Ατομική Διπλωματική Εργασία

Data-Driven Multithreading για Many-Core Αρχιτεκτονικές: TFluxSCC

Γιάννος Στυλιανού

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2012

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Data-Driven Multithreading για Many-Core Αρχιτεκτονικές: TFluxSCC

Γιάννος Στυλιανού

Επιβλέπων Καθηγητής
Pedro Trancoso

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2012

Ευχαριστίες

Με την περάτωση της παρούσης εργασίας να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα καθηγητή μου κ. Pedro Trancoso, πρωτίστως για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο θέμα, και ακολούθως για την υποστήριξη και καθοδήγηση που μου προσέφερε όλο αυτό το διάστημα. Επίσης, τις πιο θερμές μου ευχαριστίες στον διδακτορικό φοιτητή Ανδρέα Διαβαστό για την συνεισφορά του και τον υπερπολύτιμο χρόνο που μου αφιέρωσε, καθώς και στα υπόλοιπα μέλη του CASPER group, υποψήφιο διδάκτορα Παναγιώτη Πετρίδη και διδακτορικό φοιτητή Κωνσταντίνο Χριστοφή, για τις υποδείξεις και συμβουλές τους.

Θερμότατες ευχαριστίες θα πρέπει οπωσδήποτε να δοθούν και στους γονείς μου και γενικότερα στην οικογένειά μου για την όλη στήριξη και συμπαράστασή τους. Είναι ιδιαίτερα κοπιαστικό να διεκπεραιώνεις ένα πολύ σημαντικό έργο ενώ ταυτόχρονα να πρέπει να αντεπεξέρχεσαι και στις υποχρεώσεις από τον προσωπικό σου χώρο. Κάπου εδώ δράττω την ευκαιρία να πω κι ένα μεγάλο ευχαριστώ στα παιδιά της παρέας για την σύμπνοια και την κατανόηση.

Περίληψη

Η Intel, μέσα στα πλαίσια της έρευνας για την επεκτασιμότητα των πολυπύρηνων επεξεργαστών (multicores), έχει δημιουργήσει τον πειραματικό επεξεργαστή Single-chip Cloud Computer (SCC). Ο SCC έχει κατασκευαστεί για ερευνητικούς σκοπούς ούτως ώστε να μελετηθεί αν μπορούν οι μελλοντικοί επεξεργαστές να επεκταθούν σε 100 ή περισσότερα cores με αντίστοιχη διατήρηση της κατανάλωσης ενέργειας (power consumption) σε χαμηλά επίπεδα. Πρόκειται για ένα σύστημα που υλοποιεί ένα σύμπλεγμα (cluster) 48 πυρήνων οι οποίοι διασυνδέονται και επικοινωνούν μεταξύ τους με ένα τρόπο πανομοιότυπο με αυτόν πολλών υπολογιστών σε ένα δίκτυο. Ο επεξεργαστής αυτός έχει τη δυνατότητα να ρυθμίζει ανάλογα την συχνότητα (frequency) και την τάση (voltage) του ηλεκτρικού ρεύματος σε κάθε πυρήνα, ή ομάδα πυρήνων, ξεχωριστά. Για τον προγραμματισμό του χρησιμοποιείται συγκεκριμένη βιβλιοθήκη η οποία εμπεριέχεται σε ειδική διεπαφή (interface), το RCCE API.

Η πλατφόρμα TFlux έχει ως στόχο την εκτέλεση προγραμμάτων χρησιμοποιώντας το Data-Driven Multithreading (DDM) μοντέλο, το οποίο κατά την χρήση του επιδιώκει να πετύχει όσο το δυνατό υψηλότερο βαθμό παραλληλισμού. Πρόκειται για μια πλατφόρμα ανεξάρτητη της αρχιτεκτονικής και του λειτουργικού συστήματος και βασίζει τη λειτουργία της σ' ένα προ-επεξεργαστή (Preprocessor), τον TFlux C Preprocessor, ο οποίος αναλύοντας τα DDM προγράμματα παράγει αντίστοιχο C κώδικα ικανό για να εκτελεστεί σε οποιοδήποτε κοινό σύστημα. Για την χρονοδρομολόγηση (scheduling) των υπολογισμών κατά την παράλληλη εκτέλεση χρησιμοποιείται ξεχωριστή μονάδα συγχρονισμού, η Thread Synchronization Unit (TSU), υλοποιημένη είτε σε λογισμικό (software) είτε σε υλικό (hardware), και συγκεκριμένα σε *Simics* simulator. Στην εργασία αυτή ασχοληθήκαμε μόνο με το *TFluxSoft*, μια συγκεκριμένη έκδοση της πλατφόρμας TFlux, κατά την οποία η TSU είναι υλοποιημένη σε λογισμικό (software).

Ακολουθώντας διαδικασίες ανάστροφης μηχανίκευσης ή αποσυμπίλισης (reverse engineering) μεταφέραμε την πλατφόρμα TFluxSoft στον Intel SCC καταφέρνοντας

την εκτέλεση DDM εφαρμογών στον εν λόγω επεξεργαστή. Αρχικά, αναλύσαμε και επεξεργαστήκαμε τον TFlux κώδικα που παράγεται από τα DDM προγράμματα και τον προσαρμόσαμε με τα απαιτούμενα macros που προβλέπονται από το RCCE API ώστε να εκτελεστεί σε ένα μόνο πυρήνα του SCC. Στη συνέχεια, βασισμένοι στην ιδέα της ταύτισης των TFlux kernels με τους SCC cores, προσθέσαμε κάποια περαιτέρω στοιχεία στον κώδικα και εφαρμόσαμε τις απαραίτητες τροποποιήσεις που χρειαζόταν το TSU. Οι τροποποιήσεις αυτές αφορούσαν κυρίως την επικοινωνία μεταξύ των πυρήνων, έτσι ώστε να μπορεί ένας TFlux κώδικας να εκμεταλλευτεί όλους τους πυρήνες του SCC κατά την εκτέλεσή του. Τέλος, επεκτείναμε τον υφιστάμενο Preprocessor του TFlux ώστε να παράγει ο ίδιος τον κατάλληλο κώδικα που θα προορίζεται για εκτέλεση στον συγκεκριμένο επεξεργαστή. Δεν έχει γίνει καμιά αλλαγή στα υφιστάμενα TFlux directives. Πήραμε τις απαραίτητες μετρήσεις και μελετήσαμε την επίδοση και τους τρόπους με τους οποίους μπορεί να βελτιστοποιηθεί περαιτέρω. Πιο κάτω παρουσιάζονται τα αποτελέσματα της εκτέλεσης του TFlux στον SCC, καθώς αυξάνεται ο αριθμός των πυρήνων που χρησιμοποιούνται, συγκριτικά με την αντίστοιχη σειριακή εκτέλεση .

Ο στόχος και κατ' επέκταση η συνεισφορά της εργασίας αυτής είναι η δημιουργία ενός εργαλείου για ανάπτυξη DDM εφαρμογών πάνω σε συμπλεγματική (clustered) many-core αρχιτεκτονική (όπως ο SCC) χρησιμοποιώντας την κοινή (shared) μνήμη του SCC. Αυτό επιτεύχθηκε αναβαθμίζοντας την ήδη υπάρχουσα πλατφόρμα TFlux. Προσδοκία μας είναι η δυνατότητα για χρήση ενός προγραμματιστικού μοντέλου που να μπορεί να εκμεταλλευτεί το μεγάλο αριθμό πυρήνων και τον υψηλό βαθμό παραλληλισμού που μπορούν να προσφέρουν. Το DDM μοντέλο είναι ένα τέτοιο προγραμματιστικό μοντέλο το οποίο μπορεί να προσφέρει τη δυνατότητα στον προγραμματιστή να εξάξει το μεγαλύτερο ποσοστό παραλληλισμού σε ένα πρόγραμμα.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Παράλληλη Επεξεργασία και Αρχιτεκτονικές Επεξεργαστών

Η αρχική φιλοσοφία στην κατασκευή επεξεργαστών βασιζόταν στον κλασικό κανόνα πως αυξάνοντας τη συχνότητα καταφέρνουμε να έχουμε ένα επεξεργαστή ο οποίος μπορεί να εκτελεί περισσότερες πράξεις ανά μονάδα χρόνου, με αποτέλεσμα να επιτυγχάνεται βελτίωση της επίδοσης. Η αύξηση της συχνότητας όμως απαιτεί περισσότερη ενέργεια και επιφέρει αντίστοιχα αύξηση της θερμοκρασίας του συστήματος με λογικό επακόλουθο την αύξηση του κόστους ψύξης του εκάστοτε συστήματος. Αυτός είναι και ο κυριότερος λόγος που η σκέψη αυτή αποτελεί παρελθόν στις μέρες μας αφού οι συχνότητες των απλών επεξεργαστών φράσσονται από το γνωστό “power wall” και δεν μπορούν να αυξηθούν περαιτέρω. Έτσι λοιπόν οι εταιρίες παραγωγής επεξεργαστών, αναγκασμένες να

εγκαταλείψουν τον στερεότυπο κανόνα αύξησης της συχνότητας, επινόησαν μια νέα ιδέα η οποία εκμεταλλεύεται τη μείωση του μεγέθους των ηλεκτρονικών κυκλωμάτων και επικεντρώνεται στην τοποθέτηση πολλών πανομοιότυπων ή διαφορετικών πυρήνων πάνω στο ίδιο ολοκληρωμένο κύκλωμα (chip). Αυτό σαφέστατα επιτυγχάνει βελτίωση της επίδοσης αφού πλέον έχουμε περισσότερες υπολογιστικές μονάδες αλλά ταυτόχρονα καταφέρνει να λύσει και πολλά προβλήματα που είχαμε προηγουμένως όπως είναι η υψηλή κατανάλωση ενέργειας και τα συνεπακόλουθα της. Συνδυάζοντας, λοιπόν, κλασικές τεχνικές που χρησιμοποιούνταν και προηγουμένως, όπως multiple instruction issue, prefetching, out-of-order execution, branch prediction, loop unrolling, register renaming, dynamic software scheduling και διάφορες άλλες, οι πολυπύρηνοι επεξεργαστές μπορούν να πετύχουν ψηλές επιδόσεις ακόμα και με χαμηλότερες συχνότητες με διατήρηση της κατανάλωσης ενέργειας σε χαμηλά επίπεδα.

Σήμερα, η νέα αυτή φιλοσοφία έχει εδραιωθεί και αποτελεί τη βάση στην κατασκευή, πολυπύρηνων πλέον, επεξεργαστών. Ήδη σταδιακά μεταφερόμαστε από τους multi-cores στους many-cores επεξεργαστές οι οποίοι περιλαμβάνουν δεκάδες, και προσεχώς εκατοντάδες πυρήνες. Προκύπτουν όμως νέες προκλήσεις κυρίως όσον αφορά τη μνήμη και την επικοινωνία μεταξύ των υπολογιστικών μονάδων. Η κάθε κατασκευαστική εταιρία ακολουθεί τη δική της φιλοσοφία όσον αφορά την αρχιτεκτονική του επεξεργαστή επιλέγοντας όμως σε ποια βάση μοντέλου μνήμης θα κινηθεί. Οι πυρήνες στον επεξεργαστή είτε θα διαμοιράζονται την ίδια μνήμη, το λεγόμενο “shared-memory” μοντέλο, είτε θα έχει ο καθένας την δική του μνήμη και θα επικοινωνούν με ανταλλαγές μηνυμάτων όπως περιγράφει το “distributed-memory” μοντέλο. Σ’ αυτό το μοντέλο μνήμης σημαντικό ρόλο παίζει και ο τρόπος με τον οποίο είναι διασυνδεδεμένοι οι πυρήνες μεταξύ τους. Ο συνδυασμός των δύο μοντέλων είναι εφικτός και ο many-core επεξεργαστής Intel Single-chip Cloud Computer (SCC), στον οποίο αφιερώνουμε εκτενή μελέτη σε επόμενο κεφάλαιο, αποτελεί ένα παράδειγμα τέτοιου συνδυασμού.

1.2 Επεξεργαστής Intel SCC

Ο Intel SCC είναι ένας ερευνητικός μικροεπεξεργαστής (microprocessor) κατασκευασμένος από την Intel που περιέχει 48 πυρήνες (cores) στο ίδιο κύκλωμα (chip). Οι πυρήνες αυτοί είναι οργανωμένοι σε 24 “tiles”, δύο πυρήνες ανά tile, τα

οποία είναι διασυνδεδεμένα με ένα 6x4 δίκτυο πλέγματος (mesh). Κάθε tile μπορεί να έχει διαφορετική συχνότητα (frequency) και κάθε ομάδα (group) των 4 tiles μπορεί να τρέχει με διαφορετική τάση (voltage). Επιπρόσθετα, σε κάθε tile υπάρχει ένας δρομολογητής (router) για την επικοινωνία μεταξύ των πυρήνων προσφέροντας 256GB/s εύρος ζώνης (bandwidth), ενώ στο chip υπάρχουν συνολικά 4 DDR3 διαχειριστές μνήμης (memory controllers) για την διασύνδεση με την DRAM μνήμη. Γενικά, ο SCC είναι ένα κατανεμημένο (distributed) σύστημα γι' αυτό και περιέχει υλικό (hardware) που να υποστηρίζει "message-passing" επικοινωνία, αλλά υπάρχει και ένα μέρος της μνήμης που είναι κοινή (shared) για όλους τους πυρήνες χωρίς ωστόσο να υπάρχει κάποιο πρωτόκολλο για τη συνέπεια της (coherency protocol). Για τον προγραμματισμό εφαρμογών στον Intel SCC χρησιμοποιείται συγκεκριμένο API, το RCCE API. Ο Intel SCC συνοδεύεται και από έναν εξομοιωτή (emulator) που έχει την ικανότητα να αφομοιώνει την αρχιτεκτονική του SCC και να εκτελεί προγράμματα που προορίζονται γι' αυτόν, σε οποιοδήποτε κοινό σύστημα.

1.3 Μοντέλο DDM

Με την εξάπλωση των multicore συστημάτων έχει αποδειχτεί ότι το μοντέλο προγραμματισμού Dataflow Multithreading είναι ένα από τα πιο αποδοτικά σε τέτοιου είδους συστήματα όσον αφορά την επίδοση παράλληλων εφαρμογών. Σύμφωνα με το μοντέλο αυτό, ο κώδικας ενός προγράμματος μοιράζεται σε μικρότερα κομμάτια, τα λεγόμενα *Threads*, από τα οποία το καθένα αποτελείται από μια σειρά εντολών. Αυτά τα κομμάτια κώδικα μπορούν να θεωρηθούν ανεξάρτητα το ένα προς το άλλο και επομένως μπορούν να εκτελούνται παράλληλα, αλλά μόνο όταν τα δεδομένα που χρειάζονται είναι έτοιμα. Δεν πρόκειται να δρομολογηθούν για εκτέλεση εάν τα δεδομένα που απαιτούνται δεν είναι διαθέσιμα. Με αυτό τον τρόπο δημιουργείται ένας γράφος συγχρονισμού (synchronization graph) με τις εξαρτήσεις μεταξύ αυτών των κομματιών-threads. Βάσει αυτού του μοντέλου έχει γίνει ακόμα ένα βήμα παραπέρα και έχει προταθεί το Data-Driven Multithreading (DDM) μοντέλο προγραμματισμού το οποίο έχει την ίδια φιλοσοφία και χρησιμοποιεί ένα thread scheduler για να συγχρονίσει τα threads. Κάθε thread έχει ένα μετρητή (ReadyCounter) ο οποίος αντιπροσωπεύει τον αριθμό των threads από τα οποία περιμένει δεδομένα για να αρχίσει την εκτέλεσή του. Όταν ο μετρητής κάποιου thread φτάσει σε τιμή ίση με 0 τότε το thread αυτό είναι έτοιμο για εκτέλεση. Ο

thread scheduler μπορεί να είναι υλοποιημένος είτε σε λογισμικό (software) είτε σε υλικό (hardware). Το συγκεκριμένο μοντέλο προγραμματισμού χρησιμοποιείται από την πλατφόρμα TFlux [2] που θα περιγράψουμε συνοπτικά παρακάτω αλλά και εκτενέστερα σε ξεχωριστό κεφάλαιο. Υπάρχουν υλοποιημένες διάφορες DDM εφαρμογές που προορίζονται για παράλληλη εκτέλεση. Στη συγκεκριμένη εργασία θα ασχοληθούμε με κάποια από τα DDM benchmarks που έχουν υλοποιηθεί για χάρη προηγούμενων ερευνών στο εργαστήριο μας.

1.4 Φορητή Πλατφόρμα TFlux

Η πλατφόρμα TFlux (Thread Flux) έχει ως στόχο να επιτρέψει στους χρήστες της να αναπτύξουν και να εκτελέσουν DDM εφαρμογές με σχετικά εύκολο τρόπο. Έχει την ικανότητα να εικονικοποιήσει (virtualize) τα χαρακτηριστικά του συστήματος στο οποίο τρέχει, προσφέροντας ένα κοινό προγραμματιστικό μοντέλο ανεξαρτήτως αρχιτεκτονικής. Αυτό πετυχαίνεται από το TFlux Runtime Support που τρέχει ακριβώς πάνω από οποιοδήποτε λειτουργικό σύστημα και αφομοιώνει όλες τις λεπτομέρειες της αρχιτεκτονικής του “host” συστήματος. Επιπλέον, το TFlux περιλαμβάνει ένα Preprocessor και μια μονάδα συγχρονισμού, το Thread Synchronization Unit (TSU). Ο Preprocessor είναι υπεύθυνος για την σάρωση και την ανάλυση των DDM προγραμμάτων που αναπτύσσει ο προγραμματιστής και την παραγωγή του TFlux κώδικα. Παρατηρώντας τον DDM κώδικα που του δόθηκε ως είσοδος, θα εντοπίσει τον αριθμό των kernels που έχει οριστεί από τον χρήστη και στη συνέχεια θα χρησιμοποιήσει το TSU για την δημιουργία του γράφου των εξαρτήσεων μεταξύ τους. Ο κώδικας που παράγεται από τον Preprocessor είναι σε γλώσσα C (ή C++) και μπορεί να μεταγλωττιστεί από οποιοδήποτε κοινό C μεταγλωττιστή (compiler) δημιουργώντας ένα εκτελέσιμο αρχείο που δύναται να τρέξει σε οποιοδήποτε ISA. Για το TFlux έχει ήδη υλοποιηθεί TSU και σε software (το πιο κοινό) αλλά και σε hardware (σε Simics simulator) και πλέον η πλατφόρμα είναι σε θέση να λειτουργήσει σε οποιοδήποτε συνηθισμένο multicore σύστημα.

1.5 Κίνητρο

Ο Intel SCC είναι ένας από τους ιδανικότερους επεξεργαστές στους οποίους μπορεί να μελετηθεί η επίδοση και η επεκτασιμότητα παράλληλων εφαρμογών. Θα

πρέπει, λοιπόν, με κάποιο τρόπο να εκμεταλλευτούμε τους 48 πυρήνες και το σχετικά πολύ γρήγορο δίκτυο διασύνδεσής τους για να εκτελέσουμε εφαρμογές που θα μας δώσουν αντιπροσωπευτικά δείγματα για την επίδοση και τη αλλαγή του χρόνου εκτέλεσης. Η πλατφόρμα TFlux έχει μελετηθεί και είναι αποδεδειγμένο ότι πετυχαίνει πολύ καλές επιδόσεις όταν χρησιμοποιούνται περισσότερες υπολογιστικές μονάδες, δηλαδή kernels, σε κοινά συστήματα με κοινή μνήμη. Θα ήταν πολύ ενδιαφέρον να καταφέρναμε την εκτέλεση αυτής της πλατφόρμας σε μια κατανεμημένης μνήμης αρχιτεκτονική, όπως αυτή του SCC, και να επιδιώξουμε να πετύχουμε αντίστοιχες επιδόσεις. Με τη διεκπεραίωση ενός τέτοιου έργου εξυπηρετείται παράλληλα και ο στόχος που έθεσε η Intel όταν κατασκεύαζε τον ερευνητικό αυτόν επεξεργαστή: να μελετηθεί όσο το δυνατό καλύτερα και να ερευνηθεί το γεγονός αν μπορεί ένα σύστημα με τόσους πολλούς πυρήνες να προσφέρει στη μείωση του χρόνου εκτέλεσης ώστε να πειστούμε ότι αντίστοιχα συστήματα με χιλιάδες πυρήνες θα μας επέφεραν βελτιώσεις στην επίδοση που ποτέ δεν θα μπορούσαμε να φανταστούμε.

1.6 Στόχος και Αναμενόμενα Αποτελέσματα

Ο δικός μας στόχος είναι να καταφέρουμε η πλατφόρμα TFlux να γίνει διαθέσιμη και για δημιουργία και εκτέλεση εφαρμογών στον Intel SCC. Θα πρέπει, λοιπόν, να επεκτείνουμε την πλατφόρμα TFlux ώστε να είναι σε θέση να υποστηρίξει εφαρμογές που προορίζονται για εκτέλεση στον SCC. Η βάση στην οποία θα κινηθούμε είναι η ταύτιση των kernels που χρησιμοποιούνται σε μια DDM εφαρμογή με τους πυρήνες του συστήματος SCC. Αυτό αφορά αποκλειστικά το TSU το οποίο πλέον, με τις απαραίτητες αλλαγές φυσικά, θα φροντίζει να συγχρονίζει τους πυρήνες καθ' όλη τη διάρκεια της εκτέλεσης. Επιπλέον θα πρέπει να προσθέσουμε στον Preprocessor την ικανότητα να παράγει κώδικα για εκτέλεση στον SCC όποτε του ζητηθεί (μέσω παραμέτρων). Ο κώδικας αυτός θα έχει κάποιες ιδιαιτερότητες αφού μιλάμε για κατανεμημένο σύστημα. Επομένως, η επικοινωνία μεταξύ των threads θα βασίζεται σε "message-passing" μοντέλο, κάτι το οποίο θα είναι πολύ διαφορετικό από τις προηγούμενες υλοποιήσεις. Στόχος μας είναι να διευθετήσουμε αυτή η επικοινωνία να γίνεται μέσω του TSU ο οποίος θα είναι σε θέση να γνωρίζει πότε και ποιοι πυρήνες θα πρέπει να ενημερωθούν. Σαν πρώτη υλοποίηση θα εκμεταλλευτούμε την κοινή μνήμη του SCC για να μεταφέρουμε το "soft" version του TFlux πάνω στον SCC. Αυτό θα γίνει κατορθωτό με μερικές

μικρές αλλαγές κυρίως στην δέσμευση των καθολικών (global) δεδομένων της εφαρμογής. Η πρώτη υλοποίηση που επιδιώκουμε είναι βασισμένη στο shared-memory μοντέλο του SCC σύμφωνα με το οποίο τα δεδομένα της εφαρμογής που είναι προς εκτέλεση θα βρίσκονται στην κοινή μνήμη ενώ ο κάθε πυρήνας θα τρέχει το δικό του TSU, του οποίου τα δεδομένα θα μένουν στην προσωπική (private) μνήμη του κάθε πυρήνα.

Αναμένουμε με το τέλος της εργασίας να έχουμε μια λειτουργική υλοποίηση του TFlux για εκτέλεση DDM εφαρμογών στον Intel SCC. Πρωταρχικός μας σκοπός είναι η λύση αυτή να πετυχαίνει ορθές εκτελέσεις και να μην παράγει λανθασμένα αποτελέσματα. Η κάθε εφαρμογή που θα εκτελούμε στον SCC χρησιμοποιώντας το TFlux θα πρέπει να μας επιστρέφει τα ίδια αποτελέσματα με αυτά που θα πάρουμε όταν η ίδια εφαρμογή εκτελείται σε κάποιο άλλο σύστημα. Μετά το πέρας της εργασίας και αφού σιγουρευτούμε ότι έχουμε καταλήξει σε μια ορθή υλοποίηση χωρίς προβλήματα και λανθασμένα αποτελέσματα, θα επικεντρωθούμε στην αύξηση της επίδοσης και στον καθορισμό βελτιστοποιήσεων οι οποίες μπορεί να μας επιφέρουν μια ακόμα πιο αποδοτική εκτέλεση.

1.7 Οργάνωση και Μεθοδολογία

Έχουμε χωρίσει το έργο σε μια σειρά από βήματα με την προϋπόθεση ότι για να προχωρήσουμε στο επόμενο βήμα θα έπρεπε να είχαμε φέρει εις πέρας το προηγούμενό του. Να σημειώσουμε ότι μερικές από αυτές τις διαδικασίες που περιγράφονται στα βήματα ήταν επαναλαμβανόμενες μέχρι και την ολοκλήρωση του έργου αφού ανά πάσα στιγμή μπορεί να προέκυπταν βελτιστοποιήσεις.

