ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αλγόριθμοι διάσχισης γράφων σε σύγχρονες πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σταύρου Μ. Βώλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2009

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αλγόριθμοι διάσχισης γράφων σε σύγχρονες πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Σταύρου Μ. Βώλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιουλίου 2009

……	……	……
Νεκτάριος Κοζύρης	Κωστής Σαγώνας	Νικόλαος Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2009

……..

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

Περίληψη

Οι σύγχρονες αρχιτεκτονικές έχουν πλέον ξεφύγει από την χρήση ενός πυρήνα λόγω των τεχνολογικών ορίων που υπάρχουν. Στηρίζονται πλέον στην χρήση πολλαπλών πυρήνων με σκοπό την παράλληλη εκτέλεση νημάτων (threads), ώστε να αυξηθεί η απόδοση του συστήματος (ένα νήμα σε κάθε πυρήνα). Περισσότερα του ενός νήματος, μπορούν να εκτελεστούν ταυτόχρονα σε κάποιο πυρήνα του επεξεργαστή, αν αυτός υποστηρίζει SMT (simultaneous multi-threading) αρχιτεκτονική.

Η κοινή μνήμη, που υπάρχει στους πολυπύρηνους επεξεργαστές, εισάγει περιορισμούς στην παράλληλη εκτέλεση μιας πολυνηματικής εφαρμογής. Η ύπαρξη πολλαπλών νημάτων, τα οποία ζητούν αρκετές φορές πρόσβαση στις ίδιες διευθύνσεις μνήμης, απαιτεί κάποιο είδος συγχρονισμού των νημάτων, έτσι ώστε να διατηρηθεί η συνέπεια της εφαρμογής και τα αποτελέσματα της να είναι τα ίδια με την σειριακή εκτέλεση της. Ενώ το fine-grain locking δίνει καλή επίδοση, αλλά είναι επιρρεπής σε λάθη π.χ deadlocks, ενώ το coarse-grain locking είναι εύκολο, αλλά δίνει χαμηλή επίδοση.

Ένα νέο προγραμματιστικό μοντέλο, το οποίο έχει κάνει την εμφάνιση του τα τελευταία χρόνια, είναι η Transactional Memory και έχει σκοπό την επίλυση των προβλημάτων, που εισάγει ο πολυνηματικός προγραμματισμός. Σκοπός της διπλωματικής εργασίας είναι η μελέτη των σύγχρονων αρχιτεκτονικών, που εισάγουν την Transactional Memory και επίσης την αξιολόγηση εκτέλεσης αλγορίθμων διάσχισης γράφων σε σύστημα που υποστηρίζει Hardware Transactional Memory. Η σειριακή φύση των αλγορίθμων διάσχισης γράφων περιορίζει την βελτίωση της παράλληλης υλοποίησης με την χρήση mutex-locking, ενώ αρκετές φορές η υλοποίηση αυτή είναι χειρότερη από την σειριακή. Συγκεκριμένα μελετάται η δυνατότητα βελτίωσης, που δίνει ο πολυνηματικός προγραμματισμός, χρησιμοποιώντας Transactional Memory στον αλγόριθμο διάσχισης γράφου κατά πλάτος (Breadth First Search).

Εξετάζουμε την συμπεριφορά διαφορετικών υλοποιήσεων του αλγορίθμου BFS χρησιμοποιώντας τον εξομοιωτή GEMS v.2.1 (General Execution-Driven Multiprocessor Simulator) του Πανεπιστημίου του Wisconsin σε συνδυασμό με τον εξομοιωτή Simics v.3.0.31. Ο εξομοιωτής Simics παρέχει την εξομοίωση πολυπύρηνου επεξεργαστή αρχιτεκτονικής SPARC, ο οποίος τρέχει το λειτουργικό σύστημα Solaris. Η μονάδα Ruby του GEMS παρέχει λεπτομερής εξομοίωση της ιεραρχίας μνήμης του συστήματος και φορτώνεται στον Simics. Hardware Transactional Memory υποστηρίζεται από τον GEMS μέσω του συστήματος LogTM

Λέξεις Κλειδιά

Αλγόριθμος Breadth First Search, Transactional Memory, LogTM, TCC, GEMS, Simics

Abstract

The modern processors do not depend on single-core architectures, because of the technological limits. The most of them depend on multi-core architectures, so they can run one single thread on every core of the processor due to the need of performance. Using simultaneous multi-threading architecture on the cores, the potential of running also multiple threads on every single core of the multicore processor exists.

The shared memory multiprocessors causes new boundaries at the parallel execution of a multithreaded application. The existence of multiple threads, which access the most times, the same memory address, demands the synchronization of the threads, so the results of the parallel execution will be the same with those of the serial. This synchronization using mutex-locking limits the performance of the parallel application, since the threads serialize if they cannot continue their execution because some other threads have already acquired the lock. Using fine-grained locking we can have good scalability, but it is hard to avoid mistakes such as deadlocks, while coarse-grained locking is easy to be used, but its performance is limited.

Transactional Memory is a new programming model, whose purpose is to bring solution to the problems of using multithreaded programming. Purpose of this project is to study the Hardware Transactional Memory and also evaluate the execution of graph search algorithms on a system which supports Hardware Transactional Memory. The serial character of those algorithms limits the performance of the parallel implementation which uses mutex-locking, while the most times its performance is worse than serial's. Particularly we study the potential of improvement using the Transactional Memory model on Breadth First Search.

We evaluate the performance of various implementations of BFS algorithm through full-system execution-driven simulation, using the Wisconsin GEMS toolset v.2.1 in conjunction with the Simics v.3.0.31 simulator. Simics provides functional simulation of a SPARC chip multiprocessor system that boots Solaris. The GEMS Ruby module provides detailed memory system simulation and it's loaded on Simics. Hardware TM is supported in GEMS through the LogTM subsystem.

Keywords

Breadth First Search algorithm, Transactional Memory, LogTM, TCC, GEMS, Simics

Ευχαριστίες

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου, υπό την επίβλεψη του Αναπληρωτή Καθηγητή Νεκτάριου Κοζύρη.

Θα ήθελα κατ' αρχήν να ευχαριστήσω τον καθηγητή μου Νεκτάριο Κοζύρη, τόσο για την εποπτεία του κατά την εκπόνηση της εργασίας μου, όσο για τη συμβουλή του στην διαμόρφωση μου ως μηχανικού μέσα από τις διδασκαλίες του και τη γενικότερη στάση του.

Τις ευχαριστίες μου θα ήθελα να εκφράσω επίσης σε όλα τα μέλη του εργαστηρίου και ιδιαίτερα στους Υποψήφιο Διδάκτορα Νίκο Αναστόπουλο και τον ερευνητή Διδάκτορα Κωστή Νίκα για τη συνεχή καθοδήγηση, που μου προσέφεραν προς ολοκλήρωση της διπλωματικής μου εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω όλους όσους βρίσκονται στον οικογενειακό και φιλικό μου περιβάλλον και κυρίως τους γονείς μου, των οποίων η κάθε είδους υποστήριξη συνετέλεσε στην επιτυχή ολοκλήρωση των προπτυχιακών μου σπουδών κι επίσης τον αδερφό μου Χάρη για την βοήθεια, που μου προσέφερε στην διπλωματική μου εργασία και γενικότερα σε θέματα των προπτυχιακών μου σπουδών.

Περιεχόμενα

Εισαγωγή

0.1 Γενικά

Η τρομακτική τεχνολογική ανάπτυξη που έχει σημειωθεί τις τελευταίες δεκαετίες στον τομέα των VLSI είχε ως αποτέλεσμα την αύξηση της πυκνότητας ολοκλήρωσης σε μια ψηφίδα. Η δυνατότητα αυτή για χρησιμοποίηση περισσότερων τρανζίστορ είχε ως αποτέλεσμα την υλοποίηση σωληνωτών και υπερβαθμωτών αρχιτεκτονικών, οι οποίες μείωσαν τον κύκλο ρολογιού, ενώ αύξησαν την εκμετάλλευση του παραλληλισμού σε επίπεδο εντολής (ILP). Ωστόσο οι επεξεργαστές αυτοί έχουν γίνει αρκετά πολύπλοκοι, οπότε οι βελτιώσεις της απόδοσης με την αύξηση της πολυπλοκότητας άρχισαν να εξαντλούνται. Ταυτόχρονα οι βελτιώσεις της απόδοσης με την συνεχή αύξηση της συχνότητας του ρολογιού επίσης εξαντλούνται λόγω του κόστους της κατανάλωση ενέργειας και επίσης της αύξησης της θερμότητας, που προκαλείται.

Λόγω της ανικανότητας αύξησης της απόδοσης των μονοπύρηνων επεξεργαστών σε συνδυασμό με την ανάγκη περισσότερης επεξεργαστικής ισχύς, η παρουσία πολυπύρηνων αρχιτεκτονικών έκανε την εμφάνιση της. Εκμεταλλευόμενοι την αυξημένη πυκνότητα ολοκλήρωσης σε μια ψηφίδα, όλο και περισσότεροι πυρήνες ολοκληρώνονται σε κάθε επεξεργαστή. Με αυτόν τον τρόπο δίνεται η δυνατότητα εκμετάλλευσης του παραλληλισμού σε επίπεδο νήματος (TLP) στο επίπεδο λογισμικού, ενώ στο επίπεδο αρχιτεκτονικής η ταχύτητα επικοινωνίας μεταξύ πυρήνων του επεξεργαστή και επεξεργαστή – κρυφής μνήμης είναι αρκετά πιο γρήγορη, λόγω της συνύπαρξης τους στην ίδια ψηφίδα.

Πλέον το περισσότερο βάρος για αύξηση της απόδοσης των εφαρμογών σε πολυνηματικά συστήματα πέφτει στους προγραμματιστές και όχι στους αρχιτέκτονες και κατασκευαστές των συστημάτων αυτών. Ενώ οι καταναλωτές πετυχαίνουν ψηλότερη απόδοση τρέχοντας πολλαπλές εφαρμογές σε ένα πολυπύρηνο σύστημα καταναλώνοντας ταυτόχρονα λιγότερη ενέργεια, οι προγραμματιστές, οι οποίοι επιθυμούν να εκμεταλλευτούν τα πλεονεκτήματα της πολυπύρηνης αρχιτεκτονικής, οφείλουν να ειδικευτούν στον πολυνηματικό προγραμματισμό. Η αυξημένη απόδοση που παρέχει ένας πολυπύρηνος επεξεργαστής σε μια εφαρμογή είναι άρρηκτα συνδεδεμένη με την χρήση πολλαπλών νημάτων (threads) και την ικανότητα των νημάτων να εκτελεστούν παράλληλα και ανεξάρτητα στους πολλαπλούς πυρήνες του επεξεργαστή. Η δυσκολία στον πολυνηματικό προγραμματισμό έγκειται στα θέματα ταυτοχρονισμού, τα οποία οι προγραμματιστές καλούνται να αντιμετωπίσουν αναλώνοντας έτσι αρκετό χρόνο εις βάρος άλλων σημαντικών προβλημάτων που υπάρχουν σε επίπεδο τεχνολογίας λογισμικού.

Όπως αναφέρθηκε προηγουμένως, η βελτίωση μιας παράλληλης εφαρμογής στηρίζεται στον ταυτοχρονισμό (concurrency). Ωστόσο ο ταυτοχρονισμός εισάγει ένα μεγάλο πρόβλημα, αυτό του συγχρονισμού των πολλαπλών νημάτων, που τρέχουν παράλληλα. Στους πυρήνες του επεξεργαστή. Υπάρχει μεγάλη δυσκολία να επιτευχθεί ο συγχρονισμός αυτός, λόγω της ταυτόχρονης πρόσβασης τους σε κοινή μνήμη. Ως εκ τούτου η σωστή χρονική εκτέλεση των εντολών νημάτων σε ψηλή απόδοση φαντάζει ακόμη πιο δύσκολο επίτευγμα. Συγκεκριμένα αλγόριθμοι διάσχισης γράφων όπως είναι και ο Breadth First Search, είναι κατά φύση τους σειριακοί και υπάρχει μεγάλη δυσκολία παραλληλοποίησης τους λόγω θεμάτων συγχρονισμού. Η ανάγκη συγχρονισμού σε αρκετά σημεία των αλγορίθμων καθιστά αρκετές φορές την παράλληλη υλοποίηση χειρότερη από την σειριακή.

Είναι εμφανές, ότι υπάρχει η ανάγκη ύπαρξης ενός προγραμματιστικού μοντέλου, το οποίο να είναι εύκολο να χρησιμοποιηθεί από την πλειοψηφία των προγραμματιστών. Ένας τέτοιο προγραμματιστικό μοντέλο είναι η Transactional Memory. Προγραμματιστικά μοντέλα δοσοληψιών (transactional programming models) μπορούν να υλοποιηθούν σε επίπεδο λογισμικού (software) χρησιμοποιώντας software transactional memory (STM), σε επίπεδο δομικού υλικού (hardware) χρησιμοποιώντας hardware transactional memory ή συνδυασμός των δύο (Hybrid TM).

Ο προγραμματισμός με δοσοληψίες (tm-based) παρουσιάζεται σαν υποσχόμενη εναλλακτική λύση του βασισμένου σε κλειδώματα μοντέλου (lock-based). Στο lock-based μοντέλο οι προγραμματιστές βασίζονται στην χρήση κλειδωμάτων (locks) και προϋποθέσεων (conditions) για να παρεμποδίσουν ταυτόχρονη πρόσβαση των νημάτων στα ίδια μοιραζόμενα δεδομένα. Κλειδώματα, τα οποία γίνονται αρκετά νωρίς (coarse-grained locking), είναι εύκολα σε προγραμματισμό, αλλά δεν προσφέρουν αρκετή επίδοση, ενώ τα επιλεγμένα κλειδώματα (fine-grained locking) είναι δύσκολα σε προγραμματισμό κι αν δεν χρησιμοποιούνται σωστά υπάρχει ενδεχόμενο εμφάνισης αδιεξόδων. Ο προγραμματισμός με δοσοληψίες δίνει την ευκαιρία στα νήματα που εκτελούνται παράλληλα να εκτελούν αυτές τις δοσοληψίες ταυτόχρονα σε περίπτωση, που δεν υπάρχει κάποια σύγκρουση μεταξύ τους (λόγω κοινής πρόσβασης στα ίδια μοιραζόμενα δεδομένα), ενώ σε περίπτωση σύγκρουσης οι δοσοληψίες που συγκρούονται μεταξύ τους (εκτός από μια) απορρίπτονται και ξαναεκτελούνται μετά από τυχαίο χρόνο επιτυγχάνοντας έτσι την σειριοποίηση των νημάτων εκεί που επιβάλλεται. Ο τρόπος με τον οποίο γίνεται η απόρριψη των δοσοληψιών έγκειται στον διαχειριστή συγκρούσεων (conflict manager). Η δομή του conflict manager επηρεάζει σημαντικά την εκτέλεση των νημάτων, αφού αυτός καθορίζει έμμεσα την σειρά εκτέλεσης των δοσοληψιών όταν υπάρχει κάποια σύγκρουση. Στην αναφορά [23] γίνεται σαφές πως οι παράμετροι ενός συστήματος TM επηρεάζουν την συμπεριφορά διαφορετικών benchmarks

Η TM φαντάζει σαν μια λύση του προβλήματος που δημιουργήθηκε με την χρήση πολυπύρηνων αρχιτεκτονικών, ενώ υπάρχει μεγάλη αισιοδοξία από τους κατασκευαστές της, ότι θα αποτελέσει το μέλλον του παράλληλου προγραμματισμού, αφού είναι σε θέση να δώσει στους προγραμματιστές ένα αποδοτικό προγραμματιστικό μοντέλο, κάτι το οποίο χρειάζεται, έτσι ώστε να συνεχίσει η επαναστατική πρόοδος των πολυπύρηνων επεξεργαστών

0.2 Σκοπός

Σκοπός της παρούσας διπλωματικής εργασίας είναι η μελέτη διαφόρων αλγορίθμων, που εκτελούνται σε γράφους και παρουσιάζουν κατά κύριο λόγο δυσκολία στην παραλληλοποίηση τους. Στα πλαίσια της εργασίας αυτής διάφοροι αλγόριθμοι θα υλοποιηθούν παράλληλα με

χρήση locks χρησιμοποιώντας τις βιβλιοθήκες των Posix Threads, ενώ στην συνέχεια θα χρησιμοποιηθεί TM σε διάφορα κομμάτια των αλγορίθμων. Η επίδοση των υλοποιήσεων των αλγορίθμων θα μελετηθεί στον εξομοιωτή Wisconsin GEMS/Virtutech SIMICS στον οποίο θα χρησιμοποιηθεί το LogTM του Wisconsin (HTM). Θα γίνει σύγκριση των αποτελεσμάτων των παράλληλων υλοποιήσεων με την σειριακή για να δείξουμε ότι η TM μπορεί να δώσει κάποιο είδος βελτίωσης. Συγκεκριμένα θα μελετηθεί ο αλγόριθμος διάσχισης γράφων Breadth First Search. Στην τελευταία ενότητα της διπλωματικής εργασίας μελετούνται διάφορες πολιτικές εντοπισμού και διαχείρισης συγκρούσεων και έκδοσης δεδομένων πάνω στην υλοποίηση του BFS, που έχουμε υλοποιήσει.

0.3 Παρεμφερής Εργασία

Γενικά οι αλγόριθμοι που εκτελούνται σε γράφους έχουν την δυνατότητα να ευνοηθούν με την χρήση της TM. Υπάρχουν αρκετοί γράφοι, οι οποίοι είναι δύσκολο να χωριστούν σε ανεξάρτητα μέρη τα οποία θα αναλάβει κάθε νήμα, οπότε υπάρχει ανάγκη συγχρονισμού, αφού είναι πιθανόν δύο ή περισσότερα νήματα να επισκεφθούν την ίδια κορυφή.

Οι Kang και Bader [1] χρησιμοποιούν TM για παραλληλοποίηση του αλγόριθμου κατασκευής ελάχιστου συνδετικού δέντρου (Minimum Spanning Tree). Κάθε νήμα επιλέγει μια αρχική κορυφή και ακολούθως αναπτύσσει τοπικά ένα MST χρησιμοποιώντας τον αλγόριθμο του Prim, διατηρώντας τις ακμές του δέντρου σε έναν σωρό. Όταν MST δύο νημάτων συναντηθούν σε μια κορυφή, τότε γίνεται συνένωση των σωρών κι ακολούθως το ένα νήμα συνεχίζει με το νέο MST, ενώ το άλλο νήμα ξεκινάει με μια νέα κορυφή. Στην υλοποίηση αυτή χρησιμοποιείται TM στην προσθήκη των κόμβων του MST, κατά την οποία αφαιρείται μια ακμή από τον τοπικό σωρό του νήματος και αν η νέα κορυφή δεν ανήκει σε κανένα MST, τότε γίνεται ενημέρωση του σωρού με τις νέες ακμές, ενώ αν ανήκει σε άλλο MST τότε γίνεται η συνένωση των MST. Η υλοποίηση εξετάστηκε σε STM δίνοντας ικανοποιητική επιτάχυνση. Ωστόσο οι λειτουργικές δαπάνες (overhead) του STM ήταν αρκετές ώστε να καθιστούν εν τέλει την παραλληλοποίηση μη επικερδές, οπότε κατέληξαν στην ανάγκη υποστήριξης HTM ώστε να υπάρχει κάποιο κέρδος.

Οι Scott et al. [2] χρησιμοποιούν TM για παραλληλοποίηση του αλγόριθμου τριγωνισμού του Delaunay. Ο αλγόριθμος εξετάζει και ανανεώνει συνεχώς τρίγωνα μεταξύ τριάδων που προέρχονται από μεγάλο σύνολο δεδομένων, οπότε η TM χρησιμοποιείται για την αποφυγή των αναγκαίων fine-grain locks, που χρειάζονται για τον συγχρονισμό των νημάτων. Η υλοποίηση του αλγόριθμου εξετάστηκε σε STM δίνοντας ικανοποιητική επιτάχυνση.

Οι Watson et al. [3] επίσης χρησιμοποιούν TM για παραλληλοποίηση του αλγόριθμου δρομολόγησης του Lee. Ο αλγόριθμος περιλαμβάνει υπολογισμό ανεξαρτήτων μονοπατιών από ένα σύνολο σημείων. Χρησιμοποιώντας δοσοληψία για τον υπολογισμό αυτό, δίνεται η ευκαιρία σε αρκετά μονοπάτια να υπολογιστούν παράλληλα χωρίς συγχρονισμό.

Τέλος οι Koziris et al. [4] χρησιμοποιούν TM για παραλληλοποίηση του αλγορίθμου του Dijkstra, ο οποίος είναι δύσκολο να παραλληλοποιηθεί λόγω της εξαγωγής κόμβων από κοινή ουρά σε κάθε επανάληψη. Χρησιμοποιώντας TM τα νήματα μπορούν να προσπελάσουν ταυτόχρονα μοιραζόμενα δεδομένα αποφεύγοντας αρκετές φορές τον συγχρονισμό που απαιτείται. Η υλοποίηση έγινε στο LogTM [5] και έδειξε κάποια βελτίωση, η οποία αυξήθηκε με την βοήθεια βοηθητικών νημάτων (helper threads) τα οποία βοηθούν το κύριο νήμα.

0.4 Οργάνωση – Αντικείμενο της εργασίας

Στο πρώτο κεφάλαιο παρουσιάζονται βασικές αρχές της Transactional Memory κι ακολούθως περιγράφονται διάφορες αρχιτεκτονικές συστημάτων HTM, όπως του LogTM, το οποίο έχει σχεδιαστεί από το πανεπιστήμιο του Wisconsin και του TCC, το οποίο έχει σχεδιαστεί από το πανεπιστήμιο του Stanford.

Στο δεύτερο κεφάλαιο παρουσιάζεται ο αλγόριθμος BFS και στην συνέχεια περιγράφονται οι παράλληλες υλοποιήσεις που έχουν εξεταστεί με coarse-grained και fine-grained locking καθώς και Transactional Memory

Στο τρίτο κεφάλαιο περιγράφεται το σύστημα στο οποίο έγινε η εξομοίωση και επίσης η παρουσίαση των αποτελεσμάτων της εκτέλεσης των υλοποιήσεων του BFS

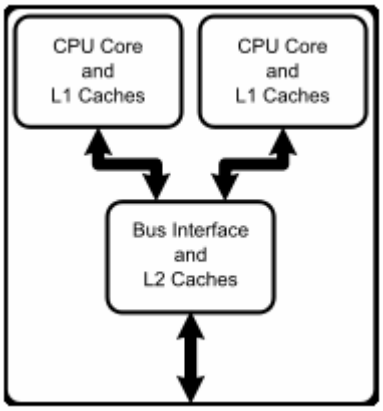
Στο τέταρτο κεφάλαιο περιγράφουμε τις διαφορετικές πολιτικές εντοπισμού και διαχείρισης συγκρούσεων και επίσης της διαχείρισης εκδόσεων δεδομένων. Μελετούμε επίσης την επίδραση διαφορετικών πολιτικών πάνω στην υλοποίηση του BFS, που έχει δώσει καλύτερη απόδοση στο κεφάλαιο 3.

Στο πέμπτο και τελευταίο κεφάλαιο γίνεται ένας μικρός επίλογος και αναφορά σε μελλοντική εργασία σαν επέκταση της παρούσας διπλωματικής εργασίας.

Κεφάλαιο 1
Πολυπύρηνες Αρχιτεκτονικές

1.1 Γενικά

Οι πολυπύρηνες αρχιτεκτονικές (chip multicore processors architectures) έχουν κάνει την εμφάνιση τους και πλέον αποτελούν μια από τις διαδεδομένες παράλληλες αρχιτεκτονικές. Στις αρχιτεκτονικές αυτές φυσικά η κύρια μνήμη είναι κοινή. Πολλαπλοί πυρήνες βρίσκονται στην ίδια ψηφίδα, ώστε οι ταχύτητες επικοινωνίας μεταξύ τους να είναι μεγάλες. Βασικά θέματα των αρχιτεκτονικών σχετίζονται με τον τρόπο συνδεσμολογίας των πυρήνων, ποια επίπεδα της κρυφής μνήμης θα είναι κοινά και ποια ιδιωτικά και επίσης τον τρόπο επικοινωνίας μεταξύ των πυρήνων.



Σχήμα 1.1: Διπύρηνος επεξεργαστής με ιδιωτική L1 και μοιραζόμενη L2

Μια από τις διαδεδομένες αρχιτεκτονικές είναι αυτή που φαίνεται στο σχήμα 1.1. Ο κάθε πυρήνας έχει ιδιωτική L1 κρυφή μνήμη και κοινή L2 κρυφή μνήμη, οπότε και απαιτείται η ύπαρξη ενός διαδρόμου μεταξύ των κρυφών μνημών L1 και L2. Χρησιμοποιώντας πρωτόκολλο παρακολούθησης διαδρόμου (bus-snooping), εξασφαλίζεται η συνεκτικότητα της κρυφής μνήμης και γενικότερα της μνήμης (cache και memory coherence). Μια επίσης διαδεδομένη αρχιτεκτονική είναι η μη ύπαρξη κοινής κρυφής μνήμης, δηλαδή η L2 να είναι επίσης ιδιωτική. Στην περίπτωση αυτή χρησιμοποιείται crossbar switch για να συνδέσει τους πυρήνες με την κύρια μνήμη. Στο παράρτημα γίνεται λεπτομερής αναφορά στους όρους cache coherence και bus-snooping

1.2 Transactional Memory

1.2.1 Γενικά

Η πρόοδος των πολυπύρηνων επεξεργαστών κοινής μνήμης έχει δημιουργήσει μια μεγάλη ευκαιρία για την εκμετάλλευση του παραλληλισμού σε επίπεδο νήματος. Όπως έχει προαναφερθεί στην εισαγωγή, οι περισσότερες παράλληλες εφαρμογές χρειάζονται συγχρονισμό για την πρόσβαση στην κοινή μνήμη. Ο συγχρονισμός μπορεί να επιτευχθεί με την χρήση mutex-locking. Η χρήση του fine-grained locking είναι πολύπλοκη και μπορεί να δημιουργήσει deadlocks, ενώ η χρήση του coarse-grained locking είναι απλή, αλλά αποτυγχάνει να δώσει βελτίωση στην απόδοση της εφαρμογής. Η Transactional Memory είναι ένα προγραμματιστικό μοντέλο, το οποίο έχει προταθεί από τους αρχιτέκτονες υπολογιστών για να λύσει τα προβλήματα συγχρονισμού, που υπάρχουν στον πολυνηματικό προγραμματισμό. Ο προγραμματιστής χρησιμοποιεί δοσοληψίες (transactions) αντί κλειδώματα (locks) χωρίς να τον ενδιαφέρει ο τρόπος εκτέλεσης του κρίσιμου κομματιού που ορίζεται από την δοσοληψία. Ο τρόπος εκτέλεσης των δοσοληψίες εναποτίθεται στην υλοποίηση της TM, που υποστηρίζει ο επεξεργαστής. Οι περισσότερες υλοποιήσεις TM επιτρέπουν στα νήματα να εκτελούνται παράλληλα, υποθέτοντας ότι δεν υπάρχουν συγκρούσεις δεδομένων. Σε περίπτωση που δεν υπάρχει κάποια σύγκρουση, τότε υπάρχει κέρδος στην απόδοση της εφαρμογής, αφού στην προκειμένη περίπτωση τα νήματα θα έπρεπε να περιμένουν την αποδέσμευση του κλειδώματος για να μπορέσουν να εκτελεστούν. Αντίθετα, σε περίπτωση που ανιχνευθεί κάποια σύγκρουση, τότε η δοσοληψία ακυρώνεται και το σύστημα επανέρχεται στην κατάσταση στην οποία βρισκόταν πριν ξεκινήσει η δοσοληψία αυτή.

