Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Πληροφορικής και Τεχνολογίας Υπολογιστών

Τεχνικές για την Βελτιστοποίηση και Αποδοτική Απεικόνιση Παράλληλων Κωδίκων σε Υπολογιστικούς Κόμβους με Πολυνηματικές και Πολυπύρηνες Αρχιτεκτονικές Μικροεπεξεργαστών

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

ΝΙΚΟΛΑΟΥ Χ. ΑΝΑΣΤΟΠΟΥΛΟΥ

Αθήνα, Μάρτιος 2010

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Πληροφορικής και Τεχνολογίας Υπολογιστών

Τεχνικές για την Βελτιστοποίηση και Αποδοτική Απεικόνιση Παράλληλων Κωδίκων σε Υπολογιστικούς Κόμβους με Πολυνηματικές και Πολυπύρηνες Αρχιτεκτονικές Μικροεπεξεργαστών

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

ΝΙΚΟΛΑΟΥ Χ. ΑΝΑΣΤΟΠΟΥΛΟΥ

Συμβουλευτική Επιτροπή: Νεκτάριος Κοζύρης
Παναγιώτης Τσανάκας
Γεώργιος Παπακωνσταντίνου

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 26η Μαρτίου 2010.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης Αναπ. Καθηγητής ΕΜΠ	Παναγιώτης Τσανάκας Καθηγητής ΕΜΠ	Γεώργιος Παπακωνσταντίνου Ομότιμος Καθηγητής ΕΜΠ
. . . . .	. . . . .	. . . . .
Ανδρέας Μπουντουβής Καθηγητής ΕΜΠ	Κωνσταντίνος Σαγώνας Αναπ. Καθηγητής ΕΜΠ	Δημήτριος Νικολόπουλος Αναπ. Καθηγητής Πανεπιστημίου Κρήτης
	. . . . .	
	Χρήστος Ζαρολιάγκης Καθηγητής Πανεπιστημίου Πατρών	

Αθήνα, Μάρτιος 2010

. . . . . .

ΝΙΚΟΛΑΟΣ Χ. ΑΝΑΣΤΟΠΟΥΛΟΣ
Διδάκτωρ Εθνικού Μετσόβιου Πολυτεχνείου

Η εκπόνηση της διατριβής χρηματοδοτήθηκε από το έργο ΠΕΝΕΔ 03ΕΔ74 με τίτλο "Ανάπτυξη και βελτιστοποίηση παράλληλων αλγόριθμων σε συστοιχίες υπολογιστών διπλοεπεξεργασίας για την επίλυση προβλημάτων της μηχανικής διεργασιών και συστημάτων". Το έργο υλοποιήθηκε στο πλαίσιο του Μέτρου 8.3 του Ε.Π. "Ανταγωνιστικότητα" του Γ' Κοινοτικού Πλαισίου Στήριξης και συγχρηματοδοτήθηκε κατά 80% της δημόσιας δαπάνης από την Ευρωπαϊκή Ένωση – Ευρωπαϊκό Κοινωνικό Ταμείο και 20% της δημόσιας δαπάνης από το Ελληνικό Δημόσιο – Υπουργείο Ανάπτυξης – Γενική Γραμματεία Έρευνας και Τεχνολογίας.

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

Συντμήσεις

CMP	Chip-level Multiprocessing – Πολυεπεξεργασία σε Επίπεδο Τσιπ
CPI	Clock-cycles Per Instruction – Κύκλοι Ρολογιού ανά Εντολή
DTLB	Data Translation Lookaside Buffer – Απομονωτής Μεταφράσεων για Δεδομένα
HT	Hyper-Threading
HTM	Hardware Transactional Memory – Μνήμη Διενεργειών υλοποιημένη στο Υλικό
ILP	Instruction Level Parallelism – Παραλληλισμός Επιπέδου Εντολών
IPC	Instructions Per Clock-cycle – Εντολές ανά Κύκλο Ρολογιού
IPI	Inter-Processor Interrupt – Δι-Επεξεργαστικό Σήμα Διακοπής
ISA	Instruction Set Architecture – Σύνολο Εντολών Αρχιτεκτονικής
ITLB	Instruction Translation Lookaside Buffer – Απομονωτής Μεταφράσεων για Εντολές
RAT	Register Alias Table – Πίνακας Ψευδωνύμων Καταχωρητών
ROB	Re-Order Buffer – Απομονωτής Αναδιάταξης Εντολών
SMP	Symmetric Multiprocessing – Συμμετρική Πολυεπεξεργασία
SMT	Simultaneous Multithreading – Ταυτόχρονος Πολυνηματισμός
SPR	Speculative Precomputation – Υποθετική Προ-εκτέλεση

SSSP Single-Source Shortest Paths – Συντομότερα Μονοπάτια από μια Αρχική Κορυφή

STM Software Transactional Memory – Μνήμη Διενεργειών υλοποιημένη στο Λογισμικό

TLB Translation Lookaside Buffer – Απομονωτής Μεταφράσεων

TLP Thread Level Parallelism/Parallelization – Παραλληλισμός/Παραλληλοποίηση Επιπέδου Νημάτων

TLS Thread Level Speculation – Υποθετική Εκτέλεση Επιπέδου Νημάτων

TM Transactional Memory – Μνήμη Διενεργειών

Αντί Προλόγου

Η παρούσα διδακτορική διατριβή εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Έχει σαν αντικείμενο τη διερεύνηση και ανάπτυξη τεχνικών λογισμικού με στόχο την αποδοτική εκμετάλλευση των πολλαπλών ροών εκτέλεσης σύγχρονων πολυνηματικών και πολυπύρηνων αρχιτεκτονικών από τις εφαρμογές του χρήστη.

Η έλευση και η ευρεία επικράτηση των αρχιτεκτονικών αυτών τα τελευταία χρόνια κατέστησε επιτακτική μια γενικότερη στροφή προς τη σχεδίαση και την ανάπτυξη παράλληλου λογισμικού. Ο παράλληλος προγραμματισμός στις νέες συνθήκες θέτει μια σειρά από πολύ σημαντικές προκλήσεις, οι οποίες θα πρέπει να αντιμετωπιστούν για όλο το φάσμα των εφαρμογών. Στα πλαίσια της διατριβής προσπάθησα να αντιμετωπίσω κάποιες από τις προκλήσεις αυτές, εστιάζοντας ιδιαίτερα σε εναλλακτικά μοντέλα παραλληλισμού που απευθύνονται σε "δύσκολα" παραλληλοποιήσιμες εφαρμογές. Τέτοιες εφαρμογές δε θα μπορούσαν να λάβουν οφέλη σε παραδοσιακά συστήματα πολυεπεξεργασίας ή χρησιμοποιώντας συμβατικές τεχνικές παραλληλισμού. Τα αποτελέσματα που προέκυψαν κατέδειξαν τις όποιες δυσκολίες εφαρμογής αλλά ταυτόχρονα και τη δυναμική των τεχνικών αυτών, και ως εκ τούτου ελπίζω να αποτελέσουν την απαρχή για μελλοντική έρευνα.

Θα ήθελα να εκφράσω τις ευχαριστίες μου σε όλους όσους βοήθησαν στην ολοκλήρωση της διατριβής. Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Αναπληρωτή Καθηγητή κ. Νεκτάριο Κοζύρη, επιβλέποντα της διατριβής, για την καθοδήγηση καθόλη τη διάρκεια εκπόνησής της. Επιπλέον, θα ήθελα να ευχαριστήσω τους διδάκτορες Γιώργο Γκούμα και Κωστή Νίκα για τη βοήθεια που προσέφεραν όλον αυτόν τον καιρό, τη συνεργασία και τις συζητήσεις που είχαμε στα πλαίσια

των διαφόρων ερευνητικών δραστηριοτήτων. Τέλος, ευχαριστώ τα μέλη της επιτροπής παρακολούθησης για τα χρήσιμα σχόλια και τις επισημάνσεις τους πάνω στη διατριβή καθώς και τις υποδείξεις για μελλοντικές επεκτάσεις της.

Νίκος Αναστόπουλος
Αθήνα, Μάρτιος 2010

Περίληψη

Οι πολυπύρηνες και πολυνηματικές αρχιτεκτονικές κερδίζουν συνεχώς έδαφος τα τελευταία χρόνια αποτελώντας πλέον τον κανόνα στη σχεδίαση των επεξεργαστών σε ένα ευρύ φάσμα εφαρμογών. Για να μπορούν να αξιοποιήσουν τα προγράμματα του χρήστη τις δυνατότητές τους, είναι απαραίτητη μια γενικότερη στροφή προς την εκμετάλλευση του *παραλληλισμού επιπέδου νημάτων* (thread-level parallelism – TLP) που μπορεί να εξαχθεί από αυτά. Σε αυτό το νέο περιβάλλον τίθενται επομένως μια σειρά από σημαντικές προκλήσεις στον προγραμματιστή, όπως ο εντοπισμός, η έκφραση και η απεικόνιση του παραλληλισμού, ο συγχρονισμός μεταξύ των νημάτων και η αποδοτική διαχείριση των πόρων της υποκείμενης αρχιτεκτονικής. Συμβατικές τεχνικές παραλληλοποίησης και συγχρονισμού που έχουν προταθεί στη βιβλιογραφία είναι θεωρητικά εφαρμόσιμες στις νέες αρχιτεκτονικές, όμως είτε καλύπτουν συγκεκριμένα είδη εφαρμογών με προφανή και άμεσα εκμεταλλεύσιμο παραλληλισμό, είτε δε λαμβάνουν υπόψη τις ιδιαιτερότητες κάθε αρχιτεκτονικής στη διαχείριση των πόρων με αποτέλεσμα να οδηγούν σε μειωμένη απόδοση.

Στα πλαίσια αυτής της διατριβής εξετάζουμε τεχνικές που έχουν σαν στόχο τον εντοπισμό και την απεικόνιση του παραλληλισμού καθώς και τον αποδοτικό συγχρονισμό σε αρχιτεκτονικές επεξεργαστών με *Ταυτόχρονο Πολυνηματισμό* (Simultaneous Multithreading – SMT) και *Πολυεπεξεργασία σε Επίπεδο Τσιπ* (Chip-level Multiprocessing – CMP). Διερευνούμε εναλλακτικές τεχνικές παραλληλοποίησης που στηρίζονται στην ιδέα της *βοηθητικής νημάτωσης* (helper threading) και οι οποίες προορίζονται κυρίως για εφαρμογές με ασαφή, ακανόνιστο ή και μηδενικό εγγενή παραλληλισμό. Τέτοιες εφαρμογές δε θα μπορούσαν να λάβουν σημαντικά οφέλη αν εκτελούνταν σε κάποιο παραδοσιακό σύστημα πολυεπεξεργασίας ή χρησιμοποιώντας κάποια παραδοσιακή τεχνική παραλληλοποίησης.

Στις αρχιτεκτονικές SMT χρησιμοποιούμε τη βοηθητική νημάτωση για να αποφορτίσουμε το κύριο νήμα μιας εφαρμογής από χρονοβόρες λειτουργίες πρόσβασης στη μνήμη. Σε αρκετές περιπτώσεις επιτυγχάνουμε αξιοσημείωτα αποτελέσματα, ωστόσο οι συγκρούσεις ανάμεσα στα εκτελούμενα νήματα για κοινούς πόρους του επεξεργαστή καθιστούν δύσκολη την επίτευξη μεγαλύτερων επιταχύνσεων. Στην κατεύθυνση αυτή προτείνουμε ένα πλαίσιο για την υλοποίηση αποδοτικών λειτουργιών συγχρονισμού, οι οποίες σε σύγκριση με άλλες υλοποιήσεις είναι σε θέση να προσφέρουν τον καλύτερο συμβιβασμό ανάμεσα στην αποδοτική διαχείριση πόρων και τη χαμηλή καθυστέρηση.

Στις αρχιτεκτονικές CMP χρησιμοποιούμε τη βοηθητική νημάτωση για να αποφορτίσουμε το κύριο νήμα από πραγματικούς υπολογισμούς, αξιοποιώντας έναν προηγμένο μηχανισμό συγχρονισμού στο υλικό, αυτόν της *μνήμης διενεργειών* (transactional memory – TM). Παρουσιάζουμε ένα σχήμα *υποθετικής παραλληλοποίησης* (speculative parallelization), μέσω του οποίου καταφέρνουμε να επιταχύνουμε μια περίπτωση εφαρμογής για την οποία οποιοδήποτε συμβατικό σχήμα παραλληλοποίησης μέχρι τώρα είχε αρνητικά αποτελέσματα.

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

1.1 Βασικό Κίνητρο

Με την εξέλιξη της τεχνολογίας επεξεργασίας ολοκληρωμένων κυκλωμάτων γίνεται εφικτή η ενσωμάτωση όλο και περισσότερων τρανζίστορ σε ένα ολοκληρωμένο, γεγονός που καθιστά αναγκαία την αναζήτηση μεθόδων για αύξηση της επίδοσης των μικροεπεξεργαστών. Μέχρι και πριν μερικά χρόνια η συνήθης κατεύθυνση όσον αφορά την εκμετάλλευση των επιπλέον τρανζίστορ ήταν η διερεύνηση αρχιτεκτονικών βελτιστοποιήσεων που σαν κοινό στόχο είχαν την αύξηση του μέσου αριθμού εκτελούμενων εντολών ανά κύκλο (Instructions Per Clock-cycle – IPC) σε μια σειριακή ροή εντολών. Τεχνικές όπως η *αρχιτεκτονική σωλήνωσης* (pipelining), η *υπερβαθμωτή εκτέλεση* (superscalar execution), η *εκτέλεση εκτός σειράς* (out-of-order execution) ή η *υποθετική εκτέλεση* (speculative execution) μαζί με την *πρόβλεψη διακλάδωσης* (branch prediction), στόχευαν στη χρησιμοποίηση των τρανζίστορ για τη δυναμική εξαγωγή παραλληλισμού και τη μεγιστοποίηση των εντολών που μπορούν να εκτελεστούν μέσα σε ένα κύκλο ρολογιού.

Με τις ολοένα και αυξανόμενες πυκνότητες ολοκλήρωσης η βελτιστοποίηση των τεχνικών αυτών για την περαιτέρω αύξηση του μέσου IPC είναι –θεωρητικά– εφικτή. Για παράδειγμα, θα ήταν δυνατόν να αφιερωθεί ακόμα μεγαλύτερο μέρος των τρανζίστορ για να αυξηθεί το εύρος της υπερβαθμωτής εκτέλεσης ή να μεγαλώσει το μέγεθος του παραθύρου μέσα στο οποίο ο επεξεργαστής αναζητά ανεξάρτητες εντολές. *Θα ήταν όμως ανταποδοτική μια τέτοια επιλογή;* Η απάντηση σε αυτό το ερώτημα είναι αρνητική για τον εξής κυρίως λόγο: το ποσοστό των

τρανζίστορ που αφιερώνονται για την εξαγωγή περισσότερου παραλληλισμού θα αυξανόταν πολλαπλασιαστικά, καταλαμβάνοντας μεγάλη έκταση στο ολοκληρωμένο, τη στιγμή που οι εφαρμογές από τη φύση τους μπορεί να μην έχουν επαρκή *παραλληλισμό επιπέδου εντολών* (instruction level parallelism – ILP) να διαθέσουν προς εκμετάλλευση [Olukotun 96]. Με απλά λόγια, το κέρδος σε απόδοση θα ήταν αναντίστοιχο της επιπλέον κατασκευαστικής πολυπλοκότητας, αν όχι μηδενικό. Στα παραπάνω, έρχονται να προστεθούν και μια σειρά από πρακτικά εμπόδια που σχετίζονται κυρίως με την κατανάλωση ισχύος. Η διαρκής προσπάθεια βελτιστοποίησης της σειριακής εκτέλεσης, είτε με τη μορφή πολύπλοκων μηχανισμών εξαγωγής όλο και περισσότερου ILP είτε μέσω της αύξησης της συχνότητας λειτουργίας του επεξεργαστή, οδηγεί σε σημαντική αύξηση της κατανάλωσης ισχύος ώστε η υλοποίηση των τεχνικών αυτών να θεωρείται απαγορευτική.

Για τους παραπάνω λόγους οι σχεδιαστές επεξεργαστών τα τελευταία χρόνια έχουν στραφεί στη χρησιμοποίηση των επιπλέον τρανζίστορ για την εκμετάλλευση παραλληλισμού από πολλαπλές, ανεξάρτητες ροές εντολών (ή αλλιώς, *νήματα*). Δύο είναι οι κυρίαρχες κατασκευαστικές προσεγγίσεις προς αυτή την κατεύθυνση: η επέκταση υφιστάμενων υπερβαθμωτών επεξεργαστών με απαιτούμενο υλικό για να υποστηρίζονται πολλαπλές, ταυτόχρονες ροές εκτέλεσης, ή η ενσωμάτωση στο ίδιο ολοκληρωμένο κύκλωμα πολλαπλών, αλλά αρκετά απλούστερων, επεξεργαστικών *πυρήνων* (cores). Οι αρχιτεκτονικές της πρώτης προσέγγισης είναι γνωστές ως *Πολυνηματικές* (Multithreaded) αρχιτεκτονικές, με κυριότερο εκπρόσωπό τους τους επεξεργαστές με *Ταυτόχρονο Πολυνηματισμό* (Simultaneous Multithreading – SMT) [Tullsen 95]. Οι αρχιτεκτονικές της δεύτερης προσέγγισης είναι γνωστές σαν αρχιτεκτονικές *Πολυεπεξεργασίας σε Επίπεδο Τσιπ* (Chip-level Multiprocessing – CMP) [Olukotun 96].

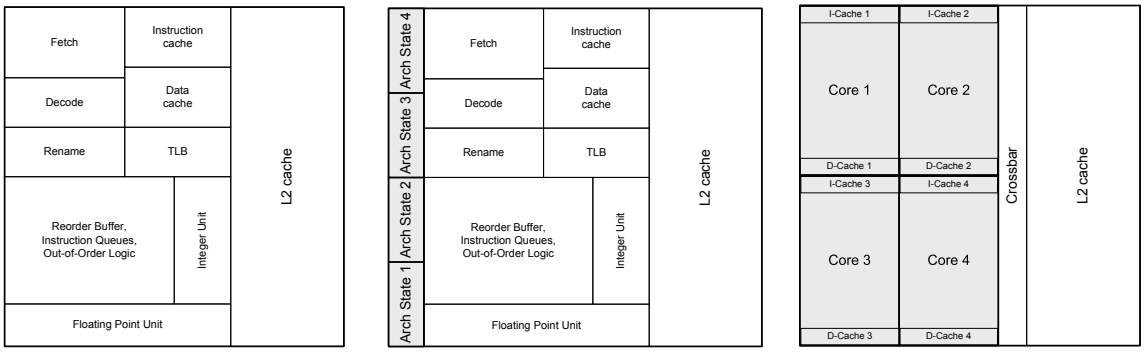
Στις αρχιτεκτονικές SMT ο επεξεργαστής μπορεί να επιλέγει δυναμικά εντολές από πολλαπλά νήματα και να τις δρομολογεί ταυτόχρονα προς εκτέλεση. Όταν ένα νήμα δεν έχει διαθέσιμες εντολές λόγω κάποιου γεγονότος μεγάλης καθυστέρησης (όπως π.χ. αστοχία στην κρυφή μνήμη), τότε ο επεξεργαστής μπορεί να επιλέγει εντολές από τα υπόλοιπα νήματα. Αυτή η τεχνική προωθεί την υψηλή χρησιμοποίηση των επεξεργαστικών πόρων προσφέροντας έναν εναλλακτικό τρόπο να γίνει ανεκτή η αναστολή της εκτέλεσης ενός νήματος λόγω γεγονότων μεγάλης καθυστέρησης. Όταν όμως την ίδια στιγμή υπάρχουν διαθέσιμες εντολές από πολλά νήματα, τότε το συνολικό εύρος του επεξεργαστή στα διάφορα στάδια της σωλήνωσης πρέπει να μοιραστεί ανάμεσά τους, οδηγώντας πολλές φορές σε συγκρούσεις για μονάδες που δεν υπάρχουν σε πολλαπλά αντίγραφα και ακολούθως στην επιβράδυνση των νημάτων. Όταν δεν υπάρχουν πολλαπλά διαθέσιμα νήματα για εκτέλεση, ένας επεξεργαστής SMT μοιάζει απλά με έναν συμβατικό υπερβαθμωτό επεξεργαστή μεγάλου εύρους (wide-issue).

Στις αρχιτεκτονικές CMP χρησιμοποιούνται συνήθως απλούστεροι, μικρότερου εύρους υπερβαθμωτοί επεξεργαστικοί πυρήνες, ανεξάρτητοι μεταξύ τους, ικανοί να εκτελούν ένα μόνο νήμα κάθε φορά. Έτσι, οι αρχιτεκτονικές αυτές μπορούν να εκμεταλλεύονται μέτρια επίπεδα

παραλληλισμού εντολών εντός ενός νήματος και να εκτελούν πολλαπλά νήματα ταυτόχρονα σε διαφορετικούς πυρήνες. Σε σχέση με τους επεξεργαστές SMT οι CMP προσφέρουν καλύτερη "απομόνωση" στα εκτελούμενα νήματα, αφού οι πυρήνες δε μοιράζονται σχεδόν κανένα επεξεργαστικό πόρο μεταξύ τους (με εξαίρεση πιθανώς τα υψηλότερα επίπεδα στην ιεραρχία κρυφής μνήμης). Όμως, όταν ένα νήμα δεν έχει να τροφοδοτήσει τον πυρήνα με εντολές, ο πυρήνας παραμένει αχρησιμοποίητος.

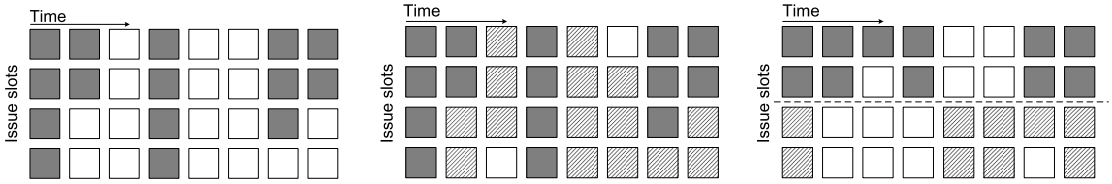
Στο Σχήμα 1.1 φαίνεται μια ενδεικτική οργάνωση του εσωτερικού τριών ολοκληρωμένων κυκλωμάτων, για έναν υπερβαθμωτό, έναν SMT και έναν CMP επεξεργαστή. Οι επεξεργαστές SMT και CMP αποτελούνται ο καθένας από τέσσερα νήματα ταυτόχρονης εκτέλεσης. Το κοινό χαρακτηριστικό και των τριών επεξεργαστών είναι ότι έχουν σχεδόν ίδιο αριθμό από τρανζίστορ, δηλαδή σχεδόν ίδια έκταση υποθέτοντας ότι έχουν κατασκευαστεί με την ίδια τεχνολογία. Διαφέρουν φυσικά ως προς τις δυνατότητές τους. Ο υπερβαθμωτός επεξεργαστής έχει μεγάλο εύρος υπερβαθμωτής εκτέλεσης (στο παράδειγμα του σχήματος λ.χ., η εσωτερική οργάνωση αντιστοιχεί σε έναν υπερβαθμωτό επεξεργαστή εύρους 6 εντολών ανά κύκλο [Olukotun 96]), αλλά βέβαια μπορεί να εκτελεί ένα νήμα κάθε φορά. Ο επεξεργαστής SMT αποτελεί μια άμεση επέκταση του υπερβαθμωτού, όπου απλά έχουν προστεθεί οι κατάλληλες μονάδες για να υποστηρίζεται η διατήρηση της αρχιτεκτονικής κατάστασης τεσσάρων νημάτων. Η επέκταση αυτή συνήθως είναι μικρή σε σχέση με τη συνολική έκταση του ολοκληρωμένου (για παράδειγμα, λιγότερο από 5% στην τεχνολογία Hyper-threading της Intel με δύο νήματα εκτέλεσης). Το σύνολο των μονάδων διαμοιράζεται –στατικά ή δυναμικά– ανάμεσα στα νήματα. Ο επεξεργαστής CMP αποτελείται από τέσσερις ανεξάρτητους πυρήνες εκτέλεσης οι οποίοι έχουν μικρότερες δυνατότητες υπερβαθμωτής εκτέλεσης σε σχέση με τον αρχικό επεξεργαστή (στο συγκεκριμένο παράδειγμα, 2 εντολές ανά κύκλο ο καθένας [Olukotun 96]). Μονάδες του αρχικού επεξεργαστή όπως κρυφές μνήμες πρώτου επιπέδου ή φυσικοί καταχωρητές, έχουν ισοκατανεμηθεί στους πυρήνες. Η κρυφή μνήμη L2 έχει διατηρηθεί όπως ήταν αρχικά και πλέον μοιράζεται από όλους τους πυρήνες.

Στο Σχήμα 1.2 φαίνονται χαρακτηριστικά οι διαφορές ανάμεσα σε έναν υπερβαθμωτό, έναν SMT και έναν CMP επεξεργαστή, όσον αφορά την ικανότητα του καθενός να επεξεργάζεται εντολές από πολλαπλά νήματα ταυτόχρονα. Συγκεκριμένα, απεικονίζεται ο τρόπος που η κάθε αρχιτεκτονική *εκδίδει* (issue) εντολές σε κάθε κύκλο, δηλαδή τις επιλέγει από την ουρά εντολών και τις αποστέλλει στις κατάλληλες μονάδες εκτέλεσης. Οι επεξεργαστές SMT και CMP αποτελούνται ο καθένας από δύο νήματα ταυτόχρονης εκτέλεσης. Τα τέσσερα κουτιά κάθε στήλης εκφράζουν τέσσερις *υποδοχές έκδοσης* (issue slots), δηλαδή τέσσερις πιθανές εντολές που ο επεξεργαστής μπορεί να επιλέξει και να στείλει για εκτέλεση. Ένα χρωματιστό κουτί υποδεικνύει ότι ο επεξεργαστής κατάφερε να βρει στο συγκεκριμένο κύκλο εντολή από κάποιο νήμα για να εκδώσει, ενώ ένα κουτί χωρίς χρώμα υποδεικνύει ότι στον κύκλο αυτό δεν βρέθηκε διαθέσιμη



(α) *Υπερβαθμωτή* **(β)** *SMT* **(γ)** *CMP*

Σχήμα 1.1: *Σύγκριση υπερβαθμωτής, SMT και CMP αρχιτεκτονικής όσον αφορά την εσωτερική οργάνωση του ολοκληρωμένου κυκλώματος.*



(α) *Υπερβαθμωτή* **(β)** *SMT* **(γ)** *CMP*

Σχήμα 1.2: *Σύγκριση υπερβαθμωτής, SMT και CMP αρχιτεκτονικής όσον αφορά την ικανότητα ταυτόχρονης έκδοσης εντολών.*

εντολή. Επιπλέον, κάθε χρώμα αντιστοιχεί σε διαφορετικό νήμα εκτέλεσης. Σε επόμενη ενότητα θα δούμε με μεγαλύτερη λεπτομέρεια πώς συσχετίζονται οι αχρησιμοποίητες υποδοχές με το προφίλ εκτέλεσης μιας εφαρμογής και πώς τις εκμεταλλεύεται η κάθε αρχιτεκτονική.

1.2 Βασικοί Ορισμοί και Συμβάσεις

Οι επεξεργαστές SMT έχουν τη δυνατότητα να εκτελούν ταυτόχρονα πολλαπλές, ανεξάρτητες ροές εντολών. Εκθέτουν στο λειτουργικό σύστημα πολλαπλά, ανεξάρτητα *περιβάλλοντα εκτέλεσης* (execution contexts) δίνοντας την εικόνα ενός συστήματος πολυεπεξεργασίας κοινής μνήμης (π.χ. SMP). Ένα περιβάλλον εκτέλεσης συνίσταται σε όλες εκείνες τις δομές που είναι απαραίτητες για τη διατήρηση της αρχιτεκτονικής κατάστασης μιας διεργασίας. Η *αρχιτεκτονική κατάσταση* περιλαμβάνει τους καταχωρητές γενικού σκοπού, τους καταχωρητές ελέγχου και τους καταχωρητές του προγραμματιζόμενου ελεγκτή διακοπών (advanced programmable

interrupt controller registers – APIC)). Στα πλαίσια της διατριβής χρησιμοποιούμε εναλλακτικά τους όρους *λογικοί επεξεργαστές* (logical processors) και *νήματα υλικού* (hardware threads) για να αναφερθούμε σε αυτά τα περιβάλλοντα εκτέλεσης. Θα χρησιμοποιούμε τον όρο *φυσικός επεξεργαστής* (physical processor) ή *φυσικό πακέτο* (physical package) για να αναφερθούμε στο φυσικό ολοκληρωμένο κύκλωμα το οποίο υλοποιεί στο εσωτερικό του ένα σύνολο από νήματα υλικού. Τα νήματα υλικού εντός του ίδιου φυσικού επεξεργαστή θα αναφέρονται σαν *ομότιμα νήματα* (peer threads).

Διευκρινίζουμε ότι οι επεξεργαστές SMT είναι μία από τις κατηγορίες πολυνηματικών επεξεργαστών. Άλλη γνωστή κατηγορία είναι οι *πολυνηματικοί επεξεργαστές λεπτού κόκκου* (fine-grain multithreaded processors), ενώ λιγότερο δημοφιλείς είναι οι *πολυνηματικοί επεξεργαστές χοντρού κόκκου* (coarse-grain multithreaded processors). Και στις δύο περιπτώσεις το λειτουργικό σύστημα "βλέπει" πολλαπλά περιβάλλοντα εκτέλεσης. Διαφέρουν όμως από το SMT, αλλά και μεταξύ τους, στο ότι ο διαμοιρασμός των πόρων ανάμεσα στα νήματα γίνεται με διαφορετικό χρονισμό. Στα πλαίσια αυτής της διατριβής, και όπου δεν το διευκρινίζουμε, όταν αναφερόμαστε σε *πολυνηματικούς επεξεργαστές* θα εννοούμε τους επεξεργαστές SMT.

Όσον αφορά τους επεξεργαστές CMP, θα χρησιμοποιούμε επιπλέον και τον όρο *πολυπύρηνοι* (multicores) για να αναφερθούμε σε αυτούς. Για τους επιμέρους μικρότερους επεξεργαστές εντός ενός επεξεργαστή CMP θα χρησιμοποιούμε τον όρο *πυρήνες* (cores). Ο πυρήνας ενός επεξεργαστή CMP μπορεί να είναι από μόνος του ένας επεξεργαστής SMT με πολλά ταυτόχρονα νήματα υλικού.

Θα χρησιμοποιούμε τον όρο *λογικός καταχωρητής* (logical register) για να αναφερθούμε σε έναν ονομαστικό καταχωρητή που προδιαγράφεται από το σύνολο εντολών αρχιτεκτονικής (instruction set architecture – ISA) του επεξεργαστή. Για τη δομή στο υλικό που αποθηκεύει τα περιεχόμενα ενός καταχωρητή θα χρησιμοποιούμε τον όρο *φυσικός καταχωρητής* (physical register). Όπως θα εξηγήσουμε, ένας λογικός καταχωρητής μπορεί να απεικονίζεται σε περισσότερους από έναν φυσικούς καταχωρητές. Για αυτό το λόγο το σύνολο των φυσικών καταχωρητών στην πράξη είναι αρκετά μεγαλύτερο από το σύνολο λογικών καταχωρητών.

Παραδοσιακά, η έννοια της παραλληλοποίησης έχει συνδεθεί με τη διαδικασία καταμερισμού του αρχικού υπολογιστικού φορτίου μιας εφαρμογής σε παράλληλα νήματα εκτέλεσης, τα οποία συνήθως χαρακτηρίζονται από κάποιο είδος *συμμετρίας* (π.χ. ως προς τις λειτουργίες που επιτελούν, ως προς το φορτίο εργασίας τους, κ.λπ.). Στα πλαίσια της διατριβής θα χρησιμοποιούμε τον όρο *παραλληλοποίηση* (parallelization) για να περιγράψουμε τη διαδικασία μετατροπής μιας μονονηματικής εφαρμογής σε πολυνηματική, χωρίς να θέτουμε αυστηρούς περιορισμούς στο τι μπορεί να εκτελεί κάθε παράλληλο νήμα. Αυτή η διαδικασία μερικές φορές αναφέρεται και ως *νημάτωση* (threading) της εφαρμογής. Με αυτό τον τρόπο μπορούμε να συμπεριλάβουμε μη-συμβατικές μεθόδους παραλληλοποίησης όπως η *βοηθητική νημάτωση*, όπου, όπως θα δούμε,

τα νήματα μπορεί να μην αναλαμβάνουν κάποιο μέρος της δουλειάς της σειριακής εφαρμογής αλλά επιχειρούν μέσω παρενεργειών να την κάνουν να εκτελεστεί γρηγορότερα.

1.3 Ορισμός του Προβλήματος

Οι πολυνηματικές και πολυπύρηνες αρχιτεκτονικές κερδίζουν συνεχώς έδαφος τα τελευταία χρόνια αποτελώντας πλέον τον κανόνα για τους σχεδιαστές επεξεργαστών σε πολλά πεδία εφαρμογών. Αυτή η τάση αναμένεται όχι απλώς να διατηρηθεί, αλλά και να κλιμακωθεί, καθώς οι επεξεργαστές προβλέπεται να ενσωματώνουν εκατοντάδες ή και χιλιάδες από ταυτόχρονες ροές εκτέλεσης σε ένα και μόνο ολοκληρωμένο. Με δεδομένο ότι πλέον δεν μπορεί κανείς να περιμένει από την τεχνολογία επεξεργαστών να συνεισφέρει πολλά παραπάνω στην απόδοση μονονηματικών εφαρμογών, γίνεται επιτακτική η στροφή προς την εκμετάλλευση του *παραλληλισμού επιπέδου νημάτων* (thread level parallelism – TLP) μιας εφαρμογής για τη βελτίωση της απόδοσής της. Με άλλα λόγια, κρίνεται αναγκαία η διερεύνηση μεθοδολογιών και τεχνικών για την εξαγωγή παραλληλισμού από μία εφαρμογή, και την αποδοτική απεικόνισή του σε μια αρχιτεκτονική πολλαπλών νημάτων υλικού ώστε η παράλληλη εκτέλεσή της να δώσει καλύτερους χρόνους σε σχέση με τη σειριακή.

Στο νέο αυτό περιβάλλον, οι συμβατικές τεχνικές παραλληλοποίησης που έχουν προταθεί στη βιβλιογραφία για παραδοσιακά πολυεπεξεργαστικά συστήματα μπορούν να καλύψουν μόνο μια κατηγορία εφαρμογών, αυτές δηλαδή που διαθέτουν προφανή και άμεσα εκμεταλλεύσιμο παραλληλισμό. Αφήνουν εκτός όμως πολλές άλλες εφαρμογές με ασαφή, ακανόνιστο ή και μηδενικό εγγενή παραλληλισμό. Η μεγάλη πρόκληση επομένως είναι ο τρόπος με τον οποίον το λογισμικό θα μπορέσει να βοηθήσει ακόμα και αυτές τις εφαρμογές να αξιοποιήσουν την ισχύ των σύγχρονων αρχιτεκτονικών στο μεγαλύτερο βαθμό.

1.4 Συμβολή της Διατριβής

Αυτή η διατριβή διερευνά και αξιολογεί μεθόδους λογισμικού που σαν στόχο έχουν την αποδοτική εκμετάλλευση του παραλληλισμού επιπέδου νημάτων που μπορεί να εξαχθεί από τις εφαρμογές, στα πλαίσια εκτέλεσής τους σε πολυνηματικές και πολυπύρηνες αρχιτεκτονικές. Σκοπός μας είναι η αξιολόγηση κλασικών μεθόδων παραλληλοποίησης και καταμερισμού φορτίου που χρησιμοποιούνταν παραδοσιακά στα συστήματα πολυεπεξεργασίας κοινής μνήμης, αλλά και η διερεύνηση μη-συμβατικών μεθόδων παραλληλοποίησης που καθιστούν εφικτές τα νέα χαρακτηριστικά λειτουργίας των υπό εξέταση αρχιτεκτονικών. Από τις μη-συμβατικές μεθόδους αυτές είναι δυνατόν να επωφεληθούν εφαρμογές που είναι δύσκολα παραλληλοποιήσιμες (με την κλασική έννοια του ισοκαταμερισμού του φορτίου) και που δε θα λάμβαναν σημαντικά οφέλη αν εκτελούνταν σε κάποιο παραδοσιακό σύστημα πολυεπεξεργασίας.

Όσον αφορά τις πολυνηματικές αρχιτεκτονικές, εξετάζουμε επεξεργαστές με την τεχνολογία Hyper-threading της Intel, που είναι μια χαμηλού κόστους υλοποίηση της τεχνικής του SMT με χρήση δύο νημάτων [Marr 02]. Η συμβολή μας σε αυτήν την περίπτωση έγκειται στα εξής:

- Αξιολογούμε την ικανότητα του επεξεργαστή να επικαλύπτει αποδοτικά ταυτόχρονες ροές εντολών, με διαφορετικές απαιτήσεις όσον αφορά το είδος και το πλήθος μονάδων εκτέλεσης.

- Αξιολογούμε τη δυνατότητα του επεξεργαστή να βελτιώσει την απόδοση εφαρμογών παραλληλοποιημένων με βάση συμβατικές μεθόδους καταμερισμού φορτίου. Χρησιμοποιούμε αποτελέσματα από μετρητές απόδοσης του επεξεργαστή ως ενδείξεις για τις πιθανές αιτίες των παρατηρούμενων διακυμάνσεων της απόδοσης.

- Προτείνουμε ένα σχήμα μη-συμβατικής παραλληλοποίησης που βασίζεται στη χρήση βοηθητικών νημάτων για προφόρτωση δεδομένων στην κρυφή μνήμη. Από όσο γνωρίζουμε, οι περισσότερες εργασίες που επέδειξαν θετικά αποτελέσματα χρησιμοποιώντας παρόμοια σχήματα σε SMT ήταν βασισμένες στη χρήση προσομοιωτή.

- Προτείνουμε ένα πλαίσιο για την υλοποίηση αποδοτικών λειτουργιών συγχρονισμού βασισμένων σε ειδικές εντολές του επεξεργαστή και προσανατολισμένων για νήματα με ασύμμετρο φορτίο εργασίας. Οι προτεινόμενες λειτουργίες συγχρονισμού παρέχουν την καλύτερη απόδοση συγκρινόμενες με άλλους υπάρχοντες μηχανισμούς συγχρονισμού.

Όσον αφορά τις πολυπύρηνες αρχιτεκτονικές, χρησιμοποιούμε ένα μοντέλο επεξεργαστή CMP που υλοποιεί στο υλικό την τεχνική της *μνήμης διενεργειών* (transactional memory – TM) [Herlihy 93]. Η συμβολή μας σε αυτήν την περίπτωση είναι η εξής:

- Προτείνουμε ένα νέο σχήμα υποθετικής παραλληλοποίησης που βασίζεται στη χρήση βοηθητικών νημάτων και μνήμης διενεργειών, και δείχνουμε πώς μπορούμε να επιταχύνουμε μια περίπτωση εφαρμογής για την οποία οποιοδήποτε συμβατικό σχήμα παραλληλοποίησης μέχρι τώρα είχε αρνητικά αποτελέσματα.

1.5 Οργάνωση της Διατριβής

Το υπόλοιπο της διατριβής είναι οργανωμένο ως εξής:

Στο Κεφάλαιο 2 παρουσιάζεται μια επισκόπηση των επεξεργαστών SMT, καθώς και της τεχνολογίας Hyper-threading της Intel. Επιπλέον, γίνεται μια αξιολόγηση της ικανότητας του Hyper-threading να επικαλύπτει διαφορετικές συνθετικές ροές από διαφορετικούς τύπους εντολών.

Στο Κεφάλαιο 3 γίνεται περιγραφή των σχημάτων παραλληλοποίησης που χρησιμοποιούμε για την νημάτωση εφαρμογών, με έμφαση στο σχήμα βοηθητικού πολυνηματισμού για προφόρτωση δεδομένων. Αξιολογείται η ικανότητα ενός Hyper-threaded επεξεργαστή να επιταχύνει την εκτέλεση μιας σειράς πραγματικών εφαρμογών παραλληλοποιημένων με βάση τα προηγούμενα σχήματα, και αναλύονται τα αποτελέσματα.

Στο Κεφάλαιο 4 παρουσιάζεται ένα πλαίσιο υλοποίησης αποδοτικών λειτουργιών συγχρονισμού για Hyper-threaded επεξεργαστές, βασισμένων σε ειδικές, προνομιούχες εντολές του επεξεργαστή. Αξιολογούμε την επίδοσή τους σε σχέση με άλλους υπάρχοντες μηχανισμούς χρησιμοποιώντας μια σειρά από τεχνητά πειράματα αλλά και πραγματικές εφαρμογές.

Στο Κεφάλαιο 5 παρουσιάζουμε ένα σχήμα υποθετικής παραλληλοποίησης που βασίζεται στη συνδυασμένη χρήση βοηθητικής νημάτωσης και μνήμης διενεργειών. Χρησιμοποιούμε μια εγγενώς σειριακή, δύσκολα παραλληλοποιήσιμη εφαρμογή, τον αλγόριθμο εύρεσης συντομότερων διαδρομών του Dijkstra [Dijkstra 59], και δείχνουμε πώς μπορεί να επιταχυνθεί σε έναν πολυπύρηνο επεξεργαστή εφαρμόζοντας το σχήμα αυτό.

Τέλος, στο Κεφάλαιο 6 συνοψίζουμε τα συμπεράσματα της διατριβής και αναδεικνύουμε περιοχές και κατευθύνσεις για μελλοντική έρευνα.

Σχετικές με τα επιμέρους ζητήματα εργασίες παρουσιάζονται στο εκάστοτε κεφάλαιο.

ΚΕΦΑΛΑΙΟ 2

Η Τεχνική του Ταυτόχρονου Πολυνηματισμού

2.1 Εισαγωγή

Οι σύγχρονοι επεξεργαστές ενσωματώνουν χαρακτηριστικά που έχουν σαν σκοπό την μείωση των καθυστερήσεων (stalls) και την αύξηση του αριθμού των εντολών που μπορούν να εκτελεστούν σε έναν κύκλο. Ένα από αυτά τα χαρακτηριστικά είναι η *εκτέλεση εκτός σειράς* (out-of-order execution) ή αλλιώς *δυναμική δρομολόγηση εντολών* (dynamic scheduling). Η τεχνική αυτή δίνει τη δυνατότητα στον επεξεργαστή να δρομολογεί μια εντολή για εκτέλεση μόλις γίνουν διαθέσιμοι όλοι οι τελεστέοι εισόδου της, ακόμα και αν μια προηγούμενη εντολή στο πρόγραμμα δεν έχει ολοκληρωθεί. Μια εντολή μπορεί να μην ολοκληρώνεται έγκαιρα ή να προκαλεί καθυστερήσεις στις εξής περιπτώσεις: είτε επειδή τα δεδομένα που ζητάει δε βρίσκονται τη δεδομένη στιγμή στην κρυφή μνήμη (επομένως καθυστερεί όλες τις επόμενες εντολές που χρησιμοποιούν το αποτέλεσμά της), είτε επειδή εκτελεί μια λειτουργία μεγάλης διάρκειας, όπως για παράδειγμα διαίρεση (με αποτέλεσμα να καθυστερεί επόμενες εντολές που χρησιμοποιούν το αποτέλεσμά της ή άλλες εντολές που θέλουν να χρησιμοποιήσουν την ίδια δομική μονάδα), είτε επειδή εκτελεί μια εντολή διακλάδωσης υπό συνθήκη (με αποτέλεσμα να καθυστερεί τις εντολές της "σωστής" κατεύθυνσης μέχρι να γίνει γνωστό το αποτέλεσμα της διακλάδωσης). Με τη δυναμική δρομολόγηση δίνεται η δυνατότητα στον επεξεργαστή να προτρέχει των εντολών μεγάλης καθυστέρησης και να εκτελεί μεταγενέστερες μη-εξαρτώμενες εντολές. Ταυτόχρονα, μηχανισμοί *πρόβλεψης διακλάδωσης* (branch prediction) χρησιμοποιούνται για να προβλέπεται η σωστή κατεύθυνση μιας εντολής διακλάδωσης υπό συνθήκη, και έτσι ο επεξεργαστής να

μπορεί να εκτελεί *υποθετικά* (speculatively) εντολές στην προβλεφθείσα κατεύθυνση εκμεταλλευόμενος το χρονικό διάστημα μέχρι να γίνει γνωστό το αποτέλεσμα της διακλάδωσης. Παρόλο που οι εντολές μπορούν να εκτελούνται εκτός σειράς, η ανανέωση της αρχιτεκτονικής κατάστασης του επεξεργαστή πρέπει να γίνεται στη σειρά με την οποία εμφανίζονται στο πρόγραμμα. Αυτό συμβαίνει με μηχανισμούς *ολοκλήρωσης εντολών* (instruction commit) όπως ο *απομονωτής αναδιάταξης εντολών* (re-order buffer – ROB). Οι μηχανισμοί αυτοί επιτρέπουν επίσης την αναίρεση των εντολών που εκτελέστηκαν υποθετικά στην περίπτωση που μια πρόβλεψη διακλάδωσης αποδειχτεί τελικά λανθασμένη.

Όλες οι παραπάνω τεχνικές στοχεύουν στη μείωση μέρους των καθυστερήσεων που συναντώνται στην εκτέλεση των προγραμμάτων. Θέτοντάς το διαφορετικά, στοχεύουν στη μείωση του μέσου αριθμού *κύκλων ρολογιού ανά εντολή* (clock-cycles per instruction – CPI) σε μια τιμή όσο πιο κοντά στη μονάδα, που θεωρείται ιδανική. Η *υπερβαθμωτή εκτέλεση* (superscalar execution) προχωρά ένα βήμα παραπέρα, στοχεύοντας στην περαιτέρω μείωση του μέσου CPI κάτω της μονάδας. Συγκεκριμένα, δίνει τη δυνατότητα στον επεξεργαστή να φορτώνει, να αποκωδικοποιεί, να εκδίδει, να εκτελεί και να ολοκληρώνει παραπάνω από μία εντολές μέσα σε έναν κύκλο. Για τον σκοπό αυτό, οι υπερβαθμωτοί επεξεργαστές διαθέτουν πολλαπλές μονάδες εκτέλεσης, καθώς και μεγάλα *παράθυρα εντολών* (instruction windows) μέσα στα οποία μπορούν να αναζητούν ανεξάρτητες εντολές για εκτέλεση. Καθώς η αναζήτηση μπορεί να προχωρά σε μεγάλο βάθος στην ουρά των αποκωδικοποιημένων εντολών, είναι πιθανή η επιλογή ανεξάρτητων εντολών που χρησιμοποιούν τον ίδιο καταχωρητή [1]. Για να επιλυθεί αυτή η αμφισημία και να είναι δυνατή η παράλληλη εκτέλεση των εντολών αυτών, ο κοινός καταχωρητής *μετονομάζεται* (rename) διαφορετικά για κάθε εντολή. Στην πραγματικότητα, χρησιμοποιείται κάποιος μηχανισμός όπως ο *πίνακας ψευδωνύμων καταχωρητών* (register alias table – RAT) για να απεικονιστεί ο κοινός (λογικός) καταχωρητής σε έναν μοναδικό φυσικό καταχωρητή στο υλικό.

2.2 Αρχιτεκτονικές Επεκτάσεις για Υποστήριξη του SMT

Προκειμένου να είναι εφικτή η εκτέλεση εντολών από πολλά νήματα ταυτόχρονα σε μια δυναμική, υπερβαθμωτή αρχιτεκτονική, οι επεκτάσεις που πρέπει να γίνουν είναι άμεσες και σχετικά απλές.

Καταρχάς, πρέπει να φορτώνονται εντολές από πολλά νήματα, επομένως χρειάζονται ξεχωριστοί *μετρητές προγράμματος* (program counters), ένας για κάθε νήμα. Επιπλέον, ο επεξεργαστής χρειάζεται να διατηρεί την αρχιτεκτονική κατάσταση κάθε νήματος, και αυτό μπορεί να το

[1] Εδώ η ανεξαρτησία ορίζεται ως προς τη στενή έννοια των *πραγματικών εξαρτήσεων δεδομένων* (true-data ή read-after-write dependences), δηλαδή όταν μια εντολή χρησιμοποιεί το αποτέλεσμα που παρήγαγε μια προηγούμενη.

κάνει με πολλαπλούς πίνακες ψευδωνύμων καταχωρητών. Με αυτόν τον τρόπο, επίσης, επιτυγχάνεται οι λογικοί καταχωρητές των νημάτων να είναι ανά πάσα στιγμή απεικονισμένοι σε ξένα μεταξύ τους σύνολα φυσικών καταχωρητών.

Αυτό δίνει τη δυνατότητα στον επεξεργαστή να διαχειριστεί στα επόμενα στάδια τις εντολές και τις μεταξύ τους εξαρτήσεις *χωρίς να χρειάζεται να γνωρίζει σε ποιο νήμα ανήκουν*. Οι εντολές δηλαδή από όλα τα νήματα, από τη στιγμή που μετονομαστούν, μπορούν να τοποθετηθούν σε μια κοινή ουρά από την οποία θα εκδίδονται όσες είναι έτοιμες προς εκτέλεση, με τον ίδιο ακριβώς τρόπο που θα γινόταν αν η αρχιτεκτονική υποστήριζε μόνο ένα νήμα εκτέλεσης. Οι μονάδες εκτέλεσης μοιράζονται δυναμικά ("κατ' απαίτηση"), με μεταβαλλόμενο αριθμό εντολών από κάθε νήμα να εκδίδονται σε κάθε κύκλο. Από τη στιγμή που το ποσοστό των μη-εξαρτώμενων εντολών είναι μεγαλύτερο, αφού προέρχονται από ανεξάρτητα νήματα, ο επεξεργαστής έχει τη δυνατότητα να εκμεταλλευτεί υψηλότερα επίπεδα παραλληλισμού.

Τέλος, προκειμένου η ολοκλήρωση των εντολών και η αναίρεση των λανθασμένων υποθετικών εκτελέσεων να μπορεί να γίνεται ανεξάρτητα για κάθε νήμα, πρέπει να παρέχεται ξεχωριστή λειτουργικότητα αναδιάταξης. Αυτό μπορεί να γίνει είτε χρησιμοποιώντας ξεχωριστό ROB για κάθε νήμα, είτε χρησιμοποιώντας αναγνωριστικά νημάτων (thread identifiers ή tags) στις εγγραφές του υπάρχοντος ROB. Αντίστοιχη διάκριση πρέπει να γίνεται και στις δομές που σχετίζονται με εντολές διακλάδωσης (π.χ. branch target buffer, return address stack), ώστε να μην συγχέονται εντολές διαφορετικών νημάτων μεταξύ τους – τη "σύγχυση" για τέτοιου είδους εντολές δεν είναι δυνατόν να τη λύσει η μετονομασία καταχωρητών.

Αξίζει να σημειώσουμε ότι τα στάδια στα οποία γίνεται *πραγματικά ταυτόχρονη* επεξεργασία εντολών από διαφορετικά νήματα, είναι αυτά της έκδοσης και της εκτέλεσης εντολών. Σε αυτά τα στάδια, όπως αναφέραμε, ο επεξεργαστής δεν έχει γνώση των νημάτων στα οποία ανήκουν οι εντολές που επεξεργάζεται εντός ενός κύκλου. Στα υπόλοιπα στάδια, δηλαδή στη φόρτωση, αποκωδικοποίηση μετονομασία, και ολοκλήρωση εντολών, το εύρος του επεξεργαστή *συνήθως δεν μοιράζεται* ανάμεσα σε πολλαπλά νήματα κατά τη διάρκεια ενός κύκλου, αλλά *πολυπλέκεται*, δηλαδή η πρόσβαση εναλλάσσεται ανάμεσα στα νήματα σε διαδοχικούς κύκλους με βάση κάποια πολιτική. Αυτός είναι και ο τρόπος που οι πραγματικές υλοποιήσεις επεξεργαστών SMT εκχωρούν στα νηματά τους το συνολικό εύρος του επεξεργαστή στα διάφορα στάδια της σωλήνωσης.

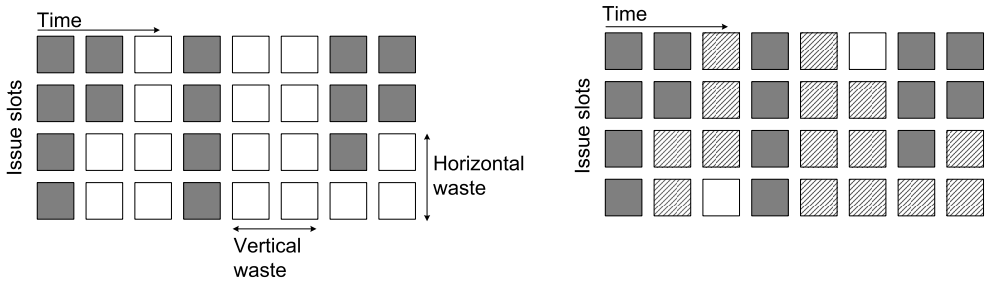
Το πρωταρχικό θεωρητικό μοντέλο SMT που προτάθηκε από τους Eggers και άλλους [Eggers 97] είχε όλα τα παραπάνω χαρακτηριστικά με κάποιες προσθήκες ή τροποποιήσεις. Ο επεξεργαστής υποστήριζε 8 νήματα ταυτόχρονης εκτέλεσης. Σε σχέση με τον αρχικό υπερβαθμωτό επεξεργαστή από τον οποίον προερχόταν (ένα μοντέλο παρόμοιο με τον MIPS R10000 [Yeager 96]), είχε μεγαλύτερο εύρος έκδοσης ώστε να δίνεται η δυνατότητα εκμετάλλευσης περισσότερου παραλληλισμού εντός ενός νήματος. Επιπλέον, λόγω του μεγάλου αριθμού νημάτων το αρχείο

καταχωρητών έπρεπε να μεγαλώσει κατά πολύ. Συνολικά έπρεπε να υποστηρίζονται τόσοι φυσικοί καταχωρητές όσοι το σύνολο των λογικών καταχωρητών όλων των νημάτων, συν κάποιων επιπλέον φυσικών καταχωρητών για μετονομασία. Αυτή η αύξηση είχε σαν αποτέλεσμα τη μεγαλύτερη καθυστέρηση πρόσβασης, που με τη σειρά της οδήγησε στην αύξηση των σταδίων της σωλήνωσης που απαιτούνταν για πρόσβαση στους καταχωρητές.

Η μονάδα φόρτωσης ήταν προσαυξημένη με τις ακόλουθες λειτουργίες: πρώτον, μπορούσε να φορτώσει εντολές στον ίδιο κύκλο ταυτόχρονα από δύο νήματα. Μοιράζοντας το εύρος φόρτωσης σε δύο νήματα, μπορούσε να μειώσει για κάθε νήμα το ποσοστό των εντολών που φορτώνονταν υποθετικά (λόγω πρόβλεψης διακλάδωσης), άρα να μειώσει και την πιθανότητα λανθασμένης εκτέλεσής τους. Αυτό είχε σαν συνέπεια τη βελτίωση της απόδοσης σε σχέση με την περίπτωση όπου σε κάθε κύκλο φορτώνονταν εντολές από ένα μόνο νήμα. Δεύτερον, η επιλογή των δύο νημάτων από τα οποία θα φορτώνονταν εντολές γινόταν βάση πολιτικής. Συγκεκριμένα, επιλέγονταν τα νήματα με τις *λιγότερες* εντολές στα στάδια της αποκωδικοποίησης, μετονομασίας καθώς και στις ουρές. Αυτή η επιλογή ευνοούσε τη συνολική απόδοση του επεξεργαστή για μια σειρά από λόγους που δε σκοπεύουμε να αναλύσουμε στα πλαίσια αυτής της διατριβής.

2.3 Χαρακτηριστικά Λειτουργίας του SMT

Παρά τις τεχνικές που αναφέραμε στην ενότητα 2.1, το μέσο CPI για πολλά προγράμματα που εκτελούνται σε μια σύγχρονη δυναμική, υπερβαθμωτή αρχιτεκτονική απέχει αρκετά από το ιδανικό. Αυτό συμβαίνει διότι οι υποδοχές έκδοσης του επεξεργαστή μένουν ανεκμετάλλευτες. Υπάρχουν δύο είδη "σπατάλης" των υποδοχών έκδοσης, όπως φαίνεται στο Σχήμα 2.1α: το πρώτο είναι η *οριζόντια σπατάλη* (horizontal waste), όπου κατά τη διάρκεια ενός κύκλου μερικές –αλλά όχι όλες– υποδοχές έκδοσης μένουν αχρησιμοποίητες. Αυτό οφείλεται κατά κύριο λόγο σε χαμηλό ILP του προγράμματος. Το δεύτερο είδος είναι η *κατακόρυφη σπατάλη* (vertical waste), στην οποία όλες οι υποδοχές έκδοσης κατά τη διάρκεια ενός ή περισσότερων διαδοχικών κύκλων παραμένουν αχρησιμοποίητες. Αυτό συνήθως οφείλεται σε γεγονότα μεγάλης καθυστέρησης, όπως αστοχίες στην κρυφή μνήμη ή λάθος προβλέψεις διακλάδωσης, που έχουν σαν αποτέλεσμα να εμποδίζουν την περαιτέρω έκδοση εντολών λόγω των υφιστάμενων εξαρτήσεων. Η τεχνική του SMT έρχεται να καλύψει ακριβώς αυτές τις περιπτώσεις υποχρησιμοποίησης του εύρους έκδοσης, όπως βλέπουμε στο Σχήμα 2.1β. Όταν ο παραλληλισμός ενός νήματος είναι χαμηλός, τότε δρομολογούνται στον ίδιο κύκλο εντολές από άλλα νήματα καλύπτοντας έτσι τις κενές υποδοχές. Όταν ένα νήμα καθυστερεί σε κάποιο γεγονός, επιλέγονται άλλα νήματα για εκτέλεση. Με αυτόν τον τρόπο επιτυγχάνεται η μεγιστοποίηση χρησιμοποίησης των πόρων του επεξεργαστή από την ταυτόχρονη επεξεργασία ανεξάρτητων εντολών.



(α) *Υπερβαθμωτή* **(β)** *SMT*

Σχήμα 2.1: *Εκμετάλλευση κενών υποδοχών έκδοσης μιας υπερβαθμωτής αρχιτεκτονικής από το SMT.*

Τα παραπάνω υποδηλώνουν ότι η τεχνική του SMT είναι πιο πολύ προσανατολισμένη στη βελτίωση της απόδοσης από την οπτική γωνία του επεξεργαστή. Όσον αφορά την απόδοση των εφαρμογών, η επιτυχία ή όχι του SMT εξαρτάται σε μεγάλο βαθμό από τη *συμπληρωματικότητα* των απαιτήσεων των νημάτων τους για επεξεργαστικούς πόρους. Η συμπληρωματικότητα αυτή εκφράζεται τόσο ως προς το είδος των πόρων, όσο και ως προς την ποσότητά τους. Για παράδειγμα, όταν ένα νήμα εκτελεί λειτουργίες μνήμης (memory intensive) τη στιγμή που κάποιο άλλο εκτελεί υπολογισμούς (computation intensive), ή όταν το ένα εκτελεί πράξεις κινητής υποδιαστολής (fp bound) και το άλλο πράξεις ακεραίων (integer bound), τότε τα νήματα αναμένεται να συνυπάρξουν αρμονικά, αφού η χρήση των αντίστοιχων μονάδων δεν προκαλεί *συγκρούσεις* (conflicts). Ομοίως, όταν δύο νήματα εκτελούν πράξεις κινητής υποδιαστολής και το καθένα χρησιμοποιεί όχι πάνω από τις μισές αντίστοιχες μονάδες, τότε και πάλι δεν ενδέχεται να υπάρξει ιδιαίτερο πρόβλημα. Πρόβλημα αναμένεται να προκύψει όταν τα νήματα διεκδικούν την ίδια χρονική στιγμή τις ίδιες μονάδες εκτέλεσης. Όσο μεγαλύτερες είναι οι απαιτήσεις των νημάτων για πολλαπλή χρήση κάποιων μονάδων εντός ενός κύκλου (λόγω υψηλού ILP), ή όσο μικρότερη είναι διαθεσιμότητα των μονάδων αυτών σε πολλαπλά αντίγραφα (λόγω περιορισμένης δυνατότητας υπερβαθμωτής εκτέλεσης), τόσο πιο έντονες αναμένεται να είναι οι συγκρούσεις με αποτέλεσμα να προκαλούνται καθυστερήσεις στα νήματα.

Γενικά, ένα πολυπρογραμματιζόμενο φορτίο αποτελούμενο από διαφορετικές μονονηματικές εφαρμογές χαρακτηρίζεται από υψηλή ετερογένεια των εντολών των νημάτων, και έτσι μπορεί να επωφεληθεί από τη χρήση του SMT. Αντίθετα, ένα φορτίο αποτελούμενο από μία παράλληλη εφαρμογή όπου τα νήματα εκτελούν ίδιες λειτουργίες σε διαφορετικά δεδομένα, χαρακτηρίζεται από υψηλή ομοιογένεια στο είδος και τον παραλληλισμό των εντολών που εκτελούνται ανά πάσα στιγμή από κάθε νήμα, με αποτέλεσμα τα κέρδη από το SMT σε αυτή την περίπτωση να είναι σαφώς μικρότερα. Συνεπώς, για να μεγιστοποιηθεί το κέρδος που μπορεί να αποκομίσει μια παράλληλη εφαρμογή από το SMT, πρέπει να εφαρμοστούν τεχνικές που στοχεύουν στην

αύξηση της ετερογένειας των νημάτων της, ξεφεύγοντας από τη συνήθη συμμετρία που διέπει τις μεθόδους παραλληλοποίησης στα παραδοσιακά συστήματα πολυεπεξεργασίας κοινής μνήμης.

2.4 Η Τεχνολογία Hyper-Threading

Ο επεξεργαστής Xeon ήταν ένας από τους πρώτους ευρέως διαθέσιμους επεξεργαστές στους οποίους η Intel εισήγαγε την τεχνική του SMT μέσω της τεχνολογίας Hyper-Threading [Marr 02]. Με το Hyper-threading υποστηρίζονται δύο ταυτόχρονα νήματα εκτέλεσης, τα οποία το λειτουργικό σύστημα αναγνωρίζει σαν δύο διαφορετικούς λογικούς επεξεργαστές για τους οποίους διατηρεί ξεχωριστές ουρές εκτέλεσης. Σχεδόν όλοι οι επεξεργαστικοί πόροι διαμοιράζονται –στατικά ή δυναμικά– ανάμεσα στα δύο νήματα (π.χ. η ιεραρχία μνήμης, οι μονάδες εκτέλεσης, η λογική για φόρτωση, αποκωδικοποίηση, δρομολόγηση και ολοκλήρωση εντολών). Μόνο η αρχιτεκτονική κατάσταση των νημάτων καθώς και δομές που αφορούν τον έλεγχο της εκτέλεσής τους (π.χ. μετρητές προγράμματος, δομές πρόβλεψης διακλάδωσης) βρίσκονται σε πολλαπλά αντίγραφα. Ως εκ τούτου, το επιπλέον κόστος σε υλικό για την υλοποίηση του Hyper-threading δεν ξεπερνούσε το 5% της έκτασης του ολοκληρωμένου του αρχικού επεξεργαστή, ενώ αντίστοιχη ήταν και η αύξηση της κατανάλωσης ισχύος.

2.4.1 Η μικροαρχιτεκτονική Netburst

Ο επεξεργαστής Xeon είναι βασισμένος στην μικροαρχιτεκτονική Netburst [Hinton 01], που χαρακτηρίζεται από βαθιά σωλήνωση, με δυνατότητα υπερβαθμωτής εκτέλεσης και δυναμικής δρομολόγησης εντολών. Στα πλαίσια της διατριβής, χρησιμοποιήσαμε μία από τις τελευταίες εκδόσεις της μικροαρχιτεκτονικής Netburst, γνωστή με το όνομα “Prescott”. Το κύριο χαρακτηριστικό της είναι η πολύ βαθιά σωλήνωση 31 σταδίων. Η ποινή λανθασμένης πρόβλεψης διακλαδώσεων είναι ίση με 20 κύκλους ρολογιού.

Ένα επιπλέον χαρακτηριστικό της μικροαρχιτεκτονικής Netburst είναι η χρήση *μικροεντολών* (micro-ops ή uops) στο μεγαλύτερο μέρος της σωλήνωσης. Οι μικροεντολές είναι επιμέρους λειτουργίες στις οποίες αποκωδικοποιείται μια IA-32 εντολή, και αποθηκεύονται στην *κρυφή μνήμη ιχνών* (trace cache) που λειτουργεί σαν κρυφή μνήμη εντολών πρώτου επιπέδου. Εκτός από γρήγορη πρόσβαση σε εντολές, η κρυφή μνήμη ιχνών απαλλάσσει τον επεξεργαστή από την αναγκαιότητα να αποκωδικοποιεί ξανά εντολές που έχει φορτώσει πρόσφατα κατά το παρελθόν (π.χ. σε βρόχους).

Ο επεξεργαστής μπορεί να φορτώσει τρεις μικροεντολές ανά κύκλο από την κρυφή μνήμη ιχνών, να εκτελέσει μέχρι και έξι μικροεντολές ανά κύκλο, και να ολοκληρώσει μέχρι και τρεις ανά κύκλο. Διαθέτει 128 φυσικούς καταχωρητές για ακέραιους και 128 φυσικούς καταχωρητές

κινητής υποδιαστολής. Διαθέτει ROB 126 θέσεων, ουρά αναγνώσεων 48 θέσεων και ουρά εγγραφών 24 θέσεων. Αυτό σημαίνει ότι ανά πάσα στιγμή 126 εντολές μπορούν να βρίσκονται "εν εκτελέσει" (εκτελούμενες ή περιμένοντας τιμές εισόδου για να εκτελεστούν), εκ των οποίων 48 μπορεί να είναι εντολές ανάγνωσης και 24 εντολές εγγραφής.

2.4.2 Στάδια σωλήνωσης και διαχείριση πόρων

Στο Σχήμα 2.2 παρουσιάζονται τα στάδια της σωλήνωσης του επεξεργαστή Xeon με τεχνολογία Hyper-threading, ενώ με διαφορετικό χρώμα απεικονίζεται η χρήση ενός πόρου από κάποιον συγκεκριμένο λογικό επεξεργαστή. Στις ακόλουθες παραγράφους περιγράφουμε αναλυτικά τα στάδια αυτά καθώς και τον τρόπο που γίνεται η διαχείριση και η εκχώρηση των πόρων στους δύο λογικούς επεξεργαστές.

Φόρτωση: Όταν υπάρχει αστοχία στην κρυφή μνήμη ιχνών, ο επεξεργαστής πρέπει να φορτώσει εντολές από την κρυφή μνήμη L2. Ο *απομονωτής μεταφράσεων για εντολές* (instruction translation lookaside buffer – ITLB) λαμβάνει την αίτηση από την κρυφή μνήμη ιχνών, μεταφράζει την επόμενη διεύθυνση στην οποία δείχνει ο *δείκτης εντολών* (instruction pointer) από εικονική σε φυσική, και την στέλνει στην L2. Η L2 κάποιους κύκλους αργότερα επιστρέφει γραμμές με τις ζητούμενες εντολές, τις οποίες τοποθετεί σε ουρά εντολών.

Κάθε λογικός επεξεργαστής έχει τον δικό του ITLB καθώς και το δικό του δείκτη εντολών, προκειμένου η φόρτωση εντολών να γίνεται ανεξάρτητα για τους δύο λογικούς επεξεργαστές. Οι αιτήσεις για φόρτωση εντολών αποστέλλονται στην L2 με τη σειρά με την οποία δημιουργούνται από τους λογικούς επεξεργαστές ("first-come-first-served"), ενώ οι γραμμές που επιστρέφει η L2 τοποθετούνται σε ξεχωριστή ουρά για κάθε λογικό επεξεργαστή. Οι δομές που σχετίζονται με εντολές διακλάδωσης, είτε βρίσκονται σε ξεχωριστά αντίγραφα για κάθε λογικό επεξεργαστή (return stack buffer, branch history buffer), είτε μοιράζονται δυναμικά με τις εγγραφές τους να εμπεριέχουν αναγνωριστικά νημάτων για τη διάκριση των λογικών επεξεργαστών.

Αποκωδικοποίηση: Η αποκωδικοποίηση IA-32 εντολών είναι σχετικά δύσκολη διαδικασία, ιδιαίτερα απαιτητική σε υλικό, εξαιτίας του ότι οι εντολές έχουν μεταβλητό μήκος και διαφορετικές επιλογές λειτουργίας. Η μονάδα αποκωδικοποίησης δέχεται εντολές από τις ουρές εντολών, τις αποκωδικοποιεί σε μικροεντολές και τις περνάει μέσω άλλων ουρών στην κρυφή μνήμη ιχνών. Όταν και οι δύο λογικοί επεξεργαστές χρειάζεται να αποκωδικοποιήσουν ταυτόχρονα εντολές, η πρόσβαση στη μονάδα εναλλάσσεται ανάμεσά τους, αποκωδικοποιώντας αρκετές εντολές από έναν λογικό επεξεργαστή προτού γίνει μετάβαση στον άλλον. Όταν κάποιος λογικός επεξεργαστής είναι άεργος ή αντιμετωπίζει κάποια καθυστέρηση, τότε ο άλλος επεξεργαστής μπορεί

να χρησιμοποιήσει τη μονάδα αποκωδικοποίησης κατ' αποκλειστικότητα, χωρίς την ανάγκη για πολυπλεξία.

Οι αποκωδικοποιημένες μικροεντολές αποθηκεύονται στην κρυφή μνήμη ιχνών. Η κρυφή μνήμη μοιράζεται δυναμικά ανάμεσα στους λογικούς επεξεργαστές, ενώ κάθε γραμμή εμπεριέχει αναγνωριστικό για κάθε λογικό επεξεργαστή. Σε ταυτόχρονη ζήτηση από τους λογικούς επεξεργαστές, η πρόσβαση στην κρυφή μνήμη ιχνών εναλλάσσεται ανάμεσά τους σε κάθε κύκλο. Όπως και πριν, αν ένας λογικός επεξεργαστής είναι άεργος ή αντιμετωπίζει καθυστέρηση, τότε ο άλλος μπορεί να χρησιμοποιήσει την κρυφή μνήμη στο έπακρο, προσπελαύνοντάς την για πολλούς διαδοχικούς κύκλους.

Οι μικροεντολές φορτώνονται στη συνέχεια από την κρυφή μνήμη ιχνών στην ουρά μικροεντολών. Για πολύπλοκες εντολές χρησιμοποιείται ειδική μνήμη ROM μικροκώδικα για την τροφοδοσία με μικροεντολές. Η ουρά μικροεντολών στην πραγματικότητα είναι ένα νοητό σύνορο ανάμεσα στα πρώτα στάδια της σωλήνωσης, όπου οι εντολές εισέρχονται σε αυτά εντός σειράς προγράμματος, και τα επόμενα στάδια, στα οποία οι εντολές εκτελούνται εκτός σειράς. Η ουρά αυτή είναι στατικά διαχωρισμένη ανάμεσα στους δύο λογικούς επεξεργαστές, ώστε καθένας να μπορεί να χρησιμοποιήσει το πολύ μισές από τις εγγραφές της. Αυτός ο στατικός διαχωρισμός εξασφαλίζει την απρόσκοπτη εκτέλεση ενός νήματος ανεξάρτητα από πιθανές καθυστερήσεις του άλλου (π.χ. αστοχίες στην κρυφή μνήμη ιχνών).

Ανάθεση: Το δυναμικό, υπερβαθμωτό τμήμα εκτέλεσης του επεξεργαστή τροφοδοτείται με μικροεντολές από την ουρά μικροεντολών. Μια *μονάδα ανάθεσης* (allocator) λαμβάνει τις μικροεντολές και αναθέτει σε αυτές διαθέσιμες εγγραφές από βασικές δομές του επεξεργαστή, όπως ROB, ουρά αναγνώσεων, ουρά εγγραφών, ουρές δρομολόγησης. Παράλληλα φροντίζει ώστε κάθε λογικός επεξεργαστής να μην μπορεί να χρησιμοποιήσει πάνω από τις μισές εγγραφές των δομών αυτών. Επιπλέον, αναθέτει με δυναμικό τρόπο διαθέσιμες εγγραφές από το αρχείο καταχωρητών.

Αν κάποια εγγραφή δεν είναι διαθέσιμη, για παράδειγμα επειδή κάποιος λογικός επεξεργαστής έχει φτάσει το όριο χρήσης που του αναλογεί, τότε η μονάδα ανάθεσης μπλοκάρει την εκτέλεση αυτού του λογικού επεξεργαστή. Αν υπάρχουν μικροεντολές στις ουρές και από τους δύο λογικούς επεξεργαστές, τότε η μονάδα ανάθεσης θα επιλέγει εναλλάξ μικροεντολές από κάθε λογικό επεξεργαστή σε κάθε κύκλο. Αν κάποιος λογικός επεξεργαστής δεν έχει μικροεντολές στην ουρά ή αν είναι μπλοκαρισμένος, η μονάδα ανάθεσης θα επιλέγει σε διαδοχικούς κύκλους μικροεντολές από τον άλλον λογικό επεξεργαστή. Βέβαια, ο λογικός επεξεργαστής αυτός θα πρέπει να περιορίζεται πάντα στο μερίδιο εγγραφών που του αντιστοιχούν. Με την επιβολή άνω ορίου στη χρήση των εγγραφών στις δομές που προαναφέραμε προωθείται η δικαιοσύνη στη χρήση πόρων και αποτρέπονται πιθανά αδιέξοδα που θα μπορούσαν να προκύψουν αν ένα πολύ απαιτητικό νήμα καταλάμβανε αυθαίρετα πολλές εγγραφές και απέκλειε το άλλο νήμα

από τη δυνατότητα εκτέλεσης.