Αρχικά έπρεπε να μελετήσουμε τα εγχειρίδια για την αρχιτεκτονική και τον προγραμματισμό του Intel SCC καθώς και τις σχετικές δημοσιεύσεις του MARC της Intel [4, 6, 7, 8, 9, 10, 11, 12] από διάφορα άλλα εργαστήρια και ινστιτούτα που έχουν ασχοληθεί με τον εν λόγω επεξεργαστή. Αναγκαία ήταν και η μελέτη των δημοσιευμένων άρθρων σχετικά με την πλατφόρμα TFlux [2] και του μοντέλου προγραμματισμού DDM [1, 3]. Έπρεπε επίσης να αναλύσουμε τον κώδικα της πλατφόρμας TFlux και κυρίως του TSU αλλά και αυτόν του Preprocessor ο οποίος συνοδεύεται με ένα εγχειρίδιο [17]. Ακολούθως, εκτελέσαμε τα διάφορα DDM benchmarks σε κάποιο κοινό σύστημα ούτως ώστε να δούμε πως συμπεριφέρονται,

δηλαδή τι αποτελέσματα επιστρέφουν και ποίος είναι ο χρόνος εκτέλεσής τους. Το επόμενο βήμα ήταν να μεταφέρουμε αυτούσιο τον TFlux κώδικα μιας οποιασδήποτε DDM εφαρμογής στον SCC και να διαπιστώσουμε ότι πράγματι εκτελείται τουλάχιστον σ' ένα μόνο πυρήνα. Στη συνέχεια τροποποιήσαμε αυτόν τον κώδικα με "hand-coded" αλλαγές, προσθέτοντας και τα RCCE macros, ούτως ώστε η εφαρμογή να διαμοιράζεται στους διαθέσιμους πυρήνες και να εκτελείται με ορθό αποτέλεσμα. Το βήμα αυτό ήταν και το δυσκολότερο, αφού εδώ έπρεπε να γίνουν και οι αλλαγές στο TSU, αλλά και το βασικότερο, αφού μετά από αυτό θα είχαμε ήδη έτοιμο τον κώδικα που επιθυμούσαμε να μας παράξει ο Preprocessor για να εκτελεστεί στον SCC. Επομένως, το μόνο που μας είχε απομείνει να κάνουμε ήταν να ανατρέξουμε στον κώδικα του Preprocessor και να εφαρμόσουμε τις απαραίτητες προσθήκες ούτως ώστε να επεκταθεί και να μπορεί να παράγει τον επιθυμητό κώδικα από μόνος του. Τέλος, αφού βεβαιωθήκαμε ότι και οι υπόλοιπες DDM εφαρμογές είναι εκτελέσιμες στον SCC, τρέξαμε τα διάφορα DDM benchmarks που είχαμε ήδη υλοποιημένα και συλλέξαμε τα αποτελέσματα από τις εκτελέσεις αναλύοντας και παρουσιάζοντας χρόνους εκτέλεσης και επιδόσεις σε γραφικές παραστάσεις.

1.8 Περιγραφή Δομής Ατομικής Διπλωματικής Εργασίας

Η εργασία αυτή είναι χωρισμένη σε 8 κεφάλαια στο καθένα από τα οποία περιγράφεται με λεπτομέρεια ένα μέρος του έργου με το οποίο ασχοληθήκαμε για να φέρουμε εις πέρας την αποστολή μας. Το Κεφάλαιο 1 είναι εισαγωγικό και έχει ως στόχο να δώσει στον αναγνώστη μια πρώτη εικόνα του προβλήματος ώστε να γίνει κατανοητό. Γίνεται μια μικρή αναφορά στους σύγχρονους πολυπύρηνους επεξεργαστές και δίνονται σύντομες περιγραφές του προγραμματιστικού μοντέλου DDM, της φορητής πλατφόρμας TFlux και του επεξεργαστή SCC. Κατόπιν αναφέρεται ο στόχος ο οποίος έχει τεθεί για τη συγκεκριμένη εργασία και γίνεται μια οργάνωση των διαδικασιών που θα ακολουθήσουμε για την εκπλήρωσή του. Στο Κεφάλαιο 2 περιγράφεται η σχετική δουλειά που έχει γίνει για τα θέματα που θα ασχοληθούμε, έτσι όπως τα έχουμε μελετήσει μέσα από τις διάφορες πηγές που αναγράφονται στη βιβλιογραφία. Στο Κεφάλαιο 3 αναλύεται με λεπτομέρεια ο επεξεργαστής Intel SCC αναφέροντας όλα τα χαρακτηριστικά του και δίνοντας έμφαση στις ιδιαιτερότητες του. Παρόμοια αναλυτική περιγραφή γίνεται και στο Κεφάλαιο 4 όπου μελετάται η φορητή πλατφόρμα TFlux με το μοντέλο

προγραμματισμού DDM και τα διάφορα συστατικά της όπως είναι το TSU και ο Preprocessor. Στο Κεφάλαιο 5 αναφέρεται λεπτομερώς η δουλειά που εμείς έχουμε κάνει ώστε να εκτελεστεί επιτυχώς το μοντέλο DDM στον επεξεργαστή SCC με χρήση της πλατφόρμας TFlux. Εδώ, μαζί με άλλα, περιγράφεται η βασική ιδέα που ακολουθήθηκε και οι αλλαγές που έχουν γίνει στον κώδικα του TSU, καθώς και οι προσθήκες που εφαρμόστηκαν στον κώδικα του Preprocessor. Στο Κεφάλαιο 6 γίνεται μια εκτενής αξιολόγηση της εκτέλεσης στον SCC ώστε να διαπιστώσουμε και να βεβαιωθούμε ότι το TFlux εκτελείται με επιτυχία σε όλους διαθέσιμους πυρήνες του επεξεργαστή χωρίς καθόλου σφάλματα και λανθασμένα αποτελέσματα. Επίσης, παρουσιάζονται οι γραφικές παραστάσεις με τα αποτελέσματα που έχουμε πάρει για τους χρόνους εκτέλεσης και τις επιδόσεις των DDM benchmarks. Τέλος, με το Κεφάλαιο 7 κλείνουμε την εργασία αυτή κάνοντας μια αναφορά στα συμπεράσματα και τις γνώσεις που έχουμε αποκομίσει καθ' όλη τη διάρκεια του έργου, καθώς και στη μελλοντική δουλειά που μπορεί να γίνει για τη βελτίωσή του.

Κεφάλαιο 2

Σχετική Δουλειά

Για να είμαστε σε θέση να ξεκινήσουμε την υλοποίηση των στόχων μας έπρεπε πρώτα να μελετήσουμε κάποια σχετικά συγγράμματα που αφορούν τους τομείς με τους οποίους θα ασχοληθούμε, έτσι ώστε να φτιάξουμε μια όσο το δυνατόν πιο πλήρη εικόνα για το δικό μας έργο. Για το λόγο αυτό έχουμε μελετήσει κάποια επιλεκτικά άρθρα με θέματα για τα οποία πρέπει να έχουμε μια πάρα πολύ καλή γνώση προτού ξεκινήσουμε. Τα άρθρα αυτά αφορούν κυρίως μελέτες για την πλατφόρμα TFlux [2,3], το DDM μοντέλο προγραμματισμού [1,3] και τον many-core Intel SCC [4-13], και καταγράφονται στη βιβλιογραφία που βρίσκεται στο τελευταίο τμήμα της συγκεκριμένης μελέτης.

Η όλη ιδέα της πλατφόρμας TFlux βασίζεται στο παράλληλο μοντέλο προγραμματισμού και εκτέλεσης Data-Driven Multithreading (DDM) το οποίο μπορεί να πετύχει κατά μέσο όρο μέχρι και 26 φορές καλύτερη επίδοση (για συγκεκριμένες εφαρμογές της SPLASH-2 suite [1]) από μια σειριακή εκτέλεση. Σύμφωνα με το μοντέλο αυτό [1], οι έννοιες υπολογισμός και συγχρονισμός αποσυνδέονται και πλέον εκτελούνται ασύγχρονα, και χάρη σ' αυτήν την ικανότητα το μοντέλο είναι non-blocking. Η γενική ιδέα του DDM λέει πως κάποιο thread επιτρέπεται να ξεκινήσει την εκτέλεση αν και μόνο αν τα δεδομένα που χρειάζεται είναι έτοιμα και βρίσκονται στη μνήμη. Με αυτόν τον τρόπο οι διάφορες καθυστερήσεις από υπολογισμούς και συγχρονισμούς ελαχιστοποιούνται αλλά ταυτόχρονα δημιουργούνται ακανόνιστες προσβάσεις στη μνήμη με αποτέλεσμα να οδηγούμαστε σε πολύπλοκα μοτίβα πρόσβασης (access patterns). Για να λύσει αυτό

το πρόβλημα το DDM μοντέλο εκτέλεσης χρησιμοποιεί την *CacheFlow πολιτική (policy)* [1], η οποία είναι μια πολιτική διαχείρισης της κρυφής (cache) μνήμης κατά την οποία τα δεδομένα που χρειάζεται κάποιο thread φορτώνονται στην cache προτού το thread δρομολογηθεί για εκτέλεση. Η συγκεκριμένη πολιτική της cache χρησιμοποιεί και κάποιες περεταίρω τεχνικές ούτως ώστε τα δεδομένα να μην απομακρυνθούν από τη μνήμη προτού χρησιμοποιηθούν από τα threads για τα οποία προορίζονται, βοηθώντας έτσι και στην μείωση των cache misses. Με το DDM μοντέλο ο επεξεργαστής «αποκρύπτει» τις μεγάλες καθυστερήσεις αναμονής εκτελώντας κάποιο άλλο κομμάτι δουλειάς και έτσι εκμεταλλεύεται σε μεγάλο βαθμό τον παραλληλισμό της εφαρμογής. Ένα πρόγραμμα γραμμένο σε DDM (χρησιμοποιώντας την πλατφόρμα TFlux) αποτελείται από διάφορα blocks κώδικα, το καθένα από τα οποία περιλαμβάνει διάφορα threads. Κάθε block περιέχει ένα Inlet Thread το οποίο χρησιμοποιείται μαζί με τα υπόλοιπα για να συγχρονίσουν τα blocks μεταξύ τους. Για τον καθορισμό των εξαρτήσεων μεταξύ των threads και το συγχρονισμό, οπωσδήποτε χρειάζεται κάποια μονάδα η οποία θα καθορίζει ποια threads έχουν σειρά για εκτέλεση. Αυτή η μονάδα, συνήθως είναι ένα memory-mapped hardware module το οποίο συνδέεται στο bus του επεξεργαστή και είναι υπεύθυνο για την at-runtime χρονοδρομολόγηση των threads. Στην πλατφόρμα TFlux όπως θα δούμε και πιο κάτω, τον ρόλο αυτής της μονάδας έχει το TSU το οποίο είναι υλοποιημένο και σε hardware αλλά και σε software. Ανά πάσα στιγμή της εκτέλεσης μιας DDM εφαρμογής κάποια από τα blocks αναλαμβάνονται από το TSU ώστε να δρομολογηθούν προς εκτέλεση τα threads που περιλαμβάνουν. Τα threads μέσα σε ένα block εκτελούνται παράλληλα αν επιτρέπεται από το TSU ή ακόμα μπορούν να εκτελεστούν με διαφορετική σειρά από την καθορισμένη. Τα blocks όμως δρομολογούνται και συγχρονίζονται από το TSU σύμφωνα πάντα με την σειρά είναι γραμμένα στον κώδικα. Για τη σημασία και την επίδοση του Data-Driven Multithreading έχει μελετηθεί η εφαρμογή του σε Data-Driven Networks κάποιων workstations από κοινούς επεξεργαστές, όπως ο Intel Pentium, και στους οποίους έχει προστεθεί στο bus του συστήματος το hardware TSU. Από τα αποτελέσματα που πάρθηκαν καταλήγουμε στο συμπέρασμα ότι αυξάνοντας τους κόμβους (nodes) των workstations η επίδοση αυξάνεται. Χαρακτηριστικά, χρησιμοποιώντας το DDM μοντέλο σε 16-nodes συστήματα έχουμε 14.4 φορές καλύτερη επίδοση, ενώ σε 32-nodes συστήματα έχουμε 26 φορές καλύτερη επίδοση.

Εκτενέστερα, έχουν μελετηθεί τα διάφορα συστατικά της πλατφόρμας TFlux όπως είναι το *Runtime Support*, το *TSU* και ο *Preprocessor*. Το Runtime Support [2]

παρέχει κάποιας μορφής εικονικοποίηση (virtualization) του συστήματος πάνω στο οποίο εγκαθίσταται η πλατφόρμα TFlux ώστε να αποκρύπτονται οι λεπτομέρειες των DDM προγραμμάτων καθώς και η υλοποίηση του TSU. Με τον τρόπο αυτό το host σύστημα μπορεί να εκτελεί και να χειρίζεται τις DDM εφαρμογές χωρίς να αντιλαμβάνεται την διαφορετικότητα τους από τις συνηθισμένες εφαρμογές. Ο κυριότερος στόχος του Runtime Support είναι να φορτώνει τα DThreads (Data-Driven Threads) στο TSU με τρόπο που να παρέχεται μια αποδοτική επικοινωνία μεταξύ εφαρμογής και TSU. Η μονάδα TSU [2] με την σειρά της είναι υπεύθυνη να δημιουργήσει ένα γράφο συγχρονισμού (synchronization graph) του οποίου οι κόμβοι αντιπροσωπεύουν τα DThreads και οι ακμές τις εξαρτήσεις μεταξύ τους. Η μονάδα αυτή έχει ήδη υλοποιηθεί και σε λογισμικό (software) αλλά και σε υλικό (hardware). Σύμφωνα με την hardware υλοποίηση, τον ρόλο του TSU έχει μια memory-mapped συσκευή ειδικά κατασκευασμένη γι' αυτόν τον σκοπό, και η οποία είναι συνδεδεμένη πάνω στο *TFluxHard* σύστημα που είναι ένα κοινής (shared) μνήμης Chip Multiprocessor. Για την δική μας δουλεία, θα ασχοληθούμε περισσότερο με την software υλοποίηση του TSU που περιέχεται στο πακέτο *TFluxSoft*, και η οποία είναι συμβατή με οποιοδήποτε επεξεργαστή που υποστηρίζει shared address space και cache coherency πρωτόκολλο. Μια άλλη υλοποίηση που ήδη έχουμε στη διάθεση μας είναι το *TFluxCell* που είναι κι αυτό σε software και είναι κατάλληλο για εκτέλεση εφαρμογών στον επεξεργαστή CELL. Σύμφωνα με αυτήν την υλοποίηση τα DThreads εκτελούνται στα SPE cores ενώ το TSU τρέχει στον PPE core. Στην πλατφόρμα TFlux βασικό ρόλο έχει ο προ-επεξεργαστής Preprocessor, ο οποίος παραλαμβάνει ένα DDM πρόγραμμα και παράγει τον αντίστοιχο C κώδικα. Ο Preprocessor [2,3] εκτελεί δύο σαρώσεις στον DDM κώδικα. Στην πρώτη σάρωση παρατηρεί τον κώδικα, βρίσκει τις εξαρτήσεις και τις παραδίδει στο TSU για να χτίσει τον γράφο. Στην δεύτερη σάρωση δημιουργεί παράλληλα τον C κώδικα που θα μεταγλωττιστεί και θα τρέξει στο σύστημα. Υπάρχει ειδική πρόνοια στον κώδικα του Preprocessor ώστε να δέχεται μέσω παραμέτρων σε ποιο σύστημα πρόκειται να τρέξει η εφαρμογή, ούτως ώστε να παράξει τον κατάλληλο κώδικα. Οι πιθανές παράμετροι μπορούν να απευθυνθούν στις μέχρι στιγμής υλοποιήσεις του TFluxSoft και TFluxHard, στην εκτέλεση στον επεξεργαστή CELL και στην εκτέλεση σε κατανεμημένο (distributed) σύστημα. Ο δικός μας στόχος είναι να δημιουργήσουμε μια επέκταση του Preprocessor ώστε να παράγει κατάλληλο κώδικα για εκτέλεση και στον Intel SCC.

Όπως προαναφέραμε ο επεξεργαστής SCC δεν έχει βγει στην παραγωγή αλλά έχει παραδοθεί κάποιος αριθμός (περίπου 100 στο σύνολο) από την Intel σε διάφορα εργαστήρια για ερευνητικούς σκοπούς. Για την κατανόηση της συμπεριφοράς του SCC κατά την εκτέλεση εφαρμογών, έχουμε μελετήσει διάφορα άρθρα τα οποία έχουν υποβληθεί από διάφορους ερευνητές στο MARC (Many-core Applications Research Community) της Intel σχετικά με την έρευνα για τον SCC [7-13]. Έχουμε μελετήσει τα διάφορα μοντέλα μνήμης που υποστηρίζει ο SCC, shared και distributed, και το πόσο αποδοτικό είναι το καθένα για την εκτέλεση κάποιων εφαρμογών-τεχνικών (πχ LDPC) όταν αυξάνουμε τον αριθμό των πυρήνων που χρησιμοποιούνται [5]. Κατά την εκτέλεση σε κατανεμημένη (distributed) μνήμη έχουμε να αντιμετωπίσουμε το κόστος της ανταλλαγής των μηνυμάτων που απαιτείται για να ενημερώνονται οι πυρήνες για τις αλλαγές στα δεδομένα. Επίσης, συνολικά χρησιμοποιείται περισσότερη μνήμη αφού για τα δεδομένα που είναι global θα πρέπει ο κάθε πυρήνας να δεσμεύσει το δικό του χώρο. Όσον αφορά το μοντέλο κοινής (shared) μνήμης, μπορεί να θεωρηθεί ελαφρά καλύτερο αφού εκμεταλλεύεται την κοινή μνήμη για τα καθολικά (global) δεδομένα και δεν απαιτούνται ανταλλαγές μηνυμάτων για τις αλλαγές πάνω σε αυτά. Το ιδανικό για μια καλή επίδοση και επίτευξη επεκτάσιμου παραλληλισμού είναι ο συνδυασμός των δυο μοντέλων και η εκμετάλλευση των πλεονεκτημάτων του καθενός. Για τον προγραμματισμό στον SCC χρησιμοποιείται το RCCE API, για το οποίο έχουμε μελετήσει ένα άρθρο όπου οι ερευνητές προτείνουν κάποιες βελτιστοποιήσεις (optimizations) [6]. Προτείνεται η εφαρμογή της Cluster-On-Chip αρχιτεκτονικής που βασίζεται στη λογική της “software-oriented” ανταλλαγής μηνυμάτων και η οποία θα αντικαταστήσει την “hardware-based” συνέπεια μνήμης. Γενικά, οι ερευνητές προωθούν κάποιας μορφής χαμηλού επιπέδου λογισμικό για την message-passing επικοινωνία καθώς και τεχνικές για αποδοτικό προγραμματισμό της κοινής μνήμης όπως η ικανότητα για caching των δεδομένων. Τέτοιες βελτιστοποιήσεις, μαζί με διάφορες άλλες που προτάθηκαν, είναι στην διάθεση των χρηστών αφού έχουν ήδη ενσωματωθεί σε μια βιβλιοθήκη, την iRCCE, η οποία μπορεί να χρησιμοποιηθεί στον RCCE emulator. Επιπρόσθετα έχει μελετηθεί το λειτουργικό σύστημα Linux, που χρησιμοποιείται στον SCC, καθώς και οι διαφορές και οι ιδιαιτερότητες του από το κοινό Linux OS [8]. Όσον αφορά την απόδοση του SCC, έχουν μελετηθεί τρόποι αποδοτικού παραλληλισμού των διεργασιών (tasks) στους πυρήνες του συστήματος [7], καθώς και διάφορες τεχνικές για αποδοτικές “message-passing” [9], “memory-copy” [10] και “broadcasting” [12] λειτουργίες.

Όλα αυτά θα μας βοηθήσουν να δημιουργήσουμε μια καλή γνώση για τα συστήματα με τα οποία θα ασχοληθούμε ώστε να αντιμετωπίσουμε τις δυσκολίες και να λύσουμε τα προβλήματα που θα συναντήσουμε διεκπεραιώνοντας το έργο μας στο μέγιστο βαθμό.

Κεφάλαιο 3

Επεξεργαστής Intel SCC

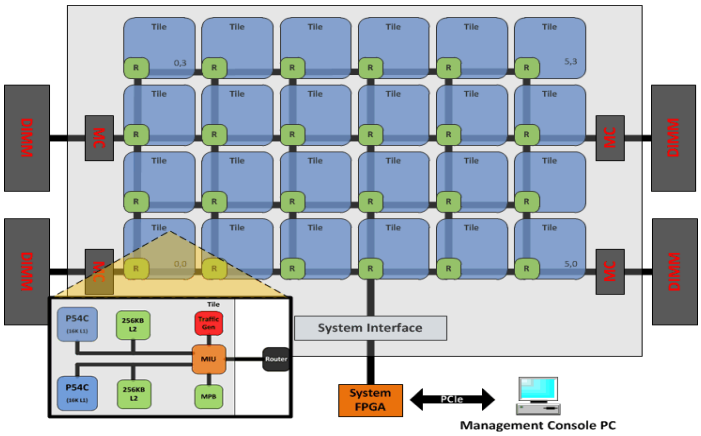
3.1 Ερευνητικό Πρόγραμμα της Intel

Η Intel αποτελεί σήμερα μια από τις εταιρίες κολοσσούς στο χώρο κατασκευής επεξεργαστών για υπολογιστικά συστήματα. Με ερευνητικά εργαστήρια σε όλες σχεδόν τις ηπείρους, η Intel έχει πλέον καθιερωθεί ως μια από τις επικρατέστερες γκάμες επεξεργαστών για όλους τους σύγχρονους ηλεκτρονικούς υπολογιστές οποιασδήποτε κατηγορίας. Άλλωστε ήταν από τις πρώτες εταιρίες που άνοιξαν τον δρόμο για τους πολυπύρηνους επεξεργαστές. Πρωτοπόρος σε θέματα επίδοσης και επεκτασιμότητας των πολυπύρηνων επεξεργαστών, η εν λόγω εταιρία δημιούργησε ένα ερευνητικό πρόγραμμα με το όνομα “Tera-scale Computing Research Program” το οποίο έχει ως στόχο την αύξηση της επίδοσης των σημερινών παράλληλων υπολογιστών για την επόμενη δεκαετία. Για τον σκοπό αυτόν, η Intel έχει κατασκευάσει κάποιους many-cores επεξεργαστές οι οποίοι αποτελούνται από πολλές δεκάδες πυρήνες και πάνω στους οποίους οι διάφοροι συνεργάτες της εταιρίας ανά το παγκόσμιο θα δοκιμάσουν τα πειράματα και τις τεχνικές τους για την εξυπηρέτηση του προαναφερθέντος σκοπού. Ένας από αυτούς τους επεξεργαστές είναι και ο Single-chip Cloud Computer (SCC) που αποσκοπεί στην

έρευνα για προγραμματιστικό λογισμικό σε many-cores συστήματα. Έχουν κατασκευαστεί περίπου 100 τέτοιοι επεξεργαστές, αποκλειστικά πειραματικοί, και οι οποίοι έχουν δοθεί σε ισάριθμα διαφορετικά ερευνητικά εργαστήρια σε όλο τον κόσμο τα οποία ενημερώνουν τακτικώς την Intel για την πρόοδο της έρευνας τους. Ένας τέτοιος επεξεργαστής υπάρχει και στο δικό μας εργαστήριο και η εργασία αυτή αποτελεί μέρος αυτής της έρευνας.

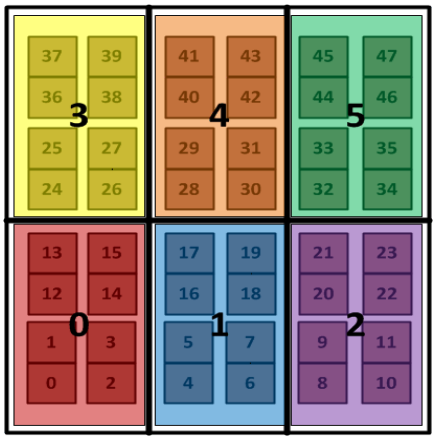
3.2 Αρχιτεκτονική και Οργάνωση

Ο επεξεργαστής SCC αποτελείται από 48 πυρήνες διαμοιρασμένους σε 24 “tiles” (2 πυρήνες σε κάθε tile) τα οποία είναι τοποθετημένα σε ένα 6x4 δίκτυο πλέγματος αλληλοσύνδεσης (mesh interconnection network). Ο κάθε πυρήνας είναι τύπου P54C και αρχιτεκτονικής Intel x86, γι’ αυτό και μπορεί να υποστηρίξει τους μεταγλωττιστές (compilers) και την τεχνολογία λειτουργικού συστήματος που απαιτούνται για τον προγραμματισμό του. Το κάθε tile έχει τον δικό του δρομολογητή (router) οι οποίοι επικοινωνούν μεταξύ τους στο δίκτυο βάσει του κλασικού TCP/IP μοντέλου επικοινωνίας. Στις άκρες του δικτύου πλέγματος υπάρχουν 4 διαχειριστές μνήμης (memory controllers) υπεύθυνοι για την επικοινωνία με την DDR3 μνήμη η οποία, ναι μεν είναι off-die, αλλά βρίσκεται πάνω στο board. Το board του SCC είναι συνδεδεμένο μέσω ενός PCIe bus με ένα Management Console (MCPC), το λεγόμενο “host” σύστημα, και το οποίο είναι ένα 64-bit PC με Linux λειτουργικό σύστημα. Ο σκοπός του είναι να μπορεί ο χρήστης να διαχειριστεί τον SCC μέσω λογισμικού που τρέχει στο MCPC. Για παράδειγμα, ο χρήστης μέσω του MCPC μπορεί να φορτώσει ένα Linux image σε ένα ή σε μια ομάδα από πυρήνες, να ρυθμίσει τους καταχωρητές του επεξεργαστή καθώς επίσης να φορτώσει τις υπό εκτέλεση εφαρμογές του στους πυρήνες του SCC.



