction and Overview

ParalOS , aims to facilitate efficient programming and DSE on VPUs. It's ultimate goal is to abstract the Hardware spesific features of the platform while allowing quick deployment without preventing, low-level optimisation and the use of characteristics that define this category of devices. The framework integrates multiple modules to both improve the development efficiency as well as allow for platform-specific optimisations. These are:

- SPM manager providing dynamic allocations, without any spatial overhead, that is tightly integrated with the cache infrastructure
- A computational unit manager that provided an intuitive High Level API for faster application development, a dynamic scheduler and cache coherency mechanism
- IO standardisation and Board initialisation support.
- Decentralised inter-process communication (IPC) with MPI-like functionality.
- a feature-full Visual Profiler

ParalOS Software Stack

The overall software stack for developing Applications for VPUs is presented on Fig. 3.1. It is noted that *ParalOS* does not substitute or overwrite the Low Level drivers and APIs that are provided by the Vendors. Instead it complements them with the developed Low Level segments and then builds upon them using the High Level segment. The later can be thought as a middleware that provides a more intuitive and elegant foundation for faster, more efficient and less error-prone application development.

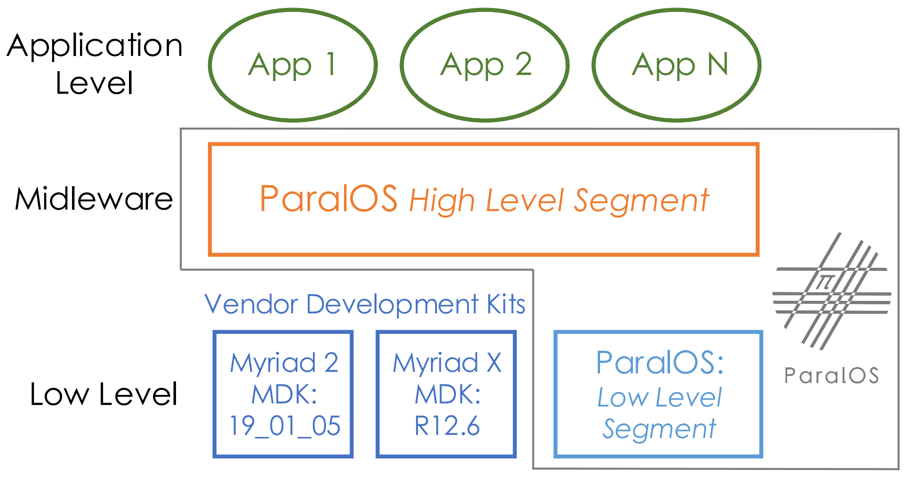


Figure 3.1: The VPU software stack and ParalOS

Programming paradigm

The programming paradigm of the proposed *ParalOS* framework, which is presented at high-level in Fig. 3.2, is based on dividing the development on the VPUs into two distinct segments. The first segment, namely *ParalOS: Low Level Segment* , includes the low-level implementation & optimisation of the functions that are designated for acceleration while the second segment, *ParalOS: High Level Segment* , includes the high-level parallelism and execution flow as well as the device configuration.

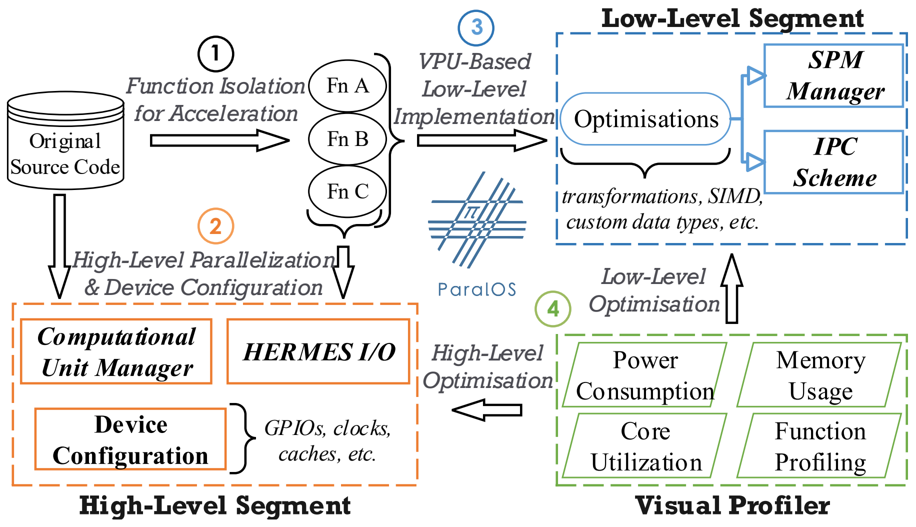


Figure 3.2: High-level architecture of the *ParalOS* framework.

The High-Level segment, which functions as *manager* and is executed on a GP core (e.g., Myriad's LRT), is responsible for orchestrating the Low-Level one. The latter is executed by the *workers*, e.g., the SHAVEs. Due to its modular design, *ParalOS* can be expanded to support more complicated workers, e.g., the SIPP filters, the NCE of MyriadX, etc. The other GP core, e.g., LOS, is purposely left unused, so it can be exploited for executing the non-parallelizable tasks with bigger cache and/or enabling any required RTOS feature.

3.2 Hermes: IO Communication Module

3.2.1 Role and Purpose

The purpose of this module is to provide a unified and standardised API for interfacing with the I/O peripherals of the target device as well as providing extensions for the necessary hardware configurations. This module is part of the *ParalOS: High Level Segment* and is executed on the manager cores. The industry has adopted various standards and APIs such as the CMSIS [16], that defines generic tool interfaces and consistent device support.

3.2.2 Structure

In order to create a cross-platform transparent API, the module which is developed on C++ consists of various submodules each corresponding to an IO peripheral and a main object that is coupled with the target board/IC that acts as a board support package. Each submodules consist of a platform agnostic interface and a platform specific implementation.

The overall idea of interfacing with the peripherals is derived from the *Everything is a File* [51] philosophy of UNIX, but differentiates from it by diving the peripherals into three classes, based on the typical message size and the communication direction. These classes are the following

- **SD** Small Size, Duplex Link. These are interfaces used mainly for commands and debug messages like I2C & UART.
- **LS** Large Size, Simplex Link. This class represents the high bandwidth IO ($> 1Gbps$) that serves as the primary data communication method. The LCD and Camera Interfaces are illustrative types of simplex data IO.
- **AD** Any Size, Duplex Link. This category is dedicated to the Ethernet and the overlaying network stack. TCP/IP messages can be used both as control and data bus and in addition they provide two-way communication

3.2.3 Implementation

As mentioned above the developed high level interface is hardware agnostic, but the low level implementation that was the target of this Thesis is the Myriad 2 IC and particularly the MV0212 [52] and EoT [53] board. In figure 3.3 the module's overall architecture is presented, while an in depth analysis of each submodule is available in the succeding sections.

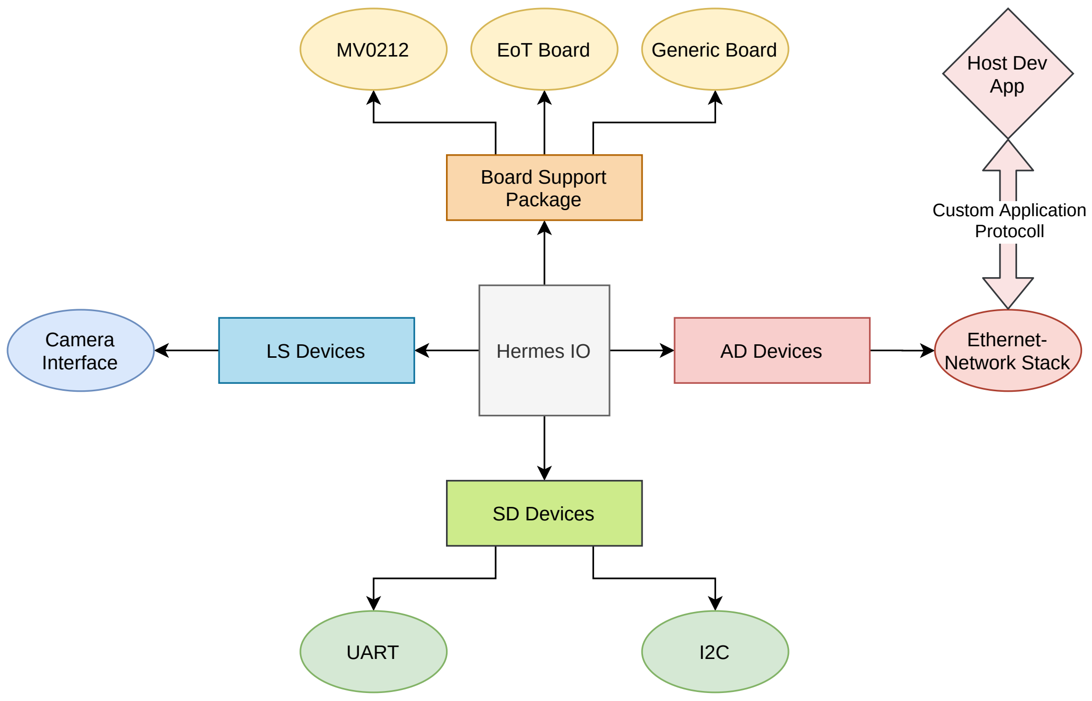


Figure 3.3: Hermes IO Architecture

Board Support Package

Modern SoCs support GPIO multiplexing [54], offering increased customisation and functionality. VPUs are no different [55], since the various boards have allocated the GPIOs in a different way. The configuration of the board is often a quite tedious process, so it has been simplified through a single API call
`int Hermes::Hermes.initialiseBoard(Hermes::BoardVersion::<Board Version>)`
Internally, the submodule performs the following operations:

1. Internal GPIO multiplexer configuration.

2. I2C Bus initialisation for the auxiliary ICs like the power supply unit.

Small Size, Duplex Link Devices

Two of these peripherals have been implemented on *Hermes* for the Myriad 2, UART and I2C. All the peripherals of this family follow the same API, that is described below. The *XXX* symbol denotes the interface used (UART or I2C)

1. `Hermes::XXXCommunicator(<configuration parameters>)`.
   This function initialises the peripheral and applies the provided configuration parameters, like the I2C peripheral or the system speed.

2. Next the `int Hermes::XXXCommunicator.connect(<configuration parameter>)` is called, and the conf. parameter can be the UART baud rate for example. During this call the last steps of the peripheral configurations are performed and depending

on the returned status code, the peripheral is ready to be used.
The following commands are responsible for the data communication and they are actually wrappers for the underlying vendor provided drivers.

3. `Hermes::XXXCommunicator.sendBytesRaw(void *buffer, int size)`
   The command transmits *size* number of bytes from th location pointed by the *buffer*.

4. `Hermes::XXXCommunicator.receiveBytesRaw(void *buffer, int* sizeRead, u32 maxSize)`
   The command receives at most *maxSize* number of bytes and stores them to the location, pointed by the *buffer*. The actual number of bytes read is returned to the location pointed by the *sizeRead*.

5. `Hermes::XXXCommunicator.insistReceive(void *buffer, u32 size)`
   Supplementary to the previous function this call blocks the program execution until exactly *size* number of bytes are received.

Large Size, Simplex Link Devices

This category consists of Camera interfaces (CIF), MIPI and LCD. Since the data handling requirements are very demanding, they usually employ other SoC subsystems such as the DMA engine for fast memory transactions. Moreover, the protocols for the referred devices, are much more complex than the ones described on the previous sections. As a result, the configuration process is less transparent as it will be described below for the implemented CIF interface.

The (CIF) is a video and still image capture input interface. Its' basic hardware image signal processing (ISP) pipeline allows it to interface with simple CMOS image sensors, with integrated ISP features, or to ISP chips. The main overview of the peripheral's architecture is presented in Fig.3.4.

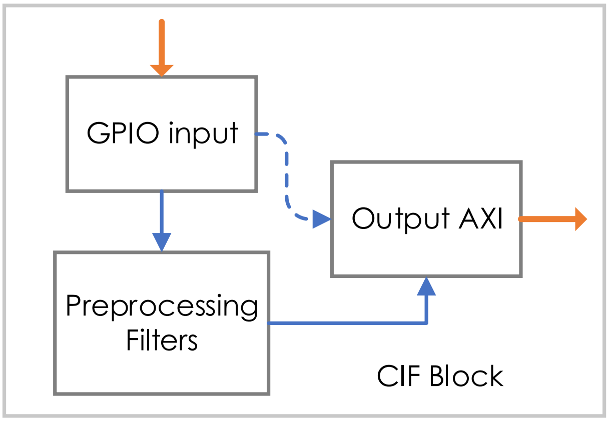


Figure 3.4: CIF block simplified architecture.

The image is acquired via the GPIO pins, then forwarded to a series of primitive input filters and then using the DMA engine, is forwarded to a specifed DDR location.

Configuring the device is not a trivial task and for this purpose two layers of drivers are offered by the MDK [39], namely the *CIF Low Level Driver* and *CamGeneric* module.

Hermes simplifies this process by dividing the configuration and the runtime phase. During the configuration phase the developers provides the required parameters using the listed API function Calls

1. `Hermes::CIFCommunicator(CamType camType, u8 **camBuffer, int numBuffers)`
   This function initialises the peripheral and provides the number and the memory location of the buffers where the image will be stored. Multiple buffers are used for IO masking and high level parallelisation of the the communication. The camType refers to the predefined camera configurations that are provided by *Hermes*.

2. `camErrorType Hermes::CIFCommunicator.config(I2CM_Device *pI2cHandle)`
   Many of the camera sensors need external configuration by the host device, e.g. Myriad, and usually this is facilitated by an I2C interface. When using such a camera this function automatically initialises the I2C peripheral which will be used later by the module.

3. `camErrorType Hermes::CIFCommunicator.config(GenericCamSpec *camSpec, CamUserSpec *userSpec)`
   With this function, the programmer provides the camera settings and the module is responsible to apply them. They are inserted by defining the above parameters structures. The *GenericCamSpec* refers to the camera specific settings like the image size, number of input channels and bit depth, while the *CamUserSpec* is used to determine the peripheral's, primitive filter parameters

The second phase is the runtime phase and most of the functions described below are self explanatory.

1. `camErrorType Hermes::CIFCommunicator.start()`
   Starts the peripherals and accepts incoming images

2. `camErrorType Hermes::CIFCommunicator.stop()`
   Stops the peripheral and no new images can be received.

3. `camErrorType Hermes::CIFCommunicator.standBy(camStatus_type standbyType)`
   Sets the peripheral into standBy mode, where it does not receive new images, but it is readily available.

4. `camErrorType Hermes::CIFCommunicator.wakeUp()`
   change the mode from standby to accepting new images.

Finally, the module provides some more information regarding the received images, i.e. the `newFrameFlag`, which informs the system that a frame was received and the `newCamFrameCtr`, that keeps track of the number of the received frames.

Any Size, Duplex Link

This category represents the Ethernet and Network stack. Due to the versatility and widespread adoption of network applications, support for TCP/IP communication is essential. Managing this Netork stack, however, is not a simple task and thus requires HW support (for PHY or MAC) in addition to the SW one. As a result this is the only module of the *ParalOS* that requires external dependencies and particularly an underlying Operating System like RTOS, which will provide a *BSD sockets* API [56]. In the Myriad case this is provided by the RTEMS RTOS, which is executed on LOS.

The *Hermes* component provides a simplified High Level API that utilises the underlying OS APIs to provide a more intuitive and tailored for VPU applications set of routines. The most important of the are the following:

1. `Hermes::EthernetCommunicator(<parameterList>)`
   This function initialises the Ethernet Communicator based on the parameter list. The parameters that are specified are the device's IP Address and subnet and port, the address of the default gateway and the input buffer size.

2. `Hermes::EthernetCommunicator.connect())`
   The systems awaits for a client to be connected to the Device.

3. `Hermes::EthernetCommunicator.sendBytes(int size, dataType_t data_type, void *buffer)`
   This function uses the custom application protocol to send *size* number of elements, that each element is of *data_type* type which are stored in the *buffer* memory address.

4. `Hermes::EthernetCommunicator.receiveBytes(void *buffer)`
   This method call receives the data sent to the device, using the custom application protocol and stores them to the specified *buffer*.

Similar functions for exchange of raw data without the application protocol (that is described below) are available, as well as auxiliary methods to modify the initial configuration parameters.

Hermes Ethernet protocol

The **Hermes Ethernet protocol** is a lightweight protocol, that provides minimal encapsulation overhead and greatly simplifies the data exchange process. It is encapsulated in a TCP/IP package.
A diagram showcasing the protocol is listed below. As shown, the protocol consists three three distinct parts. The prelude, the payload and the checksum, that are described below

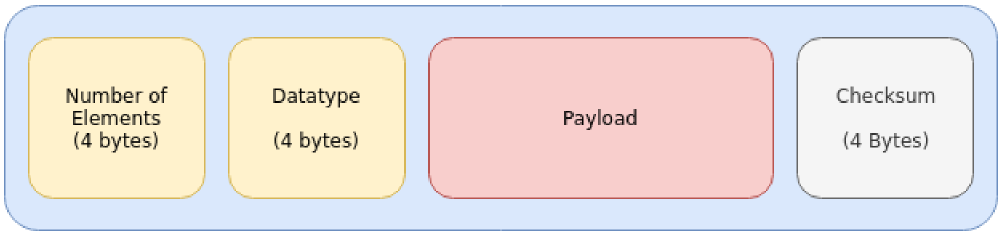


Figure 3.5: Hermes Ethernet Protocol

Prelude

The prelude is 8 bytes long in total. The first four bytes hold an int, that determines the payload's number of elements. The final four, encode an integer which determines the datatype of the elements. The following datatypes are currently supported, with a planned extension to accommodate custom defined structs, or class objects.

- *unsigned char* (1 byte)
- *unsigned half* (2 bytes)
- *unsigned int* (4 bytes)
- *unsigned long long int* (8 bytes)
- *8bit int* (1 byte)
- *half int* (2 bytes)
- *int* (4 bytes)
- *long long int* (8 bytes)
- *half float* (2 bytes)
- *float* (4 bytes)
- *double* (8 bytes)
- *char* (1 byte)
- *string* A null terminated char array

Payload

The payload consists of the actual message in byte format.

Checksum

A checksum mechanism is used as a final safe guard, to guarantee the message's integrity. Currently a simple summing is used, where the total bytes sent/received are summed and compared to the expected number of bytes. A more robust algorithm could be potentially used, but further testing required in order to make sure the additional computation overhead is kept to minimum, since this is a software decode protocol.

PC driver

A PC driver companion is also developed in python, to reduce the PC Client's development time and accelerate development and debugging. The API is similar to the one written for the Myriad. The module can be install as an external package

Limitations

There are currently some limitations, some of which can be solved after more development and some other, are dependent on the RTEMS OS.

- **Auto Network Configurations**. As mentioned earlier the ip address is configured manually. This could potentially lead to confusion, when the Myriad is connected to a network where there is no easy way to determine which addresses are already in use. Unfortunately, the RTEMS BOOTP procedure does not seem to function properly.

- **Myriad Outbound Throughput**. The RTEMS defines some buffers used to temporally store the incoming/ outgoing tcp messages before communicating with the Network. When continuous calls to BSD's sockets send function are requested, the following unexpected behaviour is observed. The OS informs the user, that all the bytes are sent, but actually, they are just sent to the intermediate buffers and not to the Network. A workaround is implemented, by reducing the buffer's size to a very small value. This forces to fragment the output message into more TCP packets, whose payload is very short. The downside of this fix is, the very low *payload to total_bytes_ratio* for the Myriad's upload.

- **User Defined Datatypes**. Currently user defined datatypes are not supported, but this could be fixed in a future version.

- **Half Float**. Half float is not supported on the PC driver's side, due to encoding issues.

- **More Clients and Client Usage** The communicator can currently act as a server to a single client. More connections and client usage, could be added, but it is out of scope for this thesis.

3.3 SMPI: Inter Processor Communication Module

3.3.1 Role and Purpose

The majority of the algorithms when parallelised, require data exchanges between the subtasks. The target platforms however, were designed to optimally execute embarrassingly parallel, or very limited communication-wise, algorithms, thus offering basic options for interprocessor communication (IPC)

In the Myriad case, sharing data between the VPU's compute units is supported via different methods such as the DMA engine or small register pipes. However, they are applicable to either very large data blocks, e.g., more than 2KB, or very small, e.g., < 128 bytes respectively. As a result a module for addressing the gap between the aforementioned methods is developed, that belongs to the *ParalOS: Low Level Segment* and is executed on the worker cores.