Μετονομασία: Παράλληλα με την ανάθεση εγγραφών γίνεται η μετονομασία των καταχωρητών. Σε αυτή τη φάση, οι αρχιτεκτονικοί (λογικοί) καταχωρητές των μικροεντολών κάθε νήματος μετονομάζονται σε μοναδικούς φυσικούς καταχωρητές του επεξεργαστή. Επειδή κάθε λογικός επεξεργαστής πρέπει να διατηρεί τη δική του αρχιτεκτονική κατάσταση, δηλαδή τη δική του απεικόνιση από λογικούς σε φυσικούς καταχωρητές, χρησιμοποιούνται ξεχωριστοί πίνακες ψευδωνύμων καταχωρητών για κάθε λογικό επεξεργαστή.

Δρομολόγηση: Αφού γίνει η ανάθεση πόρων και η μετονομασία, οι μικροεντολές τοποθετούνται σε δύο ουρές δρομολόγησης ανάλογα με το είδος τους – μία για λειτουργίες μνήμης (αναγνώσεις και εγγραφές), και μία για όλες τις άλλες λειτουργίες. Οι ουρές αυτές όπως αναφέραμε είναι στατικά διαχωρισμένες ώστε κάθε λογικός επεξεργαστής να μπορεί να χρησιμοποιεί το πολύ μισές από τις εγγραφές τους. Στέλνουν μικροεντολές σε πέντε δρομολογητές, εναλλάσσοντας την πρόσβαση ανάμεσα στους δύο λογικούς επεξεργαστές σε κάθε κύκλο.

Κάθε δρομολογητής έχει τη δική του ουρά από την οποία διαλέγει μικροεντολές να στείλει στις διάφορες μονάδες εκτέλεσης. Ο δρομολογητής ελέγχει την ετοιμότητα των τελεστέων εισόδου μιας μικροεντολής και τη διαθεσιμότητα της αντίστοιχης μονάδας εκτέλεσης, και τότε δρομολογεί για εκτέλεση την εντολή. Συνολικά οι δρομολογητές μπορούν να στείλουν μέχρι και έξι μικροεντολές για εκτέλεση σε έναν κύκλο. Δεν κάνουν διάκριση όσον αφορά το λογικό επεξεργαστή στον οποίον ανήκουν οι μικροεντολές που επιλέγουν, αφού οι καταχωρητές για όλες τις μικροεντολές έχουν απεικονιστεί σε διαφορετικούς φυσικούς καταχωρητές. Έτσι μπορούν για παράδειγμα να επιλεγούν στον ίδιο κύκλο τρεις μικροεντολές από τον έναν λογικό επεξεργαστή και δύο από τον άλλον και να σταλούν για εκτέλεση. Παρόλα αυτά, για να εξασφαλιστεί η δικαιοσύνη και να αποτραπούν τα αδιέξοδα, υπάρχει ένα άνω όριο στον αριθμό των ενεργών εγγραφών σε κάθε ουρά δρομολογητή που μπορεί να κατέχει ένας λογικός επεξεργαστής.

Εκτέλεση: Όπως και η δρομολόγηση, έτσι και το στάδιο της εκτέλεσης αγνοεί κατά πολύ την προέλευση των μικροεντολών. Η πρόσβαση στο αρχείο καταχωρητών για εγγραφή και ανάγνωση γίνεται με βάση τους φυσικούς καταχωρητές των μικροεντολών, κι έτσι δεν είναι δυνατόν να γίνει διάκριση ανάμεσα στους λογικούς επεξεργαστές. Αυτό υπονοεί ότι ο διαμοιρασμός του αρχείου καταχωρητών και των μονάδων εκτέλεσης γίνεται εντελώς δυναμικά για τους δύο λογικούς επεξεργαστές.

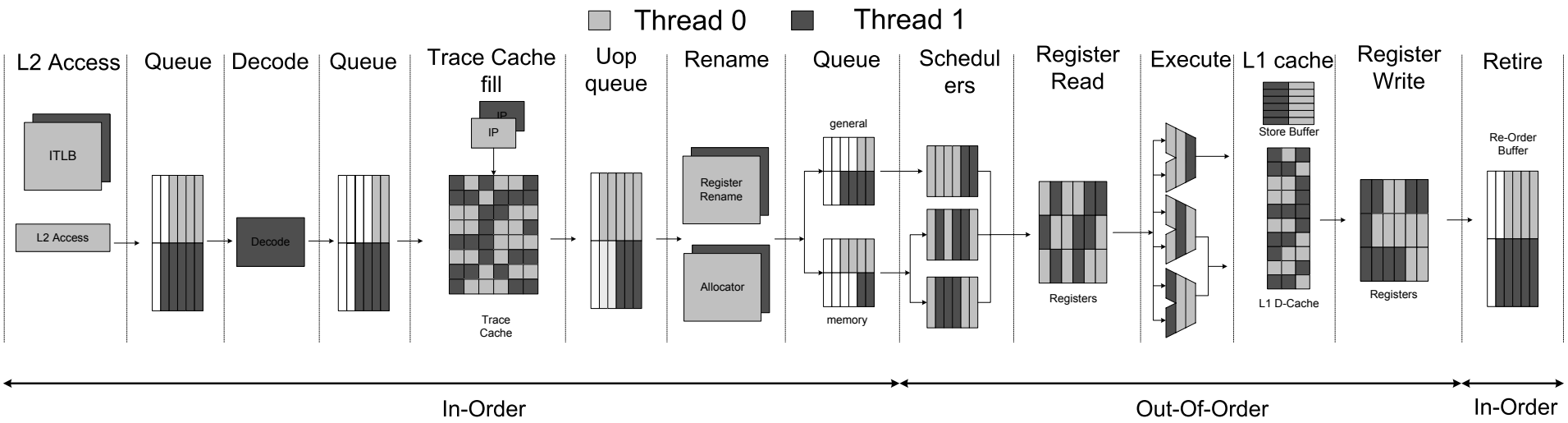
Κατά την πρόσβαση στην ιεραρχία μνήμης επίσης δε γίνονται διακρίσεις ανάμεσα στους λογικούς επεξεργαστές. Η κρυφή μνήμη δεδομένων L1 καθώς και η ενοποιημένη κρυφή μνήμη L2 διαμοιράζονται δυναμικά, και επιπλέον, κάθε λογικός επεξεργαστής μπορεί να προσπελάζει τις

γραμμές δεδομένων που έφερε αρχικά ο άλλος στην κρυφή μνήμη. Ο *απομονωτής μεταφράσεων για δεδομένα* (data translation lookaside buffer – DTLB) παρέχει γρήγορες μεταφράσεις εικονικών σε φυσικές διευθύνσεις. Ο DTLB είναι διαμοιραζόμενος ανάμεσα στους δύο λογικούς επεξεργαστές, αλλά κάθε εγγραφή του εμπεριέχει αναγνωριστικό για τη διάκριση μεταξύ τους.

Ολοκλήρωση: Το στάδιο της ολοκλήρωσης είναι υπεύθυνο για την απόσυρση των εντολών και την ανανέωση της αρχιτεκτονικής κατάστασης των λογικών επεξεργαστών στη σειρά εντολών του προγράμματος. Οι μικροεντολές έχουν τοποθετηθεί στον ROB στη σειρά προγράμματος κατά τη φάση της ανάθεσης. Μεταγενέστερες μικροεντολές μπορεί να έχουν ολοκληρώσει την εκτέλεσή τους νωρίτερα από ό,τι προγενέστερες, αλλά ο ROB δε θα τις αφήσει να ανανεώσουν την αρχιτεκτονική κατάσταση του επεξεργαστή αν πρώτα δεν ολοκληρωθούν οι παλαιότερες.

Ο ROB είναι στατικά διαχωρισμένος ώστε κάθε λογικός επεξεργαστής να μπορεί να χρησιμοποιήσει το πολύ μισές εγγραφές. Σε ταυτόχρονη ζήτηση από τους δύο λογικούς επεξεργαστές η πρόσβαση στον ROB εναλλάσσεται σε κάθε κύκλο. Όταν κάποιος λογικός επεξεργαστής δεν έχει έτοιμες εντολές να αποσύρει, τότε ο άλλος μπορεί να χρησιμοποιήσει τον ROB κατ' αποκλειστικότητα σε διαδοχικούς κύκλους.

Μετά την απόσυρση από τον ROB των λειτουργιών εγγραφής, τα αντίστοιχα δεδομένα εγγράφονται στην κρυφή μνήμη δεδομένων L1. Αυτό γίνεται μέσω της ουράς εγγραφών, η οποία επιλέγει εναλλάξ σε κάθε κύκλο εγγραφές από τους δύο λογικούς επεξεργαστές για να στείλει στην κρυφή μνήμη.



Σχήμα 2.2: *Τα στάδια της σωλήνωσης και η διαχείριση των πόρων του επεξεργαστή Χeon με τεχνολογία Hyper-threading.*

Δυναμικά διαμοιραζόμενοι πόροι	Μονάδες εκτέλεσης, λογική φόρτωσης - αποκωδικοποίησης - δρομολόγησης - ολοκλήρωσης εντολών Κρυφή μνήμη εντολών (trace cache), κρυφή μνήμη δεδομένων 1ου επιπέδου (L1 D-cache), ενοποιημένη κρυφή μνήμη 2ου επιπέδου (L2 cache) DTLB, Global history array, ROM μικροκώδικα
Πόροι σε πολλαπλά αντίγραφα	Αρχιτεκτονική κατάσταση λογικών επεξεργαστών, δείκτες εντολών, λογική μετονομασίας καταχωρητών, ITLB, streaming buffers, return stack buffer, branch history buffer
Στατικά διαχωρισμένοι πόροι	Ουρά μικροεντολών, ουρά γενικών εντολών/εντολών μνήμης, ουρά αναγνώσεων/εγγραφών, απομονωτής αναδιάταξης εντολών (re-order buffer)

Πίνακας 2.1: *Διαχείριση πόρων σε έναν επεξεργαστή με τεχνολογία Hyper-threading.*

Ο Πίνακας 2.1 συνοψίζει τον τρόπο με τον οποίο γίνεται η διαχείριση των πόρων σε έναν επεξεργαστή με τεχνολογία Hyper-threading.

2.4.3 Διαφορές με τα θεωρητικά μοντέλα SMT

Όπως σημειώνεται στο [Tuck 03], ο στατικός διαχωρισμός πολλών δομών του επεξεργαστή αντί του δυναμικού τους διαμοιρασμού είναι η βασικότερη διαφορά της τεχνολογίας Hyper-threading από το θεωρητικό μοντέλο SMT όπως αρχικά προτάθηκε στο [Tullsen 95]. Η έρευνα πάνω στην τεχνική του SMT έχει υποστηρίξει ότι ο δυναμικός διαμοιρασμός είναι πιο αποτελεσματικός από το στατικό διαχωρισμό. Όμως, όπως αναφέραμε στα προηγούμενα, ο στατικός διαχωρισμός βοηθάει ένα νήμα να μην επηρεάζεται από κάποιο άλλο που έχει "ακραία" συμπεριφορά, όπως για παράδειγμα ένα νήμα πολύ απαιτητικό στη χρήση πόρων. Όπως και να 'χει, με δύο μόνο νήματα υλικού στην τεχνολογία Hyper-threading τα όποια μειονεκτήματα του στατικού διαχωρισμού μετριάζονται και η απόδοση προσεγγίζει αυτήν μιας υλοποίησης βασισμένης στο δυναμικό διαμοιρασμό. Με περισσότερα νήματα υλικού όμως, όπως για παράδειγμα τα οχτώ του προτεινόμενου θεωρητικού μοντέλου, ο στατικός διαχωρισμός του Hyper-threading δε θα ήταν "κλιμακώσιμος" και θα οδηγούσε πιθανότατα σε σημαντική υποβάθμιση της απόδοσης.

2.5 Ποσοτική Ανάλυση των Ορίων TLP και ILP της Τεχνολογίας Hyper-threading

Για να αποκτήσουμε μια σαφέστερη εικόνα για την ικανότητα και τα όρια της τεχνολογίας Hyper-threading να επικαλύψει εντολές από ανεξάρτητα νήματα, αξιολογήσαμε την ταυτόχρονη εκτέλεση ομογενών συνθετικών ροών εντολών. Αυτές οι ροές περιλαμβάνουν βασικές αριθμητικές πράξεις (add, sub, mul, div) καθώς και λειτουργίες μνήμης (load, store) με ακέραιους αριθμούς και αριθμούς κινητής υποδιαστολής, όλοι των 64 bits. Για την κάθε περίπτωση πειραματιστήκαμε με διαφορετικούς συνδυασμούς παραλληλισμού επιπέδου εντολών και παραλληλισμού επιπέδου νημάτων, προκειμένου να εκτιμήσουμε τα άνω και κάτω όρια του Hyper-threading στο να πολυπλέξει τις διάφορες μορφές παραλληλισμού. Αποτιμήσαμε δύο διαφορετικές υλοποιήσεις της τεχνολογίας˙ την αρχική υλοποίηση που εισήγαγαν οι επεξεργαστές Xeon/Pentium 4, καθώς και μια πιο πρόσφατη υλοποίηση που ενσωματώθηκε στην μικροαρχιτεκτονική Nehalem της Intel.

Κάθε ροή εντολών κατασκευάστηκε εισάγοντας στο σώμα ενός αέναου βρόχου μεγάλο αριθμό (>20K) εντολών γλώσσας μηχανής (assembly) συγκεκριμένου τύπου. Σε κάθε επανάληψη μετράμε τον αριθμό των εντολών καθώς και τους κύκλους [2] που διήρκεσε η εκτέλεσή τους και τους συναθροίζουμε με αυτούς από τις προηγούμενες επαναλήψεις. Εκτελούμε το βρόχο για μεγάλο χρονικό διάστημα (περίπου 15 δευτερόλεπτα). Ο συνολικός αριθμός των κύκλων προς το συνολικό αριθμό εντολών μας δίνει το CPI για μια συγκεκριμένη ροή εντολών.

Όλες οι αριθμητικές πράξεις είναι μεταξύ καταχωρητών. Για τις πράξεις κινητής υποδιαστολής χρησιμοποιήσαμε τις SSE εντολές του επεξεργαστή. Η ροή εντολών μνήμης έχει κατασκευαστεί εναλλάσσοντας εντολές ανάγνωσης και εγγραφής, δηλαδή λειτουργίες μεταφοράς δεδομένων από θέσεις της κύριας μνήμης σε καταχωρητές και αντίστροφα Σε αυτήν την περίπτωση κάθε νήμα λειτουργεί πάνω σε ένα ιδιωτικό διάνυσμα αριθμών το οποίο διατρέχει ακολουθιακά. Το διάνυσμα χωρά εξολοκλήρου στην κρυφή μνήμη L1 και επαναχρησιμοποιείται σε κάθε επανάληψη.

Για κάθε ροή εξετάζουμε τρία διαφορετικά επίπεδα παραλληλισμού εντολών, ελάχιστο (ILP = 1), μεσαίο (ILP = 3) και μέγιστο (ILP $\geq$ 6). Για την υλοποίηση μιας ροής με ILP = k ρυθμίζουμε κατάλληλα τα ορίσματα των εντολών στο εσωτερικό του βρόχου ώστε να δημιουργούνται k αλληλουχίες από εξαρτώμενες εντολές. Οι αλληλουχίες είναι ανεξάρτητες μεταξύ τους, όμως οι εντολές μέσα σε κάθε αλληλουχία εξαρτώνται η κάθε μία από την προηγούμενη. Για να το επιτύχουμε αυτό, φροντίζουμε καταρχήν κάθε αλληλουχία να χρησιμοποιεί διαφορετικό υποσύνολο καταχωρητών, και στη συνέχεια κάθε εντολή μιας αλληλουχίας να χρησιμοποιεί σαν τελεστέο

[2] Χρησιμοποιώντας τους μετρητές χρονισμού του επεξεργαστή (time-stamp counters).

Σχήμα 2.3: *Παράδειγμα ροής εντολών με ILP=3.*

εισόδου τον τελεστέο εξόδου της προηγούμενης εντολής. Με αυτόν τον τρόπο δημιουργούμε τεχνητά πολλαπλές "αλυσίδες" RAW εξαρτήσεων. Σημειώνουμε ότι μόνο αυτό το είδος εξαρτήσεων είναι δυνατό να περιορίσει τον παραλληλισμό σε έναν σύγχρονο επεξεργαστή δυναμικής εκτέλεσης όπως ο Xeon, αφού τόσο οι WAW όσο και οι WAR εξαρτήσεις αντιμετωπίζονται μέσω μηχανισμών μετονομασίας καταχωρητών.

Στο Σχήμα 2.3 φαίνεται ένα παράδειγμα ροής εντολών με ILP = 3. Οι τρεις αλληλουχίες εξαρτώμενων εντολών έχουν επικαλυφθεί μεταξύ τους ώστε ο επεξεργαστής να μπορεί να ψάχνει για ανεξάρτητες εντολές σε ένα μικρό παράθυρο. Για την περίπτωση μέγιστου ILP, δεν υπάρχουν καθόλου εξαρτήσεις ανάμεσα στις εντολές. Ένας αρχιτεκτονικός καταχωρητής χρησιμοποιείται από όλες τις εντολές σαν τελεστέος εισόδου, και όλοι οι υπόλοιποι επαναχρησιμοποιούνται κυκλικά σαν τελεστέοι εξόδου. Για τα περισσότερα ήδη εντολών αυτό οδηγεί σε 15 "παράλληλες" εντολές (όσοι και οι αρχιτεκτονικοί καταχωρητές που μπορούν να χρησιμοποιηθούν σαν τελεστέοι εξόδου), που όμως πρακτικά είναι αρκετά περισσότερες αφού ένας καταχωρητής εξόδου μπορεί να απεικονιστεί δυναμικά σε διαφορετικούς φυσικούς καταχωρητές. Προφανώς όμως, οι υπερβαθμωτές μονάδες του επεξεργαστή δεν μπορούν να εκτελέσουν στον ίδιο κύκλο τέτοιο μεγάλο αριθμό εντολών.

Σημειώνουμε ότι στην περίπτωση των πράξεων πολλαπλασιασμού και διαίρεσης ακεραίων οι καταχωρητές edx και eax υπονοούνται πάντα σαν καταχωρητές εξόδου από το σύνολο εντολών της αρχιτεκτονικής. Αυτό σημαίνει ότι δεν μπορούμε να ρυθμίσουμε με τόσο ευέλικτο τρόπο το ILP για αυτές τις εντολές όπως κάνουμε για όλες τις υπόλοιπες. Επιπλέον, στις περιπτώσεις εντολών μνήμης οι διαφορετικές αλληλουχίες δουλεύουν σε διαφορετικές περιοχές του διανύσματος οι οποίες βρίσκονται αρκετά μακρυά μεταξύ τους.

Σε πρώτη φάση, εκτελούμε κάθε ροή εντολών μόνη της σε έναν λογικό επεξεργαστή, για όλα τα διαφορετικά επίπεδα ILP. Με αυτόν τον τρόπο όλοι οι επεξεργαστικοί πόροι του φυσικού πακέτου είναι διαθέσιμοι πλήρως στο νήμα που εκτελεί τη ροή. Σε δεύτερη φάση, συνεκτελούμε στο ίδιο φυσικό πακέτο δύο ανεξάρτητες ροές εντολών του ίδιου επιπέδου ILP τις οποίες απεικονίζουμε σε διαφορετικούς λογικούς επεξεργαστές. Πειραματιζόμαστε με όλους τους πιθανούς συνδυασμούς ροών εντολών. Για κάθε συνδυασμό πραγματοποιούμε αντίστοιχη με πριν μέτρηση του CPI για κάθε ροή και εν συνεχεία υπολογίζουμε τον παράγοντα κατά τον οποίον η εκτέλεση μιας συγκεκριμένης ροής επιβραδύνθηκε σε σχέση με την απομονωμένη της εκτέλεση. Αυτός ο παράγοντας μας δίνει μια ποσοτική ένδειξη για το βαθμό στον οποίον τα διάφορα είδη ροών συγκεκριμένου ILP συγκρούονται μεταξύ τους για διαμοιραζόμενους πόρους του επεξεργαστή. Για παράδειγμα, όταν η ροή μιας εντολής A συνεκτελείται με τη ροή μιας εντολής B αλλά το CPI της A δεν αυξάνεται σημαντικά, τότε αυτό υποδηλώνει χαμηλό επίπεδο σύγκρουσης για χρήση κοινών πόρων.

Υπάρχει και μια επιπλέον πληροφορία που μπορούμε να εξάγουμε από τα πειράματα αυτά. Για μια συγκεκριμένη ροή εντολών μπορούμε να εκτιμήσουμε πώς επηρεάζει την απόδοση η μετάβαση από την μονονηματική εκτέλεση υψηλού παραλληλισμού στην πολυνηματική εκτέλεση χαμηλότερου παραλληλισμού. Με άλλα λόγια, μπορούμε να εκτιμήσουμε αν είναι καλύτερο να υλοποιήσουμε ένα συγκεκριμένο επίπεδο παραλληλισμού ως ILP ή ως TLP, δεδομένου ότι τα δύο είδη παραλληλισμού είναι λειτουργικά ισοδύναμα. Για παράδειγμα, ας θεωρήσουμε ένα σενάριο όπου στην μονονηματική εκτέλεση με μέγιστο ILP η εντολή A δίνει μέσο CPI ίσο με $C_{1thr-maxILP}$, ενώ στην πολυνηματική εκτέλεση με ενδιάμεσο ILP η ίδια εντολή δίνει μέσο CPI σε κάθε ροή ίσο με $C_{2thr-medILP} > 2 \times C_{1thr-maxILP}$. Επειδή στη δεύτερη περίπτωση κάθε νήμα έχει, χονδρικά, το μισό ILP της πρώτης περίπτωσης, αυτό το αποτέλεσμα μας υποδεικνύει ότι για αυτό το είδος εντολής υπάρχουν απώλειες όταν ο υψηλός παραλληλισμός στα πλαίσια της μονονηματικής εκτέλεσης κατανέμεται σε πολλαπλά νήματα. Έτσι, πιθανότατα δε θα πρέπει να αναμένουμε κάποια βελτίωση από την παραλληλοποίηση ενός προγράμματος που στη σειριακή του εκτέλεση κάνει εκτενή χρήση αυτής της εντολής σε υψηλά επίπεδα ILP (π.χ. ξεδίπλωμα βρόχων).

2.5.1 Συνεκτελώντας ροές του ίδιου τύπου

Τα Σχήματα 2.4α και 2.4β παρουσιάζουν τα αποτελέσματα για το μέσο CPI μιας σειράς συνθετικών ροών εντολών, για τις υλοποιήσεις Xeon και Nehalem, αντίστοιχα. Παρουσιάζουν πώς διαφορετικοί συνδυασμοί TLP και ILP για μια συγκεκριμένη ροή επηρεάζουν την εκτέλεσή της. Οι ροές που παρουσιάζονται στο διάγραμμα είναι μερικές από τις πιο συνηθισμένες που συναντώνται σε πραγματικές εφαρμογές. Όπως θα δούμε και σε επόμενη ενότητα, το $fadd-mul$ είναι ένα αντιπροσωπευτικό μίγμα εντολών στα μετροπρογράμματα που εξετάζουμε. Η αντίστοιχη

ροή κατασκευάζεται εναλλάσσοντας μέσα στον κώδικα του προγράμματος εντολές κινητής υποδιαστολής πρόσθεσης και πολλαπλασιασμού.

Ας θεωρήσουμε αρχικά τη ροή $fadd$. Στην περίπτωση του ελάχιστου ILP ο μέσος αριθμός κύκλων της εντολής αλλάζει αμελητέα όταν πηγαίνουμε από το ένα στα δύο νήματα, γεγονός που υποδηλώνει πρακτικά βελτίωση του χρόνου εκτέλεσης στην πολυνηματική περίπτωση και για τις δύο αρχιτεκτονικές. Αυτό αποκαλύπτει ότι σε αυτή την περίπτωση ο επεξεργαστής μπορεί να επικαλύψει επιτυχώς εντολές από διαφορετικά νήματα, καταφέρνοντας να κρύψει τις καθυστερήσεις που δημιουργούνται στη σωλήνωση εξαιτίας των κινδύνων δεδομένων. Παρόλα αυτά, η περίπτωση του ελαχίστου ILP δεν είναι αυτή που δίνει την καλύτερη απόδοση συνολικά. Ο μεγαλύτερος ρυθμός ολοκλήρωσης εντολών επιτυγχάνεται γενικά στην μονονηματική εκτέλεση με μέγιστο ILP, όπως φαίνεται από το ίδιο διάγραμμα. Επιπλέον, αν κατανείμουμε τον παραλληλισμό αυτόν στα δύο νήματα (*2thr-medILP*) δε θα έχουμε κάποιο κέρδος σε σχέση με την περίπτωση *1thr-maxILP*, αφού το CPI της τελευταίας είναι το μισό της πρώτης. Το ίδιο ισχύει ακόμα και αν επαναλάβουμε τον παραλληλισμό στο μέγιστό του βαθμό και στο δεύτερο λογικό επεξεργαστή (*2thr-maxILP*). Αυτό σημαίνει ότι η εκτέλεση των εντολών στην πολυνηματική περίπτωση σειριοποιείται. Φαίνεται λοιπόν ότι όταν ο διαθέσιμος ILP σε ένα πρόγραμμα αυξάνεται, τα προβλήματα καθυστερήσεων στη σωλήνωση μειώνονται αναδεικνύοντας εντονότερα το συναγωνισμό για επεξεργαστικούς πόρους.

Από τα ίδια σχήματα βλέπουμε ότι και η ροή $fmul$ επιδεικνύει παρόμοια συμπεριφορά στη διακύμανση του CPI για τους διαφορετικούς συνδυασμούς. Μάλιστα σε αυτή τη ροή υπάρχει βελτίωση της πολυνηματικής εκτέλεσης όχι μόνο για το χαμηλό επίπεδο ILP αλλά και για το μεσαίο, με την υλοποίηση του Nehalem να φαίνεται αποτελεσματικότερη στην επικάλυψη τέτοιων εντολών από διαφορετικά νήματα. Τα ίδια συμπεράσματα ισχύουν και για την περίπτωση της ροής $fadd - mul$. Παρατηρούμε αρχικά ότι η απόδοσή της προσεγγίζει τον μέσο όρο των CPIs των επιμέρους ροών από τις οποίες αποτελείται. Επιπλέον, στην περίπτωση μέγιστου ILP στον επεξεργαστή Xeon βλέπουμε ότι η πολυνηματική εκτέλεση της ροής όχι μόνο σειριοποιείται αλλά φαίνεται να καθυστερείται επιπλέον, πιθανολογούμε λόγω συγκρούσεων στις μονάδες εκτέλεσης.

Για τη ροή $iadd$ στον Xeon δεν είναι ξεκάθαρο ποιος τρόπος λειτουργίας δίνει τους καλύτερους χρόνους εκτέλεσης, αφού ο ρυθμός ολοκλήρωσης εντολών παραμένει σχεδόν ο ίδιος σε όλες τις περιπτώσεις. Η πολυνηματική εκτέλεση της $fdiv$ σειριοποιείται στον Nehalem. Η ίδια περίπτωση στον Xeon φαίνεται ότι επιβραδύνεται από συγκρούσεις, αφού η πολυνηματική εκτέλεση είναι περισσότερο από 2.3 φορές πιο αργή σε σχέση με τη σειριακή. Η υλοποίηση του Nehalem φαίνεται αρκετά αποδοτική στην πολυνηματική εκτέλεση της ροής $imem$, για όλα τα επίπεδα παραλληλισμού εντολών. Αντίθετα ο Xeon είναι αποτελεσματικός μόνο στο μικρότερο επίπεδο αφού στα δύο μεγαλύτερα το CPI της πολυνηματικής περίπτωσης είναι υπερδιπλάσιο

από αυτό της μονονηματικής. Ειδικά για την περίπτωση μέγιστου ILP οι συγκρούσεις φαίνεται να είναι ιδιαίτερα έντονες.

2.5.2 Συνεκτελώντας ροές διαφορετικού τύπου

Τα Σχήματα 2.5α,β και 2.6α,β παρουσιάζουν τα αποτελέσματα από την ταυτόχρονη εκτέλεση ζευγαριών διαφορετικών ροών για τις αρχιτεκτονικές Xeon και Nehalem, αντίστοιχα. Εξετάζουμε ζευγάρια ροών ίδιου επιπέδου ILP επειδή πιστεύουμε ότι αυτή είναι η συνήθης περίπτωση στις περισσότερες πολυνηματικές εφαρμογές. Ο παράγοντας επιβράδυνσης εκφράζει το λόγο του CPI της ροής που υποδεικνύεται στον πάνω x-άξονα όταν αυτή συνεκτελείται με τις ροές που φαίνονται στον κάτω x-άξονα, προς το CPI που έχει όταν εκτελείται μόνη της. Τα βασικότερα συμπεράσματα που προκύπτουν συνοψίζονται στα εξής:

- Στον Xeon, οι ροές ακεραίων επιβραδύνονται σε σημαντικό βαθμό από την $iadd$ (κατά 150% κατά μέσο όρο). Σε μικρότερο βαθμό επηρεάζονται από την $imem$ (κατά 40% οι αριθμητικές), ενώ οι $imul$ και $idiv$ είναι γενικά αυτές που επιβραδύνουν άλλες ροές λιγότερο (κατά 20% τις αριθμητικές). Η $imem$ είναι αυτή που επηρεάζεται περισσότερο από τη συνεκτέλεση με άλλες εντολές, και στις περισσότερες περιπτώσεις επιβραδύνεται πάνω από 50%. Υφίσταται τη μεγαλύτερη επιβράδυνση όταν συνεκτελείται με όμοια ροή μέγιστου ILP (περίπου κατά 7.5 φορές). Το αξιοπερίεργο σε αυτήν την περίπτωση είναι ότι όταν συνεκτελείται με αριθμητικές εντολές τα ποσοστά επιβράδυνσης μειώνονται καθώς αυξάνεται ο ILP. Όταν συνεκτελείται με τον εαυτό της, αντίθετα, αυξάνονται και μάλιστα με μεγάλο ρυθμό.

  Καλύτερη εικόνα παρουσιάζει η υλοποίηση του Hyper-threading στον Nehalem, που όπως βλέπουμε από το Σχήμα 2.6α είναι ικανή να επικαλύψει την εκτέλεση των εντολών πολύ πιο αποδοτικά. Η βελτίωση είναι ιδιαίτερα εμφανής στην $imem$ αλλά και στις $imul$, $idiv$, οι οποίες επιβραδύνονται από όλες τις υπόλοιπες όχι πάνω από 5-10% στη συντριπτική πλειοψηφία των συνδυασμών. Εξαίρεση αποτελεί η $iadd$, που ειδικά για υψηλά επίπεδα ILP επιβραδύνεται τουλάχιστον κατά 40%.

- Στην ταυτόχρονη εκτέλεση ροών κινητής υποδιαστολής στον Xeon οι συγκρούσεις φαίνεται να είναι λιγότερο έντονες σε σχέση με τις ροές ακεραίων. Στο χαμηλότερο επίπεδο ILP, καμία ροή δεν επιβραδύνεται περισσότερο από 10% εκτός από την $fdiv$ όταν εκτελείται με τον εαυτό της. Στα υψηλότερα επίπεδα, οι περισσότερες ροές γενικά επηρεάζονται περισσότερο από τη συνεκτέλεση με τον εαυτό τους (οι $fadd$, $fmul$ και $fdiv$ επιβραδύνονται κατά 95%, 80% και 130%, αντίστοιχα). Σε αντίθεση με τις ακέραιες ροές, οι αριθμητικές πράξεις κινητής υποδιαστολής επιβραδύνονται ελάχιστα από τις λειτουργίες μνήμης (λιγότερο από 7%). Η $fmem$ αντίθετα επηρεάζεται τόσο από τις αριθμητικές εντολές (κατά

40% για μέγιστο ILP) όσο και από τον εαυτό της (7.9 φορές πιο αργή για μέγιστο ILP). Γενικά τα ζεύγη $fadd$-$fdiv$ και η $fmem$ με όλες τις αριθμητικές πράξεις μπορούν να συνυπάρχουν σε ικανοποιητικό βαθμό, με αμοιβαίες επιβραδύνσεις που δεν ξεπερνούν το 7% της σειριακής τους εκτέλεσης.

Παρόμοια συμπεριφορά στη διακύμανση των ποσοστών επιβράδυνσης παρατηρείται και στον Nehalem. Σε 33 από τα 48 ζεύγη σημειώνεται επιβράδυνση μικρότερη από 10%. Τα ζεύγη $fadd$-$fmul$, $fadd$-$fdiv$ καθώς και η $fmem$ με όλες τις αριθμητικές ροές συνυπάρχουν αρμονικά, αφού στη συντριπτική πλειοψηφία των περιπτώσεων η αμοιβαία επιβράδυνση που παρατηρείται είναι μικρότερη από 10%. Οι $fadd$, $fmul$ και $fdiv$ επιβραδύνονται περισσότερο από τη συνεκτέλεση με τον εαυτό τους, κατά 100% στη χειρότερη περίπτωση. Η $fmem$, τέλος, υφίσταται αρκετά περιορισμένη επιβράδυνση σε σχέση με την περίπτωση του Xeon, κατά μέσο όρο ίση με 12%.

Τέλος, εκτελέσαμε ταυτόχρονα ροές εντολών ακεραίων με ροές εντολών κινητής υποδιαστολής. Τέτοια ετερογενή μίγματα εντολών συναντώνται πιο πολύ σε πολυπρογραμματιζόμενα φορτία ή ακανόνιστους παράλληλους κώδικες, απ' ό,τι σε συνήθεις πολυνηματικές εφαρμογές. Τα αποτελέσματα απεικονίζονται στα Σχήματα 2.7α και 2.7β, για τους επεξεργαστές Xeon και Nehalem, αντίστοιχα. Για λόγους συμπαγούς αναπαράστασης, παρουσιάζουμε και σε αυτήν την περίπτωση συνεκτελούμενα ζεύγη με ίδιο επίπεδο ILP.

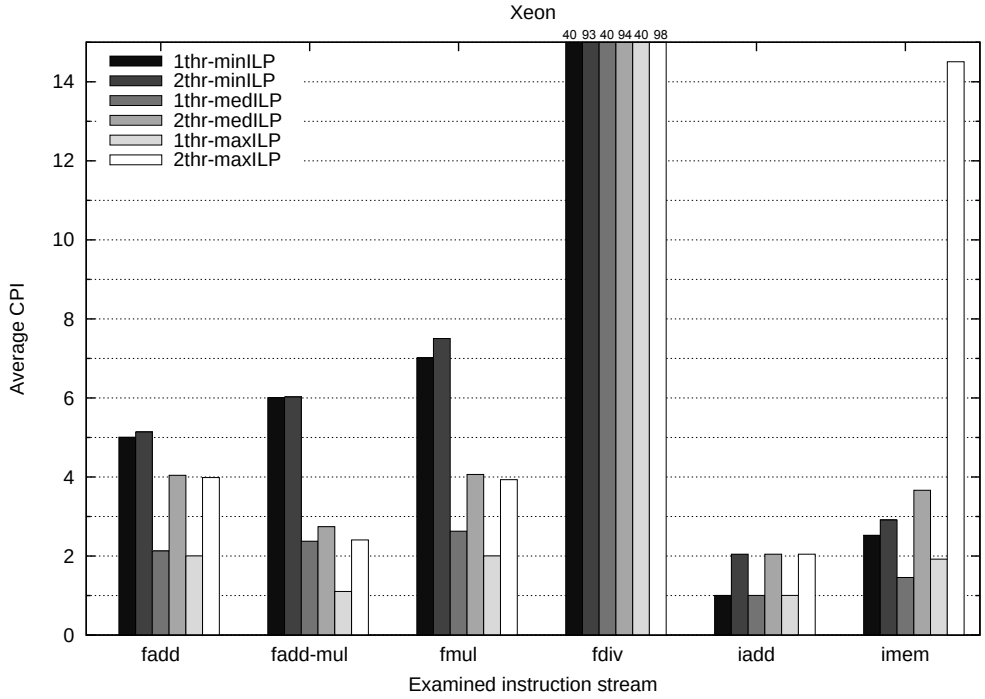
Εκ πρώτης όψεως, παρατηρούμε χαμηλότερα ποσοστά επιβράδυνσης στον Nehalem σε σχέση με τον Xeon, με 65 από τα 96 συνολικά ζεύγη ροών να σημειώνουν επιβράδυνση μικρότερη από 10% (έναντι 40 στα 96 στον Xeon). Η δεύτερη παρατήρηση που αφορά και τις δύο υλοποιήσεις είναι ότι οι ροές κινητής υποδιαστολής εμποδίζονται από τις ροές ακεραίων λιγότερο από ό,τι κάνουν οι τελευταίες με τις πρώτες. Κατά μέσο όρο (και αγνοώντας κάποιες μεμονωμένες περιπτώσεις ιδιαίτερα υψηλών ποσοστών) οι ροές κινητής υποδιαστολής επηρεάζονται κατά 47% και 8% από τις ροές ακεραίων, σε Xeon και Nehalem αντίστοιχα, ενώ οι τελευταίες επιβραδύνονται από τις πρώτες κατά 114% και 42%.

Μία άλλη ενδιαφέρουσα παρατήρηση είναι ότι οι ακέραιες αριθμητικές ροές στον Xeon επιβραδύνονται γενικά περισσότερο όταν συνεκτελούνται με τις αντίστοιχες ροές κινητής υποδιαστολής (Σχ. 2.7α), από ό,τι όταν συνεκτελούνται με όμοιές τους ακέραιες ροές (Σχ. 2.5α). Αυτή η συμπεριφορά είναι ιδιαίτερα αισθητή για τις $imul$ και $idiv$. Στον Nehalem η επιβράδυνση που υφίστανται οι ακέραιες ροές είναι γενικά στα ίδια επίπεδα τόσο για την ετερογενή όσο και για την ομογενή συνεκτέλεση, με εξαίρεση κάποιες μεμονωμένες περιπτώσεις (π.χ. $idiv$). Σε κάθε περίπτωση πάντως, η επιβράδυνση που υφίστανται οι ακέραιες ροές και στις δύο αρχιτεκτονικές κατά την ετερογενή συνεκτέλεσή τους δεν είναι αισθητά μικρότερη σε σχέση με την ομογενή συνεκτέλεση, όπως θα περίμενε κανείς με βάση τη γενικότερη πεποίθηση που θέλει το SMT να ευνοεί ετερογενή μίγματα εντολών.

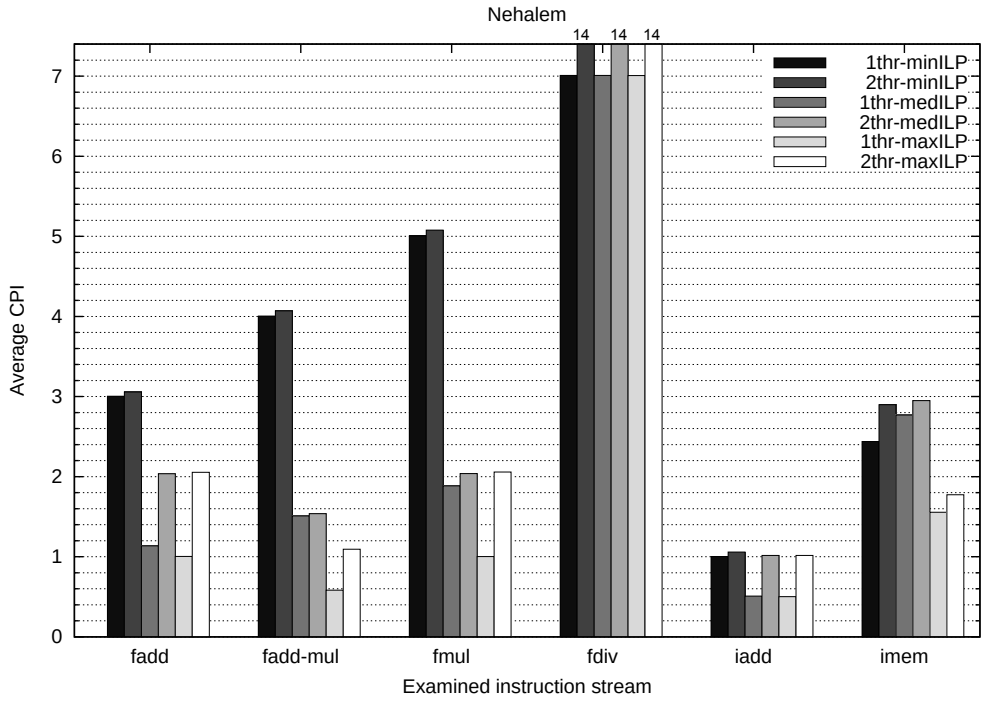
Στην αντίθετη περίπτωση, η ετερογενής συνεκτέλεση για τις ροές κινητής υποδιαστολής είναι γενικά καλύτερη από την ομογενή συνεκτέλεση και στις δύο αρχιτεκτονικές, με λιγότερες περιπτώσεις να σημειώνουν μεγάλα ποσοστά επιβράδυνσης (π.χ. $imul$ στον Nehalem, $idiv$ στον Xeon).

Από τα ζεύγη με τις μικρότερες αμοιβαίες επιβραδύνσεις, διακρίνουμε τα $imem$-$\{fadd, fmul, fdiv, fmem\}$ και $fmem$-$\{imul, idiv\}$ στον Nehalem, ενώ στον Xeon τα $iadd$-$fdiv$ και $imem$-$\{fadd, fmul, div\}$ (για ILP=1). Ανάμεσα στις ροές που υφίστανται τη μεγαλύτερη επιβράδυνση, διακρίνουμε τις $iadd$ και $idiv$ στον Nehalem, και τις $fmul$ και όλες τις ακέραιες ροές στον Xeon.

Ανακεφαλαιώνοντας, η γενική εικόνα που αποκτήσαμε από την ταυτόχρονη εκτέλεση των συνθετικών ροών συνοψίζεται στα ακόλουθα: για τα διάφορα είδη εντολών, ικανοποιητική επικάλυψη επιτυγχάνεται γενικά για χαμηλά επίπεδα ILP. Οι ροές εντολών κινητής υποδιαστολής είναι αυτές που επιβραδύνονται λιγότερο, τόσο κατά τη συνεκτέλεση με άλλες ροές κινητής υποδιαστολής αλλά ιδιαίτερα όταν εκτελούνται μαζί με ακέραιες εντολές. Οι ακέραιες ροές, αντίθετα, υφίστανται μεγαλύτερη επιβράδυνση, η οποία είναι ιδιαίτερα αισθητή κατά τη συνεκτέλεση με ροές κινητής υποδιαστολής. Σε αντίθεση με ό,τι θα περίμενε κανείς, οι εντολές μνήμης επηρεάζονται σε αξιοσημείωτο βαθμό όταν συνεκτελούνται με αριθμητικές εντολές (ειδικά στην περίπτωση των ακεραίων εντολών). Οι παραπάνω διαπιστώσεις αφορούν κυρίως την πρώτη υλοποίηση του Hyper-threading στον επεξεργαστή Xeon. Στη νεότερη υλοποίηση του Nehalem, η συνεκτέλεση των εντολών γίνεται πολύ πιο αποτελεσματικά, με σαφώς μικρότερα ποσοστά επιβράδυνσης και λίγες "προβληματικές" περιπτώσεις.

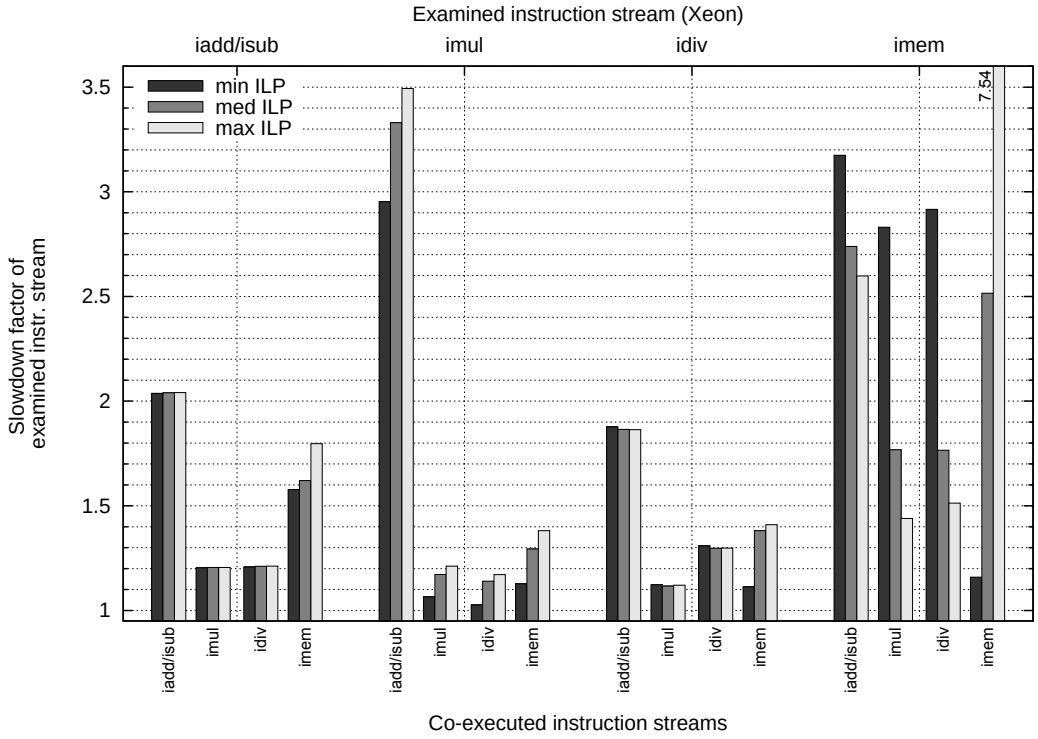


(α) *Xeon.*

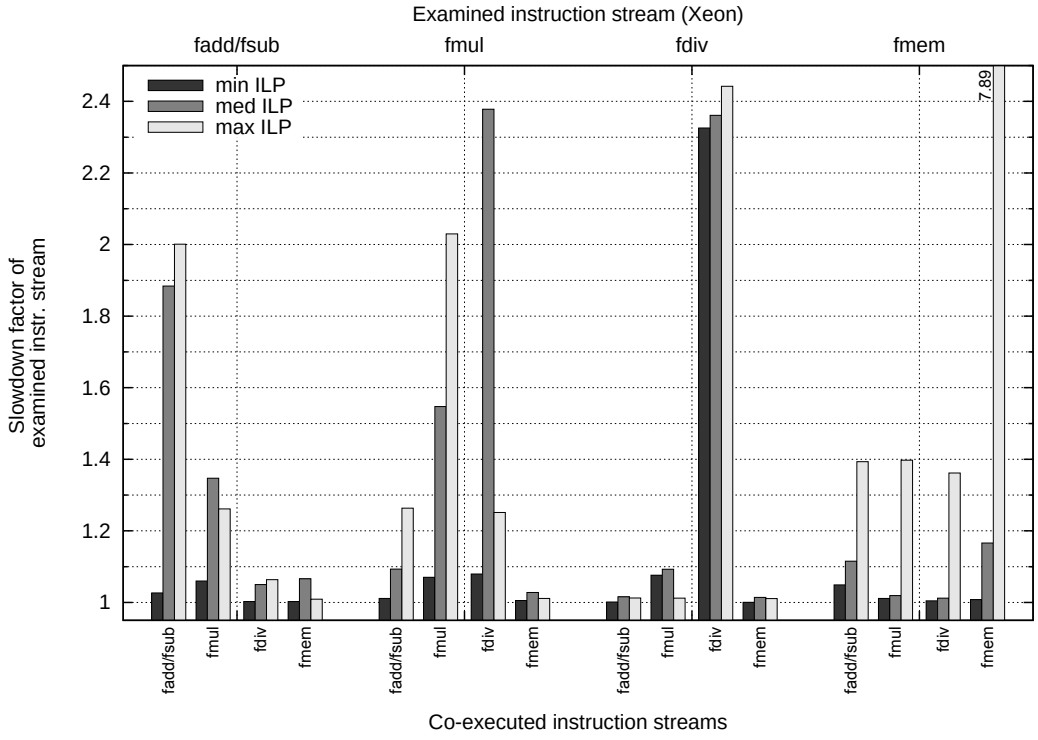


(β) *Nehalem.*

Σχήμα 2.4: *Μέσο CPI συνηθισμένων ροών εντολών για διάφορους συνδυασμούς TLP και ILP.*

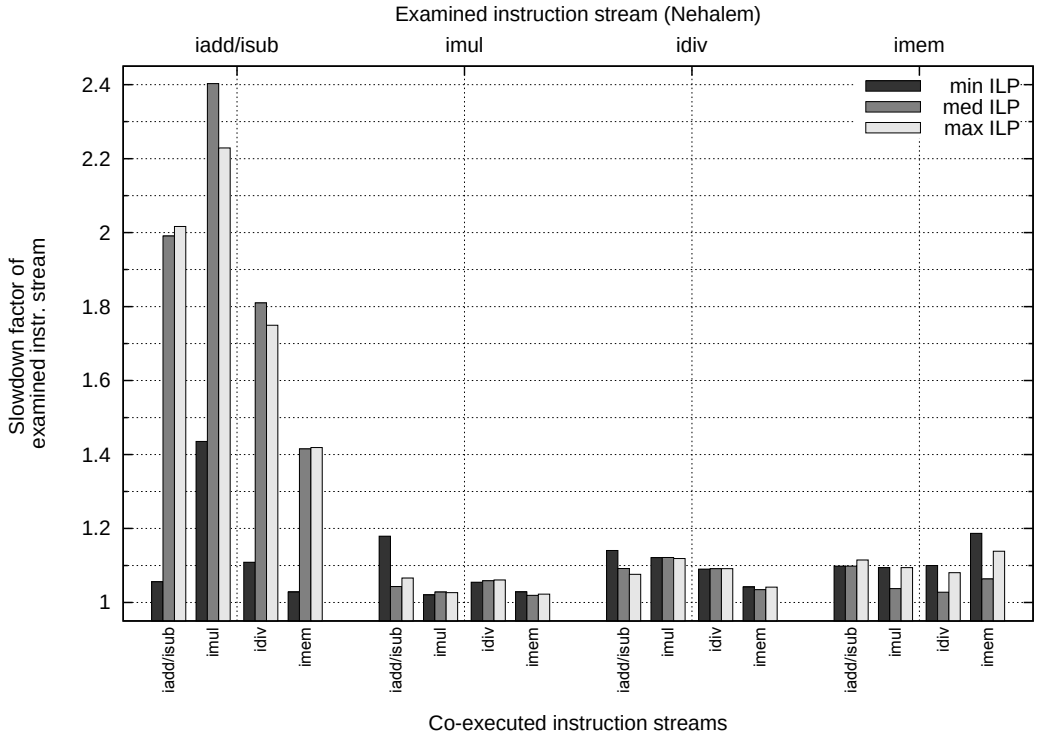


(α) *Εντολές ακεραίων.*

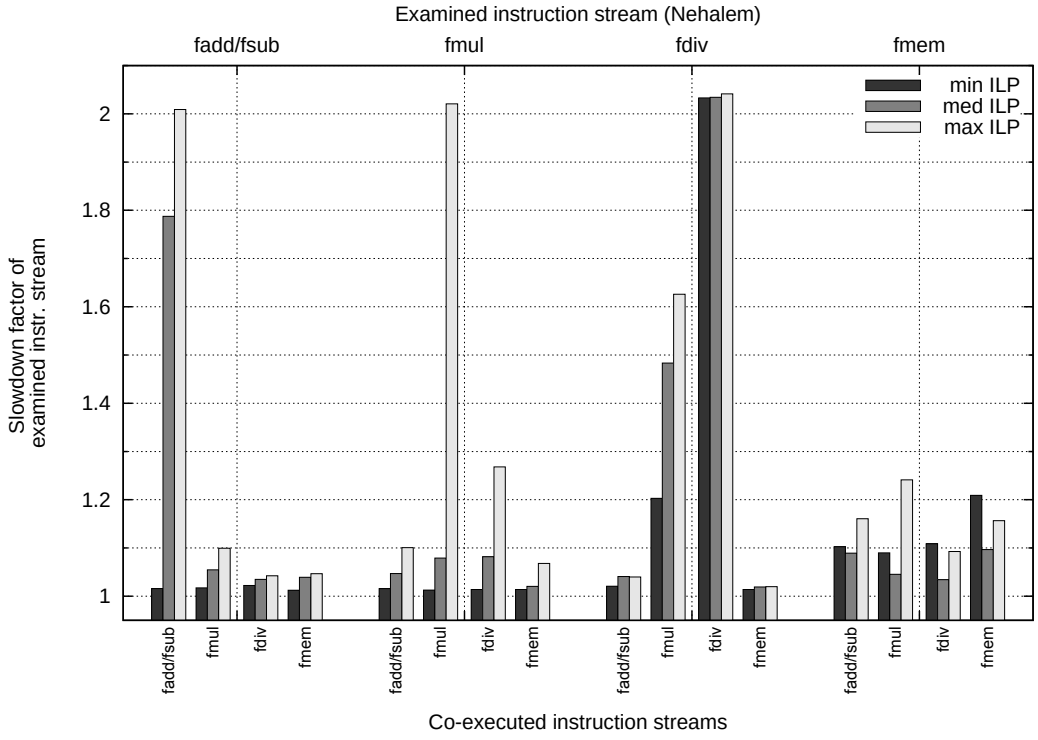


(β) *Εντολές κινητής υποδιαστολής.*

Σχήμα 2.5: *Επιβράδυνση κατά την ταυτόχρονη εκτέλεση διαφόρων ροών εντολών (Xeon).*

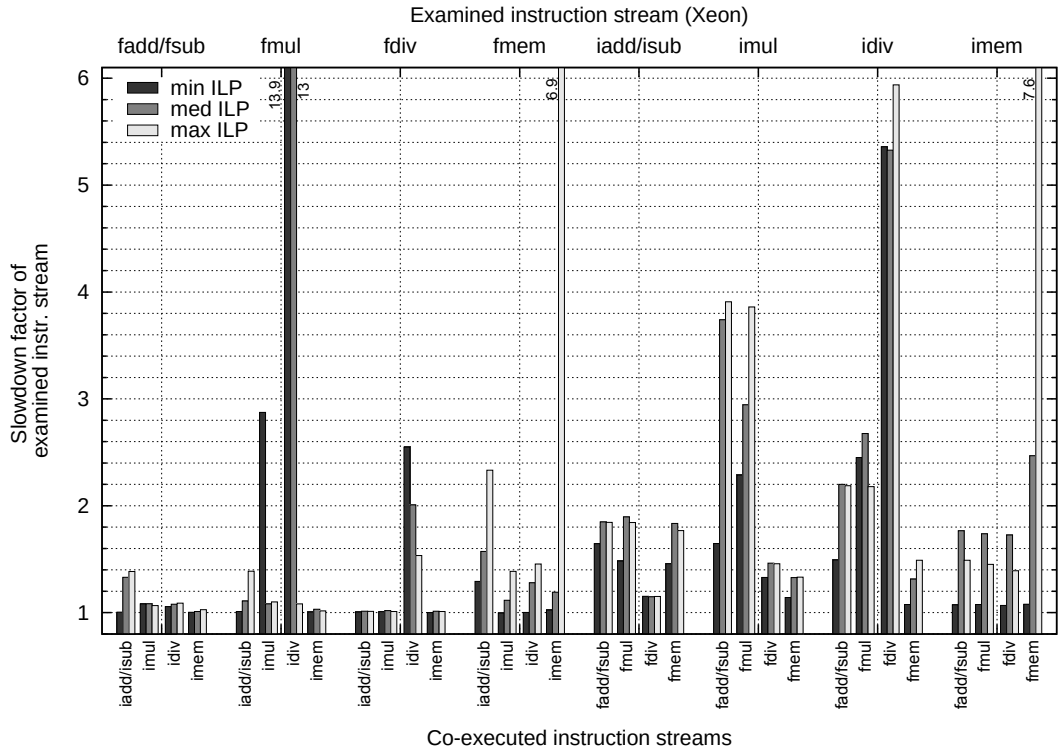


(α) *Εντολές ακεραίων.*

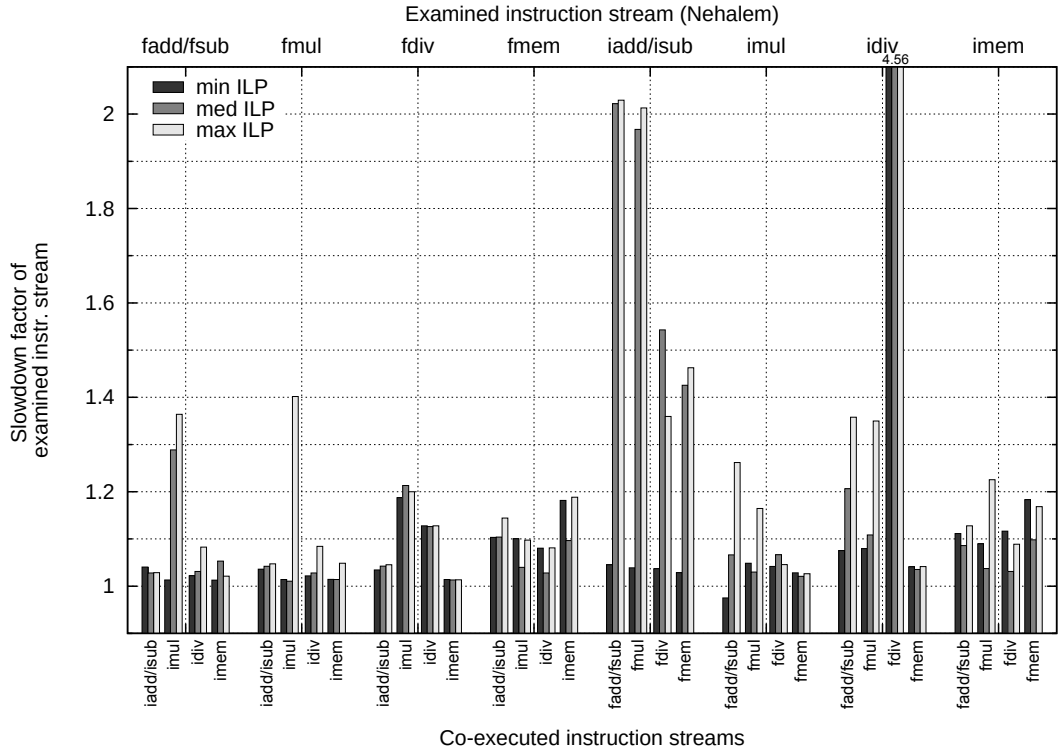


(β) *Εντολές κινητής υποδιαστολής.*

Σχήμα 2.6: *Επιβράδυνση κατά την ταυτόχρονη εκτέλεση διαφόρων ροών εντολών (Nehalem).*



(α) *Xeon.*



(β) *Nehalem.*

Σχήμα 2.7: *Επιβράδυνση κατά την ταυτόχρονη εκτέλεση διαφόρων ζευγαριών ροών εντολών ακεραίων και κινητής υποδιαστολής.*

ΚΕΦΑΛΑΙΟ 3

Πολυνηματικά Σχήματα Εκτέλεσης για Επεξεργαστές SMT

3.1 Εισαγωγή

Αρκετές εφαρμογές χαρακτηρίζονται από έλλειψη εγγενούς παραλληλισμού επιπέδου εντολών ή από συχνή εμφάνιση γεγονότων μεγάλης καθυστέρησης, όπως αστοχίες στην κρυφή μνήμη ή λάθος προβλέψεις διακλάδωσης. Τέτοιες εφαρμογές μπορούν να ευεργετηθούν από το SMT όταν παραλληλοποιούνται σε πολλαπλά νήματα εκτέλεσης. Στην πρώτη περίπτωση, η μεγιστοποίηση της χρησιμοποίησης μπορεί να επιτευχθεί όταν εντολές από πρόσθετα νήματα εφαρμογών (τα οποία μπορούν επίσης να έχουν χαμηλό ILP) χρησιμοποιούν κενές θέσεις στις μονάδες έκδοσης του επεξεργαστή. Με αυτό τον τρόπο, ο παραλληλισμός επιπέδου νημάτων μετατρέπεται ουσιαστικά σε παραλληλισμό επιπέδου εντολών. Στη δεύτερη περίπτωση, κάθε φορά που ένα νήμα καθυστερείται σε κάποιο γεγονός μεγάλης διάρκειας, εντολές από άλλα νήματα μπορούν να εκδοθούν άμεσα και να χρησιμοποιήσουν τη σωλήνωση, που διαφορετικά θα παρέμενε ανεκμετάλλευτη.

Δύο βασικές εναλλακτικές έχουν διερευνηθεί στη σχετική βιβλιογραφία όσον αφορά τον τρόπο χρήσης των πολλαπλών νημάτων υλικού ενός επεξεργαστή SMT με σκοπό την επιτάχυνση μιας σειριακής εφαρμογής: η *παραλληλοποίηση επιπέδου νημάτων* (thread-level parallelization – TLP) και ο *βοηθητική νημάτωση* (helper threading), με πιο χαρακτηριστική μορφή της

την *υποθετική προ-εκτέλεση* (speculative pre-computation – SPR) την οποία και εξετάζουμε λεπτομερώς στα πλαίσια της διατριβής.

Στην παραλληλοποίηση επιπέδου νημάτων οι σειριακοί κώδικες παραλληλοποιούνται ώστε οι υπολογισμοί να κατανέμονται ισομερώς σε διαφορετικά νήματα, όπως γίνεται στις συνήθεις μεθόδους παραλληλοποίησης στους πολυ-επεξεργαστές κοινής μνήμης. Αυτό το είδος παραλληλοποίησης είναι η προφανής επιλογή για κώδικες που βασίζονται σε πίνακες και δομές βρόχων με μεγάλη κανονικότητα, όπως είναι για παράδειγμα οι περισσότερες επιστημονικές εφαρμογές. Η παραλληλοποίηση σε αυτούς τους κώδικες δίνει νήματα που είναι συμμετρικά ως προς τη φύση και τη ποσότητα των υπολογισμών που εκτελούν.

Στην βοηθητική νημάτωση η εκτέλεση μιας εφαρμογής υποστηρίζεται από επιπρόσθετα *βοηθητικά νήματα* (helper threads) τα οποία επιτελούν λειτουργίες που διευκολύνουν, συνήθως *έμμεσα*, το κύριο νήμα εργασίας στο να εκτελεστεί αποδοτικότερα. Η βοηθητική νημάτωση στοχεύει κυρίως σε εφαρμογές που είναι δύσκολα έως και καθόλου παραλληλοποιήσιμες, ή που δεν μπορούν να κερδίσουν πολλά από την παραδοσιακή παραλληλοποίηση σε πολλαπλά νήματα. Παράδειγμα των λειτουργιών που επιτελούν τα βοηθητικά νήματα είναι η *προφόρτωση δεδομένων* (data prefetching) στην κρυφή μνήμη τα οποία θα χρησιμοποιηθούν από το κύριο νήμα στο άμεσο μέλλον [Collins 01, Luk 01], ή η προκαταβολική εκτέλεση εντολών διακλάδωσης του κύριου νήματος [Roth 01, Zilles 01]. Στην πρώτη περίπτωση κρύβεται η καθυστέρηση πρόσβασης στη μνήμη για το κύριο νήμα, ενώ στη δεύτερη περίπτωση εκπαιδεύεται ο προβλέπτης διακλαδώσεων ώστε να βελτιώνεται η ακρίβειά του και να ελαχιστοποιούνται οι χαμένοι κύκλοι λόγω εσφαλμένων προβλέψεων. Λιγότερο συνηθισμένα στη βιβλιογραφία είναι σχήματα όπου τα βοηθητικά νήματα δεν προσπαθούν απλά μέσω θετικών "παρενεργειών" να αποφορτίσουν το κύριο νήμα, αλλά αναλαμβάνουν ουσιαστικά κάποιο μέρος των υπολογισμών του.

Στα περισσότερα σενάρια χρήσης της βοηθητικής νημάτωσης τα βοηθητικά νήματα λειτουργούν *υποθετικά* (speculatively), υπό την έννοια ότι οι εργασίες που επιτελούν δεν είναι σίγουρο αν θα φανούν τελικά χρήσιμες στο κύριο νήμα. Για παράδειγμα, ένα βοηθητικό νήμα ενδέχεται να προφορτώσει δεδομένα πάνω σε ένα μονοπάτι ελέγχου στο οποίο τελικά δε θα βρεθεί το κύριο νήμα. Σε κάθε περίπτωση, ωστόσο, τα νήματα λειτουργούν με τέτοιο τρόπο ώστε να μην επηρεάζουν την αρχιτεκτονική κατάσταση και την ορθότητα εκτέλεσης του κύριου νήματος. Ακόμα και όταν αναλαμβάνουν να εκτελέσουν με υποθετικό τρόπο μέρος των υπολογισμών του κύριου νήματος, υπάρχουν κατάλληλοι μηχανισμοί ώστε όταν διαταράσσεται η ορθότητα οι υπολογισμοί των βοηθητικών νημάτων να αναιρούνται. Τέλος, βασική προϋπόθεση για την αποτελεσματικότητα των βοηθητικών σχημάτων εκτέλεσης είναι η διακριτικότητα στη λειτουργία των βοηθητικών νημάτων, ώστε να μην επιβαρύνουν αρνητικά το κύριο νήμα ακόμα και όταν οι ενέργειές τους δεν είναι στην σωστή κατεύθυνση.

Τόσο στο σχήμα TLP όσο και στο σχήμα SPR, η απεικόνιση των αρχικών λειτουργιών του προγράμματος σε πολλαπλά νήματα και η περαιτέρω ενορχήστρωσή τους προκειμένου να εκτελεστούν αποδοτικά σε έναν επεξεργαστή SMT δεν είναι σαφής διαδικασία. Όπως έχουμε ήδη δει, οι πιο πολλοί πόροι του επεξεργαστή (κρυφές μνήμες, μονάδες εκτέλεσης, ουρές εντολών, κ.λπ.) είναι είτε δυναμικά διαμοιραζόμενοι είτε στατικά διαχωρισμένοι ανάμεσα στους λογικούς επεξεργαστές, και μόνο μερικές δομές υπάρχουν σε πολλαπλά αντίγραφα. Συνεπώς, είναι δυνατόν να εμφανιστούν συγκρούσεις μεταξύ νημάτων κατά τη χρήση των πόρων αυτών, όπως για παράδειγμα όταν κάποιο νήμα εκτοπίζει από την κρυφή μνήμη δεδομένα ενός άλλου νήματος, ή όταν τα νήματα επιχειρούν να χρησιμοποιήσουν τις ίδιες μονάδες εκτέλεσης την ίδια χρονική στιγμή. Επιπλέον, το κέρδος του πολυνηματισμού στις αρχιτεκτονικές SMT εξαρτάται κατά μεγάλο μέρος από την εκάστοτε εφαρμογή και το βαθμό στον οποίον αυτή έχει βελτιστοποιηθεί [Mitchell 99]. Για παράδειγμα, μια υψηλά βελτιστοποιημένη έκδοση του πολλαπλασιασμού πινάκων (π.χ. με ξεδίπλωμα βρόχων και επεξεργασία κατά μπλοκ) είναι συνήθως προσαρμοσμένη να εκμεταλλεύεται πλήρως τους καταχωρητές, τα επίπεδα κρυφής μνήμης, και τις πολλαπλές μονάδες ενός υπερβαθμωτού επεξεργαστή. Έτσι, αν προσθέσουμε επιπλέον νήματα στον ίδιο φυσικό επεξεργαστή μπορεί να βλάψουμε την ήδη καλή χρησιμοποίηση πόρων της εφαρμογής, και συνεπώς και την απόδοσή της. Από την άλλη, μια μη-βελτιστοποιημένη έκδοση του πολλαπλασιασμού πινάκων δεν χαρακτηρίζεται από αποδοτική χρησιμοποίηση της ιεραρχίας μνήμης ή των μονάδων εκτέλεσης, και έτσι θα μπορούσε να επωφεληθεί από κάποιο σχήμα παραλληλοποίησης στο SMT.

Όσον αφορά την απόδοση μεμονωμένων εφαρμογών σε αρχιτεκτονικές SMT, οι περισσότερες προηγούμενες εργασίες έχουν επιδείξει αποτελέσματα σε θεωρητικά μοντέλα υλοποιημένα σε προσομοιωτή [Mitchell 99, Collins 01, Luk 01]. Δουλειές πάνω σε πραγματικούς επεξεργαστές SMT δεν ανέφεραν ικανοποιητικά μεγέθη επιτάχυνσης (speedup) για εφαρμογές που παραλληλοποιούνταν με βάση το TLP ή το SPR μοντέλο [Kim 04a, Tuck 03, Wang 04, Curtis-Maury 05].

Στα πλαίσια αυτής της διατριβής, εξετάζουμε τη δυνατότητα βελτίωσης της απόδοσης για μια σειρά από διαφορετικές εφαρμογές όταν εκτελούνται σε Hyper-threaded επεξεργαστές χρησιμοποιώντας τα δύο προαναφερθέντα μοντέλα εκτέλεσης, TLP και SPR. Οι εφαρμογές που εξετάζουμε περιλαμβάνουν προγράμματα αριθμητικών υπολογισμών βασισμένα στη χρήση πινάκων, τόσο με ομαλά όσο και με ακανόνιστα ή τυχαία μοτίβα προσπέλασης μνήμης, καθώς και κώδικες που συναντώνται σε γραφοθεωρητικές εφαρμογές ή εφαρμογές βάσεων δεδομένων, και οι οποίοι βασίζονται κυρίως στη χρήση δεικτών και χαρακτηρίζονται από ακανόνιστες προσπελάσεις. Παρόλο που κάποια τεχνική παραλληλοποίησης μπορεί να είναι πιο κατάλληλη από κάποια άλλη για ένα συγκεκριμένο είδος εφαρμογών, όπως συζητήσαμε και πιο πάνω (π.χ., TLP για "κανονικούς" κώδικες, SPR για πιο ακανόνιστους), στα πλαίσια της διατριβής αυτής διερευνούμε τη δυναμική και των δύο επιλογών για όλους τους κώδικες που εξετάζουμε.

Υλοποιήσαμε τα δύο βασικά σχήματα παραλληλοποίησης ως εξής: Για το TLP, διαμοιράσαμε το υπολογιστικό φορτίο κάθε εφαρμογής σε δύο νήματα εκτέλεσης, διαμερίζοντας στατικά τον χώρο επαναλήψεων του κώδικα της εφαρμογής. Στο SPR, εκτελέσαμε ένα κύριο νήμα εργασίας μαζί με ένα βοηθητικό μνήμα, το οποίο υποθετικά προεκτελεί αναφορές που ενδέχεται να προκαλέσουν αστοχίες στην κρυφή μνήμη L2, και φορτώνει τα αντίστοιχα δεδομένα στην L2. Ο συγχρονισμός των δύο νημάτων είναι βασικός παράγοντας για την αποτελεσματικότητα αυτού του μοντέλου εκτέλεσης. Αφενός, πρέπει να εξασφαλίζει ακρίβεια και σωστό χρονισμό στις προφορτώσεις, ελέγχοντας κατάλληλα την απόσταση κατά την οποία το βοηθητικό νήμα προτρέχει του κύριου νήματος και προφορτώνει δεδομένα. Αφετέρου, πρέπει να εισάγει τη μικρότερη δυνατή επιβάρυνση στο κύριο νήμα εκτέλεσης.

Αξιολογήσαμε και ερμηνεύσαμε την απόδοση με δυο τρόπους: Αρχικά, συλλέξαμε αποτελέσματα μετρικών απόδοσης από τους *μετρητές απόδοσης* (performance counters) που διαθέτει ο επεξεργαστής. Σε δεύτερη φάση, αναλύσαμε το δυναμικό μίγμα εντολών των νημάτων κάθε εφαρμογής και εντοπίσαμε τα επικρατέστερα είδη εντολών. Έχοντας εξερευνήσει τον τρόπο με τον οποίο αλληλεπιδρούν μεταξύ τους σε έναν Hyper-threaded επεξεργαστή συνθετικές ροές εντολών αποτελούμενες από αυτά τα είδη, ήμασταν σε θέση να δώσουμε περισσότερες ερμηνείες πάνω στην παρατηρούμενη απόδοση.

3.2 Σχετικές Εργασίες

Το SMT θεωρείται μια πολλά υποσχόμενη τεχνική επειδή συνδυάζει τη δυνατότητα των σύγχρονων υπερβαθμωτών επεξεργαστών να εκδίδουν πολλαπλές εντολές σε έναν κύκλο, με τη δυνατότητα των πολυνηματικών επεξεργαστών να κρύβουν την καθυστέρηση από γεγονότα όπως αστοχίες στην κρυφή μνήμη επικαλύπτοντάς τα με εντολές από άλλα νήματα. Όλα τα νήματα υλικού είναι ενεργά ταυτόχρονα και ανταγωνίζονται σε κάθε κύκλο για πρόσβαση σε κοινούς πόρους. Αυτή η δυναμική διαμοίραση των λειτουργικών μονάδων επιτρέπει στο SMT μεγαλύτερο *ρυθμό ολοκλήρωσης* (throughput) εντολών *συνολικά*, αντιμετωπίζοντας την υποχρησιμοποίηση πόρων λόγω χαμηλού ILP και γεγονότων μεγάλων καθυστερήσεων. Όμως, αυτή η ευελιξία του SMT έρχεται με κάποιο κόστος. Όταν πολλαπλά νήματα είναι ταυτόχρονα ενεργά, ο στατικός διαχωρισμός κάποιων δομών όπως π.χ. οι ουρές μικροεντολών και αναγνώσεων-εγγραφών (load-store queues) ή ο απομονωτής αναδιάταξης εντολών (re-order buffer), είναι δυνατόν να επηρεάσει κώδικες που έχουν υψηλό ρυθμό ολοκλήρωσης εντολών. Ο στατικός διαχωρισμός περιορίζει σημαντικά την απόδοση όταν πολλαπλά νήματα εκτελούν ταυτόχρονα όμοιες ροές εντολών, ενώ έχει μικρότερες επιπτώσεις όταν οι ροές εντολών είναι διαφορετικού τύπου [Tuck 03].