Σχήμα 3.1: *Αρχιτεκτονική επεξεργαστή SCC [5]*

Όπως φαίνεται και στο Σχήμα 3.1 [5], το κάθε tile περιλαμβάνει 2 πυρήνες, ένα message-passing buffer (MPB), ένα traffic generator και ένα mesh interface unit (MIU), ενώ επιπρόσθετα, όπως προαναφέραμε, είναι συνδεδεμένο με ένα δρομολογητή (router) που προωθεί τα μηνύματα από το tile στο πλησιέστερο διαχειριστή μνήμης (memory controller) και αντίστροφα. Ο κάθε P54C πυρήνας περιέχει 32KB μνήμη L1 (16KB για instruction και 16KB για data) πάνω στον ίδιο τον πυρήνα, και 256KB μνήμη L2, η οποία βρίσκεται εκτός πυρήνα, ακριβώς δίπλα. Το MPB σε κάθε tile έχει μέγεθος 16KB και είναι προσβάσιμο από όλους τους πυρήνες του επεξεργαστή. Και τα 24 MPBs μαζί χρησιμεύουν στην επικοινωνία των πυρήνων μέσω μηνυμάτων (message-passing) αφού τα δεδομένα ταξιδεύουν μέσω των buffers αυτών για να καταλήξουν από τον αποστολέα στον παραλήπτη. Το traffic generator έχει ως στόχο την παρακολούθηση της επίδοσης του δικτύου μέσω των traffic μοτίβων (patterns) που δημιουργούνται κατά την εκτέλεση. Το MIU είναι υπεύθυνο για τη σύνδεση του tile με το υπόλοιπο δίκτυο πλέγματος (mesh network). Ελέγχει και πακετάρει τα δεδομένα που φεύγουν από το tile προς το δίκτυο ενώ εφαρμόζει ακριβώς το αντίθετο όταν δεδομένα καταφτάνουν από το δίκτυο στο tile. Για το συγχρονισμό της αποστολής δεδομένων από τους 2 πυρήνες του tile προς το δίκτυο χρησιμοποιείται αλγόριθμος round-robin. Επιπλέον, το MUI χρησιμοποιεί ένα lookup table (LUT) για κάθε πυρήνα ώστε να μεταφράσει την κάθε διεύθυνση μνήμης στην οποία αναφέρεται ο εκάστοτε πυρήνας, στην αντίστοιχη διεύθυνση μνήμης του συστήματος ώστε να γίνει γνωστή και από τους υπόλοιπους πυρήνες.



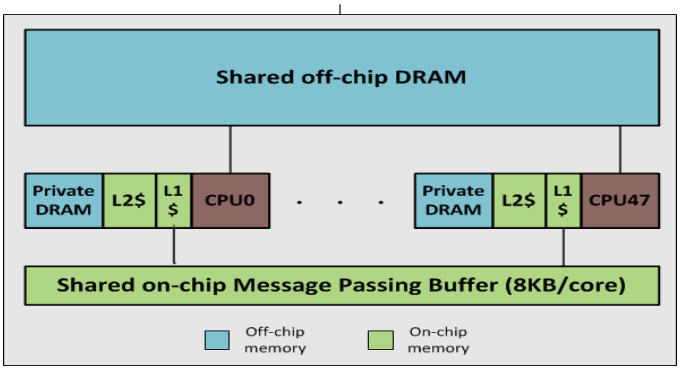
Σχήμα 3.2: *Τομείς τάσης SCC*

Ο επεξεργαστής SCC περιλαμβάνει επίσης το voltage regulator controller (VCR) το οποίο είναι ένα power controller που δίνει τη δυνατότητα στο χρήστη να ρυθμίζει αναλόγως την τάση (voltage) και την συχνότητα (frequency) στους πυρήνες του συστήματος. Κάθε tile του επεξεργαστή μπορεί να έχει τη δική του συχνότητα, ενώ κάθε ομάδα με 4 tiles μπορεί να έχει τη δική της τάση. Όπως φαίνεται και στο Σχήμα 3.2 ο χρήστης έχει την ευχέρεια να χαμηλώσει την τάση και την συχνότητα κάποιων

πυρήνων που δεν χρειάζεται σε τέτοιο σημείο που να τους αναγκάζει σχεδόν να απενεργοποιούνται. Επιπρόσθετα, κάθε πυρήνας στον SCC περιέχει ένα ψηφιακό αισθητήρα θερμοκρασίας (digital temperature sensor) ο οποίος αποθηκεύει τις μετρήσεις του σε ένα καταχωρητή ώστε να μπορούν παρακολουθηθούν από το χρήστη όταν χρειαστεί. Όλες οι πληροφορίες σχετικά με την τάση και τη συχνότητα του επεξεργαστή SCC μπορούν να εξασφαλιστούν και πάλι μέσω του MCPC χρησιμοποιώντας συγκεκριμένες εντολές. Τέλος, ανάμεσα στον επεξεργαστή και το MCPC βρίσκεται ένα FPGA το οποίο είναι υπεύθυνο για την δρομολόγηση των I/O κλήσεων του συστήματος.

3.3 Μνήμη

Η μνήμη στον επεξεργαστή SCC χωρίζεται σε δύο κατηγορίες: την on-chip SRAM που περιλαμβάνει τα 24 MPBs, ένα από κάθε tile, και την off-chip DDR3 DRAM η οποία έχει μέγεθος μέχρι και 64GB και όπως προαναφέραμε γίνεται προσβάσιμη μέσω των 4 διαχειριστών μνήμης (memory controllers) που βρίσκονται στο chip. Η SRAM μνήμη αποτελείται από το ενιαίο message-passing buffer μεγέθους 384KB που διαμοιράζεται στα 24 tiles (16KB για κάθε tile, 8KB για κάθε πυρήνα) και στο οποίο η πρόσβαση γίνεται μέσω ενός message-passing προγραμματιστικού μοντέλου όπως είναι το RCCE που θα δούμε στη συνέχεια. Όσον αφορά την DRAM μνήμη, το μεγαλύτερο μέρος της μοιράζεται στον κάθε πυρήνα και αποτελεί την private μνήμη του καθενός, ενώ το υπόλοιπο μέρος αντιπροσωπεύει την κοινή μνήμη που οι πυρήνες αξιοποιούν από κοινού. Από προεπιλογή δίνεται όσο το δυνατό περισσότερη προσωπική μνήμη στον κάθε πυρήνα και η εναπομένουσα γίνεται κοινή σε όλους. Υπάρχει η δυνατότητα ρύθμισης του μεγέθους της μνήμης που δίνεται ως προσωπική ή ως κοινή μέσω των τιμών των lookup tables (LUT) που αναφέραμε πιο πάνω. Οι τιμές αυτές μπορούν να ρυθμιστούν όποτε ο χρήστης επιθυμεί και εφαρμόζονται με το αμέσως επόμενο boot του συστήματος.



Σχήμα 3.3: *SCC on/off-chip μνήμη [5]*

Στο Σχήμα 3.3 [5] διακρίνονται ποιες μνήμες βρίσκονται μέσα στο chip του επεξεργαστή (κόκκινο χρώμα) και ποιες εκτός (πορτοκαλί χρώμα). Όσον αφορά το caching της μνήμης, τα δεδομένα του MPB προσπερνούν πάντα την L2 cache και φορτώνονται απευθείας στην L1, ενώ τα δεδομένα της κοινής μνήμης είναι εντελώς uncacheable δηλαδή μεταφέρονται απευθείας στο register file προσπερνώντας και την L1 και την L2 cache.

3.4 Προγραμματισμός με RCCE API

Για τον προγραμματισμό του επεξεργαστή SCC παρέχεται το RCCE ("rocky") API που περιλαμβάνει βιβλιοθήκες μέσω των οποίων ο προγραμματιστής μπορεί να αναπτύξει τις εφαρμογές του. Το message-passing (MPI) αυτό μοντέλο είναι το πλέον καταλληλότερο για τον προγραμματισμό του SCC αφού δίνει στον χρήστη τον πλήρη έλεγχο των χαρακτηριστικών του συστήματος και ειδικότερα των προσβάσεων στην κάθε μνήμη. Για τις αποστολές/παραλαβές των μηνυμάτων γίνεται χρήση της κοινής μνήμης, κυρίως του MPB, ενώ παράλληλα η RCCE βιβλιοθήκη περιέχει μεθόδους για πρόσβαση και στις υπόλοιπες μνήμης που περιέχονται στο chip. Κατά την αποστολή ενός μηνύματος ο πυρήνας-αποστολέας μεταφέρει τα δεδομένα που θέλει να στείλει από την L1 cache του στο MPB (συγκεκριμένα στο δικό του MPB) στο οποίο έχουν πρόσβαση όλοι οι πυρήνες. Από εκεί ο πυρήνας-παραλήπτης θα λάβει τα δεδομένα και θα τα μεταφέρει στη δική του L1 cache. Για την ανταλλαγή μηνυμάτων δεν χρησιμοποιείται καθόλου η off-chip μνήμη του συστήματος. Για την εξυπηρέτηση μιας τέτοιου είδους επικοινωνίας περιέχονται 2 ειδών μέθοδοι στην RCCE βιβλιοθήκη. Η πρώτη αφορά κλήσεις τύπου put και get, οι οποίες μπορούν να γίνουν ασύγχρονα από τον κάθε πυρήνα αλλά τα δεδομένα θα πρέπει να προστατευτούν μέχρι να παραληφθούν και γι' αυτό μεριμνεί αποκλειστικά ο προγραμματιστής (RCCE Gory Interface). Επίσης, υπάρχουν και οι συγχρονισμένες send και receive κλήσεις, οι οποίες κάνουν εσωτερική χρήση κάποιων "flags" ώστε να μπλοκάρουν-συγχρονίσουν την εκτέλεση έως ότου τα δεδομένα ληφθούν από τον παραλήπτη (RCCE Basic Interface). Αυτό δεν συμβαίνει όταν τα μηνύματα που αποστέλλονται είναι άδεια. Γενικά, το RCCE API είναι ένα περιβάλλον ανταλλαγής μηνυμάτων το οποίο με τις μεθόδους του εφαρμόζει αυτό που ονομάζουμε "one-sided" επικοινωνία και που με συνδυασμό κάποιων "flags" έχει τη δυνατότητα να συγχρονίσει την επικοινωνία αυτή.

Για δοκιμή και εκτέλεση των προγραμμάτων που προορίζονται για τον SCC διατίθεται ο RCCE εξομοιωτής (emulator) ο οποίος μπορεί να τρέχει σε οποιοδήποτε Windows ή Linux σύστημα και αφομοιώνει όλα τα χαρακτηριστικά του SCC. Επομένως, ο προγραμματιστής μπορεί να γράψει τον κώδικα της εφαρμογής του χρησιμοποιώντας το RCCE API, να τον εκτελέσει στον RCCE emulator για να βεβαιωθεί ότι δεν περιέχει σφάλματα και ακολούθως να τον μεταφέρει στον SCC. Το μυστικό της λειτουργίας του RCCE emulator είναι το γεγονός ότι πίσω από τις διάφορες μεθόδους του RCCE API που χρησιμοποιεί ο χρήστης, ο εξομοιωτής χρησιμοποιεί OpenMP για να τις ικανοποιήσει. Η χρήση της OpenMP γίνεται αντιληπτή από το εκάστοτε σύστημα που τρέχει τον εξομοιωτή όταν η εφαρμογή ξεκινά με την κλήση του *RCCE_APP()*. Στον SCC η κλήση αυτή δεν είναι απαραίτητη αφού το σύστημα αυτόματα θα την αντικαταστήσει με την κλασική κλήση της *main()*. Η κάθε RCCE εφαρμογή μπορεί να μεταγλωττιστεί με οποιοδήποτε συνηθισμένο μεταγλωττιστή (compiler) (π.χ. gcc, icc etc) συμπεριλαμβάνοντας την RCCE βιβλιοθήκη στην μεταγλώττιση, και ακολούθως μπορεί να εκτελεστεί, είτε στον emulator, είτε απευθείας στον SCC χρησιμοποιώντας την εντολή *rccerun*.

```
#include "RCCE.h"

int RCCE_APP(int argc, char **argv)
{
	int UEs, ME, YOU;
	char message[SIZE];

	RCCE_init(&argc, &argv);   //RCCE initialization

	UEs = RCCE_num_ues();      //number of Units of Execution
	ME = RCCE_ue();            //every Unit of Execution gets its own ID
	YOU = !ME;

	for (round = 0; round < ROUNDS; round++)
	{
		if (ME)
		{
			RCCE_send(message, sizeof(message), YOU);
			RCCE_recv(message, sizeof(message), YOU);
		}
		else
		{
			RCCE_recv(message, sizeof(message), YOU);
			RCCE_send(message, sizeof(message), YOU);

		}
	}

	RCCE_finalize();
	return(0);
}
```

Σχήμα 3.4: *Παράδειγμα κώδικα με χρήση RCCE βιβλιοθήκης*

Στο Σχήμα 3.4 παρατίθεται ο ψευδοκώδικας ενός προγράμματος που χρησιμοποιεί το RCCE API για την επικοινωνία μεταξύ των πυρήνων του SCC. Το συγκεκριμένο παράδειγμα αφορά ανταλλαγή μηνυμάτων μεταξύ μόνο δύο πυρήνων. Ο πυρήνας 1 αποστέλλει δεδομένα στον πυρήνα 0 και μόλις γίνει η λήψη του μηνύματος, ο πυρήνας 0 αποστέλλει τα δικά του δεδομένα πίσω στον πυρήνα 1. Η διαδικασία αυτή συνεχίζεται για 10 επαναλήψεις και αποτελεί το απλούστερο παράδειγμα message-passing επικοινωνίας μέσω της RCCE βιβλιοθήκης.

Κεφάλαιο 4

Φορητή Πλατφόρμα TFlux

4.1 Προγραμματιστικό Μοντέλο DDM

Η φορητή πλατφόρμα TFlux βασίζει τη λειτουργία της στο μοντέλο προγραμματισμού και εκτέλεσης Data-Driven Multithreading (DDM). Το μοντέλο DDM αποσκοπεί στην αποδοτική εκτέλεση παράλληλων εφαρμογών κάνοντας χρήση dataflow δρομολόγησης (dataflow scheduling). Η όλη φιλοσοφία του μοντέλου αυτού είναι ο διαχωρισμός του κώδικα της εφαρμογής σε μικρότερα κομμάτια, τα οποία ονομάζουμε *DThreads*. Τα DThreads μπορεί να είναι, είτε ανεξάρτητα μεταξύ τους, είτε να χρησιμοποιούν κάποια κοινά δεδομένα, άρα θα δημιουργούνται εξαρτήσεις μεταξύ τους, γεγονός που τα κάνει να συνδέονται με σχέσεις producer-consumer. Βάσει αυτών των εξαρτήσεων δημιουργείται ένας γράφος συγχρονισμού (synchronization graph) του οποίου οι κόμβοι παριστάνουν τα DThreads ενώ οι ακμές αντιπροσωπεύουν τις εξαρτήσεις μεταξύ τους. Ο γράφος αυτός φορτώνεται σε μια ξεχωριστή μονάδα, το thread scheduler, το οποίο έχοντας ως δεδομένο το

γράφο δρομολογεί με τη σωστή σειρά το κάθε DThread για εκτέλεση. Συγκεκριμένα, ο thread scheduler διατηρεί ένα μετρητή για κάθε DThread, που ονομάζει ReadyCounter, και ο οποίος αντιπροσωπεύει τον αριθμό όλων των παραγωγών (producers) του DThread. Κάθε φορά που ένας παραγωγός (producer) τελειώνει την εκτέλεσή του, ο ReadyCounter μειώνεται κατά ένα. Όταν η τιμή του μετρητή γίνει 0, τότε το συγκεκριμένο DThread μπορεί να δρομολογηθεί για εκτέλεση. Όσον αφορά την πλατφόρμα TFlux με την οποία ασχολούμαστε, το ρόλο του thread scheduler αναλαμβάνει η μονάδα συγχρονισμού του TFlux, δηλαδή το TSU, το οποίο θα μελετήσουμε εκτενέστερα στη συνέχεια αυτού του κεφαλαίου.

Ένα DThread για να είναι σε θέση να δρομολογηθεί για εκτέλεση θα πρέπει να ολοκληρώσουν την εκτέλεσή τους όλοι οι παραγωγοί (producers) του, ούτως ώστε όλα τα δεδομένα που χρειάζεται να είναι έτοιμα. Αυτός ο τρόπος δρομολόγησης ονομάζεται “data-driven” γιατί βασίζεται στη διαθεσιμότητα των δεδομένων του κάθε DThread. Οι εντολές που περιέχονται μέσα στον κώδικα ενός DThread εκτελούνται με τον κοινό control-flow τρόπο και μπορούν να τύχουν οποιωνδήποτε βελτιστοποιήσεων, είτε δυναμικών από τον επεξεργαστή, είτε στατικών από τον μεταγλωττιστή. Με τη data-driven δρομολόγηση, μειώνονται μεν οι καθυστερήσεις για υπολογισμούς συγχρονισμού, αλλά ταυτόχρονα δημιουργούνται πολύπλοκα μοτίβα πρόσβασης (access patterns) στη μνήμη, αφού ο τρόπος με τον οποίο θα φορτωθούν τα δεδομένα δεν είναι στατικά γνωστός. Για την επίλυση ενός τέτοιου προβλήματος το οποίο θα κοστίσει πολλά memory misses, το μοντέλο DDM χρησιμοποιεί την πολιτική CacheFlow, κατά την οποία όλα τα δεδομένα που θα χρειαστεί ένα DThread για την εκτέλεσή του φορτώνονται στη μνήμη λίγο πριν το DThread δρομολογηθεί για εκτέλεση. Τα δεδομένα θα απομακρυνθούν από τη μνήμη μόνο όταν ολοκληρώσουν την εκτέλεσή τους όλα τα DThreads που τα χρησιμοποιούν.

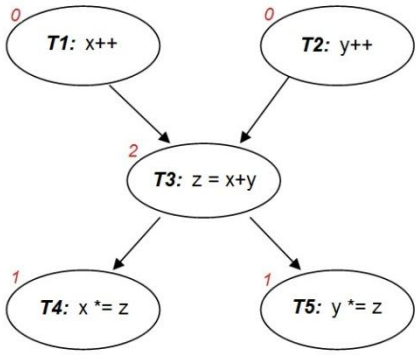
Σε περιπτώσεις όπου τα DDM προγράμματα είναι αρκετά μεγάλα, ο κώδικας τους χωρίζεται σε DDM Blocks τα οποία μπορεί να περιέχουν ένα η περισσότερα DThreads και το μέγεθος τους εξαρτάται από το μέγεθος του TSU. Επιπλέον, για κάθε Block δημιουργούνται δύο επιπρόσθετα DThreads, το Inlet και το Outlet. Το Inlet DThread αναλαμβάνει να φορτώσει στο thread scheduler, δηλαδή στο TSU, όλα τα δεδομένα των DThreads που περιλαμβάνονται στο συγκεκριμένο Block. Από την άλλη, το Outlet DThread έχει ως σκοπό την αποδέσμευση όλων των πόρων που χρησιμοποιήθηκαν κατά την εκτέλεση του Block. Με το πέρας της εκτέλεσης όλων

των DThreads ενός Block, φορτώνεται στο TSU το Inlet DThread του αμέσως επόμενου Block ώστε να συνεχίσει εκτέλεση. Σ' ένα DDM πρόγραμμα τα Blocks εκτελούνται με τη σειρά που τα συναντούμε στον κώδικα, σε αντίθεση με τα DThreads ενός Block τα οποία δύναται να εκτελεστούν με οποιαδήποτε σειρά, ακόμη και παράλληλα, ανάλογα με τις εξαρτήσεις που έχουν μεταξύ τους.

Στο TFlux ο εντοπισμός των DThreads γίνεται με τη χρήση κάποιων directives στον κώδικα της DDM εφαρμογής, τα οποία αναγνωρίζει ο TFlux Preprocessor και δημιουργεί το γράφο συγχρονισμού, παράγοντας ταυτόχρονα κώδικα C, με χρήση της βιβλιοθήκης pthreads, για παράλληλη εκτέλεση. Θα δούμε τον Preprocessor του TFlux αναλυτικότερα στη συνέχεια αυτού του κεφαλαίου. Στο Σχήμα 4.1(a) παρουσιάζεται ένα απλό DDM παράδειγμα με τα TFlux directives που προαναφέραμε στα οποία καθορίζονται, όχι μόνο τα DThreads, αλλά και οι εξαρτήσεις μεταξύ τους. Στο Σχήμα 4.1(b) φαίνεται ο γράφος συγχρονισμού που θα παραχθεί σύμφωνα με τα όσα καθορίζουν τα directives στο πρώτο σχήμα. Για κάθε DThread διακρίνεται με κόκκινο χρώμα η τιμή που θα έχει η μεταβλητή ReadyCounter του καθενός με την έναρξη της εκτέλεσης.

```
#pragma ddm block 1

#pragma ddm thread 1 kernel 1
      x++;
#pragma ddm endthread

#pragma ddm thread 2 kernel 2
      y++;
#pragma ddm endthread

#pragma ddm thread 3 kernel 2 depends (1,2)
      z = x + y;
#pragma ddm endthread

#pragma ddm thread 4 kernel 1 depends(3)
      x *= z;
#pragma ddm endthread

#pragma ddm thread 5 kernel 2 depends(3)
      y *= z;
#pragma ddm endthread

#pragma ddm endblock
```

(a)



(b)

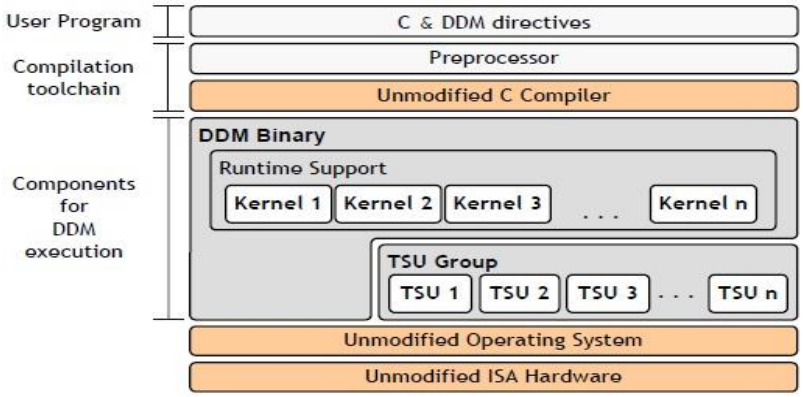
Σχήμα 4.1: *TFlux directives (pragma) και Γράφος Συγχρονισμού*

Για το DDM μοντέλο εκτέλεσης υπάρχουν και δύο hardware υλοποιήσεις. Η πρώτη ονομάζεται *D^2NOW* και προορίζεται για ένα δίκτυο από μηχανές, η καθεμιά από τις οποίες περιέχει τη μονάδα συγχρονισμού TSU υλοποιημένη επίσης σε hardware. Αυτή η υλοποίηση απαιτεί πολύ χαμηλού επιπέδου προγραμματισμό με επιπλέον προσθήκες συγκεκριμένων assembly εντολών. Η δεύτερη είναι το *TFluxHard*, που αναφέραμε και στην εισαγωγή, και το οποίο βασίζει την λειτουργία του και αυτό στην hardware υλοποίηση του TSU. Το TFluxHard μπορεί να τρέχει πάνω από οποιοδήποτε λειτουργικό σύστημα και ο προγραμματισμός του γίνεται με ψηλού

επιπέδου κώδικα, και συγκεκριμένα γλώσσας C. Παρόλα αυτά οι hardware υλοποιήσεις παραμένουν δεσμευτικές και μη ικανές να εκμεταλλευτούν όλες τις δυνατότητες του εκάστοτε συστήματος. Στην εργασία μας θα ασχοληθούμε μόνο με το *TFluxSoft*, το οποίο είναι υλοποίηση αποκλειστικά σε software.

4.2 TFluxSoft

Το TFlux είναι μια πλατφόρμα, η οποία εφαρμόζοντας κάποιας μορφής εικονικοποίηση (virtualization) στο σύστημα που εκτελείται, πετυχαίνει την αποδοτική εκτέλεση DDM εφαρμογών σε οποιαδήποτε αρχιτεκτονική ανεξαρτήτως υλικού και λειτουργικού συστήματος. Η υλοποίηση του TFlux κατά την οποία η μονάδα TSU είναι μια οντότητα λογισμικού και όχι υλικού, ονομάζεται TFluxSoft. Το TFluxSoft προορίζεται για οποιοδήποτε κοινό multicore σύστημα που χρησιμοποιεί shared-memory και περιλαμβάνει cache-coherency πρωτόκολλο για την on-chip μνήμη. Ο προγραμματιστής αναπτύσσει τον κώδικά του σε ANSI-C προσθέτοντας επίσης τα TFlux directives (pragma) που αναφέραμε στο προηγούμενο υποκεφάλαιο. Αυτούσιος ο κώδικας περνά από τον TFlux Preprocessor ο οποίος λαμβάνοντας υπόψη τα όσα ορίζονται από τα directives μεταφράζει το πρόγραμμα σε αντίστοιχο παράλληλο C κώδικα. Στη συνέχεια ο νέος κώδικας μεταγλωττίζεται από οποιοδήποτε κοινό μεταγλωττιστή (π.χ. gcc) και παράγεται ένα εκτελέσιμο αρχείο το οποίο μπορεί να εκτελεστεί πάνω από οποιαδήποτε αρχιτεκτονική αγνοώντας τα χαρακτηριστικά του συστήματος. Αυτό οφείλεται στο TFlux Runtime Support που περιλαμβάνεται στο εκτελέσιμο και το οποίο, σε επικοινωνία με τη μονάδα συγχρονισμού TSU, καταφέρνει να εκτελεί την εφαρμογή με data-driven τρόπο αποκρύπτοντας ταυτόχρονα όλες τις λεπτομέρειες των DDM προγραμμάτων. Η πιο πάνω διαδικασία φαίνεται και στο Σχήμα 4.2 [2] το οποίο αναπαριστά τον στρωματοποιημένο σχεδιασμό του TFlux.