Τα συστήματα TM προσφέρουν στις δοσοληψίες ατομικότητα (atomicity) και απομόνωση (isolation). Μια επιπλέον ιδιότητα που πρέπει να πληρεί μια δοσοληψία είναι αυτή της συνέπειας (consistency), της οποία η διασφάλιση είναι ευθύνη του προγραμματιστή. Με τον όρο ατομικότητα εννοούμε ότι η δοσοληψία πρέπει να εκτελεστεί ολόκληρη ή να μην έχει καθόλου επιδράσεις. Με τον όρο απομόνωση εννοούμε ότι οι ενδιάμεσες καταστάσεις μιας δοσοληψίας δεν είναι φανερή στις υπόλοιπες δοσοληψίες, που εκτελούνται ταυτόχρονα. Για να είναι ικανό ένα σύστημα TM να παρέχει αυτές τις ιδιότητες χρειάζεται η ύπαρξη μιας διαχείρισης των εκδόσεων δεδομένων (data version management), μηχανισμού εντοπισμού συγκρούσεων (conflict detection) και διαχειριστή συγκρούσεων (conflict resolution). Στον πίνακα 1.1 παρουσιάζονται διάφορες υλοποιήσεις συστημάτων TM, στο οποίο διαχωρίζονται αναλόγως της διαχείρισης των δεδομένων και εντοπισμού των συγκρούσεων

		Διαχείριση Έκδοσης Δεδομένων (Version Management)	
		Lazy (οκνηρή)	**Eager (πρόθυμη)**
Εντοπισμός Συγκρούσεων (Conflict Detection)	**Lazy (οκνηρός)**	OCC DBMSs Stanford TCC	
	Eager (πρόθυμος)	MIT LTM Intel/Brown VTM (on cache conflicts)	CCC DBMSs MIT UTM LogTM

Πίνακας 1.1: Ταξινομία συστημάτων TM

Ιδανικά ένα σύστημα TM πρέπει να χρησιμοποιεί πρόθυμο μηχανισμό εντοπισμού συγκρούσεων και πρόθυμη διαχείριση των εκδόσεων δεδομένων. Αφ' ενός η πρόθυμη διαχείριση εκδόσεων δεδομένων αποθηκεύοντας τις τιμές στην θέση τους κατά την διάρκεια της δοσοληψίας επιτυγχάνει πιο γρήγορα commits και πιο αργά aborts, κάτι το οποίο είναι θεμιτό, αφού commits συμβαίνουν πιο συχνά από aborts. Κι αφ' ετέρου ο πρόθυμος μηχανισμός εντοπισμού συγκρούσεων βρίσκει νωρίς τις συγκρούσεις, οπότε μειώνει την περιττή δουλειά που έχουν πραγματοποιήσει οι συγκρουόμενες δοσοληψίες.

1.2.2 Υλοποιήσεις Transactional Memory

Τα δύο κύρια πρότυπα υλοποιήσεων TM είναι βασισμένα σε hardware και software. Εκτός από τα δύο πρότυπα αυτά, υπάρχουν τα υβριδικά τους. Η υβριδική TM (HyTM) υποστηρίζει HTM και χρησιμοποιεί software δοσοληψίες, όταν αδυνατεί να εκτελέσει τις hardware δοσοληψίες. Αντίθετα το υβριδικό της STM, Hardware-assisted STM (HaSTM) συνδυάζει STM με αρχιτεκτονική υποστήριξη με σκοπό την επιτάχυνση κομματιών της STM υλοποίησης.

1.2.2.1 Hardware Transactional Memory

Οι πρώτες HTM σχεδιάσεις [20] ήταν βασισμένες στην τροποποίηση του πρωτοκόλλου της συνέπειας της κρυφής μνήμης (cache consistency protocol) και επίσης πρόσθεταν ένα μικρό αριθμό εντολών για την υποστήριξη των δοσοληψιών. Οι σχεδιάσεις αυτές κρατούσαν την υποθετική κατάσταση σε κάποιον buffer μέχρι η δοσοληψία να ολοκληρωθεί επιτυχώς ή ανεπιτυχώς. Οι δύο αυτές τροποποιήσεις είναι αρκετές για την βασική υλοποίηση του HTM. Παρακάτω παρουσιάζονται αναλυτικά οι τροποποιήσεις που χρειάζονται να γίνουν βάση του API.

Προσθήκη εντολών: Χρησιμοποιούνται οι παρακάτω εντολές. Κάποιες από αυτές δεν χρησιμοποιούνται αναλόγως της υλοποίησης.

- STR: εκκίνηση δοσοληψίας
- ETR: τέλος δοσοληψίας
- TLD: δοσοληπτική ανάγωση
- TST: δοσοληπτική εγγραφή
- ABR: χρησιμοποιείται για ακύρωση μιας δοσοληψίας (victim transaction) υπό τον έλεγχο της εφαρμογής
- VLD: χρησιμοποιείται κατά την διάρκεια της δοσοληψίας υπό τον έλεγχο της εφαρμογής για έλεγχο συγκρούσεων

Έκδοση δεδομένων και συγκρούσεις: Αλλαγές στην κρυφή μνήμη και χρήση buffers. Τα συστήματα HTM ανιχνεύουν τις συγκρούσεις σε επίπεδο γραμμής ή λέξης. Η υποστήριξη της δοσοληπτικής συνεκτικότητας και συνέπειας βασίζεται στην επέκταση ήδη υπαρχόντων πρωτοκόλλων συνεκτικότητας της κρυφής μνήμης όπως το MESI. Παρακάτω γίνεται μια μικρή αναφορά στους μηχανισμούς που πρέπει να υπάρχουν σε ένα σύστημα HTM:

- **Data version** management: διαχειρίζεται την ταυτόχρονη αποθήκευση των νέων τιμών των δεδομένων, οι οποίες γίνονται ορατές σε περίπτωση που η δοσοληψία εκτελεστεί επιτυχώς (commit) και των παλιών τιμών των δεδομένων, οι οποίες παραμένουν σε περίπτωση που η δοσοληψία διακοπεί / ακυρωθεί / απορριφθεί

(abort). Μια από αυτές τις τιμές αποθηκεύεται στην φυσική διεύθυνση μνήμης, ενώ η άλλη αποθηκεύεται προσωρινά σε κάποιο άλλο σημείο. Ένα σύστημα TM μπορεί να χρησιμοποιήσει πρόθυμη διαχείριση (eager version management) και να αποθηκεύσει την νέα τιμή στην φυσική διεύθυνση της μνήμης ή να χρησιμοποιήσει οκνηρή διαχείριση (lazy version management) και να διατηρήσει προσωρινά (μέχρι να κάνει commit η δοσοληψία) την παλιά τιμή στην θέση μνήμης της.

- **Conflict Detection**: Ο μηχανισμός εντοπισμού συγκρούσεων εντοπίζει επικάλυψη του συνόλου εγγραφής της μιας δοσοληψίας με το σύνολο εγγραφής ή ανάγνωσης των δοσοληψιών που εκτελούνται ταυτόχρονα. Υπάρχει πρόθυμη διαχείριση των συγκρούσεων, όπου ο εντοπισμός των συγκρούσεων γίνεται άμεσα κι επίσης οκνηρή διαχείριση, όπου ο εντοπισμός των συγκρούσεων γίνεται, όταν η δοσοληψία κάνει commit.
- **Conflict Resolution:** Ο διαχειριστής συγκρούσεων είναι αυτός που καθορίζει ποιες δοσοληψίες θα ακυρωθούν και ποια θα συνεχίσει, όταν κάποια σύγκρουση εντοπιστεί.

Στο κεφάλαιο 4 γίνεται αναφορά στις διάφορες πολιτικές των παραπάνω μηχανισμών και επίσης μελετάται η επίδραση τους στην συμπεριφορά μιας tm-based εφαρμογής.

1.2.2.2 Software Transactional Memory

Ενώ οι υλοποιήσεις HTM χρησιμοποιούν buffers και την κρυφή μνήμη δεδομένων για να κρατήσουν τις δύο καταστάσεις δεδομένων, η STM πρέπει να παρέχει διαφορετικές όψεις του σωρού σε κάθε νήμα κατά την διάρκεια εκτέλεσης δοσοληψιών. Επίσης πρέπει να παρέχει όπως και η HTM έναν μηχανισμό για εντοπισμό και διαχείριση των συγκρούσεων

Διαχειριστής δοσοληπτικής κατάστασης (transactional state): Ο μηχανισμός για τις ταυτόχρονες δοσοληψίες υπάρχει, ώστε αυτές να κρατάνε τις δικές τους όψεις του σωρού. Ο μηχανισμός αυτός επιτρέπει σε κάθε δοσοληψία να βλέπει τις δικές τις εγγραφές καθώς συνεχίζει να εκτελείται και επίσης επιτρέπει να απορριφθούν αυτές οι αλλαγές της μνήμης εάν η δοσοληψία ολοκληρωθεί ανεπιτυχώς. Μια διαφοροποίηση μεταξύ των υλοποιήσεων STM είναι ο τρόπος οργάνωσης των δεδομένων στην μνήμη.

Μια προσέγγιση διαχωρίζει τα δεδομένα της δοσοληψίας από τα κανονικά δεδομένα, χρησιμοποιώντας μια ξεχωριστή μορφή μνήμης για τα δοσοληπτικά αντικείμενα. Ο εντοπισμός των δεδομένων δοσοληψίας γίνεται με την χρήση της επικεφαλίδας του αντικειμένου για την εύρεση των δοσοληψιών που προσπαθούν να διαβάσουν το αντικείμενο αυτό. Δύο συναρτήσεις υλοποιούνται για το άνοιγμα της επικεφαλίδας μιας δοσοληψίας. Μια, η οποία χρησιμοποιείται για σκοπούς ανάγνωσης και ακόμη μια για σκοπούς εγγραφής. Και οι δύο επιστρέφουν ένα αντίγραφο του σκελετού του αντικειμένου, το οποίο θα χρησιμοποιήσουν ανάλογα οι δοσοληψίες είτε για ενημέρωση (αντίγραφο σκιά) είτε για ανάγνωση. Οι συναρτήσεις αυτές ονομάζονται συνήθως OpenForReading και OpenForWriting αντίστοιχα. Με αυτόν τον τρόπο γίνεται διαχωρισμός των αναγνώσεων και εγγραφών του σκελετού ενός αντικειμένου, ώστε να επιτρέπεται ανάγνωση του σκελετού από πολλαπλές δοσοληψίες.

Μια δεύτερη προσέγγιση είναι η χρησιμοποίηση μετά-δεδομένων χωρίς να γίνεται κάποιος διαχωρισμός των δεδομένων όπως στην προηγούμενη προσέγγιση. Διατηρούνται τα μετά-δεδομένα που χρησιμοποιεί η STM σε ξεχωριστές δομές. Η διεύθυνση μιας λέξης αντιστοιχεί

με μια λέξη TMW (transactional metadata word). Χρησιμοποιούνται οι συναρτήσεις OpenForReading και OpenForWriting για να ανοίξουν τις λέξεις TMW, ώστε να βρεθεί η διεύθυνση μνήμης της οποίας θα γίνει ανάγνωση ή εγγραφή αντίστοιχα. Οι δύο συναρτήσεις αυτές όπως και στην πρώτη προσέγγιση κατασκευάζουν τα σύνολα ανάγνωσης και εγγραφής μιας δοσοληψίας. Επίσης υπάρχουν οι συναρτήσεις STMRead και STMWrite με σκοπό την ανάγνωση ή εγγραφή για απευθείας αναγνώσεις ή εγγραφές (κανονικά δεδομένα).

Ανεξαρτήτως ποια προσέγγιση ακολουθείται υπάρχει κι εδώ η οκνηρή και πρόθυμη διαχείριση έκδοσης δεδομένων. Στην οκνηρή έκδοση δεδομένων κάθε δοσοληψία διατηρεί ένα αντίγραφο σκιά (με την τιμή της εγγραφής που έχει πραγματοποιηθεί) για κάθε λέξη που τροποποιεί. Αντίθετα η πρόθυμη έκδοση δεδομένων ενημερώνει τον σωρό. Ωστόσο πρέπει να διατηρηθεί ένα Log το οποίο διατηρεί τις παλιές τιμές, ώστε να μπορεί να επιστρέψει το σύστημα στην κατάσταση που βρισκόταν πριν την έναρξη της δοσοληψίας που ακυρώνεται.

Εντοπισμός και διαχείριση συγκρούσεων: Οι μηχανισμοί εντοπισμού και διαχείρισης των συγκρούσεων είναι σαφώς πιο πολύπλοκοι από αυτούς που υπάρχουν στην HTM.

Μια προσέγγιση είναι η χρήση κλειδώματος σε δύο φάσεις (two phase locking). Μια δοσοληψία δεσμεύει κλειδώματα πάνω στις μεταβλητές που έχει γράψει και τις αποδεσμεύει μόνο όταν κάνει commit. Αυτό όμως έχει χαμηλή επίδοση στους πολυεπεξεργαστές, αφού δημιουργεί μεγάλο συναγωνισμό στην ιεραρχία μνήμης.

Μια δεύτερη προσέγγιση είναι η χρήση μιας nonblocking ατομικής πολλαπλής λέξης ενημέρωσης για την επιτυχή ολοκλήρωση μιας δοσοληψίας. Γίνεται έλεγχος ότι δεν έγινε ενημέρωση στο σύνολο ανάγνωσης κι ακολούθως ενημερώνει τις επικεφαλίδες των αντικειμένων που βρίσκονται στο σύνολο εγγραφής της δοσοληψίας για να δημοσιοποιήσει τα αντίγραφα σκιά. Το σύνολο ανάγνωσης ελέγχεται χρησιμοποιώντας εντολές ανάγνωσης μνήμης, οπότε η προσέγγιση αυτή είναι πιο αποδοτική από την προηγούμενη.

1.3 LogTM: Log – based Transactional Memory[1]

Το σύστημα LogTM [5] κατασκευάζει έναν συμβατικό πολυεπεξεργαστή με κοινή μνήμη. Κάθε επεξεργαστής έχει δύο ή και περισσότερα επίπεδα τοπικής κρυφής μνήμης, τα οποία είναι συνεκτικά (coherent) μεταξύ τους με επέκταση του πρωτοκόλλου καταλόγου MOESI[2]. Στην ενότητα αυτή παρουσιάζεται η πρόθυμη διαχείριση εκδόσεων δεδομένων και η πρόθυμη διαχείριση των συγκρούσεων, που υποστηρίζει το σύστημα LogTM. Στο κεφάλαιο 5 εξετάζεται λεπτομερώς η πολιτική διαχείριση συγκρούσεων, που παρέχει το σύστημα LogTM

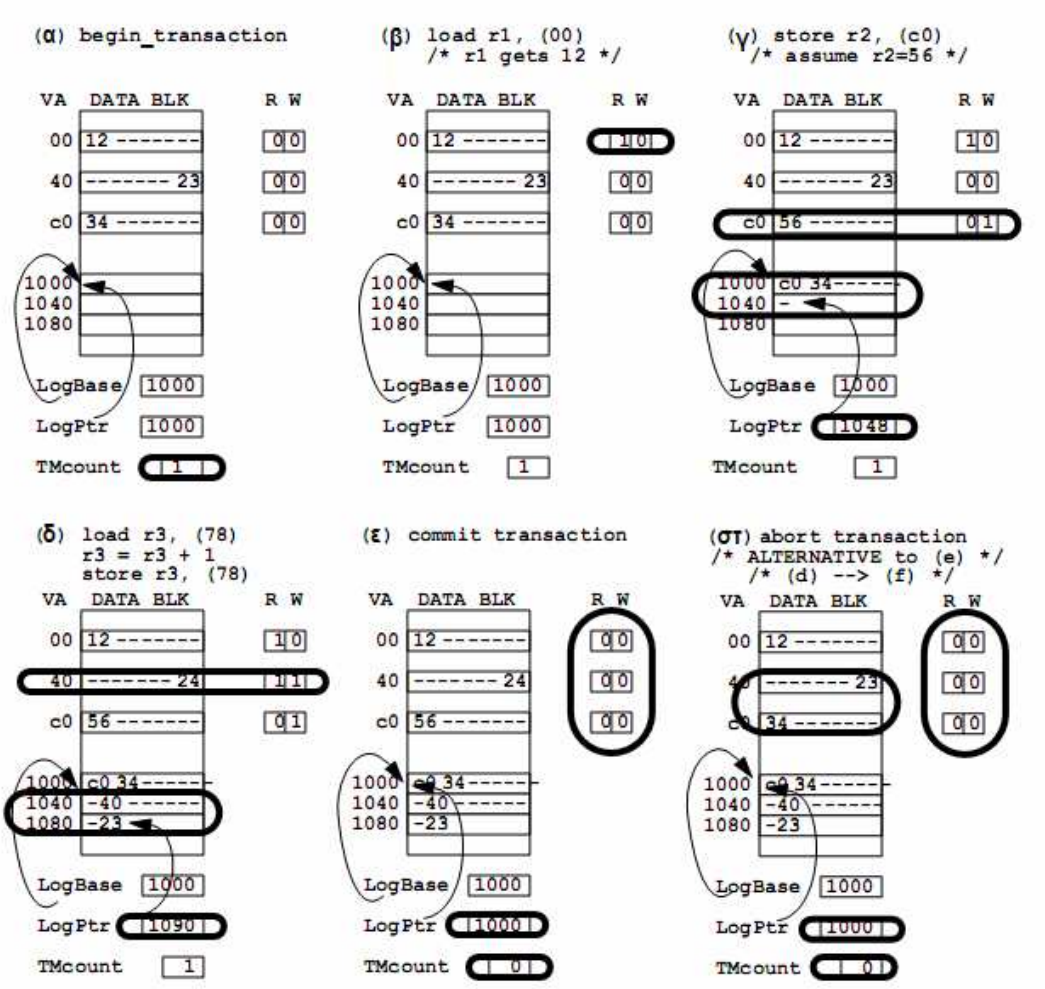
1.3.1 Version Management (διαχείριση εκδόσεων δεδομένων)

Στο σύστημα LogTM κατά την δημιουργία των νημάτων, κάθε νήμα δεσμεύει εικονική μνήμη για την δημιουργία του log, το οποίο βρίσκεται στην κρυφή μνήμη του επεξεργαστή. Σε αυτό το log το σύστημα LogTM αποθηκεύει τις παλιές τιμές των διευθύνσεων μνήμης που τροποποιεί το νήμα. Συγκεκριμένα κατά την διάρκεια μιας αποθήκευσης σε μια δοσοληψία, το LogTM προσθέτει στο log του νήματος την εικονική μνήμη του αποθηκευμένου μπλοκ και την παλιά

[1] Η παρούσα ενότητα στηρίζεται στην αναφορά [5] της βιβλιογραφίας.
[2] Στο παράρτημα περιγράφεται λεπτομερώς

τιμή του μπλοκ αυτού. Για να μειωθούν πλεονάζοντες εγγραφές στο log, το LogTM σημειώνει κάθε μπλοκ, το οποίο βρίσκεται στην κρυφή μνήμη με ένα ψηφίο εγγραφής (W – bit), το οποίο δείχνει κατά πόσο το μπλοκ έχει ήδη γραφτεί στο log. Από την στιγμή που το log βρίσκεται στην κρυφή μνήμη και επίσης οι περισσότερες δοσοληψίες γράφουν σε λίγα μπλοκ, οι εγγραφές στο log είναι επιτυχίες στην κρυφή μνήμη κι επομένως είναι γρήγορες, οπότε δεν υπάρχει μεγάλο overhead. Επίσης για να μειωθεί ο ανταγωνισμός στην L1 υπάρχει σαν επιπρόσθετο hardware ένας μικρός log buffer στον οποίο οι εγγραφές στο log του νήματος μπορούν να περιμένουν μέχρι αυτός να γεμίσει ή η δοσοληψία να πετύχει ή να αποτύχει. Σε περίπτωση που η δοσοληψία πετύχει οι εγγραφές στον log buffer δεν λαμβάνονται υπ' όψη, ενώ σε περίπτωση που η δοσοληψία αποτύχει πραγματοποιούνται.

Το πλεονέκτημα της πρόθυμης διαχείρισης των εκδόσεων είναι τα γρήγορα commits, αφού ο επεξεργαστής το μόνο που χρειάζεται είναι να καθαρίσει τα W-bits και να επαναφέρει τον δείκτη του log του νήματος για να αγνοήσει το log της δοσοληψίας. Ωστόσο τα aborts γίνονται πιο αργά αφού το LogTM πρέπει να επαναφέρει τα τροποποιημένα μπλοκ με τις αρχικές τους τιμές διαβάζοντας όλο το log του νήματος από το τέλος μέχρι την αρχή, ώστε να διατηρηθεί η σωστή σειρά επαναφοράς, αφού υπάρχει περίπτωση πολλαπλής ύπαρξης του ίδιου μπλοκ στο log του νήματος. Στο σχήμα 1.2 παρουσιάζεται η εκτέλεση της δοσοληψίας T η οποία έχει δύο διαφορετικά τέλη. Στο σχήμα 1.2ε κάνει commit, ενώ στο σχήμα 1.2στ κάνει abort.



Σχήμα 1.2: Εκτέλεση δοσοληψίας Τα

Η δοσοληψία Τ είναι η ακόλουθη:

```
begin_transaction
        load r1, (00)
        store r2, (c0)
        load  r3, (78)
        r3 = r3 + 1
        store  r3, (78)
commit_transaction
```

Στάδια εκτέλεσης δοσοληψίας

α) Η έναρξη δοσοληψίας αυξάνει το πεδίο TMcount κατά ένα. Θεωρούμε ότι το log του νήματος ξεκινάει στην εικονική διεύθυνση 1000 και είναι άδειο αφού οι δείκτες LogPtr και LogBase ισούνται. Κάθε μπλοκ έχει ψηφία εγγραφής και ανάγνωσης (R,W)

β) Η εντολή αυτή κάνει ανάγνωση από την εικονική διεύθυνση 00 θέτοντας το ψηφίο ανάγνωσης της του μπλοκ στην τιμή 1

γ) Η εντολή αυτή αποθηκεύει στην εικονική διεύθυνση c0 την τιμή του καταχωρητή R2, ενώ η παλιά τιμή γράφεται στο log του νήματος. Θέτοντας το W-bit του μπλοκ στην τιμή 1, το σύστημα δείχνει ότι το μπλοκ έχει τροποποιηθεί

δ) Εδώ υπάρχει αρχικά ανάγνωση της εικονικής διεύθυνσης 78 κι ακολούθως εγγραφή σε αυτήν οπότε τα R,W – bits θέτονται στην τιμή 1, ενώ η παλιά τιμή γράφεται στο log του νήματος.

ε) Εδώ η δοσοληψία κάνει commit οπότε μειώνεται το πεδίο TMcount κι επομένως ο δείκτης LogPtr γίνεται ίσος με τον δείκτη της βάσης και τα R and W bits θέτονται στην τιμή 0

στ) Εδώ ελέω σύγκρουσης η δοσοληψίας κάνει abort, οπότε οι παλιές τιμές επαναφέρονται και ο δείκτης LogPtr γίνεται ίσος με τον δείκτη της βάσης και τα R and W bits θέτονται στην τιμή 0

1.3.2 Conflict detection (εντοπισμός των συγκρούσεων)

Ο πρόθυμος εντοπισμός των συγκρούσεων γίνεται σε αρκετά στάδια. Ο αιτών πυρήνας που προσπαθεί να εντοπίσει σύγκρουση στέλνει στον κατάλογο αίτημα κι ακολούθως ο κατάλογος απαντάει και προωθεί το αίτημα αυτό σε ένα ή περισσότερους πυρήνες. Κάθε πυρήνας που πρέπει να απαντήσει εξετάζει την τοπική κατάσταση του μπλοκ, που έχει σχέση με το αίτημα για να εντοπίσει την ύπαρξη σύγκρουσης. Ο πυρήνας αυτός στέλνει ACK, όταν δεν υπάρχει σύγκρουση και NACK αν υπάρχει σύγκρουση στον αιτών πυρήνα, ο οποίος αντιμετωπίζει τις όποιες συγκρούσεις έχουν εντοπιστεί.

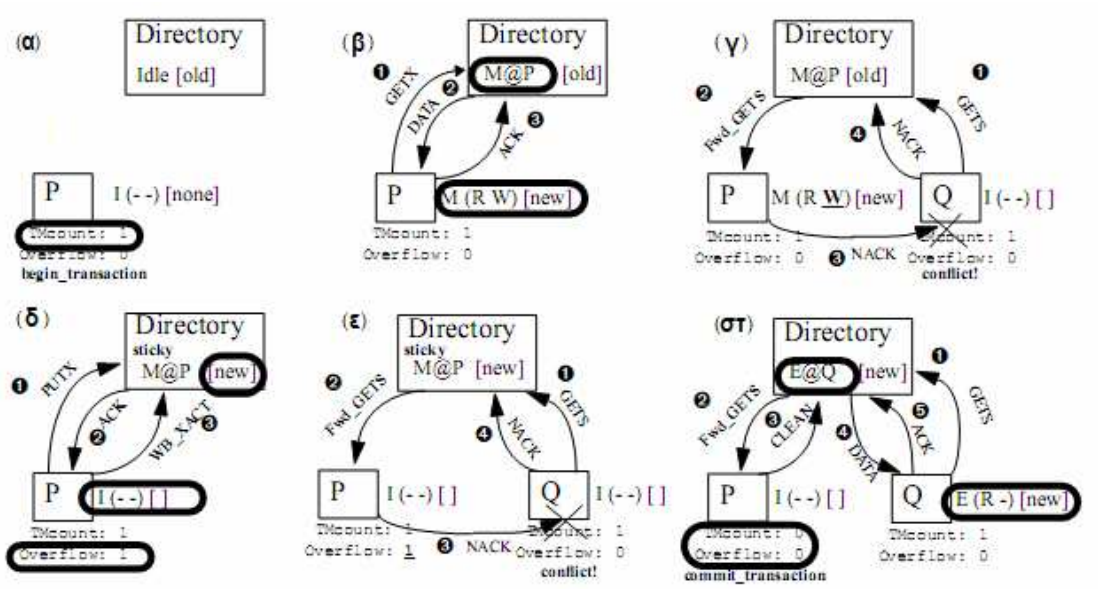
Όπως προαναφέρθηκε σε κάθε μπλοκ της κρυφής μνήμης έχουν προστεθεί δύο επιπλέον bit R, W. Για να εξασφαλιστεί ότι το πρωτόκολλο καταλόγου θα προωθήσει όλες τις πιθανές συγκρούσεις στον ενδιαφερόμενο πυρήνα, το LogTM ακολουθεί κάποια συγκεκριμένη στρατηγική για τα ψηφία R,W. Το R-bit θέτεται στην τιμή 1 μόνο για έγκυρα μπλοκ (καταστάσεις M(odified), O(wned), E(xclusive), S(hared) του πρωτοκόλλου MOESI) κι αυτό όταν υπάρχει ανάγνωση του μπλοκ κατά την διάρκεια μιας δοσοληψίας. Το W-bit θέτεται στην

τιμή 1 μόνο για τα μπλοκ που βρίσκονται σε κατάσταση M κι αυτό όταν υπάρξει κάποια εγγραφή κατά την διάρκεια μιας δοσοληψίας.

Το LogTM προσθέτοντας ένα ψηφίο υπερχείλισης σε κάθε πυρήνα, έχει την δυνατότητα να εντοπίζει τις συγκρούσεις ακόμη και στην περίπτωση που οι δοσοληψίες γεμίσουν την κρυφή μνήμη, δηλαδή έχουμε αντικατάσταση κάποιου μπλοκ μνήμης, που βρίσκεται σε κατάσταση δοσοληψίας. Θέτοντας αυτό το ψηφίο στην τιμή 1, το πρωτόκολλο του καταλόγου συνεχίζει να στέλνει αιτήματα στον πυρήνα αυτό, οπότε ο πυρήνας αυτός έχει την δυνατότητα να ελέγξει την ύπαρξη σύγκρουσης και να στείλει NACK στην περίπτωση αυτή. Συγκεκριμένα η συμπεριφορά αντικατάστασης ενός μπλοκ δοσοληψίας B, εξαρτάται από την κατάσταση στην οποία βρίσκεται το B στην κρυφή μνήμη του επεξεργαστή P, ο οποίος πρόκειται να αντικαταστήσει το B:

- **M:** Ο επεξεργαστής P γράφει στην κύρια μνήμη το B και αλλάζει την κατάσταση του καταλόγου για το μπλοκ B σε μια νέα κατάσταση sticky-M@P. Όταν ο επεξεργαστής Q αιτηθεί το B, ο κατάλογος προωθεί το αίτημα στον επεξεργαστή P, ο οποίος δεν έχει το μπλοκ B στην κρυφή του μνήμη, αλλά προωθεί σύγκρουση, αφού το ψηφίο υπερχείλισης έχει την τιμή 1
- **S:** Ο επεξεργαστής P αντικαθιστά το B, παραμένοντας στην λίστα των sharers που βρίσκεται στον κατάλογο. Όταν ο Q ζητήσει το B για εγγραφή (exclusively) τότε ο κατάλογος στέλνει σήμα στους sharers για να μεταβιβάσουν το B στην κατάσταση I. Ο B βλέποντας το ψηφίο υπερχείλισης αντιλαμβάνεται την σύγκρουση και στέλνει NACK στον επεξεργαστή Q.
- **O, E:** Ο επεξεργαστής P γράφει στην κύρια μνήμη το B και προσθέτει τον εαυτό του στην λίστα των sharers που βρίσκεται στον κατάλογο κι ακολούθως αντικαθιστά το μπλοκ B. Όταν ο Q ζητήσει το B για εγγραφή η συμπεριφορά είναι ακριβώς η ίδια με αυτήν της κατάστασης S

Στο σχήμα 1.3 παρουσιάζεται ένα παράδειγμα όπου μια δοσοληψία του επεξεργαστή P γράφει το μπλοκ B χωρίς να το έχει αρχικά στην κρυφή μνήμη.