3.3.2 Structure

The module's API is based on the popular and well established MPI [14] and offers two main yet overlapping functionalities a) Message Passing and b) Synchronisation. The algorithms and techniques will be discussed in the following sections, but first the architecture of the module will be described. The most important choice from the architecutre prespective, is the implementation of a decentralised system, meaning that the majority of functions does not require a central management mechanism. A circular buffer is assigned to each and every worker that resides in the Scratchpad memory and is used as a receiver for the incoming messages. This buffer is implemented using a two counter technique. In addition to the SPM, the hardware provided mutexes are used form primitive synchronisation and are expanded with the *Virtual Mutex* (Sec. 3.3.4) concept. An overview of this architecture is presented on Fig 3.6

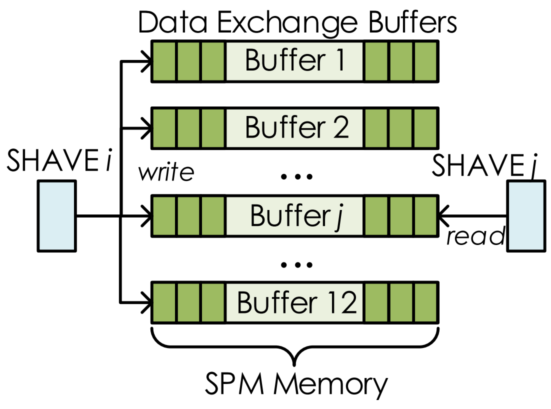


Figure 3.6: SMPI Architecture Overview

3.3.3 Management and Control Scheme

The configuration data for the SMPI Module are stored in the Shared SPM segments, because of the low memory footprint of the configuration object and the fast uncached access time on runtime. This object holds the data prepented on Lst:3.1:

```
struct SMPI_config_t {
    // Pointer to the array of Buffer Pointers
    // one buffer Struct per Workermlab_2 = [0.8500, 0.3250, 0.0980]

    struct SMPI_buffer_t **buffers {
        // the receiver circular buffer
        volatile u8* circularBufferAddr;
        // counters for the buffer
        volatile u32 counters ...;
        // other auxiliary Data and Flags
        auxiliaryData...;
    };
    // Pointer to the array of Barriers
    SMPI_barrier_t **barrier{
        // a boolean array holding the workers assigned to the barrier
        u8* workersInBarrier;
        // other auxiliary Data and Flags as described late
        auxiliaryData...;
    };
    // One Vmutex Struct per VMutex described later
    struct VMutexes**;
    // One HWMutex Struct per HW Mutex described later
    struct HWMutexes**;
    SMPI_VMutex**
    // Size of each of the the reception Buffer
    int size;
    // Number of total Workers
    int noWorkers;
};
```

Listing 3.1: SMPI Configuration Object

3.3.4 Virtual Mutex

The mutexes provided by the hardware are as expected quite limited and are utilised by other subsystems and applications, thus prohibit the use of fine grained locking. An attractive workaround is the introduction of the Virtual Mutex concept. Instead of directly accessing the hardware mutexes, the module offers virtual/software mutexes, named VMutexes to the developer. All the defined VMutexes are inserted into a pool and using a component called mapper, on runtime, they are mapped to hardware ones. Theoretically infinite number of virtual mutexes can exist, but due to the contention and limited scalability the performance degrades. The overview of the architecture is depicted on Fig. 3.7

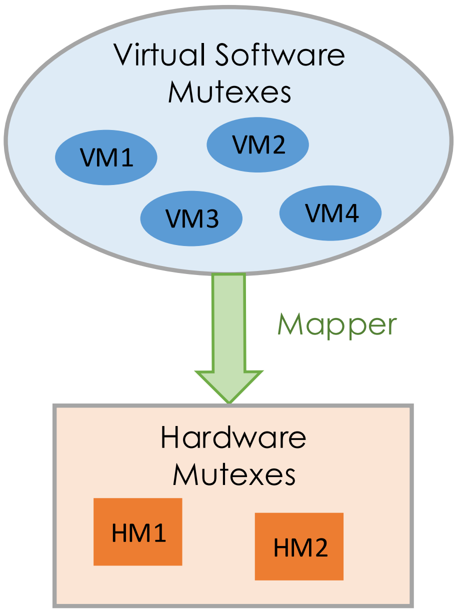


Figure 3.7: Virtual Mutexes Architecture

The number and id of the hardware mutexes available to the mapper, are defined during the initialisation phase and cannot be altered once the application starts executing. The hardware mutexes are identified using a simple integer that acts as the id. The HW mutexes are represented using the following struct (Lst:3.2):

```
struct HWMutex {
    id;      // The id of the hardware mutex
    swId;    // The id of the virtual mutex that this HW mutex
             // assigned to or HWM_FREE if not assigned to a mutex
};
```

Listing 3.2: HW Mutex represenation

Similarly the virtual mutexes are represented using the struct (Lst:3.3)

```
struct VMutex {
    id;      // The id of the hardware mutex
    hwId;    // The id of the HW mutex that this VMutex
             // assigned to or VMX_FREE if unlocked
};
```

Listing 3.3: Virtual Mutex represenation

The API support two functions `VMXLock(VMutexId)` & `VMXUnlock(VMutexId)`, that provide the locking utilites. When a call to Lock is performed, the algorithm initially checks if the VMutex is free by comparing the *swId* field with the *VMX_FREE* value and if locked, it halts the execution via spinlockicng. In the case that is free, the mapper component is then invoked, which iterates over all the available HW Mutexes for the first unused one. If such a HW one exists, the corresponding structs are updated and the HW one is locked. In no free HWMutex exists though, the execution is halted until one is freed.

The unlock operation is quite simple, as the mapping between the HW and virtual MXs is established. The HW Mutex is unlocked and then both virtual and HW mutexes are designated as free.

3.3.5 Barrier Synchronisation

Quite often parallel applications require all the assigned workers to reach the same point in order to continue the execution. This functionality is offered by the MPI using the *MPI_Barrier* call, which blocks until all processes in the communicator have reached this routine. Similarly, SMPI uses the method `SMPIBarrierSync(int id, SMPI_config_t config)`. In order to support multiple concurrently executing applications, up to 8 different barriers can be used simultaneously. The *SMPI_config_t* struct is the controller for the whole module and will be analysed later. The algorithm for the barrier method is presented in 3.4 and is derived from [17]

```
counter := total_no_procs;
loop: When Worker reaches the Barrier:
VMutexLock()
    counter := counter + 1;
VMutexUnlock()
if  counter = 0 {all procs reached the barrier} or
    resumeFlag = true {a proc has exited the barrier} then
    VMutexLock()
    if counter == 0 {first worker entering} then
        counter:= total_no_procs;
        nrPasses := nrPasses + 1;
        resumeFlag := True
    else if counter = 0 {last proc to exit} then
        reset nrPasses;
        reset resumeFlag;
    else {generic proc}
        nrPasses := nrPasses + 1;
    end
    VMutexUnlock()
    exit the barrier
end
goto loop
```

Listing 3.4: Barrier Synchronisation Implementation

In the previous algorithm the *nrPasses* indicate how many workers have reached the barrier, the *counter* variable tracks the number of workers that have not yet reached it and the *resumeFlag* is an indicator that more worker can exit it. The algorithm contains a couple of critical sections that are protected using the Virtual Mutexes.

3.3.6 Message Passing

The core of this module is the messaging passing functionality that enables the efficient development and execution of complex algorithms in the examined VPUs. There are two main function methods a **send**, which writes a message to another worker and a **receive** one, that a worker uses to read the incoming messages. As explained in the *Hermes Ethernet Communicator*, either raw data or a custom lightweight protocol can be used.

Custom Message Passing Protocol

The custom Message Passing Protocol consists of a packet that encapsulates the data and adds a 3 byte header that includes 1 byte for the sender's id and 2 bytes for the message size. Optionally a 4 byte Cyclic Redundancy Code for error detection can be inserted in the end to provide increased fault tolerance.

Sending Data

Sending Data to an other worker is a straightforward task. Initially the function checks whether the requested data fits in the destinations circular receive buffer. If so the data are copied to this buffer, otherwise an error code is returned. This procedure is protected by fain grain locking using the VMutex interface. In case that the custom protocol is used, the header is generated and is sent prior to the actual data, followed by the optional 4 byte CRC code.

Receiving Data

Receiving the data is a more complex process as additional functionalities are supported. There are four different receive functions, a) receive raw data, when no protocol is used and b) receive data with protocol c) receive from sender, d) receive all (both with and without protocol). The core of the receive function is always the same, the critical section is locked and when no protocol is utilised the user specifies the number of data to be read. When the protocol is used, the module self-deduces the number of bytes and the sender, as a result, the developer can choose to filter the incoming messages, by ignoring the received messages from other workers. Finally the "read all incoming data" function receives all the available messages in the worker's buffer.

3.4 Scratchpad Memory Management

3.4.1 Role and Purpose

The Scratchpad memory is one of the most important resources of the system that greatly contributes both in the reduction of execution time [57] and power [58]. There are two main issues with the SPM. First multiple applications that run sequentially or in parallel, require static allocation of data and secondly how is the actual allocation implemented. For this purpose a custom manual memory allocator for the SPM is provided, which is part of the *Paralos: Low Level Segment* and is executed on the worker cores. It is available in two flavours, i.e., only with static allocation, to be compatible with real-time and mission critical applications, and one with additional support for dynamic allocations.

3.4.2 Related Work & Background

The topic of SPMs is a well established research field with many interesting publications. Earlier works performed allocation for program code [59, 60], program data[61], or both [62]. Program code allocation needs to ensure that the program flow is unchanged and supports recursive SPM allocation schemes can also be classified as compile-time and runtime techniques based on the time at which SPM contents are decided. However all the previous publications, have in common that they manage cache as a software controlled cache in systems where implementing a cache coherency protocol, is very expensive in terms of power and space [63, 64]. Consequently, a new scheme is similar to the one proposed in [65], which manages both SPM and cache as a unified hybrid memory.

In terms of allocation techniques, multiple allocators are available today ranging from embedded systems, like the TLSF allocator [7], multi purpose like the Dough Lea allocator [66], and many/multi-core ones used in SotA data-centres like *jemalloc Slab* and *PHKmalloc* as described in [21]. The most prominent of those, for VPUs and embedded systems in gernealis the TLSF allocator.

3.4.3 Key Concepts

The key concepts of the SPM Manager are the following.

- **Data Only**. The Scratchpad memory is used only for allocating data and not for program code. This is deduced from internal comparisons between the available L1 instruction Cache, the available L2 Data cache and the size as well as the locality of the algorithms that are typically executed on VPUs. These comparisons show that in most cases that the SPM is used for programming code as well, the performance benefit is negligible, while in others worse execution time and energy consumption is observed, because the lack of space leads to more memory transactions.

- **Manual Allocation**. Since all the examined platforms offer a cache system hierarchy, automating the data allocations is suboptimal. Therefore, considering yet again the nature of algorithms, manual allocation is used instead. The developer, who has better insight of the execution flow of the application is responsible for deciding which data shall be placed on the SPM and those that will be accessed using the Cache Hierarchy.

- **Decentralised Management**. Similar to the previous SMPI module, there is no main controller for the whole SPM. Instead, each worker, that has preferential HW access to a specific segment of the SPM, is running a discrete manager and has exclusive access to the respective segment. This way the runtime memory operations can be executed concurrently, without requiring course grain locking.
- **Physical Separation of Data & Headers**. This is one of the most important research-wise design choices presented on this thesis. The control and configuration data on runtime are stored in a physically different memory, thus negating all memory overheads and achieving complete utilisation of the SPM.

Finally the contents of each discrete manager is clarified. The manager is no more than a collection of data structures that keeps track of the SPM memory usage and is located in the DRAM section, being accessible from both manager and worker cores.

3.4.4 Static Allocation

Regarding the static allocation implementation, it emulates the stack memory allocation, and as a result, no de/reallocations are allowed. Specifically, it integrates a simple pointer technique, similar to the FreeRTOS level-1 heap allocator [18], as shown in Lst:3.5. Initially all the available memory of each segment is reserved in a byte array which is managed by the worker's memory manager.

```
void *CMMMalloc(const u32 size, cmxMemMgrConfig_t *cmmConfig) {
    // 1. Check if static allocations are allowed.
    if(StaticAllocationsAllowed()){
        // 2. Check if there is free space available.
        if (freeSpaceAvailable()) {
            void* returnAddress =
                (void*) &(manager.byteArray[manager.stackPointer]);
                manager.stackPointer += size;
            return returnAddress;
        } else {
            // error code for Overflow
            manager.errorCode = CMM_OVERFLOW; return NULL;}
    } else
        // error code for invalid static allocation
        manager.errorCode =
        CMM_STATIC_ALLOCATION_AFTER_DYNAMIC_ALLOC; return NULL; }
```

Listing 3.5: Static Memory allocation

As shown the API offers a *pseudodynamic* allocation scheme where the `CMMMalloc` is a function that statically allocates data, but it does so on runtime. As a result, when employing multiple alogrithms that execute seqquentially and need to reserve space in the SPM, there is no contention. In the Myriad Case an alternative method is available by the vendor using the *Runtime instantiated applications*. This component packages every different algorithm that is executed on the workers, it is packaged and stored on the DDR. When an algorithm is called,the component is invoked and loads the data from the DDR to the SPM, inducing overhead. The proposed method, does not have this overhead and additionally provides a more user-friendly API.

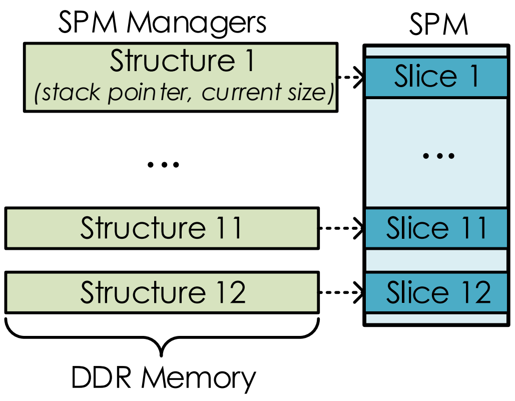


Figure 3.8: Static SPM Manager Architecture

In addition to the `CMMMalloc` function a variant of it is provided the `CMMMallocAligned` which has an additional argument the **alignmentSize**. Allocating aligned memory is important, especially for DMA transactions and vectorised accesses where alignment issues can cause stalls due to late read return. Moreover, the DMA agent, may be forced to perform more transactions, than intended due to physical limitation of the size of the memory line. Finally poorly aligned addresses, have an effect on the efficiency of the cache subsystem. Appropriately this method allocates memory addresses that are alligned to a boundary defined by the `alignmentSize` argument. The implementation is similar to the one described in Lst:3.5, with the differences being presented on Lst:3.6.

```
void *CMMMallocAligned(u32 alignmentSize, ...) {
    if(StaticAllocationsAllowed()){
        u32 resiude = manager.byteArray[stackPointer] % alignmentSize
        // 2. Check if there is free space available.
        if (freeSpaceAvailableWithResidue()) {
            void* returnAddress =
                (void*) &(manager.byteArray[manager.stackPointer]) +
                ((residue) ? ((u32)alignmentSize - residue) : 0));
            manager.stackPointer +=
                (size + ((residue) ? ((u32)alignmentSize - residue) : 0));
            return returnAddress;
        } else {
            // error code for Overflow
            manager.errorCode = CMM_OVERFLOW;
            return NULL; }
    } else
        // error code for invalid static allocation
        manager.errorCode =
        CMM_STATIC_ALLOCATION_AFTER_DYNAMIC_ALLOC;
        return NULL;
}
```

Listing 3.6: Static Alligned Memory allocation

The static allocation technique is also employed to allocate all the static structures (e.g., buffers) even if the dynamic allocator described below is used.

The final aspect of the previous codes that has yet to be explained is the configuration structure, but it will be presented in the end of the section.

3.4.5 Dynamic Memory Management: Introduction

Dynamic Allocation is a technique for allocating data dynamically during running, without a-priori knowing their size. In many critical and real-time application dynamic alloc is not allowed, or heavily restricted due to possible unexpected behaviour errors. Two allocators are presented that can be deployed on such systems the Two-Level Segregated Fit memory allocator (TLSF) [7] and a custom *Double Layer Bitfield Technique.*

3.4.6 Dynamic Memory Management: TLSF Allocator

TLSF is a general purpose dynamic memory allocator specifically designed to meet real-time requirements such as Bounded Response Time. The worst-case execution time (WCET) of memory allocation and deallocation has got to be known in advance and be independent of application data. TLSF has a constant cost $O(1)$. The overall architecture is presented in Fig.3.9

It implements a combination of segregated and bitmap-fits mechanisms. The use of bitmaps allow to implement fast, bounded-time mapping and searching functions. TLSF data structure can be represented as a two-dimension array. The first dimension splits free blocks in size-ranges a power of two apart from each other, so that first-level index i refers to free blocks of sizes in the range $[2^i, 2^{i+1}]$. The second dimension splits each first-level range linearly in a number of ranges of an equal width. The number of such ranges, $2^{\mathcal{L}}$, should not exceed the number of bits of the underlying architecture, so that a one-word bitmap Can represent the availability of free blocks in all the ranges. TLSF uses word-size bitmaps and processor bit instructions to find a suitable list in constant time. The range of sizes of the segregated lists has been chosen so that a mapping function can be used to locate the position of the segregated list given the block size, with no sequential or binary search. Also, ranges have been spread along the whole range of possible sizes in such a way that the relative width (the length of the range) of the range is similar for small blocks

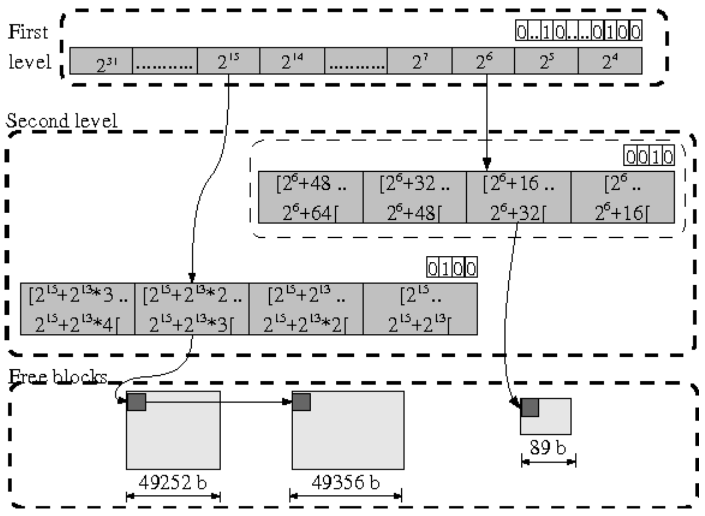


Figure 3.9: TLSF Structure [7]

than for large blocks. In other words, there are more lists used for smaller blocks than for larger blocks.

However, despite its potential it has the inherent problem of spatial overhead. In particular, TLSF has a minimum allocation block size of 4 bytes, due to the required storage for the header data. The contents of the headers are depicted in Fig.3.10

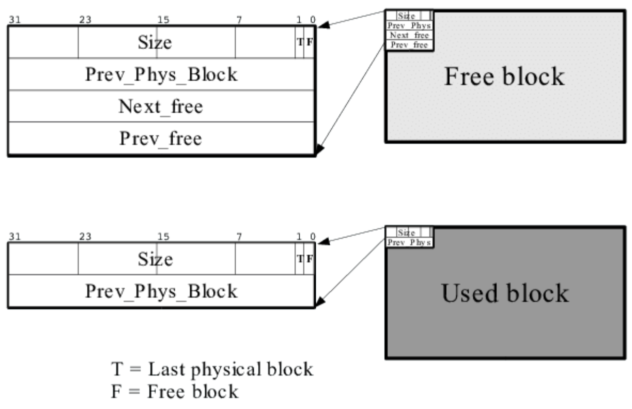


Figure 3.10: TLSF Headers [7]

3.4.7 Dynamic Memory Management: Proposed Double Layer Bitfield Technique

The proposed Double Layer Bitfield Technique aims to mitigate the induced overheads of the previous allocator, while maintaining a bounded and constant Worst Case Execution Time (WCET). The dynamic memory is performed after the static allocations have finished. This constraint is placed in order to ensure minimal memory fragmentation.

All the headers and control data are placed in optimised structures that reside in the DDR, while a bitfield array is used to directly map the SPM to the DDR. Consequently, any spatial overhead in the SPM is eliminated, and 100% utilisation is achieved. A more in depth analysis is provided in the following subsections.

Bitfield Direct Mapping

The remaining memory after the static allocation is mapped with programmable granularity to a bitfield array that is stored in the DRAM, an idea that was introduced, in a different context though, in [67]. Each bit of this map represents whether the associated byte of the SPM is free. Therefore, assuming maximum granularity of 1 byte of minimum allocation block size, this byte will require 1 bit in the DDR to be mapped. In the Myriad family of devices, where each segment of the SPM has a size of 128KB, 16KB of DDR memory will need to be reserved. If we multiply that with the number of workers, (16 in Myriad X), 256 KB will be used in total, or 0.5% of the DDR, quite insignificant in comparison. Obviously, since the granularity is programmable, this number could be potentially be reduced to facilitate applications with increased DDR memory requirements and bigger size SPM block allocations.

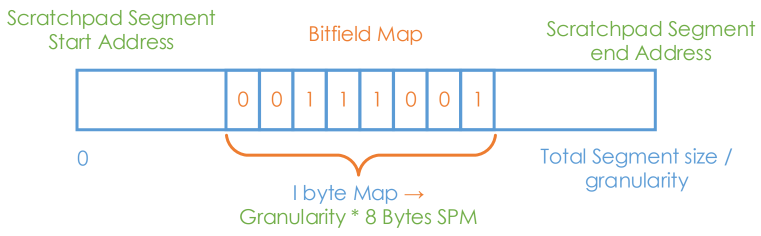

Scratchpad Segment Start Address
Bitfield Map
Scratchpad Segment end Address
0 0 1 1 1 0 0 1
0
Total Segment size / granularity
l byte Map →
Granularity * 8 Bytes SPM

Figure 3.11: Bitfield Mapping Representation

Headers & Data Structure

In addition to the memory map, the already allocated memory blocks need to be tracked, in order to know their size for the memory freeing operation. For this purpose a header system, assuming the role of a ledger is implemented. The headers have different composition depending on whether, they are allocated or free. If a header has not been assigned to an allocated block, it is considered free. When a header is allocated, it holds information regarding the block's start address and size. Each field is 4 bytes long, so overall the header has a size of 8 bytes.

All the available headers are pre-allocated in the DRAM during compilation for compliance purposes with mission critical and real time applications. The free headers are stored in a FIFO data structure, which is organised in a singly linked list (SLL) to guarantee $O(1)$ runtime performance. The allocated headers, on the other hand are stored in a Red-Black Tree (RBT), similar to the *jemalloc allocator* [19], where the key search index is the start address of the block's memory. Therefore, given N number of allocations, an operation cost of $O(\log N)$ is guaranteed, and since N has an upper bound due to the pre-allocation, the WCET is a-priori known and can be easily calculated. The headers are encapsulated within a header node structure. This structure, in addition to the header, has two more fields (Fig.3.12), the pointer-left/next-node, which is used to keep track of the nodes, the left child in the RBT and the next node in of the SLL, in the stack and the pointer-right, that points to the right child of the RBT and unused when on stack. As a result, in terms of space, the node requires 8 more bytes therefore, increasing the total size to $8 + 8 = 16$ Bytes. Assuming a total number of possible allocations of $10,000$, that require an RBT with a height of 17, the memory requirements for the headers is 16KB, which for 16 workers translates to 256KB. Overall the memory requirements for the memory manager is in the region of 512KB, or 1% of the total available DDR space.

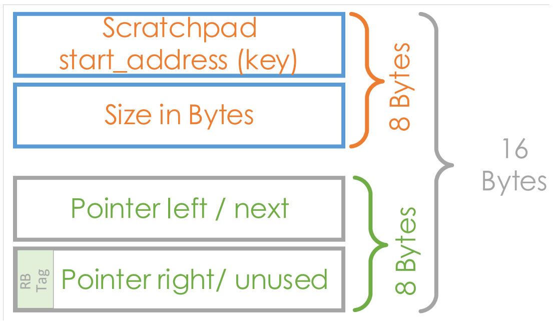


Figure 3.12: Dynamic SPM Headers

An important aspect of the performance of the model is the RBT implementation, which delivered better performance than other commonly used data, structures (e.g., Splay Trees) [20]. Implementation wise, it avoids the use of recursion, reducing potential stack overflow issues. Multiple calls to the same recursion function are a prime candidate for runtime overflow issues, therefore omitted. In addition the RBTs compared to regular binary trees, require one more bit of information, to determine whether the node is red or black. This bit can be masked in the last bit of the pointer-right field, assuming aligned addresses of the nodes. Moreover, the 16B size of the node is essential to be maintained, when the Cache line size is also taken into consideration. For the Myriad family of devices it is 16B for the L1$, thus maximising the cache efficiency, as all the headers and the bitfield which are stored in the DDR are accesed via the cache insfrastructure. Finally, due to the large extent of the implementation code no source or algorithms for the RBT is provided instead a similar yet quite customised approach was derived from [68].