Σχήμα 4.2: *Σχεδιασμός του TFlux [2]*

Κατά την εκτέλεση μιας TFlux εφαρμογής δημιουργείται ένας αριθμός από kernels. Ο αριθμός των kernels που θα δημιουργηθούν καθορίζεται από τον προγραμματιστή στον DDM κώδικα. Ο κάθε kernel είναι ένα POSIX thread το οποίο, αν είναι δυνατόν, εκτελείται σε ξεχωριστό πυρήνα από τα υπόλοιπα. Με την έναρξη της εκτέλεσης ο κάθε kernel φορτώνει στο TSU το Inlet DThread του πρώτου DDM Block που πρόκειται να εκτελεστεί και ο TSU ενημερώνει τον kernel ότι αυτό είναι το πρώτο DThread που πρέπει να εκτελέσει. Με την εκτέλεση του Inlet DThread φορτώνονται στο TSU όλες οι πληροφορίες σχετικά με τα DThreads που περιλαμβάνονται στο συγκεκριμένο Block και ο TSU επιτρέπει την εκτέλεσή τους από τους kernels. Ένας kernel ολοκληρώνει την εκτέλεσή του μόνο όταν εκτελέσει το Outlet DThread του τελευταίου DDM Block της εφαρμογής. Η εφαρμογή τερματίζει μόνο όταν όλοι οι TFlux kernels ολοκληρώσουν την εκτέλεση τους.

Στη συνέχεια αυτού του υποκεφαλαίου θα δούμε τα διάφορα συστατικά που απαρτίζουν το TFlux, όπως είναι το Runtime Support, το TSU και ο Preprocessor.

4.2.1 Runtime Support

Το TFlux Runtime Support είναι κάποιος επιπλέον κώδικας ο οποίος περιλαμβάνεται στον C κώδικα που παράγεται από τον Preprocessor. Κατά τη μεταγλώττιση του C κώδικα ενσωματώνεται στο εκτελέσιμο αρχείο της εφαρμογής και εκτελείται μαζί με την εφαρμογή χωρίς να απαιτείται κάποιο επιπλέον λογισμικό. Τρέχει πάνω από οποιοδήποτε Unix-based λειτουργικό σύστημα και αφομοιώνει όλες τις λεπτομέρειες της αρχιτεκτονικής παρέχοντας έτσι ένα virtualization του συστήματος που εγκαθίσταται η πλατφόρμα. Έχει ως απώτερο σκοπό να αποκρύπτει όλες τις ιδιαιτερότητες των DDM προγραμμάτων αναγκάζοντας το σύστημα να τις χειρίζεται ως κοινές εφαρμογές χωρίς να αντιλαμβάνεται την διαφορετικότητα τους. Επιπρόσθετα, το Runtime Support φορτώνει τα DThreads στο TSU με τρόπο που να πετυχαίνεται αποδοτική επικοινωνία μεταξύ TSU και εφαρμογής αναλαμβάνοντας ταυτόχρονα την ευθύνη να παρέχει στα DThreads πρόσβαση στις κοινές δομές της μνήμης που αφορούν τις εξαρτήσεις μεταξύ τους, είτε όταν η TFlux εφαρμογή εκτελείται σε shared-memory, είτε όταν εκτελείται σε distributed-memory σύστημα.

4.2.2 Thread Synchronization Unit

Η μονάδα συγχρονισμού TSU είναι ένα από τα κυριότερα συστατικά της πλατφόρμας TFlux αφού με τη χρήση της πετυχαίνεται η data-driven εκτέλεση της εφαρμογής. Στο TFluxSoft η μονάδα αυτή είναι υλοποιημένη ως μια οντότητα λογισμικού η οποία συγχρονίζει τα DThreads και τα δρομολογεί για εκτέλεση με τη σειρά που προβλέπει η λογική του DDM μοντέλου (data-driven scheduling). Φροντίζει να διατηρεί τις πληροφορίες-εξαρτήσεις που περιέχει ο γράφος συγχρονισμού και ταυτόχρονα ενημερώνει δυναμικά το γράφο μέσω κλήσεων στις διάφορες συναρτήσεις της. Σύμφωνα με το TFluxSoft, κάθε TFlux kernel έχει τη δική του μονάδα TSU στην οποία εγγράφονται όλες οι πληροφορίες σχετικά με τα DThreads που θα εκτελέσει ο συγκεκριμένος kernel. Παράλληλα, τα TSUs όλων των kernels μοιράζονται μια δομή στην κοινή μνήμη την οποία χρησιμοποιούν για να επικοινωνούν ούτως ώστε να ενημερώνονται τα DThreads όταν κάποιος producer τους ολοκλήρωσε την εκτέλεσή του. Με αυτό τον τρόπο επιτυγχάνεται η εύκολη εσωτερικά επικοινωνία μεταξύ των kernels της TFlux εφαρμογής χωρίς να χρειάζεται η ανάμειξη κάποιας άλλης μονάδας. Γι' αυτό το λόγο τα TSUs από όλους τους TFlux kernels ομαδοποιούνται σε ένα TSU Group, το οποίο περιλαμβάνει ξεχωριστά τους πόρους που αφορούν αποκλειστικά τον κάθε kernel, και ξεχωριστά τους κοινούς πόρους που προαναφέραμε. Βασικό ρόλο στη λειτουργία του TSU αναλαμβάνει το TFlux Emulator (συνάρτηση *emulateTSU*) το οποίο ουσιαστικά εκτελείται στο κυρίως thread της εκάστοτε εφαρμογής, δηλαδή στη *main*, και έχει ως σκοπό να διατηρεί το TSU σε λειτουργία έως ότου ολοκληρώσουν την εκτέλεσή τους όλοι οι TFlux kernels. Οι βασικές λειτουργίες-συναρτήσεις του TSU είναι η *loadThread* με την οποία ο kernel φορτώνει στο TSU πληροφορίες σχετικά με τα DThreads που πρόκειται να εκτελέσει, η *threadCompletedExecution* με την οποία ο kernel ενημερώνει το TSU για την ολοκλήρωση της εκτέλεσης κάποιου DThread, η *updateThreadInstances* με την οποία ενημερώνεται το ReadyCounter κάποιου consumer και η *clearTSU* κατά την οποία απελευθερώνονται οι δεσμευμένοι από το TSU πόροι. Επιπλέον, ο TSU έχει τη δυνατότητα να εντοπίζει ανά πάσα στιγμή ποιο DThread έχει σειρά να εκτελεστεί.

Η χρήση μιας μονάδας TSU σε κάθε kernel δεν είναι δεσμευτική. Λόγω του ότι το TSU είναι υλοποιημένο σε software, υπάρχει η ευελιξία να μπορούμε να

χρησιμοποιήσουμε το TSU ως μόνο μια υπολογιστική οντότητα αφιερώνοντας αποκλειστικά γι' αυτό ένα πυρήνα του πολύ-πύρηνου επεξεργαστή μας.

4.2.1 Preprocessor TFluxCpp

Ο TFlux C Preprocessor είναι το σημαντικότερο εργαλείο που χρησιμοποιεί η πλατφόρμα TFlux για τη μετάφραση DDM εφαρμογών. Το εργαλείο αυτό δέχεται ως είσοδο τον ANSI-C κώδικα με τα TFlux pragma που προαναφέραμε και παράγει παράλληλο κώδικα C που περιέχει όλη τη λειτουργία του Runtime Support συμπεριλαμβανομένων και των κλήσεων του TFlux για επικοινωνία του TSU με την εφαρμογή. Ο Preprocessor εκτελεί δύο σαρώσεις του κώδικα που δέχεται ως input. Κατά τη διαδικασία της πρώτης σάρωσης, η οποία ονομάζεται "front-end", παρατηρεί τον κώδικα, διαβάζει τα TFlux pragma, και εντοπίζει τις εξαρτήσεις ανάμεσα στα DThreads σχηματίζοντας έτσι τον γράφο συγχρονισμού που θα φορτωθεί στο TSU. Όλες οι πληροφορίες που συλλέχτηκαν στην πρώτη σάρωση θα μεταφερθούν στην δεύτερη, την ονομαζόμενη "back-end". Σ' αυτήν, ο Preprocessor φορτώνει στο TSU όλα τα δεδομένα που χρειάζεται, όπως είναι ο synchronization graph, και ταυτόχρονα χρησιμοποιώντας τις πληροφορίες από τη front-end διαδικασία, θα παράξει παράλληλο πρόγραμμα σε C το οποίο θα αποτελείται, όχι μόνο από τον κώδικα της εφαρμογής, αλλά και από τον κώδικα για λειτουργία του Runtime Support, όπως είναι αυτός των TFlux kernels και της ενημέρωσης του TSU.

Ο κώδικας που παράγεται μπορεί να διαφέρει από αρχιτεκτονική σε αρχιτεκτονική, γι αυτό και κατά την εκτέλεσή του ο Preprocessor δέχεται ειδική παράμετρο με την οποία καθορίζεται από τον προγραμματιστή το σύστημα στο οποίο θα τρέξει ο κώδικας που θα παραχθεί.

Κεφάλαιο 5

Εκτέλεση DDM μοντέλου στον Intel SCC μέσω TFlux

5.1 Ιδέα και Βάση Υλοποίησης

Μετά τη μελέτη και την κατανόηση όλων των χαρακτηριστικών του επεξεργαστή SCC και της πλατφόρμας TFlux, θα πρέπει να συλλογιστούμε ποια θα είναι η λογική που θα ακολουθήσουμε στις διαδικασίες για μεταφορά της πλατφόρμας στον επεξεργαστή. Το μοντέλο μνήμης που θα χρησιμοποιηθεί, ο τρόπος επικοινωνίας μεταξύ των πυρήνων και η ανάθεση του ρόλου της μονάδας TSU στο σύστημα, είναι κάποια από τα βασικότερα ζητήματα που θα μας προβληματίσουν προτού καταλήξουμε σε μια βάση για την υλοποίηση μας. Χρησιμοποιώντας τη βάση αυτή θα χαράξουμε μια γραμμή πάνω στην οποία θα κινηθούμε μέχρι και την διεκπεραίωση του έργου μας.

Η αρχική ιδέα όσον αφορά την εκτέλεση της πλατφόρμας TFlux στον επεξεργαστή SCC έχει να κάνει με την σχέση μεταξύ TFlux kernels και SCC cores. Έχουμε αποφασίσει ότι εδώ θα έχουμε μια πλήρη αντιστοιχία, δηλαδή για κάθε TFlux kernel θα αντιστοιχεί ένας SCC πυρήνας στον οποίο και θα εκτελείται ο συγκεκριμένος kernel. Αν λοιπόν η DDM εφαρμογή μας απαιτεί (μέσω των TFlux pragma) τη χρήση

N kernels, τότε η εκτέλεση της εφαρμογής στον SCC θα γίνει με συμμετοχή N πυρήνων, από τους οποίους ο καθένας θα αναλάβει την εκτέλεση ενός kernel. Ο αριθμός των kernels στην DDM εφαρμογή δεν μπορεί να υπερβαίνει το 48 αφού τόσοι πυρήνες περιλαμβάνονται στο σύστημα του SCC. Τα δεδομένα της εφαρμογής θα χρησιμοποιούν δύο διαφορετικά μοντέλα μνήμης. Όσα ορίζονται από τα TFlux directives ως προσωπικά για τον κάθε kernel θα δεσμεύονται στην προσωπική μνήμη του κάθε πυρήνα, ενώ όσα ορίζονται ως καθολικά για όλους kernels θα δεσμεύονται στην κοινή μνήμη του συστήματος στην οποία θα έχουν πρόσβαση όλοι οι πυρήνες αφού πρώτα καλέσουν τη συνάρτηση *RCCE_shmalloc()* της RCCE βιβλιοθήκης πάνω στα δεδομένα αυτά. Με αυτό τον τρόπο, δεν θα χρειάζεται καμιά επικοινωνία μεταξύ των πυρήνων όσον αφορά τα κοινά δεδομένα της εφαρμογής αφού ο καθένας τους θα μπορεί να ενημερώσει τις τιμές τους απευθείας στη shared μνήμη όποτε χρειαστεί.

Ο ρόλος της μονάδας συγχρονισμού TSU δεν θα ανατεθεί αποκλειστικά σε ένα πυρήνα ως μόνο μια υπολογιστική οντότητα. Αντίθετα, όλοι οι πυρήνες που συμμετέχουν στην εκτέλεση θα τρέχουν ο καθένας το δικό του TSU, το οποίο θα περιλαμβάνει τον γράφο συγχρονισμού στην ίδια ακριβώς κατάσταση με τους υπόλοιπους ανά πάσα στιγμή κατά τη διάρκεια της εκτέλεσης. Κάθε φορά που ο γράφος θα τυγχάνει αλλαγών, τότε ο πυρήνας που τρέχει το συγκεκριμένο TSU που έκανε την αλλαγή θα φροντίζει να ενημερώνει και τα υπόλοιπα TSUs μέσω ανταλλαγών μηνυμάτων μεταξύ των πυρήνων κάνοντας χρήση των συναρτήσεων *RCCE_send()* και *RCCE_recv_test()* της RCCE βιβλιοθήκης. Θα δούμε αναλυτικότερα τις προσθήκες που εφαρμόσαμε στον υφιστάμενο κώδικα του TSU στη συνέχεια αυτού του κεφαλαίου. Να σημειώσουμε όμως ότι κατά την εκτέλεση μιας TFlux εφαρμογής στο TFluxSoft γίνεται χρήση της βιβλιοθήκης pthread για να δημιουργηθεί ένα επιπλέον thread σε κάθε kernel το οποίο θα αναλάβει την εκτέλεση των εντολών της εφαρμογής, ενώ παράλληλα το κυρίως thread, όπως προαναφέραμε στο προηγούμενο κεφάλαιο, θα εκτελεί χρέη TSU Emulator. Επομένως, είναι λογικό επακόλουθο ότι μια τέτοια εκτέλεση στον SCC σε συνδυασμό με τη χρήση μονάδας TSU στον κάθε πυρήνα, θα απαιτεί τη χρήση δύο threads σε κάθε πυρήνα, ένα για την εφαρμογή και ένα για το TSU. Οι πυρήνες όμως στο σύστημα του SCC είναι single-threaded, άρα αυτό λογικά θα μας επιφέρει κάποιο κόστος στους χρόνους εκτέλεσης. Η επίτευξη αποδοτικής εκτέλεσης δεν είναι μέσα στους στόχους της εργασίας αυτής αλλά παρόλα αυτά θα μελετήσουμε τις επιδόσεις μας και θα προσπαθήσουμε να επινοήσουμε τρόπους βελτίωσης τους.

Τέτοιες τεχνικές για βελτίωση της επίδοσης αναφέρονται και στο κεφάλαιο με τα συμπεράσματα ως μελλοντική δουλειά.

Βασισμένοι στο πλάνο που περιγράψαμε πιο πάνω θα ακολουθήσουμε μια διαδικασία αποσυμπίλισης (reverse engineering) για να πετύχουμε τον στόχο μας. Δεν μπορούμε να ξεκινήσουμε απευθείας με τη διαμόρφωση του Preprocessor χωρίς να γνωρίζουμε πως πρέπει να είναι ο κώδικας που θα προορίζεται για εκτέλεση στον SCC. Επομένως, θα πρέπει πρώτα να τροποποιήσουμε τον TFlux κώδικα ώστε να μπορεί να εκτελείται στον επεξεργαστή και ακολούθως να εφαρμόσουμε τις όποιες αλλαγές χρειάζονται στον Preprocessor για να παράγεται αυτόματα ο επιθυμητός κώδικας.

5.2 Μετατροπή και Εκτέλεση TFlux κώδικα στον SCC

Η αρχή πρέπει να γίνει με τη μελέτη και την επεξεργασία του TFlux κώδικα που παράγεται από τον Preprocessor στο TFluxSoft. Θα πρέπει να βεβαιωθούμε ότι ο κώδικας αυτός μπορεί να τρέξει χωρίς καμιά τροποποίηση σε ένα τουλάχιστον SCC πυρήνα. Πήραμε, λοιπόν, σαν παράδειγμα μια πολύ απλή DDM εφαρμογή, συγκεκριμένα αυτή του Σχήματος 4.1 που εκτελεί μερικές αλληλοεξαρτώμενες πράξεις, και ανεξαρτήτως αριθμού kernels που χρησιμοποιεί την δώσαμε ως είσοδο στον Preprocessor. Μεταφέραμε αυτούσιο τον κώδικα που παράχθηκε από τον Preprocessor στον επεξεργαστή SCC και τον εκτελέσαμε σε ένα SCC πυρήνα χωρίς καμιά απολύτως αλλαγή. Ο κώδικας εκτελέστηκε με επιτυχία και με ορθό αποτέλεσμα, γεγονός που μας επιβεβαίωσε το αναμενόμενο. Αυτό βέβαια δεν είναι το ζητούμενο αφού όλοι οι kernels που χρησιμοποίησε η εφαρμογή εκτελέστηκαν σε ένα μόνο πυρήνα του επεξεργαστή και χωρίς να χρησιμοποιήσουν καθόλου το RCCE API. Σκοπός μας ήταν να βεβαιωθούμε ότι όντως ο TFlux κώδικας μπορεί να εκτελεστεί στην αρχιτεκτονική του SCC.

Το επόμενο βήμα είναι να προσαρμόσουμε κατάλληλα τον κώδικα αυτόν ώστε να μπορεί να τρέξει και πάλι επιτυχώς στον SCC, ταυτίζοντας παράλληλα τα TFlux kernels με τους SCC πυρήνες. Βεβαίως, για να το πετύχουμε αυτό θα πρέπει να εφαρμόσουμε ταυτόχρονα και κάποιες αλλαγές στο TSU. Οι αλλαγές αυτές αναφέρονται στο επόμενο υποκεφάλαιο. Σε αυτό το υποκεφάλαιο θα εστιάσουμε

την προσοχή μας στις τροποποιήσεις που χρειάζεται να γίνουν μόνο στον TFlux κώδικα.

Αρχικά, πρέπει να συμπεριλάβουμε στον κώδικα τη βιβλιοθήκη *RCCE.h* και να προσθέσουμε στον κώδικα όλα τα απαραίτητα RCCE macros που χρειάζονται για την εκτέλεση μιας εφαρμογής στον SCC, όπως είναι τα *RCCE_APP()*, *RCCE_init()*, *ME=RCCE_ue()*, UEs=*RCCE_num_ues()* και *RCCE_finalize()*. Οι μεταβλητές *ME* και *UEs* θα πρέπει να οριστούν έξω από το κυρίως πρόγραμμα για να μπορούν να είναι ορατές και στο TSU αλλά και στο δεύτερο thread που θα δημιουργηθεί για να εκτελέσει τις εντολές της εφαρμογής. Στον SCC δεν θα χρειαστεί καθόλου η χρήση αμοιβαίου αποκλεισμού επομένως θα πρέπει να αφαιρέσουμε οποιαδήποτε mutex χρησιμοποιούνται από το TFluxSoft. Κατόπιν, θα πρέπει να αλλάξουμε τη μεταβλητή *ddmVar_numberOfLiveKernels* η οποία περιέχει τον αριθμό των ζωντανών kernels κατά τη διάρκεια της εκτέλεσης. Η μεταβλητή αυτή είναι ορισμένη στο TFluxSoft ως ένας καθολικός (global) ακέραιος ο οποίος ορίζει ανά πάσα στιγμή της εκτέλεσης πόσοι TFlux kernels είναι ενεργοί. Ο κάθε kernel, μόλις τερματίσει την εκτέλεσή του το application thread, μειώνει την τιμή της μεταβλητής αυτής κατά 1. Όταν η τιμή της γίνει 0 τερματίζει την εκτέλεσή του και το TSU thread. Θα πρέπει λοιπόν η μεταβλητή αυτή στον SCC να γίνει ένας global δείκτης πάνω στον οποίο όλοι οι πυρήνες να καλούν τη *RCCE_shmalloc()*. Ο δείκτης αυτός θα είναι ένας πίνακας τόσων θέσεων όσος είναι και ο αριθμός των πυρήνων-kernels που χρησιμοποιούνται. Κάθε φορά που ένας πυρήνας ξεκινά την εκτέλεση του θα θέτει στη αντίστοιχη θέση του πίνακα τιμή 1, ενώ κάθε φορά που το application thread του ολοκληρώνει την εκτέλεση του θα θέτει τιμή 0. Για να βρίσκει ο κάθε πυρήνας τη θέση που τους αντιστοιχεί θα χρησιμοποιεί την μεταβλητή *ME* η οποία είναι ένας αύξων αριθμός που αντιπροσωπεύει την ταυτότητα του καθενός. Όταν το άθροισμα σε όλες τις θέσεις του πίνακα είναι ίσο με 0 τότε δεν υπάρχει κανένας άλλος kernel-πυρήνας που να εκτελεί κάτι και επομένως το πρόγραμμα μπορεί να τερματίσει. Όσον αφορά τα δεδομένα του προγράμματος, θα πρέπει να ξεχωρίσουμε ποια από αυτά είναι καθολικά (global) και ποια προσωπικά (private). Όσα δεδομένα είναι προσωπικά για τον κάθε kernel θα παραμείνουν ως έχουν, δηλαδή στην προσωπική μνήμη του κάθε πυρήνα. Όσα δεδομένα είναι καθολικά θα πρέπει όλοι οι πυρήνες, με την αρχή του προγράμματος, να καλέσουν την *RCCE_shmalloc()* πάνω τους και ένας από αυτούς να αναλάβει την αρχικοποίηση τους (συνήθως ο ME=0). Μετά την δέσμευση και την αρχικοποίηση των κοινών δεδομένων είναι αναγκαίο ένα φράγμα (barrier) έτσι ώστε κανένας πυρήνας να μην

προχωρήσει την εκτέλεση του χωρίς να έχουν οριστεί τα δεδομένα από όλους τους υπόλοιπους. Επομένως αμέσως μετά την κλήση της *RCCE_shmalloc()* θα πρέπει να ακολουθήσει μια κλήση της RCCE_barrier() για όλους τους πυρήνες που συμμετέχουν στην εκτέλεση. Το ίδιο πρέπει να συμβεί και μετά την αρχικοποίηση των δεδομένων.

Το TFluxSoft για να δημιουργήσει τους kernels που απαιτεί το πρόγραμμα χρησιμοποιεί ένα βρόγχο επανάληψης μέσα στον οποίο δημιουργεί (με χρήση της βιβλιοθήκης pthread) τόσα threads όσα είναι και τα kernels που έχουν οριστεί στο DDM πρόγραμμα. Αυτά τα threads μόλις δημιουργηθούν αναλαμβάνουν να εκτελέσουν τις διάφορες εντολές της εφαρμογής με την ανάλογη σειρά που καθορίζουν οι εξαρτήσεις μεταξύ τους. Οι εντολές της εφαρμογής, μαζί με τον κώδικα για λειτουργία του Runtime Support που αναφέραμε σε προηγούμενο κεφάλαιο, βρίσκονται στη συνάρτηση *ddm_code()* του TFlux κώδικα. Για τη δημιουργία των kernels στον SCC θα πρέπει να γίνει μια πολύ σημαντική τροποποίηση η οποία αντιπροσωπεύει και το νόημα της αντιστοίχισης των TFlux kernels με τους SCC πυρήνες. Γνωρίζοντας ότι κατά την εκτέλεση στον SCC, κάθε πυρήνας τρέχει ένα δικό του αντίγραφο του κώδικα αντιλαμβανόμαστε ότι ο βρόγχος επανάληψης για τη δημιουργία των N kernels δεν χρειάζεται. Αντίθετα, αφού έχουμε N πυρήνες που εκτελούν τον ίδιο κώδικα θα πρέπει ο καθένας να δημιουργήσει το δικό του kernel το οποίο θα τρέξει στον ίδιο τον πυρήνα που το δημιούργησε. Επομένως, είναι αρκετή μια μόνο κλήση της *pthread_create()* την οποία θα εκτελέσουν όλοι οι πυρήνες ξεχωριστά. Μετά την κλήση της συνάρτησης αυτής ακολουθεί η λειτουργία του TSU Emulator όπου επαναλαμβάνεται μέσα σε ένα βρόγχο η κλήση της συνάρτησης *emulateTSU()* μέχρι που όλοι τα application threads να ολοκληρώσουν την εκτέλεσή τους, δηλαδή όλες οι τιμές στον πίνακα *ddmVar_numberOfLiveKernels* να είναι 0. Ο βρόγχος επανάληψης αυτός θα παραμείνει ως έχει με τη διαφορά ότι θα προσθέσουμε κάποιες εντολές για τον υπολογισμό του αθροίσματος των τιμών του πίνακα *ddmVar_numberOfLiveKernels*.