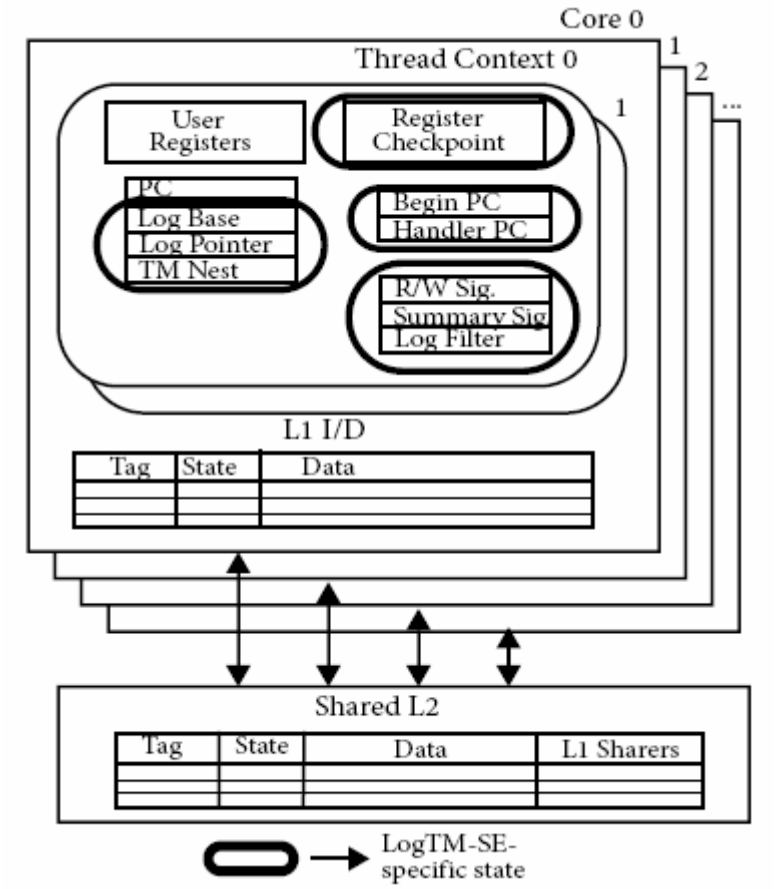


Σχήμα 1.3: Εντοπισμός σύγκρουσης στην cache (α-γ) κι εκτός cache (δ-στ)

Στο 1.3γ υπάρχει σύγκρουση, αφού ο επεξεργαστής Q ζητά ανάγνωση του μπλοκ B το οποίο όμως βρίσκεται σε κατάσταση M στον επεξεργαστή P, οπότε ανιχνεύεται η σύγκρουση. Στο 1.3δ λόγω υπερχείλισης το μπλοκ B αντικαθίσταται οπότε ακολουθείται η πολιτική M που περιγράφηκε παραπάνω και τέλος στο 1.3στ η δοσοληψία γίνεται commit οπότε ο επεξεργαστής Q ξαναδοκιμάζει να διαβάσει το B, οπότε και επιτυγχάνει

1.3.3 LogTM-SE (signature edition) [3]

Στα συστήματα HTM υπάρχει η επιθυμία να αφαιρεθεί η διαχείριση έκδοσης δεδομένων και διαχείρισης εντοπισμού συγκρούσεων από την L1 cache. Το σύστημα LogTM καταφέρνει να αφαιρέσει την διαχείριση έκδοσης δεδομένων με την χρήση του log buffer, ο οποίος βρίσκεται στην εικονική μνήμη του νήματος. Ωστόσο αποτυγχάνει να αφαιρέσει την διαχείριση εντοπισμού συγκρούσεων, αφού τα W/R bits βρίσκονται στην L1. Για να μπορεί ένα σύστημα να ανιχνεύει τις συγκρούσεις στο επίπεδο αίτησης συνεκτικότητας (coherence request), η LogTM-SE χρησιμοποιεί υπογραφές, οι οποίες βρίσκονται αποθηκευμένες στην εικονική μνήμη του νήματος, για την διατήρηση των συνόλων εγγραφής και ανάγνωσης μιας δοσοληψίας. Στο σχήμα 1.4 απεικονίζεται το LogTM-SE, όπου είναι φανερό ότι η κρυφή μνήμη δεν διατηρεί επιπλέον hardware για σκοπούς υποστήριξης της Hardware Transactional Memory



Σχήμα 1.4: LogTM-SE

[3] Η παρούσα ενότητα στηρίζεται στην αναφορά [13

Υπογραφές (signatures)
Μια N bit υπογραφή διατηρεί τα $\log_2 N$ λιγότερο σημαντικά ψηφία της διεύθυνσης ενός block. Για παράδειγμα 2 Kbit υπογραφή διατηρεί τα 11 λιγότερο σημαντικά ψηφία της διεύθυνσης. Όταν ένας πυρήνας αποτυγχάνει να διαβάσει (γράψει) ένα block A δημιουργεί μια αίτηση συνεκτικότητας GETS(A) (GETM(A)). Ένας πυρήνας, ο οποίος λαμβάνει μια τέτοια αίτηση ελέγχει τις υπογραφές εγγραφής (ανάγνωσης και εγγραφής) για να εντοπίσει πιθανή σύγκρουση. Αν το σύστημα χρησιμοποιεί 2 Kbit υπογραφής, ο έλεγχος ισότητας γίνεται στα 11 τελευταία ψηφία της διεύθυνσης μνήμης του block A. Ο μη πλήρης έλεγχος μπορεί να οδηγήσει σε εσφαλμένους εντοπισμούς αλλά δεν θα χάσει ποτέ κάποια σύγκρουση που υπάρχει.

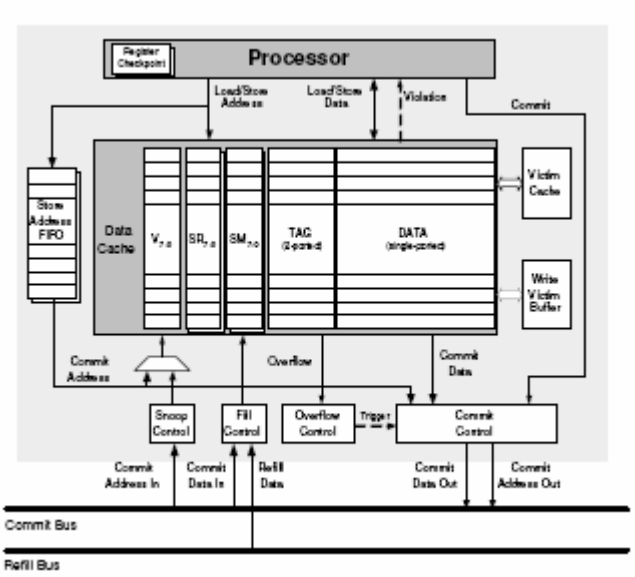
Με την απεξάρτηση από την κρυφή μνήμη και αποθήκευση της κατάστασης της δοσοληψίας στην εικονική μνήμη του νήματος, υπάρχει η δυνατότητα context switching, virtual memory paging. Η υποστήριξη του paging πραγματοποιείται με ανανέωση των υπογραφών χρησιμοποιώντας την νέα φυσική διεύθυνση. Επίσης η ικανότητα αποθήκευσης κι ακολούθως επαναφοράς των υπογραφών επιτρέπει ύπαρξη φωλιασμένων δοσοληψιών (nested transactions). Τέλος υπάρχει πλέον και η δυνατότητα thread suspension / migration. Για να επιτευχθεί η αναστολή ενός νήματος χρησιμοποιείται μια summary signature, η οποία είναι η ένωση των υπογραφών όλων των ανασταλμένων νημάτων. Κοιτάζοντας την summary signature εκτός από την signature του παρόντος νήματος, υπάρχει η δυνατότητα εντοπισμού των συγκρούσεων και με τις δοσοληψίες που δεν τρέχουν σε κάποιον πυρήνα.

1.4 TCC: Transactional Coherence and Consistency[4]

Το TCC [16,17] είναι ένα μοντέλο πολυπύρηνων επεξεργαστών με κοινή μνήμη, η οποία χρησιμοποιεί σαν βασική μονάδα παράλληλης εκτέλεσης, συγχρονισμού, συνεκτικότητας και συνέπειας δοσοληψίες. Να υπενθυμίσουμε ότι το TCC χρησιμοποιεί οκνηρή διαχείριση δεδομένων. Αυτό σημαίνει, ότι τα δεδομένα που γράφονται κατά την διάρκεια μιας δοσοληψίας γράφονται στην σωστή θέση μνήμης, όταν η δοσοληψία ολοκληρωθεί (commit). Επίσης το TCC χρησιμοποιεί οκνηρή διαχείριση συγκρούσεων. Αυτό σημαίνει ότι οι συγκρούσεις μεταξύ δύο δοσοληψιών εντοπίζονται όταν κάποια από αυτές κάνει commit.

Κάθε επεξεργαστής έχει ιδιωτική L1 κρυφή μνήμη, ενώ μοιράζεται την L2 κρυφή μνήμη μαζί με τους υπόλοιπους επεξεργαστές. Οι επεξεργαστές και η L2 είναι συνδεδεμένοι μεταξύ τους μέσω του transaction-commit διαδρόμου κι επίσης του refill διαδρόμου. Ο transaction-commit διάδρομος χρησιμοποιείται για να τοποθετήσει τις νέες τιμές των δεδομένων, που έχουν τροποποιηθεί κατά την διάρκεια της δοσοληψίας στις σωστές θέσεις μνήμης μετά το πέρας της δοσοληψίας. Ο refill διάδρομος είναι ο διάδρομος που χρησιμοποιείται για να μεταφερθούν τα refill δεδομένα (δεδομένα μιας γραμμής κρυφής μνήμης) από την L2 στον επεξεργαστή. Στο σχήμα 1.4 παρουσιάζεται η οργάνωση της κρυφής μνήμης δεδομένων για δοσοληπτική συνεκτικότητα (transactional coherence)

[4] Η παρούσα ενότητα στηρίζεται στις αναφορές [16] και [17] της βιβλιογραφίας.



Σχήμα 1.4: Οργάνωση κρυφής μνήμης δεδομένων

1.4.1 Transactional Buffering

Ο κάθε επεξεργαστής έχει έναν buffer στον οποίο αποθηκεύει τις διευθύνσεις μνήμης και τα δεδομένα για κάθε αποθήκευση (transactional store – write set), που υπάρχει μέσα σε μια δοσοληψία, μέχρι αυτή να ολοκληρωθεί (commit) ή να απορριφθεί (abort). Ταυτόχρονα ο επεξεργαστής ανιχνεύει όλες τις διευθύνσεις μνήμης, οι οποίες έχουν φορτωθεί (transaction load – read set) για να μπορέσει να ανιχνεύσει εξαρτήσεις και συγκρούσεις δεδομένων, όταν κάποια δοσοληψία κάνει commit.

Οι πληροφορίες (κάποια bits) που δείχνουν ποια είναι τα σύνολα ανάγνωσης κι εγγραφής (read-set και write-set) αποθηκεύονται στην L1, διότι έχει αρκετή χωρητικότητα και επίσης παρέχει μεγάλη ταχύτητα στον εντοπισμό των συνόλων αυτών. Είναι γνωστό, ότι η κρυφή μνήμη αποθηκεύει τα δεδομένα σε γραμμές. Ωστόσο ο εντοπισμός των συνόλων μπορεί να γίνει σε επίπεδο γραμμής ή λέξης. Το σχήμα 1.4 δείχνει την οργάνωση της L1 κρυφής μνήμης για την περίπτωση που ο εντοπισμός γίνεται σε επίπεδο λέξης.

Για να γίνει ο εντοπισμός των συνόλων σε επίπεδο λέξης, η γραμμή της κρυφή μνήμης περιλαμβάνει επιπλέον δύο ψηφία, το SM (speculatively-modified) και SR (speculatively-read) για κάθε λέξη. Για παράδειγμα ένας 32-bit επεξεργαστής με 32 byte μέγεθος γραμμής της κρυφής μνήμης 16 bits χρειάζονται επιπλέον για κάθε γραμμή.Το SM δηλώνει, ότι η λέξη έχει τροποποιηθεί από transactional store κατά την διάρκεια της δοσοληψίας, που εκτελείται την παρούσα χρονική στιγμή. Παρομοίως το SR δηλώνει, ότι η λέξη έχει διαβαστεί κατά την διάρκεια της δοσοληψίας, που εκτελείται την παρούσα χρονική στιγμή. Το ψηφίο SR μιας λέξης γίνεται 1, μόνο εάν το αντίστοιχο SM είναι 0. Με αυτόν τον τρόπο είναι εφικτό να υλοποιηθεί η αλλαγή ονομασίας που χρειάζεται για να αποφευχθούν οι κίνδυνοι δεδομένων που οφείλονται σε WAW (write after write) και WAR (write after read) εξαρτήσεις

Ο εντοπισμός των συνόλων σε επίπεδο γραμμής λειτουργεί με παρόμοιο τρόπο, αλλά περιλαμβάνει ένα ψηφίο SM κι ένα ψηφίο SR για κάθε γραμμή της κρυφής μνήμης. Αυτό εξοικονομεί χώρο στην ψηφίδα, αλλά εισάγει προβλήματα, αφού δεν μπορεί να επιλύσει τις

συγκρούσεις μεταξύ μιας τροποποιήμενης γραμμής και του commit που γίνεται στην ίδια γραμμή. Στην επόμενη ενότητα ακολουθεί λεπτομερής περιγραφή εύρεσης των συνόλων εγγραφής και ανάγνωσης.

1.4.2 Transactional Commit

Ένας επεξεργαστής πριν ολοκληρώσει μια δοσοληψία, διαιτητεύει για άδεια και ακολούθως στέλνει το σύνολο εγγραφών του στα κατώτερα επίπεδα της ιεραρχίας μνήμης. Ταυτόχρονα οι υπόλοιποι επεξεργαστές παρακολουθούν τον διάδρομο (bus-snooping) για τις διευθύνσεις μνήμης και τα δεδομένα που γίνονται commit από τον επεξεργαστή, ώστε να ανιχνεύσουν πιθανές εξαρτήσεις δεδομένων και παράλληλα να ενημερώσουν ή να ακυρώσουν τα αντίστοιχα περιεχόμενα των κρυφών μνημών τους, έτσι ώστε να διατηρηθεί η συνεκτικότητα της μνήμης (memory coherence). Ο τρόπος υλοποίησης του snooping, είναι τέτοιος που εξασφαλίζει, ότι όλοι οι επεξεργαστές βλέπουν τα commits με την ίδια σειρά, οπότε έτσι εξασφαλίζεται η συνέπεια μνήμης (memory consistency).

Καθυστέρηση στο commit μιας δοσοληψίας μπορεί να υπάρξει για διαφόρους λόγους. Το μοντέλο TCC επιτρέπει στους προγραμματιστές να καθορίσουν δοσοληψίες που θα εκτελούνται να κάνουν commit με σειρά, μερική σειρά ή χωρίς σειρά. Ωστόσο χρησιμοποιώντας σειρά για το commit, μπορεί να υπάρξουν καθυστερήσεις αφού νεότερες δοσοληψίες θα πρέπει να περιμένουν τις παλαιότερες δοσοληψίες να ολοκληρωθούν πριν κάνουν commit. Στην περίπτωση που δεν υπάρχει κάποια σειρά του commit, καθυστερήσεις στο commit μπορούν να υπάρξουν λόγω συναγωνισμού του διαδρόμου (bus contention) λόγω πολλαπλών commit σε μικρό χρονικό διάστημα.

Για την εύρεση του συνόλου εγγραφών της δοσοληψίας υπάρχει υλοποιημένο στο hardware μια ουρά αποθήκευσης διευθύνσεων (SAF - Store Address FIFO), όπου αποθηκεύονται οι διευθύνσεις των stores. Η ουρά περιέχει βασικά δείκτες, οι οποίοι δείχνουν στις τροποποιημένες γραμμές της κρυφής μνήμης. Η ουρά δεν έχει μεγάλο hardware overhead, αφού για μια κρυφή μνήμη μεγέθους 32KB με μέγεθος γραμμής 32 bytes χρειάζονται μόλις 1024 εγγραφές στην ουρά. Κάθε εγγραφή έχει μέγεθος 10bit οπότε το μέγεθος της είναι μόλις 1.25KB. Κατά την διάρκεια της εκτέλεσης μιας δοσοληψίας, οι transaction stores ελέγχουν τα ψηφία SM της γραμμής της κρυφής μνήμης στην οποία αναφέρονται παράλληλα με την σύγκριση του tag της γραμμής. Εάν όλα τα ψηφία SM είναι 0, τότε είναι το πρώτο transaction store που υπάρχει για την γραμμή αυτή, οπότε ένας δείκτης στην γραμμή αυτή προστίθεται στην SAF. Κατά την διάρκεια του commit, διαβάζονται οι δείκτες που υπάρχουν στην SAF και γίνεται η εκροή των τροποποιημένων λέξεων στις γραμμές της κρυφής μνήμης. Μετά το τέλος της εκροής του συνόλου εγγραφής, γίνεται μηδενισμός όλων των ψηφίων SM και SR για να σημειωθεί ότι πλέον οι συγκεκριμένες γραμμές δεν είναι υποθετικές.

1.4.3 Conflict detection (εντοπισμός συγκρούσεων)

Ο εντοπισμός συγκρούσεων γίνεται, όταν κάποιος πυρήνας έχει κάνει commit κάποια δοσοληψία που εκτελούσε. Οι υπόλοιποι πυρήνες παρακολουθούν τις διευθύνσεις μνήμης που έχουν τροποποιηθεί και ανιχνεύουν τις εξαρτήσεις, που υπάρχουν μαζί με τις δοσοληψίες που εκτελούν αυτοί. Σύγκρουση υπάρχει, όταν μια δοσοληψία, που εκτελείται, έχει διαβάσει υποθετικά μια από τις λέξεις που έχουν γίνει commit με το σύνολο εγγραφής του επεξεργαστή, που έχει ολοκληρώσει την δοσοληψία του. Όταν ανιχνευθεί μια σύγκρουση, τότε οι σχετικές

με την σύγκρουση γραμμές της κρυφής μνήμης, ακυρώνονται (μηδενισμός ψηφίου valid), τα ψηφία SM, SR της γραμμής επίσης μηδενίζονται και έχουμε επανεκκίνηση της δοσοληψίας

1.4.4 Invalidate vs. Update (Ακύρωση vs. Ενημέρωση)

Ένας επεξεργαστής πρέπει να αναθεωρήσει την κατάσταση των γραμμών της L1 που τροποποιούνται λόγω κάποιου commit μιας δοσοληψίας, έτσι ώστε να διατηρηθεί η συνεκτικότητα της μνήμης. Υπάρχουν δύο πρωτόκολλα, που μπορεί να ακολουθήσει ένας πυρήνας, όταν βρει τις τροποποιημένες γραμμές βάση της συνεχής παρακολούθησης διαδρόμου. Τα δύο πρωτόκολλα είναι η ενημέρωση (update) και η ακύρωση (invalidate). Η διαφορά απόδοσης των δύο πρωτοκόλλων εξαρτάται από το ποσοστό των λέξεων, που χρησιμοποιούνται αργότερα από τον επεξεργαστή.

Το πρωτόκολλο update ενημερώνει την κρυφή μνήμη με τα δεδομένα που γίνονται commit. Αυτό είναι εφικτό, αφού ο επεξεργαστής που κάνει commit, τοποθετεί στον διάδρομο commit τα δεδομένα αυτά για να τα στείλει στην L2. Το μειονέκτημα αυτού είναι, ότι τα updates μοιράζονται την θύρα εισόδου δεδομένων στην L1 με τον πυρήνα, οπότε είναι αρκετά πιθανόν να έχουμε καθυστερήσεις κατά την διάρκεια των load/stores των πυρήνων αυτών.

Το πρωτόκολλο invalidate έχει πιο πολύπλοκο μηχανισμό. Είναι γνωστό, ότι η εγκυρότητα μιας γραμμής της κρυφής μνήμης, σηματοδοτείται από ένα ψηφίο (valid-bit). Ωστόσο υπάρχει περίπτωση να πρέπει να ακυρωθεί μια λέξη, η οποία ανήκει σε μια γραμμή, όπου υπάρχουν υποθετικές λέξεις εξαιτίας κάποιας δοσοληψίας, που εκτελείται στον πυρήνα. Ακυρώνοντας όλη την γραμμή δεν είναι επιτρεπτό, αφού έτσι υπάρχει απώλεια μέρους από το σύνολο εγγραφής και ανάγνωσης της δοσοληψίας, που εκτελείται. Λύση στο πρόβλημα αυτό είναι η ύπαρξη έγκυρων ψηφίων (valid-bits) για κάθε λέξη κάθε γραμμής, το οποίο όμως αυξάνει το hardware overhead.

1.4.5 Buffer Overflow (υπερχείλιση του buffer)

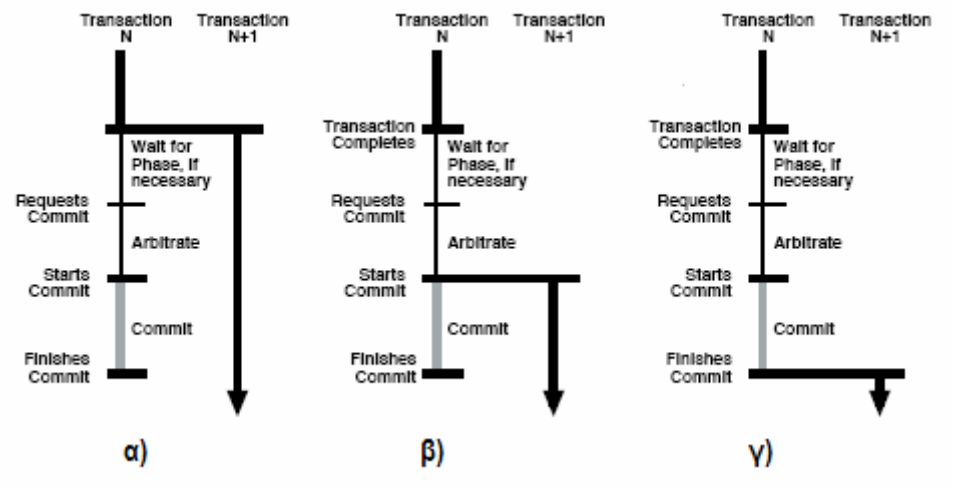
Υπάρχει σοβαρή περίπτωση υπερχείλισης του buffer. Αυτό συμβαίνει, όταν η L1 δεν μπορεί να καταγράψει τα σύνολα εγγραφής και ανάγνωσης της εκτελούμενης δοσοληψίας λόγω χωρητικότητας ή εξαιτίας του associativity της. Η υπερχείλιση μπορεί να αντιμετωπιστεί είτε με μεταπήδηση σε software transactional buffering είτε με χρήση μνήμης σε χαμηλότερο επίπεδο της ιεραρχίας μνήμης. Και στις δύο περιπτώσεις εισάγεται καθυστέρηση κι επομένως μείωση της απόδοσης του συστήματος, αφού οι νέοι αυτοί buffers είναι σαφώς πιο αργοί από αυτόν που υπάρχει στην L1. Για να αποφύγουμε αυτή την καθυστέρηση μια απλή τεχνική, που μπορεί να χρησιμοποιηθεί, είναι να γίνει commit των υπαρχόντων συνόλων εγγραφής και ανάγνωσης περιμένοντας όμως αρκετή ώρα, για να πάρει την άδεια λόγω των περιορισμών, που υπάρχουν από την σειροποιήση. Καμία άλλη δοσοληψία δεν δικαιούται να κάνει commit μέχρι η δοσοληψία που προκάλεσε την υπερχείλιση να κάνει το τελικό commit.

1.4.6 Double Buffering (διπλό buffering)

Για να αποφευχθεί η καθυστέρηση που δέχεται ένας επεξεργαστής κατά την διάρκεια αναμονής της πραγματοποίησης του commit της δοσοληψίας που εκτελείται, χρησιμοποιείται

επιπλέον hardware. Όπως φαίνεται και στο σχήμα 1.5 υπάρχουν δύο σύνολα ψηφίων SR, SM και η ύπαρξη δεύτερης SAF είναι επίσης αναγκαία, για να είναι εφικτό να ανιχνευθούν ταυτόχρονα τα σύνολα εγγραφής και ανάγνωσης των δύο δοσοληψιών. Σε περίπτωση, που οι δύο δοσοληψίες γράφουν στην ίδια λέξη, το hardware πρέπει να κρατήσει και τις δύο υποθετικές εκδόσεις της λέξης, αφού υπάρχει περίπτωση η νέα δοσοληψία να συγκρουστεί, ενώ η πιο παλιά να κάνει commit χωρίς πρόβλημα. Για να πραγματοποιηθεί αυτός ο μηχανισμός υπάρχει ένας write victim buffer, στον οποίο αντιγράφεται η γραμμή που περιέχει την λέξη. Στην συνέχεια μηδενίζει τα ψηφία SR και SM που αντιστοιχούν στην παλιά δοσοληψία. Όταν η παλιά δοσοληψία κάνει commit, κάνει commit και όλα τα δεδομένα που είναι γραμμένα στον buffer αυτό.

Στο σχήμα 1.5 παρουσιάζεται η επίδραση διαφόρων εκδόσεων του διπλού buffering. Στο 1.5α υπάρχει διπλό buffering για όλο το hardware, ενώ στο 1.5β υπάρχει μόνο για τον buffer εγγραφής. Στο 1.5γ υπάρχει μονό buffering.



