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

Σχεδιασμός και Υλοποίηση Συστήματος STM με Γρήγορη Πρόσβαση Μεταδεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΩΝΣΤΑΝΤΙΝΟΣ Δ. ΠΙΠΙΝΗΣ

Επιβλέπων : Γκούμας Γεώργιος
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2020

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

Σχεδιασμός και Υλοποίηση Συστήματος STM με Γρήγορη Πρόσβαση Μεταδεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΩΝΣΤΑΝΤΙΝΟΣ Δ. ΠΙΠΙΝΗΣ

Επιβλέπων : Γκούμας Γεώργιος
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Ιουλίου 2020.

....	....	....
Γκούμας Ι. Γεώργιος Επίκ. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2020

....

Κωνσταντίνος Δ. Πιπίνης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Κωνσταντίνος Δ. Πιπίνης, 2020.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περιεχόμενα

Περιεχόμενα . . . 5

Κατάλογος σχημάτων . . . 7

Περίληψη . . . 9

Abstract . . . 11

Ευχαριστίες . . . 13

Acknowledgements . . . 15

1. Εισαγωγή . . . 17

2. Θεωρητικό Υπόβαθρο . . . 21
- 2.1 Διεπαφή . . . 21
- 2.2 Γενικές Σχεδιαστικές Επιλογές . . . 22
  - 2.2.1 Χειρισμός Concurrency (Concurrency Control) . . . 22
  - 2.2.2 Διαχείριση Έκδοσης (Version Managment) . . . 23
  - 2.2.3 Βαθμός Λεπτομέρειας (Granularity) . . . 23
- 2.3 Software Transactional Memory . . . 23
  - 2.3.1 Metadata . . . 23
  - 2.3.2 Undo-logs και Redo-logs . . . 24
  - 2.3.3 Read Sets και Write Sets . . . 24

3. StandawayTM . . . 25
- 3.1 Δομή Tx_Thread . . . 27
- 3.2 Ανάγνωση . . . 31
- 3.3 Εγγραφή . . . 34
- 3.4 Αρχή του Transaction . . . 37
- 3.5 Committing Transaction . . . 38
- 3.6 Aborting Transaction . . . 39
- 3.7 Διαχείριση μνήμης . . . 39
- 3.8 Livelock . . . 41

4. Μεθοδολογία και Πειράματα . . . 45
- 4.1 StandawayTM . . . 45
- 4.2 Σύγκριση με TL2 . . . 46
- 4.3 STAMP Μετροπρογράμματα . . . 46
- 4.4 Υπολογιστικό Σύστημα . . . 47

5. Μετρήσεις και Αποτελέσματα 49
5.1 Genome 50
5.2 Labyrinth 51
5.3 SSCA2 52
5.4 Vacation 53
5.5 K-Means 55

6. Σχετική Βιβλιογραφία 57

7. Συμπεράσματα και επόμενα βήματα 59
7.1 Memory Footprint 59
7.2 Αρχικοποίηση των records 60
7.3 Strong atomicity και Hardware 60
7.4 Τοποθέτηση των records 60
7.5 Commit time false conflicts 61
7.6 Becoming rude 61
7.7 Επίλογος 61

Βιβλιογραφία 63

Κατάλογος σχημάτων

5.1 Speedup of TL2 and StandawayTM in Genome 50
5.2 Speed Improvement of StandawayTM compared to TL2 in Genome 50
5.3 Speedup of TL2 and StandawayTM in Labyrinth 51
5.4 Speed Improvement of StandawayTM compared to TL2 in Labyrinth 51
5.5 Speedup of TL2 and StandawayTM in ssca2 52
5.6 Speed Improvement of StandawayTM compared to TL2 in ssca2 52
5.7 Speedup of TL2 and StandawayTM in Vacation I 53
5.8 Speed Improvement of StandawayTM compared to TL2 in Vacation I 53
5.9 Speedup of TL2 and StandawayTM in Vacation II 54
5.10 Speed Improvement of StandawayTM compared to TL2 in Vacation II 54
5.11 Speedup of TL2 and StandawayTM in K-Means I 55
5.12 Speed Improvement of StandawayTM compared to TL2 in K-Means I 55
5.13 Speedup of TL2 and StandawayTM in K-Means II 56
5.14 Speed Improvement of StandawayTM compared to TL2 in K-Means II 56