Η συνάρτηση *ddm_code()*, η οποία θα εκτελεστεί από τα application threads που θα δημιουργηθούν, δεν χρειάζεται οποιεσδήποτε τροποποιήσεις παρά μόνο τα κοινά δεδομένα της εφαρμογής θα πρέπει να χειρίζονται ως δείκτες αφού πλέον έχουν γίνει global δείκτες.

5.3 Τροποποίηση μονάδας TSU

Για να είναι λειτουργικές οι τροποποιήσεις που εφαρμόσαμε στον TFlux κώδικα θα πρέπει να γίνουν και οι κατάλληλες αλλαγές στον κώδικα του TSU. Όπως προαναφέραμε, στη δική μας υλοποίηση ο TSU δεν θα εκτελείται ως μια υπολογιστική οντότητα για όλους του πυρήνες αλλά ο κάθε πυρήνας θα τρέχει το δικό του διατηρώντας τον στην ίδια κατάσταση με τους υπόλοιπους. Επομένως με κάποιο τρόπο θα πρέπει να πετύχουμε μια αποδοτική επικοινωνία μεταξύ των πυρήνων, όποτε αυτό χρειάζεται. Όλη μας η προσοχή θα εστιαστεί στη συνάρτηση *emulateTSU()* αφού εκεί γίνονται οι διάφορες ενημερώσεις στις δομές του TSU. Ο υφιστάμενος TSU του TFluxSoft περιλαμβάνει κάποια κομμάτια κώδικα στη συγκεκριμένη συνάρτηση τα οποία προορίζονται για συστήματα κατανεμημένης (distributed) μνήμης. Θα επικεντρωθούμε σε αυτά τα κομμάτια στοχεύοντας στην επίτευξη μιας επικοινωνίας όπου ο κάθε πυρήνας θα λαμβάνει μηνύματα για τις αλλαγές που έκαναν στο TSU τους οι υπόλοιποι ώστε να ενημερώσει και αυτός το δικό του.

Καταρχάς, θα πρέπει να συμπεριλάβουμε στον κώδικα του TSU τη βιβλιοθήκη RCCE και επίσης να αφαιρέσουμε οποιαδήποτε mutex υπάρχουν αφού και πάλι δεν χρειαζόμαστε καθόλου αμοιβαίο αποκλεισμό για τα δεδομένα μεταξύ των πυρήνων μιας και ο καθένας τους τρέχει το δικό του TSU. Κατόπιν, θα μεταφερθούμε στα σημεία του κώδικα της *emulateTSU()* που αφορούν κατανεμημένη (distributed) εκτέλεση. Τα σημεία αυτά θα τα αντιγράψουμε και θα τα τροποποιήσουμε ανάλογα ώστε να αφορούν εκτέλεση στο δικό μας σύστημα. Σύμφωνα με την κατανεμημένη (distributed) υλοποίηση, όπου έχουμε κλήση συστήματος (system call) *write()* θα την αντικαταστήσουμε με κλήση της συνάρτησης *RCCE_send()* και όπου έχουμε κλήση συστήματος (system call) *read()* θα την αντικαταστήσουμε με κλήση της συνάρτησης *RCCE_recv_test()*. Χρησιμοποιούμε τη *RCCE_recv_test()* και όχι τη *RCCE_recv()* ούτως ώστε να μην μπλοκάρεται η εκτέλεση όταν κάποιος πυρήνας προσπαθεί να λάβει κάποιο μήνυμα από κάποιον που δεν έχει στείλει τίποτα. Όταν ο TSU προορίζεται για εκτέλεση στον επεξεργαστή SCC θα προσθέτουμε στην αρχή του κώδικα συγκεκριμένο “#define” directive το οποίο θα καθορίζει ότι πρέπει να χρησιμοποιηθούν τα κομμάτια κώδικα που προσθέσαμε. Η γενική ιδέα βρίσκεται στο γεγονός ότι οποιοσδήποτε πυρήνας και να καλέσει την *emulate()* μπορεί να γνωρίζει ανά πάσα στιγμή ποιος πυρήνας έχει σειρά για εκτέλεση σύμφωνα με το γράφο συγχρονισμού. Αυτό γίνεται γνωστό μέσω της μεταβλητής *tsuID* που περιλαμβάνει

την ταυτότητα του πυρήνα στον οποίο θα συνεχίσει η εκτέλεση της εφαρμογής. Κατά τη δική μας υλοποίηση όταν ένας πυρήνας είναι έτοιμος να αναλάβει την εκτέλεση της εφαρμογής θα περιμένει να λάβει ένα μήνυμα από όλους τους υπόλοιπους πυρήνες οι οποίοι θα τον ενημερώνουν για την κατάσταση του δικού τους γράφου προτού ξεκινήσει την εκτέλεσή του. Με αυτό τον τρόπο κάθε φορά που η εκτέλεση της εφαρμογής μεταφέρεται κάπου αλλού, όλοι οι πυρήνες θα ενημερώσουν τον επόμενο για όλα όσα προηγήθηκαν έτσι ώστε να συνεχίσει η εκτέλεση με την πιο πρόσφατη κατάσταση του γράφου. Έτσι, όλοι οι πυρήνες θα διατηρούν το γράφο τους σε όμοια ακριβώς κατάσταση για κάθε χρονικό σημείο της εκτέλεσης. Μεταφέραμε τον τροποποιημένο κώδικα του TSU στον επεξεργαστή SCC και εκτελέσαμε το νέο TFlux κώδικα του παραδείγματος που προσαρμόσαμε προηγουμένως. Η εφαρμογή εκτελέστηκε με επιτυχία επιστρέφοντας τα ορθά αποτελέσματα.

Με τις αλλαγές αυτές έχουμε καταφέρει την μέσω μηνυμάτων (message-passing) επικοινωνία μεταξύ των πυρήνων με τρόπο που να μπορούν να ενημερώνονται για την κάθε πρόσφατη αλλαγή που γίνεται στα δεδομένα του TSU προτού ξεκινήσουν την εκτέλεση τους. Πλέον, έχουμε μια υλοποίηση κατά την οποία οι πυρήνες που συμμετέχουν μοιράζονται τα κοινά (shared) δεδομένα της εφαρμογής και τα ενημερώνουν όποτε επιθυμούν, ενώ παράλληλα, μέσω των TSUs τους, ανταλλάζουν μηνύματα με τρόπο που να εκπληρώνονται οι εξαρτήσεις μεταξύ των δεδομένων επαληθεύοντας έτσι την αρχική μας ιδέα.

5.4 Επέκταση TFlux C Preprocessor

Το τελευταίο βήμα που μας έχει απομείνει για να ολοκληρώσουμε την υλοποίηση μας είναι να επεκτείνουμε τον Preprocessor του TFlux ώστε όλες αυτές οι τροποποιήσεις που εφαρμόσαμε στον κώδικα του TFlux να γίνονται αυτόματα και να παράγεται ο επιθυμητός κώδικας απευθείας από DDM πρόγραμμα. Να τονίσουμε εδώ ότι οι οποιεσδήποτε αλλαγές και προσθήκες θα γίνουν στον Preprocessor θα έχουν να κάνουν αποκλειστικά με το πώς θα παράγεται ο κώδικας της TFlux εφαρμογής και όχι του TSU. Ο κώδικας του TSU παραμένει ο ίδιος για εκτέλεση σε οποιοδήποτε σύστημα και δεν μεταβάλλεται καθόλου από τον Preprocessor.

Ο TFlux C Preprocessor αποτελείται από πολλά αρχεία κώδικα τα οποία χειρίζονται ένα DDM κώδικα εισόδου και γράφουν την έξοδο τους σε ένα άλλο αρχείο το οποίο ουσιαστικά είναι ο τελικός TFlux κώδικας που πρόκειται να εκτελεστεί. Από τα αρχεία του Preprocessor τα τρία αρχεία που έχουν υποστεί αλλαγών και τα οποία περιλαμβάνουν τη βασική δουλειά του Preprocessor είναι το *cli.c*, το *ddmcpp.y* και το *softPthreads.c*. Στο πρώτο αρχείο διαβάζεται η γραμμή εντολής (command line) που δόθηκε από το χρήστη για να εκτελεστεί ο Preprocessor και εντοπίζεται η παράμετρος που καθορίζει την αρχιτεκτονική για την οποία προορίζεται ο κώδικας που θα παραχθεί. Οι μέχρι στιγμής πιθανές παράμετροι που γίνονται αποδεκτές είναι οι *hard*, *soft*, *cell*, *intermediate* και *atomic*. Εμείς θα προσθέσουμε, εδώ, μια νέα παράμετρο με το όνομα *scc,* ούτως ώστε όταν δοθεί να παράγεται ο TFlux κώδικας που διαμορφώσαμε πιο πάνω. Με αυτήν την παράμετρο ενεργοποιείται μια μεταβλητή-flag scc η οποία θα επιτρέπει τις αλλαγές που θα εφαρμόσουμε και στα υπόλοιπα αρχεία. Διαφορετικά, καμιά από τις πιο κάτω αλλαγές δεν θα ληφθεί υπόψη από τον Preprocessor στη παραγωγή του κώδικα.

Στη συνέχεια, στο αρχείο ddmcpp.y ορίσαμε όλες τις επιπλέον μεταβλητές που χρειαζόμαστε καθώς και τα διάφορα σταθερά RCCE macros που πρέπει να χρησιμοποιηθούν σε μια RCCE εφαρμογή, όπως *RCCE_init()*, *RCCE_ue()*, *RCCE_finalize()* κτλ. Επίσης, όποτε ο Preprocessor αντιλαμβάνεται τη χρήση TFlux directive στο οποίο ορίζονται global δεδομένα, θα παράγει κώδικα ο οποίος θα κάνει χρήση της συνάρτησης *RCCE_shmalloc()* πάνω στα δεδομένα αυτά και κατόπιν θα τυπώνει το *RCCE_barrier()*.

Τέλος, το αρχείο softpthreads.c, στο οποίο παράγεται και το μεγαλύτερο μέρος του κώδικα, είναι υπεύθυνο να συμπεριλάβει τη RCCE βιβλιοθήκη και να ορίσει τις μεταβλητές *ME*, *UEs* και *ddmVar_numberOfLiveKernels*. Επιπλέον, όσον αφορά τη δημιουργία των kernels θα γράψει στο αρχείο εξόδου τη συνάρτηση pthread_create() χωρίς τον βρόγχο επανάληψης όπως ακριβώς περιγράψαμε προηγουμένως. Το οποιοδήποτε mutex χρησιμοποιείται θα αγνοηθεί και δεν θα τυπωθεί ενώ ταυτόχρονα θα γίνουν όλες οι απαραίτητες αλλαγές για την εκτέλεση της *emulateTSU()* και τον υπολογισμό των ζωντανών kernels.

Επιπρόσθετα, ο Preprocessor θα τυπώνει κάποιους μετρητές χρόνου (timers) ενδιάμεσα στον κώδικα, οι οποίοι θα κάνουν χρήση της *RCCE_wtime()* για να υπολογίζουν το συνολικό χρόνο εκτέλεσης της εφαρμογής.

Κεφάλαιο 6

Αξιολόγηση

6.1 Διαδικασία Αξιολόγησης

Ο κύριος στόχος της εργασίας μας ήταν η μεταφορά ενός Data-Driven Multithreading (DDM) μοντέλου προγραμματισμού και εκτέλεσης στον επεξεργαστή Intel SCC χρησιμοποιώντας την πλατφόρμα TFlux. Στο προηγούμενο κεφάλαιο αναφερθήκαμε στις απαραίτητες τροποποιήσεις που έγιναν στους διάφορους κώδικες και πλέον αυτό που μας απομένει είναι να δοκιμάσουμε να εκτελέσουμε διαφορετικές DDM εφαρμογές για να βεβαιωθούμε ότι εκτελούνται στον SCC επιτυχώς, δηλαδή με σωστά αποτελέσματα. Παράλληλα όμως, ήταν αναγκαίο να μελετήσουμε και την επίδοση στην εκτέλεση αυτών των εφαρμογών έτσι ώστε να είμαστε σε θέση να γνωρίζουμε πόσο αποδοτική είναι η υλοποίηση μας και σε ποια σημεία θα χρειαστεί να εφαρμοστούν διαρρυθμίσεις μελλοντικά.

Για την αξιολόγηση του έργου μας χρησιμοποιήσαμε κάποια από τα DDM benchmarks τα οποία είχαν υλοποιηθεί στα πλαίσια προηγούμενων ερευνών για την μελέτη της πλατφόρμας TFlux. Σύμφωνα με τη δική μας υλοποίηση, στα benchmarks αυτά δεν απαιτούνταν οποιεσδήποτε αλλαγές και αναμέναμε να εκτελεστούν στον SCC με τα ίδια αποτελέσματα όπως σε οποιοδήποτε κοινό σύστημα. Το μόνο που χρειαζόταν είναι κατά την εκτέλεση του Preprocessor να δοθεί από τον προγραμματιστή η παράμετρος *scc* έτσι ώστε ο TFlux κώδικας που θα παραχθεί να περιέχει τις τροποποιήσεις του προηγούμενου κεφαλαίου. Επίσης στο αρχείο *tsuGroup.c* του κώδικα του TSU θα έπρεπε να περιληφθεί το macro #define SCC ώστε να εκτελεστούν και οι εντολές που προσθέσαμε για την επικοινωνία των πυρήνων.

Όσον αφορά τη μελέτη της επίδοσης της υλοποίησής μας, για το κάθε DDM benchmark μετρήσαμε τους μέσους χρόνους εκτέλεσης σε διαφορετικό αριθμό πυρήνων και τους συγκρίναμε μεταξύ τους. Για την εξαγωγή του μέσου χρόνου εκτέλεσης μιας DDM εφαρμογής A σε συγκεκριμένο πλήθος K πυρήνων του SCC ακολουθήσαμε τη συνηθισμένη διαδικασία εύρεσης του μέσου όρου. Τρέξαμε την A σε K πυρήνες (10) φορές λαμβάνοντας χρόνο εκτέλεσης για την καθεμιά ξεχωριστά. Ακολούθως αφαιρέσαμε τον μεγαλύτερο και τον μικρότερο χρόνο εκτέλεσης και από τους υπόλοιπους οκτώ (8) υπολογίσαμε το μέσο όρο. Το αποτέλεσμα θα ήταν ο μέσος χρόνος εκτέλεσης της A εφαρμογής χρησιμοποιώντας K SCC πυρήνες. Για την κάθε εφαρμογή θα συγκρίναμε τους μέσους χρόνους εκτέλεσης καθώς αυξάναμε τον αριθμό των πυρήνων με σκοπό να διαπιστώσουμε αν όντως πετυχαίνεται βελτίωση της επίδοσης. Επίσης, συγκρίναμε τους χρόνους αυτούς της κάθε εφαρμογής με το χρόνο εκτέλεσης της αντίστοιχης σειριακής εφαρμογής σε C κώδικα που δεν κάνει καθόλου χρήση της πλατφόρμας TFlux. Μέσω αυτών των συγκρίσεων εξαγάγαμε γραφικές παραστάσεις σχετικά με την βελτίωση της επίδοσης (speedup) καθώς αυξάνονται οι πυρήνες, που ουσιαστικά σήμαινε την αύξηση του βαθμού παραλληλισμού.

6.2 Πειραματική Διάταξη

Για την αξιολόγηση της υλοποίησης μας, όπως προαναφέραμε, χρησιμοποιήσαμε τέσσερις (4) εφαρμογές από την σουίτα αξιολόγησης του TFlux οι οποίες μπορούν να εκμεταλλευτούν τον παραλληλισμό που τους προσφέρεται από το σύστημα. Οι

εφαρμογές αυτές προέρχονται από ευρέως γνωστά benchmarks υλοποιημένα σε DDM κώδικα. Πιο κάτω περιγράφεται εν συντομία η λειτουργία των τεσσάρων αυτών DDM benchmarks.

- MMULT

  Η εφαρμογή αφορά πολλαπλασιασμό δύο πινάκων A και B, με τέτοιο τρόπο που ο υπολογισμός μιας θέσης του τελικού πίνακα C να γίνεται ανεξάρτητα από τους υπολογισμούς των υπολοίπων θέσεων. Αυτό καθιστά το MMULT ως μια “embarrassingly” παράλληλη εφαρμογή. Δίνεται ως είσοδος το μέγεθος των δύο πινάκων A και B.

- QSORT

  Η εφαρμογή αυτή ταξινομεί ένα πίνακα εισόδου αυθαίρετου πλήθους αριθμών χωρίζοντας τον σε υπό-πίνακες και ταξινομώντας τον κάθε υπό-πίνακα ξεχωριστά και παράλληλα. Στη συνέχεια εφαρμόζεται μια συγχώνευση των υπό-πινάκων σε ένα τελικό ταξινομημένο πίνακα. Δίνεται ως είσοδος το πλήθος των αριθμών που πρόκειται να ταξινομηθούν.

- RK4

  Η εφαρμογή υλοποιεί ένα αλγόριθμο ο οποίος χρησιμοποιεί 4 βήματα για επίλυση προβλημάτων αρχικής τιμής σε κανονικές διαφορικές εξισώσεις. Το κάθε βήμα είναι ένας βρόγχος επανάληψης που υπολογίζει μια τιμή πάνω στην οποία βασίζεται το επόμενο βήμα. Δίνεται ως είσοδος το πλήθος των επαναλήψεων που εκτελεί το κάθε βήμα.

- TRAPEZSIMPLE

  Η εφαρμογή υπολογίζει το ολοκλήρωμα μιας συνάρτησης σε ένα ορισμένο διάστημα. Το διάστημα αυτό χωρίζεται σε υπό-διαστήματα, στο καθένα από τα οποία εφαρμόζεται, ξεχωριστά και παράλληλα, η τραπεζοειδής μέθοδος. Το άθροισμα όλων των ενδιάμεσων αποτελεσμάτων είναι η τιμή του ολοκληρώματος. Δίνεται ως είσοδος ο αριθμός των υπό-διαστημάτων στο οποίο θα διαιρεθεί το αρχικό διάστημα του ολοκληρώματος.

Οι εφαρμογές εκτελέστηκαν στον επεξεργαστή Intel SCC, έτσι ακριβώς όπως τον έχουμε περιγράψει στο Κεφάλαιο 3, και έγιναν μετρήσεις για διάφορα μεγέθη εισόδου στην κάθε εφαρμογή. Στον Πίνακα 6.1 παρουσιάζονται όλα τα μεγέθη

εισόδου που χρησιμοποιήθηκαν για το κάθε benchmark καθώς και η τιμή του unroll factor που πάντα παρέμενε ίση με 1. Οι διάφορες μετρήσεις έγιναν λαμβάνοντας υπόψη μόνο το χρόνο εκτέλεσης του παράλληλου μέρους της εφαρμογής αφού αυτό είναι το κομμάτι που μας ενδιαφέρει. Οποιαδήποτε σειριακά μέρη, όπως επαληθεύσεις και εκτυπώσεις αποτελεσμάτων, έχουν αγνοηθεί.

Χρησιμοποιήθηκε η RockyLake έκδοση του Intel SCC ενώ οι πυρήνες του επεξεργαστή, το δίκτυο αλληλοσύνδεσης (mesh) και οι DDR διαχειριστές μνήμης (memory controllers) είχαν προγραμματιστεί να τρέχουν σε συχνότητα 800 MHz. Το σύστημα περιέχει συνολικά 32GB κύρια μνήμη, τρέχει λειτουργικό σύστημα Linux kernel σε κάθε πυρήνα, το οποίο παρέχεται από το RCCE SCC Kit 1.3.0, και χρησιμοποιεί τον μεταγλωττιστή gcc v3.4.5 για το cross compiling των εφαρμογών. Ο host υπολογιστής (MCPC) που είναι υπεύθυνος για την εκτέλεση των εφαρμογών στον SCC είναι ένα 64-bit PC με επεξεργαστή Intel i7 συχνότητας 3.7GHz και 4GB μνήμη. Η διασύνδεση του host PC με τον SCC γίνεται με τη χρήση ενός PCIe Expansion Card και ενός PCIe x4 cable. Για την μεταφορά και εκτέλεση των εφαρμογών στο σύστημα χρησιμοποιείται το RCCE 1.3.0 tool-chain.

Benchmark	**Input Sizes**						**Unroll**
MMULT	64 x 64		128 x 128		256 x 256		1
QSORT	10K	20K	50K	100K	200K	400K	1
RK4	1024		2048		4096		1
TRAPEZSIMPLE	2^{10}		2^{12}		2^{14}		2^{16}	1

Πίνακας 6.1: *Μεγέθη Εισόδου Εφαρμογών*

6.3 Παρουσίαση και Ανάλυση Αποτελεσμάτων

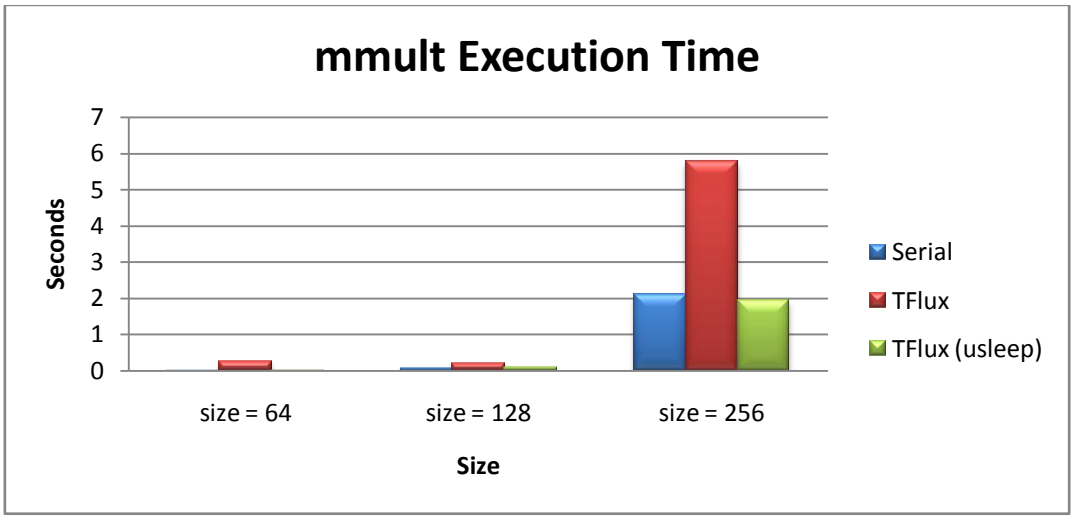
Όπως έχουμε αναφέρει προηγουμένως, πρωταρχικός στόχος της εργασίας μας ήταν να καταφέρουμε να πετύχουμε την ορθή εκτέλεση DDM εφαρμογών στον επεξεργαστή Intel SCC χωρίς σφάλματα. Ο στόχος αυτός επιτεύχθηκε αφού καταφέραμε να τρέξουμε επιτυχώς τα πιο πάνω benchmarks με ορθά αποτελέσματα

για όλα τα πιθανά μεγέθη εισόδου. Επαληθεύσαμε τα αποτελέσματα μας με τα αντίστοιχα που προκύπτουν από την εκτέλεση των ίδιων εφαρμογών σε οποιοδήποτε κοινό multicore μέσω του TFlux. Πλέον, έχουμε μια υλοποίηση η οποία δίνει την ικανότητα στο χρήστη να προγραμματίσει και να εκτελέσει εφαρμογές στο κατανεμημένης μνήμης σύστημα του SCC χρησιμοποιώντας το μοντέλο Data-Driven Multithreading και την πλατφόρμα TFlux. Η αμέσως επόμενη ενέργεια που έχουμε να κάνουμε είναι να μελετήσουμε τις επιδόσεις που παίρνουμε κατά την εκτέλεση της κάθε εφαρμογής ξεχωριστά, καθώς αυξάνουμε τον αριθμό των πυρήνων που χρησιμοποιούνται. Για την περαιτέρω βελτίωση της επίδοσης δοκιμάσαμε και κάποιες επιπλέον εκτελέσεις στις οποίες προσπαθήσαμε με κάποιες αλλαγές στον TFlux κώδικα της εφαρμογής να αναστέλλουμε για κάποιο χρονικό διάστημα την εκτέλεση του thread που τρέχει το TSU. Γνωρίζοντας ότι οι πυρήνες του SCC δεν είναι multithreaded, επιδιώξαμε με κλήση της συνάρτησης *usleep()* να απενεργοποιούμε το TSU thread για 1000 μικροδευτερόλεπτα μετά από κάθε κλήση της *emulateTSU()*. Θα δούμε αν πετυχαίνουμε βελτίωση και πόσο πιο αποδοτική είναι η κάθε εκτέλεση συγκριτικά με την αντίστοιχη της σειριακής εφαρμογής. Παρακάτω, θα δούμε για κάθε εφαρμογή την εκτέλεσή της σε ένα μόνο πυρήνα του SCC με τρείς διαφορετικούς τρόπους: την εκτέλεση της αντίστοιχης σειριακής εφαρμογής, την εκτέλεση της δικής μας υλοποίησης μέσω TFlux και την εκτέλεση της δικής μας υλοποίησης μέσω TFlux προσθέτοντας την τεχνική αναστολής του ενός thread που αναφέραμε πιο πάνω. Για την μελέτη των χρόνων εκτέλεσης και της επίδοσης όσο αυξάνονται οι πυρήνες, θα χρησιμοποιήσουμε μόνο τα αποτελέσματα της τρίτης εκτέλεσης τα οποία θα είναι κανονικοποιημένα σύμφωνα με τους χρόνους της σειριακής εκτέλεσης. Σ' αυτό το υποκεφάλαιο προσπαθήσαμε να δώσουμε έμφαση στα πιο σημαντικά αποτελέσματα. Στο Παράρτημα Α περιέχονται και οι υπόλοιπες γραφικές παραστάσεις με τα αποτελέσματα από τις διάφορες άλλες εκτελέσεις που δοκιμάσαμε.