Σχήμα 1.5: Επίδραση του διπλού buffering

1.4.7 Τροποποίηση του πυρήνα του επεξεργαστή

Πέρα της τροποποίησης στο επίπεδο της ιεραρχίας μνήμης, που έχει αναπτυχθεί στις προηγούμενες ενότητες, χρειάζονται κάποιες επιπλέον τροποποιήσεις στον πυρήνα του επεξεργαστή. Οι πυρήνες οφείλουν να δημιουργήσουν checkpoints των καταχωρητών στην έναρξη κάθε δοσοληψίας (ένα για single-buffering και δύο για double-buffering), έτσι ώστε να υπάρχει η δυνατότητα επαναφοράς του συστήματος σε περίπτωση ακύρωσης της δοσοληψίας. Επίσης είναι αναγκαία η ύπαρξη εντολών επεξεργαστή για την δημιουργία του checkpoint και του commit της δοσοληψίας

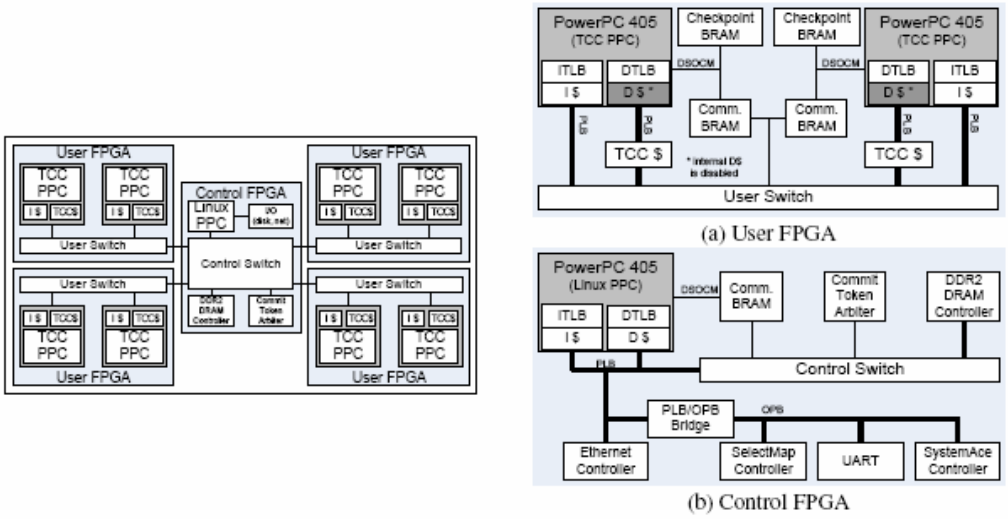
1.4.8 ATLAS: Πολυεπεξεργαστής με υποστήριξη TM

Το ATLAS [18] αποτελεί τον πρώτο CMP με υποστήριξη TM. Ακολουθεί το μοντέλο TCC για CMPs το οποίο έχει αναλυθεί στην παρούσα ενότητα. Περιλαμβάνει 8 πυρήνες PowerPC που τρέχει πολυνηματικό κώδικα και επίσης έναν επιπλέον πυρήνα, ο οποίος έχει ευθύνη για τις συσκευές εισόδου / εξόδου και στον οποίο εκτελείται το λειτουργικό σύστημα. Οι πυρήνες

τρέχουν σε συχνότητα 100MHz. Έχει υλοποιηθεί πάνω σε 5 FPGAs. Πάνω στα τέσσερα User FPGAs υλοποιούνται οι 8 πυρήνες (δύο σε κάθε FPGA). Υπάρχει επιπλέον το Control FPGA, το οποίο αποτελείται από τον 9ο πυρήνα στον οποίο τρέχει το λειτουργικό σύστημα Linux και επίσης τις συσκευές εισόδου / εξόδου. Τα User FPGA είναι ενωμένα με το Control σε συνδεσμολογία αστέρα, η οποία φαίνεται στο σχήμα 1.6

Χαρακτηριστικά του ATLAS

- Κρυφή μνήμη εντολών: 2-way 16KB. Κρυφή μνήμη δεδομένων για τους 8 πυρήνες: 4-way 32KB. Κρυφή μνήμη δεδομένων για τον 9ο πυρήνα: 2-way 16KB.
- 512MB DDR2 σε συχνότητα 200 MHz
- Είσοδος / Έξοδος: 10/100 Mbps Ethernet, RS232 UART, 512MB Compact Flash
- Λειτουργικό σύστημα: Motavista 3.1 Linux (ver 2.4.30)



Σχήμα 1.6: Συνδεσμολογία των User και Control FPGA

Κεφάλαιο 2
Αλγόριθμος Breadth First Search

2.1 Γενικά

Ο αλγόριθμος διερεύνησης πρώτα σε πλάτος (BFS) είναι ένας από τους απλούστερους αλγόριθμους διερεύνησης ενός γράφου και αποτελεί το αρχέτυπο για πολλούς σημαντικούς αλγόριθμους γράφων. Ο αλγόριθμος εύρεσης του ελάχιστου συνδετικού δέντρου του Prim και ο αλγόριθμος των ομοαφετηριακών ελαχίστων διαδρομών του Dijkstra βασίζονται σε τεχνικές παρεμφερείς με εκείνες του BFS.

Για δεδομένο γράφο G = (V,E) με n = |V| κορυφές και m = |E| ακμές και έναν δεδομένο αφετηριακό κόμβο s, ο αλγόριθμος BFS συνίσταται στη συστηματική εξέταση των ακμών του γράφου G, ώστε να εντοπιστούν όλοι οι κόμβοι που είναι προσπελάσιμοι από τον κόμβο s. Στο πλαίσιο του αλγορίθμου αυτού, υπολογίζεται η απόσταση ανάμεσα στον κόμβο s και σε κάθε προσπελάσιμο κόμβο. Δημιουργείται επίσης ένα οριζόντιο δέντρο με ρίζα τον κόμβο s, το οποίο περιέχει όλους τους προσπελάσιμους κόμβους. Για οποιονδήποτε κόμβο u προσπελάσιμο από τον κόμβο s, η διαδρομή s →…→ u στο οριζόντιο δέντρο αντιστοιχεί σε μια ελάχιστη διαδρομή από τον κόμβο s μέχρι τον κόμβο u στον γράφο G. Ο αλγόριθμος μπορεί να εφαρμοστεί τόσο σε κατευθυνόμενους γράφους όσο και σε μη κατευθυνόμενους.

Η ονομασία του αλγορίθμου οφείλεται στο γεγονός, ότι επεκτείνει το σύνορο μεταξύ εντοπισμένων και μη εντοπισμένων κόμβων ομοιόμορφα σε όλο το εύρος του συνόρου αυτού, και στο γεγονός ότι το σύνορο αυτό σχεδιάζεται κατά την οριζόντια διεύθυνση. Δηλαδή, ο αλγόριθμος εντοπίζει πρώτα όλους τους κόμβους σε απόσταση k από τον κόμβο s, και μόνο αφού εξαντλήσει αυτούς τους κόμβους προχωρά στον εντοπισμό κόμβων σε απόσταση k+1.

Ο αλγόριθμος BFS χρησιμοποιείται για υπολογισμό των ελαχίστων διαδρομών ενός κόμβου ρίζα από τους προσπελάσιμους κόμβους, τόσο σε γράφους χωρίς βάρη όσο και σε γράφους με βάρη. Ο αλγόριθμος εκ φύσεως παράγει το οριζόντιο δέντρο, το οποίο δίνει τις ζητούμενες ελάχιστες διαδρομές για γράφους χωρίς βάρη. Για γράφους με ακέραια βάρη κατασκευάζουμε τον γράφο G' όπου κάθε ακμή (u,v) με βάρος w του αρχικού γράφου G αντικαθίσταται με ένα μονοπάτι u →…→v, το οποίο αποτελείται από w ακμές βάρους 1 και w-1 επιπλέον κόμβους.

Στον αλγόριθμο που παρουσιάζουμε προυποθέτει ότι ο γράφος εισόδου G αναπαρίσταται με λίστες γειτνίασης. Χρησιμοποιούμε τον πίνακα v, στον οποίο σημειώνουμε ποιοι κόμβοι έχουν ήδη επισκεφθεί. Στον πίνακα d, σημειώνουμε την απόσταση κάθε προσπελάσιμου

κόμβου από τον κόμβο s, ενώ στον πίνακα π σημειώνουμε τον πατέρα του κάθε κόμβου, δηλαδή τον κόμβο. Επίσης χρησιμοποιούμε μια ουρά Q, στην οποία προστίθενται οι γειτονικοί κόμβοι ενός προσπελάσιμου κόμβου.

Αρχικά επισκεπτόμαστε τον κόμβο ρίζα s κι ακολούθως στην ουρά Q προσθέτουμε τους γειτονικούς κόμβους της που δεν έχουν ήδη επισκεφθεί. Για καθέναν τέτοιο κόμβο, υπολογίζεται η απόσταση του από την ρίζα βάση της απόστασης του πατέρα του ενημερώνοντας και τον πρόγονο του με το κόμβο – πατέρα. Στην συνέχεια γίνεται εξαγωγή της κεφαλής της ουράς και αυτό γίνεται συνεχώς μέχρι να αδειάσει η ουρά Q. Στο σχήμα 2.1 παρατίθεται ο ψευδοκώδικας.

```
Είσοδος : γράφος G = (V,E),  κορυφή ρίζα s, ουρά Q
Έξοδος  : πίνακας αποστάσεων d, πίνακας πρόγονων π

/*Αρχικοποιήσεις*/                /*Κυρίως σώμα αλγορίθμου*/

1 foreach u ∈ V do                10 while Q ≠ NULL do
2     d[u] ← 0;                   11    vertex ← extract(Q);
3     π[u] ← 0;                   12    father ← vertex;
4     v[u] ← 0;                   13    foreach u adjacent to vertex do
5 end                             14          if vertex not visited then
6                                 15                v[u] ← 1
7 π[s] ← -1;                      16                d[u] ← d[father]+1;
8 v[s] ←  1;                      17                π[u] ← father;
9 insert(Q,s);                    18                insert(Q,u);
                                  19    end
                                  20 end
```

Σχήμα 2.1: Αλγόριθμος BFS

Τα αποτελέσματα του αλγορίθμου BFS είναι πιθανόν να εξαρτώνται από την σειρά με την οποία εξετάζονται οι γείτονες ενός δεδομένου κόμβου, οπότε το οριζόντιο δέντρο μπορεί να ποικίλλει, αλλά οι αποστάσεις d που υπολογίζονται από τον αλγόριθμο θα είναι πάντοτε οι ίδιες. Γι' αυτό και για τον έλεγχο ορθότητας των παραλλήλων υλοποιήσεων αρκεί ένας έλεγχος ισότητας του πίνακα d που παράγεται από την παράλληλη υλοποίηση με αυτόν που παράγεται από την σειριακή.

2.2 Παραλληλοποίηση του BFS

2.2.1 Παρεμφερής Εργασία

Υπάρχουν αρκετές υλοποιήσεις σε κατανεμημένα συστήματα, οι οποίες στηρίζονται σε διαμέριση του γράφου και εξισορρόπηση φόρτου (load – balancing). Μια τέτοια υλοποίηση περιγράφεται στο [6] όπου γίνεται 1D και 2D διαμέριση του γράφου και ακολούθως παράλληλη εκτέλεση του BFS σε κάθε επεξεργαστή του κατανεμημένου συστήματος, ο οποίος έχει το δικό του σύνολο από κόμβους και ακμές.

Οι Bader και Madduri [7] δεν χρησιμοποιούν, ούτε κατανεμημένες ουρές. ούτε load-balancing. αλλά χρησιμοποιούν έναν απλό παράλληλο αλγόριθμο. ο οποίος είναι συγχρονισμένος σε επίπεδο (level-synchronization). Με τον όρο level-synchronization εννοούμε, ότι τα νήματα

πρέπει να περιμένουν το ένα το άλλο σε κάποιο είδος «barrier» μέχρις ότου ολοκληρωθεί η επίσκεψη των κόμβων, που βρίσκονται σε απόσταση k από την ρίζα (επίπεδο k), πριν προχωρήσουν στο επίπεδο k+1. Ο αλγόριθμος αποσκοπεί ταυτοχρονισμό στο γεγονός, ότι κόμβοι του ίδιου επιπέδου μπορούν να επεξεργασθούν ταυτόχρονα, ενώ οι γείτονες κάθε κόμβου μπορούν να προστεθούν στην ούρα ταυτόχρονα. Ο αλγόριθμος αυτός είναι αποδοτικός σε γράφους, όπου η διάμετρος δεν είναι μεγάλη, έτσι ώστε να υπάρχει αρκετός παραλληλισμός σε κάθε επίπεδο.

Οι Zhang και Hansen [8] χρησιμοποιούν επίσης level-synchronization σε συνδυασμό με load-balancing, ώστε τα νήματα να μην μένουν χωρίς εργασία. Χρησιμοποιούνται δύο λίστες, closed και open. Η λίστα closed είναι κοινή σε όλα τα νήματα και περιέχει όλες τις κορυφές του γράφου. Αντίθετα η λίστα open είναι ιδιωτική σε κάθε νήμα και αποτελείται από δύο ουρές current layer και next layer. Με αυτόν τον τρόπο επιτυγχάνεται το level-synchronization, αφού κάθε νήμα αδειάζοντας την current layer περιμένει τα υπόλοιπα νήματα να τελειώσουν κι αυτά, πριν προχωρήσει στην next layer. Το load-balancing επιτυγχάνεται με την ύπαρξη μιας σημαίας σε κάθε νήμα, η οποία είναι στην τιμή 1, εάν η τοπική λίστα open είναι άδεια. Γειτονικά νήματα (διάταξη δακτυλίου) μεταφέρουν κόμβους στο νήμα και ξυπνούν το νήμα αυτό, οπότε έτσι επιτυγχάνεται εξισορρόπηση του φόρτου και δεν υπάρχει μεγάλος χρόνος αδρανοποίησης για κάθε νήμα.

2.2.2 Βασισμένη σε κλειδώματα παράλληλη υλοποίηση

Έχουν εξεταστεί δύο διαφορετικές παράλληλες υλοποιήσεις. Και οι δύο υλοποιήσεις στηρίζονται στην γενική ιδέα παράλληλης επεξεργασίας κόμβων, οι οποίες βρίσκονται στο ίδιο επίπεδο. Και στις δύο υλοποιήσεις υπάρχουν δύο ουρές front και back. Η διαφορά στις δύο υλοποιήσεις έγκειται στην τοπικότητα των δύο ουρών στην πρώτη υλοποίηση, ενώ στην δεύτερη υλοποίηση οι δύο ουρές είναι κοινές..

2.2.2.1 Οι ουρές front και back είναι ιδιωτικές για κάθε νήμα

Στο σχήμα 2.2 παρατίθεται ο ψευδοκώδικας της παράλληλης υλοποίησης και στην συνεχεία περιγράφεται αναλυτικά

```
Είσοδος : γράφος G = (V,E),  κορυφή ρίζα s, ουρά Q
Έξοδος  : πίνακας αποστάσεων d, πίνακας πρόγονων π

      /*Αρχικοποιήσεις*/

 1 foreach u ∈ V do
 2    d[u] ← 0;
 3    π[u] ← 0;
 4    v[u] ← 0;
 5 end
 6 foreach thread do
 7    available[thread] ← 1;
 9 end
10 available[0] ← 0;
11 π[s] ← -1;

      /*Κυρίως σώμα αλγορίθμου*/
```

```
12 while workthreads ≠ 0 do
13    while available[tid]=1 and workthreads ≠ 0 do ;
14    if workthreads=0 then break;
15    Begin-Atomic
16          v[s] ← 1;
17    End-Atomic
18    insert(front,s);
19    Barrier
20    while front ≠ NULL or back ≠ NULL do
21          vertex ← extract(front);
22          father ← vertex;
23          foreach u adjacent to vertex do
24                Begin-Atomic
25                if back = NULL and u not visited then
26                      v[u] ← 1
27                      End-Atomic
28                      insert(back,u);
29                elif back ≠ NULL and u not visited then
30                      if available thread exists then
31                            available[av_tid] ← 0;
32                            workthreads ← workthreads + 1;
33                            End-Atomic
34                      else
35                            v[u] ← 1;
36                            End-Atomic
37                            insert(back,u);
38                d[u] ← d[father] + 1;
39                π[u] ← father;
40          end
41          if front = NULL then
42                front ← back;
43                back ← NULL;
44                if front ≠ NULL then Barrier
45    end
46    Begin-Atomic
47    available[tid] ← 1;
48    workthreads ← workthreads - 1;
49    End-Atomic
50 end
```

Σχήμα 2.2: Παράλληλη υλοποίηση του αλγόριθμου BFS

Δημιουργούνται τα νήματα κι ακολούθως όλα εκτός από ένα, μπαίνουν σε αναμονή. Στη συνέχεια το νήμα που δεν βρίσκεται σε αναμονή εκτελεί τον αλγόριθμο που περιγράφηκε στην ενότητα 2.1 με κάποιες διαφοροποιήσεις. Κατ' αρχήν προσθέτει στην ουρά back τον πρώτο γείτονα του κόμβου, που έχει εξαχθεί από την ουρά front. Στη συνέχεια κοιτάζει αν υπάρχουν διαθέσιμα νήματα, δηλαδή νήματα που βρίσκονται ακόμη σε αναμονή κι ακολούθως τα ξυπνάει δίνοντας τους ως αρχική κόμβο έναν από τους γειτονικούς κόμβους που υπάρχουν. Αν δεν υπάρχουν διαθέσιμα νήματα, οι υπόλοιποι γείτονες προστίθενται στην τοπική ουρά back. Η διαδικασία αυτή επαναλαμβάνεται ακριβώς η ίδια κι από τα υπόλοιπα νήματα. Εφ' όσον έχουμε νήματα να εκτελούν παράλληλα τον αλγόριθμο BFS χρειάζεται κάποιο είδος συγχρονισμού σε κάθε επίπεδο αναζήτησης. Έτσι υλοποιούμε ένα είδος «barrier», στον οποίο κάθε νήμα μπαίνει σε αναμονή, όταν τελειώσει με το παρών επίπεδο (δηλαδή η ουρά front είναι άδεια). Το νήμα περιμένει όλα τα νήματα τα οποία εργάζονται να φτάσουν και αυτά στον «barrier». Αφού γίνει ο απαραίτητος συγχρονισμός και έχει γίνει δηλαδή επίσκεψη όλων των κόμβων, που βρίσκονται στο επίπεδο k, τα νήματα προχωρούν στην επίσκεψη των κόμβων

του επιπέδου k+1, με μια απλή ανταλλαγή των δεικτών front και back μεταξύ τους. Τα νήματα συνεχίζουν τον αλγόριθμο BFS, μέχρι να ολοκληρωθεί η διάσχιση του γράφου.

Για να πετύχουμε το load-balancing, δηλαδή τα νήματα να ξαναπαίρνουν δουλειά όταν ολοκληρώσουν την παλιά τους εργασία, υποχρεώνουμε κάθε νήμα, όταν επισκέπτεται έναν κόμβο να ελέγχει αν υπάρχει κάποιο νήμα, το οποίο βρίσκεται σε αναμονή, έτσι ώστε να το ξυπνήσει δίνοντας του σαν αρχικό κόμβο τον κόμβο αυτό. Χρησιμοποιείται ένας πίνακας available για να δείξει ποια νήματα είναι κοιμούνται και δηλαδή είναι διαθέσιμα να ξεκινήσουν τον αλγόριθμο BFS εκ νέου. Ο έλεγχος πάνω στον πίνακα available αποτελεί κρίσιμο τμήμα της εφαρμογής, δηλαδή τμήμα το οποίο πρέπει να προστατευτεί με begin/end atomic έτσι ώστε να αποφύγουμε την περίπτωση που δύο νήματα προσπαθήσουν να ξυπνήσουν το ίδιο νήμα με διαφορετικό κόμβο..

Χρησιμοποιώντας τοπικές ουρές σε κάθε νήμα, η εξαγωγή κορυφών από την ουρά δεν αποτελεί κρίσιμο τμήμα της εφαρμογής, αφού είναι μεταξύ τους ανεξάρτητες και τοπικές σε κάθε νήμα. Ταυτόχρονη πρόσβαση από τα νήματα έχουμε στον πίνακα v, στον πίνακα available και τέλος στην μεταβλητή workthreads, η οποία δείχνει πόσα νήματα εργάζονται. Έτσι απαιτείται η χρήση mutex-locking, για να γίνει ο συγχρονισμός της πρόσβασης στις παραπάνω δομές δεδομένων. Εξετάζουμε δύο υλοποιήσεις Lock_ private_fg και Lock_ private_cg. Στην υλοποίηση ***Lock_private_cg*** η πρόσβαση στον πίνακα v γίνεται με την χρήση ενός γενικού mutex-lock *lock1*, ενώ στην υλοποίηση ***Lock_ private_fg*** χρησιμοποιείται fine-grained locking για κάθε κόμβο. Για τις υπόλοιπες δομές δεδομένων χρησιμοποιείται ο mutex-lock *lock*.

Όπως αναφέραμε και προηγουμένως για να επιτύχουμε το level-synchronization χρειαζόμαστε «barriers» σε σημεία κώδικα, όπου το κάθε νήμα τελειώνει με το επίπεδο k. Αυτό γίνεται στην γραμμή 19 κι επίσης στην γραμμή 44. Στην γραμμή 44 γίνεται έλεγχος και αν το επίπεδο k δεν είναι το τελευταίο, τότε χρησιμοποιείται ο «barrier» διαφορετικά όχι, αφού δεν υπάρχουν άλλα επίπεδα, οπότε δεν υπάρχει λόγος συγχρονισμού. Ο «barrier» που χρησιμοποιούμε υλοποιείται με mutex και μεταβλητές ελέγχου (condition variables) κι εξαρτάται από την μεταβλητή workthreads, η οποία δείχνει κάθε φορά πόσα νήματα πρέπει να περιμένει ο «barrier» για να ελευθερώσει τα νήματα που βρίσκονται σε αναμονή.

Ένα νήμα, όταν δεν έχει άλλους κόμβους να επισκεφθεί δηλαδή οι δύο τοπικές ουρές front και back είναι κενές κοιμάται και μπαίνει σε αναμονή ενημερώνοντας τον πίνακα available κι επίσης μειώνει την μεταβλητή workthreads, η οποία δείχνει πόσα νήματα εργάζονται

Όταν η μεταβλητή workthreads γίνει ίση με μηδέν, τα νήματα τερματίζουν και πλέον ο αλγόριθμος έχει ολοκληρωθεί με τα αποτελέσματα του να βρίσκονται στους πίνακες v, d και π.

2.2.2.2 Οι ουρές front και back είναι κοινές για όλα τα νήματα

Στο σχήμα 2.3 παρατίθεται ο ψευδοκώδικας της παράλληλης υλοποίησης που έχει περιγραφεί στην ενότητα αυτή.

```
Είσοδος : γράφος G = (V,E),  κορυφή ρίζα s, ουρά Q
Έξοδος  : πίνακας αποστάσεων d, πίνακας πρόγονων π

      /*Αρχικοποιήσεις*/
```

```
 1 foreach u ∈ V do
 2    d[u] ← 0;
 3    π[u] ← 0;
 4    v[u] ← 0;
 5 end
 6 π[s] ← -1;

      /*Κυρίως σώμα αλγορίθμου*/

7 if tid = 0 then
8     v[s] ← 1;
9     insert(front,s);
10 end
11
12 while front ≠ NULL or back ≠ NULL do
13    Begin-Atomic
14    if front ≠ NULL do
15          vertex ← extract(front);
16          numVer ← numVer - 1;
17          End-Atomic
18    else
19          End-Atomic
20          continue;
21    father ← vertex;
22    foreach u adjacent to vertex do
23          Begin-Atomic
24          if u not visited then
25                v[u] ← 1;
26                insert(back,u);
27                End-Atomic
28                d[u] ← d[father] + 1;
29                π[u] ← father;
30          end
31          if front = NULL then
32                front ← back;
33                back ← NULL;
34                End-Atomic
35    end
36 end
```

Σχήμα 2.3: Παράλληλη υλοποίηση του αλγόριθμου BFS

Δημιουργούνται τα νήματα κι ακολούθως όλα εκτός από ένα νήμα, μπαίνουν σε αναμονή εάν δεν υπάρχουν διαθέσιμες κορυφές για εξαγωγή από την ουρά front. Στη συνέχεια το νήμα που δεν βρίσκεται σε αναμονή εκτελεί τον αλγόριθμο που περιγράφηκε στην ενότητα 2.1 με πολύ μικρές διαφοροποιήσεις. Βασική διαφοροποίηση είναι, όπως προαναφέρθηκε, η χρήση δύο ουρών front και back, η οποίες πλέον κοινές στα νήματα. Στην υλοποίηση αυτή οι δύο ουρές δεν χρησιμοποιούνται για σκοπούς level-synchronization, αφού αυτό επιτυγχάνεται έμμεσα από την χρήση κοινών ουρών, οπότε είναι βέβαιο ότι τα νήματα θα επισκέπτονται κόμβους ίδιου επιπέδου. Χρησιμοποιούμε δύο ουρές, μια για εξαγωγή και μια για εισαγωγή κόμβων. Με αυτόν τον τρόπο μειώνουμε την σύγκρουση δύο νημάτων, όπου το ένα αφαιρεί από την ουρά front ταυτόχρονα με κάποιο άλλο που εισάγει στην ουρά back. Στην περίπτωση που χρησιμοποιούσαμε μια ουρά, τότε αυτό θα οδηγούσε την ανάγκη να χρησιμοποιήσουμε το ίδιο κλείδωμα για τα κρίσιμα τμήματα 14-17 και 24-27. Η υλοποίηση αυτή ονομάζεται ***Lock_global***.

Χρησιμοποιώντας κοινές ουρές επιτυγχάνεται καλύτερο load-balancing, αφού όλα τα νήματα κάνουν εξαγωγή κόμβου από την ίδια ουρά, οπότε τα νήματα δεν έχουν καθυστέρηση που να οφείλεται στην έλλειψη κόμβων.

Ωστόσο χρησιμοποιώντας κοινές ουρές, χρειάζεται η χρήση mutex-locks για να συγχρονιστεί η πρόσβαση των νημάτων σε αυτές. Αυτό έχει ως αποτέλεσμα τα κρίσιμα τμήματα του αλγορίθμου, να είναι μεγαλύτερα σε σχέση με την προηγούμενη υλοποίηση. Αυτό πιθανότατα θα επηρεάσει την απόδοση του αλγορίθμου, αφού τα νήματα θα περιμένουν μεγαλύτερο χρονικό διάστημα στους mutex-locks.

Να σημειώσουμε, ότι αρχικά η υλοποίηση αυτή ήταν διαφορετική. Υπήρχε μια μεταβλητή η οποία μετρούσε τον αριθμό των κορυφών, που υπάρχουν στις δύο ουρές και τότε ενεργοποιούσε τα υπόλοιπα νήματα. Αυτή η προσέγγιση έχει παραληφθεί, αφού μετά από πολύωρα debugging, ανιχνεύθηκε πρόβλημα σωστής ενεργοποίησης με αποτέλεσμα τα αποτελέσματα της υλοποίησης να μην συμφωνούν με αυτά της σειριακής.

2.2.3 Βασισμένη σε transactional memory παράλληλη υλοποίηση

2.2.3.1 Οι ουρές front και back είναι ιδιωτικές για κάθε νήμα

Έχουμε ακριβώς την ίδια υλοποίηση που περιγράφεται στον ψευδοκώδικα στο σχήμα 2.2, αλλά στα Begin-Atomic και End-Atomic, χρησιμοποιούμε BEGIN_TRANSACTION και COMMIT_TRANSACTION αντίστοιχα. Χρησιμοποιώντας δοσοληψίες σε αυτά τα σημεία αναμένεται να υπάρξει βελτίωση στην απόδοση της υλοποίησης, αφού με αυτόν τον τρόπο τα νήματα τρέχουν παράλληλα, αν δεν υπάρχει κάποια σύγκρουση. Ωστόσο η υλοποίηση του «barrier» παραμένει ίδια και αναμένουμε να περιορίσει την απόδοση, που θα σημειωθεί με την χρήση της transactional memory. Η υλοποίηση αυτή ονομάζεται ***TM_private***.

2.2.3.2 Οι ουρές front και back είναι κοινές για όλα τα νήματα

Χρησιμοποιώντας το μοντέλο TM κατασκευάζουμε δύο υλποιήσεις. Η μια υλοποίηση χρησιμοποιεί μια coarse-grained δοσοληψία ακριβώς όπως και στο σχήμα 2.3 που περιγράφεται η παράλληλη υλοποίηση και η άλλη fine-grained δοσοληψίες. Ο τροποποιημένος ψευδοκώδικας της υλοποίησης αυτή παρατίθεται στο σχήμα 2.4. Στο σχήμα 2.5 παρατίθεται ο ψευδοκώδικας της τροποποιημένης συνάρτησης insertFG.

```
Είσοδος : γράφος G = (V,E),  κορυφή ρίζα s, ουρά Q
Έξοδος  : πίνακας αποστάσεων d, πίνακας πρόγονων π
```

```
1 … 29 όπως στο σχήμα 2.2

30    foreach u adjacent to vertex do
31          if u not visited then
32                Begin-Transaction
33                      v[u] ← 1;
34                Commit-Transaction
35                insertFG(back,u);
36
37                if threads not active and numVer > N then
```

```
38                      foreach thread do
39                            locks[thread] ← 0;
40                            v[u] ← 1;
41                      end
42                else
43                      Begin-Atomic
44                            numVer ← numVer + 1;
45                      End-Atomic
46                d[u] ← d[father] + 1;
47                π[u] ← father;
48          end
49          if front = NULL then
50                front ← back;
51                back ← NULL;
52    end
```

Σχήμα 2.4: Fine-grained TM υλοποίηση του αλγόριθμου BFS

```
Είσοδος : κορυφή num, ουρά προτεραιότητας q

1     Memory-Allocate of struct node p
2     p.num ← num
3     p.next ← NIL
4     Begin-Transaction
5           if q is empty then
6                 q.first ← p
7                 q.last ← p
8           else
9                 q.last.next ← p
10                q.last ← p
11    Commit-Transaction
```

Σχήμα 2.5: Υλοποίηση της insertFG

Επιθυμούμε να εξετάσουμε την fine-grained υλοποίηση ***TM_global_fg*** απέναντι στην coarse-grained υλοποίηση ***TM_global_cg***. Αφ' ενός η fine-grained υλοποίηση έχει μεγαλύτερο overhead, αφού θα δημιουργήσει περισσότερες δοσοληψίες κι αφ' ετέρου σε περίπτωση abort μιας δοσοληψία η coarse-grained θα έχει μεγαλύτερο overhead, αφού θα χαθεί περισσότερη εργασία απ' ότι στην fine-grained υλοποίηση. Εξετάζεται δηλαδή το trade off μεταξύ μεγάλων και μικρών δοσοληψιών.