Περίληψη

Καθώς ο νόμος του Moore φτάνει στα όριά του και οδηγούμαστε σε πολυπύρηνα επεξεργαστικά συστήματα, ο νόμος του Amdahl φαίνεται να παίζει κυρίαρχο ρόλο στον χρόνο εκτέλεσης απαιτητικών εφαρμογών. Για την υλοποίηση παράλληλων προγραμμάτων έχουν σχεδιαστεί διάφορες τεχνικές συγχρονισμού όπως σημαφόροι, κλειδώματα, mutexes, φράγματα και ατομικές εντολές. Όμως όλα αυτά τα συστήματα προϋποθέτουν γνώση από τον προγραμματιστή των διαφόρων race conditions και data dependencies σε κάθε φάση εκτέλεσης του προγράμματος.

Το Transactional Memory (TM) είναι μία μέθοδος συγχρονισμού που προτάθηκε για να απελευθερώσει τον προγραμματιστή από την δυσκολία υλοποίησης παράλληλων προγραμμάτων φροντίζοντας αυτόματα για τον συγχρονισμό του παράλληλου κώδικα. Για την χρήση TM ο προγραμματιστής ορίζει μία περιοχή (atomic block) ως transactional η οποία θα εκτελεστεί ατομικά σε σχέση με τις υπόλοιπες περιοχές που έχει ο ίδιος ορίσει ομοίως. Έχουν προταθεί δύο βασικές μορφές TM. TM υλοποιημένο σε Software (STM) και TM υλοποιημένο σε hardware (HTM), ενώ έχουν προταθεί TMs τα οποία είναι μερικώς υλοποιημένα σε software και μερικώς σε hardware (Hybrid Transactional Memory). Προτερήματα και ελαττώματα του κάθε συστήματος έχουν μελετηθεί εκτενώς από την βιβλιογραφία. Σε αυτήν την διπλωματική εργασία εστιάζουμε στα συστήματα STM που δεν απαιτούν υποστήριξη υλικού.

Κάθε σύστημα STM παρακολουθεί τις προσβάσεις μνήμης των threads για να αποφανθεί αν υπάρχει ή όχι παραβίαση της ατομικότητας, αν δηλαδή κατά την εκτέλεση ενός atomic block (transaction) κάποιο άλλο thread έχει επηρεάσει τις θέσεις μνήμης που αυτό το block κάνει access. Το STM σύστημα συνήθως κρατάει μεταδεδομένα (metadata) για να βρίσκει τις παραβιάσεις ατομικότητας, όπως logs ή Bloom Filters με τις θέσεις μνήμης που κάθε thread έχει γράψει ή διαβάσει. Ωστόσο, κάθε access στην μνήμη μέσω STM είναι συνήθως πολύ πιο ακριβό από το αντίστοιχο access χωρίς STM, αφού τόσο η αναζήτηση στα logs όσο και τα Bloom Filters για την εύρεση παραβίασης της ατομικότητας είναι μία ακριβή διαδικασία.

Στόχος της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός και η υλοποίηση ενός Software Transactional Memory (STM) συστήματος σε C ονόματι StandawayTM. Με το StandawayTM προτείνουμε έναν νέο τρόπο αποθήκευσης αυτών των metadata που επιτρέπει την αναγνώριση παραβίασης ατομικότητας σε σταθερό χρόνο καθώς και πολύ μικρό overhead σε κάθε access την μνήμης μέσο STM. Μέρος των metadata είναι αποθηκευμένο σε σταθερή απόσταση στην μνήμη από τα δεδομένα στα οποία αναφέρονται. Σε αυτά αποθηκεύονται οι ταυτότητες των threads, και ελέγχονται τα accesses στην μνήμη με λογική ενός γραφέα ή πολλών αναγνωστών. Οι αλλαγές σε αυτά τα metadata γίνονται μέσα από ατομικές εντολές. Σε περίπτωση conflict, το transaction που επιθυμεί να γράψει ή να διαβάσει, οφείλει να αναγνωρίσει το εν λόγο conflict και να κάνει abort.