6.3.1 MMULT Benchmark

Στο Σχήμα 6.1(α) παρουσιάζονται οι χρόνοι εκτέλεσης της εφαρμογής MMULT σε ένα μόνο SCC πυρήνα σε τρεις διαφορετικές περιπτώσεις. Με μπλε χρώμα αντιπροσωπεύεται ο χρόνος εκτέλεσης της αντίστοιχης σειριακής εφαρμογής σε C κώδικα χωρίς καθόλου χρήση της πλατφόρμας TFlux. Οι υπόλοιπες δύο αφορούν εκτελέσεις της DDM εφαρμογής μέσω της πλατφόρμας TFlux χρησιμοποιώντας

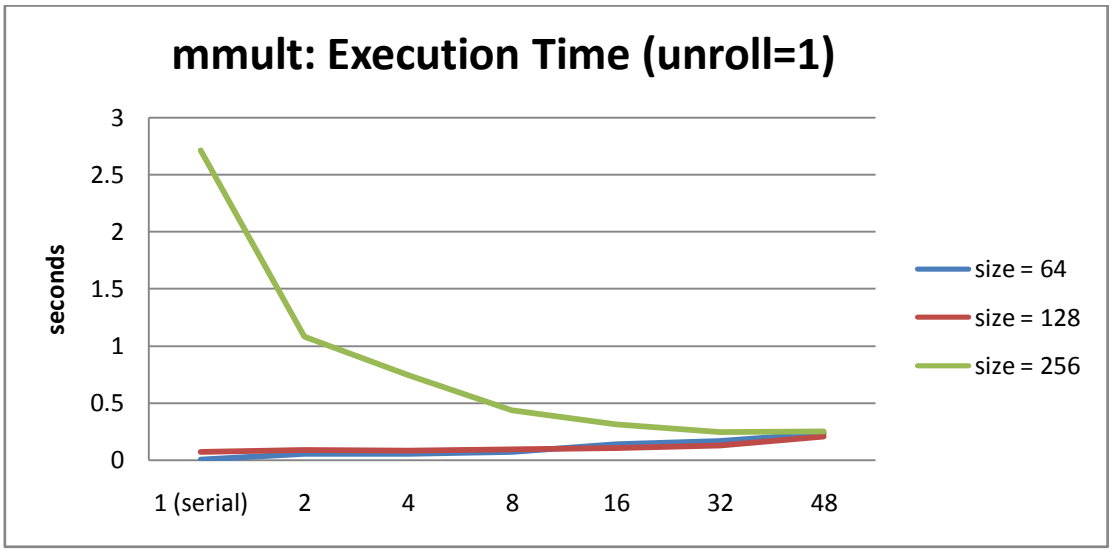
μόνο ένα (1) kernel. Με κόκκινο χρώμα είναι η απλή εκτέλεση ενώ με πράσινο χρώμα είναι η εκτέλεση που προσπαθεί να αναστέλλει τη λειτουργία του TSU thread ανά διαστήματα. Σκοπός αυτού του γραφήματος είναι να παρουσιάσουμε το overhead που προσθέτει το TFlux κατά την εκτέλεση μιας εφαρμογής στον επεξεργαστή. Διακρίνουμε ότι το κόστος χρήσης της πλατφόρμας TFlux σε σχέση με τη σειριακή εφαρμογή είναι πάρα πολύ μεγάλο. Αυτό οφείλεται στη φύση των SCC πυρήνων οι οποίοι δεν υποστηρίζουν multithreading και αποδεικνύεται τρανά από την τρίτη εκτέλεση κατά την οποία βγάζουμε εκτός πυρήνα το ένα από τα δύο threads για κάποιο χρονικό διάστημα. Παρατηρούμε ότι ο χρόνος εκτέλεσης έχει μειωθεί σε μεγάλο βαθμό και έχει κατέβει στα ίδια επίπεδα με αυτά της σειριακής εκτέλεσης.



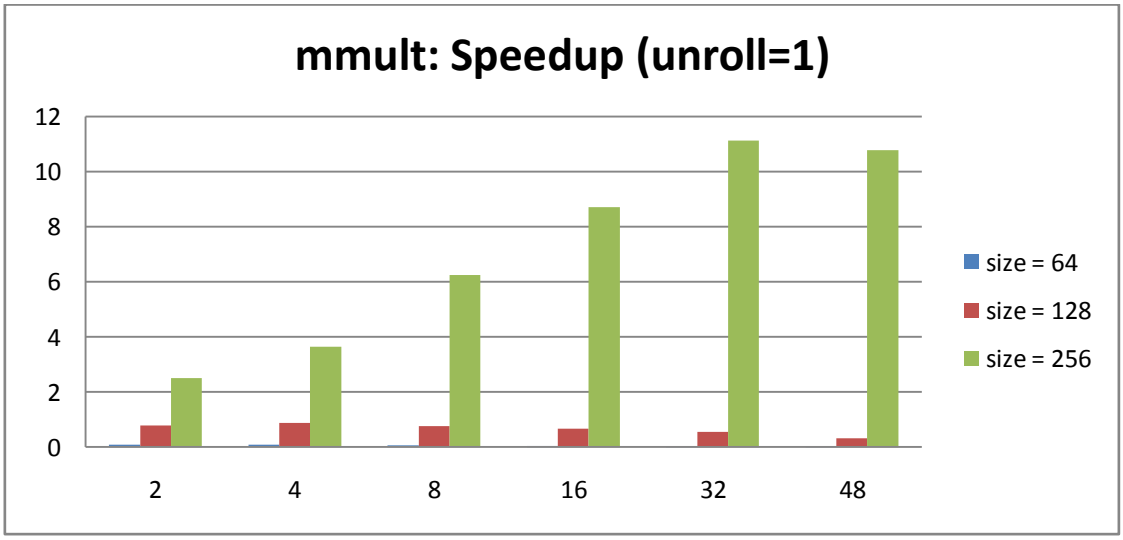
Σχήμα 6.1(α): *TFlux-SCC Overhead εφαρμογής MMULT*

Στο Σχήμα 6.1(β) παρουσιάζεται ο χρόνος εκτέλεσης του DDM benchmark MMULT στον επεξεργαστή Intel SCC αλλάζοντας κάθε φορά τον αριθμό των kernels και συνάμα των SCC πυρήνων που θα χρησιμοποιηθούν. Είναι φανερό ότι για σχετικά μικρά μεγέθη εισόδου, συγκεκριμένα πινάκων 64 και 128 θέσεων, ο χρόνος εκτέλεσης όσο αυξάνουμε τους πυρήνες διατηρείται σε ίδια επίπεδα με αυτό της σειριακής εκτέλεσης. Αιτία του γεγονότος αυτού είναι ότι κάθε πυρήνας αναλαμβάνει ένα πάρα πολύ μικρό υπολογισμό του πολλαπλασιασμού ενώ στην ουσία δεν συμφέρει να διασπαστεί το πρόβλημα για τόσο μικρό μέγεθος εισόδου. Όταν αυξήσουμε το μέγεθος των πινάκων που θα πολλαπλασιαστούν σε 256 θέσεις παρατηρούμε ότι με την πρόσθεση περισσότερων πυρήνων ο χρόνος εκτέλεσης της εφαρμογής μειώνεται σημαντικά γιατί αξιοποιείται καλύτερα ο παραλληλισμός του συστήματος. Επομένως, μελετώντας τους χρόνους αυτούς μπορούμε να συμπεράνουμε ότι για πίνακες μεγάλου μεγέθους πετυχαίνεται τεράστια αύξηση της επίδοσης. Το Σχήμα 6.1(γ) μας παρουσιάζει την επίδοση αυτή η οποία έχει

υπολογιστεί με βάση τον χρόνο εκτέλεσης της σειριακής εφαρμογής MMULT. Όσο αυξάνουμε τον αριθμό των kernels-πυρήνων μπορούμε να πούμε ότι η επίδοση βελτιώνεται με μορφή linear speedup. Κάθε φορά που διπλασιάζουμε τον αριθμό των πυρήνων ο χρόνος εκτέλεσης μοιράζεται σχεδόν στο μισό. Με αυτά τα δεδομένα μπορούμε να πούμε ότι η υλοποίηση μας είναι πολύ αποδοτική για την εκτέλεση της εφαρμογής MMULT.



Σχήμα 6.1(β): Χρόνοι εκτέλεσης MMULT

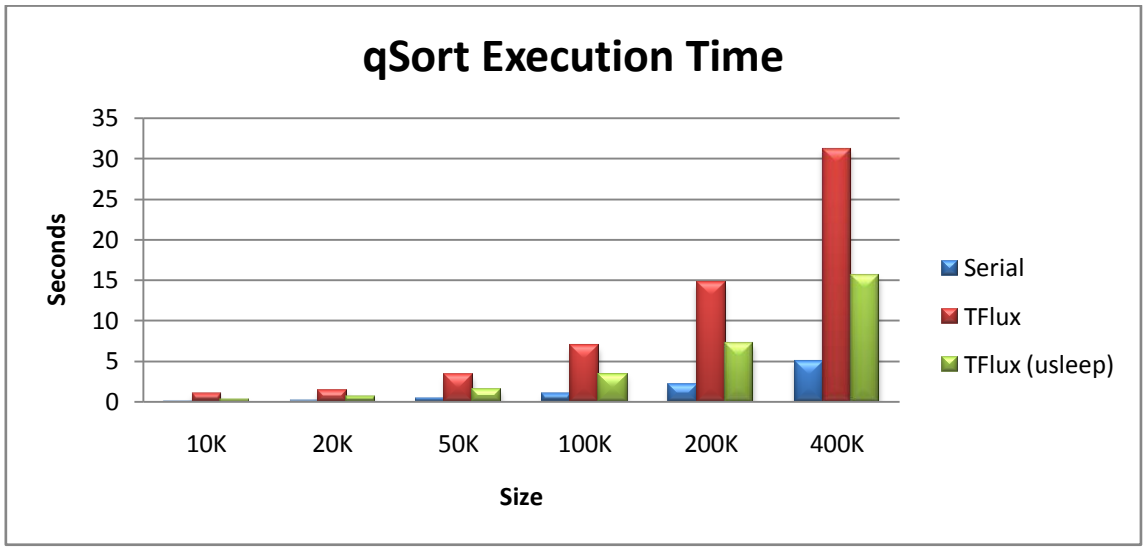


Σχήμα 6.1(γ): Επίδοση MMULT

6.3.2 QSORT Benchmark

Κατά την εκτέλεση του DDM benchmark QSORT με ένα kernel σε ένα πυρήνα του SCC παρατηρήσαμε ότι προστίθεται μεγάλο κόστος στο χρόνο εκτέλεσης σε σχέση

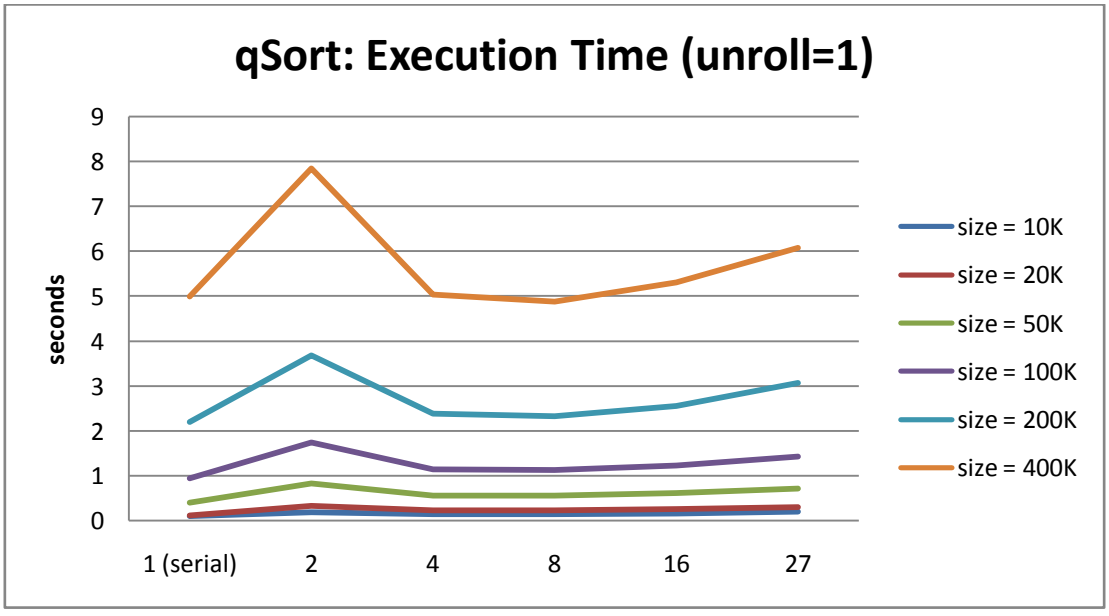
με αυτόν από την εκτέλεση της αντίστοιχης σειριακής εφαρμογής. Όπως φαίνεται και στο Σχήμα 6.2(α) η σειριακή εκτέλεση σε ένα SCC πυρήνα πετυχαίνει πάντα καλύτερο χρόνο ακόμα κι αν χρησιμοποιήσουμε την τεχνική της απενεργοποίησης του ενός thread για ορισμένο χρονικό διάστημα. Με τη συγκεκριμένη τεχνική, ναι μεν ο χρόνος μειώνεται σχεδόν στο μισό από αυτόν της απλής TFlux εκτέλεσης, αλλά και πάλι το overhead που προσθέτει η χρήση της πλατφόρμας TFlux είναι σχετικά μεγάλο. Στο επόμενο κεφάλαιο θα αναφέρουμε κάποιες επιπλέον μεθόδους που μπορούμε να εφαρμόσουμε μελλοντικά για να βελτιώσουμε ακόμα περισσότερο τους χρόνους εκτέλεσης. Στη συνέχεια, θα μελετήσουμε τους χρόνους εκτέλεσης και την επίδοση που πετυχαίνει η εκτέλεση του QSORT, όσο αυξάνουμε τον αριθμό των υπολογιστικών μονάδων, σε σύγκριση με τη σειριακή εκτέλεση στον ένα SCC πυρήνα.



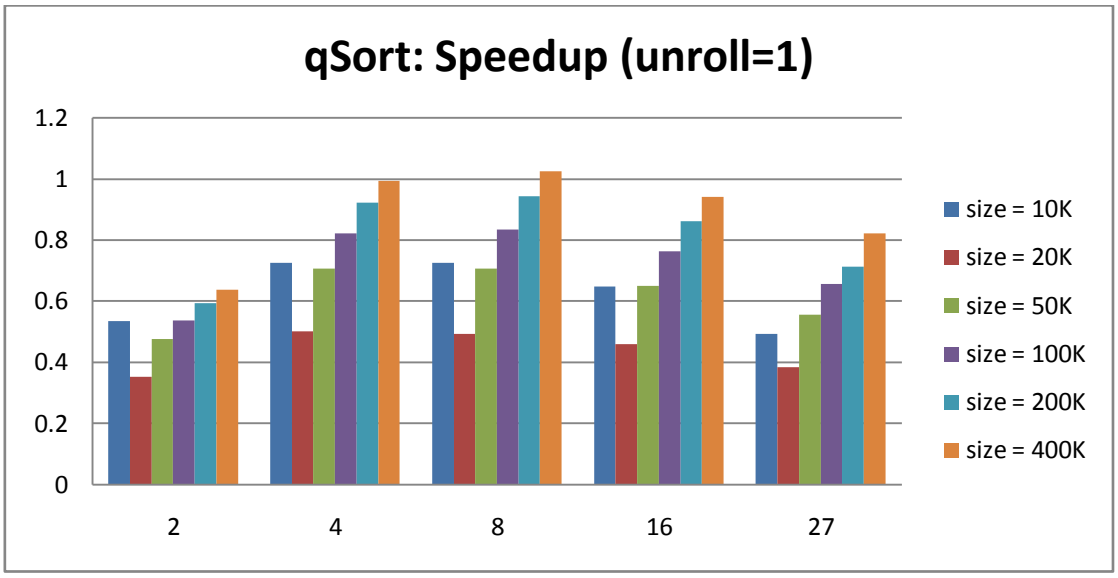
Σχήμα 6.2(α): *TFlux-SCC Overhead εφαρμογής QSORT*

Όπως ήταν αναμενόμενο βάσει της προηγούμενης γραφικής παράστασης του Σχήματος 6.2(α), και το Σχήμα 6.2(β) μας παρουσιάζει ότι η υλοποίηση μας δεν πετυχαίνει καθόλου μείωση του χρόνου εκτέλεσης κάτω από αυτόν της σειριακής εκτέλεσης. Το συγκεκριμένο συμπέρασμα αφορά οποιοδήποτε μέγεθος εισόδου δίνεται προς ταξινόμηση. Συγκεκριμένα, η εκτέλεση σε 2 SCC πυρήνες αυξάνει κατά πολύ το χρόνο σε σχέση με το σειριακό. Με την περαιτέρω πρόσθεση πυρήνων ο χρόνος κατεβαίνει στα ίδια επίπεδα με το σειριακό, αλλά ποτέ δεν είναι καλύτερος. Τα αποτελέσματα από την εκτέλεση του QSORT φαίνονται αναλυτικότερα στο Σχήμα 6.2(γ) το οποίο μας παρουσιάζει την αυξομείωση της επίδοσης για διαφορετικό αριθμό πυρήνων. Όπως παρατηρούμε, οποιοδήποτε πλήθος αριθμών και να δοθεί προς ταξινόμηση, η εκτέλεση με 2 kernels-πυρήνες

δεν είναι καθόλου αποδοτική. Η επίδοση, συγκριτικά με τη σειριακή εκτέλεση, πέφτει σχεδόν στο μισό και αυτό ίσως να οφείλεται στην τελική συγχώνευση των δύο ταξινομημένων υπό-πινάκων που προέρχονται από τους 2 πυρήνες. Οι υπό-πίνακες αν και έχουν το μισό μέγεθος του αρχικού πίνακα εντούτοις είναι αρκετά μεγάλοι με αποτέλεσμα το κόστος του *insertion-sort* αλγορίθμου που εκτελείται πάνω τους για το τελικό αποτέλεσμα να επιφέρει την αύξηση του χρόνου εκτέλεσης και παράλληλα τη μείωση της επίδοσης. Αυτό το κόστος δεν υπάρχει στην απλή ταξινόμηση του αρχικού πίνακα από τον ένα πυρήνα, γι' αυτό και ο σειριακός χρόνος εκτέλεσης είναι καλύτερος. Παρόμοια εξήγηση ισχύει και για τη μείωση της επίδοσης κατά τη χρήση 27 πυρήνων. Ο χρόνος συγχώνευσης μεγάλου αριθμού υπό-πινάκων, άσχετα με το μέγεθός τους, είναι πολύ μεγαλύτερος από το χρόνο που κερδίζεται από την επιμέρους ταξινόμηση των 27 κομματιών του αρχικού πίνακα, γι' αυτό και επίδοση κατεβαίνει κάτω από 1. Όσον αφορά τις υπόλοιπες εκτελέσεις σε 4, 8 και 16 πυρήνες, παρατηρούμε ότι ο χρόνος εκτέλεσης δεν έχει μεγάλη διαφορά από αυτόν της σειριακής και η επίδοση παραμένει στα ίδια επίπεδα. Γενικά, μπορούμε να πούμε ότι η εφαρμογή αυτή δεν είναι η καταλληλότερη για εκμετάλλευση του παραλληλισμού που προσφέρει το σύστημα. Ο αρχικός πίνακας μπορεί να μοιραστεί σε επιμέρους μικρότερα κομμάτια των οποίων η ταξινόμηση μπορεί να γίνει παράλληλα, αλλά η ταξινόμηση των κομματιών αυτών για το τελικό αποτέλεσμα γίνεται με καθαρά σειριακό τρόπο. Το κόστος του σειριακού αυτού υπολογισμού, σε συνδυασμό με το overhead που προσθέτει η πλατφόρμα TFlux στην εφαρμογή, όπως είδαμε στο Σχήμα 6.2(α), αναγκάζουν το κέρδος από την παράλληλη ταξινόμηση των υπό-πινάκων, ουσιαστικά, να χάνεται.

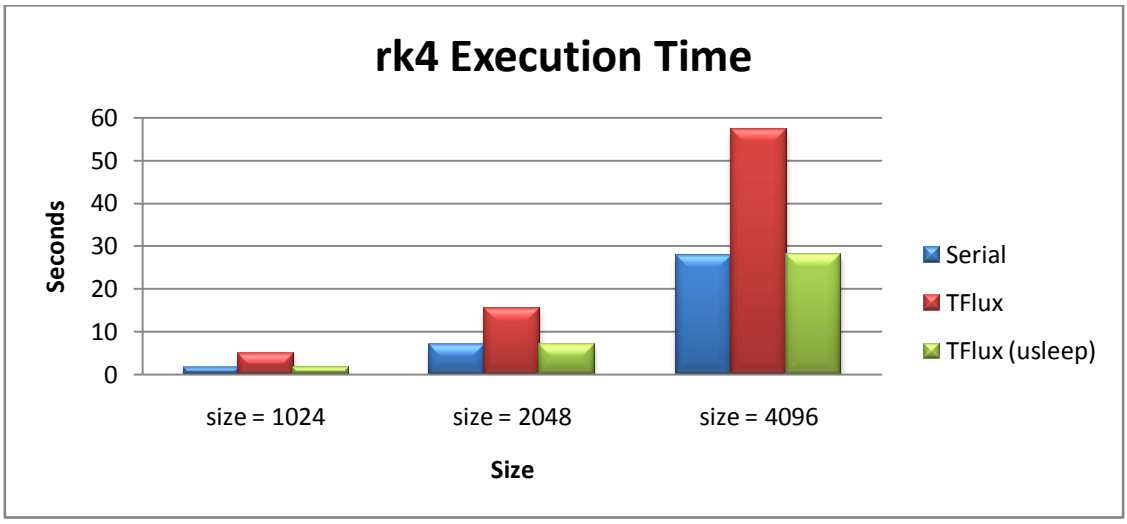


Σχήμα 6.2(β): *Χρόνοι εκτέλεσης QSORT*



Σχήμα 6.2(γ): Επίδοση QSORT

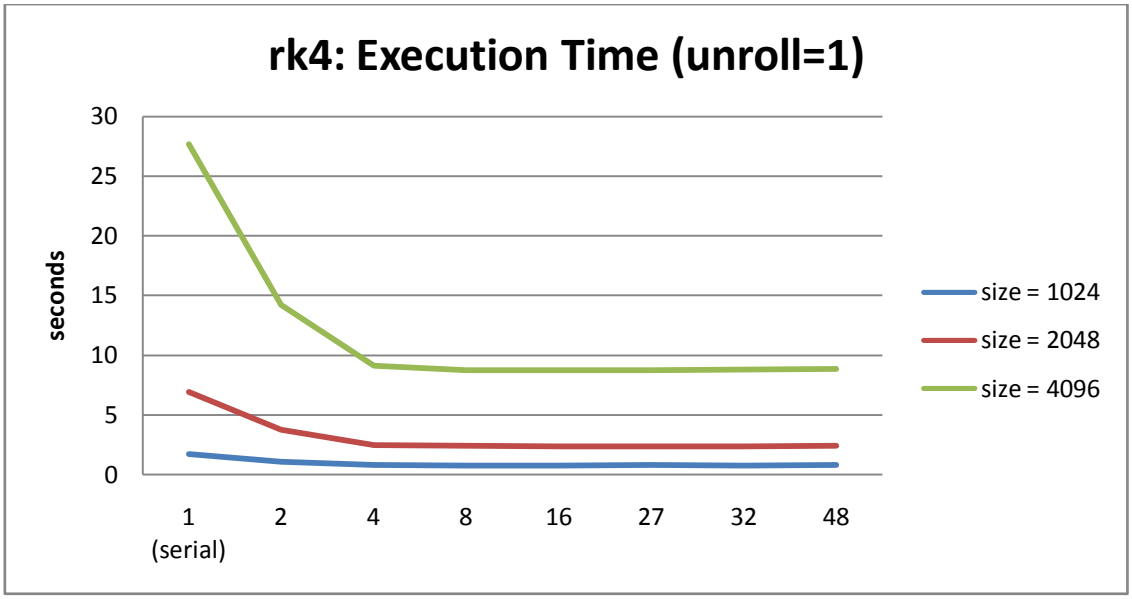
6.3.3 RK4 Benchmark



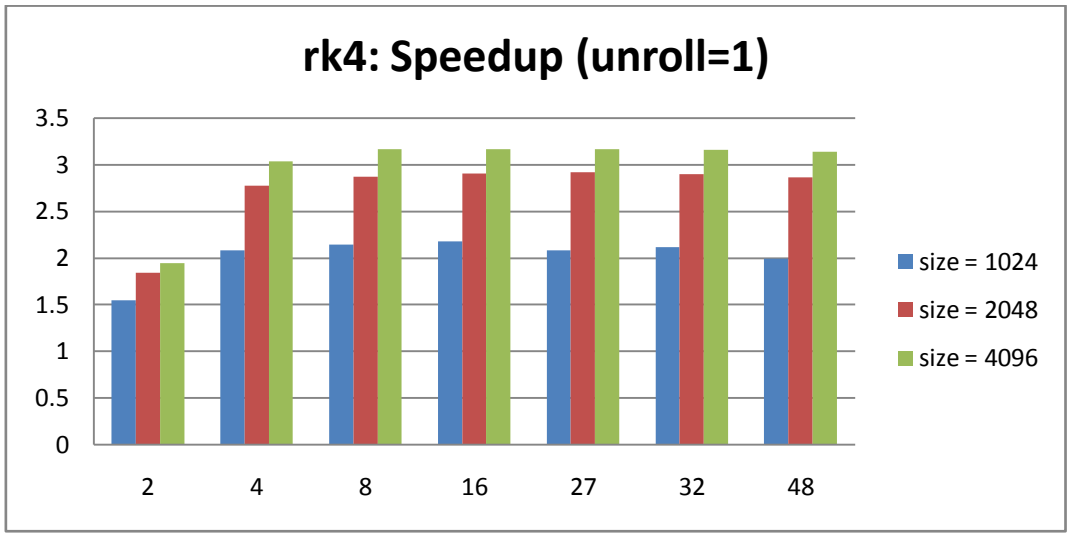
Σχήμα 6.3(α): TFlux-SCC Overhead εφαρμογής RK4

Η εφαρμογή RK4, όπως, προαναφέραμε αποτελείται από 4 βρόγχους επανάληψης, εξαρτώμενους μεταξύ τους, που ο καθένας υπολογίζει μια τιμή πάνω στην οποία θα βασιστεί ο αμέσως επόμενος βρόγχος για τον δικό του υπολογισμό. Κατά την εκτέλεση του RK4 μέσω του TFlux και χρησιμοποιώντας 1 kernel όπως φαίνεται στο Σχήμα 6.3(α), παρατηρούμε ότι ο χρόνος εκτέλεσης διπλασιάζεται σε σχέση με τη σειριακή εκτέλεση της εφαρμογής. Ο λόγος εξακολουθεί να είναι ο ίδιος με αυτόν των προηγούμενων εφαρμογών αφού η πλατφόρμα αναγκάζει το σύστημα να εκτελεί 2 threads σε ένα single-threaded πυρήνα. Χρησιμοποιώντας τη συνάρτηση

usleep() για να αναστείλουμε σε τακτά χρονικά διαστήματα την εκτέλεση του TSU thread, ο χρόνος μειώνεται στο μισό φτάνοντας στα ίδια επίπεδα με τον σειριακό.