Κεφάλαιο 3

Ανάλυση αποτελεσμάτων εξομοίωσης

3.1 Περιγραφή Συστήματος Προσομοίωσης

Η εκτίμηση της απόδοσης των διαφόρων υλοποιήσεων του αλγόριθμου BFS έγινε με την βοήθεια πλήρους συστήματος εξομοίωσης χρησιμοποιώντας το εργαλείο Wisconsin GEMS v. 2.1 [10,11] σε συνδυασμό με τον εξομοιωτή Simics v. 3.0.31 [12]. Ο Simics παρέχει εξομοίωση SPARC πολυπύρηνου επεξεργαστή (CMP), ο οποίος τρέχει λειτουργικό σύστημα Solaris 10. Η μονάδα Ruby του GEMS παρέχει λεπτομερής εξομοίωση της ιεραρχίας μνήμης και επίσης συμπεριφέρεται σαν in-order single issue επεξεργαστής για τις εντολές που δεν αναφέρονται στην μνήμη εκτελώντας μια εντολή για κάθε κύκλο προσομοίωσης.

Το προγραμματιστικό μοντέλο Hardware TM υποστηρίζεται στον GEMS μέσω του LogTM-SE υποσυστήματος [13]. Υλοποιείται πάνω σε σύστημα CMP με τοπικές L1 caches και κοινή L2 cache. Όπως αναφέρθηκε στο κεφάλαιο 1, υποστηρίζει eager version management και eager conflict detection. Για να επιταχυνθεί η ανίχνευση συγκρούσεων χρησιμοποιούνται υπογραφές (hardware signatures) για τα read/write σύνολα της δοσοληψία. Όταν δύο δοσοληψίες συγκρουστούν, μια από αυτές καθυστερεί και ξαναπροσπαθεί να εκτελεστεί ή κάνει abort αν ανιχνευθεί κάποιο πιθανό αδιέξοδο. Αναλυτική περιγραφή του LogTM έγινε στο Κεφάλαιο 1.

Στα πειραματικά αποτελέσματα που παρουσιάζονται στην επόμενη ενότητα του κεφαλαίου αυτού χρησιμοποιούνται hardware signatures μεγέθους 2Kb όπως αυτές παρουσιάζονται στο [13] για να αποδώσουν παρομοίως με ιδανικές υπογραφές. Επίσης χρησιμοποιούμε HYBRID στρατηγική για τον conflict manager, ο οποίος προτιμά τις παλιότερες δοσοληψίες απέναντι στις νεότερες, όταν ανιχνευθεί κάποια σύγκρουση. Χαρακτηριστικά του συστήματος που χρησιμοποιήσαμε παρουσιάζεται λεπτομερέστερα στο σχήμα 3.1.

Για τον πολυνηματικό προγραμματισμό χρησιμοποιήσαμε την βιβλιοθήκη των Posix Threads για δημιουργία των νημάτων και συγχρονισμό τους με «barriesr» και mutex-locking. Για να αποφύγουμε την σύγκρουση του προγράμματος με το λειτουργικό σύστημα χρησιμοποιήσαμε περισσότερους πυρήνες από τον αριθμό των νημάτων που εξετάζαμε σε κάθε προσομοίωση. Για παράδειγμα η εξομοίωση 2,3,4,5,6,7 νημάτων έγινε 8 πυρήνες ενώ για 8 νήματα σε 16 πύρηνες. Για να υποχρεώσουμε τα νήματα να δρομολογηθούν σε συγκεκριμένο επεξεργαστή χρησιμοποιήσαμε την εντολή συστήματος pset_bind του Solaris. Τέλος όλοι οι κώδικες έχουν μεταγλωττιστεί με C-compiler του Sun Studio 12 χρησιμοποιώντας βελτιστοποίηση O3.

Simics	Processor	Παραμετροποίηση μέχρι και 16 πυρήνες
Ruby	L1 caches	Ιδιωτική 64Kb, 4-way set-associative, cache line 64 byte 3 κύκλοι για hit
	L2 cache	Μοιραζόμενη και χωρισμένη σε banks 4Mb 4-way set-associative, cache line 64 byte 12 κύκλοι για hit
	Memory	4Gb, 160 κύκλοι για πρόσβαση
	TM System	HYBRID 2Kb hardware signatures

Σχήμα 3.1: Παραμετροποίηση της μονάδας Ruby

3.2 Εξεταζόμενοι Γράφοι

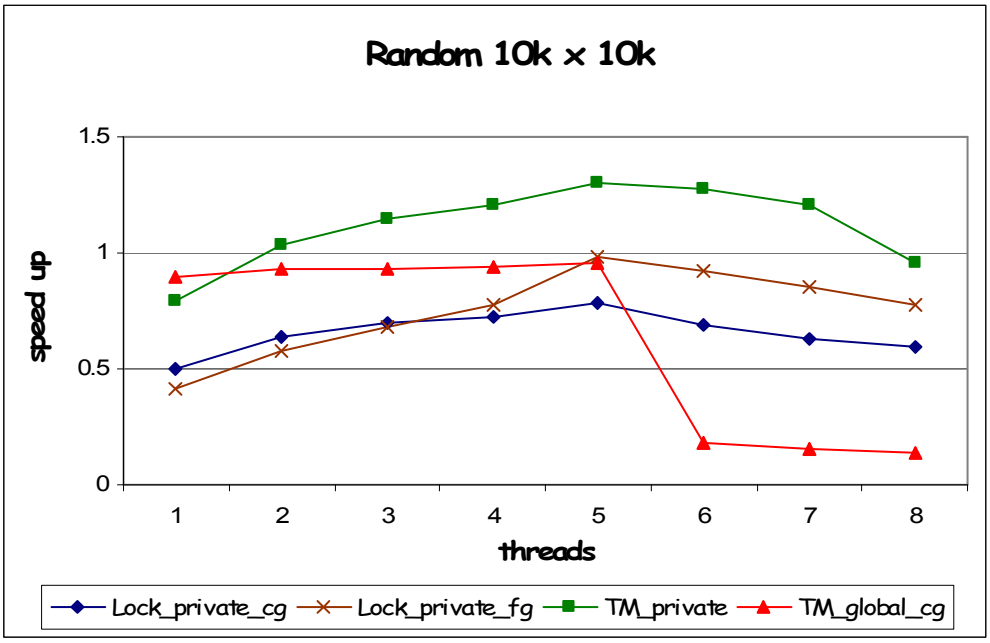
Για να εκτιμήσουμε την απόδοση των διαφορετικών υλοποιήσεων χρησιμοποιήσαμε γράφους που διαφέρουν σε μέγεθος, πυκνότητα και δομή. Χρησιμοποιήσαμε το εργαλείο GTgraph [14] για την κατασκευή των γράφων **Random** (γράφοι που δεν έχουν κάποια συγκεκριμένη δομή) και **R-MAT** (γράφοι με power-law κατανομές βαθμού, κατασκευασμένες χρησιμοποιώντας το Recursive Matrix [15]). Όλοι οι γράφοι αποτελούνται από 10K κορυφές. Για κάθε οικογένεια δημιουργήσαμε τέσσερις γράφους, ο καθένας με διαφορετική πυκνότητα (δηλαδή διαφορετικό αριθμό ακμών). Οι παράμετροι που χρησιμοποιήθηκαν για την κατασκευή των γράφων παρουσιάζεται στο σχήμα 3.2. Σε όλες τις περιπτώσεις δεν υπάρχουν παράλληλες ή κυκλικές ακμές.

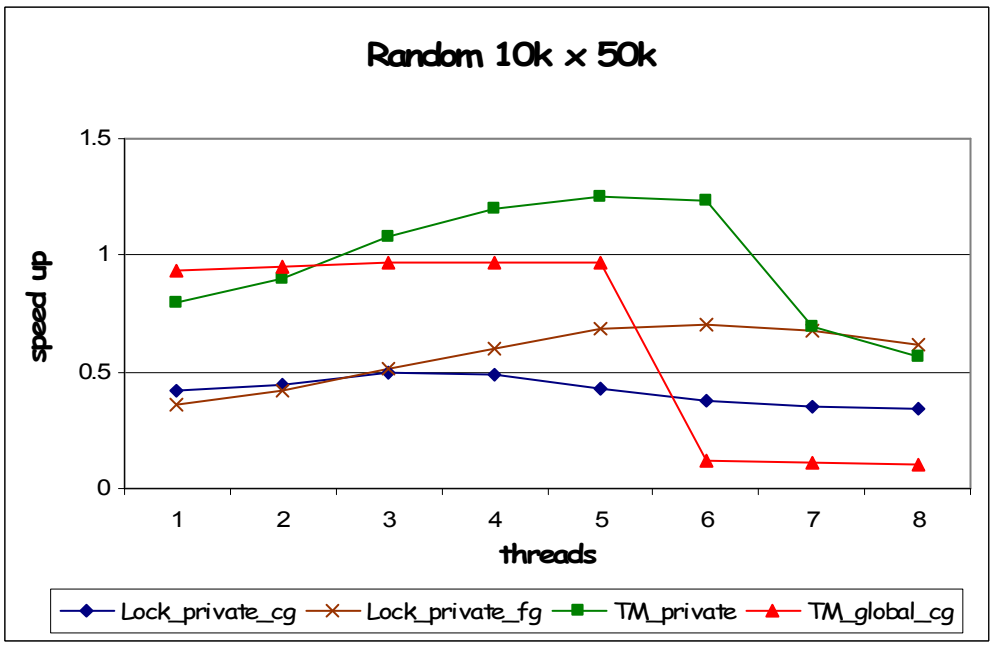
Οικογένεια	Παραμέτροι	Ακμές
Random		10K
		50K
		100K
		200K
R-MAT	a = 0.45 b = 0.15 c = 0.15 d =0.25	10K
		50K
		100K
		200K

Σχήμα 3.2: Παράμετροι κατασκευής γράφων

3.3 Αποτελέσματα Εξομοίωσης

Μετρήσεις έχουν ληφθεί για τις παράλληλες υλοποιήσεις της ενότητας 2.2.2.1 Lock_private_cg και Lock_private_fg, την υλοποίηση της ενότητας 2.2.3.1 TM_private και τέλος της coarse-grained έκδοσης της ενότητας 2.2.3.2 TM_global_cg μόνο για Random γράφους, αφού κρίνουμε μη αναγκαίο να ληφθούν για τους Rmat, αφού αναμένουμε ότι θα υπάρχει εξίσου άσχημη συμπεριφορά. Στα σχήματα που ακολουθούν παρουσιάζονται οι υλοποιήσεις που συζητήθηκαν στο κεφάλαιο 2, όπου παρατηρούμε ότι η υλοποίηση TM_private είναι πιο αποδοτική από τις υπόλοιπες υλοποιήσεις. Στα σχήματα που ακολουθούν παρουσιάζουμε την επιτάχυνση των υλοποιήσεων σε σχέση με την σειριακή υλοποίηση.



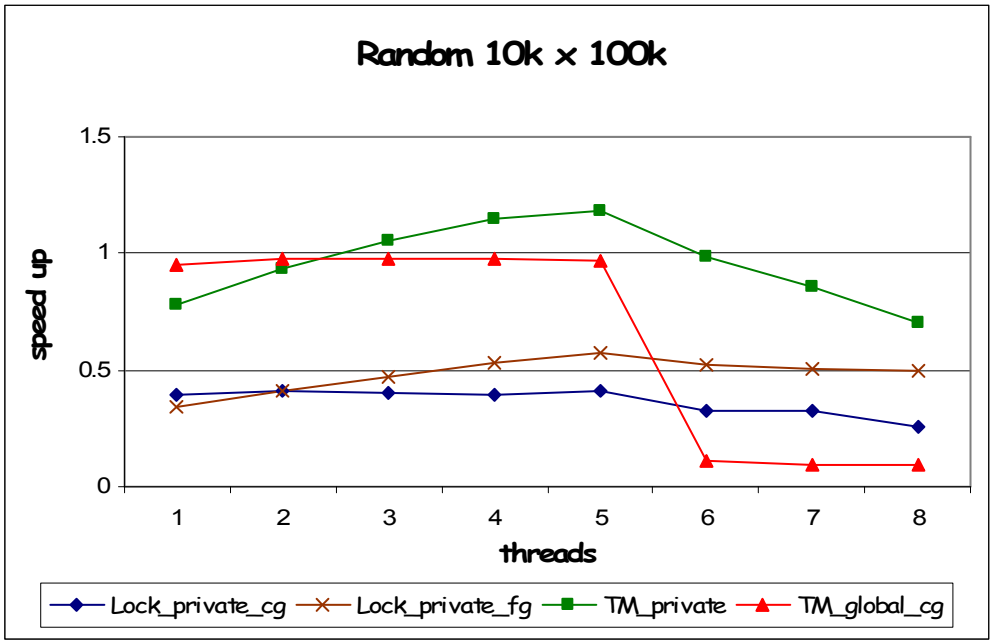


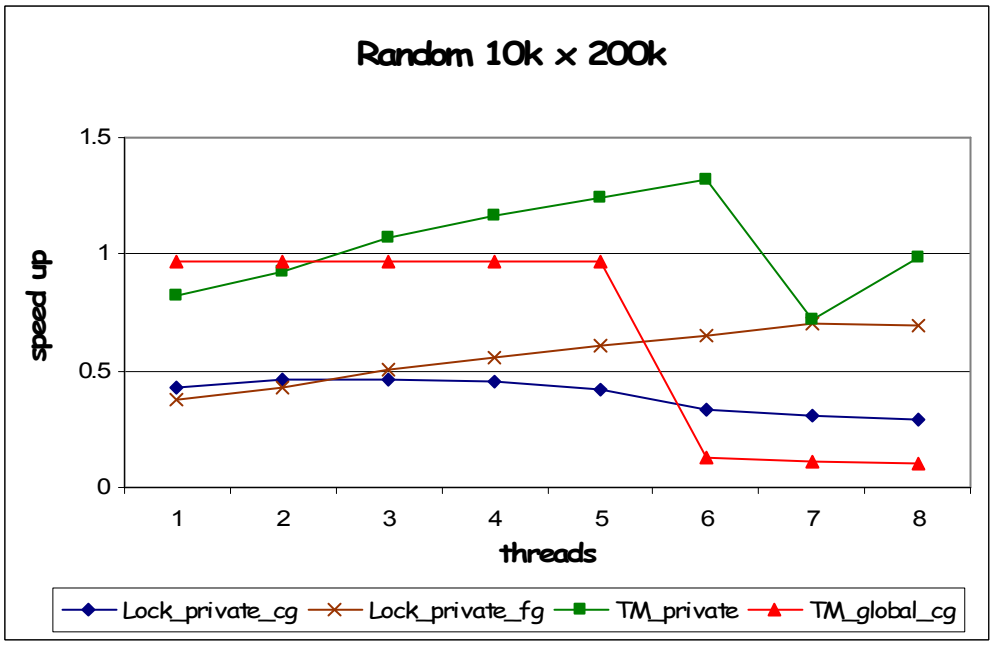
Ruby cycles

N	Vertices = 10k Edges = 10k				Vertices = 10k Edges = 50k			
	Lock_cg	Lock_fg	TM_private	TM_global	Lock_cg	Lock_fg	TM_private	TM_global
1	15967882	19173424	9988894	8846935	43614719	50806114	23185106	19738373
2	12534655	13765684	7711746	8559610	41085032	43837896	20436911	19369451
3	11398510	11621568	6943461	8530174	37115473	35703055	16976423	19053763
4	11002007	10280337	6590077	8476475	37747074	30884042	15356767	19053177
5	10175559	8098546	6090873	8308215	43008680	26771719	14742861	19055934
6	11534912	8597162	6229964	44819764	48830952	26214912	14863931	150086608
7	12562580	9297162	6561679	51335890	52412855	27324048	26381042	167641605
8	13303628	10213124	8276327	56313229	53768740	29621729	32383730	181277397

Serial = 7945560

Serial = 18399424

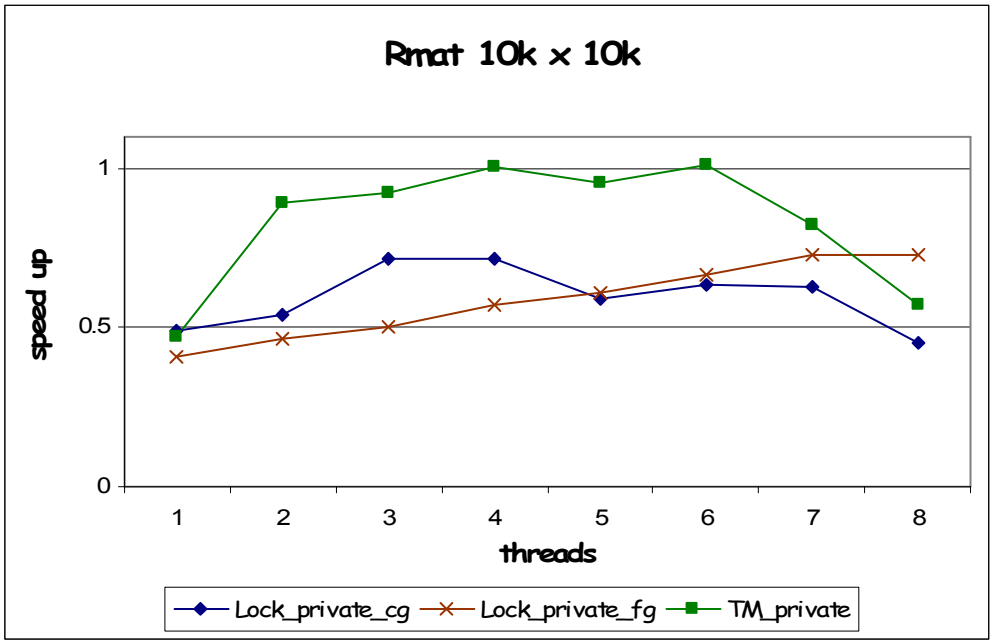


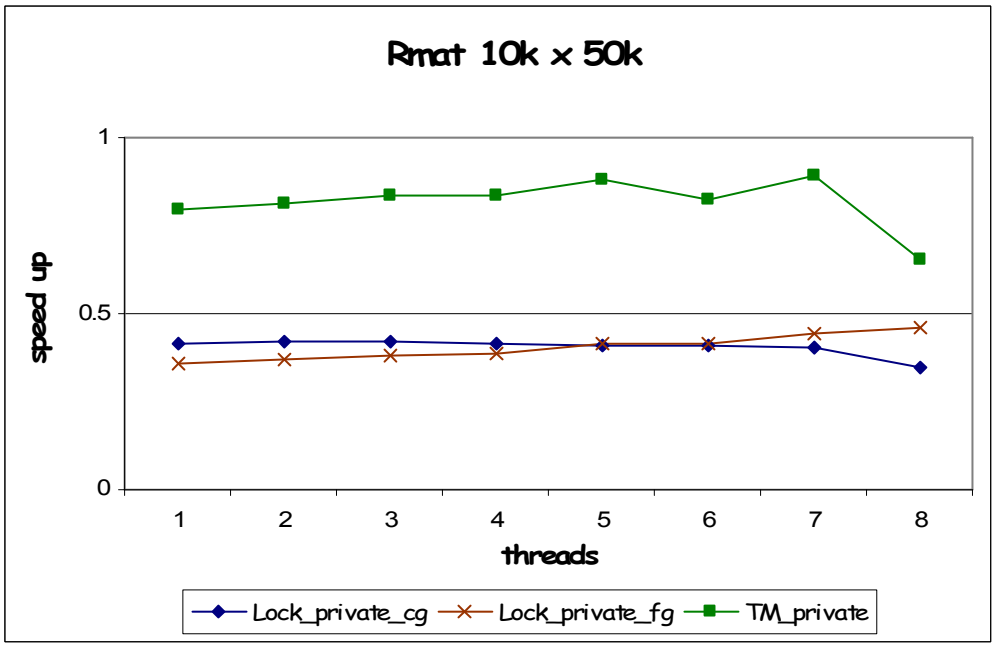


Ruby cycles

N	Vertices = 10k Edges = 100k				Vertices = 10k Edges = 200k			
	Lock_cg	Lock_fg	TM_private	TM_global	Lock_cg	Lock_fg	TM_private	TM_global
1	75978885	87499086	38345833	31364207	151415463	172591716	79735982	67676506
2	72314725	72240714	31870467	30582514	142665754	152382812	70880422	67387858
3	74760447	62933988	28322424	30575717	141246185	130022671	61196562	67349868
4	75232830	56288001	26049173	30541497	143870323	117423594	56334027	67387772
5	73244057	51816740	25306543	30972705	156259538	107862005	52625668	67411413
6	91617704	57452710	30267133	273389840	197975696	100646215	49466124	520911280
7	90905857	59571002	34940557	303758641	213813430	93355819	90943982	588222687
8	115797096	59997150	42787232	328325928	223241123	93689581	66326507	630236163

Serial = 29904604 **Serial = 67437977**

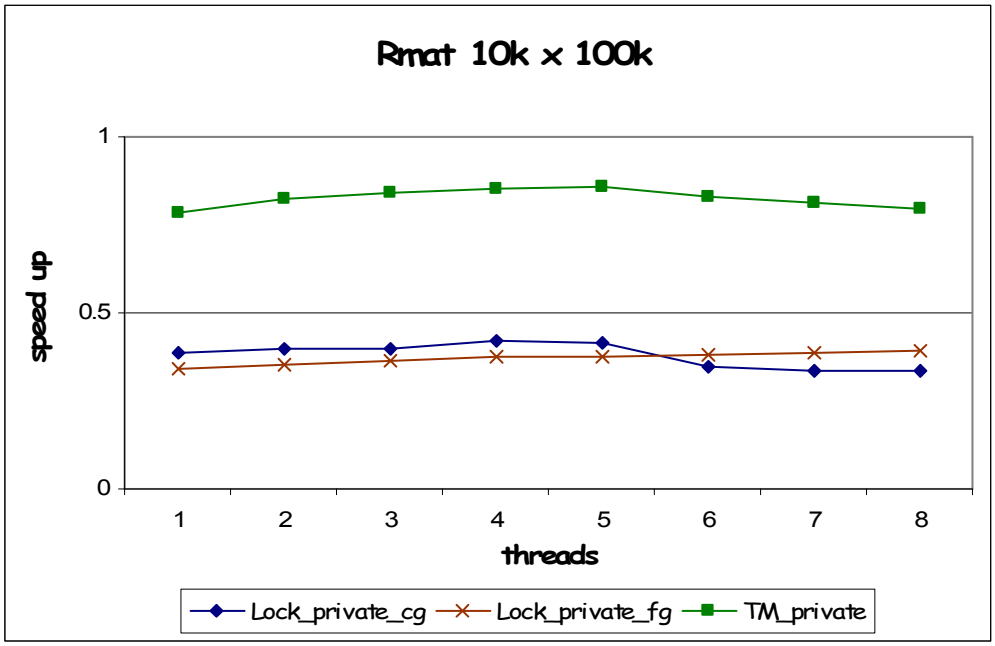


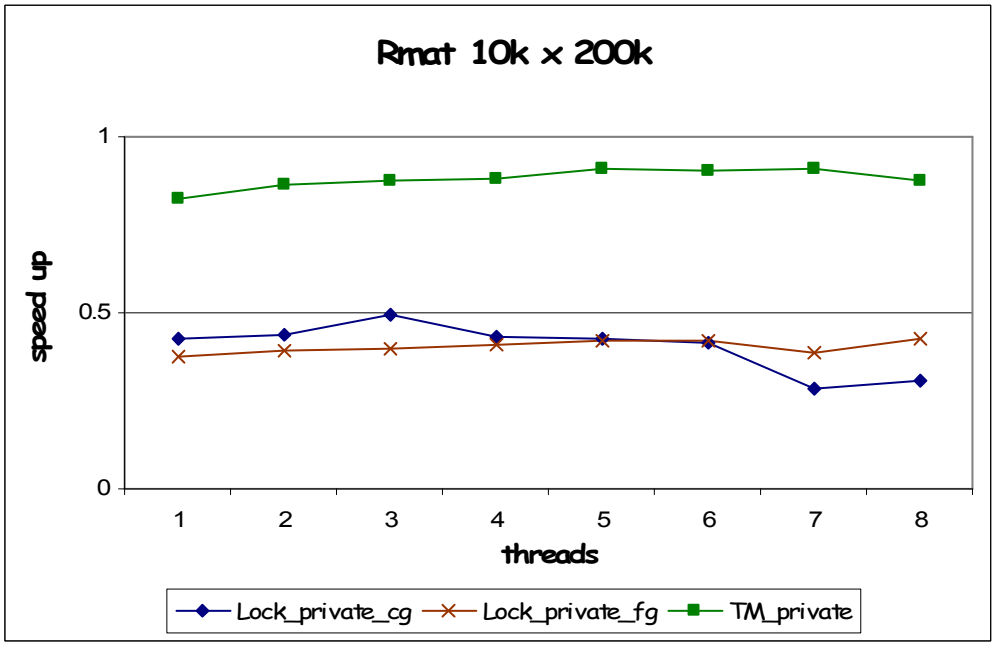


Ruby cycles

	Vertices = 10k Edges = 10k			Vertices = 10k Edges = 50k		
N	Lock_cg	Lock_fg	TM_private	Lock_cg	Lock_fg	TM_private
1	13803355	16579422	14392561	41144882	47696489	21470717
2	12517728	14492249	7588035	40646591	46464386	21026822
3	9438355	13508559	7304026	40472396	45026915	20447668
4	9439072	11761366	6705782	41275164	44267376	20400674
5	11408253	11109779	7081403	41491223	41274416	19357023
6	10647117	10166531	6670267	41611449	41127031	20711149
7	10806348	9253703	8204311	42385937	38624930	19111589
8	14854068	9246075	11835338	49553715	37251055	26097621

Serial = 6762995 **Serial = 17045993**





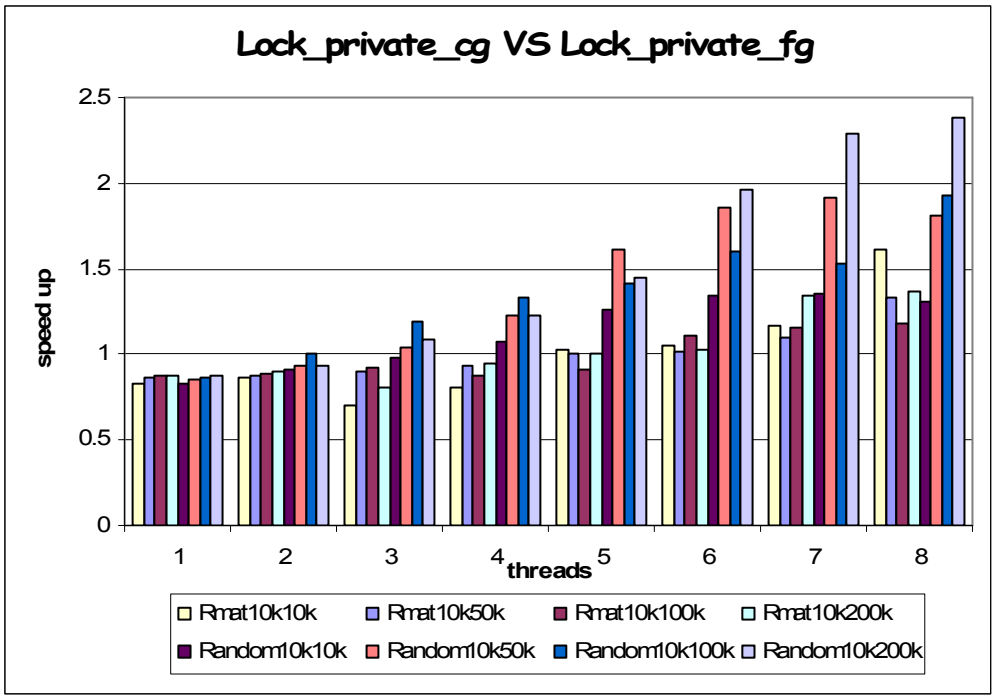
Ruby cycles

	Vertices = 10k Edges = 100k			Vertices = 10k Edges = 200k		
N	Lock_cg	Lock_fg	TM_private	Lock_cg	Lock_fg	TM_private
1	72631788	83364834	36080897	143491392	163338649	74101362
2	71091070	80277927	34364907	139660159	156090336	70748415
3	71524533	77720856	33574165	124234690	153918980	69959919
4	66796759	75863144	33228954	142211433	150432978	69698186
5	68271104	75161654	33045102	144494320	144646563	67180276
6	82157592	73928220	34025378	148520802	144856840	67723498
7	84555503	72752071	34819435	214735877	159490384	67473884
8	84571938	72000023	35434443	197770089	144323061	69837802

Serial = 28261148 **Serial = 61194974**

3.4 Ανάλυση Αποτελεσμάτων

Κατ' αρχήν παρατηρούμε, ότι η παράλληλη υλοποίηση με χρήση coarse-grained mutex-locking αδυνατεί να δώσει βελτίωση στην υλοποίηση. Μάλιστα έχει χειρότερη συμπεριφορά από την σειριακή υλοποίηση. Αυτό ήταν αναμενόμενο, αφού για κάθε κόμβο που επισκέπτεται κάποιο νήμα χρειάζεται να δεσμεύσει το lock1, για να πετύχουμε τον συγχρονισμό των νημάτων και να αποφύγουμε ταυτόχρονη πρόσβαση δύο νημάτων στην ίδια θέση μνήμης. Αυτό προσθέτει καθυστέρηση σε κάθε νήμα, αφού είναι υποχρεωμένο να περιμένει το νήμα που κρατάει το κλείδωμα. Επίσης στην υλοποίηση Lock_private_cg, που εξετάσαμε στον εξομοιωτή γίνεται χρήση τοπικών ουρών. Κάθε νήμα επισκέπτεται τους γείτονες των κόμβων της ουράς front και όταν αυτή αδειάσει, είναι υποχρεωμένο να περιμένει στον barrier που έχουμε υλοποιήσει για να γίνει ο συγχρονισμός στο επίπεδο (level-synchronization). Αυτό προκαλεί ακόμη μεγαλύτερες καθυστερήσεις, αφού τα νήματα μένουν για αρκετή ώρα ανενεργά. Σε συνδυασμό με τις καθυστερήσεις που εισάγουν τα κλειδώματα που χρησιμοποιούμε στην υλοποίηση, καθιστούν την υλοποίηση Lock_private_cg μέχρι και 4 φορές χειρότερη από την σειριακή. Η χειρότερη απόδοση παρουσιάζεται συνήθως σε μεγάλο αριθμό νημάτων, αφού ο συναγωνισμός των νημάτων για τα κλειδώματα είναι πιο κοινός απ΄ ότι σε μικρούς αριθμούς νημάτων. Στη συνέχεια χρησιμοποιούμε fine-grained locking στον πίνακα visited. Δηλαδή για κάθε κορυφή του γράφου, χρησιμοποιούμε και το αντίστοιχο κλείδωμα. Στο σχήμα 3.3 παρουσιάζεται η σύγκριση των δύο αυτών υλοποιήσεων.