Η προφόρτωση δεδομένων είναι μία από τις πιο δημοφιλείς τεχνικές για την απόκρυψη της όλο και αυξανόμενης καθυστέρησης πρόσβασης στη μνήμη. Σε αντίθεση με τον πολυνηματισμό, όπου οι καθυστερήσεις ενός νήματος μπορούν να επικαλυφθούν με εντολές από άλλα νήματα, η προφόρτωση συνήθως στοχεύει στην ανοχή της καθυστέρησης στα πλαίσια λειτουργίας ενός και μόνο νήματος εφαρμογής. Με τις εντολές προφόρτωσης μπορούμε να ζητήσουμε νωρίτερα τη μεταφορά δεδομένων από την κύρια μνήμη σε κάποιο επίπεδο κρυφής μνήμης, με την ελπίδα ότι θα φτάσουν εκεί αρκούντως νωρίς και πριν ζητηθούν πραγματικά από την εφαρμογή. Εφόσον τα δεδομένα δεν εκτοπιστούν από την κρυφή μνήμη μέχρι να χρησιμοποιηθούν, η καθυστέρηση πρόσβασης στην μνήμη θα μπορεί να έχει κρυφτεί πλήρως.

Μια σειρά από εργασίες πραγματεύονται το ζήτημα της προφόρτωσης δεδομένων σε επίπεδο λογισμικού για τη βελτίωση της απόδοσης μονονηματικών εφαρμογών. Αυτές οι εργασίες έχουν εστιάσει σε κώδικες με ομαλά μοτίβα πρόσβασης στη μνήμη [Mowry 92, Mowry 98] καθώς και σε κώδικες με έντονη χρήση δεικτών και αναδρομικές δομές δεδομένων που έχουν ακανόνιστα ή τυχαία μοτίβα πρόσβασης [Luk 96, Luk 99]. Η προφόρτωση δεδομένων στην κρυφή μνήμη μειώνει την παρατηρούμενη καθυστέρηση των προσπελάσεων μνήμης, φέρνοντας δεδομένα σε κάποιο επίπεδο της κρυφής μνήμης προτού ζητηθούν από τον επεξεργαστή. Όμως, καθώς η ρυθμαπόδοση του επεξεργαστή βελτιώνεται εξαιτίας άλλων τεχνικών που στοχεύουν στην ανοχή της καθυστέρησης της μνήμης, η προφόρτωση έρχεται αντιμέτωπη με μια σειρά από δυσκολίες. Αρχικά, η χρησιμοποίηση του εύρους ζώνης του διαύλου μνήμης αυξάνεται αφού η προφόρτωση αυξάνει την κίνηση από και προς τη μνήμη. Επιπλέον, οι απαιτήσεις για χωρητικότητα στην κρυφή μνήμη αυξάνονται, καθώς όλο και περισσότερα δεδομένα έρχονται στην κρυφή μνήμη για μελλοντική χρήση. Τέλος, η επιβάρυνση εξαιτίας της εισαγωγής στη σωλήνωση των εντολών προφόρτωσης αυτών καθεαυτών μπορεί να είναι σημαντική, ακυρώνοντας τα όποια οφέλη από την προφόρτωση των δεδομένων.

Πολλά σχήματα προφόρτωσης δεδομένων που βασίζονται στον πολυνηματισμό έχουν προταθεί στη βιβλιογραφία. Η βασική ιδέα είναι να χρησιμοποιηθούν ένα ή περισσότερα διαθέσιμα νήματα υλικού ενός πολυνηματικού ή πολυπύρηνου επεξεργαστή για να προεκτελεστούν υποθετικά μελλοντικές αναφορές μνήμης και να προφορτωθούν δεδομένα για χάριν του κύριου –ή αλλιώς, *μη υποθετικού* (non-speculative)– νήματος εκτέλεσης. Όλα αυτά τα σχήματα υποθέτουν τον διαμοιρασμό κάποιου επιπέδου της κρυφής μνήμης από όλα τα νήματα του επεξεργαστή. Με το να χρησιμοποιούνται ξεχωριστά νήματα για να εκτελέσουν ολόκληρη την ακολουθία εντολών μέχρι τον υπολογισμό συγκεκριμένων διευθύνσεων, γίνεται εφικτός ο αποτελεσματικότερος χειρισμός ακανόνιστων μοτίβων πρόσβασης τα οποία δε θα ήταν εύκολο να προφορτωθούν διαφορετικά, είτε από το υλικό (με αυτόματους μηχανισμούς), είτε από το λογισμικό (όπου δε θα αρκούσε η απλή εισαγωγή εντολών προφόρτωσης στο σώμα του κύριου νήματος).

Μια σειρά από σημαντικές εργασίες έχουν προτείνει σχήματα προφόρτωσης που βασίζονται στη χρήση βοηθητικών νημάτων. Οι εργασίες αυτές επικεντρώνονται κυρίως σε αρχιτεκτονικές SMT, ενώ τα βοηθητικά νήματα είτε είναι ελεγχόμενα από το λογισμικό είτε είναι εντελώς αόρατα σε αυτό και τα διαχειρίζεται αποκλειστικά το υλικό με κατάλληλες επεκτάσεις. Ανάμεσα στις σημαντικότερες εργασίες που βασίζονται στο λογισμικό ξεχωρίζουμε το *Speculative Precomputation* (SPR) των Collins και άλλων [Collins 01], το *Software Controlled Pre-Execution* του Luk [Luk 01] και τους αλγορίθμους μεταγλώττισης που προτείνουν οι Kim και Yeung στο [Kim 02]. Από τα πολυνηματικά σχήματα προφόρτωσης με ελεγχόμενα από το υλικό βοηθητικά νήματα, διακρίνουμε το *Speculative Data Driven Multithreading* (DDMT) των Roth και Sohi [Roth 01],τους *Slipstream Processors* των Sundaramoorthy και άλλων [Sundaramoorthy 00], τα *Speculative Slices* των Zilles και Sohi [Zilles 01] και το *Simultaneous Subordinate Microthreading* (SSMT) των Chappell και άλλων [Chappell 99].

Τα σχήματα προεκτέλεσης που εξετάζονται σε αυτές τις εργασίες στοχεύουν κυρίως σε αναφορές μνήμης που έχουν απρόβλεπτα και ακανόνιστα μοτίβα, στηρίζονται στη χρήση δεικτών ή εξαρτώνται από τις τιμές των δεδομένων. Κοινή διαπίστωση των εργασιών αυτών είναι ότι ένα μικρό συνήθως μέρος των στατικών εντολών ανάγνωσης των εξεταζόμενων προγραμμάτων, γνωστές και ως *παραβατικές εντολές ανάγνωσης* (delinquent loads), είναι υπεύθυνο για τη συντριπτική πλειοψηφία των κύκλων καθυστέρησης κατά την πρόσβαση στην κύρια μνήμη. Αξίζει να αναφέρουμε ότι τα σχήματα που βασίζονται στο υλικό για τη διαχείριση των βοηθητικών νημάτων, πέρα από την προφόρτωση δεδομένων για κρίσιμες αναφορές μνήμης, στοχεύουν και στη βελτίωση της ακρίβειας του προβλέπτη διακλάδωσης μέσω της προεκτέλεσης εντολών διακλάδωσης από βοηθητικά νήματα.

Κοινό χαρακτηριστικό των προαναφερθέντων εργασιών, ακόμα και αυτών που βασίζονται στο λογισμικό, είναι ότι υλοποιούν και εξετάζουν τις προτεινόμενες τεχνικές προεκτέλεσης σε περιβάλλοντα προσομοίωσης επεξεργαστών SMT. Αυτό έχει σαν συνέπεια να "κρύβονται" πολλές από τις δυσκολίες που συνεπάγεται η εφαρμογή των τεχνικών αυτών σε πραγματικές αρχιτεκτονικές SMT, λόγω του ότι οι τελευταίες υστερούν σε σχέση με τα θεωρητικά μοντέλα στα εξής: δεν υποστηρίζονται από ειδικό υλικό για τη γρήγορη δημιουργία, διαχείριση και συγχρονισμό των βοηθητικών νημάτων, δε διαθέτουν μεγάλο αριθμό από νήματα υλικού για ταυτόχρονη εκτέλεση πολλαπλών βοηθητικών νημάτων, και τέλος, δε χαρακτηρίζονται από ιδιαίτερα αποδοτική διαχείριση των μοιραζόμενων πόρων ανάμεσα στα νήματα. Σε αυτά τα πλαίσια, το να επιτευχθούν σε πραγματικά συστήματα SMT αντίστοιχες επιταχύνσεις όπως στα θεωρητικά μοντέλα, αποτελεί ιδιαίτερη πρόκληση.

Από τις προσεγγίσεις που έχουν εστιάσει σε σχήματα προεκτέλεσης σε πραγματικούς επεξεργαστές SMT διακρίνουμε αυτές των Kim και άλλων στο [Kim 04a], και Wang και άλλων

στο [Wang 04]. Και οι δύο αξιολογούν τις τεχνικές τους στις πρώτες υλοποιήσεις του Hyper-threading, σε επεξεργαστές Pentium 4 και Xeon. Η πρώτη χρησιμοποιεί εφαρμογές με χαρακτηριστικό την έντονη χρήση μνήμης και τα ακανόνιστα μοτίβα προσπέλασης, από τις σουίτες μετροπρογραμμάτων SPEC CPU2000 και Olden. Η δεύτερη χρησιμοποιεί επιστημονικούς κώδικες με πιο κανονικές και προβλέψιμες προσβάσεις, επιλεγμένους από τη σουίτα NAS Parallel Benchmarks ή υλοποιημένους εξ' αρχής. Και στις δύο εργασίες αναφέρονται επιταχύνσεις που σε μερικές περιπτώσεις φτάνουν και το 20%, όμως η γενική εικόνα βρίσκεται αρκετά χαμηλότερα, με επιταχύνσεις γύρω στο 5% κατά μέσο όρο. Για την αποκόμιση ουσιαστικών κερδών από την προεκτέλεση, οι συγγραφείς επισημαίνουν την ανάγκη για συνετή και επιλεκτική εκτέλεση των βοηθητικών νημάτων προκειμένου να ελαχιστοποιούνται οι συγκρούσεις με το κύριο νήμα εκτέλεσης, καθώς και για υποστήριξη από αποδοτικούς μηχανισμούς συγχρονισμού.

Τα βοηθητικά νήματα προφόρτωσης είναι ένας από τους εναλλακτικούς τρόπους να χρησιμοποιήσει κανείς τα πρόσθετα περιβάλλοντα εκτέλεσης ενός επεξεργαστή SMT, προωθώντας ταυτόχρονα την ασυμμετρία στο προφίλ εκτέλεσης των νημάτων. Στη βιβλιογραφία έχουν προταθεί διάφοροι ενδιαφέροντες τρόποι αξιοποίησης του δεύτερου περιβάλλοντος εκτέλεσης ενός Hyper-threaded επεξεργαστή. Οι Gummaraju και Rosenblum στο [Gummaraju 05] εξετάζουν την απεικόνιση ροϊκών προγραμμάτων (stream programs) σε επεξεργαστές με Hyper-threading. Το μοντέλο του ροϊκού προγραμματισμού αποσυνδέει τις λειτουργίες μνήμης από τους υπόλοιπους υπολογισμούς σε ένα πρόγραμμα, και ως εκ τούτου φαίνεται να είναι καλός υποψήφιος για εκτέλεση σε μια αρχιτεκτονική SMT. Στη συγκεκριμένη εργασία προτείνεται η χρήση ξεχωριστών ουρών εργασίας για λειτουργίες μνήμης και υπολογισμούς, οι οποίες ανατίθενται σε διαφορετικούς λογικούς επεξεργαστές του ίδιου φυσικού πακέτου. Η απόδοση που επιτυγχάνεται με αυτό το μοντέλο είναι συγκρίσιμη με τα συμβατικά μοντέλα ακολουθιακού προγραμματισμού, ωστόσο αποτελεί ερώτημα αν θα μπορούσε να ξεπεράσει και τα κλασικά σχήματα παραλληλοποίησης επιπέδου νημάτων.

Στην ίδια λογική κινείται και η μέθοδος που παρουσιάζουμε στο [Goumas 09a] για βελτιστοποίηση της απόδοσης παράλληλων εφαρμογών ανταλλαγής μηνυμάτων (MPI) οι οποίες εκτελούνται σε υπολογιστικές συστοιχίες από επεξεργαστές SMT (Hyper-threaded). Προτείνουμε ένα σχήμα επικάλυψης υπολογισμών και επικοινωνίας μέσω του οποίου αποφορτίζουμε το κύριο νήμα από τις λειτουργίες επικοινωνίας, αναθέτοντάς τις στο δεύτερο νήμα κάθε επεξεργαστή SMT. Αντίστοιχες τεχνικές επικάλυψης έχουν προταθεί κατά το παρελθόν, με τη διαφορά ότι η επικοινωνία απεικονιζόταν στο δίκτυο διασύνδεσης το οποίο ήταν εξοπλισμένο με κατάλληλο υλικό, ειδικό για να αποφορτίζει τον επεξεργαστή από τις πρωτογενείς λειτουργίες επικοινωνίας. Στην περίπτωσή μας δεν υπάρχει τέτοιου είδους υποστήριξη, για αυτό και εξετάζουμε την

απλούστερη και αποδοτικότερη σε κόστος λύση του να αναθέσουμε την επικοινωνία στον ομότιμο λογικό επεξεργαστή. Και σε αυτήν την περίπτωση, το σκεπτικό είναι ότι τα νήματα υπολογισμών και επικοινωνίας δεν αναμένεται να οδηγήσουν σε ιδιαίτερες συγκρούσεις καθώς χρησιμοποιούν διαφορετικούς πόρους του επεξεργαστή. Η προτεινόμενη μέθοδος είναι σε θέση να προσφέρει αξιόλογη βελτίωση που σε μερικές περιπτώσεις ξεπερνά και το 20% της απόδοσης του αρχικού, μη-επικαλυπτόμενου σχήματος.

Όσον αφορά τα σχήματα παραλληλοποίησης σε επίπεδο νημάτων, οι περισσότερες εργασίες πάνω σε προσομοιούμενα μοντέλα SMT έχουν δείξει περιορισμένο κέρδος στην απόδοση των εφαρμογών, κυρίως λόγω της ομοιογένειας των νημάτων [Lo 97a, Lo 97b, Tullsen 95]. Αυτό δεν ισχύει στον ίδιο βαθμό για τα πολυπρογραμματιζόμενα φορτία εργασίας, όπου η απαιτήσεις για πόρους είναι λιγότερο "συμμετρικές" και επομένως οι συγκρούσεις είναι λιγότερες. Οι διαπιστώσεις των παραπάνω εργασιών επαληθεύονται και σε πειράματα με πραγματικές αρχιτεκτονικές SMT, και συγκεκριμένα με Hyper-threaded επεξεργαστές [Tuck 03, Bulpin 04, Wang 04, Curtis-Maury 05]. Οι επιταχύνσεις που επιτυγχάνονται σε αυτές τις εργασίες πάνω σε παράλληλους κώδικες, προερχόμενους κυρίως από σουίτες παράλληλων εφαρμογών όπως τα NAS Parallel Benchmarks και η SPLASH2, κυμαίνονται κατά μέσο όρο μεταξύ 1.1 και 1.3. Οι περισσότερες από αυτές τις εργασίες χρησιμοποιούν το OpenMP σαν το βασικό προγραμματιστικό μοντέλο για παραλληλοποίηση σε επίπεδο νημάτων. Στα πλαίσια της διατριβής χρησιμοποιούμε αντίθετα *ρητό πολυνηματισμό* (explicit multithreading), σαν μια προσπάθεια να ελαχιστοποιήσουμε το κόστος διαχείρισης των νημάτων που εισάγουν οι μηχανισμοί χρόνου εκτέλεσης των υλοποιήσεων του OpenMP, και να απομονώσουμε καλύτερα τη δυναμική κάθε εφαρμογής να επωφεληθεί από το SMT.

3.3 Ζητήματα Απόδοσης και Υλοποίησης

Από την οπτική γωνία της χρησιμοποίησης επεξεργαστικών πόρων τα νήματα που πραγματοποιούν προφόρτωση στο σχήμα SPR απαιτούν συνήθως λιγότερους πόρους από το κύριο νήμα εκτέλεσης, καθώς δεν πραγματοποιούν "βαρείς" υπολογισμούς που θα μπορούσαν να επηρεάσουν την κατάσταση και τα δεδομένα του προγράμματος. Τυπικά, ο κώδικας των νημάτων προφόρτωσης περιλαμβάνει αλληλουχίες από εξαρτώμενες εντολές ανάγνωσης, αριθμητικής ακεραίων και διακλάδωσης, που οδηγούν στον υπολογισμό των διευθύνσεων των παραβατικών εντολών ανάγνωσης. Το μεγάλο μέρος των αρχικών εντολών του προγράμματος που βρίσκονται έξω από αυτό το "κρίσιμο μονοπάτι" έχουν εξαιρεθεί. Υπό αυτήν την έννοια, οι κώδικες των νημάτων προφόρτωσης έχουν χαμηλά επίπεδα ILP, ενώ ο προσανατολισμός τους στη χρήση ορισμένων μόνο εντολών αναμένεται να οδηγήσει σε περισσότερη ετερογένεια ανάμεσα στα διαφορετικά νήματα και συνεπώς σε λιγότερες συγκρούσεις λόγω διεκδίκησης κοινών πόρων.

Από την άλλη πλευρά, το σχήμα TLP δίνει περισσότερες ευκαιρίες στα προγράμματα για καλύτερη αξιοποίηση ολόκληρου του εύρους υπερβαθμωτής επεξεργασίας. Από τη στιγμή όμως που αρκετοί από τους μοιραζόμενους πόρους δε βρίσκονται σε πολλαπλά αντίγραφα προκύπτουν συχνά συγκρούσεις οι οποίες οδηγούν σε απώλειες στην απόδοση. Υπάρχουν περιπτώσεις όμως όπου τα νήματα μιας εφαρμογής μπορούν να επωφεληθούν από την χρήση κοινών πόρων, και ειδικά από την κρυφή μνήμη. Κάτι τέτοιο συμβαίνει για παράδειγμα όταν τα νήματα επεξεργάζονται γειτονικά δεδομένα. Αυτό συχνά οδηγεί στην φόρτωση γραμμών κρυφής μνήμης οι οποίες, συμπτωματικά ή μη, μπορούν να φανούν χρήσιμες και σε άλλα νήματα (π.χ. [Chen 07]).

Από άποψη υλοποίησης, οι σειριακοί κώδικες μπορούν να μετατραπούν σε πολυνηματικούς με βάση το TLP μοντέλο τις πιο πολλές φορές με άμεσο τρόπο, με την προϋπόθεση ότι έχουν επαρκή εγγενή παραλληλισμό. Η μορφή παραλληλοποίησης που συναντάται πιο συχνά είναι η παραλληλοποίηση σε επίπεδο βρόχων (loop-level parallelization). Σε αυτήν ο χώρος επαναλήψεων ενός βρόχου ή μιας εμφώλευσης βρόχων διαμερίζεται στατικά ή δυναμικά, σε μικρά ή μεγάλα κομμάτια, δίνοντας αντίστοιχα *αδρομερή* (coarse-grain) ή *λεπτομερή* (fine-grain) παραλληλισμό. Σε όλες τις περιπτώσεις η διάσπαση που γίνεται στους αρχικούς υπολογισμούς είναι τέτοια ώστε σε κάθε φάση τα παράλληλα νήματα να πραγματοποιούν ίδιους υπολογισμούς πάνω σε διαφορετικά δεδομένα (data-parallel decomposition). Λόγω της σχετικής απλότητας του σχήματος TLP, αλλά και την κατά περίπτωση εφαρμογή του στα προγράμματα που εξετάσαμε, δεν υπάρχουν γενικά ζητήματα τα οποία θα μπορούσαμε να θίξουμε όσον αφορά την υλοποίησή του. Ειδικές λεπτομέρειες γύρω από τη διαδικασία παραλληλοποίησης θα παρουσιάσουμε για κάθε εφαρμογή ξεχωριστά σε επόμενη ενότητα.

Από την άλλη πλευρά, η υλοποίηση των σχημάτων SPR δεν είναι απλή ούτε εξ' αρχής σαφής διαδικασία. Οι εφαρμογές πρέπει να υπόκεινται σε λεπτομερή ρύθμιση προκειμένου οι προφορτώσεις των βοηθητικών νημάτων να είναι ακριβείς και ταυτόχρονα να υλοποιείται αποδοτικά ο συγχρονισμός τους και η χρησιμοποίηση κοινών πόρων. Η συνύπαρξη και η ενορχήστρωση των βοηθητικών νημάτων προφόρτωσης πρέπει να εισάγει τον μικρότερο δυνατό "θόρυβο" και συνάμα να συνεισφέρει θετικά στην πρόοδο του κύριου νήματος εκτέλεσης. Στις ενότητες που ακολουθούν, περιγράφουμε διεξοδικά τα βασικότερα ζητήματα γύρω από την υλοποίηση του σχήματος SPR.

3.4 Υλοποίηση του Σχήματος SPR

Τα περισσότερα σχήματα βοηθητικής νημάτωσης και υποθετικής προεκτέλεσης που έχουν προταθεί στη βιβλιογραφία έχουν βασιστεί σε αποδοτικούς μηχανισμούς για τη δημιουργία και διαχείριση των βοηθητικών νημάτων. Τα βοηθητικά νήματα δημιουργούνται από το κύριο νήμα ή και από άλλα βοηθητικά νήματα, πολλαπλές φορές κατά τη διάρκεια της εκτέλεσης, και

σε σημεία του προγράμματος τα οποία ορίζονται χειροκίνητα από τον χρήστη ή αυτόματα από κάποιον μεταγλωττιστή. Το φορτίο εργασίας τους είναι συνήθως μικρό ενώ τις περισσότερες φορές είναι "μιας χρήσης", με την έννοια ότι αφού προφορτώσουν τα δεδομένα για μια παραβατική εντολή ανάγνωσης καταστρέφονται. Ακόμα και για τα επόμενα στιγμιότυπα της ίδιας αναφοράς πρέπει να δημιουργηθούν ξανά νέα νήματα.

Το κοινό χαρακτηριστικό αυτών των μοντέλων εκτέλεσης είναι ότι υποθέτουν εξιδανικευμένο υλικό για τους σκοπούς της βοηθητικής νημάτωσης, όπως για παράδειγμα πολλαπλούς λογικούς επεξεργαστές για την ένα-προς-ένα απεικόνιση πολλαπλών βοηθητικών νημάτων, ειδικούς μηχανισμούς υλικού που επιτρέπουν τη γρήγορη δημιουργία και εκκίνηση νέων νημάτων (π.χ. στιγμιαία αντιγραφή των καταχωρητών του νήματος-γονέα, γρήγορη εύρεση διαθέσιμου επεξεργαστή και δρομολόγηση του νήματος), ή ειδικό υλικό για τον αποδοτικό συγχρονισμό τους.

Τόσο οι Hyper-threaded επεξεργαστές όμως, όσο και οι σύγχρονες πολυπύρηνες και πολυνηματικές αρχιτεκτονικές, δεν υποστηρίζουν τις δυνατότητες αυτές στο υλικό. Για παράδειγμα, η δημιουργία ενός νήματος σε ένα πραγματικό σύστημα και η απεικόνισή του σε κάποιον επεξεργαστή είναι λειτουργίες που περνάνε μέσα από το λειτουργικό σύστημα. Σε ένα τέτοιο περιβάλλον η συνεχής δημιουργία νέων νημάτων για τους σκοπούς της προφόρτωσης θα ήταν απαγορευτική, αφού το κόστος δημιουργίας, δρομολόγησης και καταστροφής τους θα ήταν πολλαπλάσιο σε σχέση με το μέγεθος της εργασίας την οποία θα επιτελούν. Αναγκαστικά λοιπόν θα πρέπει να καταφύγουμε σε τεχνικές λογισμικού για τη διαχείριση των βοηθητικών νημάτων τις οποίες όμως θα πρέπει να προσαρμόσουμε ώστε να ανταποκρίνονται με τον καλύτερο τρόπο στους στόχους των εξιδανικευμένων θεωρητικών μοντέλων.

Στα πλαίσια αυτής της δουλειάς δημιουργούμε στην αρχή της εκτέλεσης του κύριου νήματος ένα δεύτερο βοηθητικό νήμα το οποίο απεικονίζουμε μόνιμα στον ομότιμο λογικό επεξεργαστή και διατηρούμε ενεργό καθόλη τη διάρκεια εκτέλεσης της εφαρμογής. Κάθε φορά που το βοηθητικό νήμα παραμένει άεργο περιμένοντας να ειδοποιηθεί για το επόμενο κομμάτι εργασίας διακόπτουμε προσωρινά την εκτέλεσή του αποτρέποντάς το από το να καταναλώνει άσκοπα πόρους. Συρρικνώνοντας με αυτό τον τρόπο το βοηθητικό νήμα ανά περιόδους καταφέρνουμε να προσεγγίσουμε ένα μοντέλο συνεχούς δημιουργίας νημάτων, αποφεύγοντας όμως το κόστος δημιουργίας, δρομολόγησης και καταστροφής.

Συνοπτικά, η υλοποίηση του SPR σχήματος περιλαμβάνει τα ακόλουθα βήματα:

1. *εντοπισμός παραβατικών εντολών ανάγνωσης*

2. *παραγωγή κώδικα για τα βοηθητικά νήματα προφόρτωσης*

3. *εισαγωγή σημείων συγχρονισμού ανάμεσα στο βοηθητικό και το κύριο νήμα*

4. *περαιτέρω βελτιστοποίηση του τελικού κώδικα*

Υπάρχουν δύο βασικά ζητήματα που πρέπει να ληφθούν υπόψη προκειμένου το SPR σχήμα να είναι αποδοτικό. Πρώτον, τα βοηθητικά νήματα θα πρέπει να είναι ακριβή και έγκαιρα στις προφορτώσεις (τα σωστά δεδομένα στη σωστή στιγμή), διατηρώντας πάντα μια ελεγχόμενη απόσταση από τα βασικά νήματα εργασίας. Δεύτερον, εφόσον οι περισσότεροι πόροι είναι διαμοιραζόμενοι, τα βοηθητικά νήματα θα πρέπει να εισάγουν τον ελάχιστο δυνατό "θόρυβο" στην εκτέλεση των βασικών νημάτων. Στις επόμενες ενότητες θα παρουσιάσουμε αναλυτικά τα βήματα στην υλοποίηση του SPR σχήματος καθώς και τον τρόπο με τον οποίον βοηθάμε να ικανοποιηθούν οι δύο παραπάνω απαιτήσεις.

3.4.1 Εντοπισμός παραβατικών εντολών ανάγνωσης

Προκειμένου να εντοπίσουμε τις αναφορές με το μεγαλύτερο μερίδιο στις συνολικές αστοχίες κρυφής μνήμης ακολουθήσαμε μια προσέγγιση παρόμοια με αυτή του [Wang 04]. Πραγματοποιούμε *σκιαγράφηση προφίλ* (profiling) στα προγράμματα των εφαρμογών χρησιμοποιώντας το εργαλείο Cachegrind του προσομοιωτή Valgrind [Nethercote 03]. Το εργαλείο αυτό έχει τη δυνατότητα να προσομοιώνει λειτουργικά την εκτέλεση ενός προγράμματος σε μια ιεραρχία κρυφής μνήμης οριζόμενη από τον χρήστη (I1, D1, L2 κρυφές μνήμες) και να συσχετίζει κάθε αστοχία που σημειώνεται με τη γραμμή στον κώδικα που την προκάλεσε. Στην περίπτωσή μας ρυθμίσαμε τον προσομοιωτή στις παραμέτρους της ιεραρχίας μνήμης της πραγματικής πλατφόρμας αξιολόγησης (βλ. ενότητα 3.5.1). Σημειώνουμε ότι με αυτόν τον τρόπο μοντελοποιείται *κατά προσέγγιση* η αλληλεπίδραση ενός προγράμματος με την ιεραρχία μνήμης του πραγματικού συστήματος, καθώς δε συνυπολογίζονται μια σειρά από παράμετροι λειτουργίας που μπορούν να επηρεάσουν το χρονισμό των αναφορών όπως π.χ. ο μηχανισμός προφόρτωσης δεδομένων υλικού ή η εκτός σειράς εκτέλεση εντολών. Ωστόσο, από την εμπειρία μας θεωρούμε ότι αυτή η μέθοδος σκιαγράφησης προφίλ είναι πιο ακριβής από αντίστοιχες μεθόδους άλλων εργαλείων (π.χ. Oprofile [opr 09]) οι οποίες βασίζονται στη δειγματοληψία γεγονότων που καταγράφονται στους μετρητές απόδοσης του επεξεργαστή.

Από τα αποτελέσματα που παρέχει το εργαλείο ήμασταν σε θέση να απομονώσουμε για κάθε πρόγραμμα τις εντολές ανάγνωσης που προκαλούσαν την πλειοψηφία των αστοχιών στην κρυφή μνήμη L2. Στα περισσότερα προγράμματα που εξετάσαμε ένας μικρός μόνο αριθμός στατικών εντολών ανάγνωσης ευθύνονται για την πλειοψηφία των αστοχιών. Συγκεκριμένα, σε 6 από τα 7 προγράμματα εντοπίσαμε από 1 έως 6 στατικές εντολές να προκαλούν το 92%–96% των συνολικών αστοχιών της L2. Οι εντολές αυτές καλούνται συνήθως στο εσωτερικό πολλαπλά εμφωλευμένων βρόχων, γύρω από τους οποίους αναλώνεται σημαντικό μέρος του συνολικού χρόνου εκτέλεσης των προγραμμάτων.

3.4.2 Παραγωγή κώδικα για τα βοηθητικά νήματα προφόρτωσης

Ο κώδικας για τα βοηθητικά νήματα προκύπτει από τον αρχικό σειριακό παραλείποντας όλες τις εντολές πλην εκείνων που οδηγούν στον υπολογισμό των διευθύνσεων των παραβατικών αναφορών. Για τα προγράμματα που εξετάσαμε, ο εντοπισμός και η απομόνωση των προς-τα-πίσω εξαρτώμενων εντολών ήταν σχετικά απλή διαδικασία και έγινε χειροκίνητα, καθώς ο αριθμός των παραβατικών αναφορών ήταν μικρός και οι πιο πολλές ακολουθίες εξαρτώμενων εντολών εκτείνονταν εντός μικρών περιοχών της ίδιας συνάρτησης (π.χ. σώματα βρόχων).

Σε περίπτωση που ο αριθμός των παραβατικών αναφορών ήταν μεγάλος ή η εύρεση των προς-τα-πίσω εξαρτώμενων εντολών ήταν ιδιαίτερα περίπλοκη (π.χ. εκτεινόταν πέρα από τα όρια της συνάρτησης όπου βρίσκεται η αντίστοιχη αναφορά-στόχος), θα βοηθούσε η αυτοματοποιημένη παραγωγή κώδικα για τα βοηθητικά νήματα μέσω κάποιου εργαλείου *τεμαχισμού κώδικα* (code slicing). Αυτή η προσέγγιση ακολουθείται στα [Kim 04a, Kim 04b].

Για την προφόρτωση δεδομένων στην κρυφή μνήμη L2 χρησιμοποιήσαμε τις εντολές PREFETCH*h* του συνόλου εντολών του επεξεργαστή. Θα μπορούσαμε εναλλακτικά να αντιγράφουμε τα δεδομένα σε μεταβλητές που είναι αποθηκευμένες σε καταχωρητές του επεξεργαστή [1], αλλά η πρώτη προσέγγιση έχει τα εξής πλεονεκτήματα: δεν εισάγει μεγάλη παρεμβολή στη σωλήνωση (π.χ. στις μονάδες εκτέλεσης, ουρές ανάγνωσης, κ.λπ.), και προφορτώνει δεδομένα αποκλειστικά στην κρυφή μνήμη L2 χωρίς να επηρεάζει την L1.

3.4.3 Συγχρονίζοντας το κύριο και το βοηθητικό νήμα

Το πιο σημαντικό μέρος στην υλοποίηση του SPR σχήματος είναι ο συγχρονισμός ανάμεσα στο κύριο και το βοηθητικό νήμα. Επηρεάζει τόσο την ικανότητα του βοηθητικού νήματος να καλύπτει επιτυχώς της αστοχίες του κύριου νήματος, όσο και την επιβάρυνση που αυτό ασκεί στο κύριο νήμα εκτέλεσης. Στα προγράμματά μας σχεδόν όλες οι παραβατικές εντολές ανάγνωσης βρίσκονται μέσα σε βρόχους. Γενικά, οι εμφωλευμένοι βρόχοι αποτελούν μια δομή που προσφέρεται για την εφαρμογή ενός μοντέλου *παραγωγού-καταναλωτή* (producer-consumer), σύμφωνα με το οποίο το βοηθητικό νήμα τρέχει μπροστά από το κύριο, φορτώνει μια συγκεκριμένη ποσότητα δεδομένων στην κρυφή μνήμη και περιμένει μέχρι το κύριο νήμα να αρχίσει να τα "καταναλώνει".

Προκειμένου να έχουμε υπό έλεγχο την απόσταση του βοηθητικού νήματος από το κύριο νήμα επιβάλουμε ένα άνω όριο στην ποσότητα δεδομένων που αυτό μπορεί να προφορτώνει σε κάθε φάση. Αυτή η απόσταση πρέπει να είναι αρκετά μεγάλη, ώστε τα δεδομένα να προλαβαίνουν να φορτώνονται στην κρυφή μνήμη πριν τα ζητήσει το κύριο νήμα, και ταυτόχρονα αρκετά μικρή, ώστε να μην εκτοπίζουν παλαιότερα δεδομένα από την κρυφή μνήμη τα οποία δεν έχουν ακόμα χρησιμοποιηθεί από το κύριο νήμα. Στα προγράμματά μας θέτουμε εμπειρικά αυτό το

[1] χρησιμοποιώντας τον προσδιοριστή αποθήκευσης `register` της C

όριο, φροντίζοντας ώστε τα δεδομένα που προφορτώνονται σε κάθε φάση να έχουν συνολικό μέγεθος που να μην ξεπερνά την μισή χωρητικότητα της κρυφής μνήμης L2.

Για να εντοπίσουμε τις περιοχές του κώδικα των νημάτων προφόρτωσης που καλύπτουν τη ζητούμενη ποσότητα δεδομένων ακολουθούμε την εξής διαδικασία: ξεκινούμε από το εσωτερικότερο βρόχο που περιέχει μια ή περισσότερες παραβατικές εντολές και συνεχίζουμε προς στα εξωτερικότερα επίπεδα, μέχρι οι συνολικές αναφορές μνήμης που έχουν γίνει μέχρι εκείνη τη στιγμή να έχουν αποτύπωμα στη μνήμη ίσο με το επιβαλλόμενο όριο. Τα σημεία εισόδου και εξόδου τέτοιων *περιοχών προεκτέλεσης* (precomputation spans) οριοθετούν τα σημεία συγχρονισμού ανάμεσα στο κύριο και το βοηθητικό νήμα. Όσο πιο μικρές είναι αυτές οι περιοχές, τόσο πιο συχνός γίνεται ο συγχρονισμός ανάμεσα στα νήματα, και το αντίστροφο. Ο συχνός συγχρονισμός επιτρέπει τον λεπτομερή έλεγχο πάνω στα δεδομένα που προφορτώνονται, όμως ενέχει αυξημένο κόστος. Αυτός είναι και ο βασικός λόγος που επιλέγουμε να προφορτώνουμε δεδομένα στην κρυφή μνήμη L2 αντί για την αρκετά μικρότερη L1.

Υλοποιούμε τον συγχρονισμό ανάμεσα στα νήματα εσωκλείοντας τις περιοχές προεκτέλεσης με *φράγματα συγχρονισμού* (synchronization barriers). Το φράγμα για το βοηθητικό νήμα τοποθετείται στο σημείο εξόδου μιας περιοχής, ενώ για το κύριο νήμα τοποθετείται στο σημείο εισόδου. Με αυτόν τον τρόπο το βοηθητικό νήμα τρέχει μπροστά από το κύριο νήμα. Κάθε φορά που προφορτώνει την αναμενόμενη ποσότητα δεδομένων αλλά το κύριο νήμα δεν έχει ακόμα αρχίσει να τα χρησιμοποιεί, το βοηθητικό νήμα εισέρχεται στο φράγμα και σταματάει την παραπέρα πορεία του. Μπορεί να προχωρήσει στην επόμενη φάση προφόρτωσης μόνο όταν ειδοποιηθεί ότι το κύριο νήμα έχει αρχίσει να καταναλώνει τα δεδομένα της παρούσας φάσης, που σημαίνει ότι το κύριο νήμα έχει εισέλθει και αυτό στο αντίστοιχο φράγμα συγχρονισμού.

Στη γενική περίπτωση τα βοηθητικά νήματα εισέρχονται σχεδόν πάντα πρώτα στα φράγματα συγχρονισμού, δεδομένου ότι ολοκληρώνουν πρώτα το σχετικά ελαφρύ φορτίο εργασίας τους σε κάθε φάση προφόρτωσης. Κατά συνέπεια, τα κύρια νήματα εργασίας δε χρειάζεται σχεδόν ποτέ να περιμένουν όταν εισέρχονται στα αντίστοιχα φράγματα. Στην πράξη μετρήσαμε ότι κατά μέσο όρο μόνο το 1.45% του συνολικού χρόνου εκτέλεσης του κύριου νήματος καταναλωνόταν μέσα στα φράγματα συγχρονισμού, σε όλα τα υπό εξέταση προγράμματα. Ένα παράδειγμα του τρόπου υλοποίησης του συγχρονισμού ανάμεσα στα κύρια και τα βασικά νήματα φαίνεται στο Σχήμα 3.1. Στο Σχήμα 3.2 φαίνεται ένα γενικό παράδειγμα εκτέλεσης του κύριου και του βοηθητικού νήματος στα πλαίσια του σχήματος SPR. Τα νήματα εκτελούνται σε διαφορετικούς λογικούς επεξεργαστές του ίδιου Hyper-threaded επεξεργαστή, ενώ κάθε φάση λειτουργίας αντιστοιχεί σε μια περιοχή προεκτέλεσης.

Κώδικας 3.1: *Κύριο νήμα*

```
for (k=0; k<n; k++) {
   for (ib=0; ib<n; ib+=bs) {

      barrier_wait(&barrier);
      for (jb=0; jb<n; jb+=bs) {
         for(i=ib; i<MIN(ib+bs,n); i++) {
            for(j=jb; j<MIN(jb+bs,n); j++) {
               tc[i][j] = MIN(g[i][k]+g[k][j], g[i][j]);
            }
         }
      }

   }
   dtemp = tc;
   tc = g;
   g = dtemp;
}
```

Κώδικας 3.2: *Βοηθητικό νήμα*

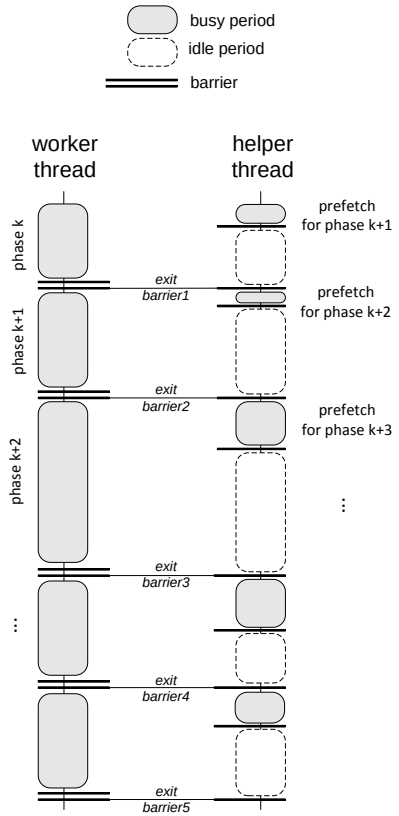
```
g_local=g;
tc_local=tc;

for (k=0; k<n; k++) {
   for (ib=0; ib<n; ib+=bs) {

      for (jb=0; jb<n; jb+=bs) {
         for(i=ib; i<MIN(ib+bs,n); i++) {
            for(j=jb; j<MIN(jb+bs,n); j++) {
               prefetch(&g_local[i][j+64]);
            }
         }
      }
      barrier_wait(&barrier);

   }
   dtemp = tc_local;
   tc_local = g_local;
   g_local = dtemp;
}
```

Σχήμα 3.1: *Συγχρονίζοντας το κύριο και το βοηθητικό νήμα στο μετροπρόγραμμα TC, στα πλαίσια του σχήματος SPR. Οι τρεις εσωτερικότεροι βρόχοι συνιστούν μια περιοχή προεκτέλεσης. Όλες οι μεταβλητές εκτός από αυτές που φαίνονται υπογραμμισμένες είναι τοπικές για κάθε νήμα.*



Σχήμα 3.2: *Παράδειγμα εκτέλεσης του SPR σχήματος.*

3.4.4 Βελτιστοποιήσεις στο συγχρονισμό

Καθοριστικός παράγοντας για την αποτελεσματικότητα του SPR είναι η αποδοτικότητα των φραγμάτων συγχρονισμού αυτών καθεαυτών. Το νήμα προφόρτωσης, που συνήθως εισέρχεται πρώτο στο φράγμα λόγω του μικρού φορτίου του, δεν πρέπει να επιβραδύνει την εκτέλεση του κύριου νήματος για όσο περιμένει στο φράγμα, καταναλώνοντας πόρους που θα μπορούσαν να φανούν χρήσιμοι για την πρόοδο του κύριου νήματος. Επιπλέον, μόλις το κύριο νήμα φτάνει και αυτό στο φράγμα, το βοηθητικό νήμα πρέπει να "ξυπνήσει" και να αναχωρήσει από αυτό όσο το δυνατόν πιο άμεσα, ώστε οι ενέργειές του για την επόμενη φάση εκτέλεσης να είναι έγκαιρες. Τέλος, ο χρόνος που το κύριο νήμα ξοδεύει στο φράγμα πρέπει να είναι ο ελάχιστος δυνατός. Συνεπώς, οι βασικές απαιτήσεις που πρέπει να ικανοποιούν τα φράγματα συγχρονισμού είναι η *χαμηλή κατανάλωση πόρων*, η *υψηλή αποκρισιμότητα* (ή αλλιώς, χαμηλή καθυστέρηση), και το *χαμηλό κόστος κλήσης*. Συνήθως αυτές οι απαιτήσεις είναι συγκρουόμενες. Για παράδειγμα, η γρήγορη γνωστοποίηση της αλλαγής μιας κατάστασης απαιτεί γρήγορους, επαναληπτικούς ελέγχους στη μνήμη, και έτσι μπορεί να καταναλώσει σημαντικούς επεξεργαστικούς πόρους ή εύρος ζώνης του διαύλου μνήμης.

Σε περιπτώσεις όπου οι περιοχές προεκτέλεσης απαιτούν συχνό συγχρονισμό η αποκρισιμότητα μπορεί να είναι πιο σημαντική ώστε το SPR σχήμα να λειτουργεί αποδοτικά. Από την άλλη, η κατανάλωση πόρων μπορεί να είναι σημαντικότερος παράγοντας για περιοχές προεκτέλεσης που περιλαμβάνουν λίγη δουλειά και μακρές περιόδους αναμονής για το νήμα προφόρτωσης. Στα πλαίσια αυτής της διατριβής εξετάσαμε αρχικά μερικές από τις πιο διαδεδομένες υλοποιήσεις φραγμάτων συγχρονισμού προκειμένου να αξιολογήσουμε το βαθμό στον οποίον ικανοποιούν τις προαναφερθείσες απαιτήσεις. Συγκεκριμένα, εξετάσαμε φράγματα βασισμένα σε *βρόχους περιδίνησης* (spin-wait loops), ή αλλιώς *βρόχους ενεργού αναμονής* (busy-wait loops), φράγματα βασισμένα στην εντολή HALT του επεξεργαστή και τέλος φράγματα που παρείχε η βιβλιοθήκη Pthreads. Αναλυτική περιγραφή των μηχανισμών αυτών καθώς και των επιπτώσεων λειτουργίας τους στα πλαίσια του Hyper-threading θα δώσουμε στο Κεφάλαιο 4. Στο ίδιο κεφάλαιο θα παρουσιάσουμε και τον μηχανισμό που εμείς προτείνουμε, και ο οποίος βασίζεται σε ειδικές εντολές του επεξεργαστή οι οποίες όμως δεν μπορούν να χρησιμοποιηθούν άμεσα από τα προγράμματα του χρήστη.

Συνοψίζουμε ωστόσο εδώ προκαταβολικά τα βασικότερα ποιοτικά συμπεράσματα που προέκυψαν: η πρώτη υλοποίηση, που βασίζεται σε βρόχους περιδίνησης και λειτουργεί εξ' ολοκλήρου σε επίπεδο χρήστη, προσφέρει τους μικρότερους χρόνους αφύπνισης για το βοηθητικό νήμα αλλά είναι η πιο "επιθετική" στην κατανάλωση πόρων και την επιβράδυνση του κύριου νήματος. Οι υπόλοιπες δύο υλοποιήσεις είναι κατά πολύ πιο ήπιες όσον αφορά την κατανάλωση πόρων, με την υλοποίηση των Pthreads να δίνει ελαφρώς καλύτερους χρόνους αφύπνισης του βοηθητικού νήματος. Για το SPR, η υλοποίηση των Pthreads ήταν αυτή που υπερίσχυσε έναντι των υπολοίπων, δίνοντας καλύτερους χρόνους εκτέλεσης. Αυτό συμβαίνει διότι οι περισσότερες παραβατικές εντολές ανάγνωσης στα προγράμματά μας συμπεριλαμβάνουν σχετικά λίγους υπολογισμούς και μεγάλες περιόδους αναμονής, πράγμα που καθιστά την αποδοτική διαχείριση πόρων πιο σημαντική από την μεγάλη αποκρισιμότητα. Σαν μελλοντική δουλειά, θα εξετάσουμε το ενδεχόμενο υβριδικών υλοποιήσεων για την προσαρμογή του είδους συγχρονισμού ανάλογα με το προφίλ εκτέλεσης κάθε φάσης προφόρτωσης (π.χ. συχνότητα συγχρονισμού, ποσότητα υπολογισμών, κ.λπ.).

3.5 Πειραματική Αξιολόγηση

3.5.1 Περιβάλλον εκτέλεσης πειραμάτων

Πλατφόρμα εκτέλεσης

Εκτελέσαμε τα πειράματα σε έναν επεξεργαστή Intel Xeon στα 2.8GHz. Βασίζεται στην μικροαρχιτεκτονική Netburst (έκδοσης "Prescott") και είναι ένας από τους πρώτους ευρέως διαθέσιμους επεξεργαστές που υλοποίησαν μια χαμηλού κόστους και περιορισμένων δυνατοτήτων

Επεξεργαστής	Intel Xeon (Prescott)
Συχνότητα ρολογιού	2.8GHz
Κρυφή μνήμη L1 δεδομένων	16KB, 8-way, 64 bytes μέγεθος γραμμής
Κρυφή μνήμη ιχνών	12K uops, 8-way
Κρυφή μνήμη L2 (ενοποιημένη)	1MB, 8-way, 64 bytes μέγεθος γραμμής
DTLB, ITLB	64 εγγραφές
Κύρια μνήμη	2GB
Συνολικός αριθμός νημάτων υλικού	4
Διάταξη νημάτων υλικού	2 SMP φυσικοί επεξεργαστές $\times$ 2 Hyper-threads

Πίνακας 3.1: *Χαρακτηριστικά πλατφόρμας εκτέλεσης.*

εκδοχή της τεχνικής του SMT, μέσω της τεχνολογίας Hyper-threading. Τα βασικότερα χαρακτηριστικά της μικροαρχιτεκτονικής Netburst έχουν περιγραφεί στην ενότητα 2.4. Στον Πίνακα 3.1 παρουσιάζονται χαρακτηριστικά του υποσυστήματος μνήμης του επεξεργαστή καθώς και οι ευρύτερες επεξεργαστικές δυνατότητες της πλατφόρμας εκτέλεσης.

Ο επεξεργαστής Xeon παρέχει δυνατότητες για προφόρτωση δεδομένων στο λογισμικό μέσω των εντολών γλώσσας μηχανής PREFETCH*h*. Ο προγραμματιστής μπορεί να χρησιμοποιήσει τις εντολές αυτές σαν *υποδείξεις* στον επεξεργαστή για το ποιες θέσεις μνήμης θα προσπελαστούν στο άμεσο μέλλον. Το αν θα εκτελεστεί ή όχι η προφόρτωση δεδομένων εξαρτάται καθαρά από τις αποφάσεις του επεξεργαστή τη δεδομένη στιγμή. Επιπλέον, οι εντολές αυτές δεν επηρεάζουν τη ροή ελέγχου του προγράμματος, π.χ. δεν προκαλούν σφάλματα αν η διεύθυνση-όρισμα είναι μη έγκυρη όπως θα γινόταν με μια συνηθισμένη εντολή ανάγνωσης.

Ο επεξεργαστής επίσης διαθέτει λειτουργία προφόρτωσης δεδομένων στο υλικό (hardware prefetching). Βασίζεται σε έναν απλό μηχανισμό πρόβλεψης ο οποίος ανιχνεύει μοτίβα αναφορών, όπως για παράδειγμα σειριακές προσπελάσεις ή προσπελάσεις σταθερού διασκελισμού, και προφορτώνει γραμμές δεδομένων από την κύρια μνήμη στην κρυφή μνήμη L2. Λειτουργεί με διαφανή τρόπο ως προς το λογισμικό, χωρίς ο προγραμματιστής να μπορεί να παρεμβαίνει στη λειτουργία του, ενώ υποστηρίζει προφόρτωση για πολλαπλές ροές αναφορών ταυτόχρονα. Τέλος, μπορεί να ενεργοποιηθεί ή να απενεργοποιηθεί κατά βούληση από τον χρήστη. Γενικά, η χρήση του μηχανισμού προφόρτωσης έχει αναφερθεί ότι προσφέρει σημαντικά οφέλη στην απόδοση (π.χ. [Goumas 08]). Παρόλα αυτά έχουν αναφερθεί περιπτώσεις όπου αυτό δε συμβαίνει πάντα, όπως π.χ. όταν η προφόρτωση στο υλικό γίνεται ταυτόχρονα με προφόρτωση στο λογισμικό με αποτέλεσμα να ασκείται σημαντική πίεση στο δίαυλο μνήμης ([Papadopoulos 08]).

Λειτουργικό σύστημα

Το λειτουργικό σύστημα που χρησιμοποιήσαμε ήταν Linux με πυρήνα 2.6.13 για αρχιτεκτονική x86_64. Ένας σημαντικός παράγοντας για την απόδοση πολυνηματικών εφαρμογών σε έναν επεξεργαστή με τεχνολογία Hyper-threading είναι ο αλγόριθμος δρομολόγησης του λειτουργικού συστήματος. Προφανώς, ακόμα και αν οι δύο λογικοί επεξεργαστές ενός φυσικού πακέτου αναγνωρίζονται ως διαφορετικές επεξεργαστικές μονάδες, θα πρέπει να ξεχωρίζονται από λογικούς επεξεργαστές άλλων πακέτων.

Ο πυρήνας του Linux από την έκδοση 2.6 υλοποιεί έναν αλγόριθμο δρομολόγησης για πολυεπεξεργαστικά συστήματα όπου οι επεξεργαστές έχουν διαφορετικούς συσχετισμούς μεταξύ τους και δεν είναι όλοι ισότιμοι. Τέτοια συστήματα είναι για παράδειγμα SMP πλατφόρμες με Hyper-threaded επεξεργαστές ή πολυεπεξεργαστικά μηχανήματα *μη-ομοιόμορφης πρόσβασης στη μνήμη* (non-uniform memory access – NUMA). Η λειτουργία του αλγορίθμου βασίζεται στα λεγόμενα *πεδία δρομολόγησης* (scheduling domains). Πρόκειται για σύνολα από επεξεργαστικές μονάδες οι οποίες έχουν κοινές ιδιότητες και στις οποίες εφαρμόζονται κοινές πολιτικές δρομολόγησης. Με τα πεδία δρομολόγησης οι διαφορετικές ιδιότητες των επεξεργαστικών μονάδων γίνονται γνωστές στο δρομολογητή και αυτός με τη σειρά του μπορεί να πάρει τις ανάλογες αποφάσεις.

Ο δρομολογητής του Linux χρησιμοποιεί την εντολή HALT για να βελτιώσει την απόδοση εφαρμογών σε ένα σύστημα με Hyper-threading. Συγκεκριμένα, όταν υπάρχει μία μόνο εκτελέσιμη εργασία σε έναν φυσικό επεξεργαστή εκτελεί την HALT για να απενεργοποιήσει τον έναν από τους δύο λογικούς επεξεργαστές και να οδηγήσει τον φυσικό επεξεργαστή σε Single-Threaded τρόπο λειτουργίας. Με αυτό τον τρόπο η εργασία έχει στη διάθεσή της όλους τους πόρους του φυσικού επεξεργαστή. Η HALT χρησιμοποιείται επίσης από το δρομολογητή όταν υπάρχουν δύο εργασίες με διαφορετικές προτεραιότητες, η κάθε μία εκτελούμενη σε έναν λογικό επεξεργαστή του ίδιου φυσικού πακέτου. Για την εργασία με τη μεγαλύτερη προτεραιότητα ο δρομολογητής θα της παρέχει το σύνολο των πόρων του φυσικού επεξεργαστή, οδηγώντας τον κατά περιόδους σε Single-Threaded τρόπο λειτουργίας.

Μετρητές απόδοσης

Οι περισσότεροι σύγχρονοι επεξεργαστές ενσωματώνουν ειδικό υλικό για μέτρηση της απόδοσης. Οι επεξεργαστές της Intel παρέχουν μηχανισμούς για την επιλογή, το φιλτράρισμα, τη μέτρηση και την ανάγνωση μικροαρχιτεκτονικών γεγονότων σχετικών με την απόδοση, μέσω *ειδικών ανά μοντέλο καταχωρητών* (model specific registers – MSRs). Στον επεξεργαστή Xeon μπορεί κανείς να μετρήσει ένα ευρύ σύνολο τέτοιων γεγονότων, όπως για παράδειγμα αστοχίες σε κάποιο επίπεδο κρυφής μνήμης, αστοχίες στο TLB, στατιστικά σχετικά με τη πρόβλεψη

διακλαδώσεων, σχετικά με τις εκτελούμενες εντολές/μικροεντολές και τις δοσοληψίες στο δίαυλο μνήμης. Με την εισαγωγή της τεχνολογίας Hyper-threading οι δυνατότητες μέτρησης της απόδοσης επεκτάθηκαν ώστε οι μετρητές να μπορούν να επιλέγουν και να καταγράφουν γεγονότα με βάση τον λογικό επεξεργαστή από τον οποίον αυτά προέρχονται, σε όποια γεγονότα αυτό είναι εφικτό.

Οι μετρητές απόδοσης βοηθούν τους προγραμματιστές να κατανοήσουν καλύτερα ζητήματα που αφορούν την εκτέλεση ενός προγράμματος και να εντοπίσουν πιθανές στενωπούς που περιορίζουν την απόδοσή του. Επιπλέον, παρατηρώντας τον τρόπο που αποκρίνεται ο επεξεργαστής σε προγράμματα με συγκεκριμένα χαρακτηριστικά, ο προγραμματιστής μπορεί να αποκτήσει καλύτερη εικόνα πάνω στον εσωτερικό τρόπο λειτουργίας του επεξεργαστή, διαφωτίζοντας σημεία του που πολλές φορές δεν τεκμηριώνονται στα επίσημα εγχειρίδια. Τέλος, όπως σημειώνουν και οι κατασκευαστές των επεξεργαστών, τα αποτελέσματα που καταγράφουν οι μετρητές απόδοσης πολλές φορές είναι προσεγγιστικά των αντίστοιχων γεγονότων. Για αυτό το λόγο δεν θα πρέπει να τα ερμηνεύουμε “κατά γράμμα” αλλά πιο πολύ να λαμβάνουμε υπόψη τις σχετικές τους αποκλίσεις για τα προγράμματα των οποίων την απόδοση θέλουμε να συγκρίνουμε. Σε κάθε περίπτωση επομένως τα αποτελέσματα των μετρητών πρέπει να λειτουργούν βοηθητικά, σαν περαιτέρω ενδείξεις στη διαδικασία αξιολόγησης της απόδοσης.

Οι MSRs στον επεξεργαστή Xeon είναι οργανωμένοι σε 45 καταχωρητές ελέγχου για επιλογή γεγονότων (event selection control registers – ESCRs), 18 καταχωρητές ελέγχου για ρύθμιση μετρητών (counter configuration control registers – CCCRs) και 18 μετρητές. Οι CCCRs συσχετίζουν τους ESCRs με συγκεκριμένους μετρητές και ρυθμίζουν τη λειτουργία των τελευταίων. Σημειώνουμε ότι συγκεκριμένοι μετρητές υποστηρίζουν συγκεκριμένα μόνο γεγονότα και δεν είναι επιτρεπτοί όλοι οι πιθανοί συνδυασμοί. Αυτές οι συσχετίσεις περιπλέκονται ακόμα περισσότερο όταν μερικά γεγονότα πρέπει να μετρηθούν ανά λογικό επεξεργαστή στα πλαίσια λειτουργίας του Hyper-threading. Αυτό μάς οδηγεί στο να εκτελούμε πολλαπλές φορές κάποιο πρόγραμμα ώστε να μετράμε διαφορετικά γεγονότα σε κάθε εκτέλεση.

Για να χρησιμοποιήσουμε τις δυνατότητες που παρέχουν οι μετρητές απόδοσης αναπτύξαμε μια απλή βιβλιοθήκη. Για την ανάγνωση και τροποποίηση των MSRs η βιβλιοθήκη χρησιμοποιεί τη συσκευή `/dev/msr` που παρέχει το Linux, η οποία εξάγει στον χρήστη τα περιεχόμενα των MSRs και δίνει τη δυνατότητα προσπέλασής τους μέσω κλήσεων συστήματος. Με αυτόν τον τρόπο προσπελαύνουμε τους MSRs που ελέγχουν τη λειτουργία των μετρητών απόδοσης (ESCRs,CCCRs [Int 09b]). Για την ανάγνωση των ίδιων των μετρητών απόδοσης χρησιμοποιήσαμε την εντολή γλώσσας μηχανής RDPMC, η οποία μπορεί να κληθεί άμεσα από το επίπεδο χρήστη χωρίς να χρειάζονται κλήσεις συστήματος [2]. Για να γίνει εφικτό αυτό χρειάστηκε να

[2] Θα μπορούσαμε για την ανάγνωση των μετρητών απόδοσης να προσπελάσουμε τη συσκευή `/dev/msr` μέσω κλήσεως συστήματος, όπως κάνουμε για τους ESCRs, CCCRs. Σε αντίθεση όμως με τους τελευταίους, όπου η πρόσβαση γίνεται μία φορά για την αρχικοποίησή τους, η ανάγνωση των μετρητών μπορεί να γίνεται πολλές φορές

θέσουμε τη σημαία PCE στον καταχωρητή ελέγχου CR4, μέσω μιας απλής επέκτασης πυρήνα (kernel module) που υλοποιήσαμε.

3.5.2 Μετροπρογράμματα

Σαν μετροπρογράμματα για την αξιολόγηση της απόδοσης χρησιμοποιήσαμε υπολογιστικούς πυρήνες από ένα ευρύ φάσμα εφαρμογών καθώς και ολοκληρωμένες εφαρμογές. Μερικά από αυτά προέρχονται από υπάρχουσες συλλογές μετροπρογραμμάτων ενώ άλλα τα συγγράψαμε εξ' αρχής. Οι αντίστοιχοι κώδικες παρουσιάζουν διαφορετικά χαρακτηριστικά ως προς τις απαιτήσεις για επεξεργαστικούς πόρους καθώς και τον τρόπο πρόσβασης στη μνήμη.

Τα πρώτα δύο μετροπρογράμματα που χρησιμοποιήσαμε, τα BT και CG, προέρχονται από τη συλλογή NAS Parallel Benchmark Suite [Bailey 94]. Το **BT** είναι μια εφαρμογή προσομοίωσης ροής ρευστών που χρησιμοποιεί έναν αλγόριθμο για να επιλύσει τρισδιάστατες συμπτύξιμες εξισώσεις Navier-Stokes. Επιλύει τριδιαγώνια συστήματα σε μπλοκ 5×5 χρησιμοποιώντας την μέθοδο των πεπερασμένων διαφορών. Η εφαρμογή αυτή χρησιμοποιεί σε μεγάλο βαθμό πολυδιάστατους πίνακες και επιδεικνύει κακή τοπικότητα αναφορών.

Το **CG** είναι μια μέθοδος συζυγών κλίσεων (conjugate gradient) η οποία χρησιμοποιείται για να υπολογίσει μια προσέγγιση της μικρότερης ιδιοτιμής ενός μεγάλου αραιού, συμμετρικού και θετικά ορισμένου πίνακα, επιλύοντας ένα αδόμητο αραιό γραμμικό σύστημα. Η εφαρμογή δεν χαρακτηρίζεται από χρονική τοπικότητα, πραγματοποιεί αρκετές έμμεσες αναφορές στη μνήμη, αλλά η εκτέλεσή της δεν περιλαμβάνει γενικά υπερβολικό αριθμό αστοχιών.

Η απόδοση και των δύο μετροπρογραμμάτων αξιολογήθηκε για μεγέθη εισόδου Class A. Οι υλοποιήσεις των μετροπρογραμμάτων βασίστηκαν στις OpenMP C εκδόσεις των NAS Parallel Benchmarks 2.3 που παρέχονταν στα πλαίσια του Omni OpenMP Compiler Project [omn 03]. Μετατρέψαμε τις εκδόσεις αυτές ώστε αντί των παράλληλων δομών της OpenMP να χρησιμοποιούν ρουτίνες πολυνηματισμού για τη δημιουργία, το συγχρονισμό και την ανάθεση δουλειάς στα νήματα.

Στο μετροπρόγραμμα **HJ** υπολογίζεται η ένωση δύο σχέσεων μιας βάσης δεδομένων χρησιμοποιώντας τον αλγόριθμο *Hash-join* [Silberschatz 01], που αντιπροσωπεύει μια από τις βασικότερες λειτουργίες στην επεξεργασία ερωτημάτων σε βάσεις δεδομένων. Ο αλγόριθμος σε πρώτη φάση διαμερίζει και τις δύο σχέσεις εφαρμόζοντας μια συνάρτηση κατακερματισμού πάνω στο πεδίο ένωσης των σχέσεων. Με αυτόν τον τρόπο οι εγγραφές μιας διαμέρισης από την πρώτη σχέση αρκεί να αντιπαραβληθούν με τις εγγραφές από την αντίστοιχη διαμέριση της δεύτερης

κατά τη διάρκεια της εκτέλεσης. Συνεπώς η συνεχής προσπέλαση μέσω κλήσεων συστήματος θα εισήγαγε σημαντική καθυστέρηση στην εκτέλεση του προγράμματος.

σχέσης, καθώς όλες οι εγγραφές και των δύο διαμερίσεων έχουν την ίδια τιμή κατακερματισμού. Στη συνέχεια, για κάθε τέτοιο ζεύγος διαμερίσεων ο αλγόριθμος προχωράει ως εξής: χτίζει έναν πίνακα κατακερματισμού στη διαμέριση της μικρότερης σχέσης ("build") και τον αντιπαραβάλλει με εγγραφές της μεγαλύτερης σχέσης ("probe") για να βρει ταιριάσματα. Ο περισσότερος χρόνος κατά την εκτέλεση του αλγορίθμου αφιερώνεται σε λειτουργίες αναζήτησης στον πίνακα κατακερματισμού κατά τη διάρκεια αυτής της φάσης αντιπαραβολής.

Χρησιμοποιήσαμε μια σχέση με 2M εγγραφές (probe) και μια σχέση με 1M εγγραφές (build), τις οποίες διαμερίσαμε σε 20 ισομεγέθη τμήματα την κάθε μία. Οι σχέσεις δημιουργήθηκαν με τέτοιον τρόπο ώστε κάθε εγγραφή της μικρής σχέσης να ταιριάζει ακριβώς με δύο εγγραφές της μεγάλης σχέσης. Για κάθε διαμέριση της μικρής σχέσης χρησιμοποιήσαμε έναν πίνακα κατακερματισμού 500 υποδοχών. Δεδομένης της τυχαίας και ομοιόμορφης κατανομής των τιμών στο πεδίο ένωσης, αυτό σημαίνει ότι από κάθε υποδοχή αρχίζει μια συνδεδεμένη λίστα 100 κόμβων καθένας εκ των οποίων δείχνει σε διαφορετική εγγραφή της μικρής σχέσης. Ως εκ τούτου, το επιμερισμένο κόστος για κάθε αναζήτηση στον πίνακα κατακερματισμού είναι 50 άλματα δεικτών. Στην περίπτωση του σχήματος TLP, διαδοχικά ζεύγη διαμερίσεων από τη μικρή και τη μεγάλη σχέση ανατίθενται σε διαφορετικά νήματα με κυκλικό τρόπο. Κάθε νήμα χτίζει έναν πίνακα κατακερματισμού για τη μικρή διαμέριση και στη συνέχεια τον αντιπαραβάλλει με την αντίστοιχη μεγάλη διαμέριση για να βρει ταιριάσματα. Δεν υπάρχουν εξαρτήσεις ανάμεσα σε διαφορετικά ζεύγη των διαμερίσεων και έτσι δεν απαιτείται συγχρονισμός ανάμεσα στα νήματα.

Το **LU** είναι ένας υπολογιστικός πυρήνας που παραγοντοποιεί έναν τετραγωνικό 1024×1024 πίνακα σε άνω και κάτω τριγωνικούς πίνακες χρησιμοποιώντας την τεχνική της *επεξεργασίας κατά μπλοκ* (blocking). Στο TLP σχήμα ο πυρήνας παραλληλοποιείται αναθέτοντας στα νήματα διαφορετικούς υποπίνακες, στην ουσία μπλοκ διαστάσεων $B \times B$, τους οποίους επεξεργάζονται παράλληλα. Αυτό το σχήμα καταμερισμού αντιστοιχεί σε αδρομερή παραλληλισμό, και λειτουργεί παρομοίως με το σχήμα που προτείνεται στο [Woo 94].

Σε κάθε επανάληψη του αλγορίθμου γίνονται υπολογισμοί πάνω σε έναν όλο και μικρότερο τετραγωνικό υποπίνακα ο οποίος προκύπτει αγνοώντας τις B πρώτες γραμμές και B πρώτες στήλες του τρέχοντος πίνακα. Κατά τη διάρκεια μιας επανάληψης τα μπλοκ του τρέχοντος πίνακα χωρίζονται σε τρεις κατηγορίες: σε ένα διαγώνιο μπλοκ, σε περιμετρικά και σε εσωτερικά. Ο υπολογισμός των μπλοκ κάθε κατηγορίας εξαρτάται από τον υπολογισμό των μπλοκ της προηγούμενης, ενώ τα μπλοκ που ανήκουν στην ίδια κατηγορία μπορούν να υπολογιστούν ανεξάρτητα. Έτσι σε κάθε επανάληψη η παράλληλη εκτέλεση γίνεται σε τρεις φάσεις οι οποίες διαχωρίζονται μεταξύ τους με φράγματα συγχρονισμού ώστε να ικανοποιούνται οι παραπάνω εξαρτήσεις. Εντός κάθε φάσης διαφορετικά μπλοκ ανατίθενται κυκλικά στα νήματα για παράλληλη επεξεργασία, χωρίς να υπάρχει ανάγκη για μεταξύ τους συγχρονισμό.

Το **MM** είναι μια υψηλά βελτιστοποιημένη έκδοση του πολλαπλασιασμού μεταξύ τριών πινάκων διαστάσεων 1024×1024. Έχουν εφαρμοστεί τεχνικές όπως η κατά μπλοκ επεξεργασία, το *ξεδίπλωμα βρόχων* (loop unrolling) και η *κατά μπλοκ διάταξη* των στοιχείων των πινάκων στη μνήμη (block array layouts). Η τελευταία τεχνική έχει σαν αποτέλεσμα η σειρά με την οποία γίνεται η διάτρεξη των στοιχείων των πινάκων να είναι η ίδια με τη σειρά με την οποία είναι αποθηκευμένα στη μνήμη, γεγονός που συμβάλλει στην καλύτερη δυνατή αξιοποίηση της ιεραρχίας μνήμης. Επιπλέον, καθιστά πιο εύκολη την αποθήκευση των πινάκων με τέτοιο τρόπο ώστε τα νήματα να εργάζονται σε δεδομένα που δεν απεικονίζονται στις ίδιες περιοχές της κρυφής μνήμης, περιορίζοντας έτσι τις συγκρούσεις και τους εκτοπισμούς γραμμών δεδομένων που είναι δυνατόν να προκαλέσει το ένα νήμα στο άλλο. Στο TLP σχήμα εκτέλεσης, συνεχόμενα μπλοκ του πίνακα γινομένου C (σε οριζόντια σειρά) ανατίθενται για υπολογισμό σε διαδοχικά νήματα με κυκλικό τρόπο. Αυτό το σχήμα καταμερισμού αντιστοιχεί σε αδρομερή παραλληλισμό. Δεν υπάρχουν εξαρτήσεις ανάμεσα στα δεδομένα που υπολογίζονται και αυτά που διαβάζονται για τους υπολογισμούς, και έτσι δεν απαιτείται συγχρονισμός ανάμεσα στα νήματα.

Το **SV** είναι ένας υπολογιστικός πυρήνας πολλαπλασιασμού αραιού πίνακα με διάνυσμα. Χρησιμοποιήσαμε έναν πίνακα διάστασης 100000×100000, με 150 μη μηδενικά στοιχεία τυχαία και ομοιόμορφα κατανεμημένα σε κάθε του γραμμή. Είναι αποθηκευμένος χρησιμοποιώντας τη διάταξη της *συμπιεσμένης αραιής γραμμής* (compressed sparse row – CSR) [Barrett 94]. Σύμφωνα με αυτή τη διάταξη, αποθηκεύονται σε συνεχόμενες θέσεις μνήμης μόνο τα μη μηδενικά στοιχεία του πίνακα όπως αυτός διατρέχεται κατά γραμμές, ενώ υπάρχουν επιπλέον πίνακες για την δεικτοδότηση της στήλης κάθε μη μηδενικού στοιχείου και της αρχής κάθε γραμμής. Όπως και στο CG, αυτό οδηγεί σε πολλές έμμεσες αναφορές στη μνήμη. Όμως το SV χαρακτηρίζεται επιπλέον από ιδιαίτερα ακανόνιστες προσπελάσεις στο διάνυσμα εισόδου επειδή, σε αντίθεση με το CG, ο αραιός πίνακας έχει τυχαία δομή.

Η παραλληλοποίηση επιπέδου νημάτων για αυτό το μετροπρόγραμμα πραγματοποιείται με τη μορφή διαχωρισμού των δεδομένων στον εξωτερικότερο βρόχο, που είναι ένα σχήμα αδρομερούς παραλληλισμού. Με αυτόν τον τρόπο ο πίνακας διαμερίζεται με βάση τις γραμμές του, κάθε νήμα αναλαμβάνει ένα διαφορετικό σύνολο διαδοχικών γραμμών και υπολογίζει το αντίστοιχο τμήμα του διανύσματος εξόδου. Αυτό το σχήμα δεν υποθέτει συγχρονισμό καθώς δεν υπάρχουν εξαρτήσεις ανάμεσα στα δεδομένα διαφορετικών νημάτων. Επιπλέον, επειδή τα μη μηδενικά στοιχεία είναι ομοιόμορφα κατανεμημένα σε όλη την έκταση του πίνακα, το συνολικό φορτίο εργασίας κατανέμεται ομοιόμορφα ανάμεσα στα νήματα. Αν ο πίνακας είχε διαφορετική δομή, αυτό το σχήμα δε θα εξασφάλιζε πιθανώς την ισοκατανομή του φορτίου επειδή κάθε τμήμα γραμμών θα περιείχε διαφορετικό αριθμό μη μηδενικών στοιχείων. Σε αυτή την περίπτωση θα έπρεπε να εφαρμοστεί ένα σχήμα διαμερισμού του πίνακα σύμφωνα με τον αριθμό των μη

μηδενικών στοιχείων, όπως προτείνουμε χαρακτηριστικά στο [Goumas 09b].

Τέλος, το **TC** υπολογίζει τη μεταβατική κλειστότητα ενός κατευθυνόμενου γραφήματος με 1600 κορυφές και 25000 πλευρές. Το γράφημα αναπαρίσταται με πίνακα γειτνίασης. Το πρόβλημα αυτό στην ουσία ανάγεται σε ένα πρόβλημα εύρεσης συντομότερων διαδρομών από όλες προς όλες τις κορυφές (all-pairs shortest paths – APSP), το οποίο μπορεί να επιλυθεί χρησιμοποιώντας τον αλγόριθμο Floyd-Warshall [Cormen 01]. Η δομή του αλγορίθμου είναι αρκετά όμοια με αυτή του κλασικού πολλαπλασιασμού πινάκων. Ωστόσο, δεν είναι δυνατή η αναδιάταξη των βρόχων με οποιαδήποτε σειρά εξαιτίας των εξαρτήσεων που επιβάλλει ο εξωτερικότερος βρόχος.

Για να βελτιώσουμε την επαναχρησιμοποίηση των δεδομένων και να διευκολύνουμε την παραλληλοποίηση, υλοποιήσαμε μια έκδοση του αλγορίθμου εφαρμόζοντας την τεχνική της επεξεργασίας κατά μπλοκ στους δύο εσωτερικότερους βρόχους. Στο TLP σχήμα, σε κάθε επανάληψη του εξωτερικότερου βρόχου διαδοχικά μπλοκ του πίνακα γειτνίασης (σε οριζόντια σειρά) ανατίθενται με κυκλικό τρόπο σε διαφορετικά νήματα. Πριν προχωρήσουν τα νήματα στην επόμενη επανάληψη συγχρονίζονται μέσω φράγματος συγχρονισμού ώστε να μην παραβιαστούν οι εξαρτήσεις του εξωτερικότερου βρόχου.

Ο Πίνακας 3.2 συνοψίζει τα μετροπρογράμματα και τα δεδομένα εισόδου τους. Σημειώνουμε ότι από αυτά τα μετροπρογράμματα, τα BT, HJ και TC είχαν τη χειρότερη απόδοση στην κρυφή μνήμη στις αρχικές τους, σειριακές υλοποιήσεις, με τοπικά ποσοστά αστοχίας στην L2 ανάμεσα σε 23% και 35%. Τα LU, MM και CG, από την άλλη, είχαν την καλύτερη απόδοση, με ποσοστά αστοχίας κάτω από 0.6%, ενώ το SV σημείωνε ποσοστό αστοχίας γύρω στο 7%.