Σχήμα 6.3(β): *Χρόνοι εκτέλεσης RK4*

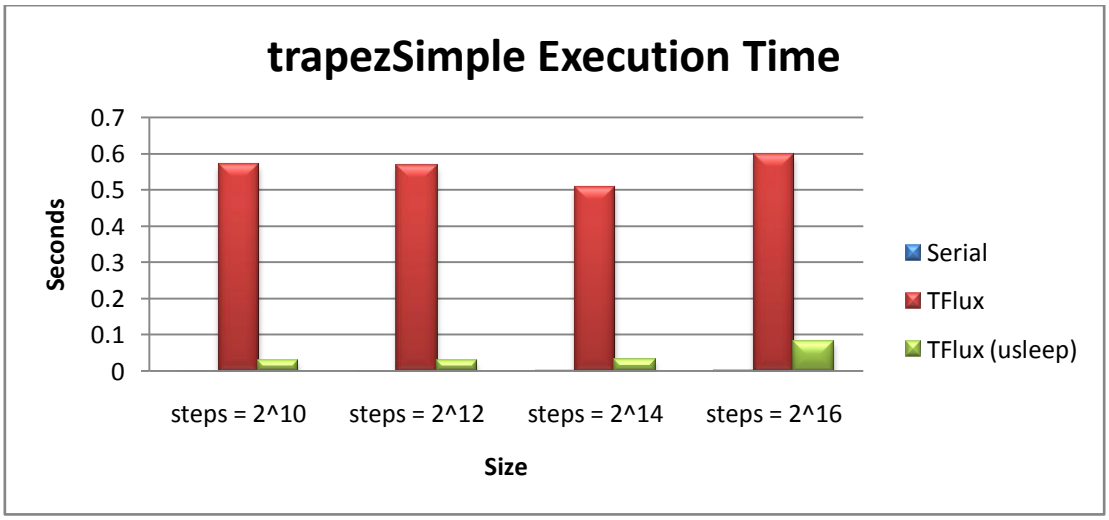


Σχήμα 6.3(γ): *Επίδοση RK4*

Οι πράξεις που περιλαμβάνει μια επανάληψη ενός βρόγχου του RK4 είναι ανεξάρτητες από αυτές των υπόλοιπων επαναλήψεων του βρόγχου. Επομένως, κατά την παραλληλοποίηση της εφαρμογής με το TFlux ο κάθε βρόγχος μοιράζει τις επαναλήψεις του ισότιμα στις διαθέσιμες υπολογιστικές μονάδες ενώ ταυτόχρονα ο επόμενος βρόγχος δεν ξεκινά ποτέ την εκτέλεσή του αν δεν ολοκληρωθεί ο προηγούμενος. Τα Σχήματα 6.3(β) και 6.3(γ) μας παρουσιάζουν αντίστοιχα, το χρόνο εκτέλεσης και την επίδοση της DDM εφαρμογής RK4 σε διαφορετικό αριθμό πυρήνων, και στην οποία χρησιμοποιείται η τεχνική της μερικής

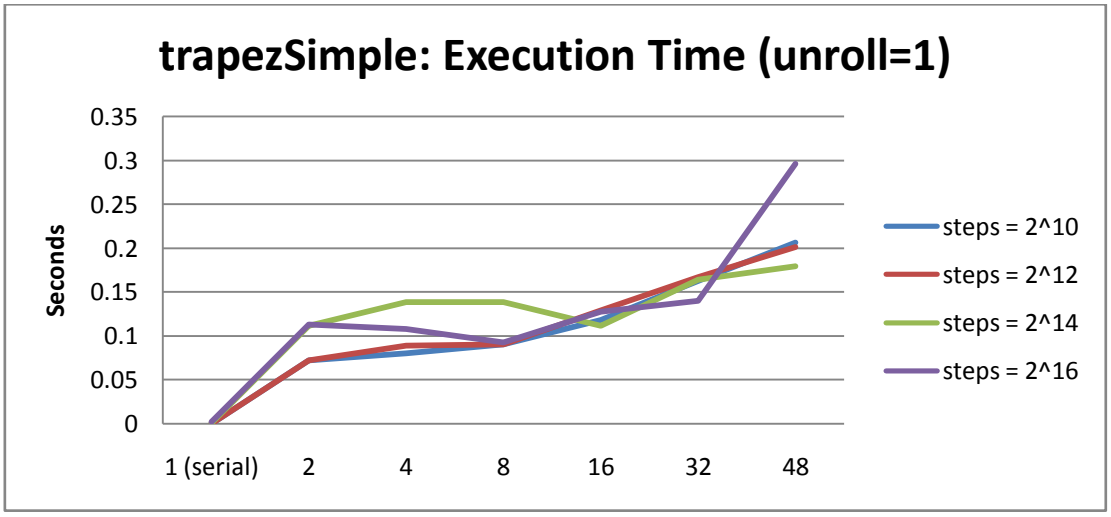
απενεργοποίησης του ενός από τα δύο threads. Παρατηρούμε ότι, για οποιοδήποτε πλήθος επαναλήψεων στους βρόγχους, πετυχαίνεται πολύ μεγάλη βελτίωση του χρόνου και της επίδοσης συγκριτικά με την σειριακή εκτέλεση. Αυτό είναι λογικό αφού οι ανεξάρτητες επαναλήψεις κάθε βρόγχου μοιράζονται στους πυρήνες και ο βρόγχος ολοκληρώνει την εκτέλεση του πολύ πιο νωρίς από την εκτέλεσή του σε ένα μόνο πυρήνα. Παρόλα αυτά, η πρόσθεση πυρήνων περάν των 4 φαίνεται να είναι αχρείαστη αφού δεν παρατηρείται περεταίρω βελτίωση. Κατά την εκτέλεση με 8, 16, 27, 32 και 48 πυρήνες, χρόνος και επίδοση παραμένουν σε σταθερά επίπεδα με τιμές ίσες με αυτές της εκτέλεσης σε 4 πυρήνες. Επομένως, η εφαρμογή δείχνει ανίκανη να εκμεταλλευτεί τον ψηλό παραλληλισμό του συστήματος και αυτό ίσως να οφείλεται στις πολλές προσβάσεις που γίνονται στην κοινή μνήμη (όπου βρίσκονται τα δεδομένα) όταν έχουμε περισσότερους πυρήνες. Η κοινή μνήμη είναι εκτός του επεξεργαστή (off-chip) επομένως όσους περισσότερους πυρήνες έχουμε τόσες περισσότερες είναι οι προσβάσεις στη μνήμη με αποτέλεσμα τα πολλά reads να επιφέρουν μεγάλο κόστος στο χρόνο εκτέλεσης. Ταυτόχρονα δημιουργείται και μεγάλο overhead στο δίκτυο του SCC αφού έχουμε περισσότερα μηνύματα να διακινούνται. Όλα αυτά τα κόστη, λοιπόν, αντισταθμίζουν τα κέρδη από τον υψηλό παραλληλισμό της εφαρμογής αναγκάζοντας τους χρόνους να παραμένουν αμετάβλητοι. Γενικά, η εφαρμογή RK4, χρησιμοποιώντας το TFlux στον SCC, παρουσιάζει μια αποδοτική εκτέλεση μέχρι ενός συγκεκριμένου αριθμού πυρήνων. Περισσότεροι πυρήνες δεν κοστίζουν στην επίδοση αλλά μας είναι αχρείαστη η χρησιμοποίηση τους.

6.3.4 TRAPEZSIMPLE Benchmark

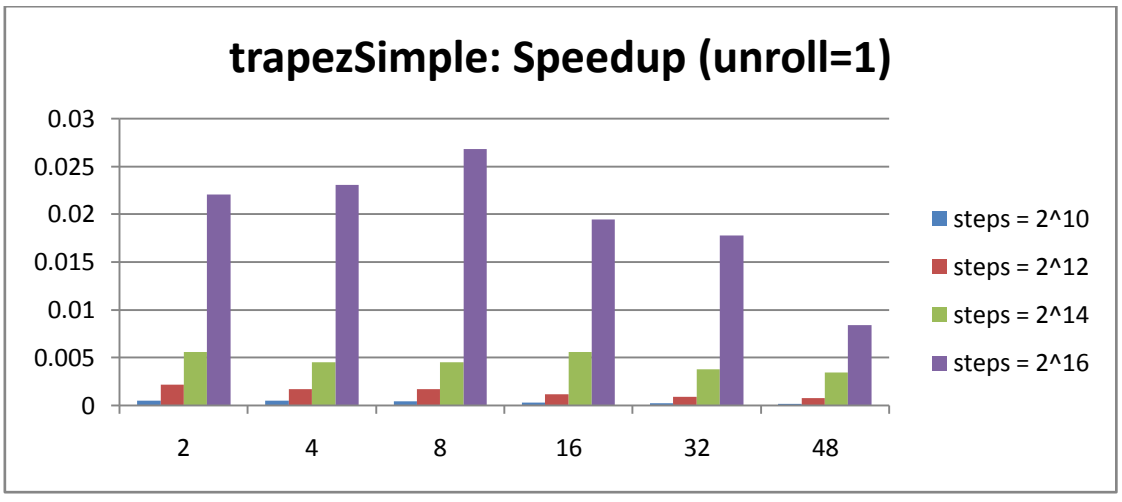


Σχήμα 6.4(α): *TFlux-SCC Overhead εφαρμογής TRAPEZSIMPLE*

Η εφαρμογή TRAPEZSIMPLE, όπως μας δείχνει και το Σχήμα 6.4(α), παρουσιάζει τα χειρότερα αποτελέσματα από τις υπόλοιπες τρεις εφαρμογές όσον αφορά τη χρήση της πλατφόρμας TFlux για την εκτέλεσή της. Σύμφωνα με τις μετρήσεις μας, το overhead που δημιουργείται από το TFlux είναι τεράστιο, με το χρόνο εκτέλεσης σε ένα πυρήνα να εκτοξεύεται και να ξεπερνά τον αντίστοιχο χρόνο της σειριακής εφαρμογής κατά πολλές εκατοντάδες φορές. Εδώ, το πρόβλημα των single-threaded πυρήνων είναι τόσο μεγάλο όσο ποτέ, αφού στην τρίτη εκτέλεση με την τακτική έξοδο του TSU thread από τον πυρήνα βλέπουμε το χρόνο να μειώνεται μέχρι και 20 φορές. Παρόλα αυτά, το overhead σε σχέση με το σειριακό χρόνο εκτέλεσης παραμένει σε πολύ ψηλά επίπεδα, γεγονός που δεν μας αφήνει καθόλου ικανοποιημένους. Στις επόμενες γραφικές παραστάσεις θα μελετήσουμε τους χρόνους και τις επιδόσεις που προκύπτουν από την εκτέλεση της εφαρμογής σε διαφορετικό αριθμό kernels-πυρήνων συγκριτικά και πάλι με το σειριακό.



Σχήμα 6.4(β): *Χρόνοι εκτέλεσης TRAPEZSIMPLE*



Σχήμα 6.4(γ): *Επίδοση TRAPEZSIMPLE*

Το Σχήμα 6.4(β) μας δείχνει ότι ο χρόνος εκτέλεσης του TRAPEZSIMPLE, για οποιοδήποτε πλήθος υπό-διαστημάτων, μεγαλώνει όσο αυξάνουμε τον αριθμό των πυρήνων που χρησιμοποιούνται. Η μόνη βελτίωση στο χρόνο παρατηρείται όταν αυξάνουμε τους πυρήνες από 4 στους 8 αλλά ακόμα και εκεί η διαφορά είναι ελάχιστη. Αντίστοιχη μορφή έχει και η γραφική παράσταση της επίδοσης στο Σχήμα 6.4(γ). Η επίδοση ίσως να είναι καλύτερη όταν χρησιμοποιούμε 8 πυρήνες παρά 4, αλλά δεν παύει να είναι μηδαμινή σε σχέση με την εκτέλεση της σειριακής εφαρμογής. Από τα αποτελέσματα αυτά συμπεραίνουμε ότι η εκτέλεση της εφαρμογής σε συνδυασμό με τα συγκεκριμένα μεγέθη εισόδου, που αντιπροσωπεύουν το διάστημα στο οποίο θα εφαρμοστεί το ολοκλήρωμα, δεν είναι καθόλου αποδοτική και θα ήταν καλύτερο, αντί να διασπαστεί ο φόρτος εργασίας στους πυρήνες, να εκτελεστεί ολόκληρος μόνο σε ένα. Αυτό ίσως να οφείλεται στα πολύ μικρά διαστήματα που χρησιμοποιήσαμε αλλά και στην διαδικασία του τελικού *reduction* που γίνεται στα ενδιάμεσα αποτελέσματα η οποία θα πρέπει να περιμένει τον υπολογισμό των τιμών από τον κάθε πυρήνα και μετά να τις αθροίσει.

6.4 Περίληψη Αποτελεσμάτων

Αξιολογήσαμε την υλοποίηση μας με 4 DDM benchmarks διαφορετικά μεταξύ τους, έτσι ώστε τα αποτελέσματα που θα παίρναμε να ήταν αντιπροσωπευτικά για όλες τις κατηγορίες εφαρμογών. Αυτό αποδεικνύεται και από την διαφορετικότητα των αποτελεσμάτων που είδαμε στο προηγούμενο υποκεφάλαιο. Αρχικά, είδαμε την πλατφόρμα TFlux να προσθέτει τεράστιο overhead εξαιτίας των δύο threads, application και TSU, που τρέχουν ταυτόχρονα στον κάθε πυρήνα, γεγονός που συμβαίνει για όλες τις εφαρμογές. Αναστέλλοντας την εκτέλεση του TSU thread ανά τακτά χρονικά διαστήματα και για διάρκεια περίπου ίση με 1 millisecond είδαμε τους χρόνους εκτέλεσης να μειώνονται αισθητά.

Όσον αφορά τις επιδόσεις των εφαρμογών, προσπαθήσαμε να προσφέρουμε σε αυτές όλη την ισχύ παραλληλισμού που παρέχει ο Intel SCC ώστε να μελετήσουμε πόσο αποδοτική γίνεται η εκτέλεσή τους. Τα αποτελέσματα από τις εκτελέσεις, όπως προαναφέραμε, δεν είναι ίδια για όλες τις εφαρμογές κι αυτό οφείλεται στην λειτουργία της εκάστοτε εφαρμογής. Αν οι υπολογισμοί της υπό εκτέλεσης εφαρμογής μπορούν να μοιραστούν και να γίνουν ξεχωριστά σε διαφορετικές υπολογιστικές μονάδες χωρίς να είναι αλληλοεξαρτώμενοι, τότε η εφαρμογή μπορεί

να εκμεταλλευτεί πλήρως τις ικανότητες του συστήματος και να πετύχει πολύ ψηλές επιδόσεις. Η εφαρμογή MMULT είναι ένα παράδειγμα μιας τέτοιας embarrassingly parallel εφαρμογής. Ο υπολογισμός μιας θέσης του τελικού πίνακα είναι εντελώς ανεξάρτητος από αυτόν των υπόλοιπων θέσεων. Επομένως, όλες οι πράξεις που εκτελεί η εφαρμογή μπορούν να διαμοιραστούν σε όσο το δυνατόν περισσότερους πυρήνες χωρίς καμιά επικοινωνία μεταξύ τους. Βέβαια, σημαντικό ρόλο διαδραματίζει και το μέγεθος εισόδου που χρησιμοποιείται από την κάθε εφαρμογή το οποίο κάποιες φορές ίσως να είναι αποδοτικότερο να υπολογιστεί με ένα μόνο kernel-πυρήνα παρά να διασπαστεί του σε πολλούς ξεχωριστούς υπολογισμούς. Επιπλέον, είδαμε εφαρμογές όπως η QSORT και η TRAPEZSIMPLE, οι οποίες περιέχουν ένα κομμάτι στην εκτέλεση τους το οποίο δεν μπορεί να παραλληλοποιηθεί και αναγκαστικά θα εκτελεστεί σειριακά από ένα μόνο πυρήνα. Η QSORT εκτελεί αλγόριθμό *insertion sort* πάνω στους ενδιάμεσους ταξινομημένους πίνακες ούτως ώστε ο τελικός πίνακας να είναι ταξινομημένος, ενώ η εφαρμογή TRAPEZSIMPLE εκτελεί μια *reduction sum* πράξη πάνω στις ενδιάμεσες τιμές που υπολογίζονται από τον κάθε πυρήνα. Αυτό επιφέρει κόστος στο χρόνο εκτέλεσης των εφαρμογών με αποτέλεσμα να είναι πιο αποδοτική η ολόκληρη εκτέλεσή τους σε ένα μόνο πυρήνα. Η εφαρμογή RK4 περιλαμβάνει πράξεις οι οποίες είναι σε μεγάλο βαθμό εξαρτημένες μεταξύ τους. Συγκεκριμένα αποτελείται από 4 βήματα στη σειρά, το καθένα από τα οποία για να είναι σε θέση να εκτελεστεί θα πρέπει να έχει ολοκληρώσει την εκτέλεσή του το προηγούμενο. Αυτή η ιδιαιτερότητα κάνει την εφαρμογή να φαίνεται μια μη-παράλληλη εφαρμογή που μόνο σειριακά μπορεί να εκτελεστεί. Εντούτοις, κατά την υλοποίηση μας προσπαθήσαμε να διασπάσουμε το κάθε βήμα σε επιμέρους ανεξάρτητες πράξεις και να τις εκτελέσουμε σε διαφορετικούς πυρήνες. Με αυτό τον τρόπο διατηρούνται οι εξαρτήσεις μεταξύ των βημάτων αλλά ταυτόχρονα επιτυγχάνεται πιο γρήγορη εκτέλεση του κάθε βήματος. Από τα αποτελέσματα μας μπορούμε να δούμε μέχρι και τρεις φορές καλύτερους χρόνους εκτέλεσης αλλά από κάποιο σημείο και μετά η πρόσθεση περισσότερων πυρήνων δεν ωφελεί αφού δεν υπάρχει περεταίρω βελτίωση.

Γενικά, μπορούμε να πούμε ότι είμαστε ικανοποιημένοι από την υλοποίηση μας αν και μπορούμε να διακρίνουμε πολλά περιθώρια βελτίωσης. Ο στόχος χρήσης της πλατφόρμας TFlux για υλοποίηση DDM μοντέλου εκτέλεσης στον Intel SCC έχει επιτευχθεί και ταυτόχρονα είδαμε κάποιες από τις DDM εφαρμογές να πετυχαίνουν ψηλές επιδόσεις. Στο επόμενο κεφάλαιο θα αναφερθούμε σε πιο αποδοτικές τεχνικές που μπορούμε να εφαρμόσουμε για να βελτιώσουμε την υλοποίηση μας.

Κεφάλαιο 7

Συμπεράσματα

7.1 Συμπεράσματα

Στην εργασία αυτή έγιναν όλες οι διαδικασίες που απαιτούνταν για την εκτέλεση του Data-Driven Multithreading μοντέλου στον επεξεργαστή Intel SCC κάνοντας χρήση της φορητής πλατφόρμας TFlux. Έγιναν οι απαραίτητες τροποποιήσεις στον TFlux C Preprocessor και στη μονάδα συγχρονισμού TSU και αξιολογήσαμε την υλοποίηση μας με την εκτέλεση διαφόρων DDM εφαρμογών που προέρχονται από γνωστά και διαδεδομένα benchmarks. Πλέον, ο προγραμματιστής έχει τη δυνατότητα να αναπτύξει τα DDM προγράμματά του και να τα εκτελέσει, όχι μόνο σε οποιοδήποτε κοινής μνήμης σύστημα, αλλά και στον πειραματικό many-core επεξεργαστή SCC ο οποίος είναι ένα συμπλεγματικό (clustered) σύστημα κατανεμημένης μνήμης.

Κατά την αξιολόγηση της εφαρμογής μας χρησιμοποιήσαμε εφαρμογές διαφορετικών κατηγοριών ώστε να δημιουργήσουμε μια όσο το δυνατό ακριβέστερη εικόνα για την απόδοση της υλοποίησης μας, γι' αυτό και τα αποτελέσματα που πήραμε διαφέρουν από εφαρμογή σε εφαρμογή. Το βασικό συμπέρασμα, το οποίο εξάγουν τα αποτελέσματα από κοινού, είναι το γεγονός ότι η εκτέλεση περισσότερων του ενός thread στους single-threaded πυρήνες του SCC δημιουργεί

τρομερά μεγάλο overhead στους χρόνους εκτέλεσης. Οι πυρήνες του Intel SCC δεν υποστηρίζουν multithreading και αυτό έχει ως αποτέλεσμα η διαδικασία του λειτουργικού συστήματος για το context switching μεταξύ των 2 threads να εκτοξεύει το χρόνο εκτέλεσης σε πολύ ψηλά επίπεδα. Έγινε προσπάθεια να βγάλουμε εκτός πυρήνα το ένα thread (όποτε δεν κάνει κάτι χρήσιμο) έτσι ώστε να εκτελείται μόνο του το application thread χωρίς να το διακόπτει το context switch του λειτουργικού συστήματος και είδαμε τους χρόνους να μειώνονται μέχρι και 6 φορές.