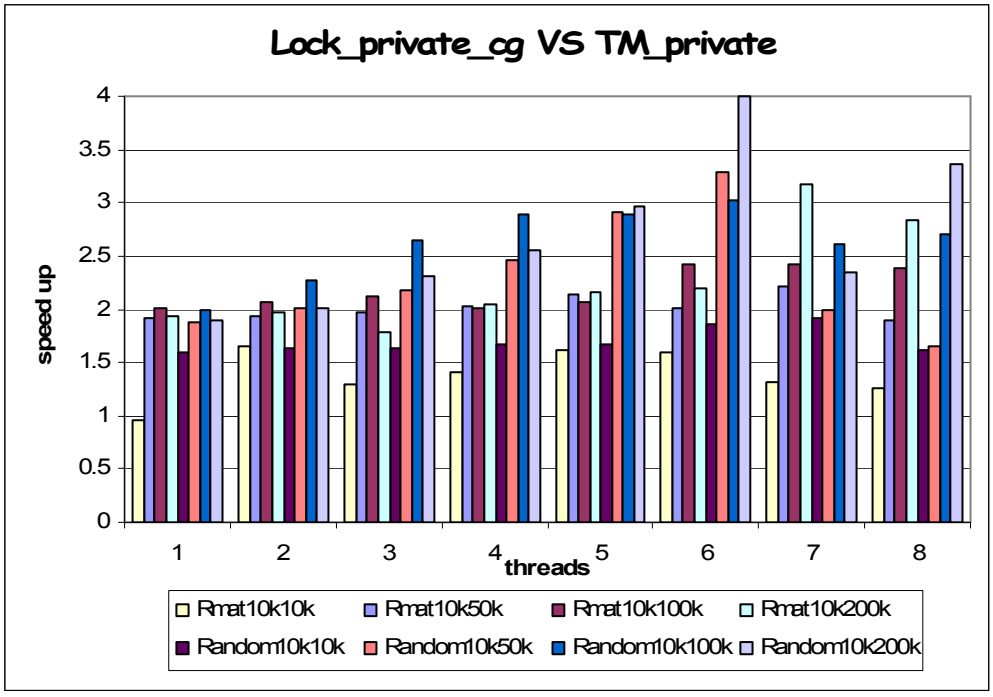
Σχήμα 3.3: Κανονικοποιημένη επιτάχυνση της Lock_private_fg ως προς Lock_cg (Lock_private_fg/Lock_private_cg)

Με αυτόν τον τρόπο, επιτρέπουμε σε πολλαπλά νήματα να μπορούν να διαβάσουν και να γράψουν στον πίνακα visited ταυτόχρονα, αν αυτά αναφέρονται σε διαφορετικές κορυφές. Συγκρίνοντας την νέα υλοποίηση παρατηρούμε βελτίωση σε σχέση με την coarse-grained locking υλοποίηση για μεγάλο αριθμό νημάτων, ενώ για μικρό αριθμό νημάτων παρατηρούμε ότι το coarse-grained locking είναι καλύτερο από το fine-grained locking. Για μεγάλο αριθμό

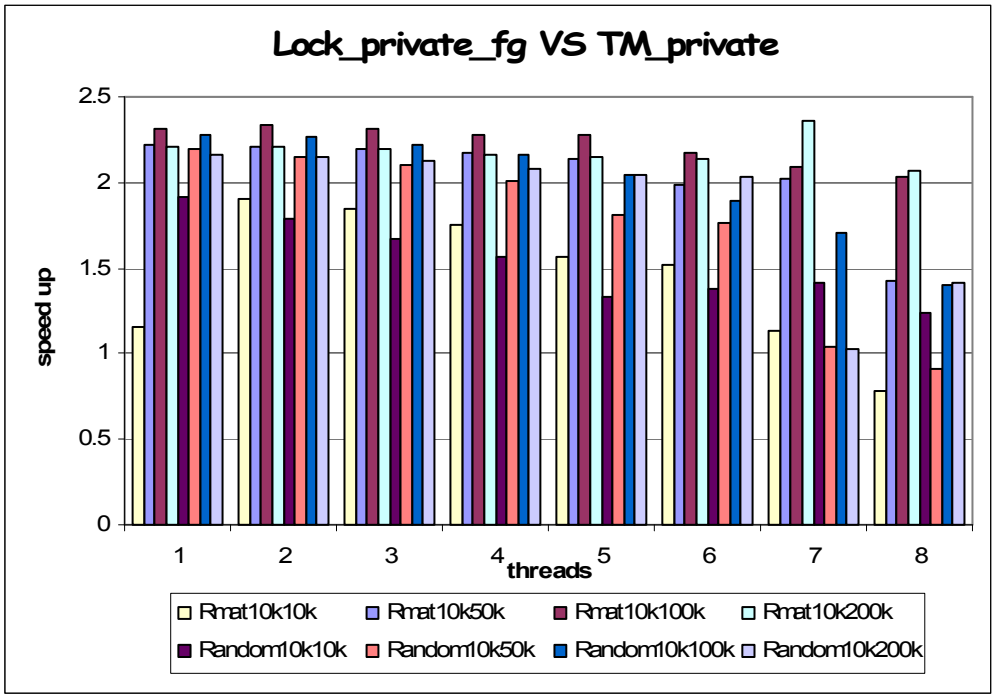
νημάτων υπάρχει μεγαλύτερος συναγωνισμός για τις κορυφές του πίνακα visited, οπότε το fine-grained locking υπερτερεί, αφού επιτρέπει στα νήματα να διαβάσουν και να τροποποιήσουν τον πίνακα visited, εκτός εάν υπάρχει όντως αναφορά στο ίδιο στοιχείο. Ωστόσο και πάλι η παράλληλη υλοποίηση είναι χειρότερη από την σειριακή. Αντίθετα για μικρό αριθμό νημάτων, όπου ο συναγωνισμός για τα στοιχεία του πίνακα visited είναι μικρός, το overhead που εισάγει το fine-grained locking είναι περιπτό και τελικά το coarse-grained locking αποδίδει καλύτερα. Το overhead οφείλεται στο γεγονός, ότι στο coarse-grained locking το κλείδωμα γίνεται σε στοιχεία του πίνακα lock_fg και όχι σε μια απλή μεταβλητή. Αυτό σε επίπεδο assembly ισοδυναμεί με επιπλέον εντολές, αφού πρέπει να υπολογιστεί το offset του πίνακα για να βρεθεί η διεύθυνση μνήμης στην οποία αναφερόμαστε.

Βλέποντας την άσχημη απόδοση της υλοποίησης με την χρήση κλειδωμάτων είτε coarse-grained είτε fine-grained, παρατηρούμε, ότι ο παράλληλος προγραμματισμός αλγορίθμων που είναι εκ φύσεως σειριακοί είναι δύσκολο να προσφέρει κάποια βελτίωση.

Στη συνέχεια εξετάζουμε το σκεπτικό της παράλληλης υλοποίησης αντικαθιστώντας κάποια κλειδώματα με δοσοληψίες, παρατηρώντας ότι δίνουν αισθητή βελτίωση. Συγκρίνουμε τις υλοποιήσεις Lock_private_cg, Lock_ private_fg με την υλοποίηση TM_private όπως παρουσιάζονται στα σχήματα 3.4 και 3.5. Συγκεκριμένα η υλοποίηση TM_private βελτιώνει την υλοποίηση Lock_private_fg, διότι πλέον δεν υπάρχει η χρήση βιβλιοθηκών pthread_mutex_lock και pthread_mutex_unlock, αλλά ο συγχρονισμός γίνεται με hardware δοσοληψίες, οι οποίες προφανώς προσφέρουν λιγότερο κόστος κύκλων. Η TM εισάγει αισιόδοξο παραλληλισμό, οπότε δίνουμε την ευκαιρία στα νήματα να τρέξουν ταυτόχρονα. Μια δοσοληψία θα αποτύχει, μόνο όταν υπάρχει πραγματική σύγκρουση σε αντίθεση με τις δύο άλλες υλοποιήσεις, όπου έχουμε προληπτικό κλείδωμα για την εκτέλεση κάποιου κρίσιμου τμήματος με κόστος την έλλειψη παραλληλοποίησης και εν τέλει οι δύο υλοποιήσεις να έχουν χειρότερη συμπεριφορά και από την σειριακή εκτέλεση.



Σχήμα 3.4: Επιτάχυνση της TM_private κανονικοποιημένη ως προς Lock_private_cg (TM_private/Lock_private_cg)



Σχήμα 3.5: Επιτάχυνση της TM_private κανονικοποιημένη ως προς Lock_private_cg (TM_private/Lock_private_fg)

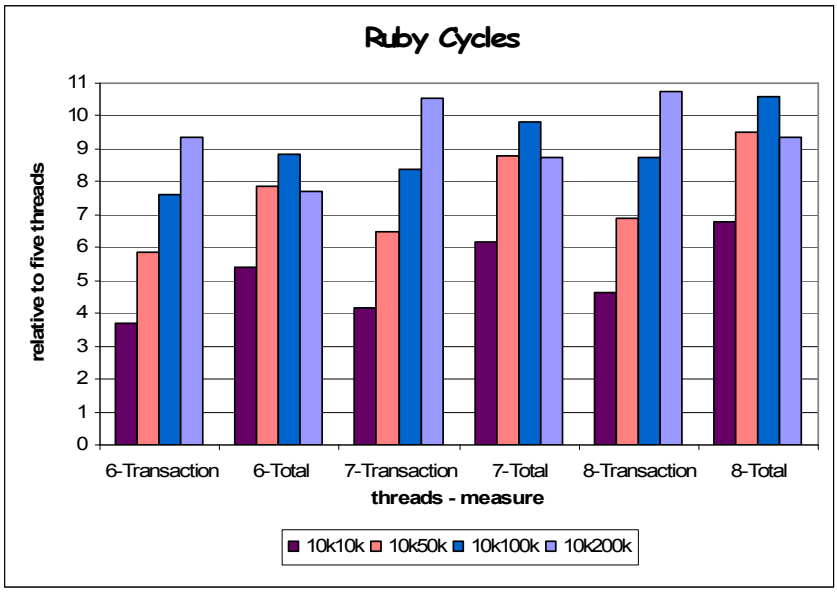
Συγκρίνοντας την απόδοση της υλοποίησης TM_private σε σχέση με την σειριακή υλοποίηση παρατηρούμε ότι καταφέρνει να πετύχει βελτίωση για κάποιον αριθμό νημάτων. Σε γράφους Random η υλοποίηση TM_private είναι μέχρι και 1.4 φορές πιο αποδοτική για αριθμό νημάτων 5 ή 6. Στην περίπτωση των γράφων Rmat αδυνατεί να δώσει βελτίωση, αφού στους γράφους αυτούς υπάρχουν κόμβοι με λίγους γείτονες και γείτονες με αρκετούς γείτονες, οπότε σε αρκετά σημεία του αλγορίθμου δεν μπορεί να εξαχθεί η απαιτούμενη παραλληλοποίηση που χρειάζεται για να έχουμε βελτίωση στην απόδοση της υλοποίησης.

Αυτό ήταν αναμενόμενο, αφού η υλοποίηση στηρίζεται στην παράλληλη εκτέλεση κορυφών του ίδιου επιπέδου, οπότε όσο πιο πυκνός είναι ο γράφος τόσο πιο μεγάλη παραλληλοποίηση επιτυγχάνεται. Η παραλληλοποίηση αυτή είναι εφικτή να αποδώσει χρησιμοποιώντας δοσοληψίες, αφού επιτρέπεται στα νήματα να έχουν πρόσβαση στον πίνακα visited ταυτόχρονα και να μπορούν να τροποποιήσουν τα δεδομένα αυτά χωρίς μεγάλες καθυστερήσεις λόγω συγχρονισμών. Η καθυστέρηση που μπορεί να υπάρξει οφείλεται στις περιπτώσεις που υπάρχει σύγκρουση μεταξύ νημάτων, οπότε και κάποια θα πρέπει να κάνουν abort και να χαθεί η δουλειά που έχουν κάνει μέχρι στιγμής.

Ωστόσο παρατηρούμε ότι η υλοποίηση TM δεν φθάνει στα επιθυμητά επίπεδα παραλληλοποίησης, κάτι το οποίο ήταν επίσης αναμενόμενο. Η παραλληλοποίηση, όπως εξηγήθηκε προηγουμένως στηρίζεται στην παράλληλη επίσκεψη κορυφών του ίδιου επιπέδου. Η παρούσα υλοποίηση δεν μπορεί να λύσει το πρόβλημα συγχρονισμού των νημάτων σε κάθε επίπεδο. Το ότι κάποια νήματα περιμένουν τα υπόλοιπα νήματα να τελειώσουν με το επίπεδο k πριν προχωρήσουν στο επίπεδο k+1, περιορίζει αισθητά την απόδοση που επιτυγχάνεται.

Επίσης μια δεύτερη υλοποίηση με χρήση TM είναι αυτή με την χρήση κοινών ουρών front και back. Η υλοποίηση TM_global_cg παρατηρούμε ότι αποτυγχάνει να δώσει βελτίωση. Για 2

μέχρι 5 νήματα αποδίδει σχεδόν όπως και η σειριακή εκτέλεση, ενώ για 6 μέχρι 8 νήματα παρατηρούμε ότι αποδίδει μέχρι και 0.1 της σειριακής εκτέλεσης. Κοιτάζοντας προσεκτικά τα στατιστικά που εξάγει ο εξομοιωτής παρατηρούμε, ότι στις περιπτώσεις όπου η υλοποίηση αποδίδει πολύ χειρότερα από την σειριακή έχουμε αρκετές δοσοληψίες, οι οποίες αποτυγχάνουν να ολοκληρωθούν, οπότε το σύστημα αναγκάζεται να υποστεί τις καθυστερήσεις που εισάγει η ακύρωση μιας δοσοληψίας όπως η επαναφορά του συστήματος στην κατάσταση που βρισκόταν πριν την έναρξη της δοσοληψίας ή τις καθυστερήσεις που δέχονται οι δοσοληψίες πριν ξαναδοκιμάσουν να εκτελεστούν ελπίζοντας ότι δεν θα βρεθεί πάλι σύγκρουση. Να υπενθυμίσουμε, ότι το LogTM έχει πρόθυμη διαχείριση έκδοσης δεδομένων, οπότε οι εγγραφές γίνονται στις πραγματικές θέσεις κι επομένως υπάρχει μεγάλο κόστος όταν υπάρχει μια δοσοληψία ολοκληρωθεί ανεπιτυχώς (abort). Στο σχήμα 3.6 παρουσιάζουμε τον συνολικό χρόνο εκτέλεσης και τον χρόνο των δοσοληψιών για 6,7,8 νήματα σε σύγκριση με τα αντίστοιχα μεγέθη για 5 νήματα.



Σχήμα 3.6: Transaction / Total Ruby Cycles για TM_global_cg

Εδώ παρατηρούμε για όλα τα μεγέθη γράφων, ότι η άσχημη συμπεριφορά της TM_global _cg οφείλεται στο ότι ο χρόνος εκτέλεσης των δοσοληψιών είναι αρκετά μεγάλος εξαιτίας των αυξημένων συγκρούσεων, κάτι το οποίο φαίνεται και από τον αριθμό ακυρώσεων δοσοληψιών, ο οποίος φαίνεται για τα νήματα 5,6,7,8 στον πίνακα 3.1

Graph/Threads	5	6	7	8
10k10k	4	18177	18684	18691
10k50k	8	97790	98948	98987
10k100k	5	196193	197765	199073
10k200k	6	388496	397673	396971

Πίνακας 3.1: Αριθμός aborts δοσοληψιών

Να σημειώσουμε ότι μετρήσεις και εκτίμηση της συμπεριφοράς της TM_global_cg έγινε μόνο στους γράφους Random, αφού διαπιστώθηκε η άσχημη συμπεριφορά της υλοποίησης. Η υλοποίηση έχει τέτοια συμπεριφορά, αφού μέσα σε κάθε δοσοληψία, υπάρχει προσπάθεια από κάθε νήμα να προσθέσει τους κόμβους του. Χρησιμοποιώντας μια global ουρά υπάρχει

συνεχώς ανταγωνισμός μεταξύ των νημάτων, οπότε το ένα νήμα κάνει commit, ενώ τα υπόλοιπα νήματα υφίστανται καθυστέρηση για να προσπαθήσουν αργότερα να ολοκληρώσουν τις δοσοληψίες τους.

Να σημειώσουμε εδώ ότι έγινε μια μικρή σύγκριση των υλοποιήσεων TM_global_cg και TM_global_fg στον γράφο Random με 10.000 κόμβους και 100.000 ακμές. Η σύγκριση αυτή έδειξε την TM_global_cg να έχει καλύτερη συμπεριφορά από την TM_global_fg περίπου 1.05 – 1.1. Ο λόγος που η TM_global_fg παρουσιάζει ελαφρώς χειρότερη συμπεριφορά είναι το γεγονός ότι το κομμάτι που εκτελείται παράλληλα, όταν δύο νήματα προσπαθούν να προσθέσουν έναν κόμβο στην ίδια ουρά είναι μικρότερο από το overhead που παρουσιάζει η δημιουργία μιας νέας δοσοληψίας. Επομένως δεν κερδίζουμε κάτι με το να χρησιμοποιήσουμε fine-grained δοσοληψία στην προκειμένη περίπτωση. Γι' αυτό και στην ανάλυση αποτελεσμάτων χρησιμοποιούμε μόνο την υλοποίηση TM_global_cg.

Κεφάλαιο 4

Παραμετροποίηση του HTM

4.1 Γενικά

Ο GEMS δίνει την δυνατότητα μέσω των πεδίων **XACT_CONFLICT_RES**, **XACT_LAZY_VM** και **XACT_EAGER_CD** να τροποποιήσουμε την πολιτική αντιμετώπισης των συγκρούσεων, την διαχείριση εκδόσης δεδομένων και τον μηχανισμό εντοπισμού συγκρούσεων αντίστοιχα.

Στατιστικά XACT_BREAKDOWN

Για να γίνει η σύγκριση των διαφορετικών πολιτικών αντιμετώπισης συγκρούσεων καθώς κι επίσης των συστημάτων HTM, χρησιμοποιούμε τα στατιστικά XACT_BREAKDOWN, τα οποία παρέχουν οι μετρήσεις κατά την εξομοίωση του συστήματος στον Simics.

- Non-Trans: κύκλοι που δεν έχουν σχέση με δοσοληψίες
- Trans: κύκλοι που εκτελούνται σε δοσοληψίες
- Stall: κύκλοι καθυστέρησης για να αντιμετωπιστεί μια σύγκρουση
- Backoff: κύκλοι καθυστέρησης μετά από ένα abort για να μειωθεί ο συναγωνισμός
- Commiting: κύκλοι για καθαρισμό του buffer εγγραφής μετά από commit. Θεωρητικά σε συστήματα με πρόθυμη διαχείριση έκδοσης δεδομένων, οι κύκλοι αυτοί είναι μηδέν, αφού κατά το commit, δεν υπάρχει η αντιγραφή από τον buffer.
- Aborting: κύκλοι δοσοληψιών για σκοπούς rollback μετά από κάποιο abort. Θεωρητικά σε συστήματα με οκνηρή διαχείριση έκδοσης δεδομένων, οι κύκλοι αυτοί είναι μηδέν, αφού κατά το abort δεν χρειάζεται να γίνει κάποιο rollback.
- Good: η διαφορά **Aborted = Trans – Good** δίνει μια ένδειξη των κύκλων των δοσοληψιών που χάνεται όταν μια δοσοληψία κάνει abort.

4.2 Πολιτική Αντιμετώπισης Συγκρούσεων

Ο GEMS δίνει την δυνατότητα επιλογής της πολιτική αντιμετώπισης των συγκρούσεων μέσα από τέσσερις επιλογές:

- **BASE:** ο αιτών πυρήνας (requestor) δέχεται NACK από τον συγκρουόμενο πυρήνα (conflicting) και κάνει stall, ενώ αν υπάρχει πιθανότητα για deadlock δηλαδή έχει στείλει NACK σε κάποιον πυρήνα που τρέχει παλαιότερη δοσοληψία, τότε κάνει abort.
- **TIMESTAMP:** ο συγκρουόμενος πυρήνας στέλνει NACK στον αιτών πυρήνα κι ακολούθως ελέγχει αν η δοσοληψία που τρέχει σε αυτόν είναι νεώτερη. Σε περίπτωση που είναι νεώτερη, τότε ο συγκρουόμενος πυρήνας κάνει abort την δοσοληψία του. Σε

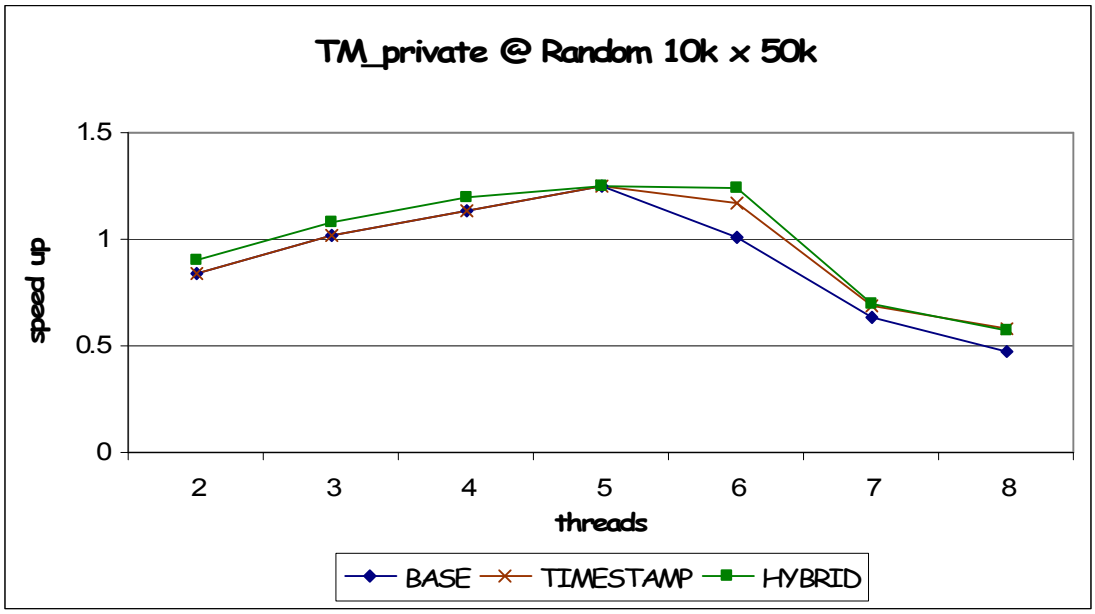
περίπτωση που η νεώτερη δοσοληψία βρίσκεται στον αιτών πυρήνα, τότε ο αιτών πυρήνας κάνει stall την δοσοληψία του.

- **HYBRID:** ο πυρήνας στον οποίο τρέχει η παλαιότερη δοσοληψία κάνει stall εάν η νεώτερη δοσοληψία που τρέχει στον άλλο πυρήνα έχει τροποποιήσει το συγκρουόμενο cache block. Αν η νεώτερη δοσοληψία έχει το συγκρουόμενο cache block μόνο στο σύνολο ανάγνωσης της, τότε γίνεται abort αυτή.
- **CYCLE:** Χρησιμοποιεί διαχειριστή συναγωνισμού υλοποιημένο στο software (software contention manager).

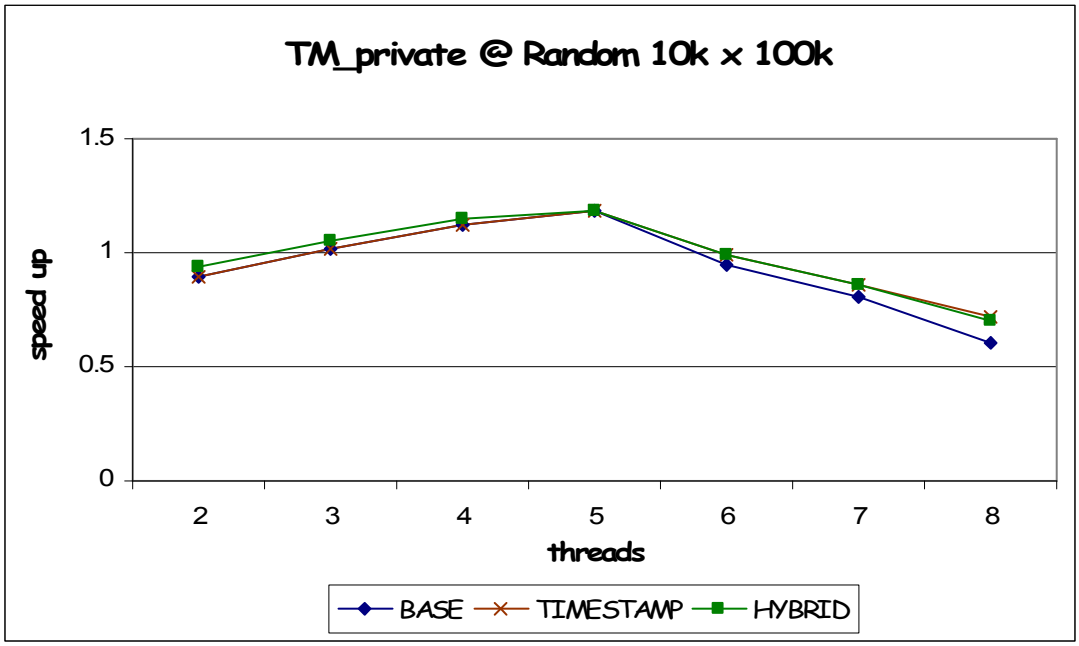
Στην ενότητα αυτή εξετάζουμε τις πολιτικές BASE, TIMESTAMP και HYBRID για την υλοποίηση TM_private για γράφους Random με αριθμό κορυφών 10000 και αριθμό ακμών 50000 και 100000.