3.5.3 Πειραματικά αποτελέσματα

Στις ακόλουθες ενότητες παρουσιάζουμε την αξιολόγηση της απόδοσης των μετροπρογραμμάτων με βάση τα δύο σχήματα παραλληλοποίησης, TLP και SPR. Σε πρώτη φάση, παραθέτουμε αποτελέσματα για διάφορα στατιστικά που συλλέχθηκαν από τους μετρητές απόδοσης του επεξεργαστή. Σε δεύτερη φάση, παρουσιάζουμε την απόδοση που επιτύχαμε για κάθε πρόγραμμα και προσπαθούμε να την ερμηνεύσουμε μέσω των παραπάνω στατιστικών. Παρουσιάζουμε μετρήσεις για τα ακόλουθα γεγονότα:

- **Αστοχίες κρυφής μνήμης L2**: ο αριθμός των αιτήσεων που έκανε η κρυφή μνήμη L2 στη μονάδα διαύλου του επεξεργαστή, προκειμένου να φορτωθούν από την κύρια μνήμη δεδομένα για τις αναφορές οι οποίες αστόχησαν σε αυτήν. Οι αναφορές αφορούν εντολές ανάγνωσης και εγγραφής ("reads for ownership" – RFOs). Στα TLP σχήματα δείχνουμε το συνολικό αριθμό των αστοχιών στην L2 εκ μέρους και των δύο λογικών επεξεργαστών.

Σύντομο όνομα	Περιγραφή	Δεδομένα εισόδου
BT	NAS BT	Class A
CG	NAS CG	Class A
HJ	Hash-join	μεγάλη σχέση 2M εγγραφών (232MB), μικρή σχέση 1M εγγραφών (116MB), 20 διαμερίσεις ανά σχέση, πίνακας κατακερματισμού 500 εγγραφών
LU	Παραγοντοποίηση LU	πίνακας 1024×1024, μπλοκ 16×16
MM	Πολλαπλασιασμός πινάκων	πίνακες 1024×1024, μπλοκ 64×64
SV	Πολλαπλασιασμός αραιού πίνακα με διάνυσμα	πίνακας 100000×100000, 150 μη μηδενικά στοιχεία ανά γραμμή
TC	Μεταβατική κλειστότητα	1600 κορυφές, 25000 πλευρές, μπλοκ 20×20

Πίνακας 3.2: *Μετροπρογράμματα προς αξιολόγηση.*

Αυτό μάς δίνει μια ένδειξη για το αν ο διαμοιρασμός της L2 από τα νήματα μιας συγκεκριμένης εφαρμογής λειτουργεί *εποικοδομητικά* ή *καταστρεπτικά*. Η πρώτη περίπτωση συμβαίνει όταν η φύση της εφαρμογής είναι τέτοια ώστε η εκτέλεση ενός νήματος να οδηγεί σε συμπτωματικές προφορτώσεις δεδομένων που θα χρησιμοποιηθούν από το άλλο νήμα στο άμεσο μέλλον. Η δεύτερη περίπτωση συμβαίνει όταν το ένα νήμα προκαλεί εκτοπίσεις στις γραμμές δεδομένων του άλλου νήματος. Στα SPR σχήματα παρουσιάζουμε τις αστοχίες μόνο του κυρίου νήματος εργασίας, προκειμένου να απομονώσουμε την ικανότητα του νήματος προφόρτωσης να καλύψει επιτυχώς τις αστοχίες του πρώτου.

- **Αστοχίες κρυφής μνήμης L1**: ο αριθμός των εντολών που ολοκληρώθηκαν και οι οποίες αντιμετώπισαν αστοχίες στην ανάγνωση από την κρυφή μνήμη δεδομένων L1. Όπως και στην προηγούμενη περίπτωση, και εδώ για τα διάφορα σχήματα εκτέλεσης παρουσιάζουμε το συνολικό αριθμό αστοχιών στους λογικούς επεξεργαστές όπου εκτελούνται νήματα εργασίας.

- **Κύκλοι καθυστέρησης για ανάθεση πόρων**: ο αριθμός των κύκλων ρολογιού που ένα νήμα καθυστέρησε στη μονάδα ανάθεσης περιμένοντας μέχρι να ελευθερωθούν θέσεις στην ουρά εγγραφών. Όπως αναφέραμε στην ενότητα 2.4.2, η μονάδα ανάθεσης είναι το μέρος του επεξεργαστή που λαμβάνει μικροεντολές από την ουρά μικροεντολών και αναθέτει σε αυτές διαθέσιμες εγγραφές από τον ROB, την ουρά αναγνώσεων/εγγραφών,

κ.λπ. Μπλοκάρει την εκτέλεση ενός λογικού επεξεργαστή όταν αυτός επιχειρεί να χρησιμοποιήσει πάνω από τις μισές εγγραφές των δομών αυτών. Παρόλο που αυτή η μετρική απόδοσης δεν αποτυπώνει τις καθυστερήσεις σε όλες τις δομές που ελέγχει η μονάδα ανάθεσης, όπως κάνουν οι μηχανισμοί μέτρησης σε πιο σύγχρονους επεξεργαστές της Intel, είναι ενδεικτική της σύγκρουσης που υπάρχει ανάμεσα στα νήματα για διαμοιραζόμενους πόρους.

Για την σειριακή εκτέλεση κάθε εφαρμογής παρουσιάζουμε το λόγο των κύκλων καθυστέρησης ως προς το συνολικό αριθμό κύκλων. Για τα TLP σχήματα, παρουσιάζουμε το λόγο των κύκλων καθυστέρησης εκ μέρους και των δύο νημάτων ως προς τον συνολικό αριθμό κύκλων εκτέλεσης του σχήματος. Όταν αυτός ο λόγος είναι ιδιαίτερα αυξημένος σε σχέση με τη σειριακή εκτέλεση, αυτό υποδεικνύει πιθανές συγκρούσεις ανάμεσα στα νήματα εργασίας που έχουν σαν αποτέλεσμα την καθυστέρηση των νημάτων για μεγαλύτερο ποσοστό του χρόνου εκτέλεσης. Για τα SPR σχήματα, δείχνουμε τον λόγο των κύκλων καθυστέρησης του νήματος εργασίας ως προς το συνολικό αριθμό κύκλων εκτέλεσης του σχήματος. Όταν αυτός ο λόγος είναι αυξημένος σε σχέση με τη σειριακή εκτέλεση, αυτό σημαίνει ότι το νήμα εργασίας εμποδίζεται από τη συνεκτέλεση του νήματος προφόρτωσης.

- **Ολοκληρωθείσες μικροεντολές**: ο αριθμός των πραγματικών (“non-bogus”) μικροεντολών που ολοκληρώθηκαν κατά την εκτέλεση του προγράμματος. Οι “μη πραγματικές” μικροεντολές είναι εκείνες που ακυρώθηκαν και απορρίφθηκαν από τη σωλήνωση εξαιτίας λανθασμένων προβλέψεων διακλάδωσης. Σε όλα τα σχήματα εκτέλεσης παρουσιάζεται ο αριθμός των μικροεντολών που ολοκληρώθηκαν συνολικά από όλα τα νήματα. Συνεπώς, για τα TLP σχήματα, πιθανή αύξηση αυτού του αριθμού υποδεικνύει τις επιπλέον εντολές εξαιτίας του κόστους παραλληλοποίησης και συγχρονισμού. Για τα SPR σχήματα εκφράζει επιπλέον το μέγεθος του φορτίου εργασίας του νήματος προφόρτωσης.

Τόσο στις TLP όσο και στις SPR εκδόσεις των μετροπρογραμμάτων δημιουργούμε εξ’ αρχής δύο νήματα τα οποία απεικονίζουμε σε διαφορετικούς λογικούς επεξεργαστές του ίδιου φυσικού πακέτου. Χρησιμοποιήσαμε τη βιβλιοθήκη NPTL για τη δημιουργία και τη διαχείριση των νημάτων. Για να υποχρεώσουμε τα νήματα να δρομολογηθούν σε έναν συγκεκριμένο επεξεργαστή χρησιμοποιήσαμε τη κλήση συστήματος `sched_setaffinity`. Όλοι οι κώδικες μεταγλωττίστηκαν με την έκδοση 4.1.2 του gcc και με επίπεδο βελτιστοποίησης O2. Η έκδοση της glibc που χρησιμοποιήθηκε ήταν η 2.5. Σημειώνουμε τέλος ότι διατηρήσαμε ενεργοποιημένη τη λειτουργία προφόρτωσης δεδομένων στο υλικό για όλα τα πειράματα που εκτελέσαμε.

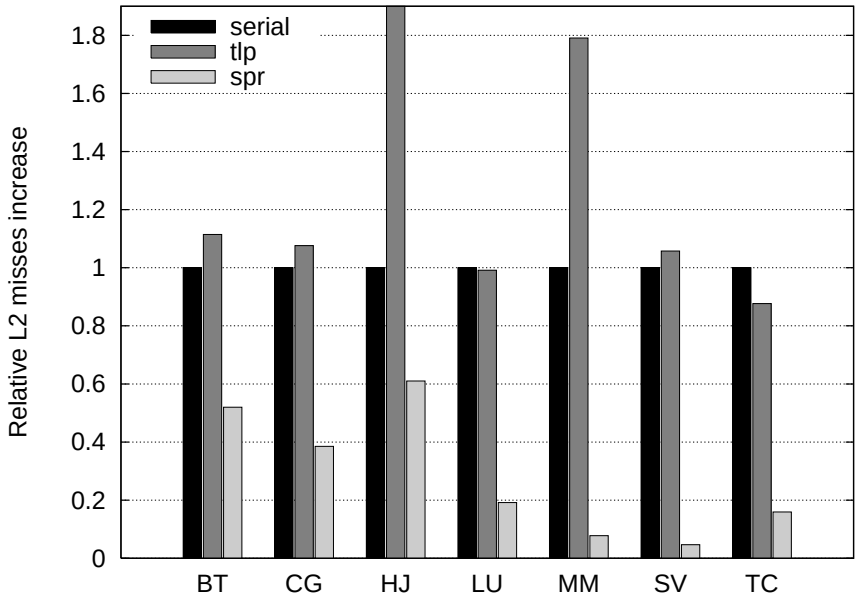
Απόδοση κρυφής μνήμης

Το Σχήμα 3.3α δείχνει τον αριθμό των αστοχιών στην κρυφή μνήμη L2 κανονικοποιημένο ως προς τον αντίστοιχο αριθμό της σειριακής εκτέλεσης. Στα TLP σχήματα οι αστοχίες στην L2 εκ μέρους και των δύο νημάτων αυξάνονται για τις περισσότερες εφαρμογές. Συγκεκριμένα, τα HJ και MM υφίστανται σημαντική αύξηση κατά ένα παράγοντα 2.44 και 1.79, αντίστοιχα. Στο HJ αυτό συμβαίνει διότι το σύνολο εργασίας των νημάτων σε κάθε φάση εκτέλεσης είναι αρκετά μεγαλύτερο από το μέγεθος της L2, που έχει σαν αποτέλεσμα μεγάλο αριθμό από εκτοπίσεις γραμμών δεδομένων του ενός νήματος από το άλλο λόγω περιορισμένης χωρητικότητας. Αυτό όμως δε συμβαίνει και με το MM. Αυτό το μετροπρόγραμμα έχει πολύ καλή τοπικότητα, με μοτίβα αναφορών εύκολα ανιχνεύσιμα από τη μονάδα προφόρτωσης δεδομένων, με πολύ μικρό αριθμό αστοχιών στην L2 και με σύνολα εργασίας των νημάτων τα οποία χωράνε στην L2. Συνεπώς, πιστεύουμε ότι η αύξηση των αστοχιών σε αυτή την περίπτωση οφείλεται σε συγκρούσεις στην L2, οι οποίες θα μπορούσαν να αντιμετωπιστούν με τεχνικές για πιο προσεκτική διάταξη των δεδομένων κάθε νήματος ώστε να ελαχιστοποιούνται οι αμοιβαίες εκτοπίσεις γραμμών κρυφής μνήμης.

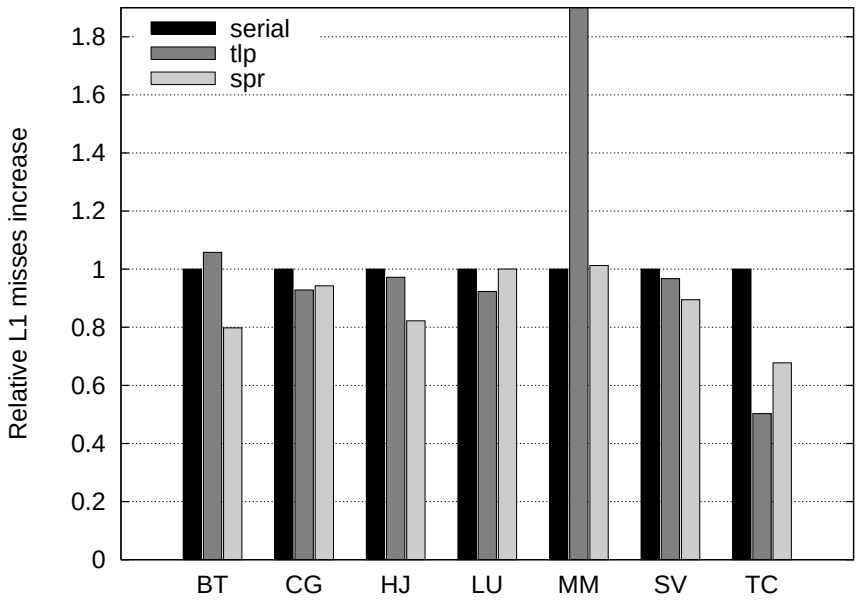
Στα υπόλοιπα μετροπρογράμματα η αύξηση των αστοχιών στην L2 ήταν μικρότερη από 11%. Από αυτά τα μετροπρογράμματα, στο TC σημειώθηκε στην πραγματικότητα μείωση του συνολικού αριθμού των αστοχιών κατά 13%. Αυτό δείχνει ότι σε αυτήν την εφαρμογή τα νήματα εργασίας δρουν με αμοιβαίο τρόπο και σαν νήματα προφόρτωσης, γεγονός που ευνοεί συνολικά την εφαρμογή όταν εκτελούνται κάτω από κοινή κρυφή μνήμη.

Στις εκδόσεις SPR τα νήματα προφόρτωσης κατάφεραν να καλύψουν τις αστοχίες των κύριων νημάτων εργασίας κατά 72% κατά μέσο όρο. Αυτό το ποσοστό καθιστά επιτυχή την επιλογή των παραβατικών εντολών ανάγνωσης καθώς και την εισαγωγή σημείων συγχρονισμού μεταξύ των νημάτων. Η μεγαλύτερη μείωση αστοχιών σημειώθηκε στο SV (κατά 96%, περίπου), και η μικρότερη στο HJ (κατά 39%). Γενικά, μεγαλύτερα ποσοστά κάλυψης επιτεύχθηκαν για κώδικες οι οποίοι έχουν λίγες παραβατικές εντολές ανάγνωσης, οι οποίες βρίσκονται μέσα σε συχνά εκτελούμενους εμφωλευμένους βρόχους (SV, MM, LU, TC). Επιπλέον, δεν απαιτείται μεγάλος αριθμός εντολών για τον υπολογισμό των διευθύνσεων αυτών των αναφορών, σε σύγκριση με κώδικες όπως ο HJ, όπου χρειάζονται πολλαπλά άλματα δεικτών για τον υπολογισμό της διεύθυνσης για μία και μόνο εντολή ανάγνωσης.

Το Σχήμα 3.3β παρουσιάζει τον αριθμό των αστοχιών στην L1 κανονικοποιημένο ως προς τη σειριακή εκτέλεση. Στις TLP υλοποιήσεις, η σχετική μεταβολή του αριθμού των αστοχιών ήταν μικρότερη από 7% για πέντε από τα εφτά μετροπρογράμματα. Οι εξαιρέσεις ήταν το MM, όπου σημειώθηκε αύξηση κατά 97%, και το TC, με μείωση κατά 50%. Στις SPR υλοποιήσεις οι αστοχίες του νήματος εργασίας στις περισσότερες εφαρμογές μειώθηκαν λιγότερο από 20%. Από την



(α) *Αστοχίες L2*



(β) *Αστοχίες L1*

Σχήμα 3.3: *Αριθμός αστοχιών στις κρυφές μνήμες L2 και L1 κανονικοποιημένος ως προς τη σειριακή εκτέλεση.*

εμπειρία μας, τέτοιες μικρές αποκλίσεις στις αστοχίες της L1 δεν αποτελούν ασφαλείς ενδείξεις για την τελική απόδοση. Γενικά, η συμπεριφορά της L1 δεν είναι κυρίαρχος παράγοντας όσον

αφορά την απόδοση, ειδικά όταν οι κώδικες δεν είναι βελτιστοποιημένοι για τοπικότητα. Αυτό ισχύει στα μετροπρογράμματα που εξετάζουμε, αφού ούτε τα TLP σχήματα διαμερίζουν τα δεδομένα ώστε να χωράν τμηματικά στην L1, ούτε τα SPR σχήματα στοχεύουν στις αστοχίες της L1.

Κύκλοι καθυστέρησης

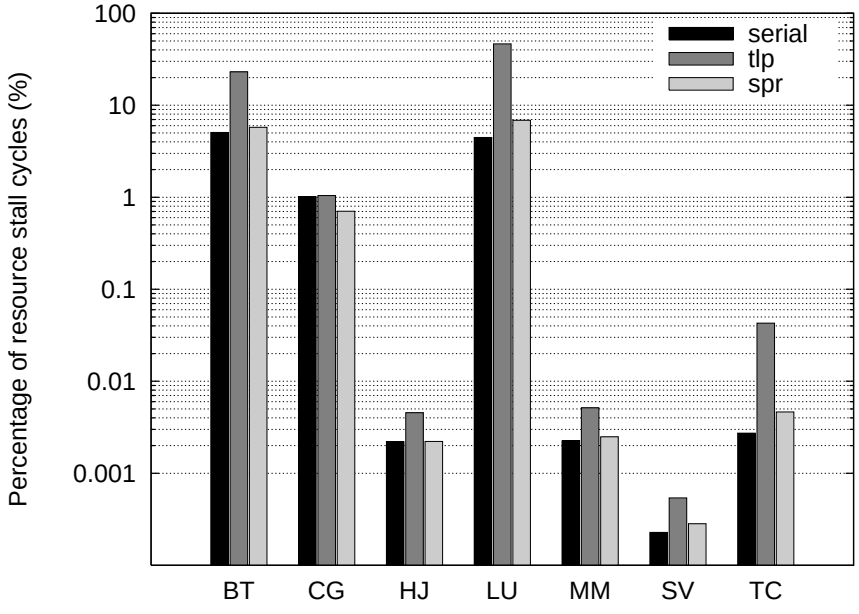
Το Σχήμα 3.4α δείχνει για κάθε σχήμα εκτέλεσης το ποσοστό του συνολικού αριθμού κύκλων που αντιστοιχεί σε κύκλους κατά τους οποίους τα νήματα εργασίας μπλοκαρίστηκαν από τη μονάδα ανάθεσης. Για τα BT και LU οι κύκλοι καθυστέρησης αποτελούν σημαντικό μέρος των συνολικών κύκλων (περίπου 5%). Στα TLP σχήματα αυτό το ποσοστό μεγαλώνει κατά 4.5 φορές για το BT και 10.4 φορές για το LU, υποδεικνύοντας πιθανές συγκρούσεις ανάμεσα στα νήματα. Για τα υπόλοιπα προγράμματα, το ποσοστό των κύκλων καθυστέρησης είναι αμελητέο (μικρότερο από 0.003% για τέσσερα από αυτά). Κατά μέσο όρο το ποσοστό των κύκλων καθυστέρησης στα TLP σχήματα είναι 5.5 φορές μεγαλύτερο σε σχέση με τις σειριακές εκδόσεις. Η μεγαλύτερη αύξηση σημειώνεται στο TC (κατά έναν παράγοντα 15.7) και η μικρότερη στο CG (κατά 1.03).

Από την άλλη πλευρά, η συνεκτέλεση νημάτων προφόρτωσης στα SPR σχήματα δε φαίνεται να αυξάνει σημαντικά τους κύκλους καθυστέρησης του νήματος εργασίας. Κατά μέσο όρο, τα νήματα εργασίας υφίστανται μια αύξηση στους κύκλους καθυστέρησης κατά 1.2 φορές. Αυτά τα αποτελέσματα υποδηλώνουν ότι νήματα με διαφορετικά προφίλ εκτέλεσης μπορούν να συνυπάρχουν καλύτερα από ό,τι νήματα με συμμετρικά προφίλ σε έναν Hyper-threaded επεξεργαστή.

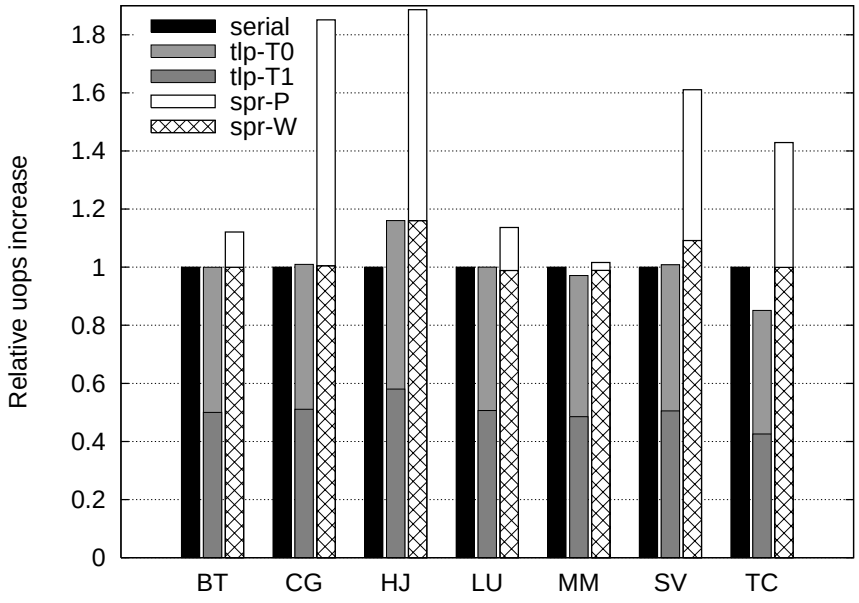
Ολοκληρωθείσες μικροεντολές

Το Σχήμα 3.4β απεικονίζει τη σχετική αύξηση των μικροεντολών που ολοκληρώθηκαν και από τα δύο νήματα στις TLP και SPR εκδόσεις. Επιπλέον, κάθε μπάρα παρουσιάζει τον καταμερισμό των μικροεντολών για τα νήματα που συμμετείχαν στα πλαίσια ενός συγκεκριμένου πολυνηματικού σχήματος. Με *tlp-T0* και *tlp-T1* υποδηλώνονται οι μικροεντολές για το πρώτο και δεύτερο, αντίστοιχα, νήμα εργασίας στο σχήμα TLP. Με *spr-W* και *spr-P* υποδηλώνονται οι μικροεντολές που αντιστοιχούν στο κύριο νήμα και το νήμα προφόρτωσης, αντίστοιχα, στο σχήμα SPR.

Στα TLP σχήματα ο συνολικός αριθμός των μικροεντολών που ολοκληρώθηκαν δεν αυξήθηκε σε σχέση με τη σειριακή εκτέλεση για τα περισσότερα μετροπρογράμματα. Εξαιρέσεις αποτέλεσαν το HJ (αύξηση κατά 16%) και το TC (μείωση κατά 15%). Σε όλες τις περιπτώσεις, το αρχικό φορτίο εργασίας μοιράστηκε ισομερώς και στα δύο νήματα εργασίας, όπως είναι εμφανές από τα σχεδόν όμοια ποσοστά των *tlp-T0* και *tlp-T1*.



(α) *Κύκλοι καθυστέρησης*



(β) *Ολοκληρωθείσες μικροεντολές*

Σχήμα 3.4: *Λόγος των κύκλων καθυστέρησης ως προς το συνολικό αριθμό κύκλων, και αριθμός μικροεντολών που ολοκληρώθηκαν κανονικοποιημένος ως προς τη σειριακή εκτέλεση.*

Στις υλοποιήσεις SPR ο συνολικός αριθμός των μικροεντολών αυξήθηκε κατά 43% κατά μέσο όρο. Όπως φαίνεται από το Σχήμα 3.4β, αυτή η αύξηση αποδίδεται κυρίως στις εντολές που εκτελέστηκαν από τα νήματα προφόρτωσης. Οι εντολές για τα κύρια νήματα εργασίας παρέμειναν

οι ίδιες σε σύγκριση με τη σειριακή εκτέλεση, εκτός από τα HJ και SV, όπου αυξήθηκαν κατά 16% και 9%, αντίστοιχα.

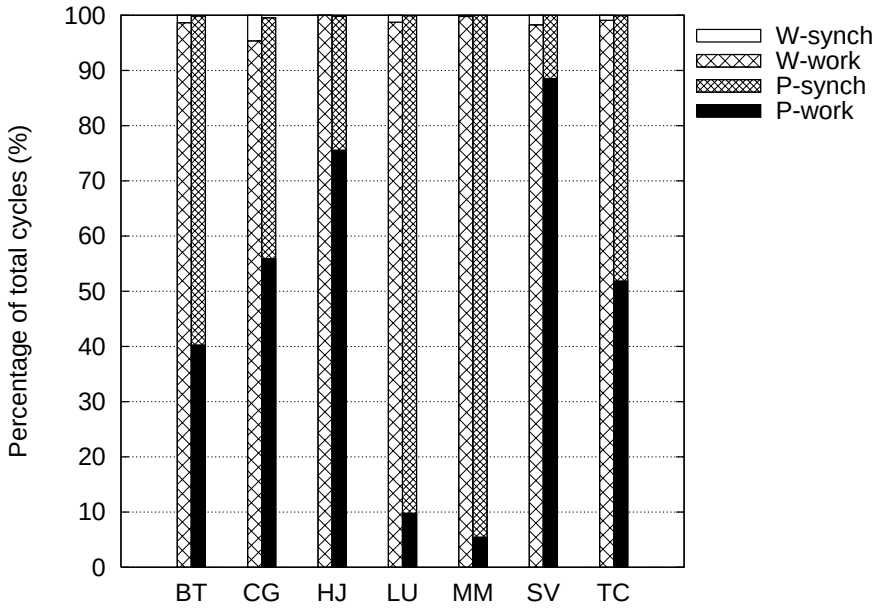
Τον μεγαλύτερο αριθμό μικροεντολών στα νήματα προφόρτωσης είχαν τα μετροπρογράμματα CG, HJ, TC και SV, κατά ένα ποσοστό από 42% έως 84% των εντολών των αντίστοιχων σειριακών υλοποιήσεων. Τη μικρότερη πρόσθετη επιβάρυνση σε μικροεντολές είχαν τα BT, LU και MM, όπου τα νήματα προφόρτωσης χρειάστηκε να εκτελέσουν περίπου το 3%-15% των μικροεντολών της σειριακής έκδοσης. Η πρώτη ομάδα προγραμμάτων χαρακτηρίζεται συνήθως από πολλά δυναμικά στιγμιότυπα παραβατικών εντολών ανάγνωσης, ή από εντολές ανάγνωσης που απαιτούν σχετικά πολλά βήματα για τον υπολογισμό των διευθύνσεών τους. Στο άλλο σύνολο προγραμμάτων υπάρχουν συνήθως λίγα στιγμιότυπα παραβατικών εντολών ανάγνωσης, ή τυχαίνει μία εντολή προφόρτωσης να αρκεί για να καλύψει πολλαπλά στιγμιότυπα (π.χ., εκείνα που βρίσκονται σε γειτονικές γραμμές κρυφής μνήμης), σαν αποτέλεσμα της κανονικότητας των προσπελάσεων στη μνήμη που χαρακτηρίζει τους κώδικες αυτούς.

Προκειμένου να αποκτήσουμε πληρέστερη εικόνα για το προφίλ εκτέλεσης των νημάτων στις SPR εκδόσεις μετρήσαμε για κάθε εφαρμογή το μέρος των κύκλων κατά τους οποίους τα νήματα εργασίας και προφόρτωσης πραγματοποιούν χρήσιμους υπολογισμούς ή περιμένουν στα φράγματα συγχρονισμού. Αυτός ο καταμερισμός κύκλων φαίνεται στο Σχήμα 3.5. Σε αυτό το διάγραμμα οι περίοδοι εργασίας και αναμονής σημειώνονται με *work* και *synch*, αντίστοιχα. Όπως είναι αναμενόμενο, μικρός αριθμός εντολών για τα νήματα προφόρτωσης μεταφράζεται σε μεγάλες περιόδους αναμονής, και αντίστροφα. Κατά μέσο όρο τα νήματα προφόρτωσης περιμένουν στα φράγματα συγχρονισμού κατά το 53% του συνολικού χρόνου εκτέλεσης. Από όλες τις περιπτώσεις, το νήμα προφόρτωσης για το MM περιμένει τον περισσότερο χρόνο (94%) ενώ για το SV τον λιγότερο (12%).

Επιτάχυνση

Το Σχήμα 3.6 παρουσιάζει την επιτάχυνση που πέτυχαν τα πολυνηματικά σχήματα εκτέλεσης σε σχέση με τη σειριακή εκτέλεση. Από μια πρώτη ματιά φαίνεται ότι τα TLP σχήματα ξεπέρασαν σε απόδοση τα SPR σχεδόν σε όλες τις περιπτώσεις. Έξι από τα εφτά μετροπρογράμματα σημείωσαν βελτίωση της απόδοσής τους με την παραλληλοποίηση επιπέδου νημάτων. Η μεγαλύτερη επιτάχυνση που σημειώθηκε ήταν 1.36 (TC) και η μικρότερη 1.03 (MM). Στο HJ σημειώθηκε επιβράδυνση κατά 1.22. Η μέση επιτάχυνση για όλα τα μετροπρογράμματα ήταν 1.14. Το SPR σχήμα κατάφερε να βελτιώσει την απόδοση σε τρεις περιπτώσεις. Η μεγαλύτερη επιτάχυνση που πέτυχε ήταν 1.34 (TC) και η μικρότερη 1.04 (LU). Παρουσίασε τη χειρότερη απόδοση στο CG, όπου σημείωσε επιβράδυνση κατά 1.45.

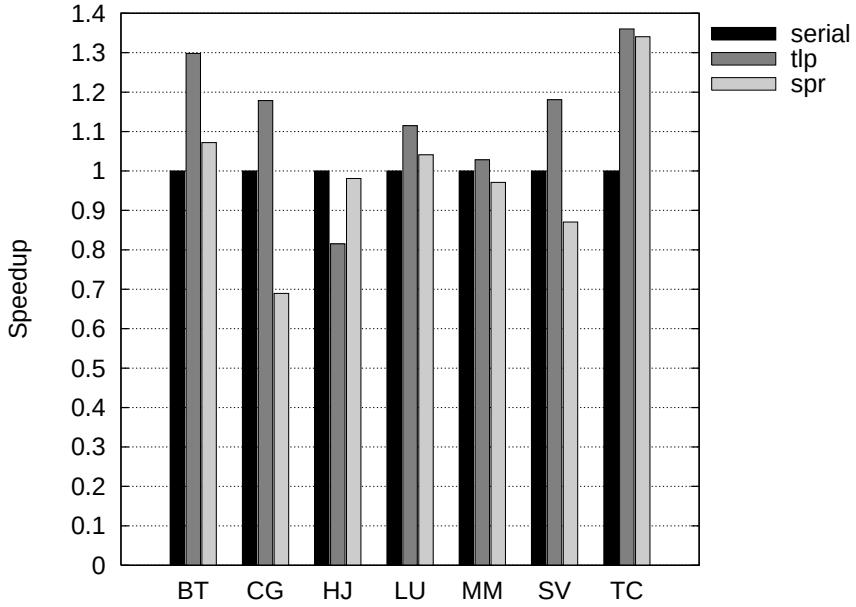
Συνδυάζοντας τα προηγούμενα αποτελέσματα και παρατηρήσεις πάνω στις μετρικές απόδοσης και τους παράγοντες επιτάχυνσης, μπορούμε να συμπεράνουμε ότι το SPR είναι μια



Σχήμα 3.5: *Καταμερισμός κύκλων για τα νήματα εργασίας(W) και τα νήματα προφόρτωσης(P) στο σχήμα SPR.*

υποσχόμενη τεχνική η οποία μπορεί να επιταχύνει την εκτέλεση του κύριου νήματος υπό τις εξής συνθήκες: όταν το νήμα προφόρτωσης εισάγει σχετικά λίγες επιπλέον εντολές, η κάλυψη των αστοχιών στην L2 που επιτυγχάνει είναι ικανοποιητική, και το ποσοστό του χρόνου κατά το οποίο βρίσκεται σε κατάσταση αναμονής, έχοντας απελευθερώσει τους πόρους του, είναι σχετικά μεγάλο. Αυτό συμβαίνει στην περίπτωση των TC, LU και BT. Το MM ανήκει και αυτό στην ίδια κατηγορία, όμως η ήδη καλή τοπικότητα αναφορών του και ο πολύ μικρός αριθμών αστοχιών στην L2 δεν αφήνουν πολλά περιθώρια για περαιτέρω βελτίωση.

Από την άλλη, ακόμα και σε περίπτωση ικανοποιητικής κάλυψης των αστοχιών, αν ο αριθμός των επιπλέον εντολών του νήματος προφόρτωσης είναι σημαντικός, και αν το ποσοστό χρόνου κατά το οποίο ανταγωνίζεται για κοινούς πόρους με το νήμα εργασίας είναι σχετικά μεγάλο, τότε το SPR σχήμα μπορεί να επηρεάσει την απόδοση αρνητικά. Αυτό ισχύει για παράδειγμα στην περίπτωση των SV και CG. Το HJ αποτελεί μια ενδιαφέρουσα εξαίρεση. Σε αυτό το πρόγραμμα το SPR επιτυγχάνει τη μικρότερη κάλυψη αστοχιών στην L2 σε σχέση με όλα τα υπόλοιπα. Επιπλέον, εκτελούνται 90% περισσότερες εντολές σε σχέση με τη σειριακή έκδοση και το νήμα προφόρτωσης κάνει χρήσιμη δουλειά για τουλάχιστον 75% του χρόνου εκτέλεσης. Παρόλα αυτά, η υποβάθμιση της απόδοσης είναι πολύ μικρή. Αυτό μάς παρακινεί να εξετάσουμε λιγότερο επιθετικά SPR σχήματα όπου θα μπορούσαμε να είμαστε πιο συντηρητικοί στην επιλογή των παραβατικών εντολών ανάγνωσης ή περισσότερο διατεθειμένοι στο να υποστούμε κάποιες από τις επιτυχώς καλυφθείσες αστοχίες προκειμένου να ευνοήσουμε συνολικά την απόδοση. Σε



Σχήμα 3.6: *Επιτάχυνση για τα σχήματα TLP και SPR.*

κάθε περίπτωση, θεωρούμε ότι η κάλυψη των αστοχιών και η επιβάρυνση που εισάγει το νήμα προφόρτωσης συνιστούν έναν ενδιαφέροντα συμβιβασμό που αξίζει περαιτέρω διερεύνησης.

Όσον αφορά το TLP, οι μετρήσεις μας συμφωνούν με τα περισσότερα προηγούμενα αποτελέσματα της σχετικής βιβλιογραφίας, σύμφωνα με τα οποία το SMT ευνοεί πιο πολύ κώδικες μη υψηλά βελτιστοποιημένους για τοπικότητα ή αποδοτική χρήση των μονάδων εκτέλεσης. Το MM ανήκει σε αυτή την κατηγορία και επιβεβαιώνει τον ισχυρισμό αυτό. Τα TC, BT και SV, από την άλλη, επιδεικνύουν κακή τοπικότητα και υψηλά ποσοστά αστοχίας, και έτσι δείχνουν να επωφελούνται περισσότερο από το SMT. Άλλοι σημαντικοί παράγοντες για την αποτελεσματικότητα των μεθόδων TLP στο SMT είναι ο βαθμός της αμοιβαίας παρεμβολής μεταξύ των νημάτων στην κρυφή μνήμη (έντονη στα HJ και MM) και το ποσό των επιπλέον εντολών εξαιτίας του κόστους παραλληλοποίησης ή του συγχρονισμού (HJ). Οι συγκρούσεις στην κρυφή μνήμη σε αυτές τις περιπτώσεις δείχνουν ότι, παρόλο που τα περισσότερα προγράμματα μπορούν να παραλληλοποιηθούν βάση TLP σχημάτων με σχετικά ευθύ τρόπο, θα πρέπει να υποστούν λεπτομερείς ρυθμίσεις όταν προορίζονται για επεξεργαστές SMT προκειμένου να εκτελεστούν αποδοτικά. Τεχνικές επεξεργασίας κατά μπλοκ ή άλλες τεχνικές μείωσης των συγκρούσεων στην κρυφή μνήμη θα μπορούσαν να χρησιμοποιηθούν για αυτόν τον σκοπό.

Ανάλυση του δυναμικού μίγματος εντολών

Ο Πίνακας 3.3 παρουσιάζει τα ποσοστά χρησιμοποίησης των πλέον απασχολούμενων μονάδων εκτέλεσης του επεξεργαστή, για κάθε ένα από τα εξεταζόμενα μετροπρογράμματα. Η πρώτη στήλη (*serial*) παρουσιάζει τα αποτελέσματα για τις σειριακές εκδόσεις. Η δεύτερη στήλη (*tlp*) παρουσιάζει τα αποτελέσματα για ένα εκ των δύο νημάτων εργασίας του TLP σχήματος. Σε αυτήν την περίπτωση, όλα τα νήματα εκτελούν σχεδόν το ίδιο φορτίο εργασίας και ως εκ τούτου τα ποσοστά χρησιμοποίησης είναι σχεδόν τα ίδια για κάθε νήμα. Η τρίτη στήλη (*spr*) παρουσιάζει στατιστικά για το νήμα προφόρτωσης στην SPR έκδοση (το κύριο νήμα σε αυτήν την περίπτωση έχει το ίδιο προφίλ όπως και στη σειριακή). Όλα τα ποσοστά στον πίνακα εκφράζουν το μέρος των συνολικών εντολών του εκάστοτε νήματος που χρησιμοποίησαν μια συγκεκριμένη μονάδα του επεξεργαστή.

Τα στατιστικά συλλέχθηκαν κάνοντας σκιαγράφηση προφίλ στα αρχικά εκτελέσιμα των εφαρμογών με το εργαλείο δυναμικής παρεμβολής κώδικα Pin [Luk 05]. Η τεχνική της *παρεμβολής κώδικα* (code instrumentation) επιτρέπει την εισαγωγή πρόσθετου κώδικα σε μια εφαρμογή προκειμένου να παρατηρήσουμε τη συμπεριφορά της. Μπορεί να πραγματοποιηθεί στατικά κατά τη μεταγλώττιση ή και δυναμικά κατά το χρόνο εκτέλεσης, όπως συμβαίνει στην περίπτωσή του Pin. Με το συγκεκριμένο εργαλείο ο χρήστης μπορεί να ορίσει επιθυμητές ενέργειες κάθε φορά που εκτελείται μια εντολή. Εμείς το χρησιμοποιούμε για να καταγράψουμε τις διαφορετικές εντολές που εκτελέστηκαν και να εξάγουμε έτσι το δυναμικό μίγμα εντολών. Γνωρίζοντας τις μονάδες όπου δρομολογούνται συγκεκριμένοι τύποι εντολών [Int 07], μπορούμε τελικά να υπολογίσουμε το ποσοστό χρησιμοποίησης των μονάδων του επεξεργαστή. Το Σχήμα 3.7 απεικονίζει τις βασικές μονάδες εκτέλεσης του επεξεργαστή Xeon μαζί με τις θύρες έκδοσης οι οποίες οδηγούν τις εντολές σε αυτές.

Αυτό που παρατηρούμε εκ πρώτης όψεως είναι ότι στις TLP υλοποιήσεις δεν αλλάζει το δυναμικό μίγμα εντολών σε σχέση με τη σειριακή εκτέλεση. Αυτό φυσικά δεν ισχύει στην περίπτωση του SPR. Για το νήμα προφόρτωσης, όχι μόνο το δυναμικό μίγμα αλλά ακόμα και ο αριθμός εντολών διαφέρουν σημαντικά από τους αντίστοιχους του νήματος εργασίας. Επιπλέον, διαφορετικά μοτίβα προσπελάσεων μνήμης απαιτούν διαφορετική προσπάθεια για τον υπολογισμό των διευθύνσεων, και προφανώς, διαφορετικό αριθμό εντολών.

Στις TLP υλοποιήσεις, τα BT, SV, MM, LU και CG επιδεικνύουν υψηλή χρησιμοποίηση των μονάδων κινητής υποδιαστολής. Η χρησιμοποίηση των αριθμητικών-λογικών μονάδων (arithmetic logic units – ALUs) είναι επίσης σημαντική. Όπως είδαμε στην ενότητα 2.5.1, τα μίγματα εντολών $fadd - mul$ μπορούν να συνεκτελεστούν αποδοτικά σε έναν Hyper-threaded επεξεργαστή για χαμηλά και μεσαία επίπεδα ILP, δίνοντας κέρδος στην απόδοση στις περισσότερες περιπτώσεις. Από την άλλη, ροές εντολών που χρησιμοποιούν τις ALUs, όπως οι *iadd*, δεν υφίστανται επιβράδυνση αλλά ούτε και επωφελούνται από το Hyper-threading. Γενικά, η

	Μονάδα εκτέλεσης	***serial***	***tlp***	***spr***
BT	ALU0+ALU1:	8.06%	8.06%	12.06%
	FP_ADD:	17.67%	17.67%	0.00%
	FP_MUL:	22.04%	22.04%	0.00%
	FP_MOVE:	10.51%	10.51%	0.00%
	MEM_LOAD:	42.70%	42.70%	44.70%
	MEM_STORE:	16.01%	16.01%	42.94%
CG	ALU0+ALU1:	28.04%	23.95%	49.93%
	FP_ADD:	8.83%	7.49%	0.00%
	FP_MUL:	8.86%	7.53%	0.00%
	FP_MOVE:	17.05%	14.05%	0.00%
	MEM_LOAD:	36.51%	45.71%	19.09%
	MEM_STORE:	9.50%	8.51%	9.54%
HJ	ALU0+ALU1:	78.61%	78.65%	79.81%
	MEM_LOAD:	40.13%	40.09%	40.07%
	MEM_STORE:	0.91%	0.91%	0.06%
LU	ALU0+ALU1:	38.84%	38.84%	38.16%
	FP_ADD:	11.15%	11.15%	0.00%
	FP_MUL:	11.15%	11.15%	0.00%
	MEM_LOAD:	49.24%	49.24%	38.40%
	MEM_STORE:	11.24%	11.24%	22.78%
MM	ALU0+ALU1:	27.06%	26.26%	37.56%
	FP_ADD:	11.70%	11.82%	0.00%
	FP_MUL:	11.70%	11.82%	4.13%
	MEM_LOAD:	38.76%	27.00%	58.30%
	MEM_STORE:	12.07%	12.02%	20.75%
SV	ALU0+ALU1:	25.05%	27.58%	24.03%
	FP_ADD:	13.32%	12.90%	0.00%
	FP_MUL:	13.32%	12.90%	0.00%
	MEM_LOAD:	51.15%	46.15%	53.77%
	MEM_STORE:	17.23%	13.27%	0.34%
TC	ALU0+ALU1:	67.14%	67.21%	79.62%
	MEM_LOAD:	40.72%	41.47%	21.93%
	MEM_STORE:	8.55%	8.52%	0.19%

Πίνακας 3.3: *Ποσοστά χρησιμοποίησης των μονάδων εκτέλεσης του επεξεργαστή για τα νήματα των διαφόρων σχημάτων εκτέλεσης.*

συνεκτέλεση εντολών κινητής υποδιαστολής είναι πιο αποδοτική από ό,τι η συνεκτέλεση εντολών ακεραίων. Από τα παραπάνω μετροπρογράμματα το BT είναι αυτό στο οποίο κυριαρχούν προσθέσεις και πολλαπλασιασμοί κινητής υποδιαστολής, έχοντας ταυτόχρονα χαμηλά επίπεδα χρησιμοποίησης των ALUs. Ίσως, επομένως, αυτό το μίγμα εντολών μαζί με το χαμηλό ILP του

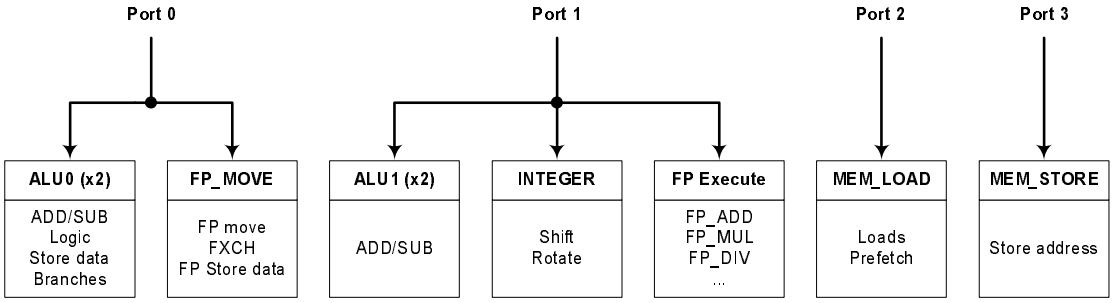
προγράμματος να είναι ανάμεσα στις αιτίες που εξηγούν την επιτάχυνση για αυτό το πρόγραμμα.

Άλλα μετροπρογράμματα όπως το MM ή το LU, όπου αυτό το μίγμα εντολών ήταν διαφορετικό, αποκόμισαν μικρότερα οφέλη. Το MM, για παράδειγμα, χαρακτηρίζεται από μεγάλο αριθμό λογικών εντολών, που αγγίζει το 26% των συνολικών εντολών, τόσο στη σειριακή όσο στην TLP έκδοση. Αυτό εξηγείται από την υλοποίηση της κατά μπλοκ διάταξης των πινάκων για την οποία χρησιμοποιήθηκαν δυαδικές μάσκες [Athanasaki 04]. Παρόλο που ο επεξεργαστικός πυρήνας του Xeon διαθέτει δύο μονάδες ALU (διπλής ταχύτητας), μόνο μία από αυτές (ALU0) μπορεί να διαχειριστεί λογικές πράξεις. Ως εκ τούτου, η ταυτόχρονη ζήτηση για αυτή τη μονάδα θα οδηγήσει σε σειριοποίηση των αντίστοιχων εντολών χωρίς να πετύχει κάποια επικάλυψη. Σε σχέση με το MM, το LU έχει μεγαλύτερα ποσοστά χρησιμοποίησης των ALUs. Σε αυτήν την περίπτωση όμως οι εντολές μπορούν να εκτελεστούν και στις δύο ALUs και έτσι πιστεύουμε ότι κατανέμονται ομοιόμορφα σε αυτές. Αυτό, μαζί με τα χαμηλότερα επίπεδα ILP που υπάρχουν στο LU, ίσως να εξηγούν σε κάποιο βαθμό την καλύτερη απόδοση του LU. Εξαίρεση στα παραπάνω αποτελεί το TC. Κυριαρχείται εξ' ολοκλήρου από πράξεις ακεραίων αλλά αυτό δε φαίνεται να αποτελεί εμπόδιο για την απόδοσή του αφού σημειώνει μεγάλη επιτάχυνση. Πιστεύουμε ωστόσο ότι το Hyper-threading σε αυτή την περίπτωση εκμεταλλεύεται περισσότερο την κακή τοπικότητα του προγράμματος και όχι τόσο το μίγμα εντολών.

Όπως ήταν αναμενόμενο, στα σχήματα SPR τα νήματα προφόρτωσης εκτελούν εντολές οι οποίες, εκτός από τις μονάδες μνήμης, χρησιμοποιούν αποκλειστικά τις ALUs. Εκτελούν τις εντολές αυτές τουλάχιστον κατά το ίδιο ποσοστό με τα αντίστοιχα νήματα εργασίας. Αυτό υποδηλώνει ότι όταν ο αριθμός των εντολών τους είναι της ίδιας τάξης μεγέθους με αυτόν των νημάτων εργασίας, τότε γίνεται έντονη χρήση των ALUs. Όπως συζητήσαμε, η υλοποίηση Hyper-threading του Xeon δεν είναι αποδοτική στο να επικαλύπτει εντολές που εκτελούνται στις ALUs, ενώ σημειώνονται ιδιαίτερες επιβραδύνσεις όταν ο αριθμός των ταυτόχρονα εκτελούμενων εντολών μνήμης είναι μεγάλος. Αυτά τα στοιχεία ίσως εξηγούν σε κάποιο βαθμό την αδυναμία του SPR να επιταχύνει εφαρμογές όπως το CG ή το HJ. Το TC αποτελεί και πάλι εξαίρεση, και η βελτίωση του χρόνου εκτέλεσής του πιστεύουμε ότι αποδίδεται πιο πολύ στην ήδη κακή τοπικότητά του και την επιτυχή κάλυψη των αστοχιών με το SPR.

3.6 Συμπεράσματα και Μελλοντικές Κατευθύνσεις

Στις προηγούμενες ενότητες παρουσιάσαμε αποτελέσματα από την αξιολόγηση μιας πραγματικής αρχιτεκτονικής SMT, της τεχνολογίας Hyper-threading, για μια σειρά από υπολογιστικούς πυρήνες και εφαρμογές. Σε αυτά τα προγράμματα εφαρμόσαμε τόσο τεχνικές καταμερισμού φορτίου για να εκμεταλλευτούμε τον παραλληλισμό επιπέδου νημάτων, όσο και τεχνικές υποθετικής προεκτέλεσης για να αντιμετωπίσουμε την καθυστέρηση πρόσβασης στη μνήμη. Η



Σχήμα 3.7: *Θύρες έκδοσης εντολών και βασικές μονάδες εκτέλεσης του επεξεργαστή Xeon [Int 07].*

αξιολόγηση βασίστηκε σε αποτελέσματα χρόνων εκτέλεσης καθώς και σε στατιστικά από τους μετρητές απόδοσης και την ανάλυση του δυναμικού μίγματος εντολών των προγραμμάτων. Τα αποτελέσματα που πήραμε κατέδειξαν τα όρια της τεχνολογίας Hyper-threading στην επίτευξη υψηλής επίδοσης για εφαρμογές παραλληλοποιημένες σύμφωνα με το TLP ή SPR μοντέλο.

Η υλοποίηση της υποθετικής προεκτέλεσης σε ένα πραγματικό μηχάνημα SMT αποτέλεσε πρόκληση, καθώς δεν ήταν η προφανής επιλογή για τα προγράμματα που χρησιμοποιήσαμε, και επιπλέον, δεν υποστηρίζεται από τους Hyper-threaded επεξεργαστές με ειδικό υλικό, όπως συμβαίνει στις περισσότερες σχετικές εργασίες. Έπρεπε να λάβουμε υπόψη συνεπώς διάφορους συμβιβασμούς και να αξιολογήσουμε διαφορετικές επιλογές προκειμένου να κάνουμε το SPR όσο το δυνατόν πιο αποδοτικό. Σε τρία από τα εφτά μετροπρογράμματα το SPR πέτυχε επιτάχυνση μεταξύ 4% και 34%, ενώ ήταν συγκρίσιμο με τη σειριακή εκτέλεση στα περισσότερα από τα υπόλοιπα. Τα νήματα προφόρτωσης κατάφεραν να καλύψουν 39%-96% από τις αστοχίες στην L2 των κύριων νημάτων, εισάγοντας στην εκτέλεση 43% περισσότερες εντολές κατά μέσο όρο. Μπορούμε να πούμε ότι η τεχνική του SPR μπορεί να είναι αποδοτική όταν η κάλυψη των αστοχιών είναι μεγάλη, οι πρόσθετες εντολές του νήματος προφόρτωσης είναι λίγες, και το ίδιο το νήμα περιμένει στα σημεία συγχρονισμού για μεγάλο διάστημα, έχοντας όμως απελευθερώσει τους πόρους του. Προκειμένου να μεγαλώσουμε την ακρίβεια της προφόρτωσης σε μερικούς κώδικες χρειάστηκε να εισάγουμε σημαντικό αριθμό εντολών στον κώδικα του νήματος προφόρτωσης. Ακόμα και σε αυτές τις περιπτώσεις, το SPR φαίνεται υποσχόμενο αφού επιβραδύνει οριακά την απόδοση. Αυτό μάς προτρέπει να διερευνήσουμε λιγότερο επιθετικά σχήματα SPR.

Με την παραλληλοποίηση επιπέδου νημάτων, έξι από τα εφτά προγράμματα βελτίωσαν τους χρόνους εκτέλεσης από 3% έως 36%. Κατά μέσο όρο το TLP επιτάχυνε τις εφαρμογές σχεδόν κατά 14%. Τα αποτελέσματά μας επιβεβαίωσαν ό,τι κατέδειξαν οι προηγούμενες ερευνητικές εργασίες μέχρι τώρα, δηλαδή ότι το SMT ευνοεί περισσότερο κώδικες που είναι μέτρια ή καθόλου βελτιστοποιημένοι για τοπικότητα και αποδοτική χρήση των πόρων. Επιπρόσθετα, οι αμοιβαίες

συγκρούσεις των νημάτων στην διαμοιραζόμενη κρυφή μνήμη μπορεί να οδηγήσουν σε αξιόλογη υποβάθμιση της απόδοσης. Για αυτό το λόγο θεωρούμε ότι η μεταφορά προγραμμάτων παραλληλοποιημένων με το TLP μοντέλο σε ένα περιβάλλον SMT δε θα πρέπει να γίνεται χωρίς επιπλέον προγραμματιστική προσπάθεια, αν στοχεύουμε στην απόδοση.

Σαν μελλοντική δουλειά, θα είχε ενδιαφέρον καταρχάς να ασχοληθούμε με την αυτοματοποίηση της διαδικασίας υλοποίησης του σχήματος SPR, με ευρύτερο στόχο πολυνηματικές και πολυπύρηνες πλατφόρμες με μοιραζόμενα επίπεδα κρυφής μνήμης. Αυτό απαιτεί την ανάπτυξη αποδοτικών μηχανισμών για τον εντοπισμό των παραβατικών εντολών καθώς και για την απομόνωση του ελάχιστου απαραίτητου κώδικα για τα νήματα προφόρτωσης. Θα είχε επίσης ενδιαφέρον να εξετάσουμε και συστήματα χρόνου εκτέλεσης που θα επέτρεπαν τη συνεχή δημιουργία και αποδοτική διαχείριση των νημάτων προφόρτωσης, υπό τη μορφή εργασιών επιπέδου χρήστη (user-level tasks), σε μια προσπάθεια να προσεγγίσουμε το μοντέλο εκτέλεσης SPR σχημάτων που υποθέτουν τέτοιου είδους υποστήριξη από το υλικό.

Επιπλέον, σκοπεύουμε να εξετάσουμε το ενδεχόμενο εναλλακτικής χρήσης TLP και SPR σχημάτων ταυτόχρονα στην ίδια εφαρμογή για ακόμα μεγαλύτερη απόδοση, καθώς και την εφαρμογή του SPR σε περισσότερες εφαρμογές οι οποίες είναι εγγενώς σειριακές. Θα αξιολογήσουμε επίσης την κλιμακωσιμότητα των TLP και SPR σε πλατφόρμες με πολλαπλούς επεξεργαστές SMT (π.χ. SMPs από CMPs από SMTs), αλλά και στην τελευταία υλοποίηση του Hyper-threading στην μικροαρχιτεκτονική Nehalem.

Ιδιαίτερο ενδιαφέρον έχει η διερεύνηση εναλλακτικών μεθόδων παραλληλοποίησης και προγραμματιστικών μοντέλων με σκοπό τη μείωση του ανταγωνισμού για κοινούς πόρους στους SMTs. Η ανίχνευση και η ταυτόχρονη δρομολόγηση φάσεων εκτέλεσης στη διάρκεια ζωής μιας εφαρμογής με συμπληρωματικές απαιτήσεις για πόρους θα αποτελούσε μια πιθανή κατεύθυνση. Σε κάθε περίπτωση, η ανάθεση διαφορετικού τύπου υπολογισμών σε διαφορετικά παράλληλα νήματα λογισμικού αποτελεί σημαντική πρόκληση για εφαρμογές που προορίζονται για πλατφόρμες SMT.

ΚΕΦΑΛΑΙΟ 4

Αποδοτικός Συγχρονισμός σε Επεξεργαστές SMT

4.1 Εισαγωγή

Όπως είδαμε αναλυτικά στο Κεφάλαιο 2, σε έναν επεξεργαστή με τεχνολογία Hyper-threading όλοι σχεδόν οι πόροι διαμοιράζονται ανάμεσα στα δύο νήματα υλικού. Σε ένα τέτοιο περιβάλλον υψηλής ανταγωνιστικότητας, ο τρόπος με τον οποίον υλοποιούνται οι *πρωτογενείς λειτουργίες συγχρονισμού* (synchronization primitives) για απλές ή πιο σύνθετες μεθόδους συγχρονισμού, αποτελεί καθοριστικό παράγοντα για την τελική απόδοση των εφαρμογών και δεν είναι σαφής διαδικασία.

Πρωτογενείς λειτουργίες συγχρονισμού που βασίζονται σε βρόχους περιδίνησης έχουν χρησιμοποιηθεί ευρέως κατά το παρελθόν σε παραδοσιακά πολυεπεξεργαστικά συστήματα, εξαιτίας της απλότητας στην υλοποίησή τους και της υψηλής αποκρισιμότητάς τους, σαν συνέπεια της λειτουργίας τους εξ' ολοκλήρου σε επίπεδο χρήστη. Σε έναν βρόχο περιδίνησης γίνεται επαναληπτικός έλεγχος της τιμής μιας μεταβλητής συγχρονισμού προκειμένου να διαπιστωθεί αν είναι αληθής κάποια συνθήκη. Συνήθως αυτή η συνθήκη σχετίζεται με τη διαθεσιμότητα κάποιου πόρου (π.χ. μιας μεταβλητής-κλειδιού). Στο παράδειγμα του Σχήματος 4.1, η μεταβλητή συγχρονισμού βρίσκεται στη θέση μνήμης `sync_var` και διαβάζεται σε κάθε επανάληψη του βρόχου μέχρι να ανανεωθεί από κάποιο νήμα στην επιθυμητή τιμή (που βρίσκεται αποθηκευμένη στον καταχωρητή `eax`).

Σε έναν επεξεργαστή SMT η εκτέλεση λειτουργιών συγχρονισμού που βασίζονται σε βρόχους περιδίνησης μπορεί να επιφέρει σημαντική μείωση της απόδοσης, ειδικά όταν η εκτέλεση

```
wait_loop: cmp eax, sync_var
           jne wait_loop
```

Σχήμα 4.1: *Τυπικό παράδειγμα βρόχου περιδίνησης.*

της εφαρμογής περιλαμβάνει μακρές περιόδου αναμονής για ένα ή περισσότερα νήματα. Σε μοντέρνους επεξεργαστές με εκτέλεση εκτός σειράς και πρόβλεψη διακλάδωσης ένας βρόχος περιδίνησης μπορεί να ξεδιπλωθεί δυναμικά πολλές φορές από τις μονάδες δρομολόγησης του επεξεργαστή. Αυτό συμβαίνει αφενός διότι η διακλάδωση μπορεί να προβλεφθεί εύκολα, και αφετέρου, διότι δεν υπάρχουν εξαρτήσεις δεδομένων ανάμεσα σε διαδοχικές επαναλήψεις του βρόχου. Με αυτό τον τρόπο το νήμα που περιδινείται, παρόλο που δεν πραγματοποιεί κάποια χρήσιμη δουλειά, εισάγει έναν σημαντικό αριθμό από εντολές στη σωλήνωση οι οποίες ανταγωνίζονται με το ομότιμο νήμα για επεξεργαστικούς πόρους. Επιπλέον, κατά την ενημέρωση της μεταβλητής συγχρονισμού, όλες οι εντολές του περιδινούμενου νήματος που εκδόθηκαν υποθετικά (λόγω πρόβλεψης διακλάδωσης) αλλά δεν έχουν ακόμη ολοκληρωθεί, πρέπει να απορριφθούν, γεγονός που συνεπάγεται χαμένους κύκλους λόγω της εκκαθάρισης της σωλήνωσης.

Για την αντιμετώπιση τέτοιων ζητημάτων, και με την ταυτόχρονη απαίτηση για υψηλή απόδοση στο συγχρονισμό, οι Tullsen και άλλοι πρότειναν επεκτάσεις στο υλικό για να υποστηρίζεται ο *λεπτομερής συγχρονισμός* (fine-grain synchronization) σε επεξεργαστές SMT [Tullsen 99]. Σε αυτό το είδος συγχρονισμού, ο λόγος των λειτουργιών συγχρονισμού σε σχέση με τις συνολικές λειτουργίες της παράλληλης εργασίας είναι σχετικά υψηλός, δηλαδή ο συγχρονισμός είναι πολύ συχνός. Στο μοντέλο SMT που προσομοίωσαν οι συγγραφείς σχεδίασαν τον επεξεργαστή ώστε να παρέχει κατευθείαν στο λογισμικό λειτουργίες *απόκτησης κλειδιού* (lock acquisition) και *απελευθέρωσης κλειδιού* (lock release) [1]. Τοποθέτησαν στον επεξεργαστή μια μικρή συσχετιστική μνήμη, το *"lock-box"*, η οποία διαθέτει μια εγγραφή για κάθε νήμα υλικού. Κάθε εγγραφή περιέχει τη διεύθυνση της μεταβλητής-κλειδιού, τη διεύθυνση της εντολής `acquire` που επιχειρεί να αποκτήσει το κλειδί, καθώς και ένα bit εγκυρότητας (valid bit). Όταν ένα νήμα αποτυγχάνει να αποκτήσει ένα κλειδί, η διεύθυνση του κλειδιού καθώς και η διεύθυνση της εντολής `acquire` αποθηκεύονται στην εγγραφή που αντιστοιχεί στο νήμα. Το νήμα τότε μπλοκάρει και αποσύρεται από τον επεξεργαστή, απελευθερώνοντας όλους τους πόρους που χρησιμοποιούσε.

Όταν ένα νήμα απελευθερώνει ένα κλειδί, τότε το υλικό πραγματοποιεί σύγκριση της διεύθυνσης του κλειδιού με τις διευθύνσεις των κλειδιών που είναι εγγεγραμμένα στο lock-box. Αν βρεθεί ένα νήμα το οποίο είναι μπλοκαρισμένο σε αυτό το κλειδί, τότε το υλικό αφυπνίζει το νήμα και το επανεκκινεί από την εντολή `acquire` όπου είχε μπλοκάρει. Αυτός ο μηχανισμός επιτρέπει τη γνωστοποίηση της κατάστασης του κλειδιού ανάμεσα στα νήματα μέσα σε πολύ

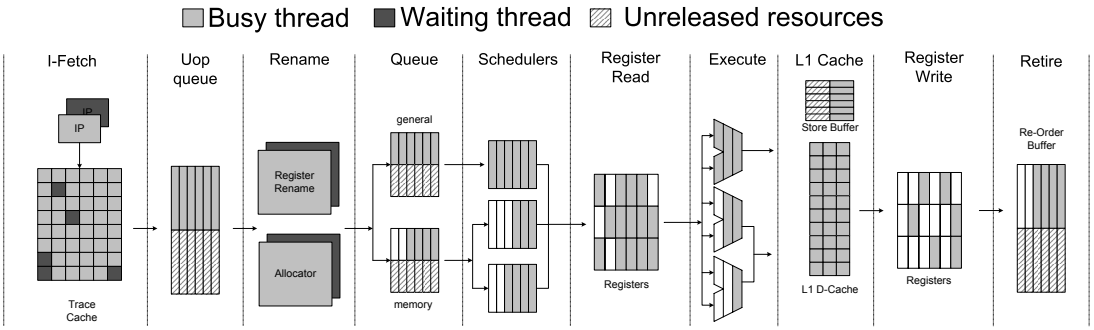
[1] Στην ουσία πρόκειται για λειτουργίες που σηματοδοτούν το κλείδωμα και το ξεκλείδωμα μιας κρίσιμης περιοχής

λίγους κύκλους, και επιπλέον ελαχιστοποιεί τη σπατάλη πόρων του επεξεργαστή.

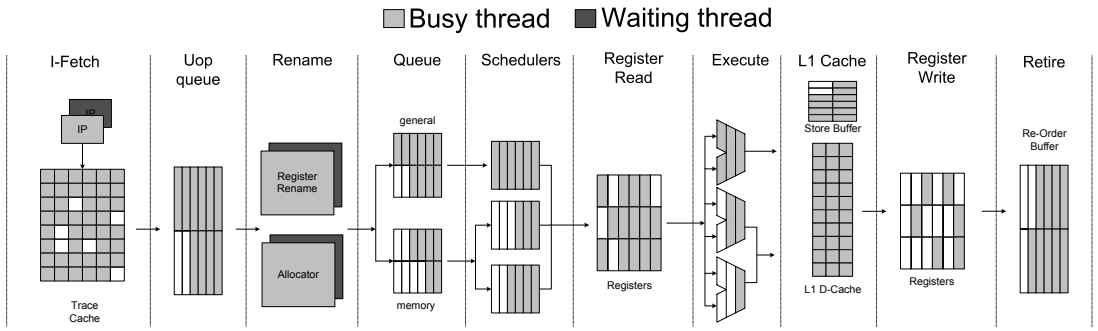
Ένας επεξεργαστής με τεχνολογία Hyper-threading δεν παρέχει τέτοιο μηχανισμό για συγχρονισμό χαμηλής καθυστέρησης και χαμηλής κατανάλωσης πόρων, κατευθείαν στο επίπεδο υλικού. Αν στοχεύσουμε στην εξοικονόμηση πόρων, τότε υπάρχουν μια σειρά από τεχνικές οι οποίες θα μπορούσαν να χρησιμοποιηθούν προκειμένου να μετριάσουμε την υπερβολική κατανάλωση πόρων που έχουν οι βρόχοι περιδίνησης, οι οποίες όμως δύσκολα συγκρίνονται μαζί τους όσον αφορά την χαμηλή καθυστέρηση που αυτοί παρέχουν. Για παράδειγμα, θα μπορούσε να χρησιμοποιήσει κανείς πρωτογενείς λειτουργίες που στηρίζονται στη μεσολάβηση του λειτουργικού συστήματος προκειμένου να διαχειριστεί μακρές περιόδους αναμονής για κάποια συνθήκη. Με αυτό τον τρόπο, το νήμα που περιμένει θα παραχωρούσε το λογικό του επεξεργαστή στο λειτουργικό και θα απελευθέρωνε όλους τους πόρους του (μετάβαση από Multi-Threaded σε Single-Threaded τρόπο λειτουργίας), αλλά η ειδοποίησή του και η επανεκκίνησή του θα ήταν ακριβή από άποψη κύκλων, εξαιτίας της επίκλησης του χρονοδρομολογητή του λειτουργικού.

Μια πιθανή βελτίωση στην αρχική σκέψη θα ήταν να χαλαρώσουμε την περιδίνηση. Η λογική πίσω από αυτό είναι ότι το νήμα που περιμένει εκτελεί το βρόχο περιδίνησης πολύ γρηγορότερα από ό,τι χρειάζεται, δηλαδή από τον χρόνο που θέλει ο δίαυλος μνήμης να πραγματοποιήσει μια απλή εγγραφή στη μνήμη. Η Intel συνιστά τη χρήση της εντολής PAUSE για αυτό το σκοπό [Int 01]. Αυτή η εντολή εισάγει μια μικρή παύση στην εκτέλεση του βρόχου με αποτέλεσμα να μη δρομολογούνται στη σωλήνωση εντολές του αντίστοιχου νήματος για όσο διαρκεί η παύση, και έτσι να αποφεύγεται η υπερβολική κατανάλωση πόρων. Σε αυτή την περίπτωση πρόκειται για πόρους που διαμοιράζονται *δυναμικά* ανάμεσα στα δύο νήματα ενός Hyper-threaded επεξεργαστή, όπως είναι οι μονάδες εκτέλεσης, οι κρυφές μνήμες, και το υλικό για τη φόρτωση, αποκωδικοποίηση, δρομολόγηση, και ολοκλήρωση των εντολών (βλ. Πίνακα 2.1).

Με αυτό τον τρόπο το κόστος της περιδίνησης μειώνεται, όμως δεν εξαλείφεται πλήρως. Ο λόγος είναι ότι μερικές δομές του επεξεργαστή που σχεδιάστηκαν να διαχωρίζονται *στατικά* ανάμεσα στα δύο νήματα, δεν απελευθερώνονται με τη χρήση της PAUSE. Τέτοιες δομές είναι κυρίως ενδιάμεσες ουρές απομόνωσης (buffering queues) όπως ουρές μικροεντολών, οι απομονωτές αναγνώσεων-εγγραφών (load-store queues) και ο απομονωτής αναδιάταξης εντολών (reorder buffer). Σε μια Hyper-threaded αρχιτεκτονική οι ουρές αυτές εγγυώνται την ανεξάρτητη ροή εντολών μέσα στη σωλήνωση για τα δύο νήματα. Στον Multi-Threaded τρόπο λειτουργίας της αρχιτεκτονικής οι ουρές είναι διαχωρισμένες ώστε κάθε νήμα να μπορεί να χρησιμοποιήσει το πολύ μισές από τις εγγραφές τους. Με αυτόν τον τρόπο, και επειδή η πρόσβαση στις εγγραφές κάθε νήματος εναλλάσσεται σε κάθε κύκλο, εξασφαλίζεται η δικαιοσύνη ανάμεσα στα



Σχήμα 4.2: *Κατανομή των πόρων ενός Hyper-threaded επεξεργαστή ανάμεσα σε ένα κύριο νήμα εργασίας (busy thread) και ένα νήμα που εκτελεί βρόχο περιδίνησης υλοποιημένο με την εντολή PAUSE (waiting thread).*



Σχήμα 4.3: *Κατανομή των πόρων ενός Hyper-threaded επεξεργαστή ανάμεσα σε ένα κύριο νήμα εργασίας (busy thread) και ένα νήμα που εκτελεί βρόχο περιδίνησης υλοποιημένο με την εντολή HALT (waiting thread).*

νήματα και αποφεύγεται ο μακρόχρονος αποκλεισμός κάποιου νήματος από τη χρήση κοινών πόρων καθώς και τα αδιέξοδα (deadlocks). Όταν ένα νήμα εκτελεί την PAUSE συνεχίζει να καταλαμβάνει το μερίδιό του σε αυτές τις ουρές. Κατά συνέπεια, το νήμα που εκτελεί το βρόχο περιδίνησης εξακολουθεί να δεσμεύει στην ουσία μέρος των πόρων του επεξεργαστή οι οποίοι θα μπορούσαν να φανούν χρήσιμοι στο δεύτερο νήμα για να εκτελεστεί αποδοτικότερα.

Η κατανομή των πόρων ενός Hyper-threaded επεξεργαστή όταν ένα κύριο νήμα εργασίας (busy thread) εκτελείται μαζί με κάποιο νήμα που βρίσκεται σε βρόχο περιδίνησης (waiting thread) γίνεται με τον ίδιο τρόπο όπως και στη γενική περίπτωση του Σχήματος 2.2. Όταν ο βρόχος περιδίνησης υλοποιείται με την εντολή PAUSE στο κυρίως σώμα του, η κατανομή των πόρων αλλάζει όπως φαίνεται στο Σχήμα 4.2. Σε αυτό το σχήμα απεικονίζονται με ρίγες οι δομές που το νήμα που περιδινείται εξακολουθεί να κατέχει χωρίς όμως να τις αξιοποιεί ουσιαστικά.

Χρησιμοποιώντας την εντολή HALT, ένας λογικός επεξεργαστής μπορεί να απελευθερώσει όλους τους στατικά διαχωρισμένους πόρους που έχει υπό την κατοχή του, να τους κάνει πλήρως διαθέσιμους στον ομότιμο επεξεργαστή και να σταματήσει την εκτέλεσή του, μεταβαίνοντας σε κατάσταση ύπνου. Ο επεξεργαστής τότε εισέρχεται σε Single-Threaded τρόπο λειτουργίας. Αργότερα, μόλις λάβει κάποιο *δι-επεξεργαστικό σήμα διακοπής* (inter-processor interrupt – IPI) από τον ενεργό επεξεργαστή, επανεκκινεί την εκτέλεσή του και οι πόροι διαμερίζονται ξανά. Ο επεξεργαστής τότε επανέρχεται σε Multi-Threaded τρόπο λειτουργίας. Στο Σχήμα 4.3 φαίνεται η κατανομή των πόρων ενός Hyper-threaded επεξεργαστή όταν το νήμα που εκτελεί το βρόχο περιδίνησης χρησιμοποιεί την εντολή HALT στο σώμα του βρόχου. Όλοι οι πόροι του επεξεργαστή μπορούν να χρησιμοποιηθούν από το κύριο νήμα εργασίας.

Ο επεξεργαστής Prescott της Intel εισήγαγε ένα νέο ζεύγος εντολών, τις MONITOR και MWAIT [Int 09a, Boggs 04]. Η MWAIT επιτρέπει σε έναν λογικό επεξεργαστή να εισέλθει σε μια κατάσταση "βελτιστοποιημένη για απόδοση", ενόσω περιμένει για μια απλή εγγραφή στο εύρος διευθύνσεων μνήμης που ορίστηκε από την MONITOR. Σε έναν Hyper-threaded επεξεργαστή, όλοι οι διαμοιραζόμενοι και διαχωρισμένοι πόροι ενός λογικού επεξεργαστή απελευθερώνονται με την MWAIT, όπως ακριβώς συμβαίνει και με την HALT. Επιπλέον, όπως και η HALT, οι MONITOR/MWAIT είναι *προνομιούχες* (privileged) εντολές, υπό την έννοια ότι απαιτούν προνόμια στην κλήση τους και μπορούν να εκτελεστούν μόνο σε επίπεδο πυρήνα. Όμως, αντίθετα με την HALT, η οποία απαιτεί μια σχετικά δαπανηρή αποστολή IPI για την αφύπνιση του αδρανούς επεξεργαστή, οι MONITOR/MWAIT απαιτούν μόνο μια απλή εγγραφή στη μνήμη για τον ίδιο σκοπό.