Allocation Policy

Regarding the selected allocation policy, extensive research has already been conducted [21], nevertheless, no single best policy is found. The author tends to slide with [22], thus, a *first-fit* policy is selected, increasing runtime performance with the hazard of higher fragmentation. In detail, a 4 byte word is selected from the bitfield. The first available free bit is found and then consequently bits are checked until they amount to the requested block size. If the end of the word is reached, the search continues to the next block. If an allocated block, is found before adequate space, the search is reset. The algorithm terminates when enough space is found, or the whole bitfield is searched.

Malloc & Free Operations

When a worker core calls malloc, it accesses the bitfield map to check if and where the first valid available block is located. If such memory block exists, the respective bits are marked as reserved, and a header is popped from the free headers struct, and populated with the start address & block size. The selected header is inserted into the allocated headers tree struct. This is pesented on the following source code Lst:3.7

```
void* CMMDynCBFMalloc(const u32 size, cmxMemMgrConfig_t *cmmConfig) {
    // 1. Make sure that static allocation is finished.
    checkStaticAllocationFinished();
    // 2. Check that maximum number of allocations is not reached
    if(manager.numAllocs > MAX_NO_ALLOCATIONS) {
        manager.errorCode =
                CMM_DYN_MAX_NO_ALLOCATIONS_EXCEEDED;
        return NULL;
    }
    // 3. Find a block of at least <size> bytes free
    void* returnAddr =
        manager.bitfield.findSpace(size);
    if(returnAddr == NULL) {
        manager.errorCode =
            CMM_DYN_NO_SUITABLE_BLOCK_FOUND;
        return NULL;
    }
    // 4. Update the bitfield
    manager.bitfield.reserve(returnAddr, size);
    // 5. Allocate a new header
    headerNode_t* node = manager.freeStack.pop();
    // 6. Populate the header
    node->populate(returnAddr, size);
    // 7. Insert the header to the RBT allocated data struct.
    manager.allocatedRBT.insert(node);
    // 8. Increase the number of allocations.
    manager.numAllocs++;
    return returnAddr;
}
```

Listing 3.7: Proposed Dynamic Allocation Operation

Freeing a memory is the reversed procedure: Firstly, the header that holds the memory address is searched and removed from the tree, then the bitfield table is updated and finaly the header is pushed to the free headers struct, while its contents are cleared. A simplified algorithm is presented in Lst:3.8

```
void CMMDynCBFFree(void*memAddress, cmxMemMgrConfig_t *cmmConfig) {
    // 1. Find and remove the header that holds the block to be removed.
    headerNode_t* node = manager.allocatedRBT.remove(memAddress);
    // 2. If no block is found, an illegal address was passed. Exit.
    if(node == NULL) {
        code.errorCode =
            CMM_DYN_TREE_REMOVAL_FAILED;
        return;
    }
    // 3. Update the bitfield.
    manager.bitfield.free(memAddress, node->size);
    // 4. Push the header into the free stack structure
    manager.freeStack.push(node);
}
```

Listing 3.8: Proposed Dynamic Free Operation

3.4.8 SPM Manager Summary

Recapping, each worker has its own discrete manager instance responsible for the SPM segment that is assigned to the worker. The manager is just a ledger containing the necessary data to keep track of the SPM usage. The data are stored in a `cmxMemMgrConfig_t` object which resides in the DDR. The contents of the object as well as the functionality of the manager is presented visually in Fig.3.13.

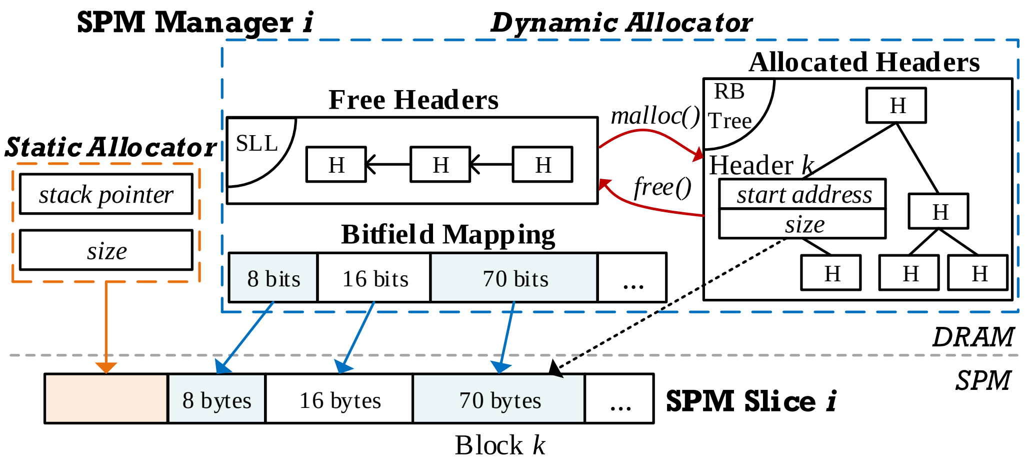


Figure 3.13: SPM Manager Overall Architecture

3.5 Computational Unit Manager

3.5.1 Role & Purpose

The *Computational Unit Manager* is part of the *ParalOS: High Level Segment* and is executed on the manager cores. It serves a triple purpose:

- **Developer Interaction & High Level API**. It allows the developer to design application in a graph based manner similar to the the *Thread Building Blocks* (TBB) [69] & *OpenMP* [13], but enriched with an intuitive yet powerfull constraint System.
- **Worker Interface and Stadarisation**. VPUs are highly heterogeneous Processors consisting of multiple different computational units. This module provides a standard interface that hides the inherent complexity of the system and handles trivial operations, in addition to the initialisation ofa various parameters like the cache subsystem.
- **Scheduling & Execution Handling**. The module when given an application which was described using the provided API, is able to perform static and dynamic scheduling during initialisation and runtime respectively, while also managing the seamless execution of it on runtime

These key functions wil be discussed in detail in the following sections.

3.5.2 High Level API

The High Level API is the primary way that the developer interacts with the *ParalOS* . The API is developed using an onion architecture. It consists of **Building Blocks** and **Constraints** (Fig 3.14). A collection of building blocks with the constraints that define the properties of the blocks as well the relationships with its environment, compose a layer of the architecture. Progressively higher layers range from a single task to applications. The building blocks and their constraints will be discussed, through an example application. Assume the following code where functions *A, C* & *D* are embarrassingly parallelisable, whereas function B can be parallelised with the support of runtime messaging.

```
.....
outputA = functionA()
outputB = functionB(outputA)
outputC = functionC(outputA)
outputD = functionD(outputB, outputC)
....
```

Listing 3.9: Example Programm to be designed using *ParalOS*

From the above code, the following characteristics can be deduced. Function *A* must be executed prior to the *B* and *C*, while the later two can be executed concurrently. Finally in order for *D* to be executed, *B* and *C* must have finished.

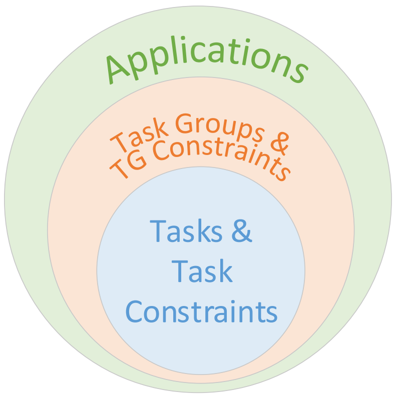


Figure 3.14: High Level API building block Layers

Tasks

The most primitive Building Block of the API is the **Task**. A single logical function(e.g. function A) is decomposed into smaller elements, each performing similar operations but on a usually different set of data. These elements are called tasks. Another perspective that a task can be thought of, is a simple POSIX thread, that is used by frameworks like OpenMP.

Each task has also a set of various types of constraints. These are:

- **Priority Constraints**. The priority constraints express the need for ordering between tasks. A priority constraint is a number in the range $[1, 10]$, with 1 being the highest priority and 10 the lowest one. Consequently, given N tasks that all are available for execution, first will be executed those who have priority closer to 1. This is particularly useful for applications with messaging constraints, when the developer knows a priori, which tasks need to be executed first, or in the case of content dependent functions, prioritizing the tasks, that have higher probability of more intensive workload.

- **Messaging Constraints**. Messaging constraints are used to describe the online and runtime instant communication needs, between tasks. Each constraint consists of three fields, i) a different task with which it will need to communicate, as well as a ii) tag field, which serves as an identifier for he messages and finally iii) a value that will be populated later by the dispatcher that shows the worker assigned to execute the dependent task.

The key implementation points will now be discussed. A task as mentioned above can be approached with a mindset similar to that of a thread. Consequently it has two attributes, a void pointer to the struct object that holds the function parameters and an entrypoint Tag. The entrypoint Tag and its necessity will be explained in section 3.5.4 but for now it can considered as the equivalent of the task's function entrypoint pointer. Additionally, the priority constraint is just an 8bit signed integer value and the messaging constraints are stored using C++ `std::vector` container from the STL. In addition, other auxiliary attributes, include but not limited to flags to determine the execution status of the task, a vector holding other tasks that have messaging constraints with this task etc. Finally, a pointer to the*Task Group* that this it belongs to is stored for performance reasons.

The Task is represented as a C++ Object, therefore offer a list of various methods to access interact with the object. This API is not presented, due to the great extend of it, but is available with the *ParalOS* Documentation.

Task Groups

The next layer of the building block Layers, is the **Task Groups** (TG). The Task Groups represent a single logical function (eg. function *A*) and consists of the tasks that this function is decomposed into. In this version of *ParalOS* for performance optimisations, the messaging constraints of the tasks are valid only when they refer to other tasks that belong to the same TG. Similar to other building blocks, it is decorated using the following constraints.

- **Priority Constraints**. Equivalently to the Tasks, TGs have a priority system with the same parameters, in order to determine the execution ordering between eligible TGs.
- **Logical Constraints**. Logical Constraints determine the logical causality between different TGs. In our example, functions *B,C*, shall be executed after *A* has finished execution, because their input parameters dependent on the results of the later. A TG can have multiple logical constraints, like the function *D*. In other words Logical Constraints are used to define the data dependencies between the TGs.
- **Worker Constraints**. When having a heterogeneous system like the VPUs there are many different compute units, that need to be mapped to the executing applications. An important yet difficult to answer question, in an automated execution system, is the following: *How fine grain should the mapping of the workers to the algorithms be ?*. In *ParalOS* the granularity is set to the Task Group Level, meaning that all the Tasks that belong to a TG will be executed by the same collection of workers. The characteristics of the required workers are defined via the Worker Constraints. It is noted that only the worker requirements are defined and no physical workers are assigned to it in this stage. The worker constraints include the following parameters:
  - **Worker Type**. In VPUs there are many available worker types ranging from specialised cores, to Hardware Filters etc. This field is used to determine the type of workers required. In this version of *ParalOS* for Myriad 2 & Myriad X, only *SHAVE* cores are supported as a worker type.

– **Number of Worker Instances**. This parameter specifies the minimum amount of number of workers that are needed for the TG's execution without the risk of deadlocks. This is quite important when having tasks with messaging constraints, in order to avoid scenarios when more tasks need to be run concurrently, than the available workers, which will lead to a never-ending program.

– **Priority**. Maintaining the same reasoning with the other priority constraints, a TG can have more than one Worker Constraint. For demonstration purposes, assume that the TG is responsible for executing a Convolution kernel on some data and the VPU, provides dedicated Hardware accelerators in addition to DSP cores. The developer using the priority fields, can elect the HW filter as the preferred computational unit, but if it is busy, instead of stalling, the execution is assigned to the secondary worker group. Another use case, where the priority system is beneficial, is the multi-generation platform support. Myriad X is equipped with a Neural Compute, whereas Myriad 2 is not, so by defining higher priority to the NCE, supported functions will be execute on NCE in the MyriadX and on the SHAVEs in the Myriad 2 case.

Regarding the implementation it is important to mention that the Task Group is just an abstract collection of tasks that is used in order to improve the efficiency of the scheduling algorithms and dispatching. As a result the Task Group objects consist of mainly data structures. In particular, C++ `std::list` containers are used to hold the pointers to other TG's as part of the logical constraints, the id's of the tasks that belong to this TG ,as well as pointer to these Task's objects.

An experienced reader is perhaps confused about the choice of the list container. An important caveat is that when allocating objects to a vector using the constructor in C++, if the size of vector exceeds a predefined value, the whole container will need to be relocated, thus any pointer that was referencing any element of that vector would be invalid. A potential workaround for it is the reservation of the container space beforehand, but still the total number of Tasks and Task Groups is not a-priory known for all the different applications and allocating extremely large blocks of potentially unused memory is not efficient, so the vector solution was dismissed.

With lists, however, another important problem arises, **Performance**. Iterating over lists and in general list operations, usually have a complexity of $O(n)$, with n being the total number of elements of the list. In order to compensate that, a mitigation mechanism was implemented, called **Double Representation Method**. After all the Task and Task Groups have created all the constraints are described during the static scheduling phase (Sec. 3.5.5), the system is fully aware of the total space requirements that will be needed, so storing pointers to elements of `std::vector` is possible. As a result these vectors are created so all the data are stored in two different containers, ie. in `std::vector` and `std::list`. This method, shows significant gains performance wise, because it allows a mixture of the best from both worlds. During the initialisation, `std::list` provide very fast 0(1) insertion and deletion, whereas during execution and runtime the `std::vector` allows for constant access time. In conclusion this tradeoff between the precious execution time and the more abundant DDR space is justified and thus implemented.

Along that data, the TGs also hold auxiliary data used during execution, like the number of tasks attached to this TG, finished flags, if all the attached tasks have started or they have finished execution, the TG's priority, a vector holding the data constraints, etc.

Finally the API is not presented due to its size, but it is noted, that the Task Group API is the most used API since that encapsulates most of the Task API as will become clear in Section 3.7

Applications

The Application layer is the top layer of the building blocks. Each application consists of Task Groups and their constraints. Contrary to the other building blocks, it does not have any constraints as it is an autonomous object. Multiple applications can be executed simultaneously.

Because of its autonomous nature, each application can be individually handled. In section 3.5.5 a detailed explanation of the scheduling technique is presented, it is mentioned though that the majority of information, regarding the task ordering is available during the initialisation phase, and is independent from the execution flow. As a result optimisations can be performed during initialisation or een offline. The later paves the way for the implementation of a **Packaging** system. With Packages the binary of the applications, at least the segment that is executed on the manager cores, can be exported during development and imported on production, thus significantly reducing the initialisation phase. Such a system is planned for *ParalOS* , but at the time of writing it is still in heavy experimental (alpha) phase, as there are pending issues regarding the memory placement and as result it is not yet officially supported.

The applications object like the Task Groups, adopts mainly a bookkeeping role, as it holds the task Group object using the previously explained double representation technique. Moreover various flags are also stored that indicate whether all the tasks that are part of this applications have been assigned, if the application is finished etc., that are used by the scheduler. Like the other building blocks applications are uniquely defined by their identification number.

The full API is once more not presented however some important functions are discussed below.

- **Top level API**. The fact that the Application layer is the highest abstraction layer, it encapsulates the majority of methods provided by the lower levels. As a result the developer mainly interacts with the Application API. Some of the most used methods provided are the following:

  - `Task *addTask(int Id, SMTAG entryPointTag, void *taskArguments)`
    This method creates a new task for this application, with the provided Id. The function to be executed is defined by the entrypoint Tag, while the associated taskArguments, are located in the memory address pointed by the taskArguments pointer. It returns a pointer to the created Task, which is useful for instantly accessing the Tasks API.
  - `TaskGroup *addTaskGroup(const int taskGroupId);`.
    Similarly to the above, this method created a new Task Group that belongs to this application, with the given id. The pointer to the created object is again returned.
  - `bool attachTaskToTaskGroup(const int taskId, const int taskGroupId)`
    This method attaches the previously created Task with the provided taskId, to the already constructed Task Group object with the respective id.

- `scheduleAndValidate()`. This method is called after the algorithm is described using the provided High Level API. The static scheduler (Sec.3.5.5) is invoked, that performs the offline optimisation. In addition the application is validated, by checking that all the constraints in all layers are valid, the id's of the objects are unique etc.

Finally, an additional reason that justifies the application level is **Namespacing**. The APIs make heavy use of identifiers to differentiate the various, building blocks. If the concept of applications was not implemented and many algorithms needed to be executed concurrently, their respective Tasks and Task Groups, need to have unique ids. This is particularly hard, or even impossible when considering that these algorithms may have been developed by a different group of programmers.

3.5.3 Worker Interface

Due to the multiple and different worker types an interface was developed that provides a unified API. In particular three core utilities must be supported : i) reset, ii) execution and iii) power management. When the developer wants to use a worker it has to initialise it. During initialisation a *handler* object is created which is responsible for managing the particular worker. The handler depending on the various system parameters prepares the underlying hardware for execution. Additionally, when a task is dispatched to the worker, the handler is responsible for resetting, executing as well as shutting it down for power preservation.

As mentioned this version of *ParalOS* , only supports SHAVEs. The initialisation/reset options include preparing and flushing the assigned L2 and L1 caches, set the stack and stack pointer and integrate the various module utilities. The execution phase, abstracts the trivial operations that are built on of the provided Low Level SHAVE Drivers.

3.5.4 System Configuration

The computational unit manager is also responsible for configuring aspects of the system that will be later managed by the module. These aspects are the *SMPI* and *SPM Manager* modules of the *ParalOS: Low Level Segment* , as well as the L2 cache subsystem for the workers. In addition, the Workers and Worker Groups are defined using this API and finally it provides methods for executing applications.

Module Initialisation

Initialisation of the *ParalOS* modules is pretty straightforward, as the initialisation is performed using the predefined available configurations. However, not all modules are required for all applications, ie. the SMPI module is only used when there are on-the-fly message passing requirements. The method `void configFrameworkOptions(SMPI_config_t smpi_config, cmxMemMgrConfig_t cmm_config)` based on which of the parameters are used informs the system about the low level modules used and initialises their configuration object.

Worker & Worker Group Creation

The target SoCs consist of multiple workers not all of which are used every time. Workers of the same type are organised into **Worker Groups**. The API call
`int addWorkerGroup( int workerGroupId, SMWGTYPE type, u32 workerMask)`
creates such a Worker Group. The group is assigned the provided id, the type parameter determines the Worker Type (currently only SHAVEs are supported) and the worker mask express which of the workers to be used when multiple instances are available. Depending on the bits that are set on the `workerMask`, the respective workers are used. The API provides addition ways to select workers, like providing the first and last of a series of worker, or selecting one worker instance per time. A worker can belong to more than one group at the same time. The first time that a worker is selected, the associated aforementioned handlers are initialised.

Worker L2 Cache configuration

VPUs have a number of different cache subsystems. One of the most difficult to manage is the worker's L2 cache subsystem as it is shared by multiple workers. Consequently, difficult to handle coherency issues, arise. In addition optimal configuration of the caches, can lead to significant performance improvements. Because of all these reasons, the L2$ subsystem along with the each worker's individual L1$ is managed by the module.

The developer specifies the preferred cache configuration using the following functions.

- `void configureCacheOptions( const SMOPT_CACHE cache,`
  `bool readOnly = false)`
  The first parameter specifies which of the L1 instruction, L1 Data and L2 cache will

be used. The caches that are not selected are powered off, disabled and bypassed. The `readOnly` parameter is an optimisation flag, that indicates if the cache will be used only by read Only data. If this is the case much more relaxed assumptions will be used during dispatching improving performance.

- `int configureCacheParameters( SMCP_GROUPS Dgroups, MCP_SIZES DsizePerGroup, SMCP_GROUPS Igroups, SMCP_SIZES IsizePerGroup)` This function is used to configure the L2 cache partitions. Before discussing its purpose, it is noted that there is a maximum number of partitions that can be created and partitions can have certain sizes. These limitations are caused by hardware and cannot be bypassed. The parameter list is divided into two pairs. The first pair is responsible for configuring the Data partitions while the second configures the instruction one. The first parameter of each pair, defines the number of partitions, while the second the individual size. The partition number can be any number in the range of $[0, 16]$, while the size can be any of the values: 16, 32, 64, 128 or 256. It is noted that Myriad 2 can have up to 8 partitions while Myriad X up tp 16. When calling this function, various validity checks are performed, to ensure that the total cache size and number of partitions are allowed. If so, the workers are automatically assigned to their partitions. Finally, the L2$ can be configured to be either data only or instruction only, by omitting the respective pair in the parameter list.

  These functions must be called after creating all the worker groups, since the respective cache attributes are populates based on the previous settings. Failure to do so will lead to undefined behaviour and will most propably cach coherency issues.

Application Execution API

The application execution API serves two main purposes. The first being the worker groups to applications assignment and the actual execution of them.

As mentioned previously when creating applications and especially the Task Groups, no actual workers are assigned, but instead worker constraints/requirements are defined. The actual mapping of workers is performed using the `bool assignWorkGroupToApplication( const int workGroupId, const int applicationId)`. The Worker Group with the referenced id is attached to the respective application. More than one Worker Group can be assigned to the same application and similarly a WG can be simultaneously be attached to many applications.

Application can be executed sequentially, in parallel or a combination of both, depending on the API call. When `int executeApplication( const int applicationId)` is used a single application is executed, whereas `int executeApplications( applicationIdV_t appIds)` will execute all the applications in the vector parameter concurrently. In the Lst:3.10 an example a more complex flow is presented.

```
.....
```

```
executeApplication(2)
executeApplications(applicationIdV_t::(0,1,3))
executeApplication(4)
....
```

Listing 3.10: Multiple Applications Example Flow

The application with id 2, is firstly executed and after it is finished, the apps 0, 1, 3 are started. When all 3 are finished, then app 4 is started.

Entrypoint Tags

In previous sections, the concept of entrypoint Tags was mentioned and described as the analogous of a thread entrypoint function. A valid question that arises is the following. *Why are the Tags used instead of pointers to the functions.* Answering that question is not a trivial task, as it requires an in-depth understanding of heterogeneous systems. Different types of workers, may be fundamentally different, supporting different ISAs or even lack one. In addition workers might not share the same address space. As a result, when the developer refers to a specific entrypoint function, it will couple this task to the specific worker, bypassing the whole framework stack. In order to compensate for this the function `void mapEntryPointTags( u32 *entrypoints, SMTAG entrypointTag)` is provided. The developer uses this function to map the different entrypoints for each worker to a logical *Entrypoint Tag.* Consequently, this mapping decouples the task from the workers and the system can freely assign it to the most optimal oneas explained in section 3.5.5.

3.5.5 Scheduling & Dispatcher

Introduction & Relative Work

Scheduling and task dispatching is perhaps the most important factor regarding the performance and efficiency of the SoC. Overall scheduling tries to answer the question: *Given a set of tasks, and a collection of workers, what is the most optimal mapping of the task to workers.* This could be described as *million dollar question*, when considering the increasingly more complex and heterogeneous systems that are introduced to the market.

Research wise, it is a very active field, the authors of [44] propose a modification of the popular OpenMP framework for embedded, though homogeneous, devices, while in [70] a novel method of SPM-based scheduling for many-core NoC SoCs is described. In [71], a full stack for offline reinforcement learning-based scheduling and mapping of tasks into processing elements is presented, however, it does not excel in content-dependent workloads. Also, the scheduler of [72] provides on-the-fly efficient mapping, but uses complex artificial neural networks for decision making.