Ένα άλλο βασικό συμπέρασμα που μπορούμε να αναφέρουμε για την υλοποίηση μας είναι το γεγονός ότι η μη ύπαρξη συνέπειας cache μνήμης (cache-coherency) στο σύστημα του SCC δεν μας επηρεάζει καθόλου. Ο λόγος που συμβαίνει αυτό είναι διότι το ίδιο το DDM μοντέλο αναλαμβάνει να καθορίζει και να διατηρεί τις εξαρτήσεις μεταξύ των δεδομένων αντικαθιστώντας έτσι, εν μέρει, τη λειτουργία ενός πρωτοκόλλου συνέπειας μνήμης (cache-coherency protocol). Σύμφωνα με το DDM μοντέλο εκτέλεσης, υπάρχουν καθορισμένοι δεσμοί μεταξύ των δεδομένων και κάθε πυρήνας είναι υπεύθυνος για τα δεδομένα που του καθορίζει ο γράφος συγχρονισμού. Με αυτό τον τρόπο ο καθένας φέρνει στην cache μνήμη του τα δεδομένα που χρειάζεται και τα οποία, όπως μας διαβεβαιώνει το DDM μοντέλο, δεν χρησιμοποιούνται κάπου αλλού τη δεδομένη στιγμή. Ταυτόχρονα όμως, η απουσία cache-coherency αναγκάζει το σύστημα να μην εφαρμόζει μεγάλου βαθμού caching των δεδομένων ούτως ώστε να μην μένουν τα δεδομένα για μεγάλο χρονικό διάστημα στη cache μνήμη και να δημιουργούνται συγκρούσεις (conflicts) μεταξύ των πυρήνων. Γι' αυτό το λόγο οι κατασκευαστές του Intel SCC προτίμησαν να αφιερώσουν τη cache μνήμη κάθε πυρήνα μόνο για την προσωπική (private) μνήμη (L1 & L2 cache) και το MPB (L1 cache), ενώ τα δεδομένα που προέρχονται από την κοινή (shared) μνήμη είναι "uncacheable" και φορτώνονται απευθείας στους καταχωρητές. Αυτή είναι και η αιτία που κατά την παράλληλη εκτέλεση κάποιων εφαρμογών είδαμε τους χρόνους και τις επιδόσεις να είναι πολύ χειρότεροι ακόμη και από την σειριακή εκτέλεση. Το πλήθος και το μέγεθος των καταχωρητών είναι πάρα πολύ μικρό και επομένως κατά τη φόρτωση δεδομένων από τη κοινή μνήμη το σύστημα αναγκάζεται να εκτελεί πολύ περισσότερες off-chip προσβάσεις, γεγονός που είναι χρονοβόρο για την εκτέλεση. Η υλοποίηση μας θα προτιμούσε τα δεδομένα που έρχονται από τη κοινή μνήμη να έμεναν στην cache των πυρήνων (cacheable) αφού η ασυνέπεια cache μνήμης (incoherency) του Intel SCC, όπως αναφέραμε και πιο πάνω, δεν επηρεάζει την υλοποίησή μας. Ένας άλλος

παράγοντας που πρέπει να ληφθεί υπόψη για την υλοποίηση μας είναι το γεγονός ότι όσοι περισσότεροι πυρήνες χρησιμοποιούνται κατά την εκτέλεση μιας εφαρμογής τόσα περισσότερα είναι τα μηνύματα που αποστέλλονται μεταξύ τους με αποτέλεσμα το δίκτυο του SCC να φορτώνεται με μεγαλύτερο overhead. Επίσης, κόστος στο χρόνο εκτέλεσης μπορεί να προκύψει και από τους πυρήνες που βρίσκονται μακριά από τους διαχειριστές μνήμης (memory controllers) και επομένως οι προσβάσεις τους στην off-chip μνήμη να είναι πιο χρονοβόρες από αυτές των πυρήνων που είναι δίπλα στους διαχειριστές.

7.2 Μελλοντική Εργασία

Σε γενικές γραμμές, είμαστε αρκετά ικανοποιημένοι από την υλοποίηση μας αφού ο στόχος για εκτέλεση DDM μοντέλου στον Intel SCC έχει επιτευχθεί, ενώ παράλληλα έχουμε αποκτήσει ένα ισχυρό κίνητρο για να συνεχίσουμε και να βελτιώσουμε την εργασία μας μελλοντικά. Μελετήσαμε τα σημεία που κοστίζουν στην επίδοση και επινοήσαμε κάποιες ιδέες που μπορούμε να εφαρμόσουμε ώστε η υλοποίηση μας να γίνει ακόμα πιο αποδοτική. Το πρόβλημα των single-threaded πυρήνων είναι ένας από τους πιο σημαντικούς παράγοντες που περιορίζουν την επίδοση. Θα προσπαθήσουμε να τρέξουμε αυτούσια την υλοποίηση μας σε ένα κανονικό κατανεμημένο σύστημα το οποίο δεν είναι άλλο από ένα δίκτυο με πολύ-πύρηνους εξυπηρετητές (servers). Κάθε πυρήνας του SCC θα αντικατασταθεί από ένα εξυπηρετητή, ο οποίος πλέον θα υποστηρίζει multithreading, άρα θα μπορεί να εκτελεί πολλαπλά threads ταυτόχρονα. Η μόνη αλλαγή που χρειάζεται να γίνει είναι η αφαίρεση του RCCE API και η αντικατάσταση του με επικοινωνία μέσω διευθύνσεων IP και sockets, κάτι που ίσως να προκαλέσει επιπρόσθετο overhead αφού η TCP/IP επικοινωνία ίσως να είναι πιο χρονοβόρα.

Όποιο και να είναι το αποτέλεσμα, εμείς θα πρέπει, στη δική μας υλοποίηση, να αποτρέψουμε την εκτέλεση περισσότερων του ενός thread στον κάθε πυρήνα του SCC. Προτεινόμενη λύση είναι η αντικατάσταση των TSUs σε κάθε πυρήνα από μια ενιαία δομή που θα έχει την ίδια λειτουργία. Αυτό μπορεί να γίνει είτε με την υλοποίηση του TSU ως μια υπολογιστική οντότητα σε ένα σταθερό και προκαθορισμένο πυρήνα, είτε με την αποθήκευση του γράφου συγχρονισμού σε ολόκληρο το MPB με τα application threads να εκτελούν ενημερώσεις (updates)

όποτε χρειάζεται, κάτι που ουσιαστικά αφαιρεί ολοκληρωτικά την ύπαρξη ξεχωριστή μονάδας TSU.

Μια άλλη ιδέα βελτίωσης της επίδοσης είναι η χρήση προ-ανάκλησης (*prefetching*) για την πρόωρη φόρτωση των δεδομένων στη μνήμη των πυρήνων. Από τη στιγμή που υπάρχει ο γράφος συγχρονισμού, είμαστε σε θέση να γνωρίζουμε ποια δεδομένα θα χρειαστούν ανά πάσα στιγμή της εκτέλεσης. Με αυτό τον τρόπο μπορούμε να αφιερώσουμε ένα πυρήνα στον οποίο να αναθέσουμε το έργο της προ-ανάκλησης των δεδομένων στους υπόλοιπους πυρήνες που συμμετέχουν στην εκτέλεση.

Μια δεύτερη υλοποίηση που θα μπορούσαμε να δοκιμάσουμε μελλοντικά είναι η μεταφορά όλων των δεδομένων που χρησιμοποιούνται σε κάθε εφαρμογή στην προσωπική μνήμη και στο MPB του κάθε πυρήνα. Σκοπός είναι ο αποκλεισμός της χρήσης της κοινής μνήμης και η επίτευξη επικοινωνίας μεταξύ των πυρήνων, όχι μόνο για τα δεδομένα του TSU, αλλά και για τα δεδομένα της υπό εκτέλεση εφαρμογής. Αυτό θα έχει ως αποτέλεσμα την εκπόνηση μιας εντελώς κατανεμημένης (distributed) υλοποίησης, η οποία θα μπορεί να εκτελεστεί σε οποιοδήποτε κατανεμημένο σύστημα που δεν χρησιμοποιεί καθόλου κοινή μνήμη.

Βιβλιογραφία

[1] Kyriacou C., Evripidou P., Trancoso P. "Data-Driven Multithreading Using Conventional Microprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol.17, no.10, pp.1176-1188, Oct. 2006

[2] Stavrou K., Nikolaides M., Pavlou D., Arandi S., Evripidou P., Trancoso P. "TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems," *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, vol., no., pp.25-34, 9-12 Sept. 2008

[3] Stavrou K., Pavlou D., Nikolaides M., Petrides P., Evripidou P., Trancoso P., Popovic Z., Giorgi R. "Programming Abstractions and Toolchain for Dataflow Multithreading Architectures," *Parallel and Distributed Computing, 2009. ISPDC '09. Eighth International Symposium on*, vol., no., pp.107-114, June 30 2009-July 4 2009

[4] Mattson T.G., Van der Wijngaart R.F., Riepen M., Lehnig T., Brett P., Haas W., Kennedy P., Howard J., Vangal S., Borkar N., Ruhl G., Dighe S. "The 48-core SCC Processor: the Programmer's View," *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, vol., no., pp.1-11, 13-19 Nov. 2010

[5] Diavastos A., Petrides P., Falcao G., Trancoso P. "LDPC Decoding on the Intel SCC," *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, vol., no., pp.57-65, 15-17 Feb. 2012

[6] Clauss C., Lankes S., Reble P., Bemmerl T. "Evaluation and improvements of programming models for the Intel SCC many-core processor," *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, vol., no., pp.525-532, 4-8 July 2011

[7] Andreas Prell and Thomas Rauber “Task Parallelism on the SCC,” *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*. Department of Computer Science, University of Bayreuth, Germany, July 5-6 2011.

[8] Jan-Arne Sobania, Peter Troger, and Andreas Polze “Linux Operating System Support for the SCC Platform - An Analysis,” *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany.* Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany, July 5-6 2011.

[9] R. Rotta “On efficient message passing on the Intel SCC” *Proceedings of the 3rd MARC Symposium,* pages 53–58, 2011.

[10] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope “Efficient Memory Copy Operations on the 48-core Intel SCC Processor” *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*. Informatics Institute, University of Amsterdam, Sciencepark 904, 1098 XH Amsterdam, The Netherlands, July 5-6 2011.

[11] Carsten Clauss, Stefan Lankes, Pablo Reble, Thomas Bemmerl “Recent Advances and Future Prospects in iRCCE and SCC-MPICH” *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*, July 2011

[12] Hayder Al-Khalissi, Mladen Berekovic “Performance of RCCE Broadcast Algorithm in SCC” , *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*, July 2011

[13] Florian Thoma, Michael Hubner, Diana Gohringer, Hasan Umitcan Yilmaz, Jurgen Becker “Power and performance optimization through MPI supported dynamic voltage and frequency scaling” , *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*, July 2011

[14] Intel Corporation (2010) *The SCC Platform Overview Revision 0.6*, Intel Labs

[15] Intel Corporation (2010) *The SCC Programmer’s Guide Revision 0.75*, Intel Labs

[16] Mattson T.G., Van der Wijngaart R.F. (2010) *RCCE: a Small Library for Many-Core Communication*, Software Version 1.0-release, May 3, 2010, Document Version 0.7, Intel Labs

[17] UCY CS Dep. (2008) *TFlux C Preprocessor TFluxCpp,* Technical Manual, July 3rd 2008, Univarsity Of Cyprus

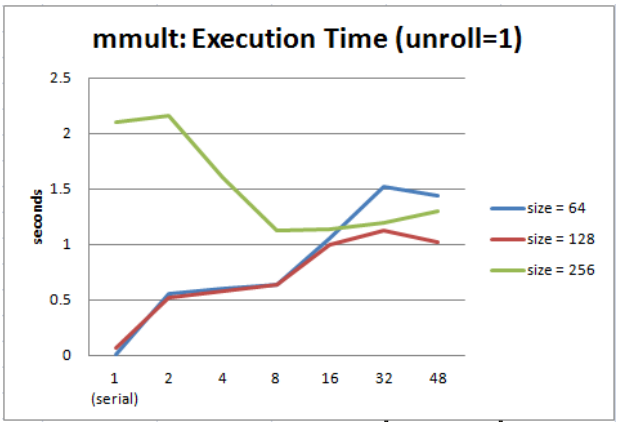
[18] Nikolaides Marios *"Preprocessor και Benchmarks για Πλατφόρμα TFlux-Soft"*, Bachelor Thesis, May 2008, University Of Cyprus CS Department

[19] Arandi, S.; Michael, G.; Evripidou, P.; Kyriacou, C.; , "Combining Compile and Run-Time Dependency Resolution in Data-Driven Multithreading," *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on* , vol., no, pp.45-52, 10-10 Oct. 2011

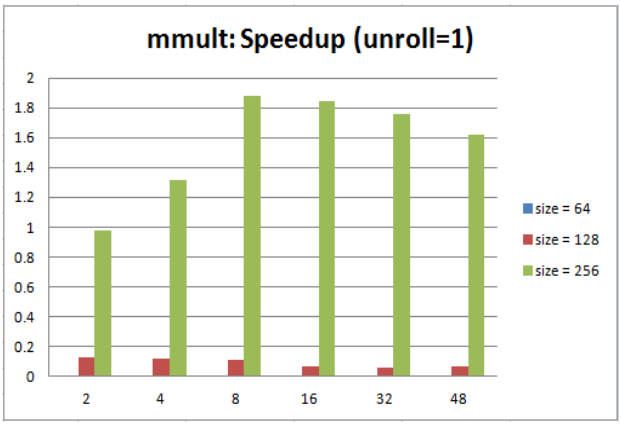
Παράρτημα Α

Το παράρτημα αυτό αναφέρεται στο Κεφάλαιο 6, και συγκεκριμένα στο υποκεφάλαιο 6.3, στο οποίο παρουσιάζονται οι γραφικές παραστάσεις με τα αποτελέσματα που πήραμε από τις εκτελέσεις των εφαρμογών.

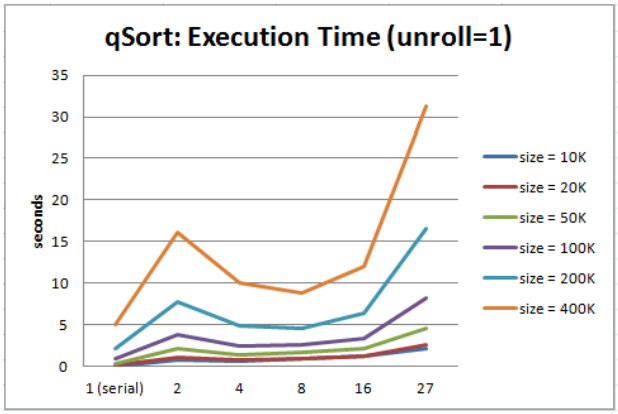
Πιο κάτω, παρατίθενται οι γραφικές παραστάσεις με τους χρόνους και τις επιδόσεις των εφαρμογών κατά την απλή εκτέλεση της υλοποίησης μας χωρίς τη χρήση της συνάρτησης *usleep()* για την ανά διαστήματα αναστολή του TSU thread. Τα αποτελέσματα είναι κανονικοποιημένα σύμφωνα με το χρόνο εκτέλεσης της αντίστοιχης σειριακής εφαρμογής σε ένα πυρήνα του SCC



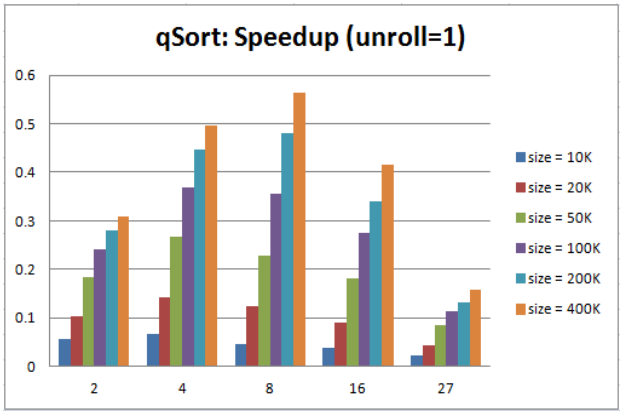
Χρόνοι Εκτέλεσης MMULT



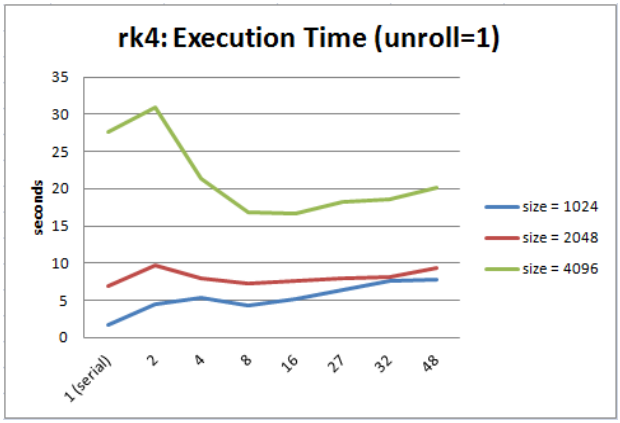
Επίδοση MMULT



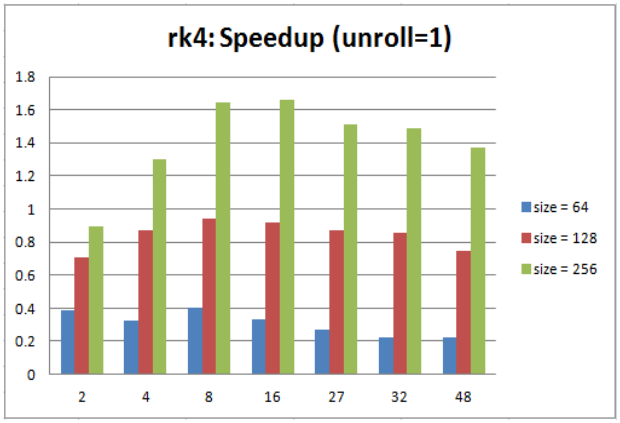
Χρόνοι Εκτέλεσης QSORT



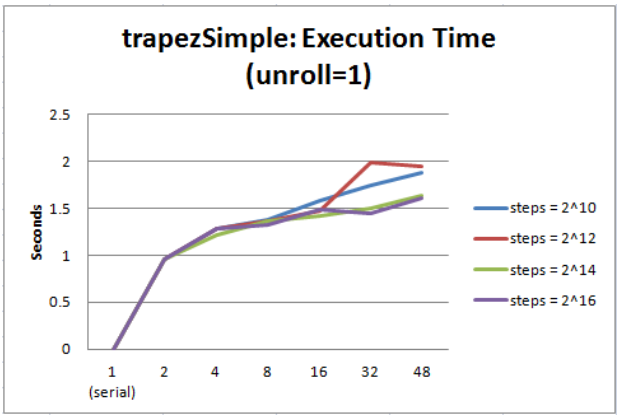
Επίδοση QSORT



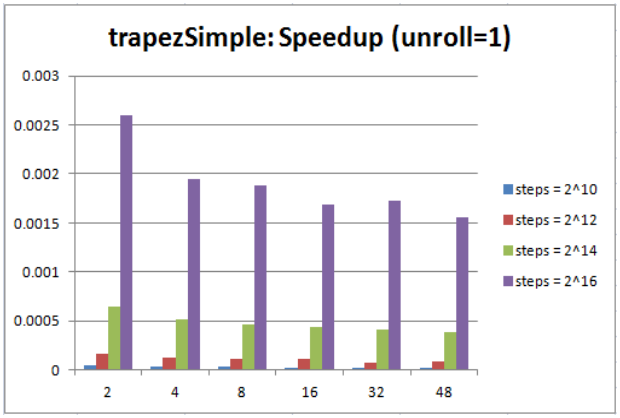
Χρόνοι Εκτέλεσης RK4



Επίδοση RK4

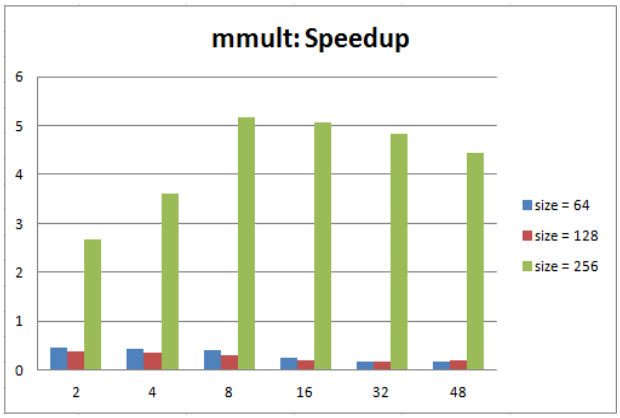


Χρόνοι Εκτέλεσης TRAPEZSIMPLE

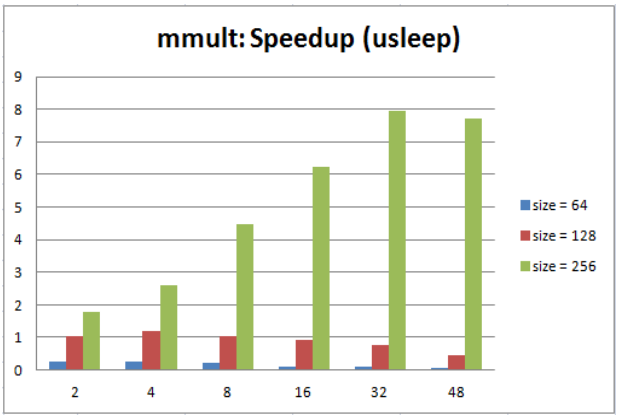


Επίδοση TRAPEZSIMPLE

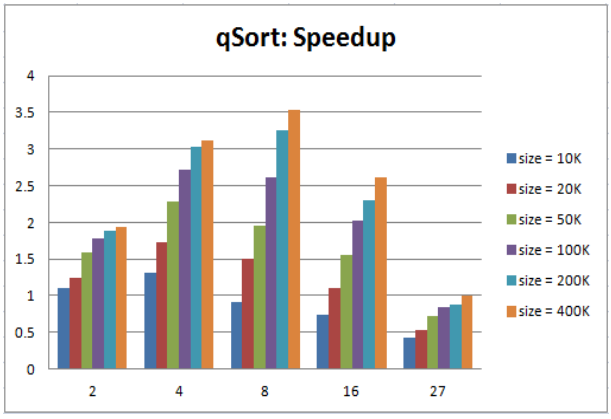
Στη συνέχεια, παρατίθενται οι γραφικές παραστάσεις με τις επιδόσεις και των δύο διαφορετικών εκτελέσεων της υλοποίησης μας (με ή χωρίς usleep), αυτή τη φορά κανονικοποιημένες με την αντίστοιχη εκτέλεση σε ένα πυρήνα του SCC και με χρήση ενός kernel.



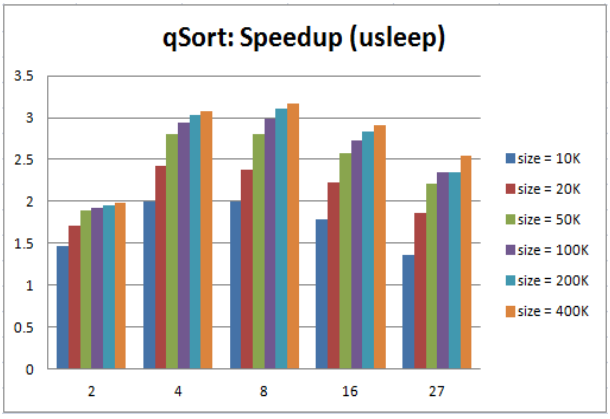
Επίδοση MMULT χωρίς usleep()



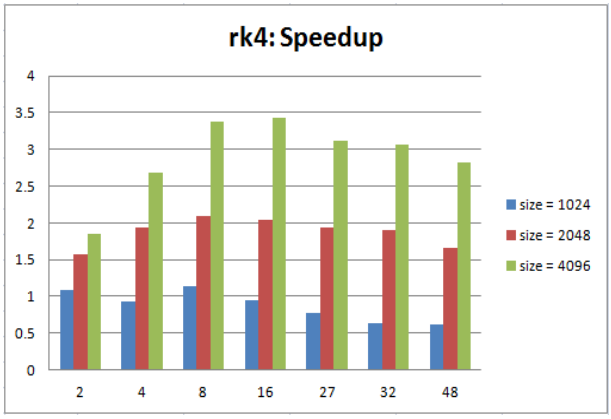
Επίδοση MMULT με usleep()



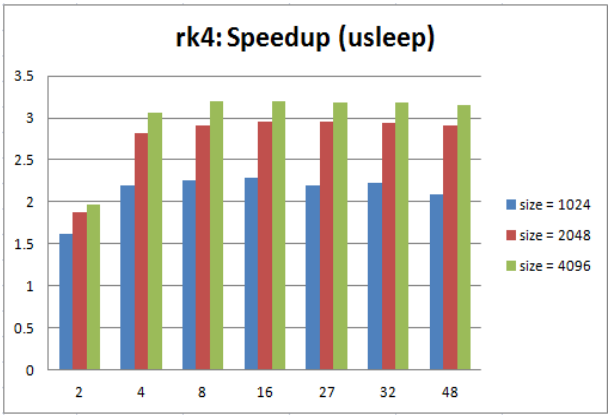
Επίδοση QSORT χωρίς usleep()



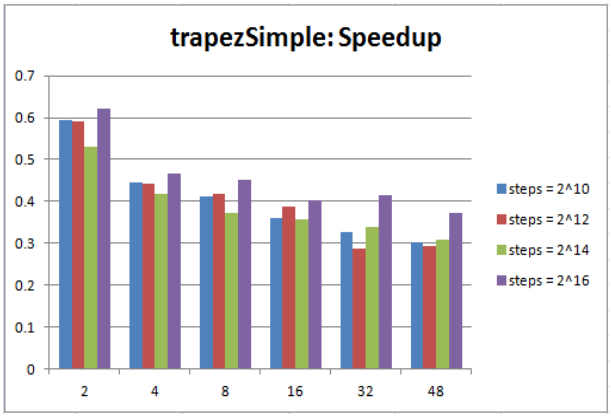
Επίδοση QSORT με usleep()



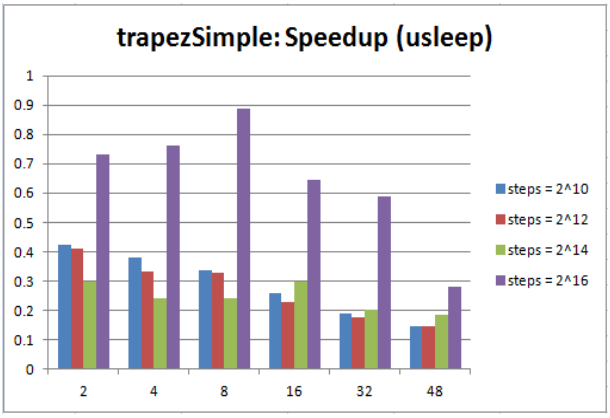
Επίδοση RK4 χωρίς usleep()



Επίδοση RK4 με usleep()



Επίδοση TRAPEZSIMPLE χωρίς usleep()



Επίδοση TRAPEZSIMPLE με usleep()