Μέχρι τώρα, οι εντολές MONITOR/MWAIT έχουν χρησιμοποιηθεί κυρίως για τους σκοπούς του δια-νηματικού συγχρονισμού στον κώδικα των λειτουργικών συστημάτων. Συγκεκριμένα, έχουν χρησιμοποιηθεί για την υλοποίηση του *αδρανούς βρόχου* (idle loop) του χρονοδρομολογητή. Σε ένα τυπικό σενάριο αδρανούς βρόχου, ένας επεξεργαστής περιδινείται σε ένα βρόχο εξετάζοντας διαρκώς μια θέση μνήμης (συγκεκριμένα, τη δομή που υλοποιεί την ουρά εργασιών) για να δει αν υπάρχει κάποια εργασία η οποία χρειάζεται δρομολόγηση. Η ειδοποίηση για την άφιξη μιας νέας εργασίας μπορεί να κωδικοποιηθεί σαν μια απλή εγγραφή στην δομή της ουράς εργασιών, γεγονός που θα προκαλέσει τον επεξεργαστή να βγει από τον αδρανή βρόχο. Για καλύτερη απόδοση, ο επεξεργαστής που εκτελεί τον αδρανή βρόχο θα μπορούσε εναλλακτικά να χρησιμοποιήσει τις MONITOR/MWAIT για να παρακολουθεί νέες αφίξεις στην ουρά, και την ίδια στιγμή να βρίσκεται σε μια "βελτιστοποιημένη" κατάσταση ύπνου. Αν ο χρονοδρομολογητής αποφασίσει να στείλει κάποια εργασία σε αυτόν τον επεξεργαστή, τότε απλά γράφει στη δομή της ουράς εργασιών του επεξεργαστή αυτού και τον αφυπνίζει. Μια υλοποίηση βασισμένη στην εντολή HALT θα απαιτούσε τη χρήση IPI για την αφύπνιση του αδρανούς επεξεργαστή. Όπως αναφέραμε όμως, η αποστολή και η εξυπηρέτηση τέτοιων σημάτων θα εισήγαγε κάποια καθυστέρηση στην εξυπηρέτηση νέων εργασιών.

Στα πλαίσια αυτής της δουλειάς εξετάζουμε τη δυναμική της χρήσης των εντολών MONITOR/MWAIT για το συγχρονισμό νημάτων σε *επίπεδο χρήστη*, τα οποία εκτελούνται σε Hyper-threaded επεξεργαστές και χαρακτηρίζονται από ασυμμετρία στο φορτίο εργασίας τους. Προτείνουμε ένα πλαίσιο μέσω του οποίου μπορεί κάποιος να χρησιμοποιήσει τις προνομιούχες αυτές εντολές για να υλοποιήσει πρωτογενείς λειτουργίες συγχρονισμού *αναμονής για συνθήκη* (condition-wait) και *ειδοποίησης* (notification), παρακάμπτοντας στο μέγιστο βαθμό την εμπλοκή του πυρήνα. Παρουσιάζουμε μια αξιολόγηση των προτεινόμενων πρωτογενών λειτουργιών, αρχίζοντας από μια ανάλυση συγκεκριμένων χαρακτηριστικών της απόδοσής τους που αντανακλούν το μοντέλο εκτέλεσης που θεωρούμε, και καταλήγοντας στην κατασκευή *φραγμάτων συγχρονισμού* (synchronization barriers) βασισμένα στις προτεινόμενες πρωτογενείς λειτουργίες και την αξιολόγησή τους στα πλαίσια πραγματικών εφαρμογών.

4.2 Οι Εντολές MONITOR/MWAIT

Η εντολή MONITOR ορίζει ένα εύρος διευθύνσεων εικονικής μνήμης να παρακολουθείται για λειτουργίες εγγραφής σε αυτό. Η MWAIT θέτει τον λογικό επεξεργαστή σε μια κατάσταση "βελτιστοποιημένη για απόδοση" (που μπορεί να διαφέρει ανάμεσα σε διαφορετικές υλοποιήσεις), μέχρι να πραγματοποιηθεί κάποια εγγραφή στην παρακολουθούμενη περιοχή μνήμης. Αυτό το ζεύγος εντολών στην ουσία υλοποιεί λειτουργικότητα αναμονής για συνθήκη όσο το δυνατόν πιο κοντά στο επίπεδο του υλικού, με το πλεονέκτημα ότι αποφεύγει την σπατάλη πόρων σε μια Hyper-threaded αρχιτεκτονική, ενώ απαιτεί για την ειδοποίηση του νήματος που αναμένει μια απλή εγγραφή στη μνήμη χωρίς την παρέμβαση του λειτουργικού συστήματος.

Πιο συγκεκριμένα, η εντολή MONITOR ρυθμίζει ειδικό υλικό στον επεξεργαστή να παρακολουθεί για εγγραφές το εύρος διευθύνσεων (τυπικά, μια γραμμή κρυφής μνήμης) που ορίζεται από τα περιεχόμενα του καταχωρητή EAX. Η λειτουργία της εντολής στηρίζεται σε μια κατάσταση του επεξεργαστή που καθορίζεται από τη *σημαία παρακολούθησης εκκρεμούς γεγονότος* (monitor event pending flag). Η εκτέλεση της MONITOR οπλίζει το υλικό παρακολούθησης και καθαρίζει τη σημαία. Μια εγγραφή στο παρακολουθούμενο εύρος διευθύνσεων καθώς και μια σειρά άλλων γεγονότων όπως σήματα διακοπής, θα πυροδοτήσει το υλικό και θα θέσει τη σημαία. Η κατάσταση του υλικού παρακολούθησης δεν είναι ορατή αρχιτεκτονικά παρά μόνο μέσα από τη συμπεριφορά της MWAIT.

Η MWAIT οδηγεί τον επεξεργαστή σε μια ειδική, βελτιστοποιημένη κατάσταση χαμηλής κατανάλωσης ισχύος, μέχρι να ανιχνευθεί κάποια εγγραφή σε οποιοδήποτε byte του παρακολουθούμενου εύρους διευθύνσεων, ή αν συμβεί κάποια διακοπή, εξαίρεση ή σφάλμα που πρέπει

```
While (!trigger_store_happened) {
   ECX=EDX=0, EAX=Logical_Address(Trigger)
   MONITOR
   If (!trigger_store_happened) {
      EAX=ECX=0
      MWAIT
   }
}
```

Σχήμα 4.4: *Τυπική χρήση των εντολών MONITOR/MWAIT.*

να εξυπηρετηθεί. Η εντολή είναι αρχιτεκτονικά ισοδύναμη με μια κενή εντολή (nop). Στην πραγματικότητα όμως υποδεικνύει στον επεξεργαστή ότι μπορεί να επιλέξει να εισέλθει σε μια κατάσταση βελτιστοποιημένη για απόδοση ενόσω περιμένει για κάποιο γεγονός ή για κάποια εγγραφή στο εύρος διευθύνσεων που ορίστηκε από την προηγούμενη εντολή MONITOR στη ροή εντολών. Σε έναν Hyper-threaded επεξεργαστή, ένα νήμα που καλεί την MWAIT οδηγεί τον λογικό επεξεργαστή του να απελευθερώσει όλους τους δυναμικά μοιραζόμενους και στατικά διαχωρισμένους πόρους και να μεταβεί σε κατάσταση ύπνου.

Όπως αναφέραμε, η έξοδος από την κατάσταση ύπνου της MWAIT μπορεί να οφείλεται σε γεγονότα διαφορετικά από μια εγγραφή στην πυροδοτούμενη διεύθυνση. Το λογισμικό συνεπώς θα πρέπει να ελέγχει ρητά την τρέχουσα τιμή της πυροδοτούμενης διεύθυνσης συγκρίνοντάς την με μια αρχική τιμή, προκειμένου να καθορίσει αν η έξοδος από την κατάσταση ύπνου έγινε εξαιτίας κάποιας εγγραφής ή εξαιτίας άλλου γεγονότος. Στην τελευταία περίπτωση, η MWAIT πρέπει να εκτελεστεί ξανά. Όμως η MWAIT δεν επαν-οπλίζει αυτόματα το υλικό παρακολούθησης, και έτσι η MONITOR θα πρέπει με τη σειρά της να εκτελεστεί εκ νέου. Με άλλα λόγια, οι MONITOR/MWAIT θα πρέπει να εκτελούνται στον ίδιο βρόχο, όπως φαίνεται στο Σχήμα 4.4 [2] [Boggs 04]. Σε αυτό το παράδειγμα κώδικα γίνεται ένας δεύτερος έλεγχος στην πυροδοτούμενη διεύθυνση για να εντοπιστούν πιθανές εγγραφές μεταξύ του πρώτου ελέγχου στη συνθήκη του while βρόχου και την εκτέλεση της εντολής MONITOR. Χωρίς το δεύτερο έλεγχο, μια εγγραφή σε αυτό το διάστημα δε θα γινόταν αντιληπτή.

Όπως αναφέραμε προηγουμένως, η διεύθυνση που δίνεται σαν όρισμα στην MONITOR ουσιαστικά ορίζει ένα εύρος διευθύνσεων μέσα στο οποίο η εγγραφή σε οποιοδήποτε byte θα οδηγήσει σε έξοδο από την κατάσταση ύπνου. Ο προγραμματιστής θα πρέπει να είναι προσεκτικός ώστε οι εγγραφές που επιδιώκουν την έξοδο από αυτήν την κατάσταση να συμβαίνουν

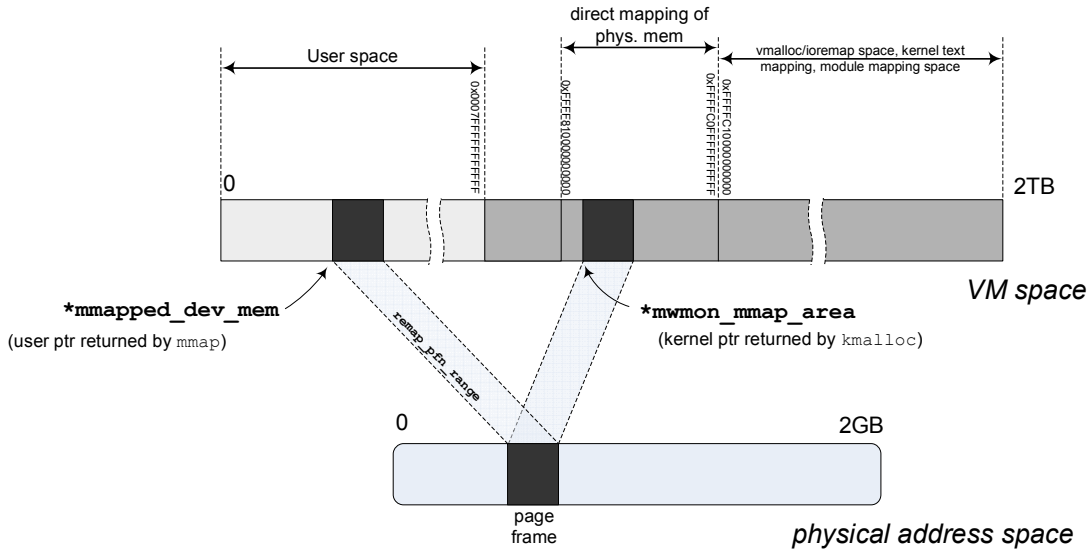
[2] Οι καταχωρητές ECX και EDX στην MONITOR, και οι EAX και ECX στην MWAIT, χρησιμοποιούνται για να διαβιβάσουν επιπλέον πληροφορία στις εντολές, όπως για παράδειγμα υποδείξεις για την επιθυμητή βελτιστοποιημένη κατάσταση καθώς και άλλες επεκτάσεις. Για την έκδοση του επεξεργαστή Xeon που εξετάζουμε (οικογένεια 15, μοντέλο 3) δεν ορίζονται τέτοιες υποδείξεις ή επεκτάσεις, και έτσι τα περιεχόμενα αυτών των καταχωρητών πριν την εκτέλεση των συγκεκριμένων εντολών καθαρίζονται.

μέσα σε αυτό το εύρος. Αυτό σημαίνει ότι, για να εξασφαλίσει ότι δε θα χάνονται οι *εκούσιες εγγραφές αφύπνισης*, η δομή δεδομένων που θα παρακολουθεί τις εγγραφές θα πρέπει να χωράει στο *μικρότερο μέγεθος γραμμής παρακολούθησης*, και θα πρέπει να είναι κατάλληλα ευθυγραμμισμένη ώστε να μην εκτείνεται πέρα από τα όρια της γραμμής. Σε διαφορετική περίπτωση, ο επεξεργαστής μπορεί να μην αφυπνιστεί μετά από μια εγγραφή που επιδιώκει την έξοδο από την κατάσταση της MWAIT. Ομοίως, λειτουργίες εγγραφής οι οποίες δεν επιδιώκουν την έξοδο από την βελτιστοποιημένη κατάσταση, δε θα πρέπει να αλλάζουν κανένα byte μέσα στο παρακολουθούμενο εύρος διευθύνσεων. Έτσι, προκειμένου να αποφευχθούν *ακούσιες εγγραφές αφύπνισης*, ο προγραμματιστής χρειάζεται ίσως να επεκτείνει τεχνητά (pad) τη δομή δεδομένων που παρακολουθεί τις εγγραφές ώστε το μέγεθός της να εξισωθεί με το *μεγαλύτερο μέγεθος γραμμής παρακολούθησης*. Αυτό θα απέκλειε πιθανή δέσμευση χώρου μη σχετικών δομών δεδομένων εντός του παρακολουθούμενου εύρους διευθύνσεων. Στο σύστημά μας, τόσο το μικρότερο όσο και το μεγαλύτερο μέγεθος γραμμής παρακολούθησης είναι 64 bytes.

4.3 Πλαίσιο για την Υλοποίηση Λειτουργιών Συγχρονισμού

Από την αρχική τους υλοποίηση, οι εντολές MONITOR και MWAIT ήταν διαθέσιμες μόνο στο επίπεδο προνομίων 0, δηλαδή μπορούσαν να χρησιμοποιηθούν μόνο σε επίπεδο πυρήνα. Για να μπορέσουμε να υλοποιήσουμε μια λειτουργία αναμονής για συνθήκη η οποία μπορεί να χρησιμοποιηθεί στο επίπεδο χρήστη, έπρεπε να επεκτείνουμε τον πυρήνα του Linux με κλήση συστήματος ώστε να μπορούμε να προσπελάζουμε τις εντολές αυτές. Το κόστος αυτής της κλήσης συστήματος είναι το ελάχιστο που πρέπει να χρεωθεί οποιαδήποτε εφαρμογή επιθυμεί να χρησιμοποιήσει τις εντολές MONITOR/MWAIT. Από εκεί και πέρα, η επιπλέον επιβάρυνση εξαρτάται από τον τρόπο που οι λειτουργίες αναμονής και ειδοποίησης διαβιβάζουν μεταξύ τους τα περιεχόμενα της πυροδοτούμενης διεύθυνσης.

Με άλλα λόγια, για την υλοποίηση των λειτουργιών συγχρονισμού έπρεπε αρχικά να καταλήξουμε στο πού θα δεσμευτεί η περιοχή μνήμης που είναι προς παρακολούθηση. Αν αυτή η περιοχή δεσμευτεί στο επίπεδο χρήστη, ο πυρήνας θα πρέπει να προσπελάζει το χώρο διευθύνσεων της διεργασίας κάθε φορά που πρέπει να ελέγξει τα περιεχόμενά της. Αυτό θα απαιτούσε την αντιγραφή των περιεχομένων της περιοχής στην εικονική μνήμη του πυρήνα, καλώντας κάποια συνάρτηση όπως την `copy_from_user` μέσα στο σώμα του βρόχου. Αν η παρακολουθούμενη περιοχή δεσμευτεί στο επίπεδο πυρήνα, τότε θα έπρεπε να υλοποιήσουμε μια επιπλέον κλήση συστήματος για να μπορούν οι εφαρμογές να μεταβάλουν τα περιεχόμενά της. Συνεπώς, και οι δύο περιπτώσεις θα εισήγαγαν σημαντική επιπλέον επιβάρυνση, είτε στην λειτουργία της αναμονής, εξαιτίας των πολλαπλών αντιγραφών των περιεχομένων της περιοχής, είτε στην λειτουργία της ειδοποίησης, εξαιτίας του κόστους της απαιτούμενης κλήσης συστήματος.



Σχήμα 4.5: *Εδραιώνοντας γρήγορη ανταλλαγή δεδομένων ανάμεσα στα επίπεδα πυρήνα και χρήστη. Η παρακολουθούμενη περιοχή μνήμης δεσμεύεται στο επίπεδο πυρήνα, εντός μιας συνεχόμενης περιοχής στη φυσική μνήμη μεγέθους ίσου με μία σελίδα, και στη συνέχεια απεικονίζεται σαν μια ειδική συσκευή χαρακτήρων στο χώρο διευθύνσεων της διεργασίας του χρήστη.*

4.3.1 Εδραιώνοντας γρήγορη ανταλλαγή δεδομένων μεταξύ των επιπέδων πυρήνα και χρήστη

Προκειμένου να εδραιώσουμε τη γρηγορότερη δυνατή επικοινωνία μεταξύ των επιπέδων πυρήνα και χρήστη ακολουθήσαμε μια εναλλακτική προσέγγιση: επιλέξαμε να δεσμεύσουμε την παρακολουθούμενη περιοχή μνήμης στο χώρο διευθύνσεων του πυρήνα, στα πλαίσια μια ειδικής, εικονικής συσκευής χαρακτήρων (character device), και στη συνέχεια να απεικονίσουμε τη συσκευή στο χώρο διευθύνσεων του χρήστη. Με αυτόν τον τρόπο η περιοχή μνήμης μπορεί να προσπελάζεται άμεσα και από το επίπεδο πυρήνα και από το επίπεδο χρήστη, χωρίς να απαιτούνται αντιγραφές ή πρόσθετες κλήσεις συστήματος. Ο τρόπος απεικόνισης φαίνεται στο Σχήμα 4.5.

Ο οδηγός για την ειδική συσκευή χαρακτήρων (*kmem_mapper*) υλοποιείται σαν μια επέκταση πυρήνα με δυνατότητα φόρτωσης και εκφόρτωσης (loadable kernel module). Η παρακολουθούμενη περιοχή μνήμης δεσμεύεται τη στιγμή που η επέκταση φορτώνεται στον πυρήνα. Συγκεκριμένα, στη συνάρτηση αρχικοποίησης της επέκτασης καλούμε τη συνάρτηση του πυρήνα `kmalloc` για να δεσμεύσουμε 4096 bytes συνεχόμενης φυσικής μνήμης (στην πραγματικότητα, ένα πλαίσιο σελίδας στην φυσική μνήμη). Η τιμή που επιστρέφει η `kmalloc` αρχικοποιεί έναν δείκτη στον χώρο ιδεατών διευθύνσεων του πυρήνα, ο οποίος υποδεικνύει την αρχή της παρακολουθούμενης περιοχής μνήμης, που έχει μέγεθος 64 bytes (`*mwmon_mmap_area`). Φυσικά,

φροντίζουμε ώστε αυτή η περιοχή να είναι κατάλληλα ευθυγραμμισμένη στα όρια του μικρότερου μεγέθους γραμμής παρακολούθησης, όπως συζητήθηκε στην ενότητα 4.2. Επιπλέον, για να αποτρέψουμε τη σελίδα μνήμης από το να εναλλαχθεί και να μεταφερθεί στο δίσκο (swap out), θέτουμε τη σημαία `PG_reserved` του αντίστοιχου πλαισίου σελίδας. Ο δείκτης `mwmon_mmap_area` είναι μέρος της προγραμματιστικής διεπαφής του πυρήνα, δηλαδή τον ορίζουμε στον κώδικα του πυρήνα και τον εξάγουμε σαν σύμβολο αυτού (kernel symbol). Όπως θα δούμε, ο δείκτης αυτός αποτελεί τη λαβή για την κλήση συστήματος που υλοποιεί τη λειτουργία αναμονής, για να μπορεί να προσπελάζει την παρακολουθούμενη περιοχή μνήμης.

Υλοποιούμε τρεις μεθόδους για τη συσκευή *kmem_mapper*: open, mmap και close. Στις open και close, όλα τα bytes της παρακολουθούμενης περιοχής τίθενται σε μια αρχική τιμή, ενδεικτική της "μη ειδοποίησης" (`MWMON_ORIGINAL_VAL`). Η mmap είναι αυτή που επιτελεί το σύνολο της λειτουργικότητας της εικονικής συσκευής: καλεί τη συνάρτηση πυρήνα `remap_pfn_range`, με τον δείκτη `mwmon_mmap_area` ανάμεσα στα ορίσματα, για να επαν-απεικονίσει τη δεσμευμένη από τον οδηγό σελίδα μνήμης στο επίπεδο χρήστη. Αυτό γίνεται όταν το πρόγραμμα του χρήστη χρησιμοποιεί την κλήση συστήματος `mmap` για να απεικονίσει στο χώρο διευθύνσεών του τη συσκευή *kmem_mapper*. Η διεύθυνση που επιστρέφει η `mmap` θα δείχνει στην αρχή της παρακολουθούμενης περιοχής μνήμης, ακριβώς όπως κάνει ο δείκτης `mwmon_mmap_area` στην πλευρά του πυρήνα. Στη συνάρτηση εκκαθάρισης της επέκτασης, που καλείται όταν εκείνη εκφορτώνεται από τον πυρήνα, η σημαία `PG_reserved` της δεσμευμένης σελίδας καθαρίζεται και η σελίδα αποδεσμεύεται με κλήση στην συνάρτηση πυρήνα `kfree`.

4.3.2 Προγραμματιστική διεπαφή για λειτουργίες συγχρονισμού βασισμένες στις MONITOR/MWAIT

Αφού εδραιωθεί η γρήγορη επικοινωνία ανάμεσα στα επίπεδα πυρήνα και χρήστη, η υλοποίηση μιας κλήσης συστήματος για λειτουργία αναμονής για συνθήκη με χρήση των MONITOR και MWAIT είναι άμεση. Επεκτείναμε τον πυρήνα του Linux με την κλήση συστήματος `mwmon_mmap_sleep`, η οποία φαίνεται στο Σχήμα 4.6.

Μετά την φόρτωση της επέκτασης για τον οδηγό της εικονικής συσκευής, τα βήματα που πρέπει να γίνουν σε ένα τυπικό πολυνηματικό πρόγραμμα για χρήση των λειτουργιών συγχρονισμού που βασίζονται στις MONITOR/MWAIT είναι τα ακόλουθα:

1. Στη φάση αρχικοποίησης, το πρόγραμμα ανοίγει τη συσκευή *kmem_mapper* για ανάγνωση και εγγραφή, και στη συνέχεια την απεικονίζει μέσω της κλήσης συστήματος `mmap` στο χώρο διευθύνσεών του (και εδώ, με τις σημαίες προστασίας ανάγνωσης και εγγραφής ενεργοποιημένες).

```
/* Pointer to memory allocated by device driver */
char *mwmon_mmap_area;
EXPORT_SYMBOL_GPL(mwmon_mmap_area);

#define MWMON_ORIGINAL_VAL 'a'
#define MWMON_NOTIFIED_VAL 'b'

asmlinkage long sys_mwmon_mmap_sleep(void)
{
   do {
      local_irq_disable();
      monitor(mwmon_mmap_area,0,0);
      local_irq_enable();
      if(*mwmon_mmap_area == MWMON_NOTIFIED_VAL)
         break;
      mwait(0,0);
   } while (*mwmon_mmap_area != MWMON_NOTIFIED_VAL);
   *mwmon_mmap_area = MWMON_ORIGINAL_VAL;

   return 0;
}
```

Σχήμα 4.6: *Κλήση συστήματος που υλοποιεί τη λειτουργία αναμονής για συνθήκη με χρήση των MONITOR/MWAIT.*

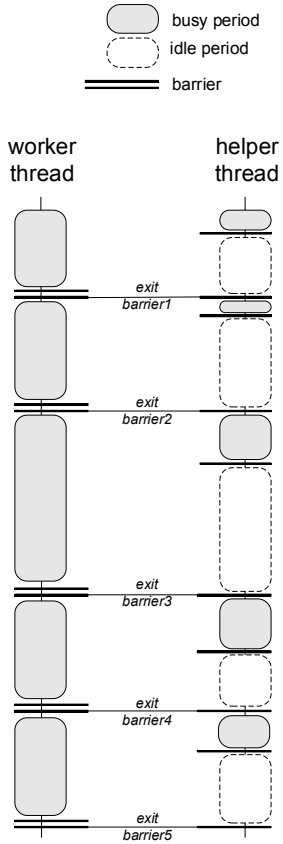
2. Ένα νήμα το οποίο επιθυμεί να περιμένει μέχρι να ικανοποιηθεί κάποια συνθήκη, καλεί την `mwmon_mmap_sleep` (χωρίς κανένα όρισμα).

3. Ένα νήμα το οποίο επιθυμεί να ειδοποιήσει το νήμα που περιμένει, θέτει ένα τυχαίο byte της παρακολουθούμενης περιοχής μνήμης στην τιμή `MWMON_NOTIFIED_VAL`. Η παρακολουθούμενη περιοχή μνήμης αρχίζει από τη διεύθυνση που επιστράφηκε από την `mmap` και τελειώνει 64 bytes μετά.

4. Κατά τον τερματισμό, το πρόγραμμα καταργεί την απεικόνιση της συσκευής (μέσω της κλήσης συστήματος `munmap`) και εν συνεχεία την κλείνει.

4.4 Αξιολόγηση Απόδοσης

Αξιολογούμε την αποδοτικότητα των πρωτογενών λειτουργιών συγχρονισμού με ιεραρχικό τρόπο: Πρώτα, συγκρίνουμε τις προτεινόμενες πρωτογενείς λειτουργίες με άλλες πιθανές υλοποιήσεις, μετρώντας απευθείας χαρακτηριστικά της απόδοσής τους, όπως κατανάλωση πόρων και αποκρισιμότητα. Σε δεύτερη φάση, χρησιμοποιούμε τις προτεινόμενες λειτουργίες για να υλοποιήσουμε φράγματα συγχρονισμού. Και σε αυτήν την περίπτωση προσπαθούμε να ποσοτικοποιήσουμε τα ίδια χαρακτηριστικά όπως πριν, και επιπλέον μετράμε την συνολική αποδοτικότητα των φραγμάτων μέσω ενός στοιχειώδους μετροπρογράμματος που υλοποιήσαμε. Σαν



Σχήμα 4.7: *Ένα τυπικό παράδειγμα του θεωρούμενου μοντέλου εφαρμογών. Ένα νήμα εργασίας και ένα βοηθητικό νήμα εκτελούνται σε διαφορετικά νήματα υλικού ενός Hyper-threaded επεξεργαστή. Σε κάθε φάση εκτέλεσης, το βοηθητικό νήμα πραγματοποιεί ένα μικρό μέρος δουλειάς η οποία προορίζεται να βοηθήσει την εκτέλεση του νήματος εργασίας στην επόμενη φάση. Οι φάσεις συγχρονίζονται μεταξύ τους με φράγματα συγχρονισμού.*

τελευταίο βήμα, αξιολογούμε τη δική μας υλοποίηση των φραγμάτων συγχρονισμού σε πραγματικά προγράμματα, και συγκεκριμένα στα πλαίσια της εφαρμογής του σχήματος SPR.

Σε κάθε βήμα της προαναφερθείσας διαδικασίας αξιολόγησης έχουμε θεωρήσει ένα μοντέλο εκτέλεσης όπου δύο νήματα λογισμικού εκτελούνται καθ' όλη τη διάρκεια εκτέλεσης σε διαφορετικά νήματα υλικού ενός Hyper-threaded επεξεργαστή. Τα νήματα αυτά θεωρούνται *ασύμμετρα*, όχι τόσο εξαιτίας του είδους της δουλειάς που επιτελούν (π.χ., λειτουργίες μνήμης έναντι υπολογισμών), όσο λόγω του μεγέθους του φορτίου εργασίας τους. Συνεπώς, το τυπικό σενάριο που εξετάζουμε περιλαμβάνει ένα νήμα με βαρύ φορτίο ("heavyweight"), που αντιπροσωπεύει συνήθως ένα κύριο νήμα εργασίας, και ένα νήμα με ελαφρύ φορτίο ("lightweight"), που αντιπροσωπεύει ένα βοηθητικό νήμα. Το βαρύ νήμα πραγματοποιεί υπολογισμούς καθ' όλη τη διάρκεια

της εκτέλεσης του προγράμματος, ενώ για το ελαφρύ νήμα η εκτέλεση εναλλάσσεται ανάμεσα σε μικρές περιόδους χρήσιμης δουλειάς και μακρές περιόδους αναμονής. Όταν βρίσκεται σε κατάσταση αναμονής, το ελαφρύ νήμα περιμένει μέχρι να ειδοποιηθεί από το βαρύ προτού προχωρήσει στην επόμενη φάση (για παράδειγμα, μέσω φραγμάτων συγχρονισμού ή *μεταβλητών συνθήκης* (condition variables)).

Ένα παράδειγμα για αυτό το σενάριο απεικονίζεται στο Σχήμα 4.7. Σε αυτήν την περίπτωση, τα δύο νήματα συγχρονίζονται με φράγματα. Το βαρύ νήμα είναι συνεχώς απασχολημένο, ενώ το ελαφρύ "συρρικνώνεται' περιοδικά κάθε φορά που τελειώνει τη σύντομη δουλειά του και εισέρχεται στο φράγμα. Σε πραγματικές εφαρμογές, το ελαφρύ νήμα θα μπορούσε να είναι ένα βοηθητικό νήμα το οποίο υποστηρίζει την εκτέλεση του κύριου νήματος με διάφορους τρόπους. Για παράδειγμα, πραγματοποιώντας προφόρτωση δεδομένων στα πλαίσια του σχήματος SPR, πραγματοποιώντας δικτυακές λειτουργίες εισόδου/εξόδου και επεξεργασία μηνυμάτων σε παράλληλες εφαρμογές κατανεμημένης μνήμης, πραγματοποιώντας λειτουργίες πρόσβασης στο δίσκο σε εφαρμογές απαιτητικές σε είσοδο/έξοδο, κ.λπ.

Η παραπάνω συζήτηση αποκαλύπτει μια σειρά από απαιτήσεις οι οποίες πρέπει να ικανοποιούνται προκειμένου οι πρωτογενείς λειτουργίες συγχρονισμού να είναι αποδοτικές για αυτό το μοντέλο εκτέλεσης. Καταρχήν, το βοηθητικό νήμα δε θα πρέπει να προκαλεί σημαντική επιβράδυνση στο νήμα εργασίας όσο περιμένει σε γεγονότα συγχρονισμού, καταναλώνοντας κοινούς πόρους που θα μπορούσαν να φανούν χρήσιμοι στο νήμα εργασίας. Ύστερα, το βοηθητικό νήμα θα πρέπει να επανέρχεται όσο το δυνατόν πιο γρήγορα από την κατάσταση ύπνου κάθε φορά που ειδοποιείται από το κύριο νήμα, προκειμένου οι ενέργειές του να είναι έγκαιρες και ακριβείς. Αυτή η απαίτηση γίνεται πιο σημαντική, καθώς η ανάγκη για πιο λεπτομερή ενορχήστρωση των ενεργειών του βοηθητικού νήματος γίνεται μεγαλύτερη. Τέλος, ο χρόνος που χρειάζεται το κύριο νήμα να καλέσει τη λειτουργία συγχρονισμού προκειμένου να ειδοποιήσει το βοηθητικό νήμα, πρέπει να είναι ο μικρότερος δυνατός. Και σε αυτήν την περίπτωση, αυτό είναι σημαντικό σε περιπτώσεις συχνού συγχρονισμού ανάμεσα στα νήματα.

Συνοψίζοντας, οι λειτουργίες συγχρονισμού για το θεωρούμενο μοντέλο εκτέλεσης πρέπει να έχουν *χαμηλή κατανάλωση πόρων*, *υψηλή αποκρισιμότητα* και *χαμηλό κόστος κλήσης*. Όπως αναφέραμε στην ενότητα 3.4.4, αυτές οι απαιτήσεις είναι συγκρουόμενες. Στα πλαίσια της έρευνάς μας, αναζητούμε την επιλογή εκείνη η οποία τις εξισορροπεί με τον αποτελεσματικότερο τρόπο.

4.4.1 Περιβάλλον εκτέλεσης πειραμάτων

Το περιβάλλον εκτέλεσης πειραμάτων που χρησιμοποιήσαμε ήταν το ίδιο με αυτό που παρουσιάστηκε στην ενότητα 3.5.1. Ο επεξεργαστής ήταν ένας Intel Xeon στα 2.8GHz (έκδοσης

"Prescott") με τεχνολογία Hyper-threading. Το λειτουργικό που χρησιμοποιήθηκε ήταν το Linux με πυρήνα 2.6.13 για αρχιτεκτονική x86_64. Για τη δημιουργία και τη διαχείριση των νημάτων χρησιμοποιήθηκε η βιβλιοθήκη NPTL από την έκδοση 2.5 της libc, ενώ για την απεικόνιση των νημάτων σε συγκεκριμένους λογικούς επεξεργαστές χρησιμοποιήθηκε η κλήση συστήματος `sched_setaffinity`. Όλοι οι κώδικες μεταγλωττίστηκαν με την έκδοση 4.1.2 του gcc και με επίπεδο βελτιστοποίησης O2. Αποτελέσματα από τους μετρητές απόδοσης ελήφθησαν όπως περιγράφηκε στην ενότητα 3.5.1.

4.4.2 Αξιολόγηση των χαρακτηριστικών απόδοσης των πρωτογενών λειτουργιών συγχρονισμού

Σε πρώτη φάση, υποθέτουμε ένα σενάριο εκτέλεσης σαν κι αυτό του Σχήματος 4.8. Θεωρούμε ότι υπάρχουν δύο νήματα τα οποία δρομολογούνται σε διαφορετικούς λογικούς επεξεργαστές του ίδιου φυσικού πακέτου. Το βαρύ νήμα πραγματοποιεί ένα συγκεκριμένο ποσό υπολογισμών κινητής υποδιαστολής (έναν 512×512 πολλαπλασιασμό πινάκων), ενώ το ελαφρύ απλά περιμένει μέχρι να ειδοποιηθεί από το πρώτο όταν αυτό ολοκληρώσει τους υπολογισμούς. Για κάθε υλοποίηση των πρωτογενών λειτουργιών συγχρονισμού που εξετάζουμε, μετράμε τους ακόλουθους χρόνους:

- T_{work}: ο χρόνος που χρειάζεται το νήμα εργασίας να ολοκληρώσει τους υπολογισμούς. Όσο μεγαλύτερος είναι αυτός ο χρόνος, τόσο μεγαλύτερη παρεμβολή προκαλεί η συνύπαρξη του νήματος που αναμένει στον ομότιμο λογικό επεξεργαστή.

- T_{wakeup}: ο χρόνος ανάμεσα στη στιγμή ειδοποίησης του νήματος που αναμένει και τη στιγμή που αυτό αφυπνίζεται πραγματικά. Αυτός ο χρόνος είναι μια άμεση ένδειξη της αποκρισιμότητας της πρωτογενούς λειτουργίας αναμονής.

- T_{call}: ο χρόνος που απαιτεί η κλήση της πρωτογενούς λειτουργίας ειδοποίησης από το νήμα εργασίας.

Αξιολογούμε και συγκρίνουμε τις εξής υλοποιήσεις για τις πρωτογενείς λειτουργίες αναμονής και ειδοποίησης: την προτεινόμενη υλοποίηση με τις εντολές MONITOR/MWAIT (*mwmon*), βρόχους περιδίνησης που χρησιμοποιούν την εντολή PAUSE (*spin-loops*), βρόχους περιδίνησης που χρησιμοποιούν την εντολή HALT (*spin-loops-halt*), και τις αντίστοιχες λειτουργίες συγχρονισμού από την βιβλιοθήκη NPTL (*pthreads*).

Στην υλοποίηση *mwmon*, το νήμα που περιμένει καλεί την `mwmon_mmap_sleep` προκειμένου να μεταβεί σε κατάσταση ύπνου, και αφυπνίζεται όταν το νήμα εργασίας τροποποιήσει την παρακολουθούμενη περιοχή μνήμης. Αυτή η διαδικασία έχει ήδη περιγραφεί στην ενότητα 4.3.

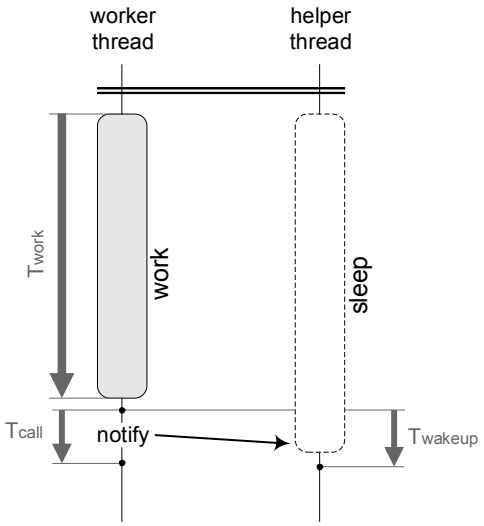
Στην υλοποίηση *spin-loops*, το νήμα που περιμένει περιδινείται σε μία μεταβλητή συγχρονισμού επιπέδου χρήστη έως ότου την ενημερώσει το νήμα εργασίας. Προκειμένου να κάνουμε την περιδίνηση όσο γρήγορη χρειάζεται, όπως εξηγήσαμε στην ενότητα 4.1, έχουμε ενσωματώσει την εντολή PAUSE στο σώμα του βρόχου.

Η υλοποίηση *spin-loops-halt* είναι παρόμοια, με τη διαφορά ότι το νήμα που αναμένει καλεί την εντολή HALT αντί της PAUSE, ενώ για την ειδοποίηση το νήμα εργασίας στέλνει επιπλέον ένα δι-επεξεργαστικό σήμα διακοπής για να αφυπνίσει το ανενεργό νήμα. Με αυτόν τον τρόπο το νήμα βάζει τον επεξεργαστή του σε κατάσταση ύπνου απελευθερώνοντας όλους τους πόρους του. Μπορεί να αφυπνίζεται περιοδικά από διάφορα IPIs που προέρχονται από το λειτουργικό (π.χ. timer interrupts), αλλά θα αντιληφθεί την αποστολή του IPI από το νήμα εργασίας μόνο όταν διαπιστώσει και την αλλαγή στη μεταβλητή συγχρονισμού. Για τη χρήση της εντολής HALT και την αποστολή IPIs από το επίπεδο χρήστη επεκτείναμε τον πυρήνα του Linux με κατάλληλες κλήσεις συστήματος.

Η τέταρτη εναλλακτική που εξετάσαμε ήταν η υλοποίηση των μεταβλητών συνθήκης της βιβλιοθήκης Pthreads. Η έκδοση της βιβλιοθήκης Pthreads που χρησιμοποιήσαμε κάνει χρήση του μηχανισμού των *futexes*, ενός μηχανισμού που παρέχει ο πυρήνας του Linux μέσω της κλήσης συστήματος `futex` σαν δομικό στοιχείο για γρήγορες λειτουργίες κλειδώματος. Περισσότερες λεπτομέρειες του τρόπου λειτουργίας του μηχανισμού αυτού είναι εκτός του αντικειμένου της διατριβής, και παραπέμπουμε τον αναγνώστη στο [Drepper 05] για μια συζήτηση γύρω από τις βασικές του ιδέες καθώς και τα επακόλουθα της χρήσης του για υλοποίηση λειτουργιών συγχρονισμού. Η κλήση της `futex` με όρισμα `FUTEX_WAIT` οδηγεί ένα νήμα να διακόψει την εκτέλεσή του και να απο-δρομολογηθεί στο επίπεδο πυρήνα. Υποθέτοντας ότι δεν υπάρχουν άλλες εκτελέσιμες διεργασίες όλοι οι πόροι του απελευθερώνονται και διατίθενται για το ομότιμο νήμα. Στην ουσία, ο επεξεργαστής μεταβαίνει από Multi-Threaded σε Single-Threaded τρόπο λειτουργίας. Η κλήση της `futex` με όρισμα `FUTEX_WAKE` αφυπνίζει και επαναδρομολογεί το νήμα. Για τους σκοπούς του σεναρίου εκτέλεσης που εξετάζουμε χρησιμοποιήσαμε τις συναρτήσεις βιβλιοθήκης `pthread_cond_wait` και `pthread_cond_signal`, οι οποίες καλούν εσωτερικά την `futex` με ορίσματα `FUTEX_WAIT` και `FUTEX_WAKE`, αντίστοιχα [3].

Ο Πίνακας 4.1 παρουσιάζει τα αποτελέσματα από την αξιολόγηση κάθε υλοποίησης για το σενάριο εκτέλεσης του Σχήματος 4.8. Εκτελέσαμε κάθε τέτοιο πείραμα για 50 επαναλήψεις και παρουσιάζουμε τις μέσες τιμές από όλες τις μετρήσεις, καθώς και την τυπική τους απόκλιση.

[3] Την εκτέλεση των `pthread_cond_wait` και `pthread_cond_signal` περιβάλλουμε επιπρόσθετα με κλήσεις σε συναρτήσεις βιβλιοθήκης για κλείδωμα και ξεκλείδωμα της μεταβλητής αμοιβαίου αποκλεισμού (mutex variable) που συσχετίζεται με την μεταβλητή συνθήκης, όπως προδιαγράφεται από το πρότυπο των Pthreads.



Σχήμα 4.8: *Σενάριο εκτέλεσης για την αξιολόγηση των πρωτογενών λειτουργιών αναμονής για συνθήκη και ειδοποίησης. Αρχικά τα δύο νήματα είναι συγχρονισμένα με φράγματα. Το νήμα εργασίας εκτελεί έναν 512×512 πολλαπλασιασμό πινάκων, ενώ το βοηθητικό νήμα δεν κάνει τίποτα από το να περιμένει μέχρι να ειδοποιηθεί από το πρώτο, όταν εκείνο ολοκληρώσει τους υπολογισμούς. Μια αποδοτική υλοποίηση για το μοντέλο εφαρμογών που εξετάζουμε θα πρέπει να προσφέρει μειωμένους χρόνους T_{work}, T_{wakeup} και T_{call} την ίδια στιγμή.*

Υλοποίηση	**Χρόνος εργασίας (σε δευτερόλεπτα)**	**Χρόνος αφύπνισης (σε κύκλους)**	**Χρόνος κλήσης (σε κύκλους)**
spin-loops	4.2446 (±0.2987)	584 (±191)	785 (±223)
spin-loops-halt	3.6243 (±0.3036)	37470 (±3975)	27813 (±3768)
pthreads	3.5919 (±0.2367)	116989 (±5795)	70042 (±3569)
mwmon	3.4821 (±0.2996)	25381 (±2426)	5470 (±545)

Πίνακας 4.1: *Απόδοση των διαφορετικών υλοποιήσεων για πρωτογενείς λειτουργίες αναμονής για συνθήκη και ειδοποίησης (βλ. Σχήμα 4.8).*

Όπως ήταν αναμενόμενο, η υλοποίηση *spin-loops* έχει ελάχιστο χρόνο απόκρισης και κόστος κλήσης αφού δεν απαιτεί τη μεσολάβηση του λειτουργικού συστήματος, όμως έχει την χειρότερη επίδοση όσον αφορά την κατανάλωση πόρων. Κατά μέσο όρο, το περιδινούμενο νήμα επιβραδύνει την εκτέλεση του νήματος εργασίας κατά 19% πιο πολύ σε σχέση με τις άλλες τρεις υλοποιήσεις. Η μειωμένη παρεμβολή όμως που εισάγεται στο νήμα εργασίας με αυτές τις υλοποιήσεις έρχεται με κόστος δύο έως τρεις τάξεις μεγέθους μεγαλύτερους χρόνους αφύπνισης

και κλήσης. Συγκεκριμένα, η υλοποίηση *pthreads* σημειώνει τους μεγαλύτερους χρόνους αφύπνισης και κλήσης, πιθανόν λόγω των μεγαλύτερων μονοπατιών ελέγχου μέσα στον πυρήνα για την ειδοποίηση και την επαναδρομολόγηση του κοιμώμενου νήματος.

Στην υλοποίηση *mwmon*, το βοηθητικό νήμα επανεκκινείται 47% πιο γρήγορα σε σχέση με την *spin-loops-halt*, και έχει σχεδόν 5 φορές μικρότερο κόστος κλήσης. Αυτό υποδηλώνει ότι η βελτιστοποιημένη κατάσταση ύπνου της MWAIT είναι πιθανώς πιο αποκρίσιμη σε σχέση με αυτή της HALT, ενώ είναι σχεδόν εξίσου αποδοτική όσον αφορά την κατανάλωση πόρων. Προφανώς, η αποστολή IPIs μέσω κλήσεων συστήματος για την αφύπνιση του βοηθητικού νήματος είναι πολύ πιο δαπανηρή σε σχέση με μια απλή εγγραφή σε μεταβλητή στο επίπεδο χρήστη. Όσον αφορά το χρόνο εργασίας, οι τελευταίες τρεις υλοποιήσεις αποδίδουν εξίσου καλά, με την *mwmon* να αποδίδει ελαφρώς καλύτερα. Συνολικά, η *mwmon* είναι η επιλογή η οποία εξισορροπεί καλύτερα τη χαμηλή κατανάλωση πόρων, την υψηλή αποκρισιμότητα και το μειωμένο κόστος κλήσης.

4.4.3 Αξιολόγηση φραγμάτων συγχρονισμού

Χρησιμοποιώντας τις πρωτογενείς λειτουργίες αναμονής για συνθήκη και ειδοποίησης που βασίζονται στις MWAIT/MONITOR, κατασκευάσαμε φράγματα συγχρονισμού. Τα φράγματα αυτά προορίζονται για χρήση στο σενάριο εκτέλεσης που παρουσιάσαμε στην ενότητα 4.4, δηλαδή από ένα ζεύγος νημάτων που εκτελούνται σε έναν Hyper-threaded επεξεργαστή. Στερούνται τη γενικότητα άλλων υλοποιήσεων, υπό την έννοια ότι υπάρχει περιορισμός στον αριθμό των νημάτων που μπορούν να συγχρονιστούν σε ένα φράγμα ή στον αριθμό των φραγμάτων που μπορούν να χρησιμοποιούνται ανά πάσα στιγμή από ένα πρόγραμμα, αλλά μπορούν να επεκταθούν σχετικά άμεσα προκειμένου να ξεπεραστούν αυτοί οι περιορισμοί.

Για παράδειγμα, για την υποστήριξη διαφορετικών φραγμάτων συγχρονισμού στα πλαίσια του ίδιου προγράμματος, θα μπορούσαν να χρησιμοποιηθούν διαφορετικές περιοχές μνήμης που θα παρακολουθούνται για λειτουργίες εγγραφής. Για την υποστήριξη μεγαλύτερου αριθμού νημάτων, θα μπορούσαν να χρησιμοποιηθούν ιεραρχικά φράγματα, όπου τα νήματα του ίδιου φυσικού πακέτου συγχρονίζονται αρχικά ανά ζευγάρια, και στη συνέχεια, εκπρόσωποι από όλα τα φυσικά πακέτα συγχρονίζονται μεταξύ τους. Στα πλαίσια αυτής της δουλειάς, το ενδιαφέρον μας επικεντρώνεται πιο πολύ στην αποδοτικότητα της υλοποίησης των φραγμάτων στο συγκεκριμένο μοντέλο εκτέλεσης, και όχι τόσο στη γενικότητα χρήσης τους.

Τα βασικά σημεία της υλοποίησης των φραγμάτων συγχρονισμού (θα την αποκαλούμε *mwmon* στο εξής) παρουσιάζονται στο Σχήμα 4.9. Κάθε νήμα που εισέρχεται στο φράγμα αυξάνει ατομικά έναν καθολικό μετρητή χρησιμοποιώντας μια μεταβλητή κλειδώματος (spin lock). Το

```
void mwait_barrier(mwait_barrier_t *barrier)
{
   unsigned int total = barrier->total;

   spin_lock(&barrier->lock);
   ++barrier->gathered;

   if (barrier->gathered == total) { /* last thread */
      *mmapped_device_memory = MWMON_NOTIFIED_VAL;

   } else { /* intermediate thread */
       spin_unlock(&barrier->lock);
       mwmon_mmap_sleep();
   }

   /* Is this the last woken thread? If yes, then unlock. */
   if (atomic_dec_and_test(&barrier->gathered))
      spin_unlock(&barrier->lock);
}
```

Σχήμα 4.9: *Ενδεικτική υλοποίηση φράγματος συγχρονισμού με πρωτογενείς λειτουργίες που βασίζονται στις MWAIT/MONITOR.*

νήμα που εισέρχεται πρώτο και πρόκειται να περιμένει στο φράγμα, απελευθερώνει το κλειδί και σταματάει την εκτέλεσή του πηγαίνοντας σε κατάσταση ύπνου. Θα αφυπνιστεί όταν το τελευταίο νήμα εισέλθει στο φράγμα και ενημερώσει την παρακολουθούμενη περιοχή μνήμης. Προκειμένου να κάνουμε το φράγμα συγχρονισμού επαναχρησιμοποιήσιμο και να αποφύγουμε τα αδιέξοδα, που είναι δυνατόν να προκύψουν όταν αυτό χρησιμοποιείται επαναληπτικά σε ένα βρόχο, καταμετράμε τόσο την άφιξη των νημάτων στο φράγμα όσο και την αναχώρησή τους από αυτό. Χρησιμοποιούμε ατομικές λειτουργίες μείωσης για το σκοπό αυτό. Το φράγμα διατίθεται για μετέπειτα χρήσεις μόνο όταν το εγκαταλείψει και το τελευταίο νήμα. Με αυτόν τον τρόπο, ένα νήμα δεν μπορεί να εισέλθει σε ένα φράγμα αν υπάρχει κάποιο άλλο νήμα το οποίο δεν έχει ακόμα αναχωρήσει από κάποιο προηγούμενο στιγμιότυπο του ίδιου φράγματος.

Οι υπόλοιπες υλοποιήσεις που εξετάσαμε ήταν μια έκδοση φραγμάτων με *εναλλασσόμενη αντίληψη εξόδου*[4] (sense-reversing) [Patterson 03], τα οποία χρησιμοποιούν εσωτερικά βρόχους περιδίνησης με την εντολή PAUSE (*spin-loops*), μια παρόμοια έκδοση η οποία χρησιμοποιεί την εντολή HALT μαζί με τους αντίστοιχους μηχανισμούς για την αποστολή IPIs (*spin-loops-halt*), και η υλοποίηση που παρέχεται από την βιβλιοθήκη NPTL (*pthreads*). Σε αυτές τις εκδόσεις, οι ενέργειες που πραγματοποιούνται από το ενδιάμεσο και το τελευταίο νήμα που εισέρχονται στο φράγμα όσον αφορά τις λειτουργίες συγχρονισμού, είναι αρκετά όμοιες με αυτές που

[4] Σε αυτά τα φράγματα η μεταβλητή που σηματοδοτεί την έξοδο ενός νήματος από το φράγμα (ή αλλιώς, η *αντίληψη εξόδου* του νήματος) *εναλλάσσεται* σε κάθε στιγμιότυπο του φράγματος ανάμεσα σε δύο τιμές, σε μια προσπάθεια να διαφοροποιηθούν μεταξύ τους δύο διαδοχικά στιγμιότυπα ενός φράγματος. Αυτό γίνεται προκειμένου το φράγμα να είναι επαναχρησιμοποιήσιμο και να αποφεύγονται αδιέξοδα σε περίπτωση διαδοχικών κλήσεών του.

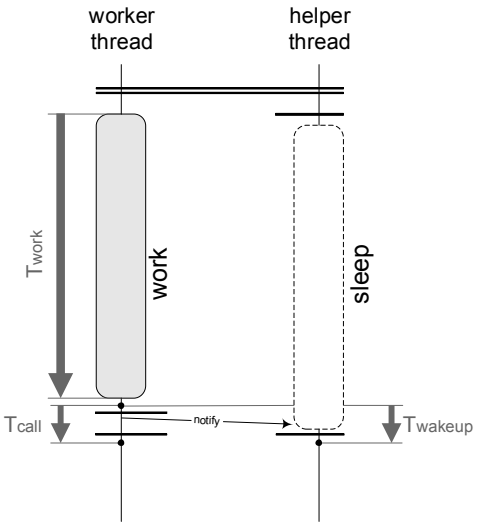
πραγματοποιούνται από το βοηθητικό και το κύριο νήμα, αντίστοιχα, στο σενάριο εκτέλεσης που εξετάσαμε στην ενότητα 4.4.2. Για παράδειγμα, στην υλοποίηση *spin-loops* τα ενδιάμεσα νήματα που βρίσκονται στο φράγμα εκτελούν έναν άεργο βρόχο γύρω από μια μεταβλητή συγχρονισμού, μέχρι να εισέλθει το τελευταίο νήμα και να αλλάξει την τιμή της.

Αρχικά, αξιολογούμε όλες τις υλοποιήσεις σε ένα απλό σχήμα εκτέλεσης όπως αυτό που παρουσιάζεται στο Σχήμα 4.10. Και εδώ θεωρούμε ένα βαρύ και ένα ελαφρύ νήμα τα οποία δρομολογούνται σε διαφορετικούς λογικούς επεξεργαστές του ίδιου φυσικού πακέτου. Το βαρύ νήμα επιτελεί τους ίδιους υπολογισμούς κινητής υποδιαστολής όπως και στο προηγούμενο σενάριο, ενώ το ελαφρύ νήμα δεν κάνει τίποτα. Τα νήματα συγχρονίζονται στο τέλος με ένα φράγμα. Το ελαφρύ νήμα εισέρχεται αμέσως στο φράγμα, περιμένοντας μέχρι να ειδοποιηθεί από το βαρύ νήμα μόλις το τελευταίο ολοκληρώσει τους υπολογισμούς του και εισέλθει και αυτό. Για κάθε υλοποίηση μετράμε τους ακόλουθους χρόνους:

- T_{work}: ο χρόνος που χρειάζεται το νήμα εργασίας να ολοκληρώσει τους υπολογισμούς.
- T_{wakeup}: ο χρόνος ανάμεσα στην άφιξη του νήματος εργασίας στο φράγμα και την αναχώρηση του βοηθητικού νήματος από αυτό.
- T_{call}: ο χρόνος που ξοδεύει το νήμα εργασίας μέσα στο φράγμα.

Ο Πίνακας 4.2 παρουσιάζει τα αποτελέσματα από την αξιολόγηση των διαφορετικών υλοποιήσεων. Έχουμε εκτελέσει το κάθε πείραμα για 50 επαναλήψεις και παρουσιάζουμε τις μέσες τιμές και τις τυπικές αποκλίσεις. Και σε αυτήν την περίπτωση, η υλοποίηση *mwmon* προσφέρει τον καλύτερο συνδυασμό από χαμηλή κατανάλωση πόρων, υψηλή αποκρισιμότητα και χαμηλό κόστος κλήσης. Σε σχέση με την *spin-loops*, το βοηθητικό νήμα στην *mwmon* εισάγει 24% λιγότερη παρεμβολή στο κύριο νήμα εργασίας. Σε σύγκριση με την *pthreads*, που είναι η καλύτερη επιλογή ανάμεσα στις τρεις πιο "φιλικές" ως προς την κατανάλωση πόρων υλοποιήσεις, έχει σχεδόν 4 φορές μικρότερο χρόνο αφύπνισης και 3.46 φορές χαμηλότερο κόστος κλήσης.

Σαν επόμενο βήμα, αξιολογούμε την επίδοση των φραγμάτων στα πλαίσια του λεπτομερούς συγχρονισμού νημάτων με μεταβαλλόμενη ασυμμετρία του φορτίου εργασίας. Για αυτόν τον σκοπό έχουμε κατασκευάσει ένα στοιχειώδες μετροπρόγραμμα, όπου πάλι δύο νήματα εκτελούνται σε διαφορετικά νήματα υλικού ενός Hyper-threaded επεξεργαστή (βλ. Σχήμα 4.11). Σε αυτήν την περίπτωση, θεωρούμε σαν μονάδα εργασίας έναν 10×10 πολλαπλασιασμό πινάκων κινητής υποδιαστολής. Στο εσωτερικό ενός βρόχου, το πρώτο νήμα ("βαρύ") εκτελεί πάντα το μέγιστο ποσό δουλειάς, που είναι 10 μονάδες εργασίας. Το δεύτερο νήμα ("ελαφρύ"), εκτελεί μέσα στο βρόχο μικρότερο ποσό δουλειάς, που κυμαίνεται από 0 έως 10 μονάδες εργασίας. Με άλλα λόγια, στη μία ακραία περίπτωση τα νήματα αυτά είναι εντελώς ασύμμετρα ενώ στην άλλη



Σχήμα 4.10: *Σενάριο εκτέλεσης για την αξιολόγηση των φραγμάτων συγχρονισμού. Το νήμα εργασίας εκτελεί έναν 512×512 πολλαπλασιασμό πινάκων, ενώ το βοηθητικό νήμα δεν κάνει τίποτε άλλο από το να περιμένει μέχρι να ειδοποιηθεί από το πρώτο όταν εκείνο εισέλθει στο φράγμα. Μια αποδοτική υλοποίηση για το θεωρούμενο μοντέλο εφαρμογών πρέπει να προσφέρει μειωμένο T_{work}, T_{wakeup} και T_{call} την ίδια στιγμή.*

Υλοποίηση	**Χρόνος εργασίας (σε δευτερόλεπτα)**	**Χρόνος αφύπνισης (σε κύκλους)**	**Χρόνος κλήσης (σε κύκλους)**
spin-loops	4.3897 (±0.3461)	1236 (±340)	1173 (±338)
spin-loops-halt	3.5720 (±0.2624)	49953 (±11502)	51329 (±11879)
pthreads	3.5917 (±0.2345)	45035 (±3608)	18968 (±1343)
mwmon	3.5266 (±0.2549)	11319 (±1770)	5470 (±644)

Πίνακας 4.2: *Απόδοση των διαφορετικών υλοποιήσεων φραγμάτων συγχρονισμού για το σενάριο εκτέλεσης του Σχήματος 4.10.*

είναι απόλυτα συμμετρικά. Ο βρόχος και για τα δύο νήματα εκτελείται για 10^6 επαναλήψεις, και μεταξύ διαδοχικών επαναλήψεων τα νήματα συγχρονίζονται με φράγματα.

Για όλα τα διαφορετικά επίπεδα του φορτίου εργασίας του ελαφρού νήματος μετράμε το συνολικό χρόνο ολοκλήρωσης του μετροπρογράμματος, για τις διαφορετικές υλοποιήσεις των φραγμάτων συγχρονισμού. Κατά ένα μεγάλο μέρος αυτός ο χρόνος αντανακλά τη ρυθμαπόδοση κάθε έκδοσης φραγμάτων συγχρονισμού, αφού η δουλειά που επιτελείται σε κάθε επανάληψη είναι σχετικά σύντομη. Από αυτή τη φάση αξιολόγησης έχουμε παραλείψει την υλοποίηση *spin-loops-halt*, καθώς έδινε υπερβολικά μεγάλους χρόνους ολοκλήρωσης για αυτό τον αριθμό

επαναλήψεων. Αυτό οφείλεται πιθανώς στο μεγάλο κόστος κλήσης και χρόνο αφύπνισης που χαρακτηρίζει τη συγκεκριμένη υλοποίηση.

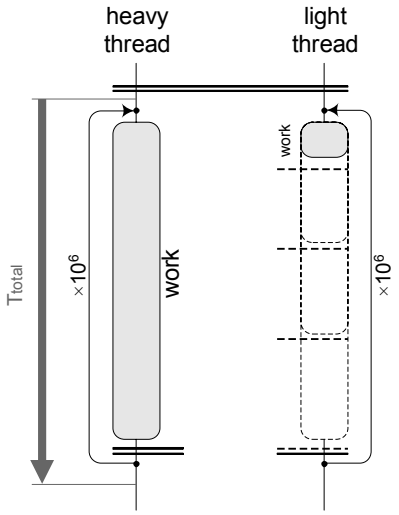
Τα αποτελέσματα παρουσιάζονται στο Σχήμα 4.12. Για όλα τα επίπεδα της ασυμμετρίας των νημάτων, η υλοποίηση *mwmon* ήταν αυτή που υπερίσχυσε έναντι όλων των άλλων κατά έναν σημαντικό παράγοντα. Κατά μέσο όρο, έδωσε 12% μεγαλύτερη ρυθμαπόδοση σε σύγκριση με την *pthreads* και 26% καλύτερη σε σχέση με την *spin-loops*. Όπως ήταν αναμενόμενο, στην περίπτωση της πλήρους ασυμμετρίας (φορτίο εργασίας ελαφρού νήματος ίσο με 0) η *mwmon* σημείωσε τους καλύτερους χρόνους, αφού αυτή η περίπτωση έχει τις μεγαλύτερες περιόδους αναμονής στην εκτέλεση του ελαφρού νήματος, που μπορούν να αναδείξουν με τον καλύτερο τρόπο την ικανότητα της *mwmon* να ελαχιστοποιήσει την κατανάλωση πόρων. Συγκεκριμένα, η *mwmon* παρείχε χρόνους ολοκλήρωσης 1.22 φορές μικρότερους σε σχέση με την *pthreads* και 1.62 σε σχέση με την *spin-loops*.

Καθώς τα προφίλ των νημάτων τείνουν να γίνουν εντελώς συμμετρικά (φορτίο εργασίας ελαφρού νήματος ίσο με 10), η ικανότητα διατήρησης πόρων κάθε υλοποίησης γίνεται λιγότερο σημαντική και αυτό που αποκτά περισσότερη σημασία είναι το κόστος κλήσης και η καθυστέρηση αφύπνισης. Και εδώ, η *mwmon* αποδίδει οριακά καλύτερα σε σύγκριση με την *spin-loops* (κατά έναν παράγοντα 1.02), και αρκετά καλύτερα σε σύγκριση με την *pthreads* (κατά έναν παράγοντα 1.09), εξαιτίας του μεγαλύτερου κόστους κλήσης και χρόνου αφύπνισης που χαρακτηρίζει την υλοποίηση *pthreads* (βλ. Πίνακα 4.2).

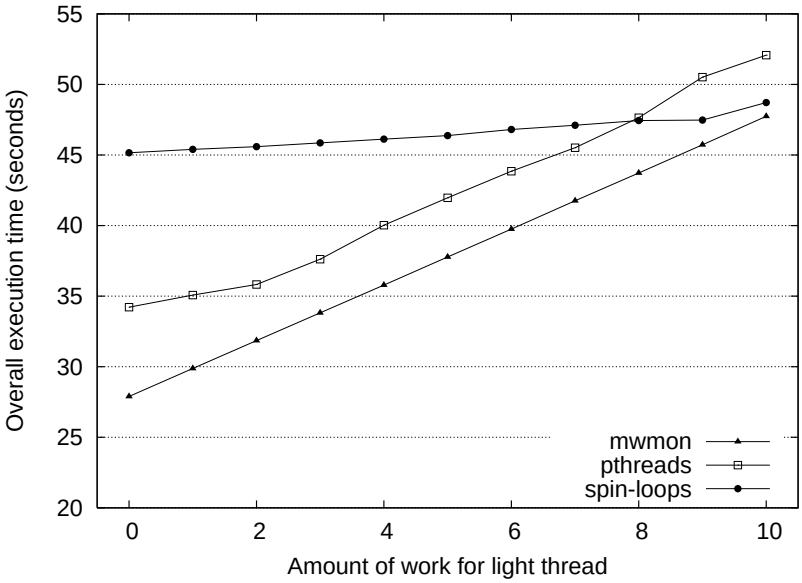
4.4.4 Αξιολόγηση φραγμάτων συγχρονισμού στα πλαίσια του σχήματος SPR

Το τελευταίο βήμα στη διαδικασία αξιολόγησης περιλαμβάνει τη δοκιμή των διαφόρων υλοποιήσεων φραγμάτων συγχρονισμού στα πλαίσια του σχήματος υποθετικής προεκτέλεσης, για μια σειρά από πραγματικές εφαρμογές. Η περιγραφή του σχήματος SPR, καθώς και των περιορισμών που επιβάλλει στην εκτέλεσή του η συγκεκριμένη αρχιτεκτονική που χρησιμοποιήσαμε, έχει γίνει αναλυτικά στην ενότητα 3.4.

Και σε αυτήν την περίπτωση, δημιουργούμε στην αρχή της εκτέλεσης ένα βοηθητικό νήμα το οποίο είναι μόνιμα απεικονισμένο στο δεύτερο λογικό επεξεργαστή του ίδιου φυσικού πακέτου με αυτόν όπου εκτελείται το κύριο νήμα της εφαρμογής. Το βοηθητικό νήμα προτρέχει και προφορτώνει δεδομένα τα οποία θα χρησιμοποιηθούν από το κύριο νήμα στο άμεσο μέλλον. Όποτε έχει προφορτώσει συγκεκριμένη ποσότητα δεδομένων συρρικνώνεται, ώστε να αποτρέπεται από το να τρέχει αρκετά μπροστά και να "μολύνει" την κρυφή μνήμη με νέα δεδομένα. Όπως είδαμε στα προηγούμενα, αυτός ο συγχρονισμός ανάμεσα στα δύο νήματα μπορεί να υλοποιηθεί με φράγματα, ακολουθώντας ένα σχήμα εκτέλεσης όπως αυτό του Σχήματος 4.7. Χαρακτηριστικά λειτουργίας της εκάστοτε υλοποίησης φραγμάτων, όπως ο χρόνος αφύπνισης ή η κατανάλωση



Σχήμα 4.11: *Σενάριο εκτέλεσης για την αξιολόγηση της ρυθμαπόδοσης των φραγμάτων συγχρονισμού. Η μονάδα εργασίας θεωρείται ένας 10×10 πολλαπλασιασμός πινάκων. Μέσα σε ένα βρόχο, το βαρύ νήμα πραγματοποιεί πάντα 10 μονάδες εργασίας (το μέγιστο), ενώ το ελαφρύ πραγματοποιεί μικρότερο ποσό (από 0 μέχρι 10 μονάδες). Ο βρόχος εκτελείται για* 10^6 *επαναλήψεις, και μεταξύ διαδοχικών επαναλήψεων τα νήματα συγχρονίζονται με φράγματα.*



Σχήμα 4.12: *Ρυθμαπόδοση των διαφόρων υλοποιήσεων φραγμάτων συγχρονισμού για διαφορετικά επίπεδα ασυμμετρίας των νημάτων (βλ. Σχήμα 4.11).*

πόρων, επηρεάζουν πλευρές λειτουργίας του σχήματος SPR όπως η ικανότητα κάλυψης αστοχιών και η επιβάρυνση που υφίσταται το κύριο νήμα εργασίας.

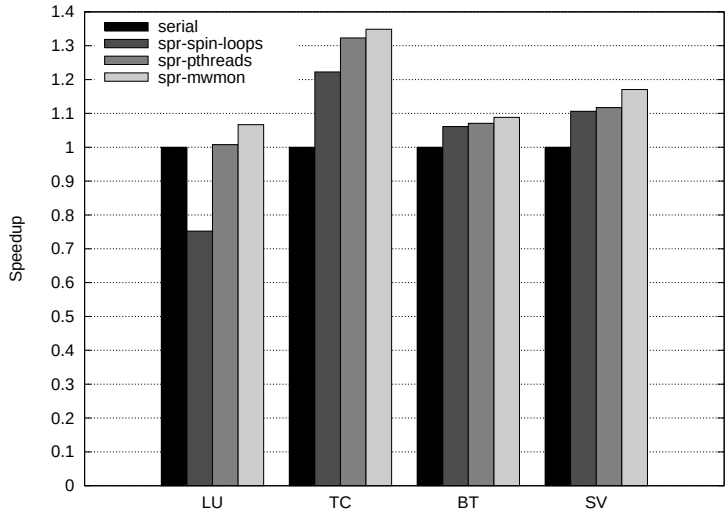
Εφαρμογή	Δεδομένα εισόδου
LU	πίνακας 2048×2048, μπλοκ 10×10
TC	1600 κορυφές, 25000 πλευρές, μπλοκ 16×16
BT	Class A
SV	πίνακας 9648×77137, 260785 μη μηδενικά στοιχεία

Πίνακας 4.3: *Μετροπρογράμματα προς αξιολόγηση.*

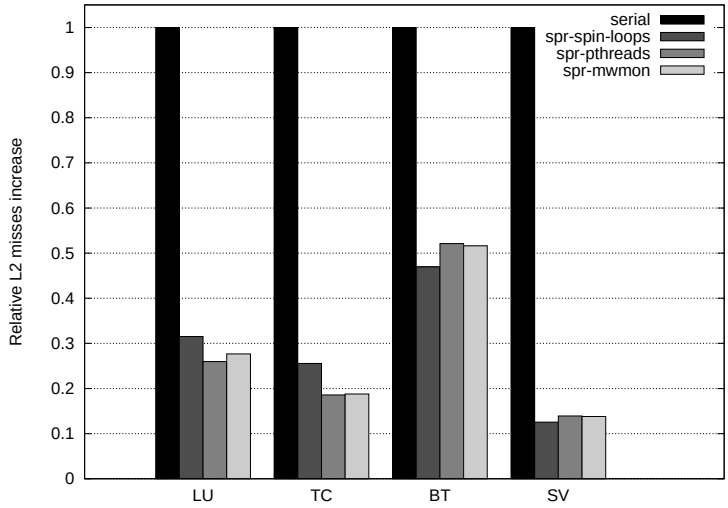
Χρησιμοποιήσαμε τέσσερα από τα μετροπρογράμματα που παρουσιάσαμε στην ενότητα 3.5.2 προκειμένου να αξιολογήσουμε τις διαφορετικές υλοποιήσεις φραγμάτων συγχρονισμού: τα LU, TC, BT και SV. Ο Πίνακας 4.3 συνοψίζει τα μετροπρογράμματα και τα δεδομένα εισόδου τους. Από αυτά, το BT είχε την χειρότερη απόδοση στην κρυφή μνήμη στην αρχική, σειριακή του έκδοση, με έναν λόγο αστοχιών L2-προς-L1 (ενδεικτικό του ποσοστού αστοχιών στην L2) ίσο με 20.4%. Ακολουθούν τα TC και SV με λόγο 11.7% και 7.3%, αντίστοιχα, και τελευταίο έρχεται το LU με λόγο σχεδόν ίσο με 1.8%.

Το Σχήμα 4.13 παρουσιάζει την επιτάχυνση στις εφαρμογές που πέτυχε κάθε μία από τις υλοποιήσεις *spin-loops*, *pthreads* και *mwmon*, στα πλαίσια του σχήματος SPR. Το Σχήμα 4.14 δείχνει τον αριθμό των αστοχιών στην L2 για το κύριο νήμα εργασίας, κανονικοποιημένο ως προς τη σειριακή εκτέλεση, που είναι μια άμεση ένδειξη της ικανότητας κάλυψης αστοχιών της κάθε υλοποίησης. Οι αριθμοί αυτοί συλλέχθηκαν χρησιμοποιώντας τους μετρητές απόδοσης του επεξεργαστή. Επιπλέον, προκειμένου να αποκτήσουμε πληρέστερη εικόνα για το προφίλ εκτέλεσης των νημάτων, μετρήσαμε για κάθε εφαρμογή το μέρος των κύκλων που αφιερώθηκαν σε χρήσιμους υπολογισμούς ή που σπαταλήθηκαν σε αναμονή στα φράγματα συγχρονισμού. Αυτός ο καταμερισμός κύκλων απεικονίζεται στο Σχήμα 4.15. Σε αυτό το διάγραμμα οι περίοδοι εργασίας και αναμονής σημειώνονται σαν *work* και *synch*, αντίστοιχα.

Περιμένουμε ότι η καλύτερη απόδοση για το SPR σχήμα μπορεί να επιτευχθεί κάτω από ικανοποιητική κάλυψη των αστοχιών, και ταυτόχρονα αδιάλειπτη εργασία του κύριου νήματος (ελάχιστος χρόνος στα φράγματα συγχρονισμού) και μικρό φορτίο εργασίας του νήματος προφόρτωσης. Ο τελευταίος παράγοντας οδηγεί στις λιγότερες δυνατές συγκρούσεις για μοιραζόμενους πόρους και σε μεγάλες άεργες περιόδους, όπου το όφελος από τις υλοποιήσεις που χαρακτηρίζονται από χαμηλή κατανάλωση πόρων μπορεί να μεγιστοποιηθεί. Σε κάθε περίπτωση, η εγγενής δυναμική κάθε εφαρμογής να ευνοηθεί από την τέλεια τοπικότητα στην κρυφή μνήμη παίζει πάντα σημαντικό ρόλο. Ο τρόπος που συμβιβάζονται οι παράγοντες αυτοί και το βάρος του καθενός στη διαμόρφωση της τελικής απόδοσης του SPR σχήματος σε πραγματικές αρχιτεκτονικές SMT αποτελεί αντικείμενο μελλοντικής έρευνας.



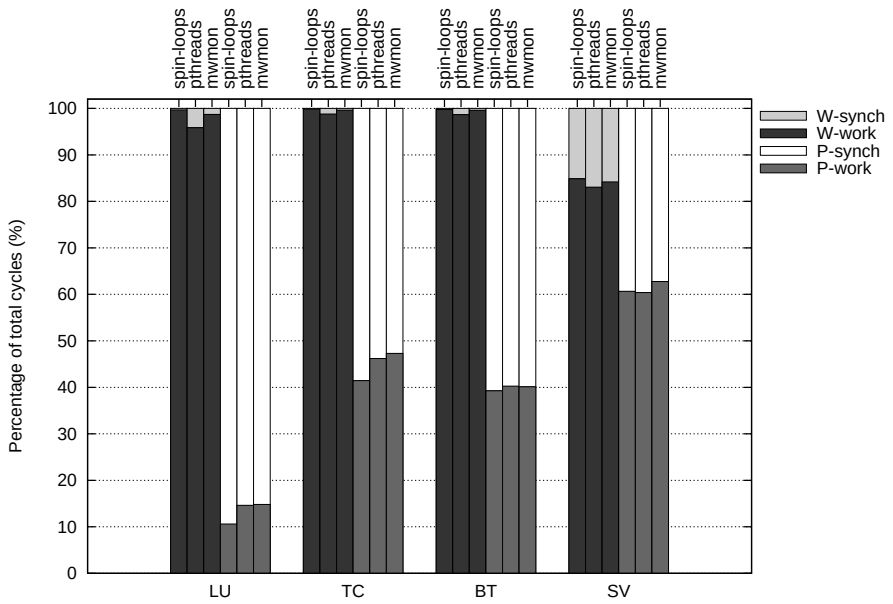
Σχήμα 4.13: *Επιτάχυνση στο σχήμα SPR με τις διαφορετικές υλοποιήσεις φραγμάτων συγχρονισμού*



Σχήμα 4.14: *Κάλυψη αστοχιών στο σχήμα SPR με τις διαφορετικές υλοποιήσεις φραγμάτων συγχρονισμού.*

Με μια πρώτη ματιά, το SPR σχήμα με τα φράγματα *mwmon* υπερτερεί έναντι όλων των άλλων εκδόσεων, επιτυγχάνοντας επιταχύνσεις μεταξύ 1.07 (LU) και 1.35 (TC). Η μέση επιτάχυνση για όλα τα μετροπρογράμματα ήταν 1.17. Κατά μέσο όρο, η *mwmon* υλοποίηση πέτυχε 15% καλύτερους χρόνους εκτέλεσης σε σχέση με την *spin-loops*, και 3.6% σε σχέση με την *pthreads*.