For *ParalOS* a two level scheduler policy is implemented. Static and Dynamic, both of which are performed transparently via the computational unit manager. The decision behind this split is reinforced by the typical application nature and the computational resources available budget. Most of the ordering decisions for the tasks can be made during initialisation and offline, where more complex and intensive applications can be deployed, while the dynamic scheduling algorithm used on runtime, is a faster and more lightweight one.

Static Scheduling

The *static scheduling* is responsible for performing the serialised ordering of the tasks i.e., what is the optimal execution of ordering for a serial processor; and checking for possible violations. The validation checks include:

- **Incompatible Worker Types.** This is the first of two parts of the static scheduling that can not be executed offline and independently of the target VPU. This validation step checks whether the actual worker groups assigned to the application are compatible with all the Worker Type constraints of the Task Groups. An example scenario is if a specific hardware filter is requested, no alternative worker is defined, but this filter is not assigned to this application, or it is unavailable in this VPU.

- **Number of concurrent tasks exceed the number of available workers.** This is the second part that is execution dependent and it is especially important for tasks that have messaging constraints. If more tasks need to be executed concurrently, than the available number of workers, it will lead to a potential deadlock, as a task might stall until it receives a message from another one that is not running, halting as a result the program's execution.

- **Duplicate IDs**. All the other validations are independent of the system. The module assumes that all building blocks have unique identifications number. As a result if duplicates are detected, the application is characterised as invalid.

- **Tasks belonging to more than one Task Groups**. Every task must belong to a single Task Group, otherwise the logical constraints between different Task Groups are ambiguous.

- **Messaging constraints between task of different Task Groups**. Messaging constraints are difficult to handle and guarantee a deadlock-free execution, thus for performance reasons associated with the dynamic scheduler, messaging constraints must be contained within the same Task Group.

- **Circular Logical Constraints** Overall an application is represented as a directional graph. If a circle is detected on this graph, this indicates a circular dependency, which will send the program to an infinite loop.

For the serialised ordering, initially, all the TGs are sorted based on their priority. Next, they are sequentially accessed and individually checked for possible logical constraints with a TG placed later in the serialised order list. In this case, the latter one is placed before the checked TG and a priority check is performed to ensure that priority ordering is maintained. This process is iterated until no more changes are performed, or a circular dependency is detected. Finally, the tasks within a TG are sorted based on their priority. Both TGs and tasks are placed in FIFO queues. The algorithm is also presented on Lst:

```
void static_scheduling() {
    // 1. Sort the TaskGroups by priority.
    allTaskGroups->sort(priorityComparator);
    // 2. Sort the logical constraints by priority. This way all the
    // changes that will happen because of the logical constraints
    // will still satisfy the priority ordering constraint
    for tg : allTaskGroups {
        lcs = tg->getLogicalConstraints();
        lcs->sort(priorityComparator);
    }
    // 3. Check That the logical constraints of the TaskGroups are met.
    // Example: If taskGroups[3]=A with priority 2 and taskGroups[7]=B
    // with priority 3, but A is logically dependent upon B
    // then B must be executed prior to A.
    changesMade = true;
    while (changesMade) {
        changesMade = false;
        for(tg : allTaskGroups) {
            lcs = tg->getLogicalConstraints();
            testIndex = tg->position();
            for(lc : lcs) {
                lcIndex = lc->position();
                if (testId < lcIndex) {
                    allTaskGroups.splice(tg, allTaskGroups);
                    changesMade = true;
                } } } }
    // 4.  sort the tasks in each taskGroup based on priority
    for(tg : allTaskGroups){
        tasks= tg->getTasks();
        tasks->sort(priorityComparator);
    }
}
```

Listing 3.11: Static scheduling Algorithm

The time complexity, given N tasks and M TGs, is equal to $O(N^2 M \log(M + N))$, with an amortised cost closer to $O(N \log N)$. This cost, however, is not critical since the static scheduling is performed during initialisation or even offline, and is exported in a binary file for runtime use.

Dynamic Scheduling

The dynamic scheduling algorithm is executed on runtime and is responsible for the fluent task dispatching and aims to minimise the worker's idle time and task starvation.

An important design choice is the implementation of a *non-preemptive policy*, since the lack of native hardware support and the nature of typical VPU workloads do not justify the additional overhead of a preemptive scheduler.

Initially all the workers assigned to the application are initialised and the various application's execution flags are reset. Then for every running application, polling is continuously performed to identify w free workers from the WGs that are assigned to it. The polling technique is the only available method for determining the worker execution status, because of the lack of a direct worker-manager interrupt system. The scheduler then tries to dispatch to them up to w free tasks. The same technique is applied to all the applications, until every task is assigned. We note that when a worker executes a task with messaging constraints, all the workers of the same WG are locked from executing a task from a different TG, thus avoiding any potential deadlocks. The algorithm is presented on Fig.3.15

The complexity of this algorithm in both the space and time domain is $O(A \cdot M)$, where A is the number of parallel executing applications and M the number of TGs per application.

Dispatcher

A key module that is technically part of the dynamic scheduling is the task dispatcher, which performs all the required low-level operations, e.g., setting the stack register and resetting register files, before assigning tasks to workers. Also, a SW mechanism for providing system-wide cache coherency is implemented. In particular, when read-write data are accessed, the dispatcher automatically invalidates and flushes the caches associated with the selected workers w.r.t. configuration parameters provided by the developer, e.g., number/size of cache partitions. Moreover the dispatcher is responsible for updating the execution flags of the assigned tasks. Finally, for power saving purposes, workers that cannot be assigned to any remaining tasks, are automatically powered off. The dispatching algorithm is provided on Lst:3.12

```
void dispatcher(task, work) {
    // Step A Populate Messaging Buffers if the task has messaging constraints
    if(task->hasMessagingConst())
        task->populateMessWorkers(worker->id());
    // STEP B. Update worker's, task and TG's parameters runtime parameters
    task->updateParameters(); worker->updateParameters();
    task->getTG()->updateParameters();
    // Step C reset the workerand prepare it for execution
    worker->reset(); worker->prepare();
    // Step D Prepare caches
    worker->prepareL1Caches();  worker->prepareL2Caches();
    // Step E. Execute the task
    worker->start(task->entrypointTag(), task->arguments());}
```

Listing 3.12: Dispatching Algorithm

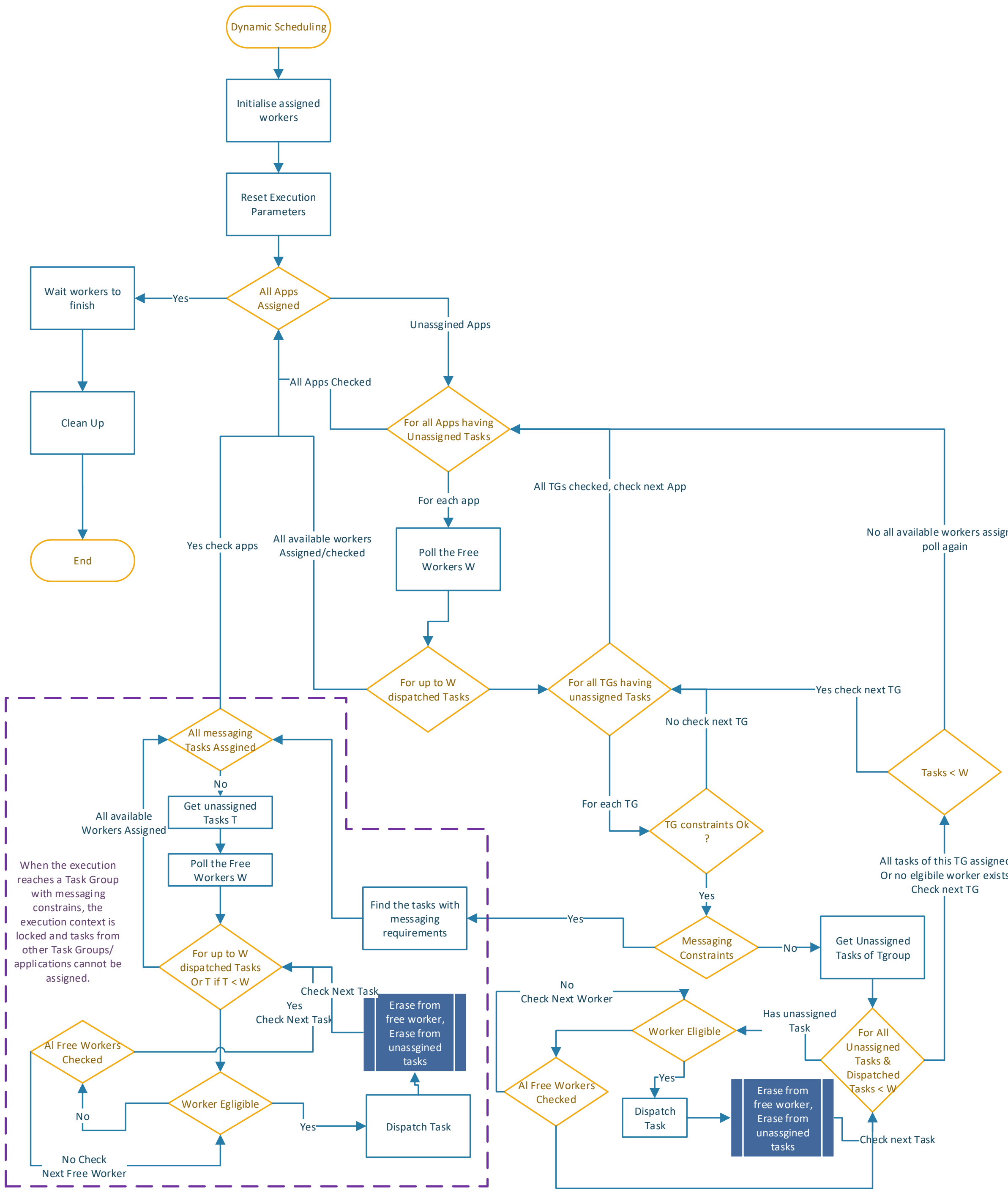

Dynamic Scheduling
Initialise assigned workers
Reset Execution Parameters
All Apps Assigned
Yes
Wait workers to finish
Clean Up
End
Unassgined Apps
All Apps Checked
For all Apps having Unassigned Tasks
For each app
Poll the Free Workers W
All TGs checked, check next App
No all available workers assigned poll again
Yes check apps
All available workers Assigned/checked
For up to W dispatched Tasks
For all TGs having unassigned Tasks
Yes check next TG
No check next TG
Tasks < W
For each TG
TG constraints Ok ?
All tasks of this TG assigned Or no elgibile worker exists Check next TG
Yes
Messaging Constraints
No
Get Unassigned Tasks of Tgroup
Find the tasks with messaging requirements
All messaging Tasks Assgined
No
Get unassigned Tasks T
Poll the Free Workers W
All available Workers Assigned
When the execution reaches a Task Group with messaging constrains, the execution context is locked and tasks from other Task Groups/ applications cannot be assigned.
For up to W dispatched Tasks Or T if T < W
Check Next Task
Yes Check Next Task
Erase from free worker, Erase from unassgined tasks
Al Free Workers Checked
Worker Egligible
No
Yes
Dispatch Task
No Check Next Free Worker
No Check Next Worker
Worker Eligible
Has unassigned Task
For All Unassigned Tasks & Dispatched Tasks < W
Al Free Workers Checked
Yes
Dispatch Task
Erase from free worker, Erase from unassgined tasks
Check next Task

Figure 3.15: Dynamic Scheduling Algorithm

Messaging Handling

This paragraph is dedicated to disambiguate the way that messaging constraints are handled. When the developer adds a messaging constraint into a task, this is added into a messaging vector and the Task Group is tagged as a messaging enabled task Group. During static scheduling it is checked if enough workers are available to satisfy all the communication needs. When the execution reaches such a task group, the TG is locked and no other task group can be executed until all the tasks of this TG are assigned. More efficient ways are perhaps available for coping with the deadlock hazards of non pre-emptive schedulers, but his is a topic of future work. When a task is dispatched the worker field of the messaging constraints is populated with the Id of the worker. From the worker perspective, the tags and worker Ids are available via accessing an object that holds all the messaging constraints. This object is passed as a pointer to the task's arguments and the memory address can be retrieved after the static scheduling is finished. Finally, it is noted that the messaging functionality is considered to be in *beta* phase, as more testing is required to guarantee correctness and there is plenty of room for optimisations regarding runtime performance.

3.5.6 Development Stages

The discussed *Computational Unit Manager* version is the last step of a gradual process of adding, improving and expanding the module's functionality. In the following paragraphs the intermediate versions will be discussed and the needs that pushed towards the improved versions.

Phase I: Task Pool

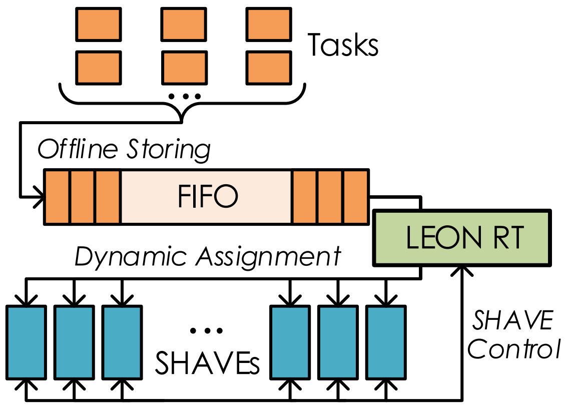


Figure 3.16: Phase I: Task Pool architecture

Initially the module was able to execute a single taskGroup, which did not have any constraints. The workflow was the following: The developer creates a task pool by splitting the workload into small (independent) tasks, and inserts them in a FIFO struct, as shown i nFig.3.16. Based on this FIFO policy, the proposed module assigns these tasks to workers at runtime, i.e., the first inserted task is assigned to the first available SHAVE. This method was extremely limiting as only very simple algorithms could be executed on the VPu which was impractical for real world applications.

Phase II: Multiple Task Pools

The next iteration of the module was the ability to execute multiple task groups sequentially. This paved the way for supporting the execution of more complex pipelines like Convolutional Neural Networks, where each layer was represented as a Task Pool. This was very efficient but yet again limiting as networks with parallel layers could not be represented. Moreover, since there is no way to determine priorities or data dependencies the developer had to spent considerably time manually configuring the execution order. Finally, the concept of abstraction layers was not implemented, thus porting the application to different platforms, was not a trivial task.

Phase III: Constraints & Graph API

This was the major redesign that lead to the API that is presented on this thesis. All te features previously discussed, were supported on this version, with the exceptions of messaging constraints. The new graph API that was based on a constraints system, proved to be a powerful method of expressing very complex algorithms. On the downside early implementations was a step backwards, performance wise as the overhead in early iteration reached value of up to 70%.

Phase IV: Double Representation & Optimisations

This version was focused purely in optimisations and ways to reduce the overhead and no extra functionality was added. The scheduling algorithms were improved and a smarter more fine grained way of managing the cache subsystems was introduced that took into consideration the read/write characteristics of the data. Finally the most significant performance gain was the introduction of the *Double Representation* Technique which mitigated the linear operation cost ($O(n)$) of the used data structures.

Phase V: Messaging Constraints

The last addition, in order to be able to execute the vast majority of applications is the support for scheduling of applications that are not embarrassingly parallel and require runtime communication. This functionality is still in beta phase, but it is currently fully functionally and passed the initial validation tests. However there is plenty of room for improvement and optimisations, that is one of the top priorities on the future work list.

In Figure 3.17 the relative performance of the computational unit manager normalised to the performance of Development Phase I is presented. As shown the Graph API (Phase

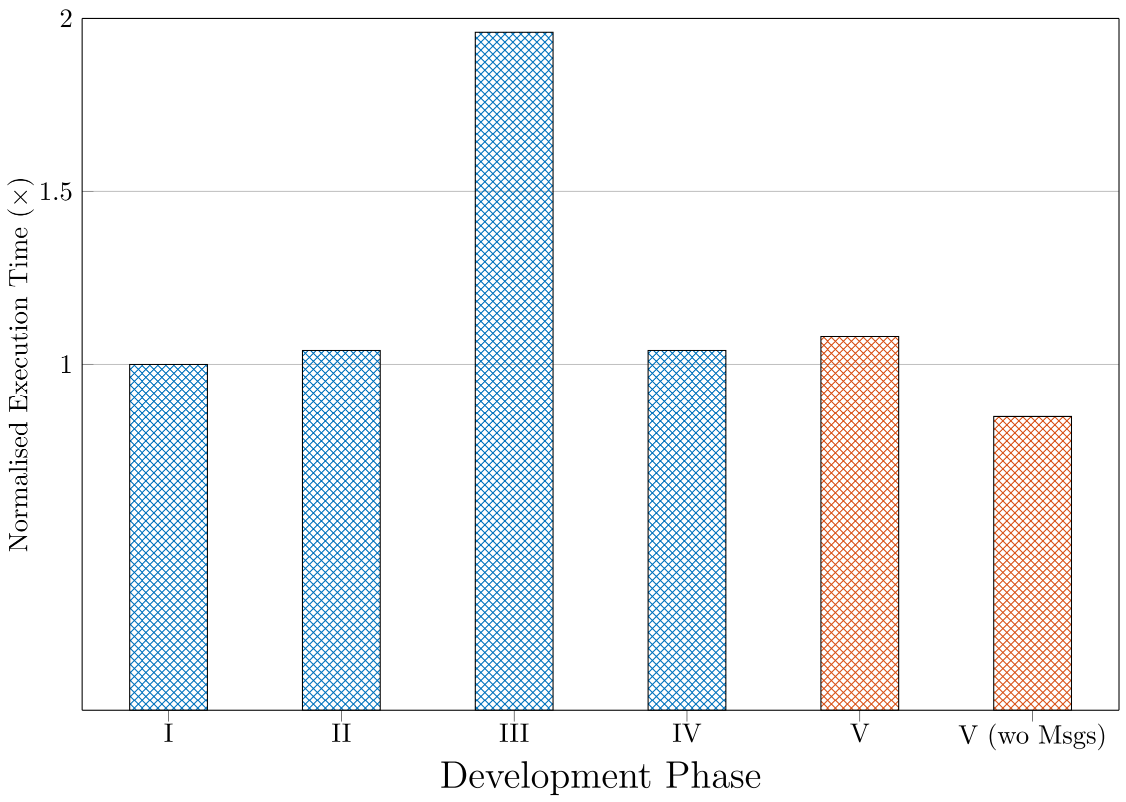


Figure 3.17: Normalised Performance of each Development Phase, relative to Phase I

III), caused significant reduction in speed which were mostly fixed on the next iteration. The latest version, despite all the added functionality shows only around 8% worse execution compared to the initial straightforward implementation when tasks with messaging constraints exist. In embarrassingly parallel problems, though, the last version shows a 25% compared to the first one, which is attributed to the more intelligent cache invalidation system. Finally it is noted that, possible future optimisation, may eventually reduce the execution time even more.

3.5.7 Summary

The Overall design of the Computational Unit Manager is presented on figure 3.18. The example program is described in the Application Design section using the High Level API, while the static & dynamic scheduling and dispatcher architecture are also presented.

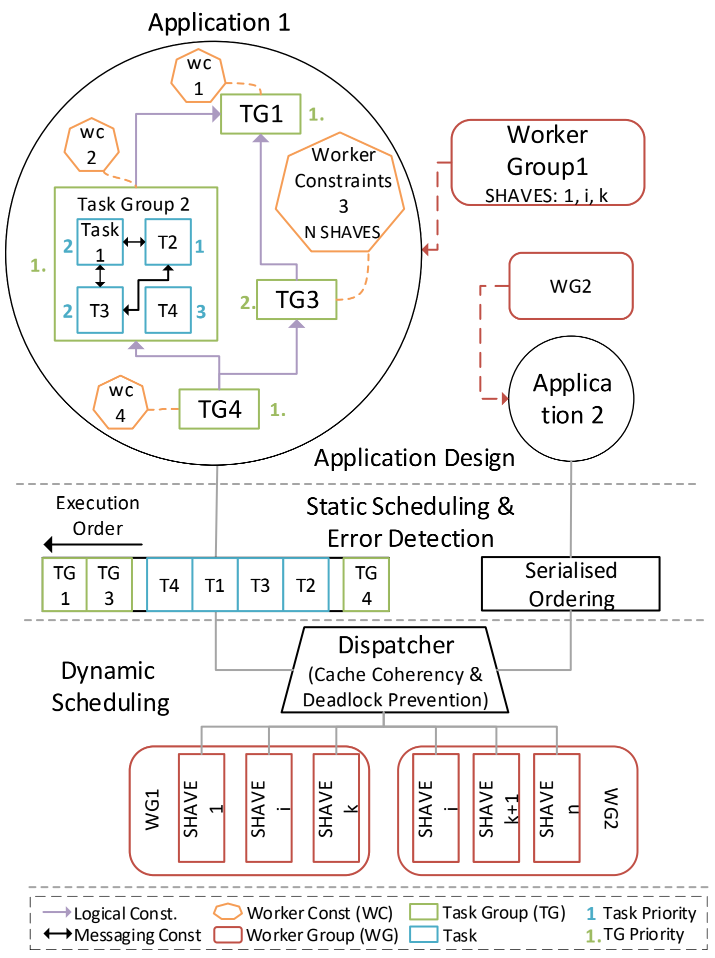


Figure 3.18: Computational Unit Manager Architecture

3.6 Visual Profiler

3.6.1 Role & Purpose

In addition to the previous modules an optional *Visual Profiler* is developed as a Quality of Life improvement. It aims to support the developers during the design space exploration and optimisation phase.

3.6.2 Implementation

The Profiler for the *ParalOS* perspective is implemented as a superclass for the Computational Unit Manager, it uses the exact same API, with the addition of two auxiliary function calls, that are used to determine the profiler options and are presented below

- `proflerOptions(SMPROF_OPTIONS)`. This method is used to configure the profiling options. The options include profiling total application time, fine grained task execution analysis, total power consumption and per worker memory usage.
- `int addTagName(SMTAG tag, const std::string& str);` This method is purely for informative purposes, as its' only use is to assign a name to the entrypoint tags. This name is shown in the Visual Output instead of the Tag's Id.

The profiler must be minimally invasive for precise measurements but more importantly it must not alter the timing of the various tasks executed. If the profiler is slow, this might mask potential racing issues between the workers or alter the cache behaviour due to different access patterns.