Το StandawayTM παρότι αρκετά γρήγορο, δεν έχει σταθερή απόδοση και εξαρτάται από το εκάστοτε πρόγραμμα σε μεγαλύτερο βαθμό από άλλα STMs. Σε ευνοϊκά προγράμματα, παρατηρείται σημαντική βελτίωση αλλά η μη σταθερή συμπεριφορά του αφήνει χώρο για βελτιώσεις. Με μικρές αλλαγές στην λογική του, μπορούμε να εξασφαλίσουμε το forward guarantee. Επίσης το StandawayTM δεσμεύει πολύ μνήμη, όμως αυτό μπορεί να περιοριστεί με μεθόδους οι οποίες δεν έχουν ακόμα δοκιμαστεί και ελεγχθεί για την επιβάρυνση τους στην ταχύτητά.

Στην παρούσα εργασία αρχικά παρέχουμε πληροφορίες σχετικά με το Transactional Memory και μετέπειτα αναλύουμε τον σχεδιασμό του StandawayTM καθώς και τους λόγους που οδήγησαν σε αυτές τις σχεδιαστικές επιλογές. Τέλος, παρουσιάζουμε και εξηγούμε αναλυτικά τα αποτελέσματα πειραματικών μετρήσεων και προτείνουμε επόμενα πιθανά βήματα πάνω στον σχεδιασμό του StandawayTM.

Λέξεις κλειδιά

Transactional Memory, TM, Software Transactional Memory, STM, Παράλληλος προγραμματισμός, Παράλληλη επεξεργασία.

Abstract

The purpose of this diploma dissertation is the design and implementation of a Software Transactional Memory (STM) system in C called StandawayTM.

As Moore's Law is reaching its limits and multi-core systems have became predominant, Amdahl's Law seems to be the most relevant in achieving high performance execution. In order to make parallel processing possible, there exist a set of tools like semaphores, locks, mutexes, barriers and atomic instruction. But all the above require of the programmer to be aware of all data dependencies and race conditions that might be present in her code making parallel programming a difficult task.

Thus, Transactional Memory (TM) was proposed, which aimed to automate parallel programming by asking the programmer to specify the regions of code that are transactional, that is, to be executed atomically in regard to all other regions that are also marked as transactional. TM can be implemented in two ways, as Software in Software Transactional Memory (STM) or as Hardware in Hardware Transactional Memory (HTM). Their comparison is the object of a lot of research and are not the subject of this thesis, and we will only examine STMs.

Every STM system monitors the memory accesses of threads in order to decide if a conflict occurred and the atomicity of a transactional region (atomic block) is in danger. That is, if during the execution of an atomic block, another thread has affected a memory location that the atomic block is accessing. Each STM system usually keeps a record of all read and write accesses by all threads in logs or Bloom Filters. Both Bloom Filters and search in logs for conflict detection is an expensive procedure. Also each transactional access of memory, both read and write, has a lot of overhead in regard to a non-transactional access.

With StandawayTM we suggest a new way of keeping track of all accesses which allows for conflict detection in constant time as well as minimum overhead in each transactional access.

In this diploma dissertation, we will initially formalize Transactional Memory and then analyze the design choices of StandawayTM as well as explain the reason behind them. Finally we will present and explain experimental results and make suggestions on further improvements in the design of StandawayTM.

Key words

StandawayTM, STM, Software Transactional Memory, TM, Transactional Memory, Parallel Programming, Parallel Computing.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Γεώργιο Γκούμα καθώς και τον μεταδιδακτορικό ερευνητή, κ. Βασίλη Καρακώστα για την καθοδήγηση και υποστήριξη τους καθ' όλη την διάρκεια αυτού του ερευνητικού. Η υπομονή και η εμπιστοσύνη που μου έδειξαν στις εξερευνήσεις πάνω στα μονοπάτια που ακολούθησα μέχρι να σχηματιστεί αυτή η διπλωματική ήταν καθοριστικής σημασίας. Ευχαριστώ επίσης τα μέλη της εξεταστικής επιτροπής για τον χρόνο τους και τις συζητήσεις που είχαμε πάνω στην εργασία αυτή. Ευχαριστώ τον ερευνητή κ. Νίκα Κωνσταντίνο για τις χρήσιμες και διαφωτιστικές συμβουλές του. Ευχαριστώ τους συγγενείς και τους φίλους μου, τα αδέρφια μου και τους γονείς μου, για την υποστήριξή τους όλα αυτά τα χρόνια και τις υπέροχες φοιτητικές στιγμές που μου χάρισαν. Τέλος θα ήθελα να ευχαριστήσω την σχολή, το διδακτικό και λειτουργικό προσωπικό, τους συμφοιτητές μου και όλους όσους ήταν κομμάτι αυτής της υπέροχης εμπειρίας των τελευταίων έξι χρόνων που με άλλαξαν ως άνθρωπο και μου δίδαξαν τόσα πολλά.