Η TC είναι η εφαρμογή που επωφελείται περισσότερο από το SPR, πιθανώς επειδή απολαμβάνει μεγάλων μειώσεων στις αστοχίες της L2, έχοντας αρχικά κακή τοπικότητα αναφορών. Παρομοίως, ενώ το BT φαίνεται να υποφέρει και αυτό από κακή τοπικότητα στην κρυφή μνήμη



Σχήμα 4.15: *Καταμερισμός κύκλων για τα νήματα εργασίας(W) και τα νήματα προφόρτωσης(P) στο σχήμα SPR.*

(λαμβάνοντας υπόψη το μεγάλο ποσοστό αστοχιών του), λαμβάνει περιορισμένη βελτίωση από το SPR, πιθανώς λόγω της ανεπαρκούς κάλυψης των αστοχιών.

Τα SV και LU είναι δύο μετροπρογράμματα όπου η υλοποίηση *mwmon* προσφέρει σημαντική βελτίωση της απόδοσης σε σύγκριση με τις υπόλοιπες υλοποιήσεις. Το LU, εξαιτίας της ήδη καλής τοπικότητας, φαίνεται να είναι περισσότερο ευαίσθητο στο "θόρυβο" που εισάγει το νήμα προφόρτωσης παρά στη μείωση των αστοχιών της L2 που επιτυγχάνεται, και η υλοποίηση *mwmon* δείχνει να είναι η επιλογή που ικανοποιεί καλύτερα αυτή την απαίτηση. Το γεγονός αυτό επιβεβαιώνεται από το ότι η υλοποίηση *spin-loops* επηρεάζει την απόδοση του LU αρνητικά.

Στο SV, από την άλλη, όχι μόνο υπάρχει σημαντική παρεμβολή στην εκτέλεση του κύριου νήματος εξαιτίας του μεγάλου φορτίου εργασίας του νήματος προφόρτωσης, αλλά το κύριο νήμα καθυστερείται επιπλέον σε γεγονότα συγχρονισμού για αξιόλογο μέρος του χρόνου εκτέλεσής του. Παρόλα αυτά, η σημαντική μείωση των αστοχιών μαζί με τα οφέλη από τη διατήρηση πόρων της *mwmon*, καθιστούν την *mwmon* την καλύτερη επιλογή και σε αυτήν την περίπτωση.

4.5 Συμπεράσματα και Μελλοντικές Κατευθύνσεις

Σε αυτό το κεφάλαιο εξετάσαμε τη δυνατότητα χρήσης των εντολών MONITOR/MWAIT για αποδοτικό συγχρονισμό νημάτων μιας εφαρμογής με ασύμμετρο φορτίο, όταν αυτή εκτελείται σε επεξεργαστές με τεχνολογία Hyper-threading. Παρουσιάσαμε ένα πλαίσιο υλοποίησης

μέσω του οποίου παρέχεται η δυνατότητα στο χρήστη να χρησιμοποιήσει με αποδοτικό τρόπο αυτές τις προνομιούχες εντολές, με το μικρότερο δυνατό κρίσιμο μονοπάτι στον πυρήνα και χωρίς πλεονάζουσες αντιγραφές. Βασιζόμενοι σε αυτή την υποδομή, υλοποιήσαμε πρωτογενείς λειτουργίες συγχρονισμού αναμονής για συνθήκη και ειδοποίησης, οι οποίες μπορούν να χρησιμοποιηθούν σαν δομικές λίθοι για πιο σύνθετες λειτουργίες συγχρονισμού. Σε σύγκριση με τη βιβλιοθήκη Pthreads καθώς και άλλες υλοποιήσεις βασισμένες σε βρόχους περιδίνησης, οι προτεινόμενες λειτουργίες αποδείχτηκαν αρκετά αποδοτικές για το θεωρούμενο μοντέλο εκτέλεσης, κάνοντας τον καλύτερο συμβιβασμό ανάμεσα στην χαμηλή κατανάλωση πόρων του επεξεργαστή και μικρούς χρόνους κλήσης και αφύπνισης.

Σαν επόμενο βήμα, υλοποιήσαμε φράγματα συγχρονισμού χρησιμοποιώντας τις προτεινόμενες πρωτογενείς λειτουργίες. Και σε αυτήν την περίπτωση η υλοποίησή μας εξισορρόπησε με τον καλύτερο τρόπο τις απαιτήσεις που θέτει το θεωρούμενο μοντέλο εκτέλεσης, και επιπρόσθετα, προσέφερε την καλύτερη ρυθμαπόδοση για τον λεπτομερή συγχρονισμό ανάμεσα στα νήματα, για όλα τα διαφορετικά επίπεδα ασυμμετρίας του φορτίου τους. Αξιόλογα κέρδη στην απόδοση σημειώθηκαν όταν δοκιμάσαμε τα προτεινόμενα φράγματα συγχρονισμού σε ένα ρεαλιστικό σενάριο, δηλαδή στα πλαίσια του σχήματος της υποθετικής προεκτέλεσης που εφαρμόστηκε σε τέσσερα μετροπρογράμματα.

Σαν μελλοντική εργασία, σκοπεύουμε να επεκτείνουμε το προτεινόμενο πλαίσιο προκειμένου να υποστηρίζονται πολλαπλές μεταβλητές συγχρονισμού βασισμένες στις MONITOR/MWAIT στα πλαίσια της ίδιας εφαρμογής, αλλά και για να υποστηρίζεται ο συγχρονισμός σε πλατφόρμες με πολλαπλούς επεξεργαστές SMT. Αυτό θα μπορούσε να γίνει, για παράδειγμα, αν χρησιμοποιήσουμε τοπικά, ανά φυσικό πακέτο φράγματα συγχρονισμού βασισμένα στις MONITOR/MWAIT, με σκοπό να κατασκευάσουμε σε υψηλότερο επίπεδο μεγαλύτερα δενδρικά φράγματα, καθολικά ως προς το σύνολο των επεξεργαστών της πλατφόρμας.

Επιπρόσθετα, σκοπεύουμε να αξιολογήσουμε τις προτεινόμενες λειτουργίες συγχρονισμού σε παράλληλα προγράμματα τα οποία έχουν απαιτήσεις για πολύ συχνό, λεπτομερή συγχρονισμό, καθώς και σε άλλες περιπτώσεις που αντανακλούν το μοντέλο εκτέλεσης παραγωγού-καταναλωτή. Τέτοιες είναι φερειπείν περιπτώσεις MPI προγραμμάτων στις οποίες στοχεύουμε στην αποδοτική επικάλυψη υπολογισμών και επικοινωνίας (στα πρότυπα του [Goumas 09a]), το μοντέλο εκτέλεσης DSWP [Rangan 04] όπου επιδιώκεται η αποτελεσματική επικάλυψη παράλληλων υπολογισμών σε κώδικες με αναδρομικές δομές δεδομένων, εφαρμογές απαιτητικές σε είσοδο/έξοδο, συστήματα χρόνου εκτέλεσης για διαχείριση παραλληλισμού βασισμένου σε εργασίες (task-level parallelism), κ.λπ..

Σε κάθε περίπτωση, άποψή μας είναι ότι με την καθιέρωση αρχιτεκτονικών που ενσωματώνουν μεγάλο αριθμό νημάτων υλικού μέσα σε ένα ολοκληρωμένο, οι μηχανισμοί συγχρονισμού πρέπει να είναι όσο το δυνατόν πιο κοντά στο επίπεδο του υλικού, έχοντας γνώση της

αρχιτεκτονικής και ακολουθώντας πιθανώς μια ιεραρχία αντίστοιχη με αυτήν της διάταξης των νημάτων υλικού. Αυτό θα συντελέσει ώστε ο συγχρονισμός να είναι κλιμακώσιμος για μεγάλο αριθμό νημάτων και να μην αποτελεί στενωπό στην απόδοση των πολυνηματικών εφαρμογών.

ΚΕΦΑΛΑΙΟ 5

Παραλληλοποίηση με Χρήση Μνήμης Διενεργειών

5.1 Εισαγωγή

Παρόλη τη μεγάλη πρόοδο που σημειώνεται διαρκώς στο επίπεδο υλικού ένα σημαντικό ζήτημα παραμένει το πώς θα μπορέσει το λογισμικό να εκμεταλλευτεί την πληθώρα πυρήνων που διαθέτουν οι σύγχρονοι επεξεργαστές για την επιτάχυνση μιας σειριακής εφαρμογής. Όπως σημειώνεται χαρακτηριστικά στο [Sutter 05], *"η «επανάσταση» που φέρνουν οι πολυπύρηνοι επεξεργαστές, είναι πρωτίστως «επανάσταση» στο λογισμικό (...) το δύσκολο πρόβλημα δεν είναι η κατασκευή πολυπύρηνων συστημάτων, αλλά ο προγραμματισμός τους με τέτοιο τρόπο που επιτρέπει και στις πιο κοινές εφαρμογές να επωφεληθούν από την αυξανόμενη απόδοση των επεξεργαστών"*. Ο παράλληλος προγραμματισμός θέτει μια σειρά από νέες προκλήσεις στον προγραμματιστή. Οι βασικότερες από τις προκλήσεις αυτές σήμερα είναι η *εξαγωγή του παραλληλισμού* από τις εφαρμογές, η *έκφραση του παραλληλισμού*, και ο *συγχρονισμός*. Στην πραγματικότητα, οι προκλήσεις αυτές αποτελούσαν πεδίο έρευνας από τότε που αναπτύχθηκαν τα πρώτα πολυεπεξεργαστικά συστήματα κοινής μνήμης. Όμως, αφορούσαν κυρίως πολύ συγκεκριμένες κατηγορίες προγραμμάτων αφήνοντας έξω έναν μεγάλο αριθμό εφαρμογών. Με τη ραγδαία εξάπλωση των πολυπύρηνων αρχιτεκτονικών τα τελευταία χρόνια οι προκλήσεις αυτές γίνονται επίκαιρες όσο ποτέ και χρήζουν προσέγγισης και αντιμετώπισης από την αρχή.

Στις αμέσως επόμενες ενότητες περιγράφουμε συνοπτικά την πρόοδο που έχει επιτευχθεί μέχρι σήμερα γύρω από τα θέματα αυτά, καθώς και τη δική μας προσέγγιση στον τρόπο χρήσης των μοντέλων και των μηχανισμών που έχουν προκύψει από τη σχετική έρευνα. Στις παραγράφους που ακολουθούν στη συνέχεια, παρουσιάζουμε αναλυτικότερα το προτεινόμενο σχήμα παραλληλοποίησης και δείχνουμε τον τρόπο εφαρμογής του στην περίπτωση μιας δύσκολα παραλληλοποιήσιμης εφαρμογής, του αλγορίθμου εύρεσης συντομότερων διαδρομών του Dijkstra [Dijkstra 59].

5.1.1 Το πρόβλημα της εξαγωγής παραλληλισμού

Ο εντοπισμός του παραλληλισμού στη σειριακή εφαρμογή και η εξαγωγή ανεξάρτητων δομών και περιοχών που μπορούν να ανατεθούν σε παράλληλα νήματα εκτέλεσης, αποτελούσε ανέκαθεν βασικό πρόβλημα για τους προγραμματιστές. Οι εξαρτήσεις δεδομένων των περιοχών του προγράμματος που δυνητικά μπορούν να αποτελέσουν παράλληλες εργασίες, άλλοτε είναι εμφανείς με απλή επισκόπηση, και άλλοτε απαιτούν σχολαστική ανάλυση ή αλγοριθμική γνώση του προβλήματος από τον προγραμματιστή για να αποδειχθεί η ύπαρξή τους. Προς αυτήν την κατεύθυνση η ερευνητική κοινότητα έχει εστιάσει το ενδιαφέρον της στην *αυτόματη παραλληλοποίηση* (automatic parallelization). Σε αυτήν την περίπτωση, ένας *μεταγλωττιστής παραλληλοποίησης* (parallelizing compiler) αναλύει το σειριακό πρόγραμμα και εντοπίζει ανεξάρτητες περιοχές κατάλληλου μεγέθους για παράλληλη εκτέλεση.

Οι μεταγλωττιστές παραλληλοποίησης εφαρμόζουν εξελιγμένους αλγορίθμους για την ανάλυση εξαρτήσεων και την εκτίμηση κόστους των παράλληλων εργασιών. Έτσι είναι σε θέση πολλές φορές να κάνουν καλύτερες επιλογές από τον χρήστη, παραλληλοποιώντας τμήματα κώδικα που δε φαίνονται εξαρχής ανεξάρτητα ή επιλέγοντας παράλληλες εργασίες με βέλτιστο μέγεθος. Παρόλα αυτά, οι μεταγλωττιστές παραλληλοποίησης έχουν φανεί μέχρι τώρα επιτυχείς μόνο σε μερικές κατηγορίες εφαρμογών, και συγκεκριμένα σε επιστημονικούς κώδικες που παραδοσιακά εκτελούνταν σε υπερυπολογιστές και βασίζονται σε πίνακες και δομές βρόχων με μεγάλη κανονικότητα [Goff 91, Bugnion 96]. Αποτυγχάνουν να παραλληλοποιήσουν πολλές άλλες εφαρμογές, όπως κώδικες με ακανόνιστα μοτίβα πρόσβασης στη μνήμη, περίπλοκες ροές ελέγχου και δυναμικές εξαρτήσεις δεδομένων. Τέτοιοι κώδικες είναι λόγου χάρη προγράμματα με έντονη χρήση δεικτών προς αντικείμενα δεσμευμένα στο σωρό (heap-allocated objects), αναδρομικές δομές δεδομένων (π.χ. δένδρα, συνδεδεμένες λίστες, γραφήματα), έμμεσες αναφορές στη μνήμη (indirect references), και γενικά μη στατικές εξαρτήσεις, που η ύπαρξή τους ή όχι δεν μπορεί να αποδειχθεί εκ των προτέρων και εξαρτάται από δυναμικές παραμέτρους όπως τα δεδομένα εισόδου του προγράμματος. Σε αυτές τις περιπτώσεις, ακόμα και ο πιο εξελιγμένος μεταγλωττιστής παραλληλοποίησης λειτουργεί συντηρητικά και εξαιρεί τέτοιες περιοχές από

την παραλληλοποίηση.

Για αυτού του είδους τις εφαρμογές έχει προταθεί στη βιβλιογραφία το μοντέλο του *υποθετικού πολυνηματισμού* (speculative multithreading), ή αλλιώς, της *υποθετικής εκτέλεσης επιπέδου νημάτων* (thread-level speculation – TLS) [Sohi 95, Hammond 98, Steffan 98, Krishnan 99, Steffan 00, Renau 05, Tian 08]. Σύμφωνα με το μοντέλο αυτό, τμήματα ενός προγράμματος μπορούν να εκτελεστούν παράλληλα και υποθετικά, έτσι ώστε αν η εκτέλεση γίνει σωστά, οι υπολογισμοί που έκαναν τα υποθετικά νήματα να μπορούν να επικυρωθούν και να μη χρειάζεται να επαναληφθούν από το κύριο, μη-υποθετικό νήμα. Οι δομές του προγράμματος στις οποίες στοχεύει ο υποθετικός πολυνηματισμός είναι συνήθως δομημένα μπλοκ, όπως επαναλήψεις βρόχων, συναρτήσεις ή δομές ελέγχου "if-then-else".

Πιο συγκεκριμένα, τα τμήματα του σειριακού προγράμματος διασπώνται σε επιμέρους εργασίες οι οποίες ανατίθενται και εκτελούνται παράλληλα από διαφορετικά νήματα. Η προγενέστερη εργασία στη σειρά προγράμματος είναι η μόνη *μη-υποθετική* (non-speculative), υπό την έννοια ότι από αυτήν καθορίζεται ανά πάσα στιγμή η κατάσταση της εκτέλεσης. Όλες οι υπόλοιπες (αν υπάρχουν) είναι *υποθετικές* (speculative), καθώς εκτελούνται χωρίς να υπάρχουν εγγυήσεις για το αν οι τιμές που χρησιμοποιούν για τους υπολογισμούς τους είναι σωστές, εξαιτίας εξαρτήσεων δεδομένων που ενδέχεται να προκύψουν με προγενέστερες εργασίες. Ένας υποκείμενος μηχανισμός, υλοποιημένος συνήθως σε υλικό, αναλαμβάνει την όλη διαχείριση της υποθετικής εκτέλεσης: διατηρεί την αρχιτεκτονική κατάσταση κάθε υποθετικής εργασίας (speculative state), παρακολουθεί τυχόν παραβιάσεις εξαρτήσεων μεταξύ των εργασιών, *ακυρώνει* (squash) όσες από αυτές προκαλούν παραβιάσεις (συνήθως τις μεταγενέστερες), και *επικυρώνει* (validate) αυτές που εκτελέστηκαν χωρίς πρόβλημα.

Από την μέχρι τώρα ερευνητική δραστηριότητα γύρω από την ιδέα του υποθετικού πολυνηματισμού έχουν προκύψει διάφορα μοντέλα που διαφέρουν μεταξύ τους ως προς τα χαρακτηριστικά, τη στόχευση και τις δυνατότητες. Ωστόσο, οι πλέον αποτελεσματικές προσεγγίσεις βασίζονται σε ειδικούς και εξιδανικευμένους μηχανισμούς υλικού, που στοχεύουν κυρίως στην αποδοτική διαχείριση των υποθετικών εργασιών. Τα υπάρχοντα πολυπύρηνα συστήματα όμως, καθώς και οι τάσεις που διαγράφονται στην τεχνολογία των επεξεργαστών, βρίσκονται ακόμη πολύ μακριά από το ενσωματώσουν αντίστοιχους μηχανισμούς στο υλικό. Ως εκ τούτου, η προοπτική της ρητής υποστήριξης κάποιας μορφής υποθετικού πολυνηματισμού από τους επεξεργαστές στα επόμενα χρόνια φαντάζει δύσκολη.

5.1.2 Το πρόβλημα του συγχρονισμού

Ο συγχρονισμός των νημάτων μιας παράλληλης εφαρμογής στην πρόσβαση στα κοινά δεδομένα αποτελεί μία από τις σημαντικότερες προκλήσεις του παράλληλου προγραμματισμού.

Παραδοσιακά, οι προγραμματιστές παράλληλων συστημάτων χρησιμοποιούν μηχανισμούς *αμοιβαίου αποκλεισμού* (mutual exclusion), όπως τα κλειδώματα, προκειμένου να επιβάλλουν την ατομική εκτέλεση μιας προστατευόμενης (κρίσιμης) περιοχής, επιτρέποντας σε ένα νήμα κάθε φορά να εισέλθει στην περιοχή. Τέτοιοι μηχανισμοί είναι γνωστό ότι συχνά ενέχουν κινδύνους και μπορούν να οδηγήσουν σε προβλήματα.

Τα *αδρομερή κλειδώματα* (coarse-grain locks) σε μεγάλα τμήματα κώδικα με χρήση μίας καθολικής μεταβλητής-κλειδιού, είναι απλοϊκά στη σύλληψη και εύκολα στην υλοποίηση, όμως αναστέλλουν την παράλληλη εκτέλεση των νημάτων και περιορίζουν τον εκμεταλλεύσιμο παραλληλισμό. Στις περισσότερες περιπτώσεις συνεπώς δεν μπορούν να προσφέρουν κλιμακωσιμότητα στην απόδοση της εφαρμογής. Αντίθετα, τα *λεπτομερή κλειδώματα* (fine-grain locks) με χρήση ξεχωριστών κλειδιών για κάθε στοιχειώδες διαμοιραζόμενο αντικείμενο (π.χ. για τα διαφορετικά στοιχεία ενός πίνακα) επιτρέπουν μεγαλύτερα επίπεδα παραλληλισμού, κλιμακώνουν καλά, αλλά είναι δύσκολα στη σύλληψη και επιρρεπή σε προγραμματιστικά λάθη, οδηγώντας συχνά σε προβλήματα όπως τα *αδιέξοδα*, *στάσιμα* (deadlocks) ή *ενεργά* (livelocks). Επιπλέον, εισάγουν συνολικά μεγάλο κόστος κλήσης ενώ δεν επιτρέπουν τη *σύνθεση* (composition) επιμέρους ατομικών λειτουργιών σε μεγαλύτερες.

Μερικοί έμπειροι προγραμματιστές αφιερώνουν την προσπάθειά τους στη συγγραφή παράλληλων αλγορίθμων και δομών δεδομένων *απελευθερωμένων από κλειδώματα* (lock-free). Αυτοί οι αλγόριθμοι αποφεύγουν τις *συνθήκες ανταγωνισμού* (race conditions) ανάμεσα στα νήματα χρησιμοποιώντας καινοτόμα αλγοριθμικά πρότυπα αντί για αμοιβαίο αποκλεισμό. Ωστόσο, η σχεδίαση τέτοιων αλγορίθμων απαιτεί βαθιά κατανόηση του αντίστοιχου σειριακού αλγορίθμου καθώς και της υποκείμενης πλατφόρμας. Κατά συνέπεια, οι αλγόριθμοι που προκύπτουν είναι δύσκολοι στην κατανόηση και στηρίζονται σε αρχιτεκτονικές υποθέσεις για τη συγκεκριμένη πλατφόρμα που προορίζονται. Εκτός αυτού, προϋποθέτουν πολλές φορές πρόσθετους υπολογισμούς για την αποφυγή των συνθηκών ανταγωνισμού.

Για την αντιμετώπιση των προβλημάτων προγραμματισμού και κλιμακωσιμότητας που έχει ο βασισμένος σε κλειδώματα συγχρονισμός, η ερευνητική κοινότητα έχει στρέψει το ενδιαφέρον της τα τελευταία χρόνια σε εναλλακτικούς μηχανισμούς συγχρονισμού, με πιο εξέχοντα αυτόν της *μνήμης διενεργειών* (transactional memory – TM) [Herlihy 93, Adl-Tabatabai 06]. Η μνήμη διενεργειών επιτρέπει τον *μη-ανασταλτικό συγχρονισμό* (non-blocking synchronization) ανάμεσα στα νήματα μιας παράλληλης εφαρμογής, ενώ ταυτόχρονα υπόσχεται σημαντική ευκολία στον προγραμματισμό. Δίνει τη δυνατότητα στον προγραμματιστή να εκφράσει σε υψηλό επίπεδο περιοχές του προγράμματος που επιθυμεί να εκτελεστούν *ατομικά* (σαν μια ενιαία λειτουργία) και *απομονωμένα* (ανεπηρέαστες από άλλες ταυτόχρονα εκτελούμενες τέτοιες περιοχές).

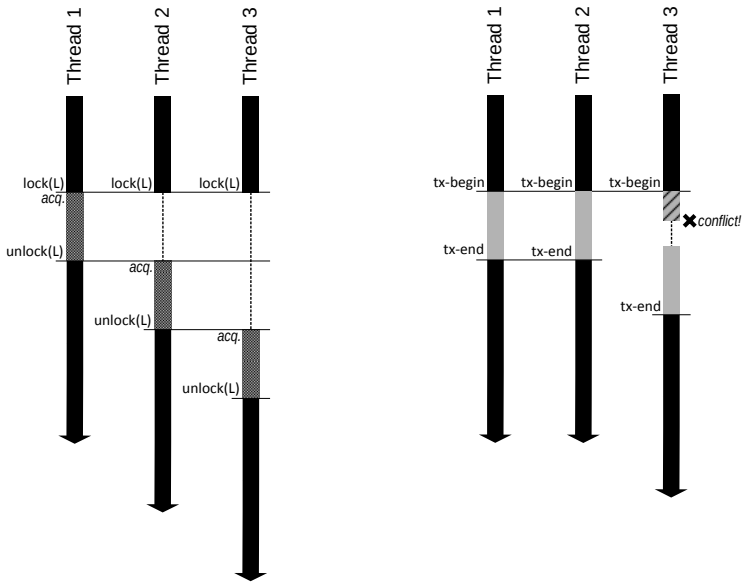
Η αλληλουχία των λειτουργιών μνήμης (αναγνώσεις ή εγγραφές) που πραγματοποιούνται μέσα σε κάθε τέτοια προστατευόμενη περιοχή είναι γνωστή σαν *διενέργεια μνήμης* (memory

transaction) [1]. Μία διενέργεια μνήμης είτε εκτελείται μέχρι τέλους είτε δεν εκτελείται καθόλου. Το υποκείμενο *σύστημα μνήμης διενεργειών*, υλοποιημένο σε *υλικό* (hardware transactional memory – HTM) ή *λογισμικό* (software transactional memory – STM) επιτρέπει την ταυτόχρονη είσοδο των νημάτων σε μια προστατευόμενη περιοχή και παρακολουθεί τις λειτουργίες τους. Αν κάποιο νήμα τροποποιήσει εντός μιας διενέργειας μια θέση μνήμης που έχει διαβαστεί ή τροποποιηθεί από διενέργεια άλλου νήματος, τότε προκύπτει *διένεξη* (conflict). Όταν το σύστημα εντοπίζει διενέξεις ανάμεσα σε δύο ή περισσότερες διενέργειες, επιτρέπει σε ένα μόνο από τα νήματα να *ολοκληρώσει* (commit) τη διενέργεια και επιβάλλει στα υπόλοιπα να *ματαιώσουν* (abort) και να επανεκτελέσουν τις δικές τους, αφού πρώτα ακυρώσουν όποιες αλλαγές έκαναν στην αρχιτεκτονική κατάσταση του συστήματος (τιμές καταχωρητών και περιεχόμενα μνήμης) μέχρι εκείνη τη στιγμή.

Λειτουργώντας επιτρεπτικά, σύμφωνα με τον παραπάνω τρόπο, η μνήμη διενεργειών καθιστά δυνατό τον *αισιόδοξο συγχρονισμό* (optimistic synchronization) παράλληλων περιοχών στις οποίες *δεν αναμένεται* να σημειωθούν διενέξεις κατά το χρόνο εκτέλεσης, οδηγώντας στην αξιοποίηση περισσότερου παραλληλισμού. Την ίδια στιγμή, ο χρήστης επικεντρώνει την προσπάθειά του στην περιγραφή των απαιτήσεων συγχρονισμού στο πρόγραμμα (σε επίπεδο περιοχών κώδικα, αντικειμένων, λειτουργιών, ή και συνθέσεών τους), χωρίς να χρειάζεται να ασχοληθεί με τις λεπτομέρειες υλοποίησης του συγχρονισμού.

Στο Σχήμα 5.1 παρουσιάζεται ένα τυπικό σενάριο χρήσης της μνήμης διενεργειών για ατομική εκτέλεση μιας κρίσιμης περιοχής μιας παράλληλης εφαρμογής. Στην αριστερή εικόνα η είσοδος των νημάτων στην κρίσιμη περιοχή ελέγχεται με μηχανισμό κλειδώματος. Τα νήματα επιχειρούν να αποκτήσουν ταυτόχρονα το κλειδί και να εισέλθουν στην περιοχή. Όμως ο μηχανισμός κλειδώματος εξασφαλίζει ότι ένα το πολύ νήμα μπορεί να κατέχει το κλειδί ανά πάσα στιγμή, με αποτέλεσμα η εκτέλεση της κρίσιμης περιοχής να σειριοποιείται. Αυτή η προσέγγιση στο συγχρονισμό θεωρείται συντηρητική, αφού αν τα νήματα προσπελάζουν κοινές θέσεις μνήμης αρκετά σπάνια, ο παραλληλισμός περιορίζεται χωρίς να υπάρχει πραγματικά ανάγκη. Ακόμα και στην περίπτωση που ο συγχρονισμός είναι πιο λεπτομερής με αποτέλεσμα να μην περιορίζει πολύ τον παραλληλισμό, το συναθροιστικό κόστος των κλήσεων στα λεπτομερή κλειδώματα μπορεί να εισάγει τελικά σημαντική επιβάρυνση, τη στιγμή που οι προσβάσεις στα δεδομένα μπορεί πάλι να είναι μη-συγκρουόμενες. Η μνήμη διενεργειών, αντίθετα, αποτελεί την αισιόδοξη προσέγγιση αφού επιτρέπει στα νήματα να εισέλθουν ταυτόχρονα στην προστατευόμενη περιοχή, με την ελπίδα ότι δε θα επεξεργαστούν τα ίδια δεδομένα. Αν αυτή είναι η συνήθης περίπτωση στην εκτέλεση μιας εφαρμογής τότε η μνήμη διενεργειών έχει –θεωρητικά–

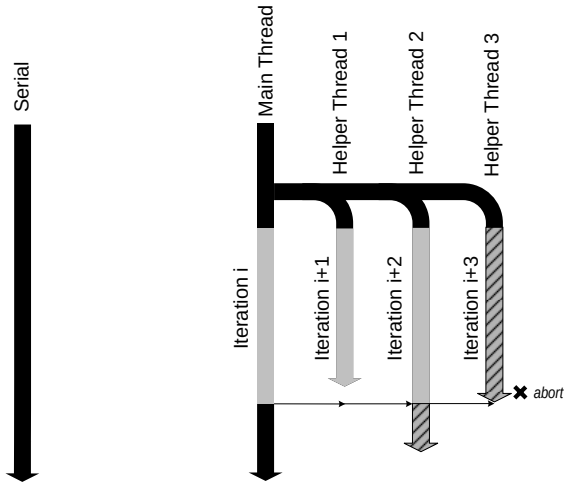
[1] Στη σχετική βιβλιογραφία έχει επικρατήσει ο όρος *διενέργεια* (transaction) να χρησιμοποιείται για την περιγραφή της ίδιας της προστατευόμενης περιοχής του προγράμματος.



Σχήμα 5.1: *Αμοιβαίος αποκλεισμός με χρήση κλειδωμάτων (αριστερά), έναντι μη-ανασταλτικού συγχρονισμού με χρήση μνήμης διενεργειών (δεξιά).*

πλεονέκτημα σε σχέση με τον αμοιβαίο αποκλεισμό, αφού οι λειτουργίες μνήμης εντός των περιοχών πραγματοποιούνται χωρίς καμία (σχεδόν) επιβάρυνση ενώ οι περίοδοι αναστολής ενός νήματος (λόγω ενεργειών ανάκαμψης από διενέξεις) είναι σαφώς μικρότερες.

Τα τελευταία χρόνια η βιομηχανία έχει αρχίσει να εκδηλώνει έμπρακτα το ενδιαφέρον της για συστήματα μνήμης διενεργειών με υλοποιήσεις υλικού όπως ο επεξεργαστής Rock της Sun [Tremblay 08], προδιαγραφές για επεκτάσεις στο σύνολο εντολών όπως το Advanced Synchronization Facility από την AMD [AMD 09], σχέδια από πλευράς της Intel για υλοποίηση στο υλικό μηχανισμών επιτάχυνσης STM συστημάτων στα πρότυπα του HaSTM [Saha 06b], και περιβάλλοντα λογισμικού όπως το STM και ο μεταγλωττιστής της Intel [Saha 06a, Ni 08]. Η μνήμη διενεργειών δεν είναι "πανάκεια" για τα προβλήματα του παράλληλου προγραμματισμού καθώς έχει αρκετές αδυναμίες που σχετίζονται με την επιβάρυνση που εισάγουν οι εσωτερικοί μηχανισμοί του συστήματος μνήμης διενεργειών στην εκτελούμενη εφαρμογή. Λόγω τέτοιων πρακτικών ζητημάτων, ο τρόπος με τον οποίον η μνήμη διενεργειών μπορεί να βελτιώσει την εκτέλεση πραγματικών, μη-στοιχειωδών παράλληλων εφαρμογών δεν είναι πάντοτε σαφής και αποτελεί αντικείμενο προς διερεύνηση [Scott 07, Watson 07, Kang 09]. Παρόλα αυτά, είναι κοινή παραδοχή ότι συνιστά ένα από τα πιο σημαντικά βήματα προς την απλοποίηση του παράλληλου προγραμματισμού τα τελευταία χρόνια και την υιοθέτησή του από τον μέσο προγραμματιστή.



Σχήμα 5.2: *Χρήση μνήμης διενεργειών και βοηθητικών νημάτων για υποθετική παραλληλοποίηση μιας εφαρμογής. Οι μελλοντικές επαναλήψεις ενός βρόχου ανατίθενται διαδοχικά σε διαφορετικά βοηθητικά νήματα και εκτελούνται με υποθετικό τρόπο εντός διενεργειών.*

5.1.3 Προχωρώντας ένα βήμα παραπέρα: υποθετική παραλληλοποίηση με χρήση μνήμης διενεργειών

Στα πλαίσια αυτής της διατριβής εξετάζουμε έναν εναλλακτικό τρόπο χρήσης της μνήμης διενεργειών. Προτείνουμε ένα σχήμα υποθετικής παραλληλοποίησης που βασίζεται στη συνδυασμένη χρήση μνήμης διενεργειών και βοηθητικών νημάτων. Με βάση αυτό καταφέρνουμε να επιταχύνουμε τον αλγόριθμο εύρεσης συντομότερων διαδρομών του Dijkstra, μια περίπτωση δύσκολα παραλληλοποιήσιμης εφαρμογής που διαθέτει τα χαρακτηριστικά που περιγράψαμε στην ενότητα 5.1.1.

Το τυπικό σενάριο χρήσης της μνήμης διενεργειών που παρουσιάσαμε στο Σχήμα 5.1 προϋποθέτει ότι η εφαρμογή έχει ήδη παραλληλοποιηθεί με βάση κάποιο προγραμματιστικό μοντέλο. Αντίθετα, στο προτεινόμενο σχήμα ξεκινάμε από τη σειριακή εφαρμογή και προσπαθούμε να την παραλληλοποιήσουμε χρησιμοποιώντας τις δυνατότητες που μας δίνει η μνήμη διενεργειών, με έναν τρόπο που ακολουθεί περισσότερο τη λογική των μοντέλων υποθετικού πολυνηματισμού που έχουν προταθεί στη βιβλιογραφία. Το προτεινόμενο σχήμα απεικονίζεται στο Σχήμα 5.2. Σε αυτό, οι μελλοντικές επαναλήψεις ενός βρόχου της σειριακής εφαρμογής ανατίθενται διαδοχικά σε διαφορετικά βοηθητικά νήματα και εκτελούνται παράλληλα εντός διενεργειών, *με την ελπίδα* ότι δεν υφίστανται εξαρτήσεις δεδομένων ανάμεσά τους. Όπως αναφέραμε στην ενότητα 5.1.1, οι εξαρτήσεις για αυτήν τη κατηγορία εφαρμογών είναι δύσκολο ή αδύνατο να αποδειχθούν λόγω της δυναμικής φύσης τους και για αυτό το λόγο επιλέγουμε να καταφύγουμε σε υποθέσεις. Αν όντως δε σημειωθούν διενέξεις ανάμεσα στις διενέργειες, τότε οι υποθετικοί

υπολογισμοί των βοηθητικών νημάτων έχουν στηριχθεί σε σωστές τιμές εισόδου και άρα είναι έγκυροι, με αποτέλεσμα το κύριο νήμα να απαλλάσσεται από αυτούς τους υπολογισμούς και να βελτιώνεται η απόδοση της εφαρμογής. Διαφορετικά, αν σημειωθούν διενέξεις, που σημαίνει ότι κάποιο νήμα διαβάζει ή τροποποιεί δεδομένα που έχουν τροποποιηθεί από κάποιο άλλο, είναι πολύ πιθανό ότι παραβιάζονται οι εξαρτήσεις που υπαγορεύονται από τη σειρά του προγράμματος. Σε αυτήν την περίπτωση επικυρώνονται οι ενέργειες του προγενέστερου (στη σειρά προγράμματος) από τα "συγκρουόμενα" νήματα και ακυρώνονται των υπολοίπων. Οι επαναλήψεις που ακυρώθηκαν θα εκτελεστούν κάποια στιγμή στο μέλλον.

Η λειτουργία αυτού του μοντέλου θα πρέπει να είναι τέτοια ώστε να μην παρεμποδίζεται η εκτέλεση του κύριου νήματος υπολογισμών. Ακόμα και στην χειρότερη περίπτωση όπου οι εξαρτήσεις δεδομένων είναι πολύ συχνές ή όταν δεν υπάρχει αρκετή δουλειά για τα βοηθητικά νήματα, η λειτουργία τους δε θα πρέπει τελικά να επιβαρύνει την εκτέλεση του κύριου νήματος οδηγώντας σε χειρότερους χρόνους σε σχέση με τη σειριακή περίπτωση. Στην εφαρμογή που εξετάσαμε τα βοηθητικά νήματα κατάφεραν όντως να απαλλάξουν το κύριο νήμα από ένα σημαντικό μέρος των υπολογισμών του, ενώ όπου αυτό δεν ήταν εφικτό οι καθυστερήσεις που εισήγαγαν ήταν ελάχιστες.

5.2 Διατύπωση και Προσέγγιση του Προβλήματος

Ο αλγόριθμος του Dijkstra είναι ένας βασικός αλγόριθμος γραφημάτων που επιλύει το πρόβλημα της εύρεσης *συντομότερων διαδρομών από μία αρχική κορυφή* (single-source shortest paths – SSSP) για γραφήματα με μη μηδενικά βάρη πλευρών. Το SSSP είναι ένα κλασικό πρόβλημα συνδυαστικής βελτιστοποίησης που χρησιμοποιείται σε διάφορα πεδία εφαρμογών, όπως δρομολόγηση σε δίκτυα και σχεδίαση VLSI συστημάτων. Ο αλγόριθμος διατηρεί ένα σύνολο σταθεροποιημένων κορυφών, S, των οποίων οι συντομότερες διαδρομές από την αρχική κορυφή έχουν ήδη υπολογιστεί. Σε κάθε επανάληψη του αλγορίθμου επιλέγεται η μη-σταθεροποιημένη κορυφή με τη μικρότερη απόσταση από το S, εισάγεται στο S και ανανεώνονται οι αποστάσεις της από τις γειτονικές της κορυφές. Το σύνολο των μη-σταθεροποιημένων κορυφών υλοποιείται σαν μία *ουρά προτεραιότητας* (priority queue). Η υλοποίηση της ουράς προτεραιότητας σαν ένα *δυαδικό σωρό* (binary heap) δίνει χρονική πολυπλοκότητα ίση με $O((|E|+|V|log|V|)$, όπου $|E|$ είναι ο αριθμός των πλευρών του γραφήματος και $|V|$ ο αριθμός των κορυφών.

Από την πλευρά της υλοποίησης, ο αλγόριθμος περιλαμβάνει ένα διπλά εμφωλευμένο βρόχο: ο εξωτερικός βρόχος διατρέχει όλες τις κορυφές του γραφήματος εξάγοντας κάθε φορά από την ουρά την πλησιέστερη κορυφή στο σύνολο S, ενώ ο εσωτερικός βρόχος ανανεώνει τις αποστάσεις από το S όλων των γειτόνων της εξαχθείσας κορυφής. Το γεγονός ότι οι εξαγωγές

από την ουρά πρέπει να γίνονται σειριακά προκειμένου να τηρείται η σημασιολογία του αλγορίθμου, έχει σαν αποτέλεσμα τη σειριοποίηση μεγάλου μέρους του λειτουργιών του αλγορίθμου, καθιστώντας τον έτσι δύσκολο προς παραλληλοποίηση [Brodal 98, Meyer 98].

Για την υλοποίηση παράλληλων εκδόσεων του αλγορίθμου οι ερευνητές έχουν ακολουθήσει μέχρι τώρα δύο γενικές στρατηγικές. Η πρώτη στρατηγική επιχειρεί να χαλαρώσει την αυστηρή σειριακή φύση του αλγορίθμου δημιουργώντας περισσότερο παραλληλισμό στον εξωτερικό βρόχο. Αυτό οδηγεί σε εναλλακτικούς αλγορίθμους, όπως ο Δ-stepping [Meyer 98, Madduri 06], που επιτρέπουν την ταυτόχρονη εξαγωγή πολλαπλών κορυφών από το σύνολο των μη-σταθεροποιημένων κορυφών. Η συνέπεια αυτής της αύξησης του παραλληλισμού είναι μια αύξηση στον αριθμό των βημάτων που απαιτούνται από το σειριακό αλγόριθμο, μιας και μερικές από τις σταθεροποιημένες κορυφές μπορεί να χρειαστεί να επανατοποθετηθούν στο σύνολο των μη-σταθεροποιημένων κορυφών και να επανεξεταστούν. Η ακραία έκφραση αυτής της προσέγγισης είναι ο αλγόριθμος Bellman-Ford [Bellman 58, Ford 62], που εκτελείται σε $|V|$ εξωτερικά βήματα με άφθονο παραλληλισμό, τα οποία όμως όμως περιλαμβάνουν πολύ περισσότερες λειτουργίες με αποτέλεσμα η χρονική πολυπλοκότητα να είναι αρκετά χειρότερη σε σχέση με αυτή του Dijkstra ($O(|E||V|)$).

Η δεύτερη στρατηγική λειτουργεί πάνω στον αλγόριθμο του Dijkstra αυτόν καθεαυτόν και αναζητεί τον παραλληλισμό στον εσωτερικό βρόχο, μέσω ταυτόχρονων προσβάσεων στην ουρά προτεραιότητας. Αυτή η προσέγγιση συσχετίζεται άμεσα με ένα μεγάλο μέρος θεωρητικής έρευνας πάνω σε παράλληλες ουρές προτεραιότητας (π.χ., [Brodal 98]), οι οποίες όμως δεν οδηγούν σε αποδοτικές υλοποιήσεις σε σύγχρονες παράλληλες αρχιτεκτονικές. Επιπλέον, πρακτικές υλοποιήσεις ταυτόχρονων δυαδικών σωρών σαν ουρές προτεραιότητας [Hunt 96] βασίζονται σε αναπόφευκτα λεπτομερή κλειδώματα στο δυαδικό σωρό. Αυτές μπορεί να εξασφαλίζουν δικαιοσύνη και σχετική βελτίωση της απόδοσης για εφαρμογές με ανεξάρτητα νήματα που προσπελάζουν μοιραζόμενα δεδομένα, όμως αναμένεται ότι θα βλάψουν την λεπτομερή παραλληλοποίηση ενός αλγορίθμου όπως ο Dijkstra. Επιπρόσθετα με το κλείδωμα της ουράς προτεραιότητας, τα νήματα που ανανεώνουν παράλληλα τους γείτονες μιας κορυφής θα πρέπει να συγχρονιστούν στο τέλος του εσωτερικού βρόχου, όπως θα εξηγήσουμε στα επόμενα, μέσω ενός κοινού φράγματος. Από την εμπειρία μας, η έντονη χρήση φραγμάτων συγχρονισμού σε μια παράλληλη εφαρμογή μπορεί να αποτελέσει την στενωπό στην κλιμάκωση της απόδοσης, ειδικά για μεγάλους αριθμούς νημάτων.

5.2.1 Η προτεινόμενη προσέγγιση

Στα πλαίσια αυτής της διατριβής ασχολούμαστε με τον αλγόριθμο του Dijkstra και προσπαθούμε να αντιμετωπίσουμε τις προκλήσεις της παραλληλοποίησής του σε μια πολυπύρηνη αρχιτεκτονική. Σημειώνουμε ότι στόχος μας δεν είναι να προτείνουμε κάποιον πιο αποδοτικό

αλγόριθμο ή υλοποίηση για το SSSP πρόβλημα, αλλά να δοκιμάσουμε την εφαρμοσιμότητα και την αποτελεσματικότητα διαφόρων τεχνικών σε ένα δύσκολα παραλληλοποιήσιμο πρόβλημα όπως ο αλγόριθμος του Dijkstra. Σε αυτήν την κατεύθυνση, χρειάζεται να αντιμετωπίσουμε δύο βασικά ζητήματα εγγενή στον αλγόριθμο, αυτό του *περιορισμένου προφανούς παραλληλισμού* και αυτό του *συχνού συγχρονισμού*.

Προκειμένου να μειώσουμε το κόστος του συγχρονισμού χρησιμοποιούμε τη μνήμη διενεργειών, σαν έναν μηχανισμό για αποδοτικό έλεγχο των ταυτόχρονων προσβάσεων στις διαμοιραζόμενες δομές δεδομένων. Τα προγραμματιστικά μοντέλα που υποστηρίζουν τη χρήση μνήμης διενεργειών δίνουν τη δυνατότητα στον προγραμματιστή να εσωκλείσει τμήματα του κώδικα εντός διενεργειών, δηλώνοντας με αυτόν τον τρόπο ότι εντός αυτών των τμημάτων γίνονται λειτουργίες σε θέσεις μνήμης στις οποίες ενδέχεται να επενεργήσουν και άλλα νήματα. Το σύστημα μνήμης διενεργειών παρακολουθεί τις διενέργειες των νημάτων, και αν δύο ή περισσότερα από αυτά πραγματοποιήσουν συγκρουόμενες προσπελάσεις, αποφασίζει πώς να διαχειριστεί αυτή τη διένεξη. Η συνήθης προσέγγιση είναι να επιτρέψει σε ένα από τα συγκρουόμενα νήματα να ολοκληρώσει τη διενέργειά του και να επιβάλει στα υπόλοιπα να ματαιώσουν και να επανεκτελέσουν τις δικές τους. Γενικά, η μνήμη διενεργειών φαίνεται ενδιαφέρουσα προσέγγιση για δυναμικές δομές δεδομένων και εφαρμογές με πολλαπλά και ανεξάρτητα νήματα. Απομένει ωστόσο να διερευνηθεί κατά πόσο μπορεί να συμβάλει στην επιτάχυνση ενός σειριακού αλγορίθμου.

Η παραλληλοποίηση του εσωτερικού βρόχου του Dijkstra δεν αρκεί για να παραχθεί επαρκής παραλληλισμός. Ως εκ τούτου, καταφεύγουμε στην ιδέα του βοηθητικού νηματισμού προκειμένου να αυξήσουμε την αδρομέρεια του παραλληλισμού και εξετάζουμε εάν η ενσωμάτωση των βοηθητικών νημάτων είναι καλή στρατηγική για να επιτύχουμε βελτιώσεις στην απόδοση. Η βασική ιδέα είναι να χρησιμοποιήσουμε έναν αριθμό νημάτων τα οποία θα αποφορτίζουν το κύριο νήμα από ένα μέρος των αρχικών λειτουργιών του με διαφανή τρόπο. Συγκεκριμένα, όσο το κύριο νήμα εξάγει την κορυφή στην κεφαλή της ουράς προτεραιότητας και ανανεώνει τους γείτονές της, k βοηθητικά νήματα ανανεώνουν τους γείτονες των επόμενων k κορυφών στην ουρά. Αυτή η προσέγγιση εκμεταλλεύεται τον παραλληλισμό στον εξωτερικό βρόχο, ενώ όπως θα δούμε και στα επόμενα δε θέτει σε κίνδυνο τη σημασιολογία του αλγορίθμου.

Υλοποιήσαμε διάφορες πολυνηματικές εκδόσεις του αλγορίθμου του Dijkstra χρησιμοποιώντας παραδοσιακές μεθόδους συγχρονισμού (κλειδώματα και φράγματα συγχρονισμού), μνήμη διενεργειών και βοηθητικά νήματα. Τις αξιολογήσαμε σε περιβάλλον προσομοίωσης βασισμένο στα συστήματα Simics [Magnusson 02] και GEMS [Martin 05, gem 08], τα οποία επιτρέπουν την προσομοίωση πολυπύρηνων συστημάτων και παρέχουν υποστήριξη για μνήμη διενεργειών στο υλικό. Τα αποτελέσματά μας δείχνουν ότι ο συνδυασμός μνήμης διενεργειών και βοηθητικού νηματισμού επιτυγχάνει σημαντικά επίπεδα επιτάχυνσης για μία δύσκολα παραλληλοποιήσιμη εφαρμογή όμως ο αλγόριθμος του Dijkstra, απαιτώντας απλές, σχετικά, επεκτάσεις στον αρχικό σειριακό κώδικα.

5.3 Σχετικές Εργασίες

Οι ανανεώσεις των αποστάσεων στην ουρά προτεραιότητας καταναλώνουν μεγάλο ποσοστό του χρόνου του Dijkstra. Για αυτό το λόγο, το να επιτρέψουμε τις ταυτόχρονες προσβάσεις σε αυτή τη δομή δεδομένων φαίνεται καλή προσέγγιση για την επίτευξη υψηλότερης απόδοσης. Μεγάλο μέρος της θεωρητικής έρευνας επικεντρώνεται στον ορισμό μιας δομής δεδομένων με αποδοτικές παράλληλες λειτουργίες για την ουρά προτεραιότητας, στο μοντέλο εκτέλεσης Parallel Random Access Machine (PRAM – [JáJá 92]): `Insert`, `ExtractMin`, `DecreaseKey`, `Delete`, και άλλες (π.χ. `Merge`) [Cormen 01]. Η `DecreaseKey` είναι η πιο σημαντική λειτουργία στον αλγόριθμο του Dijkstra. Μια παράλληλη ουρά προτεραιότητας μπορεί να χρησιμοποιήσει πολλαπλούς επεξεργαστές για να επιταχύνει τις λειτουργίες σε ένα απλό αντικείμενο [Brodal 99] και να υλοποιήσει παράλληλες λειτουργίες `Insert`, `Delete` [Chen 94, Pinotti 96] ή `DecreaseKey` [Brodal 98]. Οι Brodal και άλλοι στο [Brodal 98] ανάμεσα στα άλλα συζητούν την εφαρμοσιμότητα της προσέγγισής τους στον αλγόριθμο του Dijkstra. Ωστόσο, οι παραπάνω προσεγγίσεις εστιάζουν στη μείωση της θεωρητικής ασυμπτωτικής συμπεριφοράς των λειτουργιών της παράλληλης ουράς προτεραιότητας στο PRAM μοντέλο, και δεν μπορούν να αποτελέσουν τη βάση για μια αποδοτική υλοποίηση με άμεσο τρόπο του αλγορίθμου του Dijkstra σε σύγχρονες πολυπύρηνες πλατφόρμες. Από την άλλη, οι Hunt και άλλοι στο [Hunt 96] πραγματεύονται την υλοποίηση μιας ταυτόχρονης ουράς προτεραιότητας, βασισμένης σε δυαδικούς σωρούς, η οποία υποστηρίζει παράλληλες λειτουργίες `Insert` και `Delete` χρησιμοποιώντας λεπτομερή κλειδώματα στους κόμβους του σωρού. Εφόσον οι παραπάνω λειτουργίες δε διατρέχουν ολόκληρη τη δομή δεδομένων, τα τοπικά, λεπτομερή κλειδώματα οδηγούν στη βελτίωση της απόδοσης. Όμως αυτά τα λεπτομερή κλειδώματα στην περίπτωση της λειτουργίας `DecreaseKey` η οποία πραγματοποιεί ευρείες διατρέξεις στη δομή υποβαθμίζουν σημαντικά την απόδοση, εκτός κι αν η υποκείμενη πλατφόρμα υποστηρίζει εξειδικευμένους μηχανισμούς συγχρονισμού στο υλικό.

Η σειρά με την οποία ο αλγόριθμος του Dijkstra εξάγει κορυφές από την ουρά προτεραιότητας περιορίζει τον παραλληλισμό. Για να εκθέσουμε περισσότερο παραλληλισμό θα ήταν επιθυμητό να μπορούμε να εξάγουμε ταυτόχρονα αρκετές κορυφές από την ουρά. Αυτό μπορεί να επιτευχθεί αν αρκετές κορυφές έχουν ίσες αποστάσεις από το σύνολο S. Έτσι, αν η ουρά προτεραιότητας είναι οργανωμένη σε "κάδους" με κορυφές ίσων αποστάσεων τότε η εξαγωγή και οι ανανεώσεις των γειτόνων μπορεί να γίνουν παράλληλα για κάθε κάδο (αλγόριθμος του Dial [Dial 69]). Μία γενίκευση του αλγορίθμου του Dial, ο αλγόριθμος Δ-stepping, προτάθηκε από τους Meyer και Sanders [Meyer 98]. Σε αυτήν την περίπτωση, κάθε κάδος δεν περιέχει ισαπέχουσες κορυφές από το σύνολο S, αλλά κορυφές με απόσταση μεταξύ $i\Delta$ και $(i+1)\Delta$, όπου Δ είναι ένας θετικός πραγματικός αριθμός που εκφράζει το εύρος του κάδου. Η εξαγωγή όλων των κορυφών που περιέχονται στον πλησιέστερο στο S κάδο γίνεται παράλληλα, ενώ οι ανανεώσεις

των γειτόνων οδηγούν σε μετακινήσεις κορυφών ανάμεσα στους κάδους. Εξαιτίας αυτής της επιτρεπτής ευελιξίας στο εύρος των κάδων, ο αλγόριθμος μπορεί να χρειαστεί να επανεισάγει σε κάποιον κάδο μια κορυφή που εξήγαγε σε προηγούμενο βήμα. Ο αλγόριθμος τερματίζει όταν όλοι οι κάδοι αδειάσουν.

Οι Madduri και άλλοι στο [Madduri 06] χρησιμοποιούν τον Δ-stepping σαν τον αλγόριθμο βάσης στον Cray MTA-2, μια πολυνηματική αρχιτεκτονική με δυνατότητα εκμετάλλευσης λεπτομερούς παραλληλισμού μέσω λειτουργιών συγχρονισμού στο υλικό, και επιτυγχάνουν σημαντικά μεγέθη επιτάχυνσης για το SSSP πρόβλημα. Στην Parallel Boost Graph Library [Edmonds 06], ο αλγόριθμος του Dijkstra παραλληλοποιείται για μια μηχανή κατανεμημένης μνήμης. Η στρατηγική εδώ είναι ο διαμοιρασμός της ουράς προτεραιότητας στις τοπικές μνήμες των κόμβων του συστήματος και η διαίρεση του αλγορίθμου σε υπερ-βήματα ("supersteps") στα οποία κάθε κόμβος εξάγει μια κορυφή από την τοπική ουρά. Με αυτόν τον τρόπο οι εξαγωγές και οι ανανεώσεις των γειτόνων πραγματοποιούνται κι εδώ παράλληλα, με αδρομερή τρόπο. Στο τέλος κάθε υπερ-βήματος, οι κορυφές που εξήχθησαν λανθασμένα από την ουρά προτεραιότητας (επειδή οι ανανεώσεις οδήγησαν άλλες κορυφές πιο κοντά στο σύνολο S) επανεισάγονται στις ουρές. Οι προαναφερθείσες προσεγγίσεις βασίζονται σε σημαντικές τροποποιήσεις στις δομές και τη λειτουργία του αλγορίθμου του Dijkstra προκειμένου να αυξήσουν την αδρομέρεια του παραλληλισμού και να οδηγήσουν σε υποσχόμενες παράλληλες υλοποιήσεις. Στα πλαίσια αυτής της διατριβής, αντίθετα, επικεντρωνόμαστε στην "αυθεντική" έκδοση του αλγορίθμου, προσπαθώντας να αντιμετωπίσουμε τις προκλήσεις της παραλληλοποίησής του και να εξετάσουμε τη δυναμική της μνήμης διενεργειών και του βοηθητικού πολυνηματισμού. Συνεπώς, η ευρύτερη αξιολόγηση και ενσωμάτωση των τεχνικών που πραγματευόμαστε εδώ και σε άλλους SSSP αλγορίθμους αφήνεται σαν δουλειά για το μέλλον.

Η μνήμη διενεργειών έχει προσελκύσει έντονα το ερευνητικό ενδιαφέρον τα τελευταία χρόνια, κυρίως όσον αφορά το σχεδιασμό και την υλοποίηση HTM και STM συστημάτων. Ωστόσο, η αποτελεσματικότητά της σε ένα ευρύ σύνολο πραγματικών, μη-στοιχειωδών εφαρμογών βρίσκεται υπό διερεύνηση. Παρόλο που η μνήμη διενεργειών φαίνεται αναμφίβολα καλή επιλογή για εφαρμογές που είναι ήδη παραλληλοποιημένες με βάση κάποιο μοντέλο και πραγματοποιούν ακανόνιστες ταυτόχρονες προσπελάσεις, η παραλληλοποίηση ενός απλού αλγορίθμου χρησιμοποιώντας τους "αισιόδοξους" μηχανισμούς της μοιάζει αρκετά πιο περίπλοκη διαδικασία.

Οι Scott και άλλοι στο [Scott 07] χρησιμοποιούν τη μνήμη διενεργειών για να παραλληλοποιήσουν τον αλγόριθμο τριγωνοποίησης Delaunay. Ο αλγόριθμος αυτός εξετάζει και ανανεώνει τρίγωνα ανάμεσα σε τριάδες σημείων από ένα πολύ μεγάλο σύνολο δεδομένων, οπότε το να χρησιμοποιηθεί η μνήμη διενεργειών για να αποφευχθούν τα (σημασιολογικά) επιβεβλημένα αλλά σπανίως χρειαζούμενα λεπτομερή κλειδώματα, φαντάζει καλή επιλογή. Η υλοποίηση είναι βασισμένη σε STM σύστημα και πράγματι επιδεικνύει καλή κλιμακωσιμότητα σε ένα σύστημα

με 16 επεξεργαστές. Οι Watson και άλλοι στο [Watson 07] χρησιμοποιούν τη μνήμη διενεργειών για να παραλληλοποιήσουν τον αλγόριθμο δρομολόγησης του Lee. Ο αλγόριθμος περιλαμβάνει τον υπολογισμό ανεξάρτητων διαδρομών μέσα σε ένα πλέγμα σημείων. Αν αυτός ο υπολογισμός προστατεύεται εντός μιας διενέργειας, τότε πολλοί υπολογισμοί διαδρομών μπορούν να εκτελεστούν παράλληλα χωρίς περαιτέρω συγχρονισμό.

Οι Kang και Bader στο [Kang 09] παρουσιάζουν έναν παράλληλο αλγόριθμο για υπολογισμό του ελάχιστου γεννητικού δάσους σε αραιά γραφήματα. Κάθε νήμα υπολογίζει ανεξάρτητα ένα ελάχιστο γεννητικό δέντρο (με βάση τον αλγόριθμο του Prim) αρχίζοντας από μια τυχαία κορυφή. Αν ένα νήμα επιχειρήσει να προσθέσει μια κορυφή στο δέντρο κάποιου άλλου, τότε τα δύο δέντρα συγχωνεύονται. Για τον εντοπισμό τέτοιων επικαλύψεων ανάμεσα σε δύο ή περισσότερα δέντρα οι συγγραφείς χρησιμοποιούν μνήμη διενεργειών. Η υλοποίησή τους κλιμακώνει πολύ καλά σε ένα σύστημα με 64 νήματα υλικού, αλλά η μεγάλη επιβάρυνση που εισάγει το STM σύστημα δεν επιτρέπει τελικά βελτιώσεις σε σχέση με το σειριακό αλγόριθμο.

Στην ίδια κατεύθυνση με τις προηγούμενες περιπτώσεις, εστιάζουμε το ενδιαφέρον μας στον αλγόριθμο του Dijkstra, εφόσον διαθέτει όλα εκείνα τα χαρακτηριστικά που καθιστούν κατάλληλη την υιοθέτηση της μνήμης διενεργειών (δυναμικές δομές δεδομένων, ακανόνιστες προσβάσεις μνήμης). Ωστόσο, ο συγκεκριμένος αλγόριθμος θέτει αρκετά επίπεδα δυσκολίας παραπάνω και νέες προκλήσεις, σαν αποτέλεσμα της εγγενώς σειριακής φύσης του.

5.4 Παραλληλοποίηση του Αλγορίθμου του Dijkstra

5.4.1 Περιγραφή του σειριακού αλγορίθμου

Όπως αναφέρθηκε στα προηγούμενα, ο αλγόριθμος του Dijkstra επιλύει το SSSP πρόβλημα για ένα κατευθυνόμενο γράφημα με μη μηδενικά βάρη πλευρών. Συγκεκριμένα, έστω $G = (V, E)$ ένα κατευθυνόμενο γράφημα με $n = |V|$ κορυφές, $m = |E|$ πλευρές, και $w : E \rightarrow \mathbf{R}^+$ μια συνάρτηση βάρους που αναθέτει μη μηδενικά πραγματικά βάρη στις πλευρές του G. Για κάθε κορυφή v, το SSSP πρόβλημα υπολογίζει το $\delta(v)$, το βάρος της συντομότερης διαδρομής από μία αρχική κορυφή s στη v. Το βάρος της διαδρομής είναι το άθροισμα των βαρών των πλευρών από τις οποίες αποτελείται. Αν η v δεν είναι προσπελάσιμη από την s, τότε $\delta(v) = \infty$. Για κάθε κορυφή v, ο αλγόριθμος διατηρεί μια *εκτίμηση απόστασης*, $d(v)$, που είναι ένα άνω όριο για το πραγματικό βάρος της συντομότερης διαδρομής από την s στη v, το $\delta(v)$. Αρχικά, η $d(v)$ τίθεται ίση με ∞ και μέσα από διαδοχικές *χαλαρώσεις πλευρών* (edge relaxations) μειώνεται σταδιακά, συγκλίνοντας τελικά στην $\delta(v)$. Η χαλάρωση μιας πλευράς (v, w) θέτει την $d(w)$ ίση με $min\{d(w), d(v) + w(v, w)\}$, και έχει το νόημα ότι ο αλγόριθμος δοκιμάζει αν μπορεί να μειώσει περαιτέρω το βάρος της συντομότερης διαδρομής από την s στην w πηγαίνοντας μέσω της v.

Ο αλγόριθμος διατηρεί μια διαμέριση του συνόλου κορυφών V σε *σταθεροποιημένες* (settled), *εκκρεμείς* (queued) και *μη διατρεγμένες* (unreached) κορυφές (οι τελευταίες δύο κατηγορίες συνιστούν τις *μη-σταθεροποιημένες* κορυφές). Οι σταθεροποιημένες κορυφές είναι αυτές για τις οποίες $d(v) = \delta(v)$ · οι εκκρεμείς έχουν $d(v) > \delta(v)$ και $d(v) \neq \infty$ · οι μη διατρεγμένες έχουν $d(v) = \infty$. Αρχικά, μόνο η s είναι εκκρεμής με $d(s) = 0$ ενώ όλες οι άλλες κορυφές είναι μη διατρεγμένες. Το κυρίως σώμα του αλγορίθμου περιλαμβάνει ένα διπλά εμφωλευμένο βρόχο. Σε κάθε επανάληψη του εξωτερικού βρόχου επιλέγεται η κορυφή με τη μικρότερη εκτίμηση απόστασης και η κατάστασή της αλλάζει μόνιμα σε σταθεροποιημένη. Στη συνέχεια, ο εσωτερικός βρόχος διατρέχει όλες τις εξερχόμενες πλευρές της και τις χαλαρώνει, με αποτέλεσμα κάποιες από τις γειτονικές της κορυφές που ήταν μη διατρεγμένες μέχρι εκείνη τη στιγμή να γίνουν εκκρεμείς. Ο αλγόριθμος παρουσιάζεται με περισσότερη λεπτομέρεια στο Σχήμα 5.3.

Η βασική δομή δεδομένων που βρίσκεται στην "καρδιά" του αλγορίθμου του Dijkstra είναι μια ουρά προτεραιότητας ελαχίστων τιμών. Η ουρά αποθηκεύει κορυφές από το γράφημα, συσχετίζοντάς τις με μια τιμή κλειδιού, που στην προκειμένη περίπτωση είναι η εκτίμηση απόστασης $d(\cdot)$ κάθε κορυφής. Η ουρά διατηρεί μόνο τις μη-σταθεροποιημένες κορυφές του γραφήματος. Σε κάθε επανάληψη του αλγορίθμου η κορυφή με το μικρότερο κλειδί αφαιρείται από την ουρά (λειτουργία `ExtractMin`) και οι εξερχόμενες πλευρές της χαλαρώνονται, πράγμα που ενδέχεται να μειώσει τα κλειδιά των αντίστοιχων γειτόνων της (λειτουργία `DecreaseKey`).

Ένα καλό ισοζύγιο που επιμερίζει το κόστος των πολλαπλών λειτουργιών `ExtractMin` και `DecreaseKey` που γίνονται κατά την εκτέλεση του αλγορίθμου, ειδικά για ρεαλιστικά, αραιά γραφήματα, είναι η υλοποίηση της ουράς προτεραιότητας σαν ένα δυαδικό σωρό. Ο δυαδικός σωρός είναι ένα σχεδόν πλήρες δυαδικό δέντρο που υλοποιείται σαν ένας γραμμικός πίνακας. Ένα παράδειγμα φαίνεται στο Σχήμα 5.5α. Κάθε στοιχείο του πίνακα αντιστοιχεί σε έναν κόμβο του σωρού. Αντί να χρησιμοποιούμε δείκτες προκειμένου να κινούμαστε ανάμεσα στους κόμβους, μπορούμε να χρησιμοποιήσουμε απλές δυαδικές και αριθμητικές πράξεις για να υπολογίσουμε τις θέσεις στον πίνακα του πατέρα και των παιδιών ενός κόμβου με βάση τη θέση του ίδιου του κόμβου. Οι τιμές των κόμβων ικανοποιούν την *ιδιότητα δυαδικού σωρού ελαχίστων τιμών* (min-heap property), που ορίζει ότι η τιμή ενός κόμβου είναι τουλάχιστον ίση ή μεγαλύτερη από την τιμή του πατέρα του. Έτσι, το μικρότερο στοιχείο στο σωρό βρίσκεται αποθηκευμένο στη ρίζα και το υποδέντρο που έχει σαν ρίζα κάποιον κόμβο περιέχει τιμές όχι μικρότερες από την τιμή του κόμβου αυτού. Κατά τη διάρκεια μιας λειτουργίας `DecreaseKey` μία κορυφή αποκτά νέα, μικρότερη τιμή για την εκτίμηση απόστασης από την αρχική κορυφή. Αν αυτή η νέα τιμή είναι μικρότερη από αυτή του πατέρα της στον σωρό, η κορυφή θα πρέπει να διασχίσει προς τα πάνω το δέντρο μέχρι να τοποθετηθεί σε κάποια θέση που ικανοποιεί την ιδιότητα του σωρού ελαχίστων τιμών. Κατά τη διάρκεια αυτής της διάσχισης η τιμή του κόμβου συγκρίνεται συνεχώς με αυτή του πατέρα του, και αν είναι μικρότερη, οι κόμβοι εναλλάσσονται.

Αλγόριθμος 1: Ο αλγόριθμος του Dijkstra.

Input : Κατευθυνόμενο γράφημα $G = (V, E)$, συνάρτηση βάρους $w : E \to \mathbf{R}^+$, αρχική κορυφή s, ουρά προτεραιότητας ελαχίστων τιμών Q
Output : πίνακας αποστάσεων d, πίνακας προγόνων π

```
   /* Φάση αρχικοποίησης */
 1 foreach v ∈ V do
 2     d[v] ← INF;
 3     π[v] ← NIL;
 4     Insert(Q, v);
 5 end
 6 d[s] ← 0;
   /* Κυρίως σώμα του αλγορίθμου */
 7 while Q ≠ ∅ do
 8     u ← ExtractMin(Q);
 9     foreach v adjacent to u do
10         sum ← d[u] + w(u, v);
11         if d[v] > sum then
12             DecreaseKey(Q, v, sum);
13             d[v] ← sum;
14             π[v] ← u;
15     end
16 end
```

Σχήμα 5.3: *Σειριακή υλοποίηση του αλγορίθμου του Dijkstra.*

5.4.2 Παράλληλη υλοποίηση βασισμένη σε κλειδώματα

Ο εξωτερικός βρόχος του αλγορίθμου του Dijkstra λειτουργεί με βάση την προτεραιότητα που υπάρχει ανάμεσα στις μη-σταθεροποιημένες κορυφές του γραφήματος σε μια δεδομένη στιγμή, γεγονός που σειριοποιεί την εκτέλεση του αλγορίθμου. Μια προφανής επιλογή για την παραλληλοποίηση του αλγορίθμου είναι να εκμεταλλευτούμε τον παραλληλισμό στον εσωτερικό βρόχο χαλαρώνοντας παράλληλα όλες τις εξερχόμενες πλευρές της κορυφής u. Αυτό είναι δυνατόν να γίνει, καθώς δεν παίζει ρόλο η σειρά με την οποίαν χαλαρώνονται οι πλευρές. Σε κάθε βήμα, ένα νήμα εξάγει την u από την κορυφή του σωρού και έπειτα οι εξερχόμενες πλευρές της ανατίθενται (π.χ. στατικά και κυκλικά) σε παράλληλα νήματα για χαλάρωση. Αυτό είναι ένα σχήμα λεπτομερούς παραλληλοποίησης. Μια γενική υλοποίησή του παρουσιάζεται στο Σχήμα 5.4, ενώ το μοτίβο εκτέλεσής του φαίνεται στο Σχήμα 5.6β.

Αναφορικά με αυτό το σχήμα παραλληλοποίησης μπορούμε να κάνουμε κάποιες παρατηρήσεις. Πρώτον, η επιτάχυνση που μπορεί να περιμένει κανείς περιορίζεται από το μέσο βαθμό εξόδου των κορυφών, δηλαδή από την πυκνότητα του γραφήματος. Και αυτό διότι αν οι κορυφές έχουν κατά μέσο όρο μικρό αριθμό γειτόνων, τότε το παράλληλο τμήμα του αλγορίθμου (γραμμές 6–14) θα καταναλώσει ένα μικρό μέρος του συνολικού χρόνου εκτέλεσης με αποτέλεσμα

Αλγόριθμος 2: Λεπτομερής παράλληλη υλοποίηση του αλγορίθμου του Dijkstra.

Input : Κατευθυνόμενο γράφημα $G = (V, E)$, συνάρτηση βάρους $w : E \to \mathbf{R}^+$, αρχική κορυφή s, ουρά προτεραιότητας ελαχίστων τιμών Q
Output : πίνακας αποστάσεων d, πίνακας προγόνων π

/ Φάση αρχικοποίησης όπως και στον σειριακό κώδικα */*

/ Κυρίως σώμα του αλγορίθμου */*

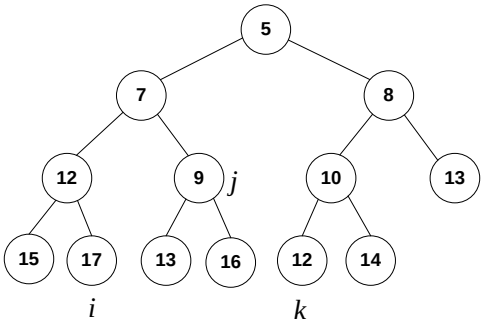
```
1  while Q ≠ ∅ do
2      Barrier
3      if tid = 0 then
4          u ← ExtractMin(Q);
5      Barrier
6      foreach v adjacent to u do in parallel
7          sum ← d[u] + w(u, v);
8          if d[v] > sum then
9              Begin-Atomic
10             DecreaseKey(Q, v, sum);
11             End-Atomic
12             d[v] ← sum;
13             π[v] ← u;
14     end
15 end
```

Σχήμα 5.4: *Λεπτομερής παράλληλη υλοποίηση του αλγορίθμου του Dijkstra.*

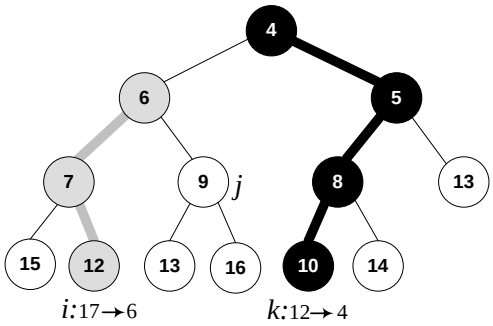
να επικρατεί το σειριακό τμήμα (γραμμές 3–4, `ExtractMin`). Τα Σχήματα 5.6α και 5.6β παρουσιάζουν τα μοτίβα εκτέλεσης για τη σειριακή και τη λεπτομερή πολυνηματική έκδοση, αντίστοιχα. Σε αυτά τα σχήματα δείχνουμε σκόπιμα ότι, σε αντίθεση με το σχετικά αμετάβλητο σειριακό τμήμα, ο χρόνος εκτέλεσης του παράλληλου τμήματος μπορεί να διαφέρει αρκετά ανάμεσα σε διαφορετικές επαναλήψεις, ανάλογα με τον αριθμό των γειτόνων της εξαχθείσας κορυφής.

Η δεύτερη παρατήρηση αφορά τις παράλληλες χαλαρώσεις πλευρών και τις ταυτόχρονες προσπελάσεις στο δυαδικό σωρό σαν αποτέλεσμα των λειτουργιών `DecreaseKey`. Στο Σχήμα 5.5 φαίνονται δύο παραδείγματα σχετικά με τον τρόπο με τον οποίον οι παράλληλες χαλαρώσεις μπορούν να οδηγήσουν ή όχι σε συγκρουόμενες λειτουργίες `DecreaseKey`. Στο Σχήμα 5.5β, οι ανανεώσεις των αποστάσεων των κορυφών i και k οδηγούν σε δύο αλληλουχίες εναλλασσόμενων κόμβων οι οποίες δεν επικαλύπτονται, και έτσι δε δημιουργούν πρόβλημα στο σωρό. Στην περίπτωση του Σχήματος 5.5γ, αντίθετα, οι δύο αλληλουχίες επεξεργάζονται κοινά τμήματα του σωρού, με αποτέλεσμα να είναι πιθανό να αφήσουν τα περιεχόμενά του σε ασυνεπή κατάσταση [2]. Κατά συνέπεια, για να διατηρήσουμε τη σημασιολογία του αλγορίθμου είναι απαραίτητο να συγχρονίσουμε τις προσβάσεις των νημάτων στο σωρό με τον κατάλληλο τρόπο.