In this context, benchmarking code and SPM usage indicators are automatically injected in the computational unit module and the SPM Manager. This code consists of just tracking a hardware timer of the precise moment various events like a task has finished occurred. Regarding the SPM, counters are used to measure average and maximum usage. The last part of the profiler is to generate the results. Depending on whether a console or a visual output is preferred after the application has finished, the developer chooses one of the following two functions :

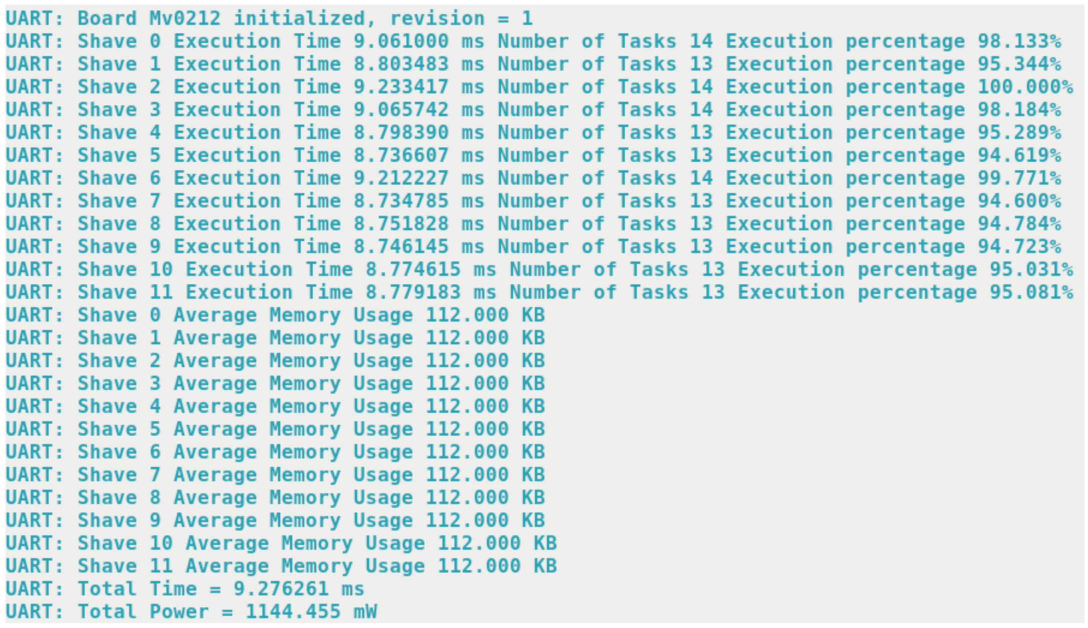
- `printResults()`. This method produces a console output and the results are calculated in the inside the VPU as a result only a subset of the possible parameters is presented. An example output is presented on Fig.3.19.
- `serialiseResults()`. This is the first step for generating a visual report.

Contrary to the console output, in the visual one no calculations are performed on the target platform. Instead all the measurements are serialised using the Google's Flatbuffers [73] and the generated object is transferred via the JTAG debugger to the host PC. In the host PC a python script is developed that using the *numpy* [74] & *matplotlib* [75], calculates and visualises the produced results. Finally, using *Jinja Templates* [1] and the *Bulma CSS Framework* [2] an HTML report is automatically compiled by calling the makefile directive `make profiler_results`

3.6.3 Output

Console Output

The Console Output provides a basic yet sufficient breakdown of the system's execution characteristics is presented as shown in Figure 3.19



```
UART: Board Mv0212 initialized, revision = 1
UART: Shave 0 Execution Time 9.061000 ms Number of Tasks 14 Execution percentage 98.133%
UART: Shave 1 Execution Time 8.803483 ms Number of Tasks 13 Execution percentage 95.344%
UART: Shave 2 Execution Time 9.233417 ms Number of Tasks 14 Execution percentage 100.000%
UART: Shave 3 Execution Time 9.065742 ms Number of Tasks 14 Execution percentage 98.184%
UART: Shave 4 Execution Time 8.798390 ms Number of Tasks 13 Execution percentage 95.289%
UART: Shave 5 Execution Time 8.736607 ms Number of Tasks 13 Execution percentage 94.619%
UART: Shave 6 Execution Time 9.212227 ms Number of Tasks 14 Execution percentage 99.771%
UART: Shave 7 Execution Time 8.734785 ms Number of Tasks 13 Execution percentage 94.600%
UART: Shave 8 Execution Time 8.751828 ms Number of Tasks 13 Execution percentage 94.784%
UART: Shave 9 Execution Time 8.746145 ms Number of Tasks 13 Execution percentage 94.723%
UART: Shave 10 Execution Time 8.774615 ms Number of Tasks 13 Execution percentage 95.031%
UART: Shave 11 Execution Time 8.779183 ms Number of Tasks 13 Execution percentage 95.081%
UART: Shave 0 Average Memory Usage 112.000 KB
UART: Shave 1 Average Memory Usage 112.000 KB
UART: Shave 2 Average Memory Usage 112.000 KB
UART: Shave 3 Average Memory Usage 112.000 KB
UART: Shave 4 Average Memory Usage 112.000 KB
UART: Shave 5 Average Memory Usage 112.000 KB
UART: Shave 6 Average Memory Usage 112.000 KB
UART: Shave 7 Average Memory Usage 112.000 KB
UART: Shave 8 Average Memory Usage 112.000 KB
UART: Shave 9 Average Memory Usage 112.000 KB
UART: Shave 10 Average Memory Usage 112.000 KB
UART: Shave 11 Average Memory Usage 112.000 KB
UART: Total Time = 9.276261 ms
UART: Total Power = 1144.455 mW
```

Figure 3.19: Example Console Output

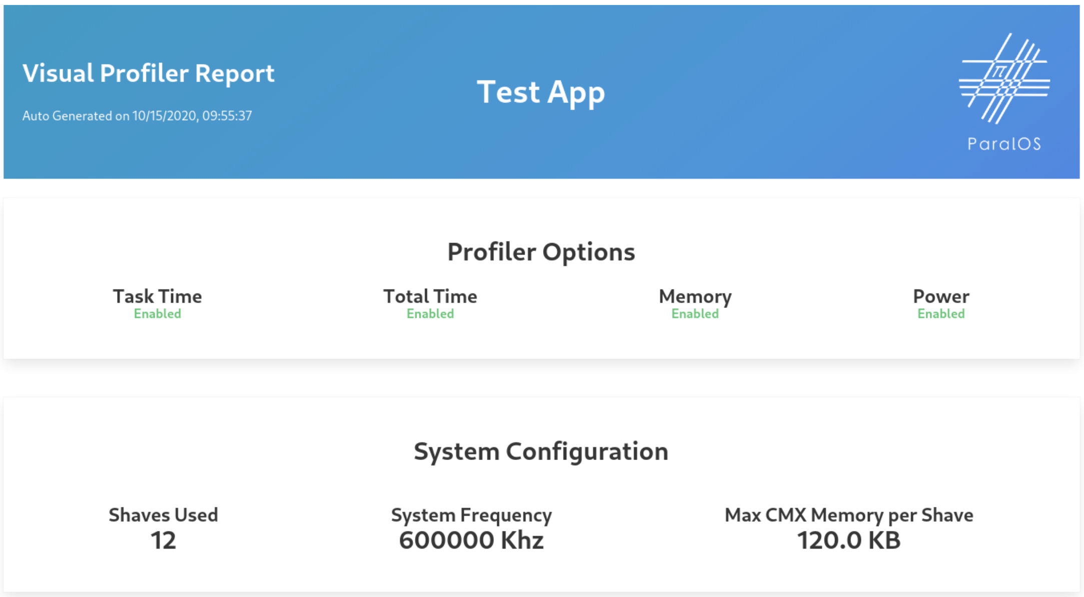
Visual Output

The produced HTML report is organised in a number of cards. The first card (Fig 3.20), shows information regarding the selected profiler results and System's Configuration. More specifically the number of workers (SHAVEs) deployed, the system's frequency and the maximum possible SPM (CMX) that is available to each worker. In addition the application's name and the generation time is also presented.

The Profiler can also measure the VPU's power consumption. This feature depends however on the SoC's board as it requires an external power meter IC that is connected to the host platform via an I2C interface. Currently only the *MA2x50 MV0212 & MV0180* evaluation boards are supported.

The total system's results (Fig. 3.21) are generated when the total time or power options are selected. It includes, depending on the developer's choice, the total application

[1] https://jinja.palletsprojects.com/en/2.11.x/
[2] https://bulma.io/



Visual Profiler Report

Auto Generated on 10/15/2020, 09:55:37

Test App

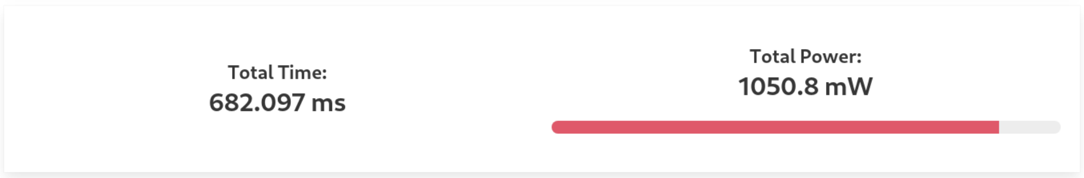
Profiler Options

Task Time	Total Time	Memory	Power
Enabled	Enabled	Enabled	Enabled

System Configuration

Shaves Used	System Frequency	Max CMX Memory per Shave
12	600000 Khz	120.0 KB

Figure 3.20: Visual Profiler System Configuration



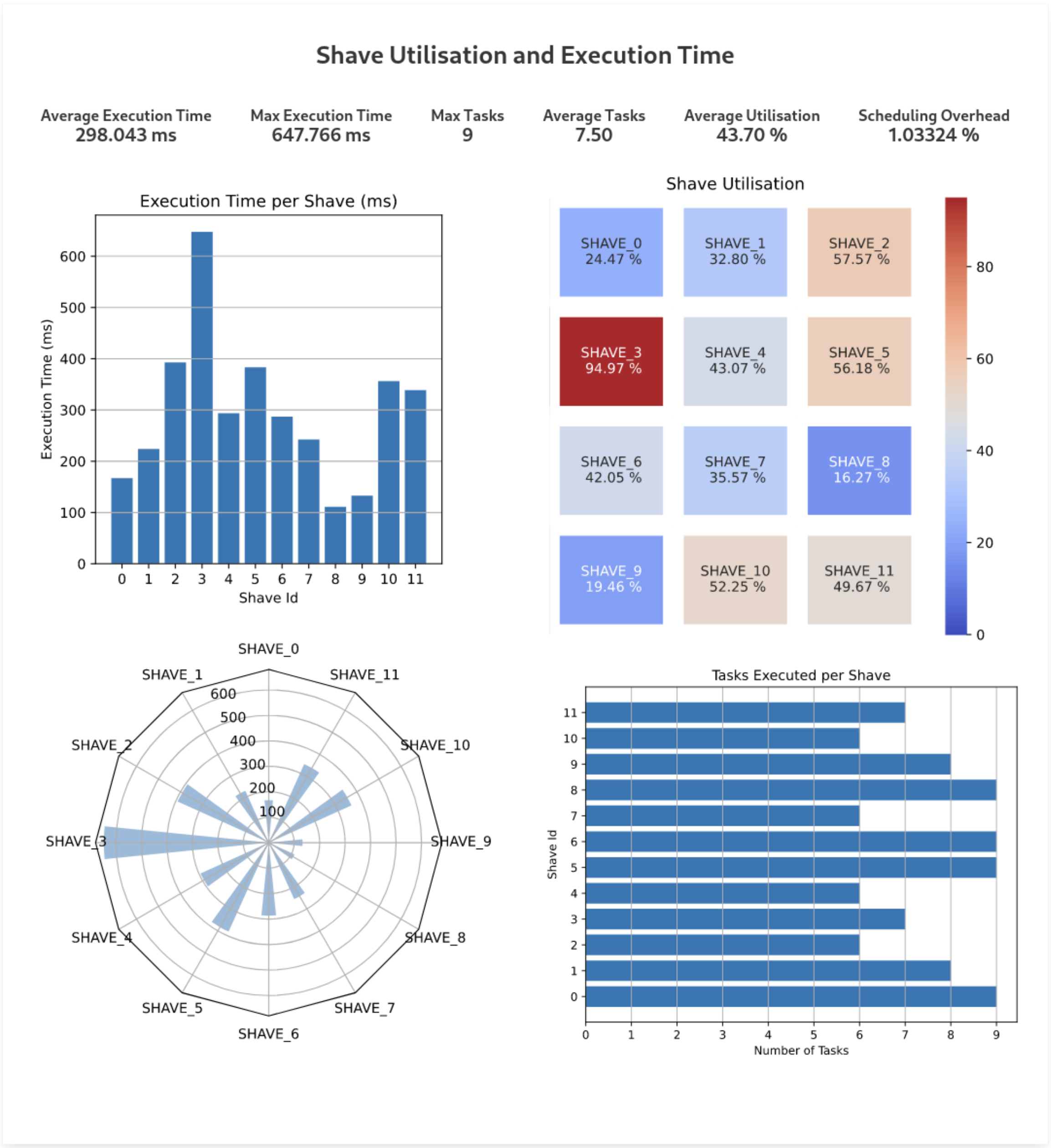
Total Time:
682.097 ms

Total Power:
1050.8 mW

Figure 3.21: Visual Profiler Total Results

execution time and the maximum power consumption of the VPU. The bar below the power consumption compares the measured power to the platforms power limit.

In Fig. 3.22 the worker related results are presented. These are generated when both task and total time options are enabled. The first line provides aggregated worker information, like the average and maximum task execution time, the average worker utilisation, and the induced computational unit manager's overhead. The later is calculated as the maximum cumulative worker execution time subtracted from the total execution time. The presented figures include the number of tasks assigned to each worker, a heatmap to express the shave's utilisation time and the final figures, that compare the worker's execution time.



Shave Utilisation and Execution Time

Average Execution Time	Max Execution Time	Max Tasks	Average Tasks	Average Utilisation	Scheduling Overhead
298.043 ms	647.766 ms	9	7.50	43.70 %	1.03324 %

Figure 3.22: Visual Profiler Worker Performance

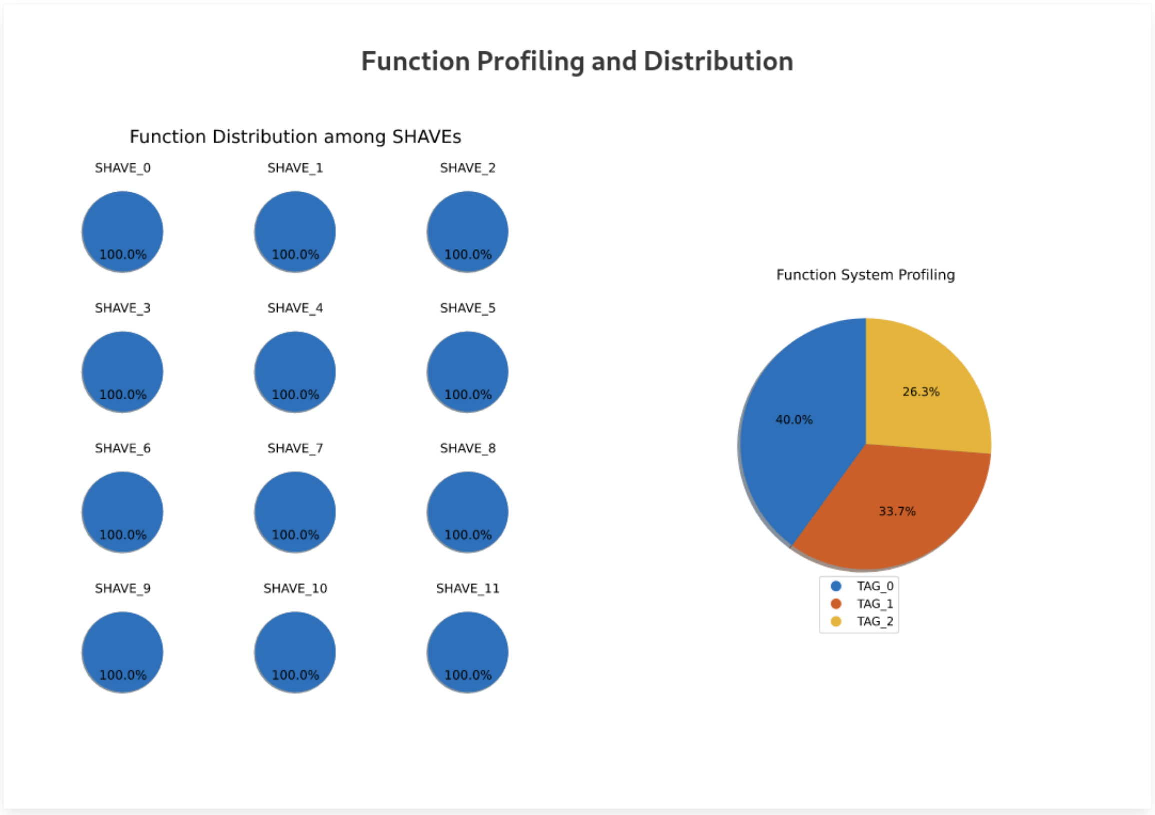
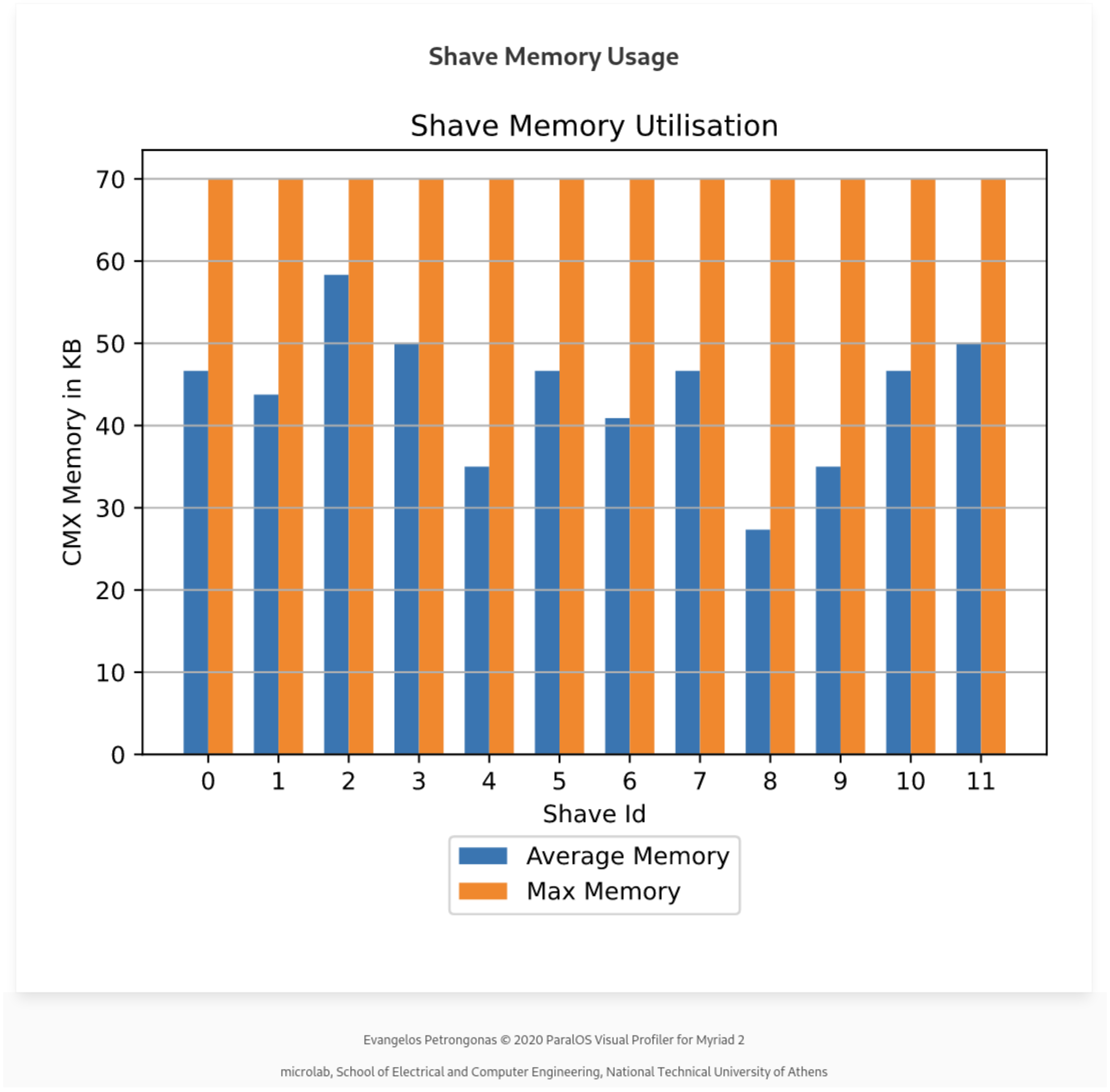


Figure 3.23: Visual Profiler Function and Task Benchmarking

The next card (Fig.3.23) showcases the function profiling when more than one tags/-functions are used. The left figure indicates how many different tags are executed on each worker, while the right one presents the total distribution of functions into all the workers.

The last card (Fig.3.24) contains the scratchpad memory usage results, when the respective option is selected. The figure shows the average and maximum SPM usage for each worker.

Finally, it is mentioned that in the current version of ParalOS only Myriad 2 is supported and support for Myriad X is scheduled for the next iteration of the framework.



Evangelos Petrongonas © 2020 ParalOS Visual Profiler for Myriad 2

microlab, School of Electrical and Computer Engineering, National Technical University of Athens

Figure 3.24: Visual Profiler Worker Memory Report

3.7 Workflow

3.7.1 High Level API Usage Example

In this section a brief example of how the High Level Segment is used to develop Applications using the *ParalOS* High Level Segment. As an example the application described in section 3.5.2 will be presented in the flowchart of Fig.3.25.

Some of the key usage notes are described below.

- (Step 1) The whole system is managed by a single object of the Computational Unit Manager class.
- (Step 2) Depending on which of the 2 parameters are provided, the system deduces, which low level submodules are used and initialises those who are.
- (Step 6) Each of the the `worker_entrypointsX` array holds the memory address of the entrypoint of the different functions that thw workers can execute. It is mentioned, that despite whether all the workers will be used, it is imperative for the correct system execution that all possible entrypoints are defined for every worker.
- (step 8) The API provides various methods to perform the same operation, like the priority definition.
- (Step 10) The Static Scheduling can be substituted with the `schedulingAndValidation` method during development.
- (Step 12) During this step the information regarding the messaging execution are passed to the task arguments in order to be available from the worker's execution perspective.

3.7.2 Low Level Development

The low level development is pretty straightforward and follows the same paradigm as the one proposed by the various Programming Guides [39, 76]. The only difference is the use of *ParalOS Low Level Segment* modules for memory management and IPC

3.7.3 13 Steps to Success

Despite relating the number 13 with misfortune and a rival football team[3], in the *ParalOS* context it represents the maximum number of steps required from the initialisation of VPU to the application's execution. This number is of great importance, especially for beginners, who by simply reusing the same function code, with small alterations, can execute their programms to the VPUs. Equally important is the ability to speedup Development and DSE, without compromising or restricting their ability to perform low level *Ninja* Optimisations [77].