Στα σχήματα 4.1 και 4.2 παρουσιάζεται η απόδοση της υλοποίησης για τις παραπάνω στρατηγικές.

Από τα σχήματα 4.1 και 4.2 παρατηρούμε ότι και οι τρεις πολιτικές διαχείρισης συγκρούσεων έχουν ίδια συμπεριφορά για αριθμό νημάτων 2,3,4,5. Από μετρήσεις, που έχουν ληφθεί (δεν παρουσιάζονται στο παρών έγγραφο) παρατηρούμε επίσης ότι ο αριθμός κύκλων, που αναλώνονται σε retries (aborted και stalls) είναι περιορισμένος. Επομένως είναι αναμενόμενο, ότι διαφορετικές πολιτικές διαχείρισης συγκρούσεων δεν επηρεάζουν τα αποτελέσματα, αφού δεν υπάρχουν ακυρώσεις. Ωστόσο με την αύξηση του αριθμού των νημάτων σε 6,7,8 παρατηρούμε ότι οι ακυρώσεις (aborted cycles, xact_aborts) και οι καθυστερήσεις λόγω συγκρούσεων (stall cycles) αυξάνονται. Στο σχήμα 4.3 παρουσιάζεται η μέση τιμή των κύκλων που εκτελούνται από τις δοσοληψίες, οι οποίες κάνουν abort για κάθε abort που υπάρχει για τις τρεις πολιτικές που εξετάζουμε. Στο σχήμα 4.4 παρουσιάζεται το ποσοστό των aborted, stalling, aborting cycles στο overhead λόγω των συγκρούσεων για 6,7,8 νήματα (αριστερά προς δεξιά για κάθε ομάδα)

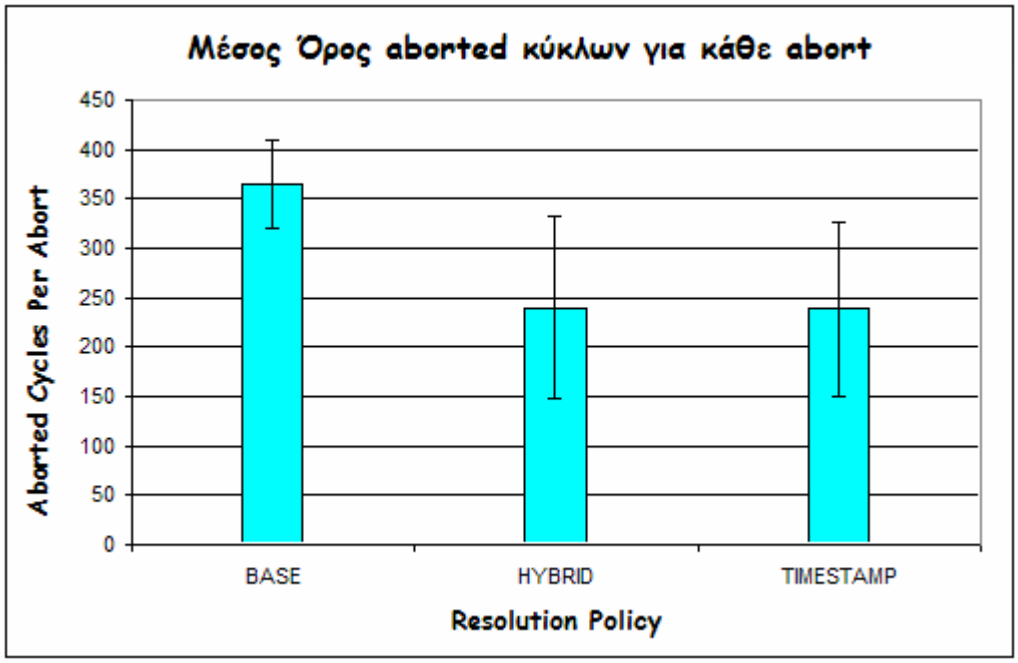


Σχήμα 4.1: Πολιτικές Αντιμετώπισης Συγκρούσεων για γράφο 50000 ακμών



Σχήμα 4.2: Πολιτικές Αντιμετώπισης Συγκρούσεων για γράφο 100000 ακμών

Στο σχήμα 4.3 παρουσιάζουμε τον μέσο όρο κύκλων, που έχει ήδη εκτελέσει μια δοσοληψία αλλά χάνονται επειδή αυτή γίνεται abort. Διαισθητικά ο λόγος aborted cycles per abort χρησιμοποιείται, για να παρουσιάσουμε μια ένδειξη του πόσο προχωρημένες δοσοληψίες ακυρώνονται από τον μηχανισμό διαχείρισης συγκρούσεων.



Σχήμα 4.3: Μέση τιμή των aborted κύκλων όταν μια δοσοληψία ακυρώνεται

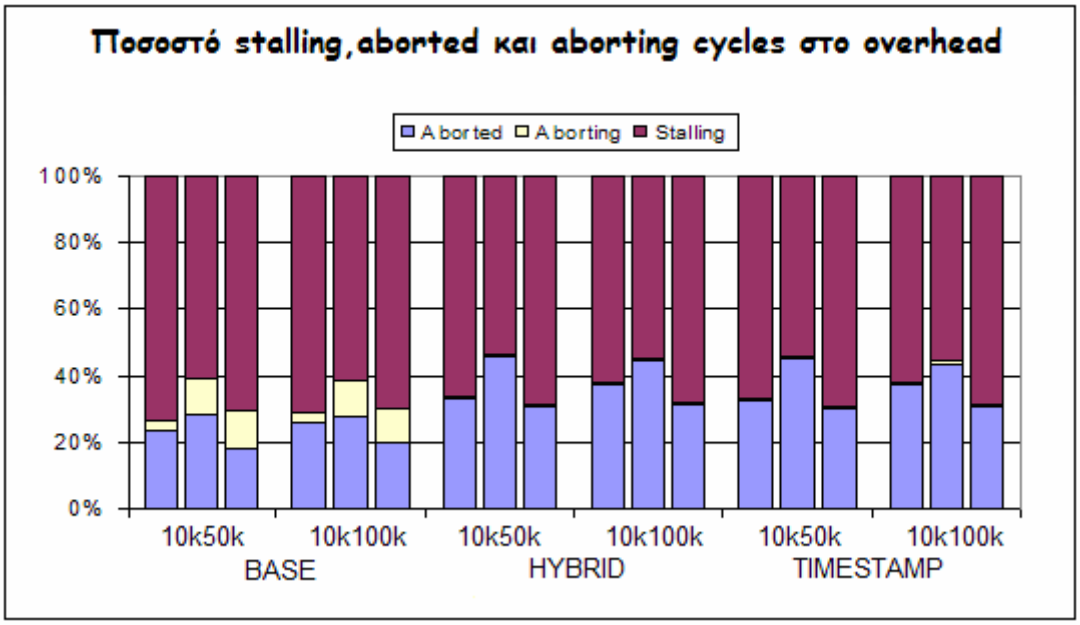
Από το σχήμα 4.3 παρατηρούμε ότι για την υλοποίηση μας η πολιτική BASE ακυρώνει δοσοληψίες, οι οποίες έχουν εκτελέσει περισσότερους κύκλους από τις αντίστοιχες στις δύο άλλες πολιτικές. Οι πολιτικές HYBRID και TIMESTAMP έχουν σχεδόν την ίδια συμπεριφορά

τόσο στον αριθμό κύκλων που χάνονται από ένα abort μιας δοσοληψίας όσο και την απόδοση των δύο υλοποιήσεων σε σχέση με την σειριακή, η οποία παρουσιάζεται στα σχήματα 4.1 και 4.2 για τους δύο γράφους.

Στον πίνακα 4.1 παρουσιάζουμε τον αριθμό των κύκλων overhead, που χάνονται λόγω καθυστερήσεων (stalling) και ακυρώσεων (aborting για rollback και aborted για τους κύκλους της δοσοληψίας που έχουν εκτελεστεί), ενώ στο σχήμα 4.4 παρουσιάζουμε το ποσοστό των επί μέρους μετρικών στο συνολικό overhead.

BASE		
threads	**Random 10k50k**	**Random 10k100k**
6	5693742	8855189
7	26094047	33478717
8	39700013	57476433
HYBRID		
threads	**Random 10k50k**	**Random 10k100k**
6	5967518	8971241
7	25046304	32014852
8	35292437	53437596
TIMESTAMP		
threads	**Random 10k50k**	**Random 10k100k**
6	6593366	9256835
7	26445806	34296855
8	35685512	53538395

Πίνακας 4.1: overhead λόγω συγκρούσεων



Σχήμα 4.4: aborted, stalling, aborting cycles στο overhead λόγω των συγκρούσεων

Από το σχήμα 4.4 παρατηρούμε ότι η υλοποίηση μας έχει μεγαλύτερο ποσοστό aborted cycles στις πολιτικές HYBRID και TIMESTAMP απ' ότι στην πολιτική BASE και αντίστροφα για τα stalling και aborting cycles.

Οι παραπάνω παρατηρήσεις ήταν αναμενόμενες, αφού στην υλοποίηση μας χρησιμοποιούμε δοσοληψίες για να τροποποιήσουμε τον πίνακα ν κι επίσης δοσοληψίες για να διαβάσουμε κάποιο στοιχείο του πίνακα αυτού. Αυξανομένου των νημάτων, αυξάνονται οι συγκρούσεις, αφού ο συναγωνισμός για τον πίνακα ν είναι μεγαλύτερος, κάτι το οποίο φαίνεται και από το overhead για 6,7,8 νήματα που παρουσιάζεται στον πίνακα 4.1.

Παρατηρούμε, ότι η πολιτική BASE έχει μεγαλύτερο overhead από τις πολιτικές HYBRID και TIMESTAMP, ενώ η TIMESTAMP έχει μεγαλύτερο από την HYBRID. Γι' αυτόν τον λόγο η HYBRID παρουσιάζει καλύτερο speed up από την BASE και ελαφρώς καλύτερο από την TIMESTAMP.

Συγκεκριμένα η πολιτική BASE παρουσιάζει την χειρότερη συμπεριφορά από τις τρεις πολιτικές που εξετάζονται. Η πολιτική αυτή κάνει stall την δοσοληψία που ζητά την ανίχνευση και σε περίπτωση που υπάρχει περίπτωση deadlock μόνο τότε την κάνει abort. Επομένως ο αριθμός των aborts κι επομένως των aborted κύκλων είναι μικρός, ενώ ο αριθμός των stalling cycles είναι μεγάλος. Ο λόγος που ο αριθμός των aborting cycles είναι μεγάλος οφείλεται στο γεγονός ότι η πολιτική αυτή δεν ευνοεί κάποια δοσοληψία που έχει τροποποιήσει κάποιο cache block, οπότε απορρίπτει αρκετές παλαιότερες δοσοληψίες που έχουν τροποποιήσει κάποιο cache block κι επομένως το rollback αυτών κοστίζει αρκετά.

Αντίθετα η πολιτική HYBRID, η οποία ήταν κι αυτή που χρησιμοποιήθηκε στις μετρήσεις μας στο κεφάλαιο 3 έχει ελαφρώς την καλύτερη συμπεριφορά. Η πολιτική αυτή έχει την τάση να ακυρώνει τις νεώτερες δοσοληψίες που έχουν διαβάσει την συγκρουόμενη διεύθυνση μνήμης. Ως εκ τούτου αντί να ακυρώνεται αυτή που έχει τροποποιήσει το block μνήμης, ακυρώνονται όλες οι νεώτερες, οι οποίες προσπαθούν να το διαβάσουν. Επομένως έχουμε αρκετά aborts εν αντιθέσει με την BASE. Ωστόσο ο αριθμός των aborting cycles είναι μικρός, αφού το rollback που χρειάζεται μια νέα δοσοληψία έχει μικρότερο κόστος από μια παλαιότερη. Ο αριθμός των stalling cycles είναι μικρότερος από αυτόν στην πολιτική BASE, αφού στην περίπτωση της πολιτικής HYBRID, stall έχουμε μόνο, όταν μια παλαιότερη προσπαθεί να διαβάσει κάποιο τροποποιημένο block μνήμης.

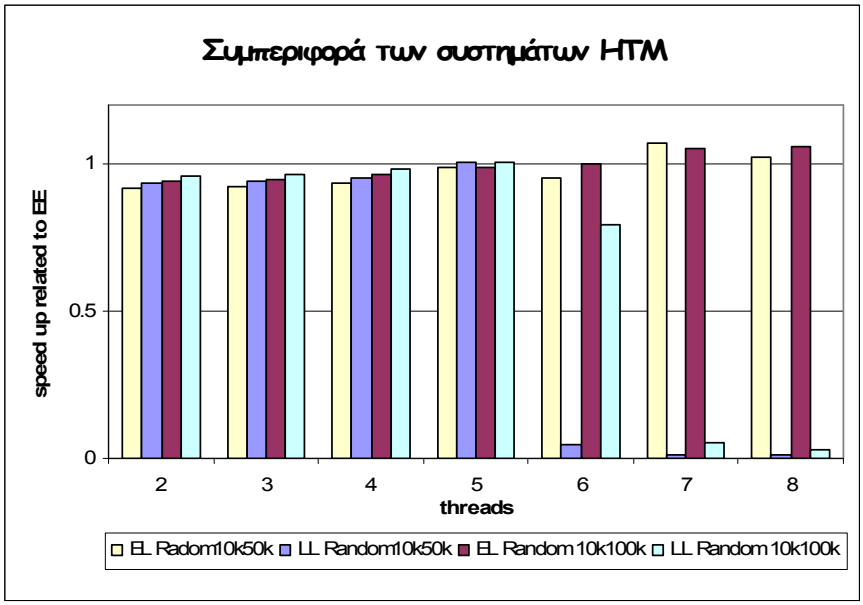
Η πολιτική TIMESTAMP ακυρώνει την αιτούσα δοσοληψία εάν είναι νεώτερη από αυτήν στην οποία έχει ανιχνευθεί η σύγκρουση, ενώ καθυστερεί την νεώτερη αν αυτή έχει εντοπίσει την σύγκρουση. Με απλά λόγια, οι παλαιότερες δοσοληψίες δεν ακυρώνονται. Κοιτάζοντας το είδος δοσοληψιών που χρησιμοποιούμε στην υλοποίηση μας, παρατηρούμε ότι πιθανές συγκρούσεις οφείλονται όταν μια νεώτερη δοσοληψία διαβάζει κάτι το οποίο έχει τροποποιηθεί από μια παλαιότερη. Αυτό έχει σαν αποτέλεσμα η TIMESTAMP να έχει ελαφρώς παρόμοια συμπεριφορά με την HYBRID.

4.3 Παραμετροποίηση VM και CD

Στην ενότητα αυτή θα εξετάσουμε δύο ακόμη συστήματα HTM, τα οποία υποστηρίζουν:

- οκνηρή διαχείριση εκδόσεων δεδομένων και οκνηρό εντοπισμό συγκρούσεων (LL)
- οκνηρή διαχείριση εκδόσεων δεδομένων και πρόθυμο εντοπισμό συγκρούσεων (EL)

και στην συνέχεια θα γίνει σύγκριση αυτών με το σύστημα EE (LogTM) . Η σύγκριση θα γίνει διατηρώντας σαν πολιτική διαχείρισης συγκρούσεων HYBRID για το LogTM, TIMESTAMP για το σύστημα EL και BASE για το σύστημα LL χωρίς χρήση BACKOFF. Ένα σύστημα δύναται να έχει μόνο πολιτική BASE, ενώ ένα σύστημα EL είτε BASE είτε TIMESTAMP. Για το σύστημα EL επιλέγουμε το TIMESTAMP, αφού στο σύστημα EE έχει δείξει καλύτερη συμπεριφορά κάτι το οποίο αναμένουμε και για το σύστημα EL. Για το σύστημα EE επιλέγουμε το HYBRID, αφού αυτό είχε την καλύτερη συμπεριφορά. Στο σχήμα 4.3 παρουσιάζουμε την συμπεριφορά των διαφόρων συστημάτων HTM για την TM_private κανονικοποιήμενη ως προς την συμπεριφορά του EE.



Σχήμα 4.3: Συμπεριφορά των συστημάτων HTM για την υλοποίηση TM_private

Για να καταλάβουμε γιατί ένα σύστημα HTM υπερτερεί απέναντι στα υπόλοιπα για συγκεκριμένο αριθμό νημάτων και γράφο μελετούμε και πάλι τα στατιστικά XACT_BREAKDOWN. Στον πίνακα 4.2 παρουσιάζουμε το overhead των τριών συστημάτων. Για το σύστημα EE το overhead είναι ακριβώς το ίδιο με το αντίστοιχο στον Πίνακα 4.1, ενώ για το σύστημα EL, είναι το άθροισμα των stalling, aborted και επιπλέον commiting και backoff κύκλων. Για το σύστημα LL είναι το άθροισμα των stalling, aborted και επιπλέον commiting και aborting. Να σημειώσουμε εδώ ότι θεωρητικά το σύστημα LL δεν έπρεπε να παρουσιάσει aborting κύκλους, αφού όπως εξηγήθηκε στην ενότητα 4.1 συστήματα με οκνηρή διαχείριση έκδοσης δεδομένων δεν χρειάζεται να κάνουν rollback μετά από κάποιο abort.

Lazy CD – Lazy VM – BASE		
threads	**Random 10k50k**	**Random 10k100k**
6	1598580376	100520386
7	11264550279	3824433858
8	24295247900	10134207436
Eager CD - Eager VM – HYBRID		
threads	**Random 10k50k**	**Random 10k100k**
6	5967518	8971241
7	25046304	32014852
8	35292437	53437596

Eager CD – Lazy VM - TIMESTAMP		
threads	**Random 10k50k**	**Random 10k100k**
6	8596316	13207239
7	33762527	43279729
8	51896812	74856162

Πίνακας 4.2: overhead λόγω συγκρούσεων

Ανάλυση για το σύστημα Eager CD - Lazy VM – TIMESTAMP

Στο σύστημα EL με πολιτική διαχείρισης συγκρούσεων TIMESTAMP παρατηρούμε, ότι δεν υπάρχουν κύκλοι aborting κι αυτό ήταν αναμενόμενο, αφού ακολουθούμε οκνηρή διαχείριση έκδοσης δεδομένων, οπότε δεν υπάρχει το κόστος του rollback όταν μια δοσοληψία κάνει abort. Με την αύξηση των νημάτων, παρατηρούμε ότι αυξάνονται οι συγκρούσεις, αφού η διαφορά των ποσοστών trans και good trans αυξάνεται (αυτό σημαίνει ότι αυξάνεται ο αριθμός των δοσοληψιών που κάνουν abort). Παράλληλα αυξάνεται και το κόστος backoff, αφού αυξανομένου του αριθμού των aborts αυξάνονται και οι καθυστερήσεις, που εισάγει το σύστημα στην δοσοληψία μετά το abort της.

Ανάλυση για το σύστημα Lazy CD - Lazy VM – BASE

Στο σύστημα LL με πολιτική διαχείρισης συγκρούσεων BASE παρατηρούμε, ότι με την αύξηση των νημάτων, παρατηρούμε ότι αυξάνονται οι συγκρούσεις, αφού η διαφορά των ποσοστών trans και good trans αυξάνεται (αυτό σημαίνει ότι αυξάνεται ο αριθμός των δοσοληψιών που κάνουν abort). Για έξι και επτά νήματα, οι διαφορές αυτές εκτοξεύονται σε ~50% και ~83%, κι αυτό οφείλεται στο γεγονός ότι το σύστημα LL χρησιμοποιεί οκνηρό εντοπισμό συγκρούσεων οπότε οι συγκρούσεις ανιχνεύονται, όταν μια δοσοληψία προσπαθήσει να κάνει commit. Η πολιτική BASE απορρίπτει την δοσοληψία που έχει ζητήσει την ανίχνευση σύγκρουσης, στην προκειμένη περίπτωση η δοσοληψία που κάνει commit. Επομένως ο συνδυασμός αυτός είναι αναμενόμενο, ότι θα αυξήσει τον αριθμό των κύκλων που χάνονται μετά από abort, κάτι το οποίο φαίνεται και από τα στατιστικά του πίνακα 4.4.

Συμπεριφορά των συστημάτων

Τα συστήματα EL, LL έχουν καθαρά χειρότερη συμπεριφορά από το σύστημα EE για αριθμό νημάτων μέχρι πέντε, αφού όπως βλέπουμε στους σχετικούς πίνακες, ο αριθμός των δοσοληψιών που κάνουν abort είναι μηδενικός κι επομένως το EE ευνοείται, αφού όπως αναφέρθηκε προηγουμένως τα συστήματα με πρόθυμη διαχείριση έκδοσης δεδομένων έχουν γρήγορα commits και αργά aborts σε αντίθεση με αυτά που έχουν οκνηρή, τα οποία έχουν αργά commits, αφού πρέπει να υποστούν το overhead του commit, δηλαδή να μεταφέρουν τα δεδομένα του buffer εγγραφής στις πραγματικές τους θέσεις. Το EE υποφέρει, όταν υπάρχουν πολλά aborts, αφού χρειάζεται να γίνει rollback. Εν αντιθέσει το EL πλεονεκτεί απέναντι στο EE, αφού δεν χρειάζεται να υποστεί το κόστος του rollback, αφού χρησιμοποιεί οκνηρή διαχείριση έκδοσης δεδομένων. Ωστόσο το EL μειονεκτεί απέναντι στο EE, επειδή τα commits σε οκνηρή VM κοστίζουν, αφού πρέπει να γίνει μεταφορά των δεδομένων από τους buffers στην μνήμη. Στην συγκεκριμένη υλοποίηση TM_private το overhead που εισάγει το σύστημα EL μετά από μια σύγκρουση είναι μεγαλύτερο απ' ότι στην περίπτωση του συστήματος EE. Όπως εξηγήσαμε και στην προηγούμενη ενότητα, τα aborts γίνονται σε νεώτερες δοσοληψίες οι οποίες έχουν διαβάσει το συγκρουόμενο block, οπότε το rollback δεν κοστίζει αρκετά. Στο σύστημα EL το αντίστοιχο penalty με το rollback, δηλαδή το commiting έχει μεγαλύτερο κόστος απ' ότι το rollback στο EE, ενώ επίσης το BACKOFF για αποφυγή του φαινομένου livelock (δύο δοσοληψίες ακυρώνονται συνεχώς και δεν ολοκληρώνεται ποτέ η εφαρμογή) κοστίζει αρκετά.

Παρατηρούμε ότι το EE για αριθμό νημάτων 7,8 συμπεριφέρεται καλύτερα από το EL. Αυτό οφείλεται στο γεγονός ότι οι Non_Trans κύκλοι στο EE είναι περισσότεροι από ότι στο EL, οπότε το speed up του EL είναι καλύτερο. Η διαφορά στους Non_Trans κύκλους οφείλεται στις παρενέργειες που οφείλονται στην δρομολόγηση. Ο τρόπος διάσχισης του γράφου εξαρτάται από την σειρά που γίνονται commit οι δοσοληψίες μας. Σε κάθε σύστημα έχουμε δδιαφορετική σειρά commits των δοσοληψιών, οπότε τελικά δημιουργείται διαφορετικό path εκτέλεσης της διάσχισης του γράφου, όποτε διαφορετικοί Non_Trans κύκλοι.

Επίλογος ενότητας

Στην υλοποίηση TM_private είναι προφανές ότι το EE δίνει καλύτερα αποτελέσματα από τα άλλα για δύο μέχρι και έξι νήματα. Είναι προφανές ότι κάποιο σύστημα έχει καλύτερη συμπεριφορά από ένα άλλο σύστημα σε κάποιες περιπτώσεις, ενώ σε κάποιες άλλες χειρότερη.

Οι συγγραφείς της αναφοράς [23] παρουσιάζουν λεπτομερής ανάλυση αποτελεσμάτων εκτέλεσης διαφόρων benchmarks στα συστήματα HTM, που χρησιμοποιήσαμε κι εμείς στην παρούσα ενότητα. Συστήματα LL υποφέρουν από το φαινόμενο RestartConvoy, στο οποίο μια δοσοληψία συγκρούεται με άλλα στιγμιότυπα της ίδιας δοσοληψίας, τα οποία ακυρώνονται. Το ίδιο συνεχίζει μεταξύ των υπολοίπων στιγμιότυπων, οπότε τελικά έχουμε σειροποίηση των δοσοληψιών. Αντίθετα τα συστήματα EE και EL υποφέρουν από το φαινόμενο FriendlyFire, όπου οι δοσοληψίες ακυρώνουν κάποιες άλλες και στην συνέχεια ακυρώνονται και αυτές, οπότε δεν υπάρχει κάποια σημαντική πρόοδος. Αρκετά άλλα φαινόμενα υπάρχουν στα συστήματα EE, EL και LL, τα οποία αναλύονται πλήρως στην αναφορά [23] της βιβλιογραφίας.

Συμπέρασμα είναι ότι δεν υπάρχει ένα τέλειο σύστημα HTM, το οποίο να έχει την ίδια συμπεριφορά κι αυτό είναι ένα πρόβλημα, αφού σε περίπτωση υλοποίησης κάποιου είδους συστήματος σε εμπορικό επεξεργαστή, αυτό θα συμπεριφέρεται καλά σε κάποιες εφαρμογές, ενώ σε άλλες θα υποφέρει από χαμηλή απόδοση

Κεφάλαιο 5

Επίλογος – Μελλοντική Εργασία

Στα πλαίσια της διπλωματικής εργασίας έγινε αναφορά στην Transactional Memory σαν προγραμματιστικό μοντέλο πολυνηματικού προγραμματισμού. Επίσης χρησιμοποιήσαμε διαφόρους τρόπους για να παραλληλοποιήσουμε τον αλγόριθμο διάσχισης γράφων Breadth First Search. Ο αλγόριθμος αυτός είναι γνωστό ότι είναι δύσκολο να παραλληλοποιηθεί, αφού σε κάθε επανάληψη χρειάζεται να γίνει πρόσβαση στον ίδιο πίνακα, ενώ σε ουρά να προστεθούν οι γείτονες της εξεταζόμενης κορυφής που δεν έχουν ήδη επισκεφθεί. Σαν λύση στο πρόβλημα αναφοράς στην ίδια ουρά ήταν να χρησιμοποιήσαμε τοπικές ουρές για το κάθε νήμα και να προσπαθήσουμε να χρησιμοποιήσουμε load-balancing, ώστε τα νήματα να έχουν σχεδόν ίση εργασία. Ωστόσο παρατηρήσαμε ότι οι fine-grained και coarse-grained υλοποιήσεις δεν μπορούσαν να πετύχουν κάποια βελτίωση στην παράλληλε εκτέλεση της υλοποίησης. Για να χειριστούμε την χαμηλή αυτή απόδοση, χρησιμοποιήσαμε Transactional Memory, η οποία βασίζεται σε αισιόδοξο παραλληλισμό, επιτρέποντας τα νήματα να εκτελούνται ταυτόχρονα και στα κρίσιμα τους τμήματα. Η Transactional Memory είναι ικανή να δώσει βελτίωση σε αρκετές περιπτώσεις. Ωστόσο είναι αδύνατον να δώσει αρκετή βελτίωση οι «barriesr» που χρειάζονται για να συγχρονίσουν τα νήματα σε κάθε επίπεδο δεν μπορούν να αποφευχθούν. Με ένα καλύτερο load-balancing είναι αρκετά πιθανό να επιτύχουμε καλύτερη βελτίωση στον αλγόριθμο κι αυτό αποτελεί σαν μελλοντική επέκταση αυτή της διπλωματικής εργασίας.