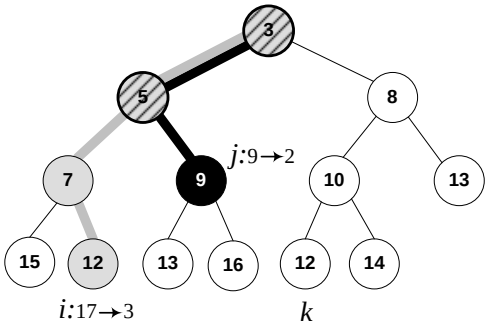
[2] Το ότι οι αλληλουχίες των δύο παράλληλων νημάτων επικαλύπτονται *χωρικά*, δε σημαίνει αναγκαστικά ότι ο σωρός θα οδηγηθεί σε ασυνεπή κατάσταση. Για να ισχύει αυτό θα πρέπει να υπάρχει και *χρονική* επικάλυψη, δηλαδή θα πρέπει τα νήματα να επιχειρήσουν να επεξεργαστούν κάποιον κοινό κόμβο την ίδια χρονική στιγμή.



(α) *Ουρά προτεραιότητας ελαχίστων τιμών υλοποιημένη σαν δυαδικός σωρός.*



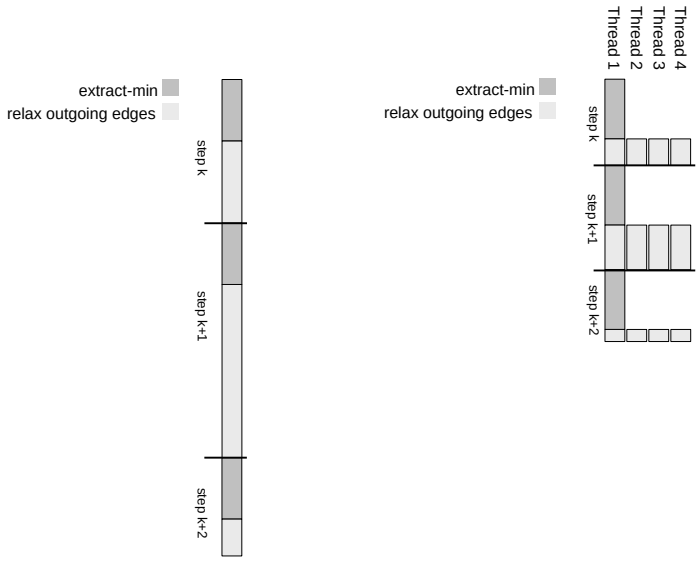
(β) *Μη-συγκρουόμενες λειτουργίες* `DecreaseKey`.



(γ) *Συγκρουόμενες λειτουργίες* `DecreaseKey`.

Σχήμα 5.5: *Ουρά προτεραιότητας ελαχίστων τιμών και ταυτόχρονες λειτουργίες* `DecreaseKey`.

Η πρώτη και μάλλον απλοϊκή προσέγγιση για την παραλληλοποίηση της φάσης της χαλάρωσης είναι να χρησιμοποιήσουμε μια μοναδική, καθολική μεταβλητή κλειδιού για να κλειδώσουμε ολόκληρο το δυαδικό σωρό κατά τη διάρκεια κάθε πράξης `DecreaseKey`. Αυτό συνιστά ένα συντηρητικό σχήμα αδρομερούς συγχρονισμού που επιτρέπει μόνο μία πράξη `DecreaseKey` ανά πάσα στιγμή και προφανώς περιορίζει τον παραλληλισμό. Στο εξής θα αναφερόμαστε στο σχήμα αυτό ως *cgs-lock*. Η εναλλακτική, πιο αισιόδοξη προσέγγιση είναι να επιτρέψουμε πολλαπλές αλληλουχίες εναλλαγών κόμβων να εκτελούνται ταυτόχρονα, με την προϋπόθεση όμως ότι κάθε αλληλουχία προσπελάζει διαφορετικό τμήμα του σωρού. Πιο συγκεκριμένα, αντί για ένα καθολικό κλειδί για ολόκληρο το σωρό μπορούμε να χρησιμοποιήσουμε ξεχωριστό κλειδί για κάθε ζευγάρι κόμβων πατέρα-παιδιού. Αυτό συνιστά ένα σχήμα λεπτομερούς συγχρονισμού. Κάθε φορά που κάποιο νήμα χρειάζεται να πραγματοποιήσει μια εναλλαγή κόμβων στα πλαίσια της λειτουργίας `DecreaseKey`, θα πρέπει πρώτα να αποκτήσει το κατάλληλο κλειδί που προστατεύει το συγκεκριμένο ζεύγος κόμβων (ένα σχήμα παρόμοιο με αυτό που παρουσιάζεται

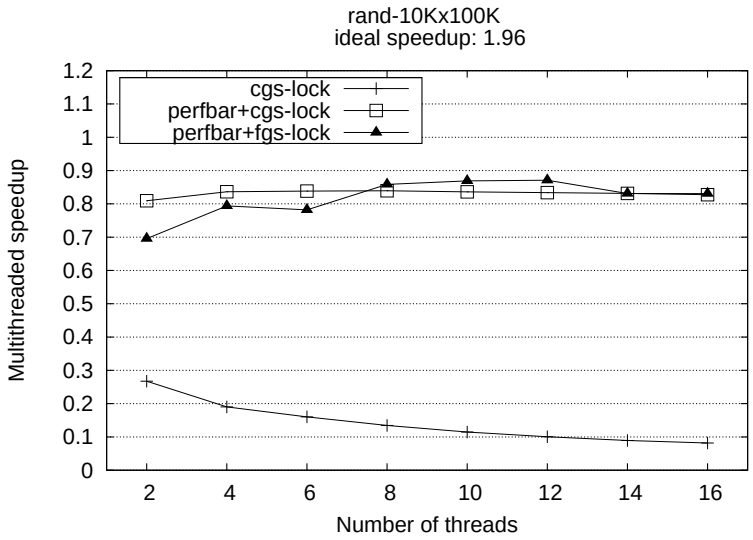


(α) *Σειριακή εκτέλεση.* **(β)** *Λεπτομερής πολυνηματική εκτέλεση.*

Σχήμα 5.6: *Μοτίβα εκτέλεσης για τη σειριακή και τη λεπτομερή πολυνηματική έκδοση του αλγορίθμου Dijkstra.*

στο [Hunt 96]). Με αυτόν τον τρόπο εγγυόμαστε την ατομικότητα στις τροποποιήσεις του σωρού από διαφορετικά νήματα, και ο αλγόριθμος μπορεί να εκτελεστεί παράλληλα με ασφάλεια. Θα αναφερόμαστε στο σχήμα αυτό ως *fgs-lock*.

Για να αποκτήσουμε μια πρώτη εικόνα για την αποδοτικότητα αυτών των δύο σχημάτων προσομοιώσαμε την εκτέλεσή τους σε ένα γράφημα 10K κορυφών και 100K πλευρών, οι οποίες προστέθηκαν ανάμεσα σε τυχαία επιλεγμένα ζεύγη κορυφών. Λεπτομέρειες αναφορικά με το περιβάλλον προσομοίωσης δίνουμε στην ενότητα 5.5.1. Το Σχήμα 5.7 παρουσιάζει την επιτάχυνση των δύο σχημάτων για αριθμό νημάτων από 2 έως 16. Η επιτάχυνση υπολογίζεται σαν το λόγο του χρόνου εκτέλεσης (σε κύκλους μηχανής) του σειριακού αλγορίθμου σε σχέση με το χρόνο εκτέλεσης του εκάστοτε πολυνηματικού σχήματος. Όπως βλέπουμε, η απόδοση του σχήματος *cgs-lock* είναι αρκετά απογοητευτική. Παρόλο που ο περιορισμένος παραλληλισμός του σχήματος αυτού εξηγεί κάπως την απουσία επιτάχυνσης, μια πιο λεπτομερής σκιαγράφηση του προφίλ εκτέλεσης του σχήματος μάς αποκάλυψε ότι η μεγαλύτερη υποβάθμιση της απόδοσης οφείλεται στην επιβάρυνση που εισάγουν τα φράγματα συγχρονισμού που περιβάλλουν τη λειτουργία `ExtractMin`. Στην ουσία, τα φράγματα αυτά απομονώνουν τη σειριακή από την παράλληλη φάση του αλγορίθμου. Το δεύτερο φράγμα εξασφαλίζει ότι τα παράλληλα νήματα μπορούν να αρχίσουν να χαλαρώνουν τις πλευρές μιας κορυφής αφού πρώτα ολοκληρωθεί η



Σχήμα 5.7: *Επιτάχυνση των παράλληλων υλοποιήσεων βασισμένων σε κλειδώματα, με πραγματικά και ιδανικά φράγματα συγχρονισμού.*

εξαγωγή της από το σωρό. Το πρώτο φράγμα εξασφαλίζει ότι δε θα εξαχθεί η επόμενη κορυφή αν πρώτα δεν ολοκληρωθούν οι χαλαρώσεις των πλευρών της προηγούμενης. Για δύο νήματα, ο χρόνος που ξοδεύεται στα φράγματα αντιστοιχεί στο 71% του συνολικού χρόνου εκτέλεσης. Αυτό το ποσοστό ανεβαίνει στο 88% για 8 νήματα, πράγμα που εξηγεί γιατί η απόδοση υποβαθμίζεται όταν χρησιμοποιούνται περισσότερα νήματα. Χρησιμοποιήσαμε τα φράγματα που παρέχονται από τη βιβλιοθήκη Pthreads, ωστόσο θεωρούμε ότι αυτή η συμπεριφορά δεν είναι θέμα της συγκεκριμένης υλοποίησης φραγμάτων αλλά ακόμα και εναλλακτικές υλοποιήσεις λογισμικού αναμένεται να δώσουν παρόμοια αποτελέσματα. Με βάση την εμπειρία μας, φράγματα συγχρονισμού που βασίζονται σε βρόχους περιδίνησης αποδίδουν καλύτερα για μικρό αριθμό νημάτων αλλά δεν κλιμακώνουν καλά.

Σε μια προσπάθεια να απομονώσουμε την επίδραση των φραγμάτων συγχρονισμού, υλοποιήσαμε μια έκδοση εξιδανικευμένων φραγμάτων μηδενικής καθυστέρησης, των οποίων η εκτέλεση υποθέτουμε ότι βασίζεται εξολοκλήρου σε ειδικό υλικό στην προσομοιωμένη πλατφόρμα μας (λεπτομέρειες για την υλοποίηση δίνουμε στην ενότητα 5.5.1). Ονομάζουμε το σχήμα αδρομερούς συγχρονισμού που χρησιμοποιεί τα ιδανικά αυτά φράγματα *perfbar+cgs-lock*. Από το Σχήμα 5.7 είναι ξεκάθαρο ότι η αντικατάσταση των φραγμάτων λογισμικού με μια "τέλεια" υλοποίηση στο υλικό αντιμετωπίζει το πρόβλημα της μη κλιμακωσιμότητας του σχήματος *cgs-lock*. Ωστόσο, ακόμα και η νέα έκδοση αποδίδει χειρότερα σε σχέση με τη σειριακή εκτέλεση του αλγορίθμου, αποδεικνύοντας ότι ο αδρομερής συγχρονισμός είναι πολύ συντηρητικός και δεν μπορεί να εκθέσει αρκετό παραλληλισμό.

Τέλος, το σχήμα *fgs-lock* με χρήση ιδανικών φραγμάτων αδυνατεί να υπερισχύσει της σειριακής εκτέλεσης, παρόλο που φαίνεται να έχει πιο ανοδική τάση σε σχέση με το *cgs-lock*. Καθώς ο αριθμός των νημάτων αυξάνει, η απόδοσή του βελτιώνεται ελαφρά, δείχνοντας ότι όντως υπάρχει κάποιος παραλληλισμός. Ωστόσο, το σχήμα *fgs-lock* αποτυγχάνει να τον εκμεταλλευτεί αποδοτικά και για αυτό υπάρχουν δύο πιθανές αιτίες. Πρώτον, προκειμένου να επιτρέψουμε ταυτόχρονες προσβάσεις στο σωρό χρειάζεται να χρησιμοποιήσουμε ένα ζεύγος μεταβλητών-κλειδιών για κάθε ζεύγος εναλλασσόμενων κόμβων του σωρού, αυξάνοντας έτσι τη συνολική επιβάρυνση λόγω των πολλών κλειδωμάτων και περιορίζοντας τα όποια κέρδη από την αξιοποίηση του παραλληλισμού. Δεύτερον, το σχήμα *fgs-lock* επιτρέπει ταυτόχρονες προσβάσεις στο σωρό μόνο όταν τα νήματα δουλεύουν σε διαφορετικές περιοχές του. Η πιθανότητα τα νήματα να επεξεργάζονται τους ίδιους κόμβους του σωρού εξαρτάται από τη δομή του γραφήματος καθώς και από τη σειρά με την οποίαν εξετάζονται οι γείτονες μιας κορυφής στα πλαίσια των λειτουργιών `DecreaseKey`. Όποτε αυτό συμβαίνει η εκτέλεση των νημάτων σειριοποιείται, περιορίζοντας τον συνολικό διαθέσιμο παραλληλισμό.

5.4.3 Παράλληλη υλοποίηση βασισμένη σε μνήμη διενεργειών

Το σχήμα *fgs-lock* που περιγράψαμε στην προηγούμενη ενότητα επιτρέπει ταυτόχρονες προσβάσεις στο δυαδικό σωρό, όμως εισάγει υψηλή επιβάρυνση εξαιτίας των πολλών κλειδωμάτων που απαιτούνται περιορίζοντας επομένως σημαντικά την αποδοτικότητά του. Αναζητώντας εναλλακτικές επιλογές, δοκιμάσαμε την αποτελεσματικότητα της μνήμης διενεργειών ως έναν "αισιόδοξο" μηχανισμό ελέγχου των ταυτόχρονων προσβάσεων στο σωρό. Η πρώτη προσέγγιση ήταν να εσωκλείσουμε κάθε λειτουργία `DecreaseKey` μέσα σε μία διενέργεια και να βασιστούμε στο υποκείμενο σύστημα για να εγγυηθούμε την ατομικότητα. Όταν ταυτόχρονες διενέργειες προσπελάζουν τους ίδιους κόμβους στο σωρό και τουλάχιστον μία από τις προσπελάσεις είναι εγγραφή, τότε προκύπτει διένεξη και το σύστημα αναλαμβάνει να την επιλύσει, δίνοντας δικαίωμα σε μία από τις διενέργειες να ολοκληρωθούν. Η λειτουργία `DecreaseKey` περιλαμβάνει μια αλληλουχία από εναλλαγές κόμβων, στα πλαίσια της μετακίνησης ενός κόμβου σε κάποιο υψηλότερο επίπεδο στο σωρό. Όταν δύο ή περισσότερες πλευρές χαλαρώνονται παράλληλα, τα αντίστοιχα ίχνη των μετακινούμενων κόμβων στο σωρό μπορεί να μοιράζονται έναν ή περισσότερους κοινούς κόμβους. Το σύστημα μνήμης διενεργειών θα εντοπίσει αυτή τη διένεξη, θα επιτρέψει σε μία μόνο διενέργεια να ολοκληρωθεί και, κατά συνέπεια, σε μία πλευρά να χαλαρωθεί. Τα υπόλοιπα νήματα θα ανασταλούν προσωρινά ή θα αναγκαστούν να ματαιώσουν και να επαναλάβουν τους υπολογισμούς τους, ανάλογα με την συγκεκριμένη υλοποίηση του συστήματος μνήμης διενεργειών. Αυτό το σχήμα συγχρονισμού δεν είναι τόσο λεπτομερές όσο το *fgs-lock*, στο οποίο η ατομικότητα επιβάλλεται σε επίπεδο μιας απλής εναλλαγής κόμβων και όχι για μια αλληλουχία εναλλαγών. Θα αναφερόμαστε στο σχήμα αυτό ως *cgs-tm*. Υλοποιείται όπως

Αλγόριθμος 3: Λειτουργία DecreaseKey

Input : Ουρά προτεραιότητας ελαχίστων τιμών Q, κορυφή u, νέα τιμή κλειδιού $value$ για την u

```
1 Q[u] ← value;
2 i ← u;
3 while (parent(i).key ≥ value) do
4     Begin-Transaction
5     if (parent(i).key ≥ value) then
6         swap(u, i);
7         i ← parent(i);
8     End-Transaction
9 end
```

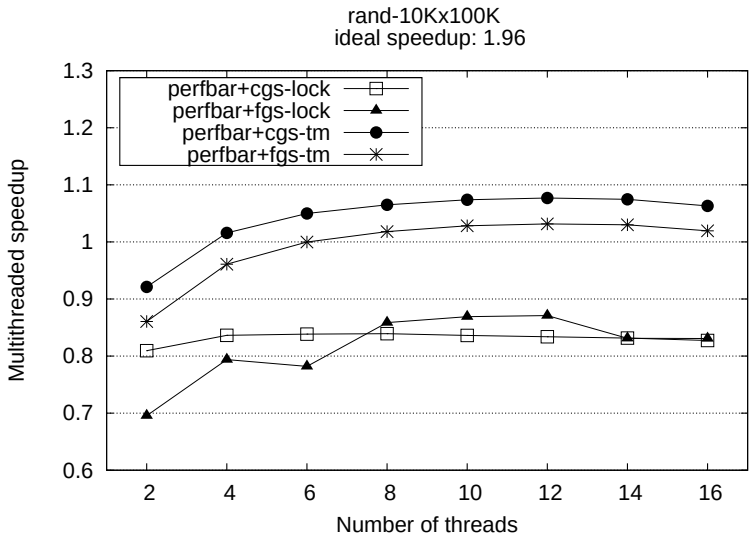
Σχήμα 5.8: *Υλοποίηση λειτουργίας* `DecreaseKey` *για το σχήμα fgs-tm.*

φαίνεται στον Αλγόριθμο 2 αντικαθιστώντας τις λειτουργίες `Begin-Atomic` και `End-Atomic` με κατάλληλες πρωτογενείς κλήσεις `Begin-Transaction` και `End-Transaction`.

Μια δεύτερη εναλλακτική είναι να υλοποιήσουμε ένα λεπτομερές σχήμα συγχρονισμού όπως το *fgs-lock* χρησιμοποιώντας μνήμη διενεργειών. Για να το επιτύχουμε αυτό, χρειάζεται να εσωκλείσουμε κάθε ξεχωριστή εναλλαγή κόμβων μέσα σε μία διενέργεια. Αυτό έμμεσα σημαίνει ότι οι διενέργειες σε αυτήν την περίπτωση θα είναι μικρότερες σε έκταση από τις αντίστοιχες του σχήματος *cgs-tm*, οδηγώντας σε λιγότερες διενέξεις και έτσι σε περισσότερο παραλληλισμό. Θα αναφερόμαστε στο σχήμα αυτό σαν *fgs-tm*. Για την υλοποίηση του σχήματος αυτού χρησιμοποιούμε την υλοποίηση της `DecreaseKey` που φαίνεται στον Αλγόριθμο 3 [3]. Όπως στα βασισμένα σε κλειδώματα σχήματα, έτσι κι εδώ χρειάζεται να χρησιμοποιήσουμε φράγματα συγχρονισμού για να διαχωρίσουμε τη σειριακή από την παράλληλη φάση. Έχοντας παρατηρήσει την αρνητική επίδραση των συμβατικών φραγμάτων στην απόδοση των προηγούμενων πολυνηματικών σχημάτων, χρησιμοποιούμε πάλι τα προσομοιωμένα, ιδανικά φράγματα για την αξιολόγηση των υλοποιήσεων που βασίζονται σε μνήμη διενεργειών.

Για να αποκτήσουμε μια πρώτη εικόνα όσον αφορά τη συμπεριφορά των σχημάτων που κάνουν χρήση μνήμης διενεργειών χρησιμοποιήσαμε το ίδιο γράφημα με αυτό της ενότητας 5.4.2 και παρουσιάζουμε την επιτάχυνση που παίρνουμε στο Σχήμα 5.9. Σε αντίθεση με τα βασισμένα σε κλειδώματα σχήματα, τα παρόντα σχήματα εκτελούνται ταχύτερα σε σχέση με το σειριακό αλγόριθμο για περισσότερα από 4 νήματα. Για 2 νήματα, το κόστος των σχημάτων με μνήμη διενεργειών φαίνεται να είναι αρκετά υψηλό και να ακυρώνει τα όποια κέρδη από την αξιοποίηση

[3] Όπως φαίνεται από τον Αλγόριθμο 3, εκτός από τη λειτουργία της εναλλαγής (γραμμή 6), περιλαμβάνουμε μέσα στη διενέργεια τον έλεγχο για το αν χρειάζεται να γίνει η εναλλαγή (γραμμή 5) καθώς και την ανανέωση του δείκτη προς τον τρέχοντα κόμβο (γραμμή 7). Με αυτόν τον τρόπο, όλες αυτές οι ενέργειες πραγματοποιούνται ενιαία ή δεν πραγματοποιούνται καθόλου. Αντίστοιχη ομαδοποίηση κάνουμε και στην περίπτωση του σχήματος *fgs-lock*.



Σχήμα 5.9: *Επιτάχυνση των παράλληλων υλοποιήσεων βασισμένων σε κλειδώματα και μνήμη διενεργειών.*

του παραλληλισμού. Καθώς εισάγονται περισσότερα νήματα, ωστόσο, η απόδοση βελτιώνεται δίνοντας τελικά επιτάχυνση σχεδόν μέχρι 1.1. Αναλυτικότερα αποτελέσματα παρουσιάζουμε σε επόμενη ενότητα. Ακόμα και από αυτήν την περίπτωση ωστόσο, φαίνεται ότι η μνήμη διενεργειών είναι ένας υποσχόμενος μηχανισμός για να εκμεταλλευτούμε τον παραλληλισμό των λειτουργιών στο σωρό. Υπενθυμίζουμε όμως ότι για να επιτύχουμε αυτές τις βελτιώσεις χρειάστηκε να χρησιμοποιήσουμε ιδανικά φράγματα συγχρονισμού.

5.4.4 Πολυνηματική έκδοση βασισμένη σε βοηθητικά νήματα και μνήμη διενεργειών

Σε αυτήν την ενότητα παρουσιάζουμε τη δική μας προσέγγιση στην παραλληλοποίηση του αλγορίθμου που βασίζεται στη συνδυασμένη χρήση βοηθητικών νημάτων και μνήμης διενεργειών. Με το σχήμα που προτείνουμε, προσπαθούμε να αντιμετωπίσουμε τα δύο βασικά προβλήματα της παραλληλοποίησης του αλγορίθμου, δηλαδή τον περιορισμένο προφανή παραλληλισμό και το συχνό συγχρονισμό.

Εξάγοντας περισσότερο παραλληλισμό

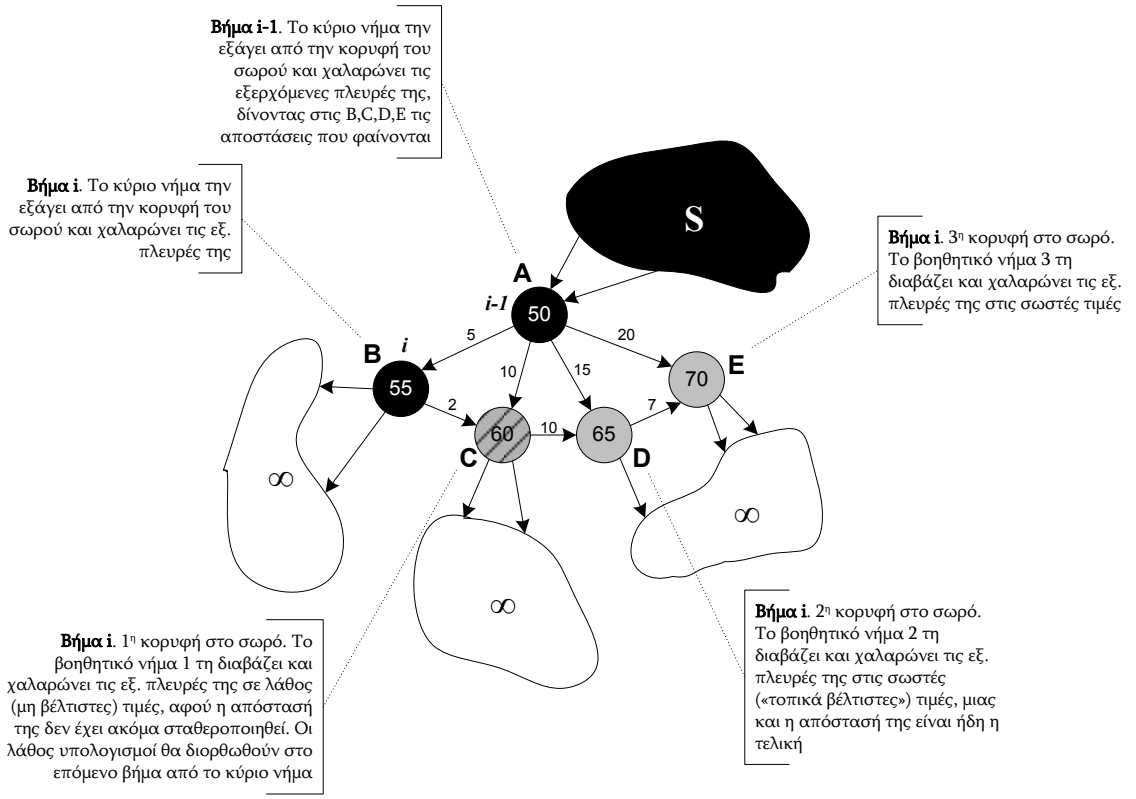
Όπως περιγράψαμε στην ενότητα 5.4.2 ο προφανής παραλληλισμός στον αλγόριθμο υφίσταται μόνο στον εσωτερικό βρόχο. Σκοπός μας είναι να κάνουμε τον παραλληλισμό πιο αδρομερή, όπως στις [Meyer 98, Edmonds 06, Madduri 06], χωρίς όμως να αλλάξουμε ριζικά τον αλγόριθμο. Συνεπώς, αντί να διαμερίζουμε τον εσωτερικό βρόχο και να αναθέτουμε λίγους μόνο

γείτονες σε κάθε νήμα για χαλάρωση, παραλληλοποιούμε τον εξωτερικό βρόχο και αναθέτουμε σε κάθε νήμα όλους τους γείτονες μιας κορυφής.

Ειδικότερα, εκμεταλλευόμαστε την εξής βασική ιδιότητα του αλγορίθμου: οι χαλαρώσεις των πλευρών οδηγούν σε μονότονα φθίνουσες τιμές για τις αποστάσεις των μη-σταθεροποιημένων κορυφών, μέχρι κάθε απόσταση να αποκτήσει την τελική, ελάχιστη τιμή της. Από τη στιγμή που μια κορυφή εισάγεται στο σύνολο εκκρεμών κορυφών (δηλαδή η απόστασή της από το S γίνει μη άπειρη) οι γείτονές της θα μπορούσαν επίσης να ανανεωθούν σε νέες τιμές μέσω χαλαρώσεων των πλευρών της. Ο σειριακός αλγόριθμος δεν εκμεταλλεύεται αυτήν την ιδιότητα, καθώς πραγματοποιεί όλες τις ανανεώσεις μόνο για τους γείτονες της εξαχθείσας κορυφής, αποφεύγοντας να υπολογίσει ενδιάμεσες αποστάσεις που τελικά θα επανεγγραφούν. Η βασική μας ιδέα εδώ είναι ότι θα μπορούσαν να χρησιμοποιηθούν πολλαπλά νήματα σαν *βοηθητικά νήματα* τα οποία θα χαλαρώνουν τις εξερχόμενες πλευρές κορυφών που είναι ακόμα εκκρεμείς. Ευελπιστούμε ότι με αυτόν τον τρόπο το *κύριο νήμα* θα απαλλαχθεί από το φορτίο που αντιστοιχεί σε κάποιες από αυτές τις χαλαρώσεις.

Το σκεπτικό πίσω από το προτεινόμενο σχήμα είναι ότι οι κορυφές που καταλαμβάνουν τις k υψηλότερες θέσεις στην ουρά προτεραιότητας μπορεί να είναι ήδη σταθεροποιημένες, με κάποια πιθανότητα. Έτσι, όταν τα βοηθητικά νήματα διαβάζουν τις αποστάσεις τους και χαλαρώνουν τις εξερχόμενες πλευρές τους, είναι αρκετά πιθανό οι χαλαρώσεις αυτές να είναι σωστές, δηλαδή να είναι αυτές που θα γίνονταν από τον σειριακό αλγόριθμο όταν κάθε τέτοια κορυφή εξαγόταν από την κορυφή του σωρού. Κατά συνέπεια, όταν το κύριο νήμα ελέγξει αργότερα αυτές τις κορυφές δε θα χρειάζεται να πραγματοποιήσει εκ νέου χαλαρώσεις. Αντίθετα, αν ένα βοηθητικό νήμα διαβάσει μια κορυφή που δεν έχει ακόμα σταθεροποιηθεί, θα ανανεώσει τους γείτονές της σε μη βέλτιστες αποστάσεις. Ωστόσο, όταν αργότερα αυτή η κορυφή φτάσει στην κεφαλή της ουράς και εξαχθεί από το κύριο νήμα, όλες οι εξερχόμενες πλευρές της θα χαλαρωθούν ξανά, με βάση τη σωστή, τελική της απόσταση.

Ένα παράδειγμα αυτής της ιδέας φαίνεται στο Σχήμα 5.10, όπου απεικονίζεται η i-στή επανάληψη του εξωτερικού βρόχου του αλγορίθμου. Στην προηγούμενη επανάληψη, το κύριο νήμα εξήγαγε την κορυφή A από το σωρό και χαλάρωσε όλες τις εξερχόμενες πλευρές της, δίνοντας στους γείτονές της τις αποστάσεις που φαίνονται. Στην τρέχουσα επανάληψη, το κύριο νήμα εξάγει την κορυφή B, ενώ τα βοηθητικά νήματα αναλαμβάνουν το καθένα τις τρεις επόμενες κορυφές στην ουρά προτεραιότητας, C, D και E. Οι πλευρές της C θα χαλαρωθούν χρησιμοποιώντας σαν απόσταση την τιμή 60. Όμως, στο τέλος αυτής της επανάληψης η απόσταση της C θα ανανεωθεί από το κύριο νήμα σε 57. Παρόλα αυτά, στην επανάληψη $i + 1$ το κύριο νήμα θα εξάγει την C και θα χαλαρώσει πάλι τις πλευρές της με βάση τη σωστή απόσταση. Σε αυτήν την περίπτωση, η δουλειά του βοηθητικού νήματος πήγε χαμένη. Αντίθετα, οι αποστάσεις των κορυφών D και Ε δε θα μεταβληθούν, καθώς έχουν ήδη αποκτήσει την ελάχιστή τους τιμή από την επανάληψη $i - 1$. Συνεπώς, στην επανάληψη i τα βοηθητικά νήματα χαλαρώνουν σωστά

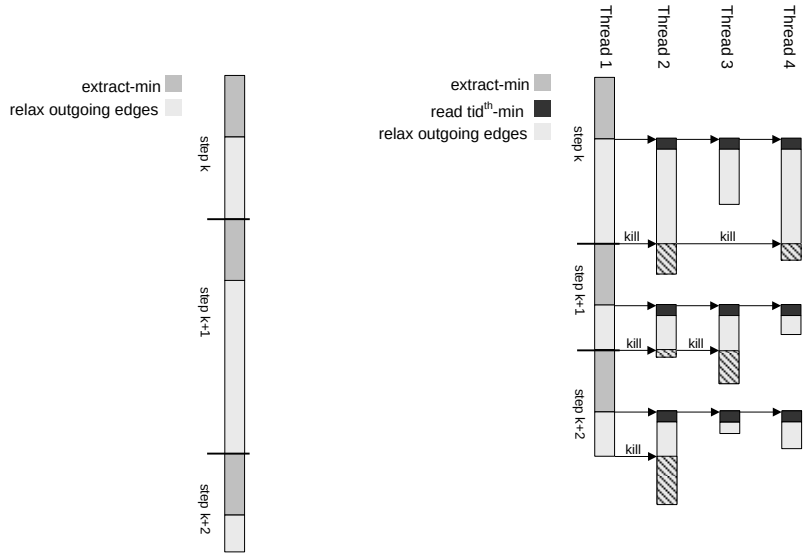


Σχήμα 5.10: *Παράδειγμα εκτέλεσης του προτεινόμενου σχήματος.*

τις εξερχόμενες πλευρές τους, με αποτέλεσμα όταν αργότερα το κύριο νήμα τις εξάγει να μην χρειάζεται να πραγματοποιήσει εκ νέου χαλαρώσεις.

Στην υλοποίησή μας, το κύριο νήμα λειτουργεί όπως στο σειριακό αλγόριθμο, *εξάγοντας* σε κάθε επανάληψη την κορυφή με την μικρότερη απόσταση από το σωρό και χαλαρώνοντας τις πλευρές της. Την ίδια στιγμή, το k-στό βοηθητικό νήμα *διαβάζει* την απόσταση της k-στής κορυφής στο σωρό (ας την αποκαλούμε x_k, στο εξής) και επιχειρεί να χαλαρώσει τις πλευρές της με βάση την απόσταση αυτή. Όταν το κύριο νήμα ολοκληρώσει όλες τις χαλαρώσεις για την τρέχουσα επανάληψη, ειδοποιεί τα βοηθητικά νήματα να σταματήσουν τους υπολογισμούς και να προχωρήσουν όλα τα νήματα μαζί στην επόμενη επανάληψη. Αυτό το μοτίβο εκτέλεσης παρουσιάζεται στο Σχήμα 5.11.

Ο τρόπος αυτός με τον οποίον το κύριο νήμα ελέγχει την εκτέλεση έχει μια πιθανή παρενέργεια. Είναι πιθανό σε αυτό το σημείο ένα βοηθητικό νήμα να μην έχει προλάβει να χαλαρώσει όλες τις εξερχόμενες πλευρές της x_k, αφήνοντας έτσι κάποιους από τους γείτονες της x_k με τις παλιές, πιθανώς μη βέλτιστες αποστάσεις τους. Όπως εξηγήσαμε παραπάνω όμως, αυτό δεν αποτελεί πρόβλημα αφού όλοι οι γείτονες της x_k με λανθασμένες αποστάσεις θα ανανεωθούν



(α) *Σειριακή εκτέλεση.* **(β)** *Πολυνηματική εκτέλεση βασισμένη σε βοηθητικά νήματα.*

Σχήμα 5.11: *Μοτίβο εκτέλεσης για το πολυνηματικό σχήμα βασισμένο σε βοηθητικά νήματα. Για σύγκριση παρατίθεται και το μοτίβο της σειριακή εκτέλεσης.*

στις σωστές τιμές όταν η x_k φτάσει στην κορυφή της ουράς αναμονής και την επεξεργαστεί το κύριο νήμα.

Αποδοτικός έλεγχος του ταυτοχρονισμού

Στο προτεινόμενο σχήμα τα νήματα χρειάζεται να προσπελάζουν ταυτόχρονα τόσο το δυαδικό σωρό όσο και τις υπόλοιπες δομές που είναι απαραίτητες για τον αλγόριθμο (γραμμές 10–14 στον Αλγόριθμο 1). Για αποδοτικό έλεγχο του ταυτοχρονισμού προτείνουμε τη χρήση μνήμης διενεργειών.

Όπως έχουμε αναφέρει, ένα σύστημα μνήμης διενεργειών επιτρέπει μη-συγκρουόμενες λειτουργίες σε δεδομένα, όπως αυτές που απεικονίζονται στο Σχήμα 5.5β, να συμβαίνουν παράλληλα χωρίς επιβάρυνση. Την ίδια στιγμή, εγγυάται την ατομικότητα, που σημαίνει ότι αν συμβεί κάποια διένεξη θα επιτρέψει σε ένα από τα νήματα να ολοκληρώσει την ενημέρωση του σωρού και των υπολοίπων δομών (π.χ. στο Σχήμα 5.5γ, να μετακινήσει τον κόμβο i μέχρι την κορυφή του σωρού) και θα επιβάλει στα υπόλοιπα να επαναλάβουν τη δουλειά τους (π.χ. στο Σχήμα 5.5γ, να υπολογίσει εκ νέου την απόσταση για τον j και να τον μετακινήσει στην κατάλληλη θέση στο σωρό). Για να υλοποιήσουμε αυτό το σενάριο, εσωκλείουμε κάθε λειτουργία `DecreaseKey` καθώς

και τις ενημερώσεις των πινάκων προγόνων και αποστάσεων μέσα σε μία διενέργεια, χρησιμοποιώντας τις κατάλληλες πρωτογενείς κλήσεις `Begin-Transaction` και `End-Transaction`, όπως φαίνεται στους Αλγορίθμους 4 και 5, για το κύριο και τα βοηθητικά νήματα, αντίστοιχα. Σημειώνουμε ότι με αυτόν τον τρόπο μία διένεξη είναι δυνατόν να προκύψει αν δύο ή περισσότερα νήματα ανανεώνουν ταυτόχρονα την ίδια γειτονική κορυφή ή αν ανανεώνουν διαφορετικές κορυφές αλλά τροποποιούν κοινές περιοχές στο σωρού.

Στην αρχή κάθε επανάληψης, το κύριο νήμα εξάγει την κορυφή που βρίσκεται στην κεφαλή της ουράς. Την ίδια στιγμή, τα βοηθητικά νήματα βρίσκονται σε ενεργό αναμονή μέχρι το κύριο νήμα να ολοκληρώσει την εξαγωγή, και έπειτα κάθε ένα διαβάζει –χωρίς να εξάγει– μία από τις k κορυφές στις υψηλότερες θέσεις της ουράς (αυτό υλοποιείται από τη συνάρτηση `ReadMin`). Στη συνέχεια, όλα τα νήματα χαλαρώνουν παράλληλα τις εξερχόμενες πλευρές των κορυφών που ανέλαβαν. Συγκριτικά με το σειριακό αλγόριθμο, αναμένεται μια βελτίωση στην απόδοση αφού, εξαιτίας της δράσης των βοηθητικών νημάτων, το κύριο νήμα θα αξιολογήσει τη συνθήκη της γραμμής 7 στον Αλγόριθμο 4 λιγότερες φορές σαν αληθή, με αποτέλεσμα να μη χρειαστεί να εκτελέσει τις λειτουργίες στις γραμμές 8–10.

Το σχήμα μας χρησιμοποιεί μνήμη διενεργειών όχι μόνο για τις ταυτόχρονες προσπελάσεις των διαφόρων δομών δεδομένων, αλλά και για την ενορχήστρωση και τον έλεγχο των βοηθητικών νημάτων. Ειδικότερα, όταν το κύριο νήμα ολοκληρώσει τις δικές του χαλαρώσεις, θέτει τη μεταβλητή ειδοποίησης `done` ίση με 1 εντός μιας ξεχωριστής διενέργειας. Αυτή η μεταβλητή σηματοδοτεί μια κατάσταση όπου το κύριο νήμα θέλει να προχωρήσει στην επόμενη επανάληψη και απαιτεί από τα βοηθητικά νήματα να σταματήσουν και να το ακολουθήσουν, τερματίζοντας όποιες λειτουργίες εκτελούσαν μέχρι εκείνη τη στιγμή. Όλα τα βοηθητικά νήματα που εκτελούσαν διενέργειες σε αυτή τη χρονική στιγμή θα τις ματαιώσουν, αφού η μεταβλητή `done` περιλαμβάνεται στο σύνολο αναγνώσεων και των δικών τους διενεργειών. Ακολούθως, θα επανεκτελέσουν τις διενέργειές τους, αλλά υπάρχει καλή πιθανότητα να βρουν τη μεταβλητή `done` στην τιμή 1, να σταματήσουν την εξέταση των υπολοίπων γειτόνων στον εσωτερικό βρόχο και να συνεχίσουν στην επόμενη επανάληψη του εξωτερικού βρόχου. Αν τύχει το κύριο νήμα να πραγματοποιήσει τη λειτουργία `ExtractMin` πολύ γρήγορα, η μεταβλητή `done` θα τεθεί πάλι ίση με 0 και τα βοηθητικά νήματα θα χάσουν την τελευταία ειδοποίηση, συνεχίζοντας από το σημείο όπου είχαν σταματήσει. Αυτό μπορεί να οδηγήσει σε μη βέλτιστες ανανεώσεις των αποστάσεων των γειτόνων, αλλά όπως εξηγήσαμε και πιο πάνω, οι αποστάσεις θα επανεγγραφούν στις σωστές τιμές όταν οι κορυφές που αναλαμβάνουν τα βοηθητικά νήματα φτάσουν κάποια στιγμή στην κεφαλή της ουράς. Συνεπώς, η ορθότητα του σχήματος είναι εγγυημένη.

Η χρήση της μνήμης διενεργειών έναντι παραδοσιακών μεθόδων συγχρονισμού, όπως κλειδώματα και φράγματα συγχρονισμού, προσφέρει δύο σημαντικά πλεονεκτήματα: Πρώτον, μάς διευκολύνει ιδιαίτερα στον προγραμματισμό του προτεινόμενου σχήματος η δυνατότητά της

Αλγόριθμος 4: Κώδικας κύριου νήματος.

Input : Κατευθυνόμενο γράφημα $G = (V, E)$, συνάρτηση βάρους $w : E \rightarrow \mathbf{R}^+$, αρχική κορυφή s, ουρά προτεραιότητας ελαχίστων τιμών Q
Output : πίνακας αποστάσεων d, πίνακας προγόνων π

/ Φάση αρχικοποίησης όπως και στον σειριακό κώδικα */*

```
1  while Q ≠ ∅ do
2      u ← ExtractMin(Q);
3      done ← 0;
4      foreach v adjacent to u do
5          sum ← d[u] + w(u, v);
6          Begin-Transaction
7          if d[v] > sum then
8              DecreaseKey(Q, v, sum);
9              d[v] ← sum;
10             π[v] ← u;
11         End-Transaction
12     end
13     Begin-Transaction
14     done ← 1;
15     End-Transaction
16 end
```

Αλγόριθμος 5: Κώδικας βοηθητικών νημάτων.

```
1  while Q ≠ ∅ do
2      while done = 1 do ;
3      x ← ReadMin(Q, tid);
4      stop ← 0;
5      foreach y adjacent to x and while stop = 0 do
6          Begin-Transaction
7          if done = 0 then
8              sum ← d[x] + w(x, y);
9              if d[y] > sum then
10                 DecreaseKey(Q, y, sum);
11                 d[y] ← sum;
12                 π[y] ← x;
13         else
14             stop ← 1;
15         End-Transaction
16     end
17 end
```

Σχήμα 5.12: *Πολυνηματική υλοποίηση του αλγορίθμου Dijkstra βασισμένη σε βοηθητικά νήματα.*

να συνθέτει επιμέρους λειτουργίες σε μία ενιαία ατομική λειτουργία. Συγκεκριμένα, τόσο στον κώδικα του κύριου νήματος όσο και σε αυτόν των βοηθητικών νημάτων υπάρχουν τρεις βασικές λειτουργίες που η εκτέλεση της επόμενης εξαρτάται από την έκβαση της προηγούμενης: ο

έλεγχος για το αν μια πλευρά χρειάζεται χαλάρωση (γραμμή 7 στον Αλγόριθμο 4), η ανανέωση της απόστασης του αντίστοιχου γείτονα στο σωρό (γραμμή 8) και η ανανέωση των πινάκων προγόνων και αποστάσεων (γραμμές 9–10). Εσωκλείοντας αυτές τις λειτουργίες εντός μίας διενέργειας, εξασφαλίζουμε ότι τόσο το κύριο όσο και τα βοηθητικά νήματα θα τις πραγματοποιήσουν σαν μια ενιαία λειτουργία, δηλαδή, είτε όλες μαζί είτε καμία από αυτές. Αντιθέτως, θα ήταν δύσκολο και επιρρεπές σε λάθη να αναπτύξουμε αντίστοιχο σχήμα με λεπτομερή κλειδώματα χωρίς να περιορίζουμε τον ταυτοχρονισμό ή χωρίς να καθυστερούμε την εκτέλεση του κύριου νήματος. Θα χρειαζόταν πιθανώς να χρησιμοποιήσουμε πολλαπλά κλειδώματα με τρόπο όμως που να αποτυπώνει τις εξαρτήσεις ανάμεσα στις τρεις αυτές λειτουργίες. Κάτι τέτοιο θα ήταν αρκετά περίπλοκο, αφού η ορθότητα απαιτεί την αποφυγή πιθανών αδιεξόδων (στάσιμων ή ενεργών), ενώ η αποδοτικότητα απαιτεί την αποφυγή της σειριοποίησης των προσπελάσεων στις δομές κατά το δυνατόν.

Δεύτερον, ακόμα και αν μπορούσε σχετικά απλά να υλοποιηθεί τέτοιο σχήμα βασισμένο σε κλειδώματα, θα επέσυρε υψηλή επιβάρυνση για τις μη-συγκρουόμενες παράλληλες προσπελάσεις. Αυτό θα ήταν αποδεκτό αν η πλειοψηφία των παράλληλων προσπελάσεων οδηγούσε σε διενέξεις. Όμως, στην περίπτωση του αλγορίθμου που εξετάζουμε ισχύει το αντίθετο. Κατά συνέπεια, ο αισιόδοξος τρόπος λειτουργίας της μνήμης διενεργειών, κατά τον οποίον οι μη-συγκρουόμενες προσπελάσεις γίνονται χωρίς επιπλέον κόστος, την καθιστά σαφώς καλύτερη επιλογή.

5.5 Πειραματική Αξιολόγηση

5.5.1 Περιβάλλον εκτέλεσης πειραμάτων

Αξιολογήσαμε την απόδοση των διαφόρων υλοποιήσεων του αλγορίθμου του Dijkstra μέσω προσομοίωσης πλήρους συστήματος, καθοδηγούμενη από εκτέλεση (full-system execution driven). Χρησιμοποιήσαμε το εργαλείο GEMS από το πανεπιστήμιο του Wisconsin [Martin 05, gem 08], έκδοσης 2.1, σε συνδυασμό με τον προσομοιωτή Simics [Magnusson 02], έκδοσης 3.0.31. Ο Simics παρέχει λειτουργική προσομοίωση ενός SPARC πολυπύρηνου συστήματος το οποίο φορτώνει και τρέχει ατροποποίητο λειτουργικό σύστημα Solaris 10. Η επέκταση Ruby του GEMS πραγματοποιεί λεπτομερή προσομοίωση του υποσυστήματος μνήμης, για όσες εντολές πραγματοποιούν λειτουργίες μνήμης. Για τις υπόλοιπες εντολές συμπεριφέρεται ως ένας απλός επεξεργαστής εντός σειράς και μονής έκδοσης εντολών (in-order, single-issue), που σημαίνει ότι ολοκληρώνει κάθε τέτοια εντολή σε έναν προσομοιούμενο κύκλο.

Ο GEMS υποστηρίζει μνήμη διενεργειών στο υλικό μέσω του υποσυστήματος LogTM-SE [Yen 07]. Το HTM σύστημα αυτό είναι σχεδιασμένο να λειτουργεί για πολυπύρηνα συστήματα ενός τσιπ με MESI πρωτόκολλο συνάφειας μνήμης (`MESI_CMP_filter_directory`), ιδιωτικές

ανά πυρήνα κρυφές μνήμες L1 και μία καθολική διαμοιραζόμενη κρυφή μνήμη L2. Εφαρμόζει *πρόθυμη πολιτική για τη διαχείριση εκδόσεων των δεδομένων* (eager version management), που σημαίνει ότι τα δεδομένα που τροποποιούνται εντός μιας διενέργειας εγγράφονται απευθείας στην κύρια μνήμη, αφού όμως πρώτα οι παλιές εκδόσεις τους καταγραφούν σε μια ειδική δομή λογισμικού (software log). Επίσης εφαρμόζει *πρόθυμη ανίχνευση διενέξεων* (eager conflict detection), που σημαίνει ότι οι διενέξεις, δηλαδή οι επικαλύψεις ανάμεσα στο σύνολο εγγραφών μιας διενέργειας και το σύνολο εγγραφών ή αναγνώσεων μιας άλλης, ανιχνεύονται ακριβώς τη στιγμή που συμβαίνουν. Για να επιταχύνει την ανίχνευση διενέξεων το σύστημα χρησιμοποιεί *υπογραφές υλικού* (hardware signatures) οι οποίες συνοψίζουν τα σύνολα εγγραφών και αναγνώσεων των διενεργειών. Όταν συγκρούονται δύο διενέργειες, η μία από αυτές αναστέλλεται και είτε επιχειρεί ξανά την αναφορά μνήμης που προκάλεσε τη διένεξη, με την ελπίδα ότι η άλλη διενέργεια έχει ολοκληρωθεί, είτε ματαιώνεται εντελώς αν το HTM σύστημα εντοπίσει πιθανό αδιέξοδο. Στη δεύτερη περίπτωση, το σύστημα αναιρεί τις τροποποιήσεις που έκανε η διενέργεια χρησιμοποιώντας τα αρχικά δεδομένα από την ειδική δομή καταγραφής και στη συνέχεια την επανεκτελεί.

Στα πειράματά μας, παραμετροποιήσαμε τον προσομοιωτή ώστε να χρησιμοποιεί ρεαλιστικές (υλοποιήσιμες) υπογραφές υλικού μεγέθους 2Kbits, μιας και αυτές αναφέρθηκαν στο [Yen 07] ότι συμπεριφέρονται παρόμοια με ιδανικές, μη ρεαλιστικές υπογραφές. Σε πειράματα που πραγματοποιήσαμε για την εφαρμογή που εξετάζουμε, πράγματι επιβεβαιώσαμε ότι οι προτεινόμενες ρεαλιστικές υπογραφές συμπεριφέρονται παρόμοια με τις ιδανικές. Επιπλέον, χρησιμοποιήσαμε την *πολιτική επίλυσης διενέξεων* (conflict resolution policy) `HYBRID`, η οποία έχει την τάση να δίνει προτεραιότητα ολοκλήρωσης σε παλαιότερες διενέργειες έναντι νεοτέρων, κάθε φορά που προκύπτει διένεξη. Επιλέξαμε αυτήν την πολιτική καθώς επιδεικνύει την καλύτερη συμπεριφορά για τις διάφορες εκδόσεις του αλγορίθμου ανάμεσα σε άλλες αντίστοιχες πολιτικές του LogTM. Η αξιολόγηση της επίδρασης διαφορετικών πολιτικών επίλυσης διενέξεων ή άλλων παραμέτρων του HTM συστήματος στην απόδοση των διαφόρων εκδόσεων, αν και αρκετά ενδιαφέρουσα, παραμένει εκτός του άμεσου αντικειμένου αυτής της διατριβής. Μία λεπτομερής παρουσίαση των παραμέτρων του περιβάλλοντος προσομοίωσης δίνεται στον Πίνακα 5.1.

Στα προγράμματά μας χρησιμοποιήσαμε τη βιβλιοθήκη Pthreads για τη δημιουργία των νημάτων και συμβατικές λειτουργίες συγχρονισμού όπως κλειδώματα βασισμένα σε βρόχους περιδίνησης (spin-locks) και φράγματα. Για την υλοποίηση των ιδανικών φραγμάτων μηδενικής καθυστέρησης, κωδικοποιήσαμε ένα καθολικό φράγμα σαν μια απλή εντολή μηχανής χρησιμοποιώντας τις "μαγικές εντολές" του Simics. Με αυτόν τον τρόπο ο συγχρονισμός των νημάτων ενορχηστρώνεται πλήρως από τον προσομοιωτή και όχι από το λειτουργικό σύστημα, με αποτέλεσμα η αναστολή των νημάτων όταν φτάνουν στο φράγμα και η επανεκκίνησή τους όταν

Simics	Επεξεργαστής	Διαμορφώσεις έως και 32 πυρήνες UltraSPARC III Cu (III+)
GEMS/Ruby	Κρυφές μνήμες L1	Ιδιωτικές, 64KB, 4-way set-associative, 64 bytes μέγεθος γραμμής, 4 κύκλοι χρόνος πρόσβασης
	Κρυφή μνήμη L2	Ενοποιημένη και μοιραζόμενη, 2MB, 8 τμήματα (banks), 4-way set-associative, 64 bytes μέγεθος γραμμής, 10 κύκλοι χρόνος πρόσβασης
	Κύρια μνήμη	160 κύκλοι χρόνος πρόσβασης
	HTM σύστημα	πολιτική επίλυσης διενέξεων HYBRID, υπογραφές υλικού μεγέθους 2Kbits

Πίνακας 5.1: *Περιβάλλον προσομοίωσης.*

φεύγουν από αυτό να γίνεται σε έναν κύκλο. Για να προσφέρουμε ένα όσο το δυνατόν πιο απομονωμένο περιβάλλον εκτέλεσης στα νήματα της εφαρμογής, αποφεύγοντας διενέξεις για κοινούς πόρους με διεργασίες του λειτουργικού συστήματος, χρησιμοποιήσαμε διαμορφώσεις για το σύστημα CMP με παραπάνω πυρήνες από τον εκάστοτε αριθμό νημάτων της εφαρμογής. Έτσι, για παράδειγμα, τα πειράματα για 2 και 4 νήματα πραγματοποιήθηκαν σε ένα σύστημα CMP 8 πυρήνων, για 8 νήματα σε ένα σύστημα 16 πυρήνων, κ.ο.κ. Για να επιβάλουμε σε κάθε νήμα να εκτελείται σε έναν συγκεκριμένο πυρήνα αποφεύγοντας τη μετακίνηση σε διαφορετικούς πυρήνες, χρησιμοποιήσαμε την κλήση συστήματος `pset_bind` του Solaris. Τέλος, όλοι οι κώδικες μεταγλωττίστηκαν με τον μεταγλωττιστή της C που παρέχει το Sun Studio 12 χρησιμοποιώντας το O3 επίπεδο βελτιστοποίησης.

5.5.2 Γράφοι αναφοράς

Για να αποκτήσουμε μια όσο το δυνατόν πιο αντιπροσωπευτική εικόνα για την αποτελεσματικότητα κάθε πολυνηματικού σχήματος, χρησιμοποιήσαμε γραφήματα που διαφέρουν όχι μόνο ως προς την πυκνότητα αλλά και ως προς τη δομή. Σε αυτήν την κατεύθυνση, χρησιμοποιήσαμε το εργαλείο GTgraph [Bader 06] με το οποίο κατασκευάσαμε γραφήματα από τις ακόλουθες οικογένειες:

Random: Τυχαία γραφήματα χωρίς καμία συγκεκριμένη δομή. Ακολουθούν το μοντέλο $G(n, m)$, σύμφωνα με το οποίο m πλευρές προστίθεται σε n κορυφές επιλέγοντας τυχαία ζεύγη κορυφών.

R-MAT: Γραφήματα όπου οι βαθμοί εισόδου των κορυφών ακολουθούν *κατανομή υπερνομοθετικής ισχύος* (power-law distribution), σύμφωνα με την οποία υπάρχουν λίγοι κόμβοι με μεγάλο βαθμό εισόδου και πολλοί με μικρό βαθμό εισόδου. Τα γραφήματα αυτά κατασκευάζονται χρησιμοποιώντας το μοντέλο αναδρομικού μητρείου (recursive matrix – R-MAT) [Chakrabarti 04].

SSCA#2: Γραφήματα που χρησιμοποιούνται στα πλαίσια του μετροπρογράμματος ανάλυσης γραφημάτων DARPA HPCS SSCA#2 [Bader 05]. Τα γραφήματα αυτά παράγονται δημιουργώντας αρχικά κλίκες με μέγεθος ομοιόμορφα κατανεμημένο ανάμεσα σε 1 και C, και στη συνέχεια προσθέτοντας πλευρές ανάμεσα σε διαφορετικές κλίκες με πιθανότητα P.

Όλα τα γραφήματα αποτελούνται από 10K κορυφές. Στις πλευρές έχουν ανατεθεί με τυχαίο και ομοιόμορφο τρόπο ακέραια βάρη μεταξύ 1 και 100. Για κάθε οικογένεια δημιουργήσαμε 6 γραφήματα, καθένα με διαφορετική πυκνότητα. Οι παράμετροι που χρησιμοποιήσαμε για την παραγωγή των γραφημάτων μαζί με τον αριθμό των πλευρών τους φαίνεται στον Πίνακα 5.2. Σε όλα τα γραφήματα εξαλείψαμε μετά την παραγωγή τους παράλληλες πλευρές ή πλευρές-βρόχους (loop-edges), με αρχή και τέλος την ίδια κορυφή.

Για να αποκτήσουμε μια εικόνα της επιτάχυνσης που μπορούμε να αναμένουμε από το προτεινόμενο σχήμα για κάθε ένα από τα γραφήματα, σκιαγραφήσαμε το προφίλ του κύριου νήματος όταν αυτό εκτελείται χωρίς την παρουσία βοηθητικών νημάτων και καταγράψαμε το ποσοστό που καταλαμβάνει το σειριακό μέρος στο συνολικό χρόνο εκτέλεσης. Σαν σειριακό μέρος ορίζουμε το τμήμα του κώδικα του κύριου νήματος που εκτελείται εκτός διενεργειών, το οποίο περιλαμβάνει κυρίως τις λειτουργίες `ExtractMin`. Στην ιδανική περίπτωση όπου τα βοηθητικά νήματα θα επιτύγχαναν να αποφορτίσουν το κύριο νήμα από όλες τις χαλαρώσεις πλευρών, η επιτάχυνση θα ήταν ίση με $\frac{100\%}{\%\text{σειριακού μέρους}}$. Σημειώνουμε ότι αυτή είναι μια αισιόδοξη εκτίμηση της επιτάχυνσης, καθώς ακόμα και σε αυτήν την περίπτωση το κύριο νήμα θα έπρεπε να ελέγξει αν απαιτούνται ή όχι χαλαρώσεις. Σε γενικές γραμμές, συνιστά ένα θεωρητικό άνω όριο για οποιαδήποτε βελτίωση της απόδοσης και παρουσιάζεται στον Πίνακα 5.2.

5.5.3 Πειραματικά αποτελέσματα

Στα Σχήματα 5.13, 5.14 και 5.15 παρουσιάζονται οι επιταχύνσεις που πέτυχαν οι διάφορες πολυνηματικές εκδόσεις για τις οικογένειες γραφημάτων Random, R-MAT και SSCA#2, αντίστοιχα. Ως επιτάχυνση ορίζουμε το λόγο του χρόνου εκτέλεσης (σε κύκλους μηχανής) του σειριακού αλγορίθμου σε σχέση με το χρόνο εκτέλεσης του εκάστοτε πολυνηματικού σχήματος. Για το προτεινόμενο σχήμα που βασίζεται σε βοηθητικά νήματα (*helper*), η επιτάχυνση που παρουσιάζουμε για p νήματα αφορά την εκτέλεση με ένα κύριο νήμα και $p-1$ βοηθητικά.

Οικογένεια γραφημάτων	Παράμετροι κατασκευής	Αριθμός πλευρών	Σειριακό μέρος (%)	Ιδανική επιτάχυνση
Random		10K	52.9	1.89
		50K	62.2	1.61
		100K	50.9	1.96
		200K	40.1	2.49
		500K	28.4	3.52
		1000K	22.6	4.42
R-MAT	$a = 0.45, b = 0.15$ $c = 0.15, d = 0.25$	10K	68.4	1.46
		50K	58.8	1.70
		100K	48.3	2.07
		200K	38.0	2.63
		500K	27.3	3.66
		1000K	22.2	4.50
SSCA#2	$(P, C) = (0.25, 5)$	28K	45.0	2.22
	$(P, C) = (0.5, 10)$	60K	55.2	1.81
	$(P, C) = (0.5, 20)$	118K	46.6	2.15
	$(P, C) = (0.5, 30)$	177K	41.5	2.41
	$(P, C) = (0.5, 100)$	590K	27.4	3.65
	$(P, C) = (0.5, 200)$	1157K	22.4	4.64

Πίνακας 5.2: *Παράμετροι παραγωγής γραφημάτων και χαρακτηριστικά τους ως προς το προφίλ εκτέλεσης του σειριακού αλγορίθμου Dijkstra.*

Όπως είναι εμφανές, το σχήμα *helper* υπερτερεί σε κάθε περίπτωση έναντι των υπολοίπων και σε σημαντικό βαθμό. Η μέγιστη επιτάχυνση που επιτυγχάνει είναι 1.84 για 14 νήματα, στην περίπτωση του Σχήματος 5.13ζ. Λαμβάνοντας υπόψη τη σειριακή φύση του αλγορίθμου και τις εγγενείς δυσκολίες στην παραλληλοποίησή του, πρόκειται για σημαντικό κέρδος στην απόδοση (τη στιγμή που η ιδανική επιτάχυνση είναι περίπου ίση με 4.42).

Σε συμφωνία με τα προκαταρκτικά αποτελέσματα που παρουσιάσαμε στις ενότητες 5.4.2 και 5.4.3, οι ταυτόχρονες προσβάσεις σε μοιραζόμενα δεδομένα υλοποιημένες με τη βοήθεια μνήμης διενεργειών (*perfbar-cgs-tm*, *perfbar-fgs-tm*) υπερισχύουν ξεκάθαρα των βασισμένων σε κλειδώματα υλοποιήσεων (*perfbar-cgs-lock*, *perfbar-fgs-lock*). Αυτό αποκαλύπτει την ύπαρξη λεπτομερούς παραλληλισμού κατά τις ανανεώσεις του σωρού, υπό την έννοια ότι, στατιστικά, τα

ίχνη στο σωρό από τις ταυτόχρονες ανανεώσεις είναι με μεγάλη πιθανότητα μη επικαλυπτόμενα. Ως εκ τούτου, ο αισιόδοξος παραλληλισμός φαίνεται καλή προσέγγιση για τον αλγόριθμο του Dijkstra. Ωστόσο, μόνο με την χρησιμοποίηση ιδανικών φραγμάτων συγχρονισμού μπορούν να προσφέρουν τα σχήματα λεπτομερούς παραλληλοποίησης κάποια βελτίωση σε σχέση με το σειριακό αλγόριθμο.

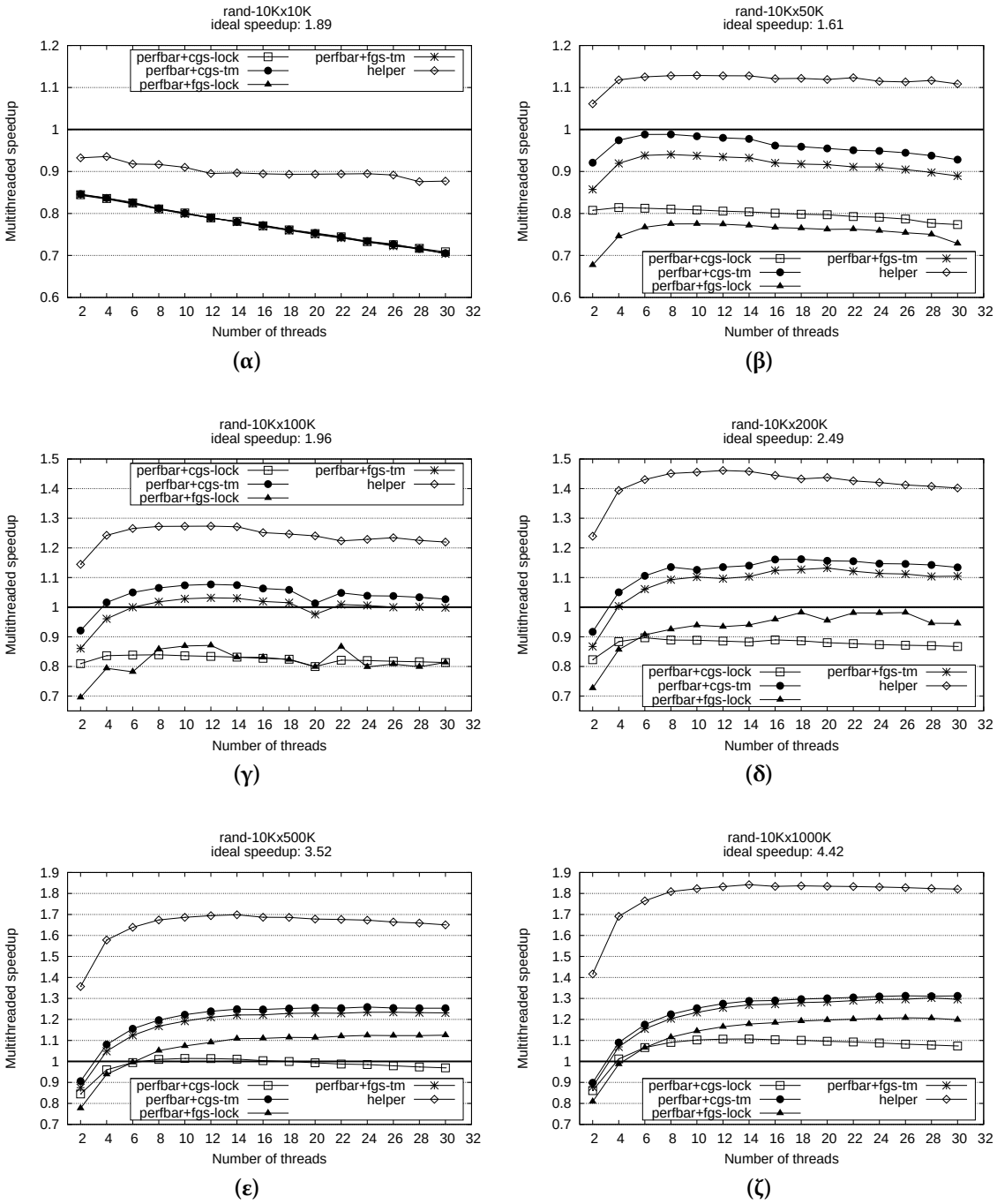
Το σχήμα *helper*, αντίθετα, όχι μόνο είναι αποδεσμευμένο από την αποδοτικότητα των εκάστοτε φραγμάτων, αλλά είναι σε θέση να εκμεταλλευτεί και πιο αδρομερή παραλληλισμό. Σημειώνουμε επίσης ότι η απόδοσή του συνδέεται σε μεγάλο βαθμό με την πυκνότητα του γραφήματος. Στη σειριακή περίπτωση ο χρόνος εκτέλεσης μπορεί να προσεγγιστεί ως εξής:

$$T_{serial} = n \times O(\lg n) + d \times n \times O(\lg n) \tag{5.1}$$

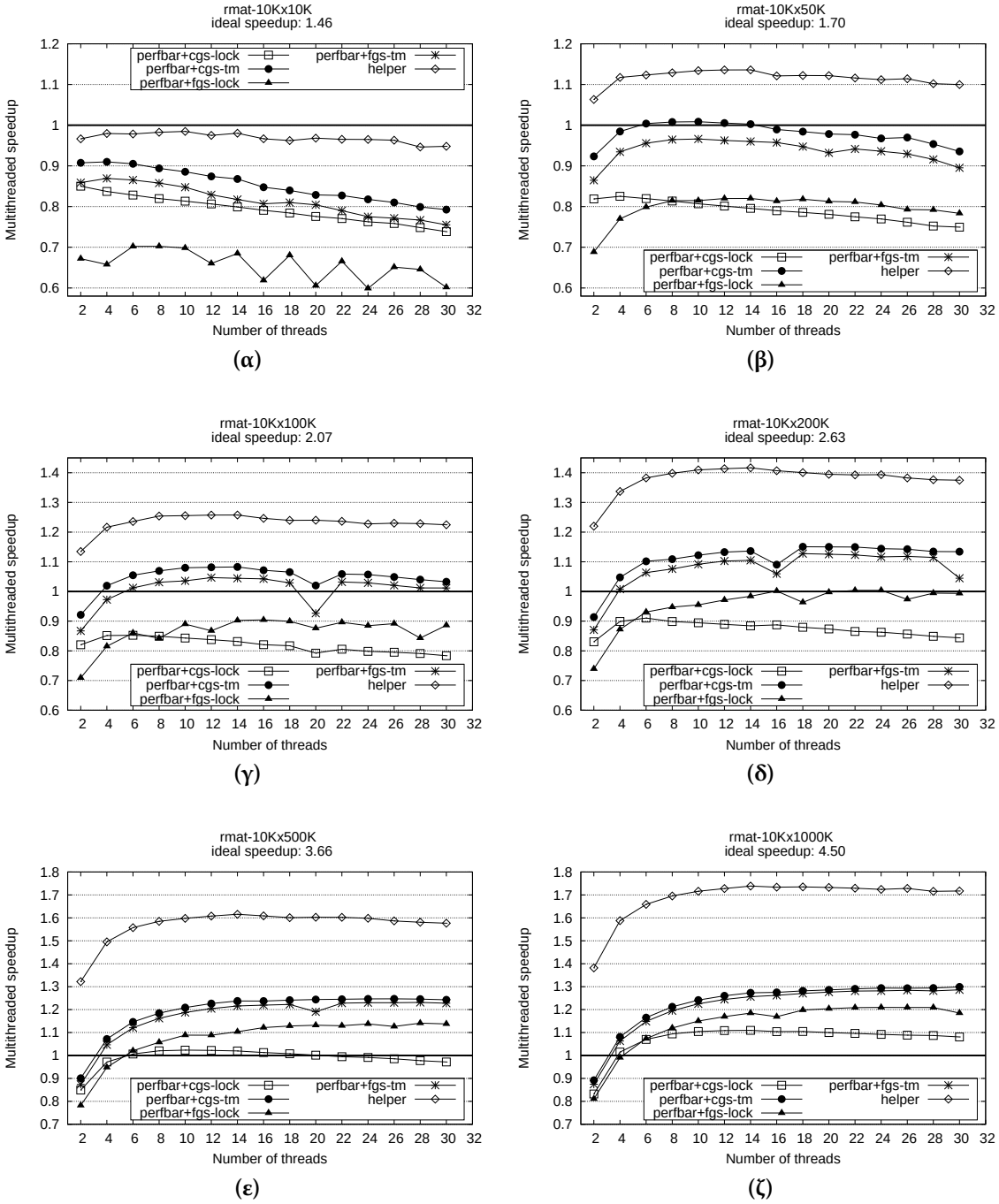
όπου n είναι ο αριθμός κορυφών του γραφήματος και d ο μέσος βαθμός εξόδου των κορυφών. Το πρώτο μέρος της Εξίσωσης 5.1 προσεγγίζει το χρόνο που αφιερώνεται στις λειτουργίες `ExtractMin`, ενώ το δεύτερο το χρόνο που αφιερώνεται στις λειτουργίες `DecreaseKey`. Παρομοίως, ο χρόνος εκτέλεσης του σχήματος *helper* μπορεί να προσεγγιστεί ως εξής:

$$T_{helper} = n \times O(\lg n) + a \times d \times n \times O(\lg n), a < 1 \tag{5.2}$$

όπου a είναι ο λόγος των λειτουργιών `DecreaseKey` που εκτελεί το κύριο νήμα σε σχέση με αυτές που εκτελούνται στη σειριακή περίπτωση. Αυτή είναι μια απλοϊκή προσέγγιση που δε λαμβάνει υπόψη της το χρόνο που ξοδεύεται στην ενορχήστρωση των νημάτων ή την καθυστέρηση εξαιτίας συγκρουόμενων διενεργειών. Η αναμενόμενη επιτάχυνση s του σχήματος *helper* θα μπορούσε συνεπώς να προσεγγιστεί από τη σχέση $s = \frac{T_{serial}}{T_{helper}} = \frac{1+d}{1+a \times d}$, που υπονοεί ότι το s θα πρέπει να αυξάνει με το μέσο βαθμό εξόδου, και συνεπώς με την αυξανόμενη πυκνότητα του γραφήματος, εξηγώντας έτσι τα αποτελέσματα στα Σχήματα 5.13, 5.14 και 5.15. Στα σχήματα αυτά επίσης φαίνεται για την περίπτωση του *helper* ότι η επιτάχυνση αυξάνει καθώς χρησιμοποιούνται περισσότερα βοηθητικά νήματα. Αυτή η ανοδική τάση φτάνει μέχρι κάποιο μέγιστο σημείο, πέρα από το οποίο η χρήση περισσότερων νημάτων οδηγεί σε ελαφρά υποβάθμιση της απόδοσης. Ο αριθμός των νημάτων που απαιτούνται για να φτάσουμε αυτό το μέγιστο σημείο, εξαρτάται πάλι από την πυκνότητα του γραφήματος.