[3]Panathinaikos AO

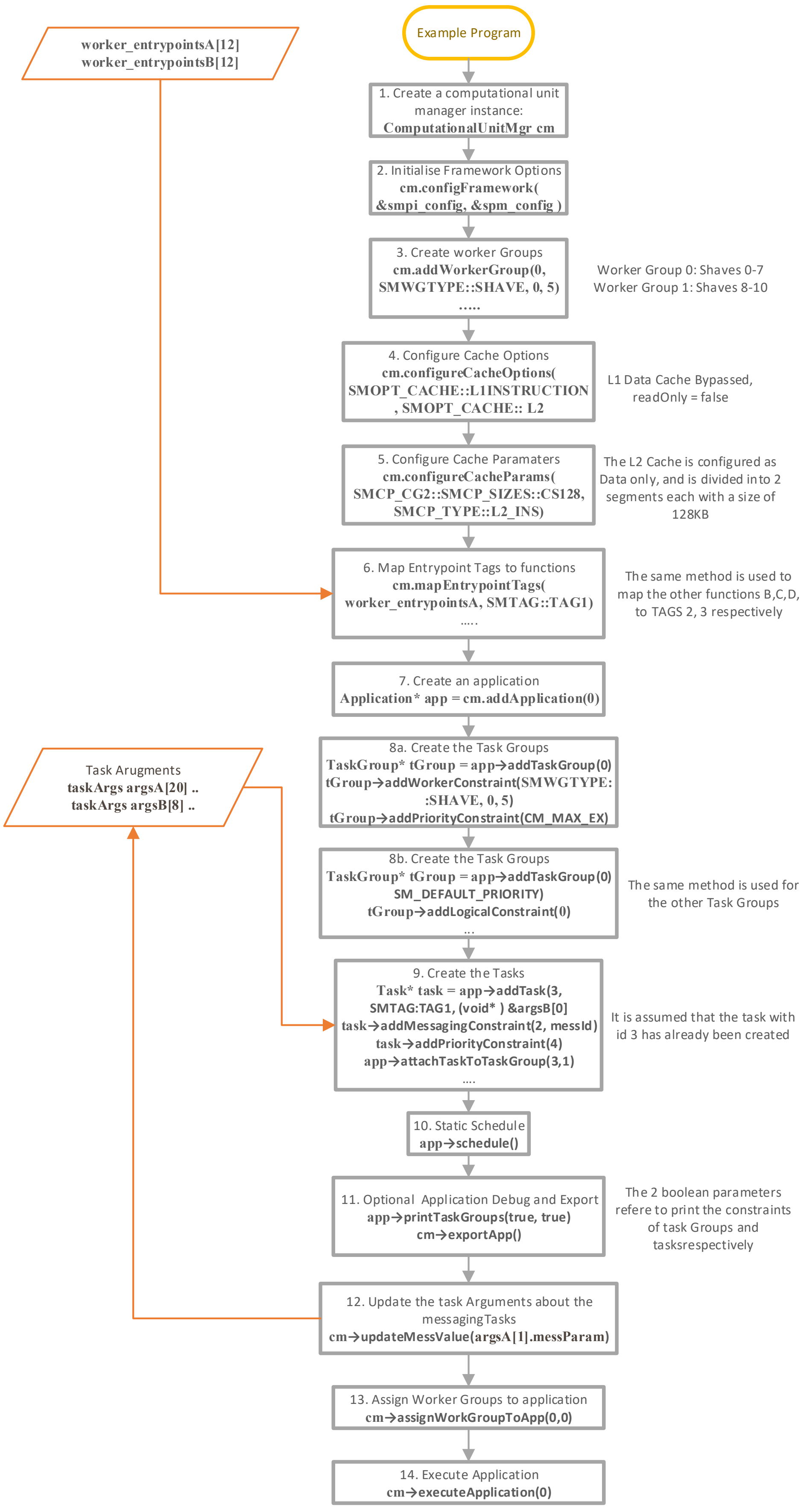

Example Program
worker_entrypointsA[12]
worker_entrypointsB[12]
1. Create a computational unit manager instance:
ComputationalUnitMgr cm
2. Initialise Framework Options
cm.configFramework(&smpi_config, &spm_config)
3. Create worker Groups
cm.addWorkerGroup(0, SMWGTYPE::SHAVE, 0, 5)
.....
Worker Group 0: Shaves 0-7
Worker Group 1: Shaves 8-10
4. Configure Cache Options
cm.configureCacheOptions(SMOPT_CACHE::L1INSTRUCTION , SMOPT_CACHE:: L2
L1 Data Cache Bypassed, readOnly = false
5. Configure Cache Paramaters
cm.configureCacheParams(SMCP_CG2::SMCP_SIZES::CS128, SMCP_TYPE::L2_INS)
The L2 Cache is configured as Data only, and is divided into 2 segments each with a size of 128KB
6. Map Entrypoint Tags to functions
cm.mapEntrypointTags(worker_entrypointsA, SMTAG::TAG1)
.....
The same method is used to map the other functions B,C,D, to TAGS 2, 3 respectively
7. Create an application
Application* app = cm.addApplication(0)
Task Arugments
taskArgs argsA[20] ..
taskArgs argsB[8] ..
8a. Create the Task Groups
TaskGroup* tGroup = app→addTaskGroup(0)
tGroup→addWorkerConstraint(SMWGTYPE::SHAVE, 0, 5)
tGroup→addPriorityConstraint(CM_MAX_EX)
8b. Create the Task Groups
TaskGroup* tGroup = app→addTaskGroup(0) SM_DEFAULT_PRIORITY)
tGroup→addLogicalConstraint(0)
...
The same method is used for the other Task Groups
9. Create the Tasks
Task* task = app→addTask(3, SMTAG:TAG1, (void*) &argsB[0]
task→addMessagingConstraint(2, messId)
task→addPriorityConstraint(4)
app→attachTaskToTaskGroup(3,1)
....
It is assumed that the task with id 3 has already been created
10. Static Schedule
app→schedule()
11. Optional Application Debug and Export
app→printTaskGroups(true, true)
cm→exportApp()
The 2 boolean parameters refere to print the constraints of task Groups and tasksrespectively
12. Update the task Arguments about the messagingTasks
cm→updateMessValue(argsA[1].messParam)
13. Assign Worker Groups to application
cm→assignWorkGroupToApp(0,0)
14. Execute Application
cm→executeApplication(0)

Figure 3.25: Example Workflow Application

3.8 Reliability & Fault Tolerance: A first approach

An important aspect of embedded systems design in gerneal, is the concept of reliability and fault tolerance. Quite often these devices are employed in real-time (RT) and mission critical scenarios. Vision Processing Units, due to their high performance, are even more likely to be used in RT applications like Advanced Driver Asistance (ADAS) in autonomous cars, or for autonomous operations in UAV [78].

The main purpose of *ParalOS* was not to tackle theese issues however in every step of development design choices were made in order to facilitate, future extensions that address these important issues. Nevertheless in its current state *ParalOS* provides some utilities to improve reliability these are:

- **Hermes & SMPI Protocol checksums**. The data exchanges both within the SoC's subunits and the various IO Interfaces are a common suspect for execution abnormalities. A compensation mechanism is provided via the custom message protocol, via the optional crc mechanism. If the CRC code and the received data do not match the system can be programmed to request again the data.

- **SPM Manager & Dynamic Allocations** Coding standards , like the *JSF Air Vehicle C++* [79] and *MISRA C++08* [80] are adopted. These standards prohibit the use of dynamic heap memory, while more recent ones [81] permit it under strict conditions. For compliance's purposes the *SPM Manager* is provided in two flavors. One that only supports static memory allocation and one with dynamic. Even so the dynamic manager, provided a guaranteed WCET therefore can be utilised with caution.

- **Computational Unit Capabilities** The Computational unit manager does not employ any mitigation techniques. However, it can be manipulated in a way to increase the application's robustness. Triplication and polling is very easily developed, while the cache subsystem can be bypassed. In addition due to its modularity, developers can add extensions to support more advance functions.

Chapter 4

Appications and Evaluation

The evaluation is performed on *Myriad 2* and *MyriadX* using both synthetic tests, designed to stress specific aspects of the systems as well as, real applications that are representative of the algorithms typically deployed in such platforms. The applications include Convolutional Neural Networks and Visual Based Navigation Algorithms (VBN). Finally a qualitatevly analysis on the reduction of development effort is presented.

4.1 Evaluation using Synthetic Benchmarks

4.1.1 Memory Manager

The *SPM Manager* of the *ParalOS: Low Level Segment* is compared ro the TLSF [7], the most prominent allocator alternative, as established previously for the VPUs and embedded systems in general. Both allocators are implemented and evaluated on the MyriadX VPU. For the evaluation,the following test is employed.

The evaluation program consists of a single task that is executed on one shave, since the SPM is decentralised. Additionally the L2 cache is configured in 1 partition of 16Bytes for data, better simulating the real use conditions. Overall it performs $3n$ allocations each having a size of B bytes. The test is divided in two phases: In the first phase n successive allocations are performed, followed by the same number of frees, aiming to measure the maximum used memory and the overall spatial performance, The second performs another $2n$ operations in a random pattern, i.e., which they are either malloc or free, in order to examine the timing performance. It is guaranteed that the pattern will never request to free a block that has not been allocated, or request to free the same block more than once. The results of our evaluation test are presented in Table 4.1.

Regarding the memory usage, the results show that the proposed method does not induce any spatial overhead. Additionally, up to 33% reduced memory footprint is achieved compared to TLSF. In extreme cases, e.g., when a single byte is requested, the proposed framework achieves 93% better spatial performance. The latter improvement was observed when allocating single bytes and it has to be noted that TLSF does not support direct

Table 4.1: Evaluation of the proposed *SPM Manager*

Test	Memory Usage (KB)			Execution Time (ms)		
$(n \times B)$	TLSF[7]	Proposed	Diff (%)	TLSF[7]	Proposed	Diff (%)
5Kx4B	30	20	-33	1.342	1.457	+8.6
1Kx8B	10	8	-20	0.458	0.472	+3.2
500x16B	9	8	-11	0.343	0.362	+5.8
100x128B	13	12.8	-1.5	0.289	0.304	+5.3
10x1KB	10.02	10	-0.2	0.206	0.210	+2.1

allocations of less than 4 Bytes. In this case a 4 byte block is returned instead. This reduction, however, tends to decrease as the requested block size increases, providing negligible benefits for block sizes greater than 1KB.

Regarding the timing performance, higher execution times are recorded which are justified by the different physical location of the manager's metadata. Nevertheless, this overhead is steadily below the 10% mark, maintaining comparable execution times. The limited overhead cost is attributed to the exploitation of the cache hierarchy. In particular, the module's control structures, which are stored in DDR, are accessed via the L1$ & L2$ infrastructure, thus providing a compensation mechanism.

4.1.2 IPC Throughput

The next benchmark, evaluates the performance of the *SMPI's IPC* scheme on MyriadX. The test's purpose is to broadcast a fixed amount of total initial information among workers, which use fixed size packets to transmit sequentially their own share of data to every other worker. In other words, assuming an image I is spread equally among the workers, the algorithm's purpose is at the end of the execution for all workers to have received the whole image. The comparison relies on the method used to transmit the messages. On the one hand the default technique provided by the vendor is using, which involves the DMA Engine for runtime communication and the other is utilise the proposed *SMPI* API.

The results presented in Fig. 4.1 indicate that the *SMPI's IPC Scheme* offers up to $6\times$ better scaling. In particular, as the number of workers increases, the total number of messages increases and floods with requests the single DMA engine available on the chip overloading it and consequently, it becomes the main performance bottleneck. In contrast, the proposed decentralised buffering technique uses more efficiently the high-bandwidth SPM interconnect for parallel exchanges. Hence, instead of being almost proportional to the total number of messages (DMA approach), our IPC time becomes proportional only to the messages per single worker. This corollary justifies, why the DMA engine surpasses in performance the SMPI one, when the number of workers is small $(2-3)$, the DMA requests are limited, but each request must handle greater amount of a data, an area that the DMA engine excels into. Another thesis [82], using a different measurement method

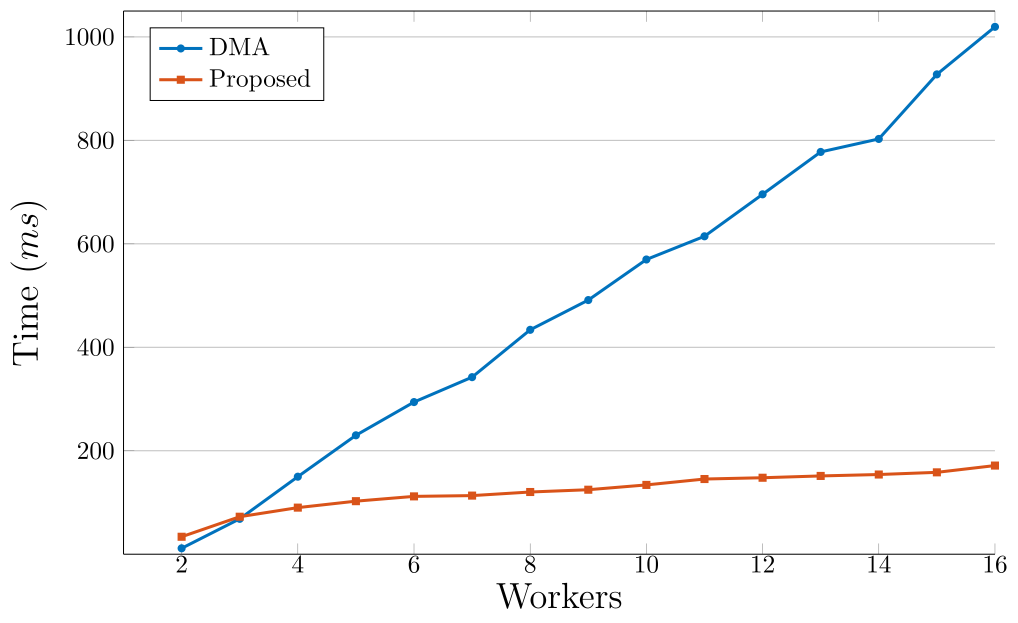


Figure 4.1: Scaling of the proposed SMPI IPC w.r.t. the number of workers.

has reached the same conclusion that the DMA engine can be easily overloaded.

4.1.3 Computational Unit Manager: Scaling

The next synthetic test examines the scaling capabilities of our dynamic *Task Scheduler* on MyriadX. A test application consisting of n tasks is employed. The tasks are organised randomly in task groups, which have artificial logical constraints among them. No messaging constraints are defined for the tasks, so the messaging susbsystem is not utilised. Finally, it is guaranteed that there will be no circular dependencies. Each task is a function that returns immediately without any computations. The cache is configured to expect read/write data, so that the cache coherency protocol is deployed. All the workers are assigned to a the same partition for Data and Instructions. These two partitions have a size of $128KB$ each. By assigning all the workers to the same partition, maximum congestion and more flushes/invalidations occur, stressing the system even more.

The results presented in Fig. 4.2 show that the scheduler exhibits linear scaling w.r.t the number of tasks. In addition, the cost of scheduling per task appears to be inversely proportional to the number of tasks.

4.1.4 Computational Unit Manager: Multiple Application Execution

Typically, the concurrent execution of applications suffers from inherent issues such as deadlocks, cache coherency, etc. To overcome these bottlenecks, *ParalOS* employs mechanisms, which, however, induce extra overhead, like the deadlock prevention mechanism that locks the execution of tasks with messaging constraints. In addition, workers that execute different applications but nevertheless share the same caches, may have undesired

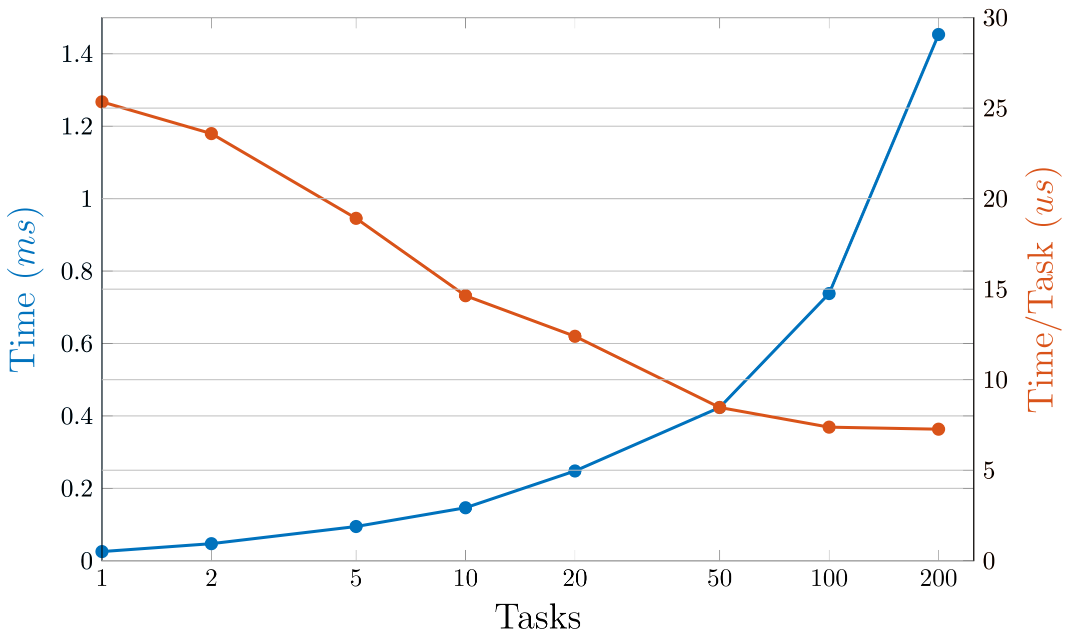


Figure 4.2: Scaling of the proposed Computational Unit Manager Scheduler w.r.t. the number of tasks

flushes and invalidation may occur to them.

This overhead is measured by running multiple distinct instances, both in parallel and sequentially, of the custom CNN engine that is developed and discussed in Section 4.3, using a ship detection network. The benchmark is executed on *Myriad 2* partially for better comparison with literature and as discussed the CNN engine does not yet support the NCE unit on MyriadX.

The results are depicted in Fig. 4.3, where the blue dotted bar is the execution time w.r.t. the number of applications running sequentially.In contrast, the orange dotted bars indicate the execution time when the same number of applications is executed concurrently. As shown, the performance overhead is small, i.e., it ranges from 1.7% to 9.6% depending on the number of applications executed.

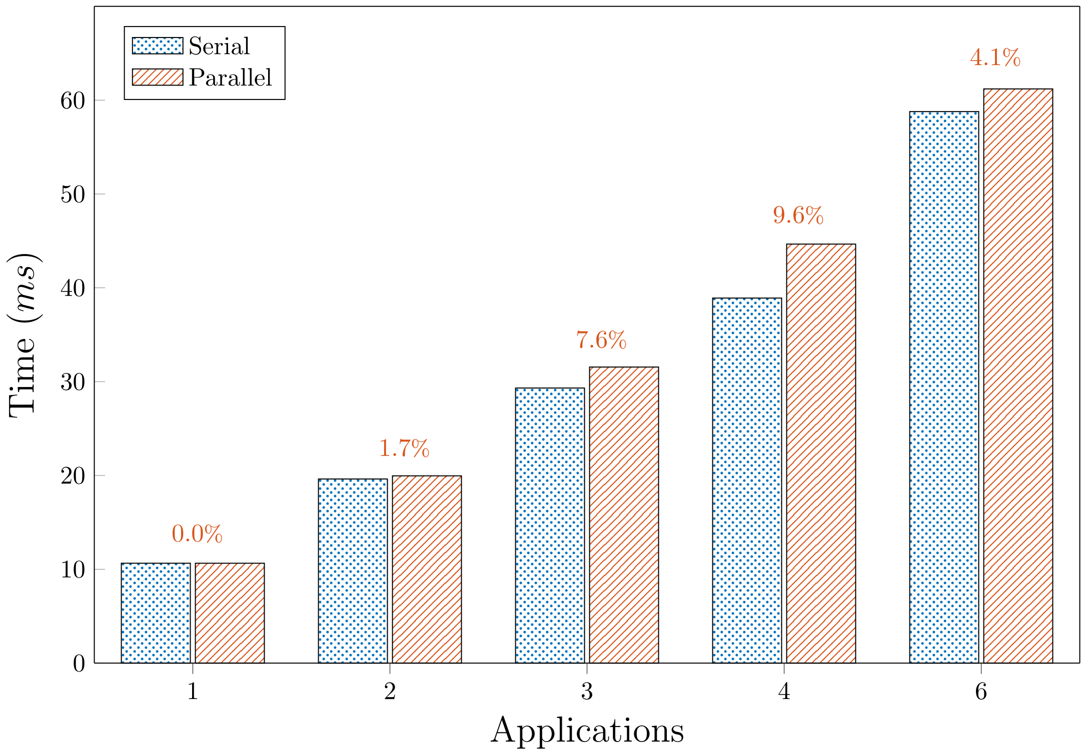


Figure 4.3: Parallel vs serial application execution (CNN)

4.2 Visual Based Navigation

4.2.1 Introduction

Visual-based navigation (VBN) uses computer vision algorithms and optical sensors, to extract the visual features required to the localisation of the surrounding environment. Camera sensors, coupled with advanced image processing and tracking algorithms, can provide accurate sensing capability to obtain full 6 Degrees of freedom (DOF) relative pose information. A number of existing image processing algorithms for pose estimation and tracking utilise fiducial markings on the tracked object. However, retrofitting the existing infrastructure with these fiducial markings is impractical. Other pose estimation algorithms use an object surface model representation to track the object, avoiding the use fiducial markings by taking advantage of the known structural configuration [83]. This kind of complex image processing pipeline that benefits from execution on the edge, is considered a prime candidate for the VPUs.

4.2.2 HIPNOS algorithm & *ParalOS*

An algorithm that makes use of such a structural model is *HIPNOS* [84, 8]. In particular, pose tracking of a target satellite [23], namely *ENVISAT* is performed in the following 5 Steps (Fig. 4.4):

1. Edge detection in the input image
2. Depth map rendering
3. Edge detection in the rendered depth image
4. Perpendicular edge matching
5. Pose refinement

This algorithm is implemented in the Myriad 2 VPU with the support of *ParalOS* . All the functions apart from pose refinement (executed on LOS), utilise the proposed *SPM Manager* for the scratchpad memory allocations. *ParalOS* , however, is essential. The efficient implementation of the edge detector and the rendering as explained in the following sections and in [85].