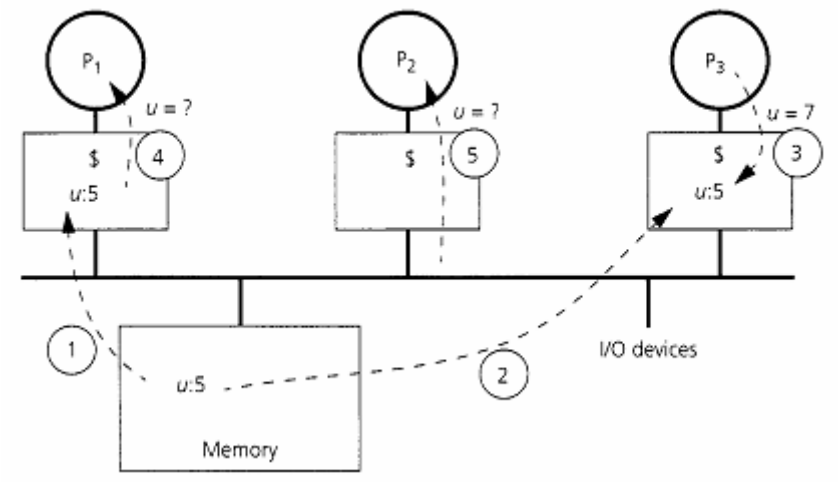
Επίσης στα πλαίσια της διπλωματικής εργασίας εξομοιώθηκαν συστήματα TM, τα οποία υποστηρίζουν διαφορετικούς μηχανισμούς εντοπισμού και διαχείρισης των συγκρούσεων καθώς και διαφορετικό μηχανισμό στην έκδοση δεδομένων. Εξετάστηκε ο αλγόριθμος BFS και παρατηρήσαμε ότι πρόθυμη διαχείριση συγκρούσεων, πρόθυμη διαχείριση έκδοσης δεδομένων αποδίδει καλύτερα.

Τελικό συμπέρασμα είναι ότι όντως η Hardware Transactional Memory αποτελεί ένα προγραμματιστικό μοντέλο, το οποίο μπορεί να ευκολύνει τον πολυνηματικό προγραμματισμό και ταυτόχρονα να δώσει βελτίωση στην απόδοση της εκτέλεσης αρκετών αλλά όχι όλων των πολυνηματικών εφαρμογών. Αυτό οφείλεται στο γεγονός ότι στην απόδοση μιας εφαρμογής παίζουν σημαντικό ρόλο τα ιδιαίτερα χαρακτηριστικά τη. Για συγκεκριμένα “patterns” πολυνηματικών εφαρμογών, κάθε σύστημα HTM έχει συγκεκριμένες παθογένειες, όπως παρουσιάζονται και στην σχετική αναφορά της βιβλιογραφίας [23]. Αυτό αποτελεί μεγάλο πρόβλημα εμπορικής υλοποίησης ενός επεξεργαστή που υποστηρίζει HTM, αφού θα πρέπει ο κατασκευαστής να διαλέξει κάποιο σύστημα HTM, το οποίο όμως δεν είναι τέλειο για όλες τις πολυνηματικές εφαρμογές.

Παράρτημα

Συνεκτικότητα Κρυφής Μνήμης (Cache Coherence)[5]

Σε πολυεπεξεργαστές μοιραζόμενης μνήμης, όπου ο κάθε ένας έχει τοπικό επίπεδο κρυφής υπάρχει το πρόβλημα συνεκτικότητας κρυφής μνήμης. Το πρόβλημα αυτό εμφανίζεται, όταν αντίγραφο του ίδιου μπλοκ μνήμης υπάρχει στο τοπικό επίπεδο της κρυφής μνήμης δύο ή περισσότερων επεξεργαστών. Στη συνέχεια κάποιος επεξεργαστής τροποποιεί το μπλοκ της μνήμης αυτό, και αν δεν ληφθεί κάποια λύση το αντίγραφο που υπάρχει στα τοπικά επίπεδα της κρυφής μνήμης των υπολοίπων επεξεργαστών έχει την λάθος τιμή. Ένα σύστημα μνήμης είναι συνεκτικό, όταν οποιαδήποτε εκτέλεση του παράλληλου προγράμματος έχει τα ίδια αποτελέσματα οποιαδήποτε χρονική στιγμή με αυτά της υποθετικής σειριακής εκτέλεσης του προγράμματος. Το πρόβλημα συνεκτικότητας κρυφής μνήμης παρουσιάζεται αναλυτικά με το παρακάτω παράδειγμα. Στο σχήμα Π.1 έχουμε έναν πολυεπεξεργαστή που αποτελείται από τρεις επεξεργαστές με τοπική κρυφή μνήμη και έναν διάδρομο που ενώνει τις κρυφές μνήμες των επεξεργαστών με την κύρια μνήμη. Αυτό το σχήμα μπορεί να θεωρηθεί παρόμοιο με την περίπτωση ενός πολυπύρηνου επεξεργαστή, όπου τον ρόλο της κύριας μνήμης διαδραματίζει η κρυφή μνήμη L2, εάν αυτή είναι μοιραζόμενη.



Σχήμα Π.1: Παράδειγμα προβλήματος συνεκτικότητας κρυφής μνήμης

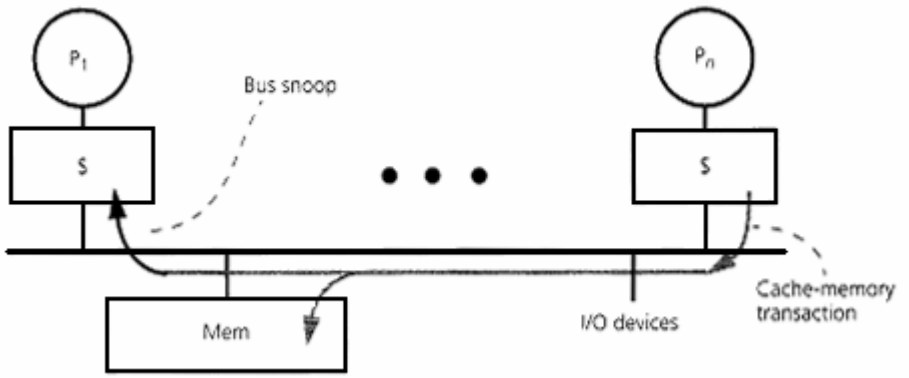
Θεωρούμε ότι ο επεξεργαστής P1 διαβάζει πρώτα την τιμή του u από την κύρια μνήμη κι ακολούθως το ίδιο και ο επεξεργαστής P3. Πλέον οι επεξεργαστές P1 και P3 έχουν αντίγραφο του μπλοκ μνήμης που περιέχει την τιμή του u. Στη συνέχεια ο επεξεργαστής P3 τροποποιεί

[5] Το παρών κομμάτι του παραρτήματος στηρίζεται στην αναφορά [21] της βιβλιογραφίας

την τιμή του u. Χρησιμοποιώντας write-through πολιτική, η νέα τιμή του u γράφεται και στην κύρια μνήμη. Ωστόσο ο επεξεργαστής P1 διαβάζοντας εκ νέου την τιμή του u, θα διαβάσει την παλιά τιμή, αφού αυτή υπάρχει στην κρυφή του μνήμη και δεν πρόκειται να την ψάξει από την κύρια μνήμη. Χρησιμοποιώντας write-back πολιτική, η νέα τιμή του u δεν γράφεται στην κύρια μνήμη αμέσως, αλλά μόνο όταν το αντίστοιχο μπλοκ της κρυφής μνήμης του P3 αντικατασταθεί. Ως εκ τούτου το πρόβλημα με τον επεξεργαστή P1 παραμένει, αλλά παρουσιάζεται και πρόβλημα με τον επεξεργαστή P2. Στην περίπτωση, που ο επεξεργαστής P2 ζητήσει την τιμή του u, θα την βρει στην κύρια μνήμη, η οποία όμως δεν είναι σωστά ενημερωμένη λόγω της write-back πολιτικής.

Συνεκτικότητα κρυφής μνήμης με πρωτόκολλο παρακολούθησης

Οι επεξεργαστές με τοπικές κρυφές μνήμες τοποθετούνται πάνω σε έναν κοινό διάδρομο όπως φαίνεται και στο σχήμα Π.2. Σε κάθε κρυφή μνήμη υπάρχει μια μονάδα, η οποία παρακολουθεί συνεχώς τον διάδρομο (bus-snooping). Σε περίπτωση που υπάρξει κάποια τροποποίηση ενός μπλοκ, ο επεξεργαστής στέλνει στον διάδρομο μια συναλλαγή (cache-memory transaction). Οι υπόλοιποι επεξεργαστές θα δουν αυτήν την συναλλαγή και αν είναι σχετική με κάποιο αντίγραφο μπλοκ μνήμης που έχουν στην τοπική τους κρυφή μνήμη, αλλάζουν την κατάσταση στην οποία βρίσκεται το αντίγραφο. Ταυτόχρονα και η κύρια μνήμη ενημερώνεται με αυτήν την συναλλαγή, έτσι ώστε να γνωρίζει ποια μπλοκ μνήμης της είναι έγκυρα (αν ακολουθείται write-back πολιτική).



Σχήμα Π.2: Πολυεπεξεργαστής με παρακολούθηση (bus-snooping)

Στο παράδειγμα που παρουσιάστηκε προηγουμένως, ο επεξεργαστής P1 βλέποντας την συναλλαγή που προέρχεται από τον P3 και που σχετίζεται με την εγγραφή της u, θα ακυρώσει το αντίγραφο που έχει στην τοπική του κρυφή μνήμη. Στην συνέχεια όταν ζητήσει να διαβάσει την τιμή της u, θα αποτύχει στην τοπική κρυφή μνήμη και στην περίπτωση write-through πολιτικής θα διαβάσει την τιμή που υπάρχει στην κύρια μνήμη, η οποία είναι και η σωστή.

Συνεκτικότητα κρυφής μνήμης με πρωτόκολλο καταλόγου

Η διαφορά που υπάρχει μεταξύ του πρωτοκόλλου καταλόγου (directory-based) είναι ότι εδώ χρησιμοποιείται crossbar switch. Για παράδειγμα, σε έναν επεξεργαστή, όπου η L2 είναι κοινή για τους πυρήνες, οι πυρήνες επικοινωνούν με την L2 και μεταξύ τους με την χρήση crossbar switch. Η L2 διαδραματίζει τον ρόλο του καταλόγου. Όταν ένας πυρήνας επιθυμεί να διαβάσει ένα block μνήμης, το οποίο δεν βρίσκεται στην κρυφή του μνήμη, επικοινωνεί πρώτα με τον κατάλογο. Ο κατάλογος απαντάει ποιος πυρήνας έχει το ενημερωμένο block κι

ακολούθως ο αιτών πυρήνας επικοινωνεί με τον πυρήνα που έχει το block. Σε περίπτωση που ο πυρήνας θέλει να τροποποιήσει κάποιο block μνήμης το οποίο βρίσκεται σε shared κατάσταση ή δεν υπάρχει στην κρυφή του μνήμη, τότε επικοινωνεί και πάλι με τον κατάλογο για να ζητήσει ποιοι πυρήνες έχουν shared το εν λόγω block. Ακολούθως ο πυρήνας ή ο κατάλογος στέλνει μηνύματα ακύρωσης ενώ οι shared πυρήνες στέλνουν στον αιτών πυρήνα μήνυμα ACK. Παρακάτω περιγράφεται το πρωτόκολλο MOESI. Το πρωτόκολλο MOESI επεκτείνεται από το LogTM για να μπορεί να ανιχνεύει τις συγκρούσεις προσθέτοντας μηνύματα NACK.

Πρωτόκολλο MOESI [6]

Κάθε memory / cache block έχει 5 καταστάσεις, οι οποίες αναλυτικά είναι οι εξής:

- *Άκυρο (Invalid)*. Η κρυφή μνήμη δεν έχει έγκυρο αντίγραφο του δεδομένου. Το έγκυρο αντίγραφο μπορεί να υπάρχει είτε στην κύρια μνήμη, είτε στην κρυφή μνήμη κάποιου άλλου επεξεργαστή.
- *Αποκλειστικό (Exclusive)*. Το δεδομένο της συγκεκριμένης κρυφής μνήμης είναι το σωστό αντίγραφο και το πιο πρόσφατο. Το αντίγραφο στην κύρια μνήμη είναι, επίσης, το πιο πρόσφατο, σωστό αντίγραφο του δεδομένου.
- *Κοινό (Shared)*. Το δεδομένο της συγκεκριμένης κρυφής μνήμης είναι το σωστό αντίγραφο και το πιο πρόσφατο. Οι υπόλοιποι επεξεργαστές, όμως, (όπως και η μνήμη) μπορούν να κρατούν αντίγραφα του δεδομένου γιατί κανείς επεξεργαστής δεν έχει, ακόμα, τροποποιήσει το δεδομένο.
- *Τροποποιημένο (Modified)*. Το δεδομένο της συγκεκριμένης κρυφής μνήμης είναι το σωστό αντίγραφο και το πιο πρόσφατο. Το αντίγραφο που έχει η κύρια μνήμη είναι παλιό (λανθασμένο), ενώ κανένας άλλος επεξεργαστής δε διαθέτει αντίγραφο του δεδομένου στην δική του κρυφή μνήμη.
- *Κατεχόμενο (Owned)*. Η κατάσταση αυτή είναι παρόμοια με την κοινή κατάσταση. Μόνο ένας επεξεργαστής μπορεί να κρατά ένα δεδομένο στην κατεχόμενη κατάσταση (το αντίστοιχο δεδομένο στους υπόλοιπους επεξεργαστές πρέπει να είναι στην κοινή κατάσταση). Αντίθετα, όμως, από την κοινή κατάσταση, το αντίγραφο στην κύρια μνήμη μπορεί να είναι παλιό.

Το διάγραμμα καταστάσεων φαίνεται στο σχήμα Π.3, ενώ παρακάτω ακολουθεί μια περιγραφή του διαγράμματος καταστάσεων.

Ξεκινώντας από την άκυρη κατάσταση, αν ο επεξεργαστής ζητήσει να διαβάσει το δεδομένο, θα αποτύχει, όποτε θα ελέγξει τους υπόλοιπους επεξεργαστές να δει αν κρατάνε κάποιο έγκυρο αντίγραφο στη κρυφή μνήμη τους. Αν βρει έγκυρο αντίγραφο, τότε το παίρνει στην κρυφή μνήμη του και το δεδομένο μεταβαίνει στην κοινή κατάσταση. Αν δεν βρει έγκυρο αντίγραφο, τότε παίρνει το δεδομένο από την κύρια μνήμη και το δεδομένο μεταβαίνει στην αποκλειστική κατάσταση.

Αν από την άκυρη κατάσταση είτε την κοινή είτε την αποκλειστική, μεταβαίνει απευθείας στην τροποποιημένη κατάσταση. Από την τροποποιημένη κατάσταση, όσες αιτήσεις εγγραφής και να γίνουν από τον ίδιο επεξεργαστή το δεδομένο παραμένει στην ίδια κατάσταση.

Όταν το δεδομένο βρίσκεται στην κοινή κατάσταση και γίνει αίτηση εγγραφής για το συγκεκριμένο δεδομένο από το δίαυλο, τότε μεταβαίνει στην άκυρη κατάσταση.

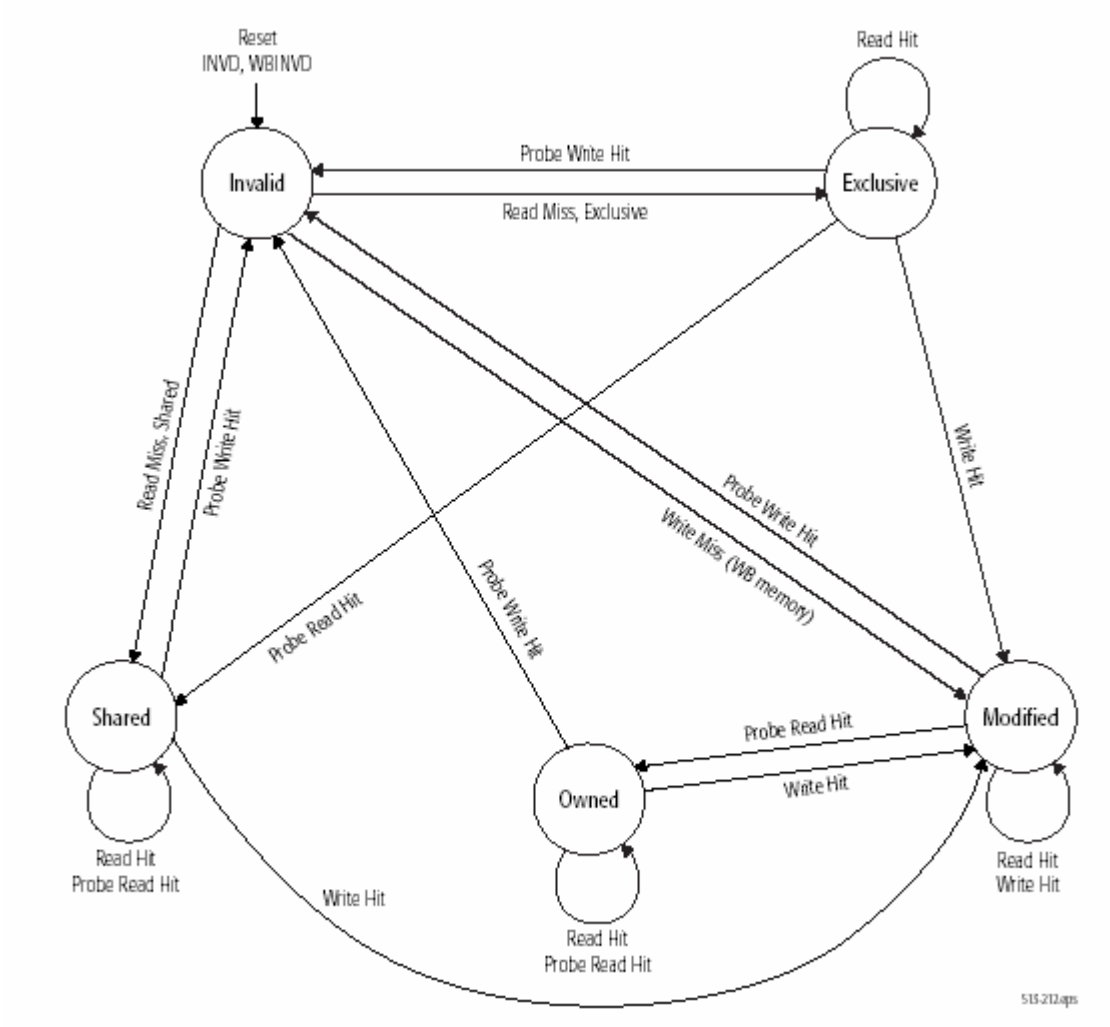
[6] Το παρών κομμάτι του παραρτήματος στηρίζεται στην αναφορά [22] της βιβλιογραφίας

Όταν το δεδομένο βρίσκεται στην αποκλειστική κατάσταση και γίνει μια αίτηση εγγραφής από το δίαυλο, τότε το δεδομένο μεταβαίνει στην άκυρη κατάσταση. Αν γίνει αίτηση ανάγνωσης από το δίαυλο, τότε το δεδομένο μεταβαίνει στην κοινή κατάσταση.

Όταν το δεδομένο βρίσκεται στην τροποποιημένη κατάσταση και δεχθεί μια αίτηση εγγραφής από το δίαυλο, τότε μεταβαίνει στην άκυρη κατάσταση. Αν δεχθεί μια αίτηση ανάγνωσης από το δίαυλο, τότε μεταβαίνει στην κατεχόμενη κατάσταση.

Από την κατεχόμενη κατάσταση, αν το δεδομένο δεχθεί αίτηση εγγραφής από τον ίδιο επεξεργαστή, τότε μεταβαίνει στην τροποποιημένη κατάσταση, ενώ αν δεχθεί αίτηση εγγραφής από το δίαυλο, τότε μεταβαίνει στην άκυρη κατάσταση.

Όταν ένας επεξεργαστής έχει κάποια αποτυχία-ανάγνωσης (read-miss) ή κάποια αποτυχία εγγραφής (write-miss), ελέγχει τους υπόλοιπους επεξεργαστές για να προσδιορίσει εάν το πιο πρόσφατο αντίγραφο του δεδομένου κρατείται σε κάποια κρυφή μνήμη. Αν κάποιος από τους υπόλοιπους επεξεργαστές κρατάει το πιο πρόσφατο αντίγραφο, το δίνει στον επεξεργαστή που το ζήτησε. Αλλιώς το αντίγραφο του δεδομένου παρέχεται από την κύρια μνήμη.



Σχήμα Π.3 Διάγραμμα καταστάσεων του πρωτοκόλλου MOESI.

Πίνακες Αποτελεσμάτων Εξομοίωσης

BASE RESOLUTION POLICY				
threads	**abort**	**stall**	**transactional**	**speed up**
5	0	75	5919647	1.25
6	233754	3629936	14728544	1.01
7	2878445	15778192	30207893	0.63
8	6018703	23297299	39707901	0.47
HYBRID RESOLUTION POLICY				
threads	**abort**	**stall**	**transactional**	**speed up**
5	8	0	5969966	1.25
6	35459	2925678	15816879	1.24
7	147298	13469853	36095260	0.70
8	187595	19171182	45135024	0.57
TIMESTAMP RESOLUTION POLICY				
threads	**abort**	**stall**	**transactional**	**speed up**
5	8	25	5928598	1.25
6	52648	3303118	16694720	1.17
7	231582	14304622	37073248	0.69
8	342943	19481235	45322110	0.58

Πίνακας Π.1: Στατιστικά XACT_BREAKDOWN για γράφο 50000 ακμών

BASE RESOLUTION POLICY				
threads	**abort**	**stall**	**transactional**	**speed up**
5	15	192	8467782	1.18
6	424537	5242347	22015857	0.95
7	3469545	20592159	38736589	0.81
8	8452734	32414432	54341470	0.61
HYBRID RESOLUTION POLICY				
threads	**abort**	**stall**	**transactional**	**speed up**
5	17	67	8474068	1.18
6	55187	3483234	25161072	0.99
7	181687	17538985	45584165	0.89
8	268733	28280524	66895343	0.70
TIMESTAMP RESOLUTION POLICY				
threads	**abort**	**stall**	**transactional**	**speed up**
5	18	67	8474119	1.18
6	85391	3598865	25283508	0.99
7	338374	18957794	47161469	0.86

8	420396	28953709	65731427	0.71

Πίνακας Π.2: Στατιστικά XACT_BREAKDOWN για γράφο 50000 ακμών

threads	non-trans	trans	good trans	aborting	stall
Random 10k50k					
2	89.13	10.87	10.87	0	0
3	90.28	9.72	9.72	0.00	0
4	91.18	8.82	8.82	0.00	0
5	87.66	12.34	12.26	0.00	0
6	79.11	17.59	14.25	0.04	3.25
7	73.36	19.35	13.22	0.08	7.22
8	75.32	17.28	11.18	0.07	7.33
Random 10k100k					
2	88.62	11.38	11.38	0.00	0
3	90.94	9.06	9.06	0.00	0
4	92.21	7.79	7.78	0.00	0
5	93.27	6.75	6.75	0.00	0.00
6	83.90	14.12	11.07	0.03	1.95
7	73.72	18.93	12.99	0.08	7.28
8	71.77	19.79	12.42	0.08	8.36

Πίνακας Π.3: Ποσοστά στατιστικών XACT_BREAKDOWN για Eager CD - Eager VM – HYBRID

threads	non-trans	trans	good trans	backoff	commiting	stall
Random 10k50k						
2	88.90	10.02	10.02	0.00	1.08	0
3	90.14	8.99	8.99	0.00	0.87	0
4	91.00	8.27	8.27	0.00	0.73	0
5	91.41	7.94	7.94	0.00	0.64	0.00
6	77.32	16.67	13.55	2.05	0.85	3.11
7	67.33	20.06	13.28	3.24	0.61	8.76
8	68.56	18.74	11.18	3.30	0.36	9.04
Random 10k100k						
2	87.96	10.70	10.70	0	1.34	0
3	90.40	8.59	8.59	0.00	1.00	0
4	90.83	8.36	7.43	0.00	0.82	0
5	92.66	6.65	6.65	0.00	0.69	0.00
6	82.04	13.31	10.55	1.84	0.76	2.05
7	68.38	19.41	12.90	3.12	0.71	8.38
8	64.32	21.26	12.33	3.93	0.49	10.00

Πίνακας Π.4: Ποσοστά στατιστικών XACT_BREAKDOWN για Eager CD - Lazy VM – TIMESTAMP

threads	non-trans	trans	good trans	aborting	commiting	stall
Random 10k50k						

2	89.81	8.73	8.73	0.00	1.19	0.28
3	90.89	7.84	7.84	0.00	1.01	0.25
4	91.68	7.24	7.24	0.00	0.87	0.21
5	92.09	6.93	6.93	0.00	0.79	0.19
6	18.41	81.12	0.35	0.40	0.05	0.03
7	15.31	84.41	0.06	0.27	0.01	0.01
8	0.81	98.86	0.03	0.31	0.004	0.003
Random 10k100k						
2	88.90	9.31	9.31	0.00	1.44	0.35
3	91.17	7.47	7.46	0.00	1.10	0.27
4	92.34	6.51	6.51	0.00	0.91	0.23
5	93.21	5.82	5.82	0.00	0.77	0.20
6	41.21	57.48	4.69	0.23	0.69	0.39
7	16.00	83.65	0.20	0.31	0.02	0.01
8	16.08	83.65	0.08	0.26	0.01	0.00

Πίνακας Π.5: Ποσοστά στατιστικών XACT_BREAKDOWN για Lazy CD - Lazy VM – BASE

Βιβλιογραφία

[1] S. Kang, and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and practice of parallel programming,* New York, NY, USA, 2009. ACM

[2] M. L. Scott, M. F. Spear, L. Daless and V. J. Marathe. Delaunay triangulation with transactional memory and barriers. In *IEEE Intl. Symp. on Workload Characterization,*2007.

[3] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388-398, Washingto, DC, USA, 2007. IEEE Computer Society.

[4] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris. Early experiences on accelerating Dijkstra's Algorithm using transactional memory. In *3rd Workshop on Multithreaded Architectures and Applications*, 2009.

[5] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High Perfomance Computer Architecture*, Austin, TX, 2006

[6] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and ¨ U. V. Catalyurek. A scalable distributed parallel breadth-first search algorithm on Bluegene/L. In *Proceedings of Supercomputing*, Seattle, WA, Nov. 2005

[7] D.A. Bader, and K. Madduri. Designing Multithreaded Algorithms for Breadth-First Search and *st*-connectivity on the Cray MTA-2. In *Proc. The 35th International Conference on Parallel Processing (ICPP)*, Columbus, OH, August 2006.

[8] Y. Zhang and E. Hansen. Parallel Breadth-First Heuristic Search on a Shared-Memory Architecture. In *AAAI-06 Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, Boston, MA, July 2006

[9] D. Dice, Y. Lev, M. Moir and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *ASPLOS-09 Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009. ACM

[10] Wisconsin multifacet gems simulator. http://www.cs.wisc.edu/gems/

[11] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill and D. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News,* 33(4):92-99, 2005

[12] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larson, A. Moestedt and B. Werner. Simics: A full system simulation platform. *Computer,* 35(2):50-58, Feb 2002

[13] L. Yen, J. Bobba, M.R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. *High Perfomance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on,* pages 261-272, Feb 2007

[14] D.A. Bader and K. Madduri. Gtgraph: A suite of synthetic graph generators. Feb 2006 http://www.cc.gatech.edu/~kamesh/GTgraph/.

[15] D. Chakrabarti, Y. Zhan and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proc. of 4th International Conference on Data Mining*, Apr 2004

[16] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, K. Olukotun: Characterization of TCC on Chip-Multiprocessors. In *the Proceedings of the Fourteenth International Conference on Parallel Architectures and Compilation Techniques*, Saint Louis, Missouri, 19 September 2005

[17] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H.Wijaya, C. Kozyrakis,and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, June 2004.

[18] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, K. Olukotun: ATLAS: A Chip-Multiprocessor with Transactional Memory Support. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*, Nice, France, April 2007

[19] T. Harris, A. Crystal, A. O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, M. Valero: Transactional Memory: An overview

[20] M. Herlihy, J. Eliot, and B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' Proc. 20th Ann. *Int'l Symp. Computer Architecture (ISCA 93)*, IEEE Press, 1993, pp. 289-300.

[21] D. E. Culler, J. P. Singh. Parallel Computer Architecture A hardware/software Approach.

[22] AMD64 Architecture Programmer's Manual Volume 2: System Programming pp 167-169 http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

[23] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, D. A. Wood, Performance Pathologies in Hardware Transactional Memory. In *ISCA '07, June 9-13, 2007, San Diego, California, USA*.