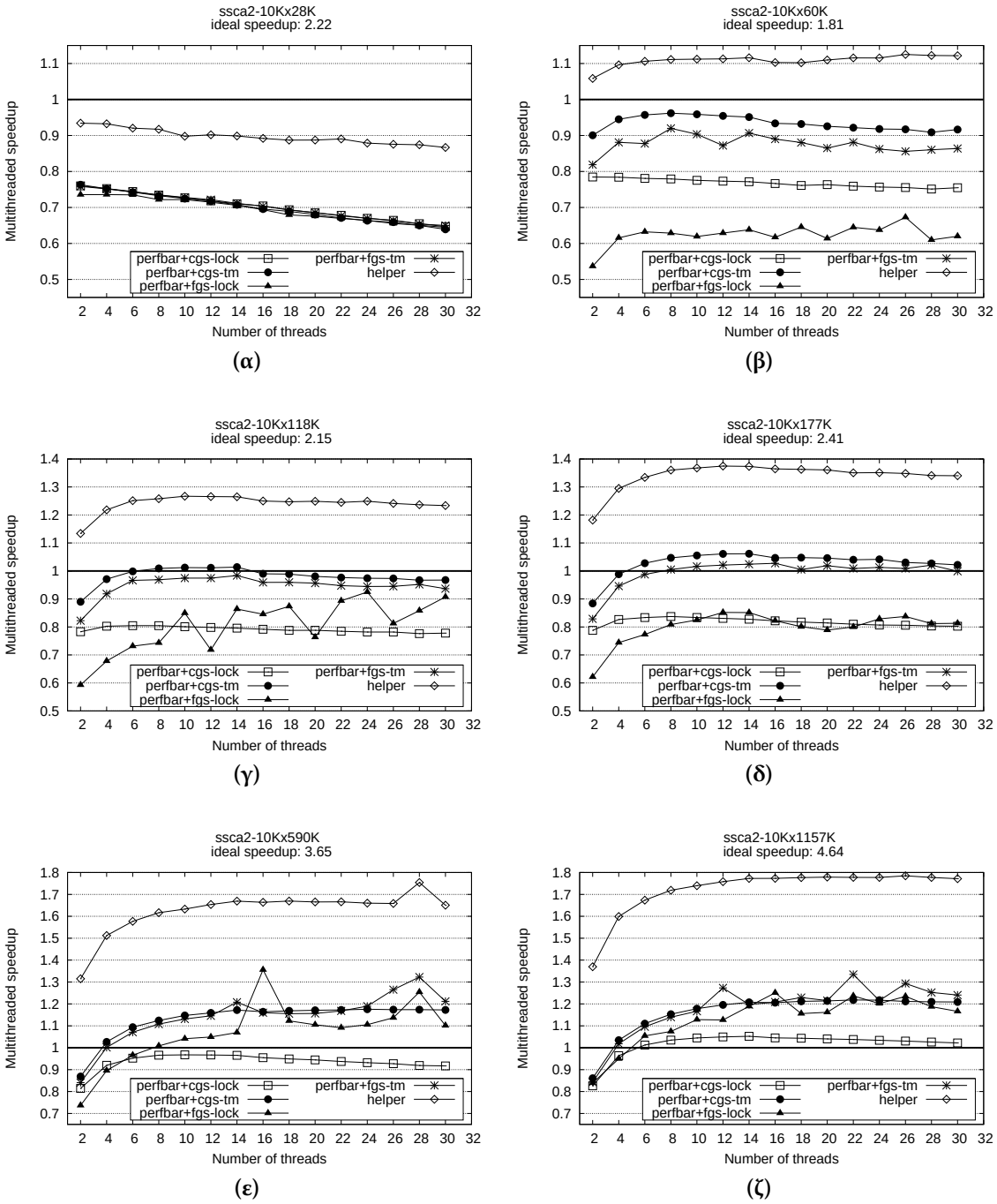
Σχήμα 5.13: *Επιτάχυνση των διαφόρων πολυνηματικών σχημάτων για τα γραφήματα της οικογένειας* Random.



Σχήμα 5.14: *Επιτάχυνση των διαφόρων πολυνηματικών σχημάτων για τα γραφήματα της οικογένειας* R-MAT.

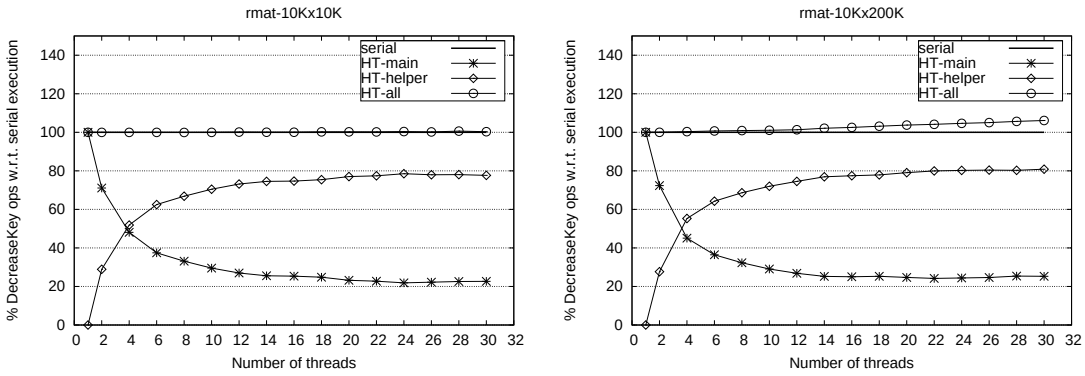
5.5.4 Ερμηνεύοντας την απόδοση του σχήματος *helper*

Σε αυτήν την ενότητα επιχειρούμε να αποκτήσουμε μια πιο πλήρη εικόνα για τη συμπεριφορά του προτεινόμενου σχήματος χρησιμοποιώντας επιπλέον μετρικές και στατιστικά. Εστιάζουμε σε μία μόνο οικογένεια γραφημάτων, την R-MAT, καθώς οι υπόλοιπες οικογένειες επιδεικνύουν ανάλογη συμπεριφορά, και επιλέγουμε τρία αντιπροσωπευτικά γραφήματα με διαφορετική πυκνότητα: *μικρή* (10K), *μεσαία* (200K) και *μεγάλη* (1000K).



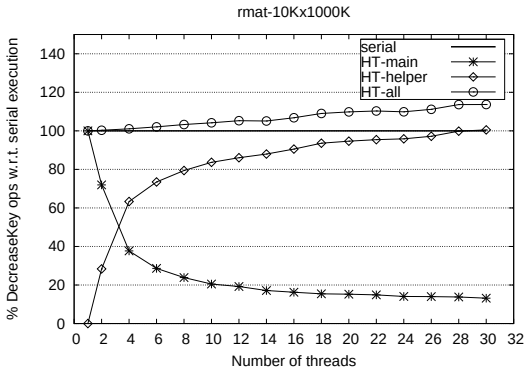
Σχήμα 5.15: *Επιτάχυνση των διαφόρων πολυνηματικών σχημάτων για τα γραφήματα της οικογένειας* SSCA#2*.*

Το Σχήμα 5.16 απεικονίζει την κατανομή των λειτουργιών `DecreaseKey` ανάμεσα στο κύριο και τα βοηθητικά νήματα και τις συγκρίνει με αυτές που πραγματοποιήθηκαν στα πλαίσια της



(α) $m = 10K$

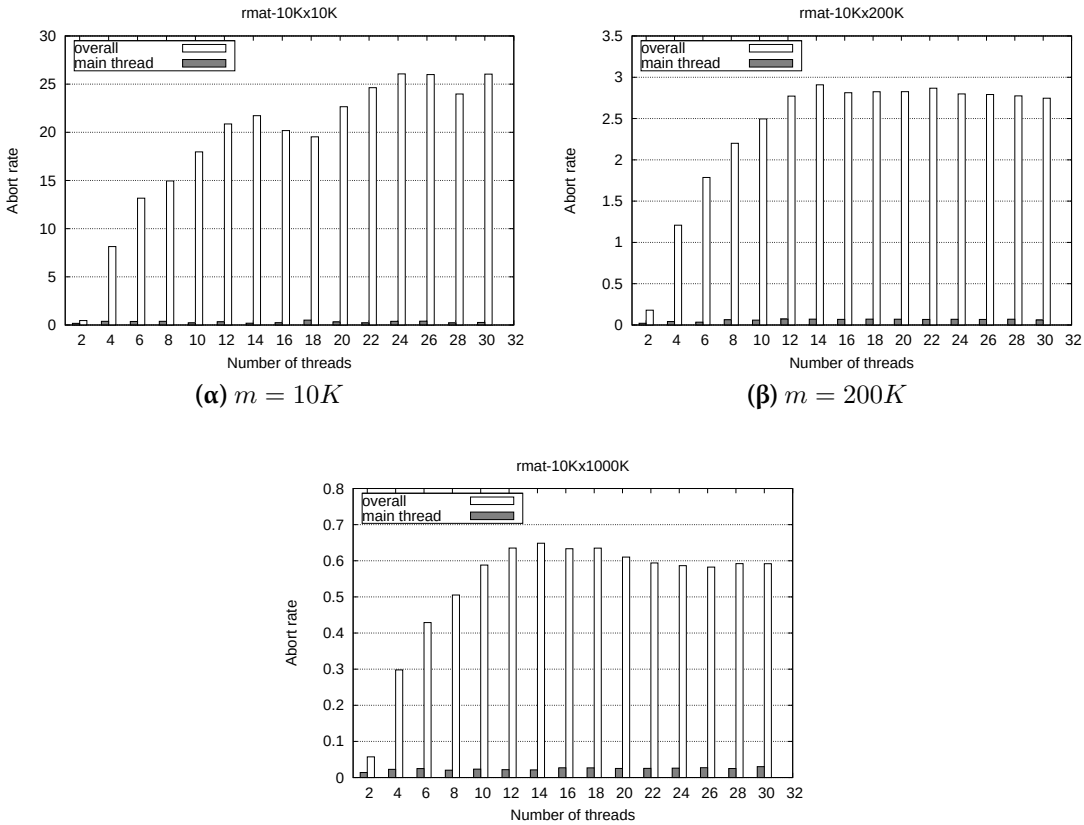
(β) $m = 200K$



(γ) $m = 1000K$

Σχήμα 5.16: *Κατανομή των λειτουργιών* `DecreaseKey` *ανάμεσα στο κύριο και τα βοηθητικά νήματα. Για το σχήμα* helper *απεικονίζονται τα ποσοστά λειτουργιών* `DecreaseKey` *που εκτέλεσαν το κύριο νήμα (*HT-main*), τα βοηθητικά νήματα (*HT-helper*) και όλα τα νήματα συνολικά (*HT-all*), σε σχέση με το συνολικό αριθμό λειτουργιών* `DecreaseKey` *της σειριακής περίπτωση (*serial*).*

σειριακής εκτέλεσης. Καθώς χρησιμοποιούνται περισσότερα νήματα, οι λειτουργίες `DecreaseKey` του κύριου νήματος μειώνονται, εξηγώντας τη βελτίωση της απόδοσης. Ωστόσο, δεν είναι όλες οι λειτουργίες που εκτελούν τα βοηθητικά νήματα χρήσιμες, όπως φαίνεται στα Σχήματα 5.16β και 5.16γ, όπου ο συνολικός αριθμός λειτουργιών `DecreaseKey` είναι μεγαλύτερος από αυτόν της σειριακής περίπτωσης. Αυτή η συμπεριφορά εξηγεί γιατί η απόδοση δε συνεχίζει να βελτιώνεται μετά από κάποιο αριθμό νημάτων. Είναι αξιοσημείωτο ότι αντίστοιχες μειώσεις στις λειτουργίες `DecreaseKey` του κύριου νήματος παρατηρούνται επίσης για την περίπτωση του αραιού γραφήματος, όπως φαίνεται στο Σχήμα 5.16α. Ωστόσο, το Σχήμα 5.14α δείχνει ότι σε αυτήν την περίπτωση το σχήμα *helper* αποδίδει χειρότερα σε σχέση με το σειριακό. Αυτό μπορεί να αποδοθεί στο *ποσοστό ματαιώσεων* (abort rate) των διενεργειών σε αυτήν την περίπτωση,



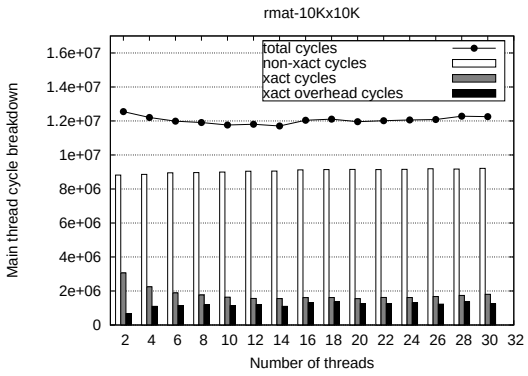
(α) $m = 10K$ **(β)** $m = 200K$

(γ) $m = 1000K$

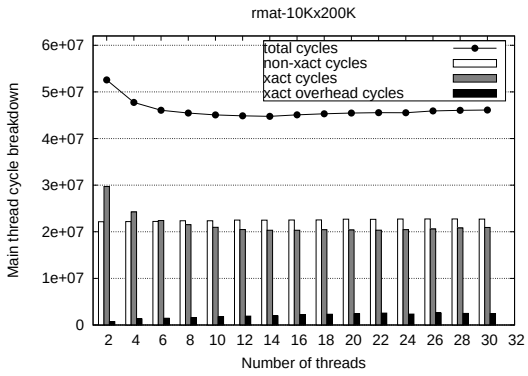
Σχήμα 5.17: *Ποσοστά ματαιώσεων για όλες τις διενέργειες (*overall*) και για τις διενέργειες του κύριου νήματος (*main thread*).*

δηλαδή το λόγο του συνολικού αριθμού ματαιωμένων διενεργειών προς το συνολικό αριθμό ολοκληρωμένων διενεργειών, ο οποίος παρουσιάζεται στο Σχήμα 5.17. Είναι προφανές ότι για το αραιό γράφημα το ποσοστό ματαιώσεων είναι αρκετά υψηλό, απαλείφοντας οποιαδήποτε κέρδη στην απόδοση από την εκμετάλλευση του παραλληλισμού. Παρόλα αυτά, για τα πιο πυκνά γραφήματα το ποσοστό ματαιώσεων μειώνεται σημαντικά με αποτέλεσμα να επιτυγχάνονται βελτιώσεις. Μια σημαντική παρατήρηση ωστόσο είναι ότι σε κάθε περίπτωση το ποσοστό ματαιώσεων του κύριου νήματος είναι ιδιαίτερα χαμηλό, που δείχνει ότι δεν παρεμποδίζεται από τα βοηθητικά νήματα. Αυτό εξηγεί την ισχύ του προτεινόμενου σχήματος, καθώς στην χειρότερη περίπτωση σημειώνεται επιβράδυνση της τάξεως του 0.95.

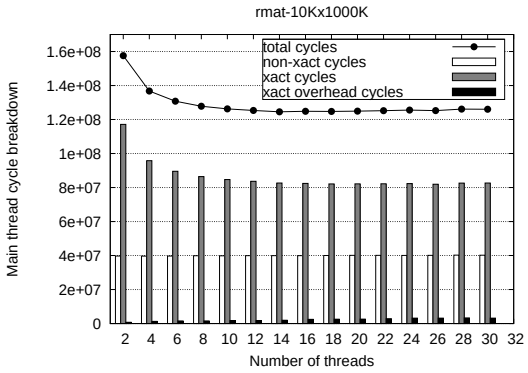
Στο ίδιο συμπέρασμα καταλήγουμε από το Σχήμα 5.18, όπου παρουσιάζεται μια αναλυτική κατανομή των κύκλων εκτέλεσης του κύριου νήματος σε κύκλους εκτός διενεργειών (*non-xact*), κύκλους εντός διενεργειών (*xact*) και μη-χρήσιμους κύκλους που σπαταλήθηκαν σε ματαιώσεις



(α) $m = 10K$



(β) $m = 200K$

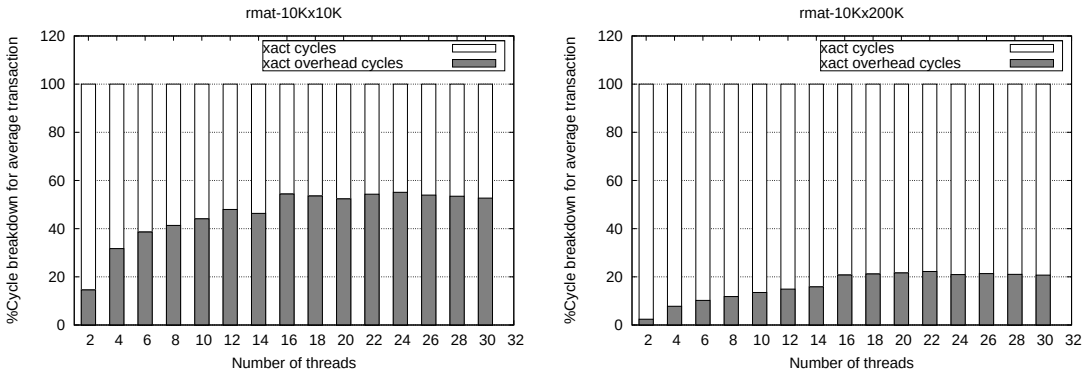


(γ) $m = 1000K$

Σχήμα 5.18: *Αναλυτική κατανομή των κύκλων του κύριου νήματος σε κύκλους εκτός διενεργειών (*non-xact*), κύκλους εντός διενεργειών (*xact*) και μη-χρήσιμους κύκλους που σπαταλήθηκαν λόγω διενέξεων (*xact overhead*).*

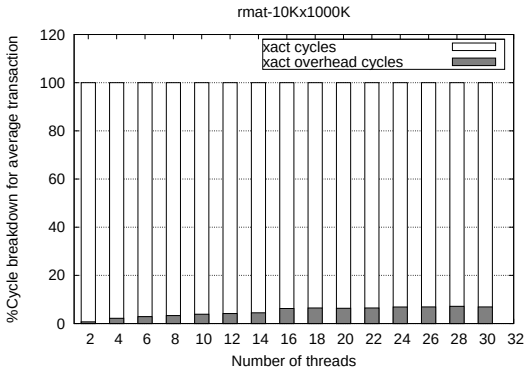
ή καθυστερήσεις λόγω διενέξεων μεταξύ των διενεργειών (*xact overhead*). Οι κύκλοι εκτός διενεργειών παραμένουν σταθεροί για κάθε γράφημα, αφού αντιπροσωπεύουν κυρίως το χρόνο που αφιερώθηκε στις λειτουργίες `ExtractMin` οι οποίες δεν επηρεάζονται από το σχήμα μας. Η προσθήκη των βοηθητικών νημάτων μειώνει το χρόνο που ξοδεύεται μέσα σε διενέργειες, δηλαδή στην ουσία το παράλληλο τμήμα του σχήματός μας, αφού το κύριο νήμα εκτελεί λιγότερες λειτουργίες `DecreaseKey`, όπως δείξαμε προηγουμένως. Οι μη-χρήσιμοι κύκλοι που σπαταλώνται λόγω διενέξεων είναι σχετικά λίγοι, πράγμα που δείχνει για ακόμη μια φορά ότι το κύριο νήμα δεν επηρεάζεται από αρνητικές καταστάσεις των βοηθητικών νημάτων.

Για την καλύτερη κατανόηση της δουλειάς που σπαταλάται σε περιπτώσεις διενέξεων, το Σχήμα 5.19 παρουσιάζει το προφίλ εκτέλεσης μιας μέσης διενέργειας. Στην ουσία απεικονίζει το ποσοστό των συνολικών μη-χρήσιμων κύκλων που σπαταλήθηκαν σε όλα τα νήματα εξαιτίας ματαιώσεων και καθυστερήσεων, σε σχέση με το συνολικό αριθμό κύκλων που ξοδεύτηκαν



(α) $m = 10K$

(β) $m = 200K$



(γ) $m = 1000K$

Σχήμα 5.19: *Ποσοστό μη-χρήσιμων κύκλων που σπαταλήθηκαν λόγω διενέξεων (*xact overhead*) ως προς το συνολικό αριθμό κύκλων που κάποιο νήμα ξόδεψε μέσα σε διενέργειες (*xact*).*

σε διενέργειες. Και σε αυτήν την περίπτωση, για γραφήματα με μέση ή μεγάλη πυκνότητα το ποσοστό της μη-χρήσιμης δουλειάς είναι σχετικά χαμηλό, δικαιολογώντας τις παρατηρούμενες επιταχύνσεις. Αντίθετα, σημαντικό μέρος των κύκλων πάει χαμένο στο αραιό γράφημα, εξηγώντας την έλλειψη οποιασδήποτε βελτίωσης σε αυτήν την περίπτωση.

Σε γενικές γραμμές, το μικρό ποσοστό χαμένης δουλειάς δείχνει ότι οι περισσότερες από τις ταυτόχρονες προσβάσεις στις μοιραζόμενες δομές δεδομένων είναι μη-συγκρουόμενες. Η συχνότητα εμφάνισης διενέξεων εξαρτάται επίσης από το μέσο μέγεθος του συνόλου εγγραφών μιας μέσης διενέργειας. Όσο περισσότερα είναι τα δεδομένα που τροποποιούνται εντός μιας διενέργειας τόσο μεγαλύτερη πιθανότητα υπάρχει να προκύψει κάποια διένεξη με άλλες διενέργειες. Ο Πίνακας 5.3 παρουσιάζει για κάθε διαφορετικό μέγεθος γραφήματος το προφίλ μιας μέσης διενέργειας ως προς το μέγεθος του συνόλου εγγραφών της. Η δεύτερη στήλη δείχνει το εύρος του μέσου αριθμού εγγραφών για όλες τις διενέργειες όλων των νημάτων, ενώ η τρίτη στήλη παρουσιάζει την ίδια πληροφορία μόνο για τις διενέργειες εκείνες που περιλαμβάνουν

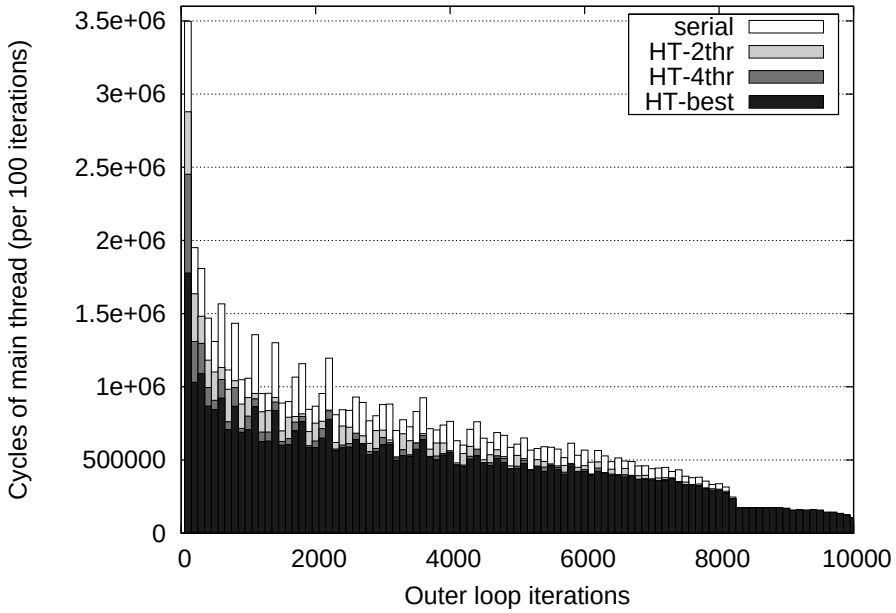
Πυκνότητα γραφήματος	Μέσο μέγεθος συνόλου εγγραφών	Μέσο μέγεθος συνόλου εγγραφών για διενέργειες με `DecreaseKey`'s	Μέγιστο μέγεθος συνόλου εγγραφών
10K	1.31 - 3.14	12.44 - 20.02	28 - 31
50K	1.16 - 2.07	8.26 - 12.08	29 - 31
100K	1.08 - 1.71	7.84 - 10.79	28 - 30
200K	1.04 - 1.52	7.66 - 9.83	28 - 31
500K	1.02 - 1.20	7.54 - 8.81	27 - 31
1000K	1.01 - 1.12	7.67 - 8.68	29 - 36

Πίνακας 5.3: *Προφίλ διενεργειών ως προς το μέγεθος του συνόλου εγγραφών.*

την εκτέλεση λειτουργιών `DecreaseKey`. Η τέταρτη στήλη δείχνει το μέγιστο πλήθος εγγραφών που σημειώθηκαν σε κάποια διενέργεια. Παρατηρούμε ότι το μέσο μέγεθος του συνόλου εγγραφών είναι γενικά μικρό, οδηγώντας έτσι σε μικρή πιθανότητα εμφάνισης διενέξεων. Αυτά τα ευρήματα επιβεβαιώνουν ότι, εξαιτίας της αισιόδοξης φύσης της, η μνήμη διενεργειών είναι καλύτερη προσέγγιση από τα κλειδώματα για την υλοποίηση του προτεινόμενους σχήματος, όπως εξηγήσαμε και στην ενότητα 5.4.4.

Ένα επιπλέον σχόλιο που μπορούμε να κάνουμε είναι ότι το μικρό σχετικά μέγεθος των διενεργειών τις καθιστά κατάλληλες για εκτέλεση σε κάποιο ρεαλιστικό, "βέλτιστης-προσπάθειας" HTM σύστημα, το οποίο συνήθως θέτει περιορισμούς όσον αφορά την επιτυχή εκτέλεση διενεργειών. Ένα τέτοιο παράδειγμα είναι ο επεξεργαστής Rock της Sun [Tremblay 08], στον οποίον οι διενέργειες που υπερβαίνουν τα όρια χρήσης κάποιου φυσικού πόρου του τσιπ (π.χ., χωρητικότητα ενός συνόλου της κρυφής μνήμης ή του απομονωτή εγγραφών) ματαιώνονται αυτόματα από τον επεξεργαστή. Ο Rock, για παράδειγμα, επιτρέπει την εκτέλεση διενεργειών με μέγιστο πλήθος εγγραφών ίσο με 32, που είναι και το μέγεθος του απομονωτή εγγραφών του επεξεργαστή. Όπως βλέπουμε από τον Πίνακα 5.3, η μέση διενέργεια του σχήματός μας τηρεί αυτόν τον περιορισμό.

Τέλος, το Σχήμα 5.20 συγκρίνει τους κύκλους που χρειάζεται το κύριο νήμα ανά 100 επαναλήψεις του εξωτερικού βρόχου, όταν εκτελείται παράλληλα με 0 (*serial*), 1 (*HT-2thr*), 3 (*HT-4thr*) και 13 (*HT-best*) βοηθητικά νήματα. Τα αποτελέσματα αυτά αφορούν το γράφημα R-MAT με 200K πλευρές. Η πρώτη παρατήρηση που κάνουμε είναι ότι η πλειοψηφία του χρόνου εκτέλεσης αντιστοιχεί στο πρώτο 30% των επαναλήψεων. Παρατηρούμε επίσης ότι καθώς εξελίσσεται η εκτέλεση, ο διαθέσιμος παραλληλισμός περιορίζεται και τα κέρδη από τη χρήση περισσότερων βοηθητικών νημάτων είναι σχεδόν αμελητέα. Στην πραγματικότητα, στο τελευταίο 20% των



Σχήμα 5.20: *Χρονική εξέλιξη της εκτέλεσης ανά 100 επαναλήψεις του εξωτερικού βρόχου για το σειριακό αλγόριθμο (*serial*), και το σχήμα* helper *με 1 (*HT-2thr*), 3 (*HT-4thr*) και 13 (*HT-best*) βοηθητικά νήματα.*

επαναλήψεων το κύριο νήμα ξοδεύει τον ίδιο χρόνο τόσο αν εκτελείται μόνο του, όσο και αν εκτελείται με 13 βοηθητικά νήματα ταυτόχρονα. Αυτό θα μπορούσε να αποτελέσει τη βάση για διερεύνηση δυναμικών σχημάτων πολυνηματισμού όπου ο αριθμός των βοηθητικών νημάτων θα μπορούσε να προσαρμόζεται αυτόματα κατά τον χρόνο εκτέλεσης.

5.6 Συμπεράσματα και Μελλοντικές Κατευθύνσεις

Στα πλαίσια αυτής της δουλειάς προσπαθήσαμε να παραλληλοποιήσουμε τον αλγόριθμο του Dijkstra, μια εγγενώς σειριακή εφαρμογή που είναι γνωστό ότι η παραλληλοποίησή της αποτελεί δύσκολο πρόβλημα. Αρχικά, αναπτύξαμε λεπτομερή πολυνηματικά σχήματα τα οποία παραλληλοποιούν κάθε εξωτερικό, σειριακό βήμα του αλγορίθμου χρησιμοποιώντας παραδοσιακές μεθόδους συγχρονισμού (κλειδώματα και φράγματα). Αυτά τα σχήματα παρουσιάζουν χαμηλή απόδοση, ακόμα και όταν τα χρονοβόρα φράγματα λογισμικού αντικαθίστανται με ιδανικά φράγματα υλικού μηδενικής καθυστέρησης. Για να αποφύγουμε το αυξημένο κόστος των κλειδωμάτων καταφεύγουμε στη χρήση του μοντέλου της μνήμης διενεργειών. Τα πολυνηματικά σχήματα που χρησιμοποιούν αυτόν τον μηχανισμό πράγματι καταφέρνουν να μειώσουν συνολικά το κόστος του συγχρονισμού, δίνοντας καλύτερα αποτελέσματα σε σχέση με τις εκδόσεις που χρησιμοποιούν κλειδώματα, όμως εξακολουθούν να προϋποθέτουν την ύπαρξη ιδανικών φραγμάτων για την επίτευξη επιτάχυνσης.

Για να αποφύγουμε τη χρήση φραγμάτων και να εξάγουμε πιο αδρομερή παραλληλισμό, προτείναμε ένα σχήμα υποθετικής παραλληλοποίησης που συνδυάζει την ιδέα των βοηθητικών νημάτων και της μνήμης διενεργειών. Τα βοηθητικά νήματα εκτελούνται παράλληλα με το κύριο νήμα της εφαρμογής και το αποφορτίζουν από υπολογισμούς, εκτελώντας με υποθετικό τρόπο ένα σημαντικό μέρος των λειτουργιών `DecreaseKey`. Για την υλοποίηση υιοθετούμε τη μνήμη διενεργειών, όχι μόνο για την ευκολία προγραμματισμού που προσφέρει, αλλά και εξαιτίας της φύσης της, που μας δίνει την ευκαιρία να αναδείξουμε τον μη προφανή παραλληλισμό που εμπεριέχεται στην εφαρμογή. Η πειραματική αξιολόγηση αποκάλυψε ότι το σχήμα αυτό είναι σε θέση να δώσει σημαντικά μεγέθη επιτάχυνσης στην πλειονότητα των περιπτώσεων, τα οποία φτάνουν μέχρι και το 1.84, ενώ υπερτερεί σημαντικά όλων των υπολοίπων πολυνηματικών εκδόσεων οι οποίες στηρίζονται σε ιδανικά φράγματα. Η περαιτέρω ανάλυση των στατιστικών επιβεβαίωσε την ύπαρξη του αισιόδοξου παραλληλισμού δικαιώνοντας έτσι την επιλογή της μνήμης διενεργειών.

Σαν μελλοντική δουλειά, θα εξετάσουμε την εφαρμοσιμότητα του προτεινόμενου σχήματος και σε άλλους αλγορίθμους που επιλύουν το SSSP πρόβλημα, όπως ο Δ-stepping και ο Bellman-Ford. Επίσης, θα διερευνήσουμε την εφαρμοσιμότητά του και σε αλγόριθμους που, όπως ο Dijkstra, έχουν το χαρακτηριστικό να λειτουργούν *άπληστα* (greedy algorithms), αναζητώντας σε κάθε βήμα τοπικά βέλτιστες λύσεις. Ένας τέτοιο παράδειγμα είναι ο αλγόριθμος του Kruskal για εύρεση ελάχιστων γεννητικών δέντρων σε ένα γράφημα. Και εδώ, το σκεπτικό είναι να έχουμε βοηθητικά νήματα να επεξεργάζονται για λογαριασμό του κύριου νήματος τυχόν επόμενες βέλτιστες λύσεις. Επιπλέον, σκοπεύουμε να εξετάσουμε την επίδραση που έχουν στην απόδοση του προτεινόμενου σχήματος διάφορα χαρακτηριστικά λειτουργίας της μνήμης διενεργειών, όπως η πολιτική επίλυσης διενέξεων ή η διαχείριση των εκδόσεων των δεδομένων. Αναφορικά με το μεταβαλλόμενο προφίλ που έχει η εκτέλεση του αλγορίθμου (Σχήμα 5.20), θα είχε ενδιαφέρον να διερευνήσουμε περαιτέρω βελτιστοποιήσεις και μεθόδους βάση των οποίων ο αριθμός των νημάτων θα μπορούσε να ρυθμίζεται αυτόματα, είτε επειδή τα επιπλέον νήματα δε συνεισφέρουν στην απόδοση (σπαταλώντας έτσι άσκοπα πόρους), είτε επειδή λειτουργούν καταστρεπτικά (αυξάνοντας τις διενέξεις, εκτοπίζοντας δεδομένα από την κρυφή μνήμη, κ.λπ.). Η δυναμική συλλογή γενικών μετρικών απόδοσης (π.χ. κύκλοι μηχανής, αστοχίες) ή συγκεκριμένων στατιστικών του HTM συστήματος (π.χ. αριθμός ματαιώσεων) θα μπορούσε να φανεί χρήσιμη προς αυτήν την κατεύθυνση.

5.6.1 Ρητή υποστήριξη υποθετικού πολυνηματισμού

Στην περίπτωση εφαρμογής του προτεινόμενου σχήματος υποθετικής παραλληλοποίησης στον αλγόριθμο του Dijkstra στηριχθήκαμε σε ένα βαθμό στη σημασιολογία του ίδιου του αλγορίθμου για τη σωστή διαχείριση της υποθετικής εκτέλεσης, κυρίως όσον αφορά τη διόρθωση υπολογισμών των βοηθητικών νημάτων οι οποίοι βασίζονται σε λάθος υποθέσεις. Πώς θα μπορούσαμε να εφαρμόσουμε το προτεινόμενο σχήμα αν μια εφαρμογή δεν έχει εκ φύσεως τέτοια χαρακτηριστικά; Το ερώτημα αυτό μεταφράζεται στο εξής: πώς θα μπορούσαμε να κάνουμε εφικτή την υποστήριξη ενός γενικότερου μοντέλου υποθετικού πολυνηματισμού πάνω από ένα σύστημα μνήμης διενεργειών υλικού; *Είναι δυνατόν με μικρές επεκτάσεις να διευρύνουμε το πεδίο εφαρμογής ενός συστήματος μνήμης διενεργειών ώστε να μπορεί να χρησιμοποιηθεί σε σενάρια πέρα από αυτά για τα οποία προορίζεται;* Αυτό αποτελεί κατά τη γνώμη μας την πιο ενδιαφέρουσα ίσως κατεύθυνση για μελλοντική έρευνα που προέκυψε από την ενασχόλησή μας με την παραλληλοποίηση του Dijkstra. Εστιάζουμε σε σύστημα μνήμης διενεργειών υλικού, λόγω της πολύ μικρής επιβάρυνσης που εισάγουν οι διαχειριστικοί του μηχανισμοί σε σχέση με ένα σύστημα λογισμικού.

Προτού συζητήσουμε πώς πιστεύουμε ότι θα μπορούσε να γίνει κάτι τέτοιο, ας δούμε πρώτα κάποιες από τις ομοιότητες και διαφορές ανάμεσα σε ένα σύστημα μνήμης διενεργειών και σε ένα σύστημα υποθετικού πολυνηματισμού. Οι ομοιότητές τους αφορούν κυρίως τους εσωτερικούς μηχανισμούς λειτουργίας, με βασικότερους αυτούς για τη διαχείριση των εκδόσεων των δεδομένων, την ανίχνευση των διενέξεων και την επανεκκίνηση των διενεργειών/υποθετικών εργασιών. Η πιο βασική διαφορά είναι ότι τα συστήματα υποθετικού πολυνηματισμού υποθέτουν συγκεκριμένη προτεραιότητα ανάμεσα στις υποθετικές εργασίες όσον αφορά την επίλυση των μεταξύ τους διενέξεων και την επικύρωση των αποτελεσμάτων τους. Η προτεραιότητα αυτή υπαγορεύεται από τη σειρά των εργασιών στο αρχικό, σειριακό πρόγραμμα. Αντίθετα, οι διενέξεις ανάμεσα σε διενέργειες επιλύονται με βάση την εκάστοτε πολιτική, ενώ η ολοκλήρωση των διενεργειών δεν γίνεται με κάποια συγκεκριμένη σειρά. Μία άλλη διαφορά είναι ότι τα συστήματα υποθετικού πολυνηματισμού διαθέτουν μηχανισμούς για μεταβίβαση των απαραίτητων δεδομένων εισόδου από μια εργασία σε μια μεταγενέστερή της (live-in values forwarding). Τέλος, υποθέτουν κάποιο συγκεκριμένο μοντέλο δημιουργίας και τερματισμού των υποθετικών εργασιών, κάτι που δεν προβλέπεται στη μνήμη διενεργειών.

Με δεδομένες αυτές τις ομοιότητες και διαφορές, θεωρούμε ότι με απλές επεκτάσεις σε ένα HTM σύστημα θα μπορούσαμε να θέσουμε στον έλεγχο του χρήστη τους εσωτερικούς του μηχανισμούς ώστε να κάνουμε αμεσότερη την υποστήριξη του υποθετικού πολυνηματισμού. Επιπλέον, προκειμένου να διατηρείται μικρή η κατασκευαστική πολυπλοκότητα των επεκτάσεων θεωρούμε εξίσου απαραίτητη τη συνδρομή από το επίπεδο του λογισμικού.

Στο επίπεδο υλικού, το HTM σύστημα καθώς και το σύνολο εντολών του επεξεργαστή θα μπορούσε να επεκταθεί ώστε να δίνει τη δυνατότητα στον χρήστη να αναθέτει προτεραιότητες στις διενέργειες, που θα καθορίζουν τον "νικητή" σε περίπτωση διενέξεων αλλά και τη σειρά ολοκλήρωσής τους. Με αυτόν τον τρόπο, στα πλαίσια ενός μοντέλου υποθετικού πολυνηματισμού οι διενέργειες στις οποίες απεικονίζονται προγενέστερα τμήματα του κώδικα μπορούν να προγραμματιστούν ώστε να έχουν υψηλότερη προτεραιότητα έναντι αυτών που εκτελούν μεταγενέστερα τμήματα.

Άλλη επέκταση η οποία πιθανώς θα βοηθούσε είναι η δυνατότητα για έναν επεξεργαστή να ελέγχει άμεσα μία ή περισσότερες διενέργειες που εκτελούνται σε άλλους επεξεργαστές, και να επιβάλλει την επικύρωση ή την ματαίωσή τους. Αυτό θα μπορούσε να χρησιμοποιηθεί από το μη-υποθετικό νήμα όταν ολοκληρώνει τους υπολογισμούς του και θέλει να ελέγξει την κατάσταση των υπολοίπων υποθετικών νημάτων, ώστε να ανακηρύξει το παλαιότερο από αυτά σαν το νέο μη-υποθετικό και να αναλάβει το επόμενο κομμάτι υπολογισμών. Επίσης, στην περίπτωση του προτεινόμενου σχήματος παραλληλοποίησης του Dijkstra που παρουσιάσαμε, ένας τέτοιος μηχανισμός θα έδινε τη δυνατότητα στο κύριο νήμα κάθε φορά που θέλει να προχωρήσει στην επόμενη επανάληψη να σταματά άμεσα τα βοηθητικά νήματα.

Όπως αναφέραμε στην ενότητα 5.1.1, τα περισσότερα συστήματα υποθετικού πολυνηματισμού που έχουν προταθεί στη βιβλιογραφία βασίζονται σε ειδικό υλικό για τη δημιουργία, δρομολόγηση και τερματισμό των υποθετικών εργασιών, καθώς και για μεταβίβαση δεδομένων εισόδου μεταξύ τους. Με τέτοιους μηχανισμούς οι λειτουργίες αυτές γίνονται σε λίγους μόνο κύκλους, κάτι όμως που είναι μη-ρεαλιστικό αφού οι σύγχρονοι επεξεργαστές δεν ενσωματώνουν αντίστοιχους μηχανισμούς ούτε προβλέπεται να το κάνουν στο άμεσο μέλλον. Ωστόσο, το διαχειριστικό κόστος και μόνο των υποθετικών εργασιών, όπως αποτυπώνεται στις λειτουργίες αυτές, αποτελεί καθοριστικό παράγοντα για την επιτυχία του υποθετικού πολυνηματισμού.

Σαν πιο πραγματιστική προσέγγιση για την υλοποίηση αυτών των λειτουργιών θεωρούμε την χρησιμοποίηση κάποιου συστήματος χρόνου εκτέλεσης. Το σύστημα αυτό θα πρέπει να είναι προσανατολισμένο στην αποδοτική διαχείριση μεγάλου αριθμού παράλληλων εργασιών μικρής έκτασης (fine-grain tasks) όσον αφορά τη δημιουργία, τη δρομολόγηση και τον τερματισμό τους. Επίσης, θα ελέγχει ανά πάσα στιγμή τι εργασίες εκτελούνται, σε τι κατάσταση βρίσκονται, τι προτεραιότητα έχουν, και θα φροντίζει ώστε να ολοκληρώνονται με βάση την προτεραιότητά τους ώστε να τηρούνται οι εξαρτήσεις της σειριακής εκτέλεσης. Επιπλέον, θα διαχειρίζεται κατάλληλα τις διενέξεις ανάμεσα στις υποθετικές εργασίες, εξασφαλίζοντας την απρόσκοπτη εκτέλεση της "νικήτριας" εργασίας και δρομολογώντας την επανεκτέλεση εκείνων που ματαιώνονται. Τα τελευταία χρόνια έχουν αναπτυχθεί διάφορα συστήματα χρόνου εκτέλεσης τα οποία εστιάζουν στον λεπτομερή παραλληλισμό βασισμένο σε εργασίες και τα οποία θα μπορούσαμε να χρησιμοποιήσουμε και να προσαρμόσουμε για τις ανάγκες του υποθετικού πολυνηματισμού. Χαρακτηριστικά αναφέρουμε τα συστήματα Cilk [Frigo 98], Intel Threading Building Blocks [tbb 08]

και υλοποιήσεις της έκδοσης 3.0 του OpenMP [omp 08].

Τέλος, σε υψηλότερο επίπεδο θα θέλαμε να αναπτύξουμε κάποιο προγραμματιστικό μοντέλο ή επεκτάσεις γλωσσών προγραμματισμού που θα επιτρέψουν την εύκολη και αποδοτική έκφραση του υποθετικού παραλληλισμού στο σειριακό πρόγραμμα, σε δομές όπως επαναλήψεις βρόχων ή κλήσεις συναρτήσεων. Το προγραμματιστικό μοντέλο αυτό θα είναι υψηλού επιπέδου και θα δίνει τη δυνατότητα στον χρήστη να γράφει το παράλληλο πρόγραμμα με απλές επαυξήσεις πάνω στο αρχικό σειριακό, με ειδικούς ιδιωματισμούς, επισημειώσεις ή προσδιοριστές, χωρίς να χρειάζεται να αναπτύξει περίπλοκο πολυνηματικό κώδικα χαμηλού επιπέδου. Μέσω κάποιου εργαλείου μεταγλώττισης, η υψηλού επιπέδου αναπαράσταση του υποθετικού παραλληλισμού μπορεί να μεταφράζεται σε χαμηλού επιπέδου κώδικα ο οποίος θα χρησιμοποιεί τη προγραμματιστική διεπαφή του συστήματος χρόνου εκτέλεσης και το επεκτεταμένο σύνολο εντολών του επεξεργαστή.

Σε αντίθεση με τα αρχιτεκτονικά μοντέλα, η σχετική βιβλιογραφία γύρω από τον υποθετικό πολυνηματισμό δεν έχει να επιδείξει αρκετές εργασίες πάνω στην προδιαγραφή προγραμματιστικών διεπαφών για την έκφραση του υποθετικού παραλληλισμού σε υψηλό επίπεδο. Και αυτό διότι τα περισσότερα TLS συστήματα στηρίζονται στη χρήση ειδικών εντολών γλώσσας μηχανής για τη δημιουργία και διαχείριση των υποθετικών εργασιών, οι οποίες είτε εισάγονται χειροκίνητα από τον προγραμματιστή σε κάποια χαμηλού επιπέδου αναπαράσταση του προγράμματος, είτε εισάγονται αυτόματα από ειδικό μεταγλωττιστή (π.χ. [Liu 06]).

Ανακεφαλαιώνοντας, πιστεύουμε ότι με απλές σχετικά επεκτάσεις σε ένα HTM σύστημα και με κατάλληλη υποστήριξη από αποδοτικά συστήματα λογισμικού, μπορούμε να διευρύνουμε τα σενάρια χρήσης της μνήμης διενεργειών και να προτείνουμε μια ολοκληρωμένη πλατφόρμα για την αποδοτική και εύκολα προγραμματίσιμη υποθετική εκτέλεση σε επίπεδο νημάτων. Τα τελευταία χρόνια έχουν δημοσιευτεί εργασίες που προσεγγίζουν αυτό ακριβώς το ζήτημα, η κάθε μία με διαφορετική εστίαση ως προς τις επεκτάσεις στο υλικό ή στο λογισμικό [Yoo 08, Porter 09, Guo 08, Baek 07, von Praun 07].

ΚΕΦΑΛΑΙΟ 6

Επίλογος

Οι πολυνηματικές και πολυπύρηνες αρχιτεκτονικές αποτελούν πλέον τον κανόνα στην σχεδίαση επεξεργαστών σε πολλά πεδία εφαρμογών, από μεγάλες υπερυπολογιστικές πλατφόρμες μέχρι προσωπικούς υπολογιστές και ενσωματωμένα συστήματα. Σε αντίθεση με τις παραδοσιακές τεχνικές βελτιστοποίησης που είχαν σαν στόχο την αποδοτικότερη εκτέλεση μιας σειριακής ροής εντολών, οι πολυνηματικές και πολυπύρηνες αρχιτεκτονικές απαιτούν σε πολύ μεγαλύτερο βαθμό τη συνδρομή του λογισμικού για την αξιοποίηση των δυνατοτήτων τους: από το επίπεδο της εφαρμογής και το μοντέλο προγραμματισμού, μέχρι και το επίπεδο του λειτουργικού συστήματος ή άλλων υποστηρικτικών μηχανισμών χρόνου εκτέλεσης. Θέτουν επομένως μια σειρά από σημαντικές προκλήσεις στον προγραμματιστή, όπως είναι ο *εντοπισμός* του παραλληλισμού στις εφαρμογές, η *έκφρασή* του στο πρόγραμμα, η *απεικόνισή* του στην υποκείμενη αρχιτεκτονική, ο *συγχρονισμός* των νημάτων, καθώς και η *αποδοτική διαχείριση των πόρων* της αρχιτεκτονικής.

Στα πλαίσια αυτής της διατριβής ασχοληθήκαμε με ζητήματα που αφορούν την εξαγωγή και απεικόνιση παραλληλισμού σε αρχιτεκτονικές SMT, τον συγχρονισμό σε αρχιτεκτονικές SMT με σκοπό την αποδοτική αξιοποίηση μοιραζόμενων πόρων, και την εξαγωγή μη προφανούς παραλληλισμού από εγγενώς σειριακές εφαρμογές σε πλατφόρμες CMP με μνήμη διενεργειών υλικού. Διερευνήσαμε εναλλακτικά σχήματα νημάτωσης που στοχεύουν κατά κύριο λόγο στη βελτίωση της απόδοσης δύσκολα παραλληλοποιήσιμων εφαρμογών (με βάση παραδοσιακά μοντέλα παραλληλοποίησης) και τα οποία στηρίζονται στη γενικότερη ιδέα της βοηθητικής

νημάτωσης. Στους επεξεργαστές SMT χρησιμοποιούμε τα βοηθητικά νήματα για να αποφορτίσουμε το κύριο νήμα από χρονοβόρες λειτουργίες πρόσβασης στη μνήμη, ενώ στους επεξεργαστές CMP αναθέτουμε σε αυτά πραγματικούς υπολογισμούς του κύριου νήματος τους οποίους εκτελούν με υποθετικό τρόπο αξιοποιώντας τους μηχανισμούς του συστήματος μνήμης διενεργειών.

Στην περίπτωση του SMT, το σχήμα βοηθητικού νηματισμού αποδείχτηκε αποτελεσματικό στην κάλυψη των αστοχιών του κύριου νήματος στην κρυφή μνήμη, ωστόσο ο υψηλός βαθμός διαμοιρασμού πόρων μεταξύ των νημάτων του επεξεργαστή απαιτεί λεπτομερή ενορχήστρωση των λειτουργιών του βοηθητικού νήματος ώστε να εισάγεται η μικρότερη δυνατή παρεμβολή στο κύριο νήμα. Στην κατεύθυνση αυτή μάς απασχόλησε ιδιαίτερα το ζήτημα του αποδοτικού συγχρονισμού ανάμεσα σε νήματα που εκτελούνται στον ίδιο επεξεργαστή SMT, ώστε να ελαχιστοποιούνται οι διενέξεις για μοιραζόμενους πόρους και ταυτόχρονα να εξασφαλίζεται η χαμηλή χρονική καθυστέρηση. Αναπτύξαμε ένα πλαίσιο για την υλοποίηση πρωτογενών λειτουργιών συγχρονισμού προσανατολισμένων στις απαιτήσεις αυτές το οποίο κάνει χρήση προνομιούχων εντολών του επεξεργαστή. Χρησιμοποιώντας τις λειτουργίες του πλαισίου, αναπτύξαμε πιο σύνθετους μηχανισμούς συγχρονισμού τους οποίους αξιολογήσαμε μεταξύ άλλων στα πλαίσια ενός πραγματικού σεναρίου ασύμμετρης εκτέλεσης, δηλαδή αυτού των βοηθητικών νημάτων προφόρτωσης. Στα διάφορα σενάρια αξιολόγησης οι προτεινόμενοι μηχανισμοί παρείχαν τον καλύτερο συμβιβασμό ανάμεσα στην αποδοτική διαχείριση πόρων και τη χαμηλή καθυστέρηση σε σύγκριση με άλλες γνωστές υλοποιήσεις.

Στην περίπτωση των επεξεργαστών CMP μάς απασχόλησε η προοπτική χρήσης ενός προηγμένου μηχανισμού συγχρονισμού, της μνήμης διενεργειών υλικού, με σκοπό την παραλληλοποίηση με "αισιόδοξο" τρόπο μιας περίπτωσης εγγενώς σειριακής εφαρμογής. Στην περίπτωση αυτή τα βοηθητικά νήματα ακολουθούν ένα μοντέλο εκτέλεσης παρόμοιο με αυτό των γνωστών από τη βιβλιογραφία σχημάτων υποθετικού πολυνηματισμού, αναλαμβάνοντας μελλοντικούς υπολογισμούς του κύριου νήματος και εκτελώντας τους με υποθετικό τρόπο. Το υποκείμενο σύστημα μνήμης διενεργειών αναλαμβάνει να αναιρέσει όσους υπολογισμούς βασίστηκαν σε λάθος υποθέσεις. Το προτεινόμενο σχήμα κατάφερε να αποφορτίσει το κύριο νήμα από σημαντικό μέρος των αρχικών υπολογισμών του, οι οποίοι εκτελέστηκαν με επιτυχία από τα βοηθητικά νήματα, επαληθεύοντας μεταξύ άλλων την ύπαρξη αισιόδοξου παραλληλισμού στη συγκεκριμένη εφαρμογή.

Όπως αναφέραμε και στις παραγράφους με τις οποίες κλείσαμε τα τρία προηγούμενα κεφάλαια, τα συμπεράσματα, τις τεχνικές και τις μεθοδολογίες που προέκυψαν από αυτή τη δουλειά θα θέλαμε μελλοντικά να τις αξιολογήσουμε και σε άλλες εφαρμογές, αλλά και να εξετάσουμε το βαθμό γενίκευσής τους στα πλαίσια ολοκληρωμένων συστημάτων (π.χ. σύστημα υποθετικού πολυνηματισμού, αυτοματοποιημένο σύστημα βοηθητικών νημάτων προφόρτωσης,

κ.λπ.). Σε κάθε περίπτωση, αποτελεί ιδιαίτερη πρόκληση για μας το να βοηθήσουμε εφαρμογές με μη προφανή ή περιορισμένο παραλληλισμό να αξιοποιήσουν την ισχύ των σύγχρονων πολυπύρηνων αρχιτεκτονικών με τον καλύτερο τρόπο. Εδώ και μια δεκαετία τουλάχιστον η διεθνής βιβλιογραφία έχει να επιδείξει αρκετές σημαντικές εργασίες σε αυτήν την κατεύθυνση, οι οποίες εστιάζουν στη βελτίωση της απόδοσης τέτοιων "δύσκολων" κατηγοριών εφαρμογών. Οι περισσότερες από αυτές όμως στηρίζουν την επιτυχία τους ή και την εφαρμοσιμότητά τους σε εξιδανικευμένα αρχιτεκτονικά μοντέλα, αγνοώντας σε μεγάλο βαθμό τα χαρακτηριστικά, τις παραμέτρους εκτέλεσης, αλλά και τις προγραμματιστικές διεπαφές των πραγματικών πολυπύρηνων συστημάτων. Στα πλαίσια αυτής της διατριβής επιχειρήσαμε να παρουσιάσουμε μερικές από τις δυσκολίες που κρύβει η εφαρμογή εναλλακτικών μοντέλων πολυνηματισμού στην πράξη, αλλά ταυτόχρονα αναδείξαμε και τη σημαντική δυναμική τους. Έτσι, παρόλα τα πρακτικά εμπόδια, τα αποτελέσματα που πήραμε σε αρκετές περιπτώσεις ήταν ιδιαίτερα ενθαρρυντικά, αποδεικνύοντας ότι οι προτεινόμενες τεχνικές και μεθοδολογίες είναι υποσχόμενες και αξίζουν περαιτέρω διερεύνησης.

Δημοσιεύσεις

Περιοδικά

- *Performance Evaluation of the Sparse Matrix-Vector Multiplication on Modern Architectures.* Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis and Nectarios Koziris. In The Journal of Supercomputing, Volume 50, Issue 1 (October 2009).

- *Exploring the Performance Limits of Simultaneous Multithreading for Memory Intensive Applications.* Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris. In The Journal of Supercomputing, Volume 44 , Issue 1 (April 2008).

Συνέδρια

- *Overlapping Computation and Communication in SMT Clusters with Commodity Interconnects.* Georgios Goumas, Nikos Anastopoulos, Nikolas Ioannou and Nectarios Koziris. In The 2009 IEEE International Conference on Cluster Computing (CLUSTER 2009).

- *Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm.* Konstantinos Nikas, Nikos Anastopoulos, Georgios Goumas and Nectarios Koziris. In The 38th International Conference on Parallel Processing (ICPP 2009).

- *Early Experiences on Accelerating Dijkstra's Algorithm Using Transactional Memory.* Nikos Anastopoulos, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris. In 3rd Workshop on Multithreaded Architectures and Applications (MTAAP 2009).

- *Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors.* Nikos Anastopoulos and Nectarios Koziris. In 2nd Workshop on Multithreaded Architectures and Applications (MTAAP 2008).
- *Understanding the Performance of Sparse Matrix-Vector Multiplication.* Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis and Nectarios Koziris. In 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2008).
- *Exploring the Capacity of a Modern SMT Architecture to Deliver High Scientific Application Performance.* Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris. In The 2006 International Conference on High Performance Computing and Communications (HPCC 2006).
- *Exploring the Performance Limits of Simultaneous Multithreading for Scientific Codes.* Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris. In The 35th International Conference on Parallel Processing (ICPP 2006).
- *Tuning Blocked Array Layouts to Exploit Memory Hierarchy in SMT Architectures.* Evangelia Athanasaki, Kornilios Kourtis, Nikos Anastopoulos and Nectarios Koziris. In 10th Panhellenic Conference on Informatics (PCI 2005).

Βιβλιογραφία

[Adl-Tabatabai 06] Ali-Reza Adl-Tabatabai, Christos Kozyrakis & Bratin Saha. *Unlocking Concurrency: Multicore Programming with Transactional Memory*. ACM Queue, vol. 4, no. 10, pages 24–33, Dec 2006.

[AMD 09] AMD Corporation. *Advanced Synchronization Facility – Proposed Architectural Specification*, Mar 2009. Revision 2.1.

[Athanasaki 04] Evangelia Athanasaki & Nectarios Koziris. *Fast Indexing for Blocked Array Layouts to Improve Multi-Level Cache Locality*. In Proc. of the 8-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT '04), pages 109–119, Madrid, Spain, Feb 2004.

[Bader 05] David Bader & Kamesh Madduri. *Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors*. In Proc. of the 12th International Conference on High-Performance Computing (HiPC '05), pages 465–476, 2005.

[Bader 06] David Bader & Kamesh Madduri. *GTgraph: A Suite of Synthetic Graph Generators*. `http://www.cc.gatech.edu/~kamesh/GTgraph/`, Feb 2006.

[Baek 07] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis & Kunle Olukotun. *The OpenTM Transactional Application Programming Interface*. In Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07), pages 376–387, 2007.

[Bailey 94] David Bailey, Eric Barszcz, John Barton, D.S. Browning, Robert Carter, Leonardo Dagum, Rod Fatoohi, S. Fineberg, Paul Frederickson, T.A. Lasinski, Robert Schreiber, Horst Simon, V. Venkatakrishnan & Sisira Weeratunga. *The NAS Parallel Benchmarks*. Technical Report RNR-94-007, NASA, Mar 1994.

[Barrett 94] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine & Henk van der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods. SIAM, 1994.

[Bellman 58] Richard Bellman. *On a Routing Problem*. Quarterly of Applied Mathematics, vol. 16, pages 87–90, 1958.

[Boggs 04] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll & K.S. Venkatraman. *The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology*. vol. 8, no. 1, Feb 2004.

[Brodal 98] Gerth Brodal, Jesper Traff & Christos Zaroliagis. *A Parallel Priority Queue with Constant Time Operations*. Journal of Parallel and Distributed Computing, vol. 49, pages 4–21, 1998.

[Brodal 99] Gerth Brodal. *Priority Queues on Parallel Machines*. Parallel Computing, vol. 25, no. 8, pages 987–1011, 1999.

[Bugnion 96] Edouard Bugnion, Jennifer Anderson, Todd Mowry, Mendel Rosenblum & Monica Lam. *Compiler-Directed Page Coloring for Multiprocessors*. SIGPLAN Not., vol. 31, no. 9, pages 244–255, 1996.

[Bulpin 04] James Bulpin & Ian Pratt. *Multiprogramming Performance of the Pentium 4 with Hyper-Threading*. In Proc. of the Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD '04), pages 53–62, Munich, Germany, Jun 2004.

[Chakrabarti 04] Deepayan Chakrabarti, Yiping Zhan & Christos Faloutsos. *R-MAT: A Recursive Model for Graph Mining*. In Proc. of the 2004 SIAM International Conference on Data Mining (SDM '04), Apr 2004.

[Chappell 99] Robert Chappell, Jared Stark, Sangwook Kim, Steven Reinhardt & Yale Patt. *Simultaneous Subordinate Microthreading (SSMT)*. In Proc. of the

26th Annual International Symposium on Computer architecture (ISCA '99), numéro 2, pages 186–195, 1999.

[Chen 94] Danny Chen & Xiaobo Hu. *Fast and Efficient Operations on Parallel Priority Queues*. In Proc. of the 5th International Symposium on Algorithms and Computation (ISAAC '94), pages 279–287, 1994.

[Chen 07] Shimin Chen, Phillip Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd Mowry & Chris Wilkerson. *Scheduling Threads for Constructive Cache Sharing on CMPs*. In Proc. of the nineteenth annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07), pages 105–115, 2007.

[Collins 01] Jamison Collins, Hong Wang, Dean Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery & John Shen. *Speculative Precomputation: Long-Range Prefetching of Delinquent Loads*. In Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA '01), pages 14–25, Göteborg, Sweden, Jul 2001.

[Cormen 01] Thomas Cormen, Charles Leiserson, Ronald Rivest & Clifford Stein. Introduction to algorithms. The MIT Press, Cambridge, MA, USA, 2001.

[Curtis-Maury 05] Matthew Curtis-Maury, Tanping Wang, Christos Antonopoulos & Dimitrios Nikolopoulos. *Integrating Multiple Forms of Multithreaded Execution on multi-SMT Systems: A Study with Scientific Applications*. In ICQES, 2005.

[Dial 69] Robert Dial. *Algorithm 360: Shortest Path Forest with Topological Ordering*. Communications of the ACM, vol. 12, pages 632–633, 1969.

[Dijkstra 59] Edsger Dijkstra. *A Note on Two Problems in Connexion with Graphs*. Numerische Mathematik, vol. 1, pages 269–271, 1959.

[Drepper 05] Ulrich Drepper. *Futexes are Tricky*. Dec 2005.

[Edmonds 06] Nick Edmonds, Alex Breuer, Douglas Gregor & Andrew Lumsdaine. *Single-Source Shortest Paths with the Parallel Boost Graph Library*. In 9th DIMACS Implementation Challenge – The Shortest Path Problem, DIMACS Center, Rutgers University, Piscataway, Nov 2006.

[Eggers 97] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm & Dean Tullsen. *Simultaneous Multithreading: A Platform for Next-Generation Processors*. IEEE Micro, vol. 17, no. 5, pages 12–19, 1997.

[Ford 62] Lester Ford & Delbert Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[Frigo 98] Matteo Frigo, Charles Leiserson & Keith Randall. *The Implementation of the Cilk-5 Multithreaded Language*. SIGPLAN Not., vol. 33, no. 5, pages 212–223, 1998.

[gem 08] *Wisconsin Multifacet GEMS Simulator*. `http://www.cs.wisc.edu/gems/`, 2008.

[Goff 91] Gina Goff, Ken Kennedy & Chau-Wen Tseng. *Practical Dependence Testing*. SIGPLAN Not., vol. 26, no. 6, pages 15–29, 1991.

[Goumas 08] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis & Nectarios Koziris. *Understanding the Performance of Sparse Matrix-Vector Multiplication*. In Proc. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '08), pages 283–292, 2008.

[Goumas 09a] Georgios Goumas, Nikos Anastopoulos, Nikolas Ioannou & Nectarios Koziris. *Overlapping Computation and Communication in SMT Clusters with Commodity Interconnects*. In Proc. of the 2009 IEEE International Conference on Cluster Computing (CLUSTER '09), New Orleans, LA, USA, 2009.

[Goumas 09b] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis & Nectarios Koziris. *Performance Evaluation of the Sparse Matrix-Vector Multiplication on Modern Architectures*. Journal of Supercomputing, vol. 50, no. 1, pages 36–77, 2009.

[Gummaraju 05] Jayanth Gummaraju & Mendel Rosenblum. *Stream Programming on General-Purpose Processors*. In Proc. of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '05), pages 343–354, 2005.

[Guo 08] Rui Guo, Hong An, Ruiling Dou, Ming Cong, Yaobin Wang & Qi Li. *LogSPoTM: A Scalable Thread Level Speculation Model Based on*

Transactional Memory. In Proc. of the 13th Asia-Pacific Computer Systems Architecture Conference (ACSAC '08), Aug 2008.

[Hammond 98] Lance Hammond, Mark Willey & Kunle Olukotun. *Data Speculation Support for a Chip Multiprocessor*. SIGOPS Oper. Syst. Rev., vol. 32, no. 5, pages 58–69, 1998.

[Herlihy 93] Maurice Herlihy & J. Eliot B. Moss. *Transactional Memory: Architectural Support for Lock-Free Data Structures*. In Proc. of the 20th Annual International Symposium on Computer Architecture (ISCA '93), pages 289–300. May 1993.

[Hinton 01] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker & Patrice Roussel. *The Microarchitecture of the Pentium 4 Processor*. Feb 2001.

[Hunt 96] Galen Hunt, Maged Michael, Srinivasan Parthasarathy & Michael Scott. *An Efficient Algorithm for Concurrent Priority Queue Heaps*. Information Processing Letters, vol. 60, pages 151–157, 1996.

[Int 01] Intel Corporation. *Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor*, May 2001. Order Number: 248674-002.

[Int 07] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Nov 2007. Order Number: 248966-016.

[Int 09a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A: System Programming Guide, Part 1*, Mar 2009. Order Number: 253668-030US.

[Int 09b] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3B: System Programming Guide, Part 2*, Mar 2009. Order Number: 253669-030US.

[JáJá 92] Joseph JáJá. An introduction to parallel algorithms. 1992.

[Kang 09] Seunghwa Kang & David Bader. *An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs*. In Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09), Feb 2009.

[Kim 02] Dongkeun Kim & Donald Yeung. *Design and Evaluation of Compiler Algorithms for Pre-Execution*. In Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02), pages 159–170, 2002.

[Kim 04a] Dongkeun Kim, Shih-Wei Liao, Perry Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar & John Shen. *Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors*. In Proc. of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO '04), pages 27–38, San Jose, CA, Mar 2004.

[Kim 04b] Dongkeun Kim & Donald Yeung. *A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code*. ACM Trans. Comput. Syst., vol. 22, no. 3, pages 326–379, 2004.

[Krishnan 99] Venkata Krishnan & Josep Torrellas. *A Chip-Multiprocessor Architecture with Speculative Multithreading*. IEEE Trans. Comput., vol. 48, no. 9, pages 866–880, 1999.

[Liu 06] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau & Josep Torrellas. *POSH: A TLS Compiler that Exploits Program Structure*. In Proc. of the 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '06), pages 158–167, 2006.

[Lo 97a] Jack Lo, Susan Eggers, Joel Emer, Henry Levy, Rebecca Stamm & Dean Tullsen. *Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading*. ACM Transactions on Computer Systems, vol. 15, no. 3, pages 322–354, Aug 1997.

[Lo 97b] Jack Lo, Susan Eggers, Henry Levy, Sujay Parekh & Dean Tullsen. *Tuning Compiler Optimizations for Simultaneous Multithreading*. In Proc. of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '97), pages 114–124, Research Triangle Park, NC, Dec 1997.

[Luk 96] Chi-Keung Luk & Todd Mowry. *Compiler-Based Prefetching for Recursive Data Structures*. In Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96), pages 222–233, Boston, MA, Oct 1996.

[Luk 99] Chi-Keung Luk & Todd Mowry. *Automatic Compiler-Inserted Prefetching for Pointer-based Applications*. IEEE Transactions on Computers, vol. 48, no. 2, pages 134–141, Feb 1999.

[Luk 01] Chi-Keung Luk. *Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors*. In Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA '01), pages 40–51, Göteborg, Sweden, Jul 2001.

[Luk 05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi & Kim Hazelwood. *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*. SIGPLAN Not., vol. 40, no. 6, pages 190–200, 2005.

[Madduri 06] Kamesh Madduri, David Bader, Jonathan Berry & Joseph Crobak. *Parallel Shortest Path Algorithms for Solving Large-Scale Instances*. In 9th DIMACS Implementation Challenge – The Shortest Path Problem, DIMACS Center, Rutgers University, Piscataway, Nov 2006.

[Magnusson 02] Peter Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt & Bengt Werner. *Simics: A Full System Simulation Platform*. Computer, vol. 35, no. 2, pages 50–58, Feb 2002.

[Marr 02] Deborah Marr, Frank Binns, David Hill, Glenn Hinton, David Koufaty, J. Alan Miller & Michael Upton. *Hyper-Threading Technology Architecture and Microarchitecture*. Intel Technology Journal, Feb 2002.

[Martin 05] Milo Martin, Daniel Sorin, Bradford Beckmann, Michael Marty, Min Xu, Alaa Alameldeen, Kevin Moore, Mark Hill & David Wood. *Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset*. SIGARCH Computer Architecture News, vol. 33, no. 4, pages 92–99, 2005.

[Meyer 98] Ulrich Meyer & Peter Sanders. *Delta-stepping: A Parallel Single Source Shortest Path Algorithm*. In Proc. of the 6th Annual European Symposium on Algorithms (ESA '98), pages 393–404, 1998.

[Mitchell 99] Nicholas Mitchell, Larry Carter, Jeanne Ferrante & Dean Tullsen. *ILP versus TLP on SMT*. In Proc. of the 1999 ACM/IEEE conference on Supercomputing (SC '99), Nov 1999.

[Mowry 92] Todd Mowry, Monica Lam & Anoop Gupta. *Design and Evaluation of a Compiler Algorithm for Prefetching*. In Proc. of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS '92), pages 62–73, 1992.

[Mowry 98] Todd Mowry. *Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching*. ACM Trans. Comput. Syst., vol. 16, no. 1, pages 55–92, Feb 1998.

[Nethercote 03] Nicholas Nethercote & Julian Seward. *Valgrind: A Program Supervision Framework*. In Proc. of the 3rd Workshop on Runtime Verification (RV '03), Boulder, CO, Jul 2003.

[Ni 08] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal & Xinmin Tian. *Design and Implementation of Transactional Constructs for C/C++*. In Proc. of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08), pages 195–212, 2008.

[Olukotun 96] Kunle Olukotun, Basem Nayfeh, Lance Hammond, Ken Wilson & Kunyung Chang. *The case for a single-chip multiprocessor*. SIGPLAN Not., vol. 31, no. 9, pages 2–11, 1996.

[omn 03] *Omni OpenMP Compiler Project*. Released in the International Conference for High Performance Computing, Networking and Storage (SC '03), `http://phase.hpcc.jp/Omni/home.html`, Nov 2003.

[omp 08] *OpenMP Application Program Interface, version 3.0*. `http://www.openmp.org/mp-documents/spec30.pdf`, 2008.

[opr 09] *OProfile - A System Profiler for Linux*. `http://oprofile.sourceforge.net/`, 2009.

[Papadopoulos 08] Kostas Papadopoulos, Kyriakos Stavrou & Pedro Trancoso. *HelperCoreDB: Exploiting Multicore Technology to Improve Database*

Performance. In Proc. of the 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08), pages 1–11, 2008.

[Patterson 03] David Patterson & John Hennessy. Computer architecture. a quantitative approach. Morgan Kaufmann, 3rd edition, 2003.

[Pinotti 96] Maria Pinotti, Sajal Das & Vincenzo Crupi. *Parallel and Distributed Meldable Priority Queues Based on Binomial Heaps*. In Proc. of the 1996 International Conference on Parallel Processing (ICPP '96), Aug 1996.

[Porter 09] Leo Porter, Bumyong Choi & Dean Tullsen. *Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading*. In Proc. of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09), Los Alamitos, CA, USA, 2009.

[Rangan 04] Ram Rangan, Neil Vachharajani, Manish Vachharajani & David I. August. *Decoupled Software Pipelining with the Synchronization Array*. In Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04), pages 177–188, 2004.

[Renau 05] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss & Josep Torrellas. *Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation*. In Proc. of the 19th annual international conference on Supercomputing (ICS '05), pages 179–188, 2005.

[Roth 01] Amir Roth & Gurindar Sohi. *Speculative Data-Driven Multithreading*. In Proc. of the 7th International Symposium on High Performance Computer Architecture (HPCA '01), pages 37–48, Nuevo Leone, Mexico, Jan 2001.

[Saha 06a] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard Hudson, Chi Cao Minh & Benjamin Hertzberg. *McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime*. In Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06), pages 187–197, 2006.

[Saha 06b] Bratin Saha, Ali-Reza Adl-Tabatabai & Quinn Jacobson. *Architectural Support for Software Transactional Memory*. In Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06), pages 185–196, 2006.

[Scott 07] Michael Scott, Michael Spear, Luke Dalessandro & Virendra Marathe. *Delaunay Triangulation with Transactions and Barriers*. In Proc. of the IEEE 10th International Symposium on Workload Characterization (IISWC '07), 2007.

[Silberschatz 01] Abraham Silberschatz, Henry Korth & S. Sudarshan. Database systems concepts. McGraw-Hill Higher Education, 4th edition, 2001.

[Sohi 95] Gurindar Sohi, Scott Breach & T. N. Vijaykumar. *Multiscalar Processors*. In Proc. of the 22nd Annual International Symposium on Computer architecture (ISCA '95), pages 414–425, 1995.

[Steffan 98] Greggory Steffan & Todd Mowry. *The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization*. In Proc. of the IEEE 4th International Symposium on High Performance Computer Architecture (HPCA '98), pages 2–13, 1998.

[Steffan 00] Greggory Steffan, Christopher Colohan, Antonia Zhai & Todd Mowry. *A Scalable Approach to Thread-Level Speculation*. In Proc. of the 27th annual international symposium on Computer architecture (ISCA '00), pages 1–12, 2000.

[Sundaramoorthy 00] Karthik Sundaramoorthy, Zachary Purser & Eric Rotenberg. *Slipstream Processors: Improving both Performance and Fault Tolerance*. In Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00), pages 257–268, Cambridge, MA, Nov 2000.

[Sutter 05] Herb Sutter & James Larus. *Software and the Concurrency Revolution*. Queue, vol. 3, no. 7, pages 54–62, 2005.

[tbb 08] *Intel Threading Building Blocks*. `http://www.threadingbuildingblocks.org/`, 2008.

[Tian 08] Chen Tian, Min Feng, Vijay Nagarajan & Rajiv Gupta. *Copy or Discard Execution Model for Speculative Parallelization on Multicores*. In Proc. of the 2008 41st IEEE/ACM International Symposium on Microarchitecture (MICRO '08), pages 330–341, 2008.

[Tremblay 08] Mark Tremblay & Shailender Chaudhry. *A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARCÂ® Processor*. In Proc.

of the 2008 IEEE International multi-threaded Solid State Circuits Conference (ISSCC '08), pages 82–83, Feb. 2008.

[Tuck 03] Nathan Tuck & Dean Tullsen. *Initial Observations of the Simultaneous Multithreading Pentium 4 Processor*. In Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03), New Orleans, LA, USA, Sep 2003.

[Tullsen 95] Dean Tullsen, Susan Eggers & Henry Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. In Proc. of the 22nd Annual International Symposium on Computer Architecture (ISCA '95), pages 392–403, Santa Margherita Ligure, Italy, Jun 1995.

[Tullsen 99] Dean Tullsen, Jack Lo, Susan Eggers & Henry Levy. *Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor*. In Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA '99), page 54, Washington, DC, USA, 1999.

[von Praun 07] Christoph von Praun, Luis Ceze & Calin Caşcaval. *Implicit Parallelism with Ordered Transactions*. In Proc. of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '07), pages 79–89, 2007.

[Wang 04] Tanping Wang, Filip Blagojevic & Dimitrios Nikolopoulos. *Runtime Support for Integrating Precomputation and Thread-Level Parallelism on Simultaneous Multithreaded Processors*. In Proc. of the 7th ACM SIGPLAN Workshop on Languages, Compilers, and Runtime Support for Scalable Systems (LCR '04), Houston, TX, USA, Oct 2004.

[Watson 07] Ian Watson, Chris Kirkham & Mikel Lujan. *A Study of a Transactional Parallel Routing Algorithm*. In Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07), pages 388–398, 2007.

[Woo 94] Steven Cameron Woo, Jaswinder Pal Singh & John Hennessy. *The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors*. In Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '94), pages 219–229, 1994.

[Yeager 96] Kenneth Yeager. *The MIPS R10000 Superscalar Microprocessor*. IEEE Micro, vol. 16, no. 2, pages 28–40, 1996.

[Yen 07] Luke Yen, Jayram Bobba, Michael Marty, Kevin Moore, Haris Volos, Mark Hill, Michael Swift & David Wood. *LogTM-SE: Decoupling Hardware Transactional Memory from Caches*. In Proc. of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07), pages 261–272, Feb 2007.

[Yoo 08] Richard Yoo & Hsien-Hsin Lee. *Helper Transactions: Enabling Thread-Level Speculation via A Transactional Memory System*. In Proc. of the 2008 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA '08), Jun 2008.

[Zilles 01] Craig Zilles & Gurindar Sohi. *Execution-Based Prediction Using Speculative Slices*. In Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA '01), pages 2–13, Göteborg, Sweden, Jul 2001.