4.2.3 Canny Edge Detection

The Edge Detection is materialised with a Canny edge detector [86] on both the intensity and depth images. It implements a Sobel convolution for the gradient calculation and uses hysterisis thressholding to identify the strong edgels and ignore the weak ones. This last part is a recursive process, as weak edges are labelled as strong when there is a strong edge in their neighbourhoods. Thus, due to the labelling of new strong edges, the whole

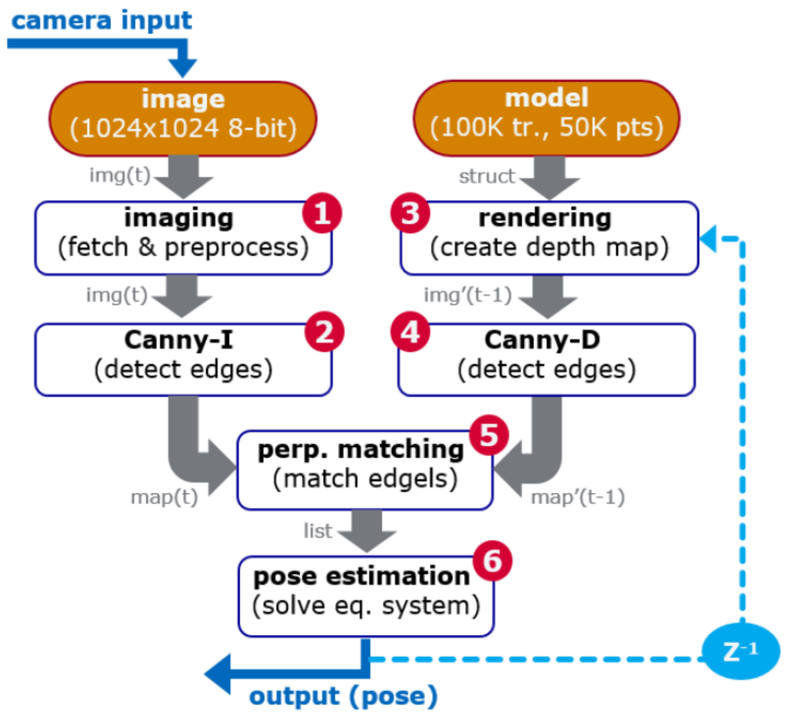


Figure 4.4: HIPNOS algorithm [8]

edge map must be re-examined. When parallelised though, messaging constraints arise, as neighbouring bands must exchange their border strong edgels. This was an important limitation that was addressed with the *SMPI* module of the *ParalOS: Low Level Segment* . The workers use the `SMPISend` and `SMPIReceive` function for messaging. Additionally, the custom protocol was utilised, in order to identify the band from which the edgels were received.

4.2.4 Rendering

For Depth Rendering, the algorithm employs a triangle mesh model and the current 6D pose vector to generate an image, which encodes the distance of the modem's surface to the camera. A *Rasterisation* algorithm handles the projection of the model's triangles on the frame. Using a bounding box traversal in the projected vertices, the pixels that reside within the projected triangles can be found and then calculate for each of them their distance from the model.

This is a characteristic content dependent problem as the execution is determined mainly by two parameters i) the mesh model which is known and ii) the relative camera position, that is received on runtime. This position greatly impacts the execution time because it determines what parts of the model are projected onto the image.

The parallelisation scheme of the renderer is straightforward, as rendering overall is an embarrassingly parallel application (this is why the GPUs excel at it). The image is divided into N bands and every band is assigned to a task.

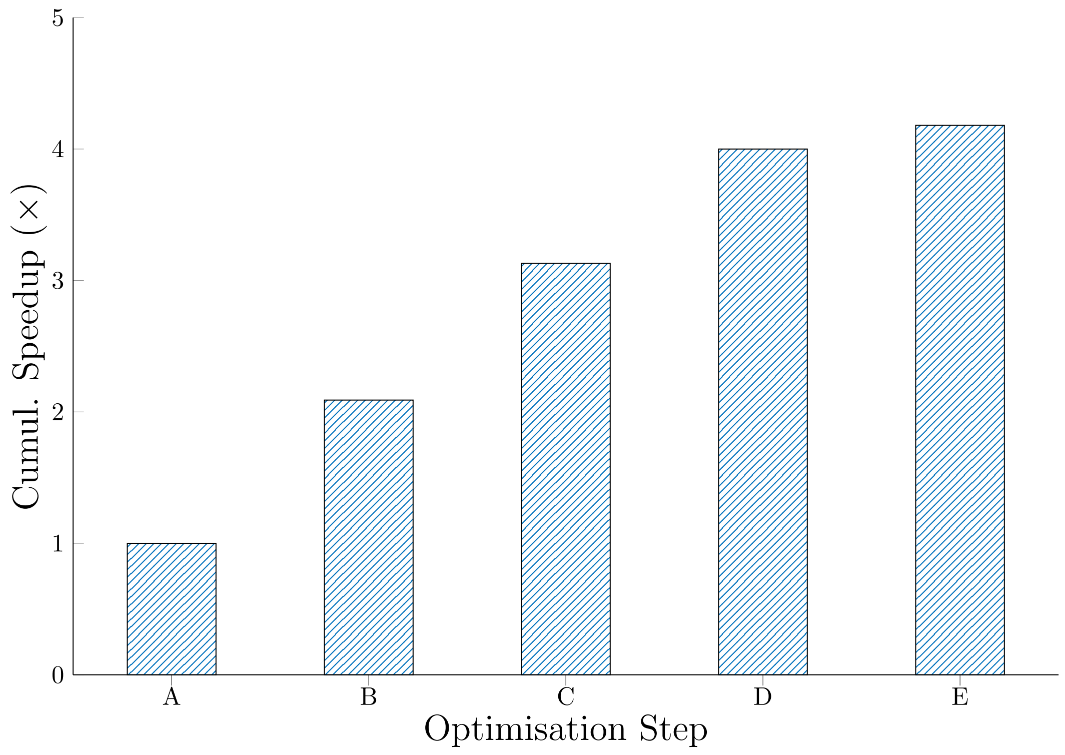


Figure 4.5: Speedup of Rendering when using *ParalOS* optimisations

The *ParalOS: High Level Segment* offers an array of options for performance optimisations the impact of which are presented on Figure 4.5. Each optimisation step is calculated as the speedup compared to the static scheduling of *Step A*.

- **Step A** represents the low-level optimised kernel of rendering, that is executed on the 12 SHAVEs of Myriad2 without using the proposed framework via static scheduling.
- **Step B** employs the *ParalOS* dynamic scheduler, which delivers a speedup of 2.09× by minimising the idle time of each worker.
- **Step C** is the result of performing Design Space Exploration using the *Visual Profiler*, to find the optimal parallelisation scheme increasing the speedup to 3.13×.
- **Step D** reflects the 4× total improvement due to the optimisation of the cache configuration, i.e. exploring the different size and partition configurations. In addition, the mesh model is only read by the workers thus it can be tagged as read-only hence, avoiding unnecessary cache flushes.
- **Step E** makes use of application-specific characteristics to deduce the priority of the tasks. In particular, tasks that are responsible for generating the central frame bands are more likely to contain the rendered model than the border ones, and thus, they were assigned a higher priority in order to be executed first. This final step brings the speedup to a total of 4.18×.

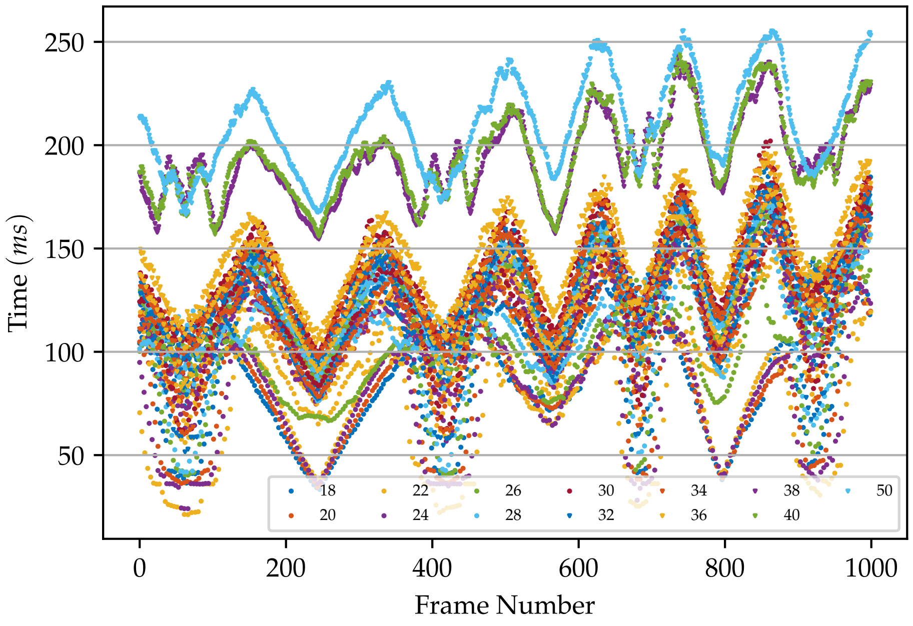


Figure 4.6: Design Space Exploration for the various number of bands

Another deciding factor is the *deviation* between the maximum an minimum execution of time. Due to the implementation, the rendering process shows a preference to the vertical alignment of the model, i.e. if the angle between the axis of the model and the horizon, is close to 90°. Therefore "vertical" images tend to perform better than their "horizontal" counterparts. For real-time operations though, criticality constraints might require this deviation to be kept low. For example an implementation with a much faster average execution time but with high outliers, may be dropped in favour of a slower yet more stable one.

Taking all the above into consideration, the following experiment was performed to find the optimal number of bands. 1000 different camera positions were used to render the model from multiple perspectives and distances. The results are presented on Figure 4.6 showing comprehensive execution time of all the tests & Figure 4.7, which illustrates the pareto front between the average execution time and standard deviation for the different band numbers. The collected data suggest that the optimal band number in terms of performance is 22, but when we also consider the *stDev* a more attractive solution might be 32. The smallest possible value due to memory limitation is 18, which interestingly performs worse than other option.

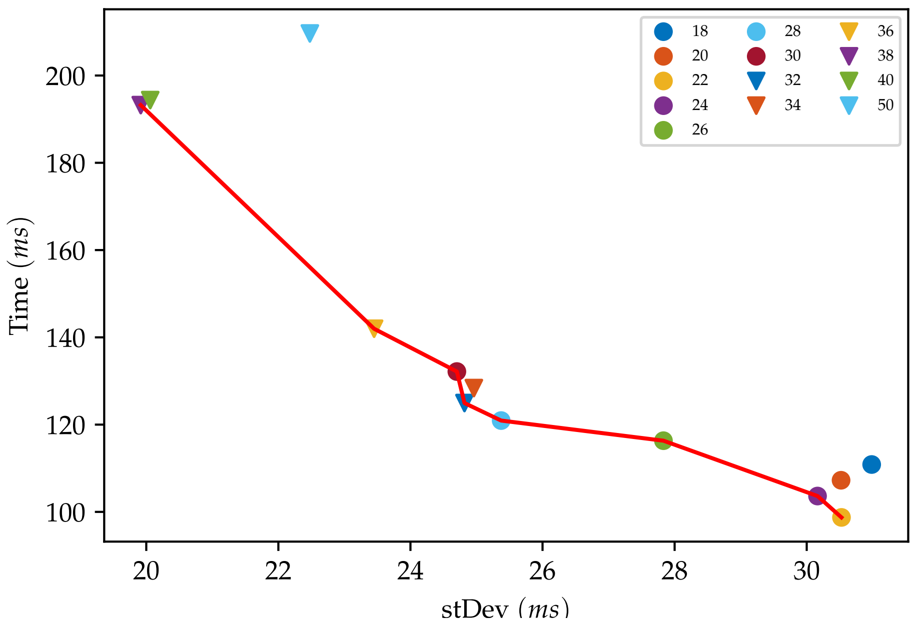


Figure 4.7: Pareto front of average execution time and deviation for different band numbers.

4.3 CNN Engine

4.3.1 Introduction & Motivation

Convolutional Neural Networks (CNNs) have emerged as one the most characteristic and desired application for edge devices. CNNs on VPUs have been used for a *"myriad"* of applications ranging from space [87] to small scale always-on (AON) scenarios [25, 88]. VPUs have reached the performance capabilities of FPGA based implementations [89, 25, 90, 91] and in terms of performance/watt are in course to surpass them.

CNN execution for the Myriad family of VPUs is already provided via the discussed *OpenVINO* Toolkit [15] and using custom implementations [24, 82, 92]. The downside of all the described solutions, is that they make exclusive use of the whole VPU, thus not allowing the execution of other algorithms or utilise CNNs as a part of an extended image processing pipeline.

Consequently, a new CNN engine for inference running is developed, which is built using the underlying *ParalOS* infrastructure. With this Engine a CNN is considered nothing more than an application (Fig.4.8) and is handled as such by the system.

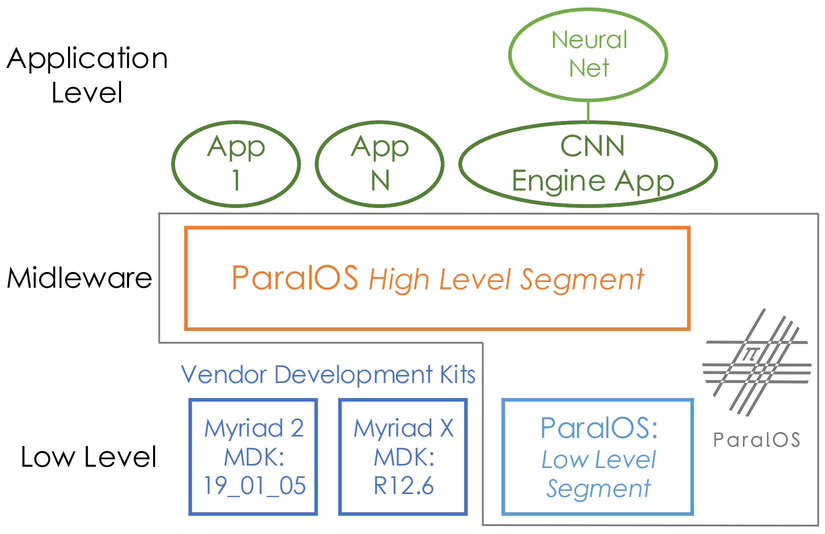


Figure 4.8: CNN engine as part of the *ParalOS* stack

4.3.2 Implementation

Overal Design

The implementation design was based on two important key concepts

- **Cross Compilable.** The engine shall be designed for ease porting between various platforms, like general purpose conventional CPUs, VPUs etc. Despite the increased design complexity and development time, it has some very important benefits.

  - *Easier Debugging.* Debugging application on CPUs is a much easier process, than in any other available platform.
  - *New functionality* Adding new functionality or trying new techniques, for research purposes, shall be first checked in an established and guaranteed platform before attempting porting it to a heterogeneous one.

- **Compatible with High Level NN frameworks**. The vast majority of CNNs is developed using industry standard frameworks like *TensorFlow* [48], *PyTorch* [49], *Caffe* [93]. The proposed CNN engine shall seamlessly support execution of models, generated from at least on of these platforms. Otherwise manual porting, would be tedious error-prone and overall reduce the effectiveness of the engine.

Grounded on these guidelines, the CNN Engine was implemented with a modular 4 layer architecture that is presented on figure 4.9 and will be explained below.

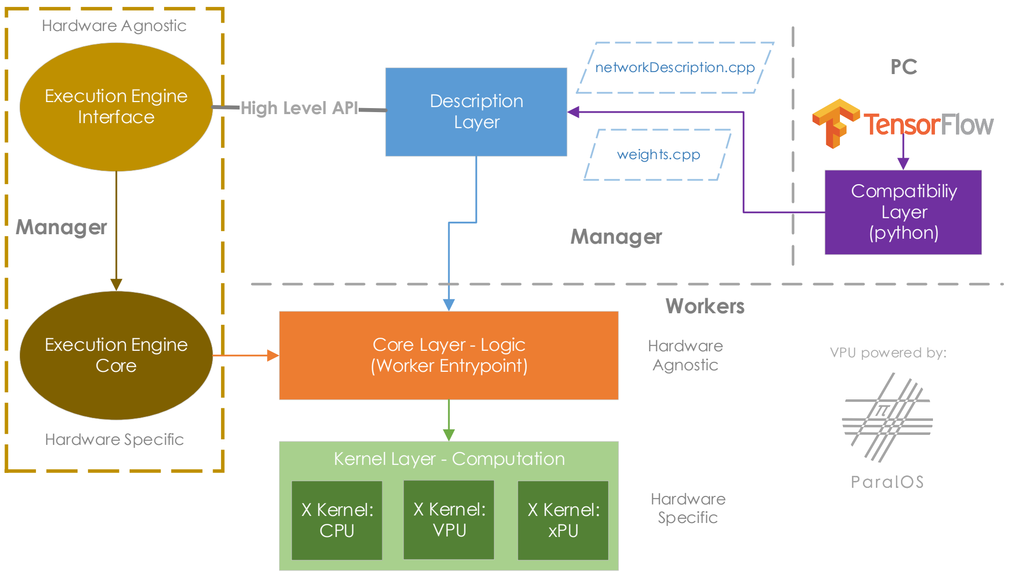


Figure 4.9: CNN Engine Architecture

Framework Compatibility Layer

The compatibility layer is a python script that is responsible for translating models created using the *TensorFlow* framework, to the Engine's Immediate Representation (IR). More specifically a *".h5"* file that stores the model is given as input to the script and the later automatically outputs three files, namely "*networkDescritption.cpp*", "*network-Weights.c*", "*networkWeights.h*". The first describes the network, while the other two define the various weights. In addition to that, the script can also be used, to convert the weights of the network, to a different precision arithmetic like the 16bit floating point, that is supported by Myriad.

Description Layer

The Description Layer, is the top layer of the engine and is executed on the Manager Cores .It is hardware agnostic and developed in C++. It is used either directly by the developer or by the *Compatibility Layer* to describe the network, using the various supported Network Layers. In this early version a handful yet fundamental layers are supported, *2D Convolution*, *Max Pooling*, *Fully Connected* and the activation functions of *ReLU* and *SoftMax*. It is important to clarify that this layer, does not take part in the actual execution of the network but it used only for its description.

Core Layer

This layer is perhaps the most important one, in terms of performance as it expresses the logic behind the execution of each layer. Each core layer is couple with its description counterpart, and is executed on the worker. For this reason the cores are written in C for better compatibility. The cores for the most part are platform agnostic, although specific aspects of them like the memory management are dependent on the platform, but well-thought use of macros, abstract their complexity.

Kernel Layer

The cores despite being the "*brains*" of the execution, they do not perform any computation. These are left to the "*muscle*" the Kernel Layers. The Kernels are platform dependent but have a common calling convention. In particular, each kernel call is responsible for producing a single output line, which leads to faster and more efficient programming. Thy are developed using C or assembly but can support other languages and methods as well. The kernels for the VPU written in SHAVE assembly are reused from previous theses [94, 82, 92].

Execution Engine

The execution engine is responsible for orchestrating the actual execution of the engine and is divided into two parts. The high level part provides a standard interface across all the platforms to interact with the lower part that is actually responsible for the execution. The lower part in the VPU case invokes *ParalOS* to both create the application and execute, while in the CPU uses a version of the *ParalOS: High Level Segment* , which is ported to the CPU. Implementation Details are not important for this particular discussion therefore omitted.

API

The API for the CNN Engine is pretty straightforward and briefly described:

- `CCCN::Model m`
  Create an instance of the CNN engine model

- `createModel(m)`
  This function is automatically generated by the COnversion Script. If it is not used the developer manually inserts the layers into the model.

- `m.finaliseModel()`
  It freezes the model and prepares it for execution.

- `m.executionEngine->initialise(<cm object>)`
  The underlying engine core is invoked initialising the execution system. For the VPUs, the translates to creating the application and sceduling it.

- `m.executionEngine->execute().`
  This method executes the network based on the target platform. For the VPU's it calls the `executeApplication` method. When multiple inferences need to be performed, only this function is called without reinitialising the network. An alternative execution method is using *ParalOS* API by specifying the network's application id.

4.3.3 Benchmarking

The evaluation of the CNN engine was performed using three networks, one based one *CIFARR-10* as presented on [94], a *MNIST* classifier from [25] and a custom ship detector from satellite images which will be described shortly. The *MNIST* classifier due to its small size fits as a whole in the SPM, thus it is a great comparison for measuring the *ParalOS* Overhead. This is also valid for the *CIFARR-10* engine, because the lower levels of the proposed CNN Engine are quite similar.

The custom Ship Detector Network consists of 4 Convolution Layers of 32, 16, 64 and 32 filters accordingly followed by two Fully connected layers. Max pooling layers are used to reduce the spatial dimensions, between the convolutions. The complete architecture is presented on Fig.4.10. The model was trained in TensorFlow using the dataset from[95], with 3000 images and validated using 1000. After 12 epochs it achieved an accuracy of 96.8%. The dataset initially consisted of 80×80 RGB images, but were upscaled to $128x128$ during preprocessing.

The results are presented on Table 4.2 and show that the performance overhead (%difference) induced by *ParalOS* is less than 10%. The only exception is the *MNIST* classifier. Its' implementation [25] is highly customised and does not use the DDR. In the proposed

Table 4.2: Evaluation of *ParalOS* with CNN applications

CNN Network	**Execution Time (ms)**			
	[24]	[25]	Proposed	Diff. (%)
CIFAR-10	7.24	-	7.80	+7.7
Ship-Detection	9.18	-	9.91	+7.9
MNIST	-	0.35	0.64	+82.9

engine, the network is organised in 59 tasks. When comparing that number with the results of Fig.4.2, the execution almost matches that of the overhead. As a result, it is clear that this is perhaps the worst case scenario for te proposed framework, but yet again it manages to execute the network without exceeding the 1ms mark.

4.4 Devopment Effort

A significant yet difficult to measure aspect of the *ParalOS* framework is the reduction of the development effort. The contribution of the framework is evaluated in terms of both experienced developers that transition from bare metal programming and amateur programmers in the VPU domain.

Regarding the experienced developers, considerable decrease in development time and effort is measured. Overall rough estimations suggest a $2 - 3\times$ development acceleration when deploying new algorithms, which directly translates to to faster *time-to-market*. The gain mostly originates from i) the easier Design Space Exploration, ii) the readily available functions of the *ParalOS: Low Level Segment* , and iii) the high level system configuration. Equally important is the reduction in *Software Maintainability* as the *ParalOS: High Level Segment* allows for less error-prone programs due to its simpler programming paradigm. Finally *ParalOS* exhibits negligible cost when porting the same application from one VPU to another.

Regarding newcomers to the VPU world, the learning curve is much smother than the one provided by the default vendor development kit. The multiple abstraction layers and the transparent Graph API, provide an interface that the developers are more familiar with. In particular since the parallelisation scheme is conceptually close to the use of threading, developers can better utilise their skill-set without worrying fot the overwhelming details of the platform like cache coherency.

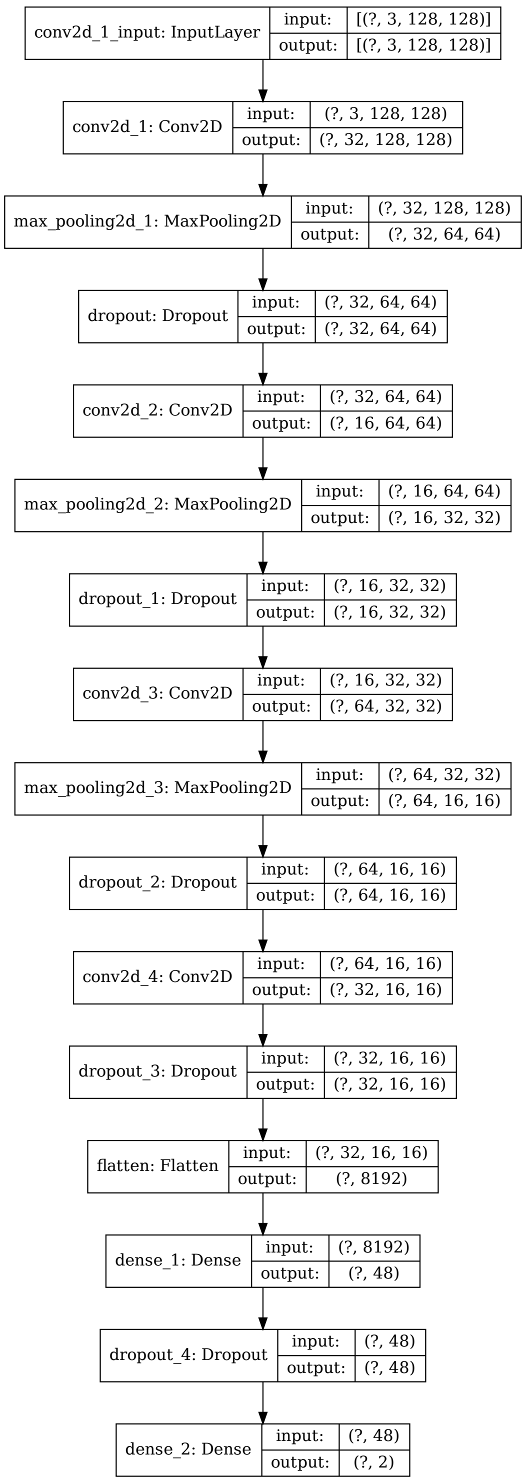

conv2d_1_input: InputLayer | input: [(?, 3, 128, 128)] | output: [(?, 3, 128, 128)]
conv2d_1: Conv2D | input: (?, 3, 128, 128) | output: (?, 32, 128, 128)
max_pooling2d_1: MaxPooling2D | input: (?, 32, 128, 128) | output: (?, 32, 64, 64)
dropout: Dropout | input: (?, 32, 64, 64) | output: (?, 32, 64, 64)
conv2d_2: Conv2D | input: (?, 32, 64, 64) | output: (?, 16, 64, 64)
max_pooling2d_2: MaxPooling2D | input: (?, 16, 64, 64) | output: (?, 16, 32, 32)
dropout_1: Dropout | input: (?, 16, 32, 32) | output: (?, 16, 32, 32)
conv2d_3: Conv2D | input: (?, 16, 32, 32) | output: (?, 64, 32, 32)
max_pooling2d_3: MaxPooling2D | input: (?, 64, 32, 32) | output: (?, 64, 16, 16)
dropout_2: Dropout | input: (?, 64, 16, 16) | output: (?, 64, 16, 16)
conv2d_4: Conv2D | input: (?, 64, 16, 16) | output: (?, 32, 16, 16)
dropout_3: Dropout | input: (?, 32, 16, 16) | output: (?, 32, 16, 16)
flatten: Flatten | input: (?, 32, 16, 16) | output: (?, 8192)
dense_1: Dense | input: (?, 8192) | output: (?, 48)
dropout_4: Dropout | input: (?, 48) | output: (?, 48)
dense_2: Dense | input: (?, 48) | output: (?, 2)

Figure 4.10: Ship Detector Architecture

Chapter 5

Conclusion and Future Work

5.1 Future Work

5.1.1 Hardware Accelerator Support

The current version of *ParalOS* only support *SHAVE* worker types. This does not allow for maximum efficiency of the SoC and total exploitation of its heterogeneity. A future research direction is the integration of the hardware filters into the worker infrastructure, along with necessary modifications of the scheduler and dispatcher.

5.1.2 Validation and Optimisations

The pursuit of yet another optimisation is never ending. Every year decades of worthy publications regarding scheduling and ipc are presented.

5.1.3 Porting to Different Platforms

The Myriad Family of devices are not the only VPUs available. Attractive alternatives based on the risc-V architecture like the GAP8 AI accelerator are targets for a future porting of the framework.

5.1.4 Source to Source Compilation

A very interesting future research direction is the development of a source to source compiler. Essentially this will be a tool, that given a source code in C/C++, with some pragmas and directives, it will automatically generate optimised code for the Myriad family of VPU's using the *ParalOS* framework.

5.1.5 Fault Tolerance and Mitigation Techniques

VPUs manage to fit very high performance in an quite small power envelope. This makes them an attractive choice for the ADAS and autonomous navigation market. These application however, require strict deterministic operation and correctness guarantees.

Algorithms and methods applicable to *ParalOS* may be examined in order to mitigate errors and offer fault tolerance.

5.2 Thesis Conclusion

In this thesis the development and evaluation of , *ParalOS* framework is presented. It is a Software Framework for accelerating the development on extremely Heterogeneous VPUs, while still allowing the exploitation of their full Hardware potential. All the modules of the framework were fine tuned for heterogeneous embedded architectures and are organised into two segments. *ParalOS: High Level Segment* offers an intuitive Graph API for application development, a dynamic task scheduler & transparent system configuration, IO standardisation and visual profiling. *ParalOS: Low Level Segment* features a novel scratchpad memory management scheme and decentralised inter-process communication techniques.

The experimental results reveal limited performance overhead vs manually customised DSE implementations, which can be regarded as a small trade-off towards significant time-to-market improvement and software maintanability. Overall, *ParalOS* achieves up to 4.2× better performance for content-dependent applications, whereas individual sub-modules deliver considerable gains in SPM space utilisation and communication time vs the default or well-established solutions.

5.3 Publications

- A. Kyriakos, E. Papatheofanous, B. Charalampos, **E. Petrongonas**, D. Soudris and D. Reisis, *"Design and Performance Comparison of CNN Accelerators Based on the Intel Movidius Myriad2 SoC and FPGA Embedded Prototype"*, 2019 International Conference on Control, Artificial Intelligence, Robotics & Optimization (ICCAIRO), Athens, Greece, 2019, pp. 142-147, doi: 10.1109/ICCAIRO47923.2019.00030.
  A collaboration between NTUA and NKUA comparing VPUs with FPGAs

- **E. Petrongonas**, V. Leon, G. Lentaris and D. Soudris, *ParalOS: A Scheduling & Memory Management Framework for Heterogeneous VPUs*, 2021 Design Automation and Testing in Europe (DATE) conference. [submitted]
  ParalOS Framework Presentation

- V. Leon, T. Paparouni, **E. Petrongonas**, D. Soudris, K. Pekmestzi, *Improving Power of DSP and CNN Hardware Accelerators using Approximate Floating-Point Multipliers*, ACM Transactions On Embedded Computing Systems. [under major revision]
  Approximate Computing on Convolutional Neural Network applications.

- V. Leon, G. Lentaris, **E. Petrongonas**, D. Soudris, G.Furano, A. Tavoularis and D. Moloney, *Improving Performance-Power-Programmability in Space Avionics with*

Edge Devices: VBN on Myriad2 SoC, ACM Transactions On Embedded Computing Systems. [under major revision]
A collaboration between NTUA, ESA and Intel, exploring the Myriad 2 VPU Potential for Space Applications.

5.4 The End of a Journey

The conclusion of this thesis marks also the conclusion of a difficult yet exciting 6 year journey which have taught me invaluable work and life lessons. Some of them I would like to share.

In a more technical perspective, I have valued the importance of soft skills and communication, which in some cases eclipses that of the actual product. If its purpose (research or product-wise) is not clear enough and easily understood, it will most probably be discarded. Another valuable lesson, that comes with the over 20,000 lines of code of *ParalOS* is the significance of Documentation, when expecting to share your work with others. During the *ParalOS* development I came across various bugs in Intel's code. My first instinct was to wonder *"How Can Intel Make Mistakes"*. To this day, I believe that this was the point where I officially transitioned from a student to an engineer, because I demystified the industry. Everything is built by engineers just like us and everyone is capable of making the same mistakes.

"Fortune favours the Bold" is a life stance rather than a motto. In my experience, I have gained more benefits, by being determined rather than inactive, as sometimes the fastest road is the least paved one. I firmly believe that it is better to try something and fail rather than regret not attempting it in the future. A-posteriori the most pivotal point in my *student-career* was an email to the head of the microlab, Prof. Soudris, expressing my desire to learn, two years earlier than the academic norm would suggest. He embraced it and the rest is history... Finally, I can not stress enough the importance of developing a personality that has more dimensions than the engineering one. Try new things, contribute, socialise and overall become better individuals and better members of the society, as in the end, we are humans after all ...

Bibliography

[1] D. Amodei and D. Hernandez, "AI and compute," *https://openai.com/blog/ai-and-compute/*, 2018.

[2] J. Shalf, "The future of computing beyond moore's law," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190061, 2020.

[3] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe, "Myriad 2: Eye of the computational vision storm," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, Aug 2014, pp. 1–18.

[4] Intel Movidius, "Myriad X press release," Aug 2017. [Online]. Available: https://newsroom.intel.com/news/intel-unveils-neural-compute-engine-movidius-myriad-x-vpu-unleash-ai-edge/

[5] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "Gap-8: A risc-v soc for ai at the edge of the iot," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–4.

[6] Intel, "Distribution of OpenVINO™ toolkit for linux with fpga support," https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.

[7] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: a new dynamic memory allocator for real-time systems," in *16th Euromicro Conf. on Real-Time Systems (ECRTS)*, 2004, pp. 79–88.

[8] G. Lentaris, K. Maragos, I. Stratakos, L. Papadopoulos, O. Papanikolaou, D. Soudris, M. Lourakis, X. Zabulis, D. Gonzalez-Arjona, and G. Furano, "High-performance embedded computing in space: Evaluation of platforms for vision-based navigation," *Journal of Aerospace Information Systems*, vol. In press, 02 2018.

[9] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan. 2019.

[10] N. Thompson and S. Spanuth, "The decline of computers as a general purpose technology: Why deep learning and the end of moore's law are fragmenting computing," *SSRN Electronic Journal*, Nov 2018.

[11] C. E. Leiserson *et al.*, “There’s plenty of room at the top: What will drive computer performance after moore’s law?” *Science*, vol. 368, no. 6495, 2020.

[12] Intel-Movidius, “Myriad 2 ma2x5x vision processor,” https://movidius-uploads.s3.amazonaws.com/1532512604-1503680554-2016-12-12_VPU_ProductBrief.pdf, accessed: 16-10-2019.

[13] L. Dagum and R. Menon, “Openmp: an industry standard api for shared-memory programming,” *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[14] S. Huss-Lederman, B. Gropp, A. Skjellum, A. Lumsdaine, B. Saphir, J. Squyres *et al.*, “Mpi-2: Extensions to the message passing interface,” *University of Tennessee, available online at http://www. mpiforum. org/docs/docs. html*, 1997.

[15] Intel, “Distribution of OpenVINO™ toolkit for linux with fpga support,” https://docs.openvinotoolkit.org/latest/index.html.

[16] Arm Ltd., “CMSIS,” https://developer.arm.com/tools-and-software/embedded/cmsis, accessed: 11-08-2020.

[17] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*, 1st ed. Chapman & Hall/CRC, 2015.

[18] R. Barry, “Mastering the freertos real time kernel,” *Real Time Engineers Ltd*, 2016.

[19] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo, “An experimental study on memory allocators in multicore and multithreaded applications,” in *12th Int’l Conf. on Parallel and Distributed Computing, Applications and Technologies*, 2011, pp. 92–98.

[20] E. K. Lee and C. U. Martel, “When to use splay trees,” *Software: Practice and Experience*, vol. 37, no. 15, pp. 1559–1575, 2007.

[21] G. Barootkoob, E. M. Khaneghah, M. Sharifi, and S. L. Mirtaheri, “Parameters affecting the functionality of memory allocators,” in *2011 IEEE 3rd International Conference on Communication Software and Networks.* IEEE, 2011, pp. 499–503.

[22] M. S. Johnstone and P. R. Wilson, “The memory fragmentation problem: Solved?” in *Int’l Symposium on Memory Management*, 1998, p. 26–36.

[23] M. Lourakis and X. Zabulis, “Model-based visual tracking of orbiting satellites using edges,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, 2017, pp. 3791–3796.

[24] F. Tsimpourlas, L. Papadopoulos, A. Bartsokas, and D. Soudris, “A design space exploration framework for convolutional neural networks implemented on edge devices,”

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 11, pp. 2212–2221, 2018.

[25] A. Kyriakos, E.-A. Papatheofanous, B. Charalampos, E. Petrongonas, D. Soudris, and D. Reisis, "Design and performance comparison of cnn accelerators based on the intel movidius myriad2 soc and fpga embedded prototype," in *2019 International Conference on Control, Artificial Intelligence, Robotics & Optimization (ICCAIRO)*. IEEE, 2019, pp. 142–147.

[26] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[27] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[28] R. Sites, "It's the memory, stupid," *Microprocessor Report*, vol. 10, no. 10, pp. 2–3, 1996.

[29] A. Shan, "Heterogeneous processing: a strategy for augmenting moore's law," *Linux Journal*, vol. 2006, no. 142, p. 7, 2006.

[30] R. Koduri, "No transistor left behind," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, [keynote].

[31] R. P. Feynman, "There's plenty of room at the bottom," *California Institute of Technology, Engineering and Science magazine*, 1960.

[32] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the programmability gap in heterogeneous-isa platforms," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741962

[33] N. Wirth, "A plea for lean software," *Computer*, vol. 28, no. 2, pp. 64–68, 1995.

[34] European Space Agency and CERN, "Esa myriad 2 radiation tests," https://m.esa.int/Enabling_Support/Space_Engineering_Technology/ESA_team_blasts_Intel_s_new_AI_chip_with_radiation_at_CERN, accessed: 16-10-2019.

[35] "RTEMS distribution Homepage," www.rtems.org, accessed: 17-10-2019.

[36] Cobham Gaisler, "Leon 4 processor," https://www.gaisler.com/index.php/products/processors/leon4, accessed: 17-10-2019.

[37] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, Mar 2015.

[38] P. Odysseas, "Implementation of computer vision algorithms on embedded architectures," http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/13305, accessed: 17-10-2019.

[39] Intel Movidius Ltd, *Movidius Myriad2 Development Kit: Programmer's Guide*, under non-disclosure licence.

[40] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "Pulp: A parallel ultra low power platform for next generation iot applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–39.

[41] A. Loquercio, A. I. Maqueda, C. R. Del-Blanco, and D. Scaramuzza, "Dronet: Learning to fly by driving," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 1088–1095, 2018.

[42] P. H. Becker, J. D. Souza, and A. C. Beck, "Tuning the isa for increased heterogeneous computation in mpsocs," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1722–1727.

[43] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, K. Gulati, L. Benini, and D. Wentzlaff, "Byoc: A "bring your own core" framework for heterogeneous-isa research," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 699–714. [Online]. Available: https://doi.org/10.1145/3373376.3378479

[44] A. Munera, S. Royuela, and E. Quiñones, "Towards a qualifiable openmp framework for embedded systems," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, p. 903–908.

[45] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[46] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Application acceleration on fpgas with ompss@ fpga," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 70–77.

[47] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide.* Pearson Education, 2011.

[48] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[49] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.

[50] O. Salvador and D. Angolini, *Embedded Linux Development with Yocto Project.* Packt Publishing Ltd, 2014.

[51] “The everything-is-a-file principle (linus torvalds),” accessed: 11-08-2020. [Online]. Available: https://yarchive.net/comp/linux/everything_is_file.html

[52] Intel Movidius Ltd, *MDK-MA2x5x: MV0212 UserManual*, under non-disclosure licence.

[53] O. Deniz, N. Vallez, J. Espinosa-Aranda, J. Rico-Saavedra, J. Parra-Patino, G. Bueno, D. Moloney, A. Dehghani, A. Dunne, A. Pagani, and et al., “Eyes of things,” *Sensors*, vol. 17, no. 5, p. 1173, May 2017. [Online]. Available: http://dx.doi.org/10.3390/s17051173

[54] J. Beningo, *HAL Design for GPIO.* Berkeley, CA: Apress, 2017, pp. 167–200. [Online]. Available: https://doi.org/10.1007/978-1-4842-3297-2_7

[55] Intel Movidius Ltd, *Movidius Myriad2 2450 Databook*, under non-disclosure licence.

[56] “RTEMS Networking Guide,” Nov 2016. [Online]. Available: https://docs.rtems.org/releases/rtems-docs-4.11.0/networking.html

[57] V. Suhendra, C. Raghavan, and T. Mitra, “Integrated scratchpad memory optimization and task scheduling for mpsoc architectures,” in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 401–410. [Online]. Available: https://doi.org/10.1145/1176760.1176809

[58] S. Steinke, L. Wehmeyer, Bo-Sik Lee, and P. Marwedel, “Assigning program and data objects to scratchpad for energy reduction,” in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 409–415.

[59] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri, “A post-compiler approach to scratchpad mapping of code,” in *Proceedings of the 2004 international*

conference on Compilers, architecture, and synthesis for embedded systems, 2004, pp. 259–267.

[60] A. Janapsatya, A. Ignjatović, and S. Parameswaran, “A novel instruction scratchpad memory optimization method based on concomitance metric,” in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, 2006, pp. 612–617.

[61] S. Udayakumaran and R. Barua, “Compiler-decided dynamic memory allocation for scratch-pad based embedded systems,” in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 276–286.

[62] M. Verma, L. Wehmeyer, and P. Marwedel, “Dynamic overlay of scratchpad memory for energy minimization,” in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004, pp. 104–109.

[63] K. Bai and A. Shrivastava, “Heap data management for limited local memory (llm) multi-core processors,” in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp. 317–325.

[64] V. Venkataramani, M. C. Chan, and T. Mitra, “Scratchpad-memory management for multi-threaded applications on many-core architectures,” *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 1, Feb. 2019.

[65] L. Alvarez *et al.*, “Runtime-guided management of scratchpad memories in multicore architectures,” in *Int’l Conf. on Parallel Architecture and Compilation (PACT)*, 2015, pp. 379–391.

[66] D. Lea and W. Gloger, “A memory allocator,” http://web.mit.edu/sage/export/singular-3-0-4-2+20080405/omalloc/Misc/dlmalloc/malloc.ps.gz, 1996.

[67] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris, “Mitigating memory-induced dark silicon in many-accelerator architectures,” *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 136–139, 2015.

[68] Xieqing, “xieqing/red-black-tree,” https://github.com/xieqing/red-black-tree.

[69] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* ” O’Reilly Media, Inc.”, 2007.

[70] V. Venkataramani, A. Pathania, and T. Mitra, “Unified thread-and data-mapping for multi-threaded multi-phase applications on spm many-cores,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, p. 1496–1501.

[71] Y. Xiao, S. Nazarian, and P. Bogdan, “Self-optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1416–1427, 2019.

[72] A. Edun, R. Vazquez, A. Gordon-Ross, and G. Stitt, "Dynamic scheduling on heterogeneous multicores," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1685–1690.

[73] Google, "Flatbuffers," https://github.com/google/flatbuffers, Oct. 2020.

[74] T. E. Oliphant, *A guide to NumPy.* Trelgol Publishing USA, 2006, vol. 1.

[75] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[76] Intel Movidius Ltd, *Movidius MyriadX Development Kit: Programmer's Guide*, under non-disclosure licence.

[77] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 440–451.

[78] L. Puglia, M. Ionica, G. Raiconi, and D. Moloney, "Passive dense stereo vision on the myriad2 vpu," in *2016 IEEE Hot Chips 28 Symposium (HCS).* IEEE Computer Society, 2016, pp. 1–5.

[79] L. Martin, "Joint strike fighter air vehicle c++ coding standards for the system development and demonstration program," 2005.

[80] C. MISRA, "2008 guidelines for the use of the c++ language in critical systems, june 2008," 2008.

[81] AUTOSAR, "Guidelines for the use of the c++14 language in critical and safety-related systems," https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf, 2017.

[82] T. Foivos, "Resource management techniques for embedded architectures executing deep neural networks," 2018. [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/13775

[83] J. M. Kelsey, J. Byrne, M. Cosgrove, S. Seereeram, and R. K. Mehra, "Vision-based relative pose estimation for autonomous rendezvous and docking," in *2006 IEEE Aerospace Conference*, 2006, pp. 20 pp.–.

[84] G. Lentaris, I. Stratakos, I. Stamoulias, K. Maragos, D. Soudris, M. Lourakis, X. Zabulis, and D. Gonzalez-Arjona, "Project hipnos: Case study of high performance avionics for active debris removal in space," in *IEEE Computer Society Annual Symposium on VLSI*, 07 2017, pp. 350–355.

[85] V. Leon, G. Lentaris, E. Petrongonas, D. Soudris, G. Furano, A. Tavoularis, and D. Moloney, “Improving performance-power-programmability in space avionics with edge devices: VBN on Myriad2 SoC,” *ACM Transactions on Embedded Computing Systems (TECS)*, pp. 1–22, 2021.

[86] J. Canny, “A Computational Approach to Edge Detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986.

[87] G. Giuffrida, L. Diana, F. de Gioia, G. Benelli, G. Meoni, M. Donati, and L. Fanucci, “Cloudscout: A deep neural network for on-board cloud detection on hyperspectral images,” *Remote Sensing*, vol. 12, no. 14, p. 2205, 2020.

[88] C. Marantos, N. Karavalakis, V. Leon, V. Tsoutsouras, K. Pekmestzi, and D. Soudris, “Efficient support vector machines implementation on intel/movidius myriad 2,” in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2018, pp. 1–4.

[89] G. Dinelli, G. Meoni, E. Rapuano, G. Benelli, and L. Fanucci, “An fpga-based hardware accelerator for cnns using on-chip memories only: Design and benchmarking with intel movidius neural compute stick,” *International Journal of Reconfigurable Computing*, vol. 2019, 2019.

[90] V. Leon, S. Mouselinos, K. Koliogeorgi, S. Xydis, D. Soudris, and K. Pekmestzi, “A tensorflow extension framework for optimized generation of hardware cnn inference engines,” *Technologies*, vol. 8, p. 6, 01 2020.

[91] S. Mouselinos, V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi, “Tf2fpga: A framework for projecting and accelerating tensorflow cnns on fpga platforms,” in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2019, pp. 1–4.

[92] A. Mpartsokas, “ Υλοποίηση έντονων υπολογιστικά δικτύων σε ενσωματωμένες αρχιτεκτωνικές με περιορισμένους πόρους. ,” 2018.

[93] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[94] Ξύγκης Αθανάσιος, “ Υλοποίηση συνελικτικών δικτύων σε ενσωματωμένες αρχιτεκτονικές.” 2017.

[95] Rhammell, “Ships in satellite imagery,” https://www.kaggle.com/rhammell/ships-in-satellite-imagery, Jul 2018. [Online]. Available: https://www.kaggle.com/rhammell/ships-in-satellite-imagery