Κωνσταντίνος Δ. Πιπίνης,

Αθήνα, 16η Ιουλίου 2020

Acknowledgements

I warmly thank the supervising professor of this dissertation, Prof. Georgios Goumas, as well as the postdoctoral researcher, Vasileios Karakostas, for their guidance and support throughout this research. The patience and confidence they showed me in the explorations on the paths I followed until this diploma thesis was formed were crucial. I would like to thank the members of the Advisory Committee for their time, and the discussions we had on this work. I thank the researcher Nikas Konstantinos for his helpful and enlightening advice. I thank my relatives and friends, my brother, my sister and my parents, for their support over the years and the wonderful student moments they have given me. Finally, I would like to thank the university, the teaching and operating staff, my classmates and all those who were part of this wonderful experience of the last six years who changed me as a person and taught me so much.

Konstantinos D. Pipinis,

Athens, July 16, 2020

Κεφάλαιο 1

Εισαγωγή

Το 2004 μετά από 50 χρόνια εκθετικής αύξησης της ταχύτητας των μονοπύρηνων επεξεργαστών επήλθε κορεσμός. Έτσι, οι σχεδιαστές στράφηκαν στους πολυπύρηνους επεξεργαστές τοποθετώντας δύο ή περισσότερους πυρήνες με κοινή μνήμη σε ένα chip. Εκμεταλλευόμενοι τα υπάρχοντα εργαλεία της παράλληλης επεξεργασίας αλλά και σχεδόν 50 χρόνια εμπειρίας, οι προγραμματιστές μπόρεσαν να φτιάξουν ολοένα και πιο απαιτητικά σε πόρους προγράμματα μειώνοντας τον χρόνο εκτέλεσης μοιράζοντάς τον στους διάφορους πυρήνες.

Δυστυχώς οι παράλληλοι αλγόριθμοι είναι πολύ πιο δύσκολοι να σχεδιαστούν, να συγγράφουν, να ελεγχθούν και να διορθωθούν από τους αντίστοιχους σειριακούς. Ένας προγραμματιστής πρέπει όχι μόνο να ανακαλύψει την παραλληλότητα εγγενή στον εκάστοτε αλγόριθμο, αλλά και να παρακολουθεί πολλαπλά ταυτόχρονα συμβάντα και να αντιμετωπίσει μία σειρά από προβλήματα που σχετίζονται μόνο με παράλληλη επεξεργασία, όπως έλλειψη ντετερμινιστικότητας και ατομικότητας, ή προβλήματα χρονισμού. Σίγουρα δεν βοηθάει και το γεγονός ότι δεν έχουν αναπτυχθεί δυνατά εργαλεία για την βοήθεια ανάπτυξης παράλληλου προγραμματισμού, είτε εξαιτίας της έλλειψης ενδιαφέροντος, είτε εξαιτίας της δυσκολίας ανάπτυξής τους.

Με εξαίρεση τα Transactional Memory συστήματα που θα αναλύσουμε παρακάτω, τα εργαλεία που υπάρχουν για τον προγραμματισμό παράλληλου κώδικα είναι της μορφής αποκλεισμού από πρόσβαση σε δεδομένα (locks, semaphores, mutexes, etc), της μορφής συγχρονισμού μεταξύ threads (barriers), και της μορφής primitive ατομικών εντολών (atomic add, atomic subtract, atomic or etc.). Σε επίπεδο hardware, υλοποιούνται είτε μέσω compare-and-swap είτε μέσω acquire-release εντολών (συνήθως ένα από τα δύο, καθώς είναι ισοδύναμα). Τα παραπάνω εργαλεία θέλουν προσεκτικό χειρισμό και επίγνωση όλων των πιθανών εκτελέσεων και χρονισμών. Bugs στον παράλληλο προγραμματισμό μπορεί να μην εμφανιστούν σε κάθε εκτέλεση του προγράμματος και να μην είναι εύκολος ο εντοπισμός τους. Επίσης, η μεταφερσιμότητά τους από μία πλατφόρμα σε άλλη δεν είναι πάντα απλή υπόθεση. Εκτός από την υποστήριξη του λειτουργικού (τα περισσότερα λειτουργικά συστήματα είναι πλέον συμβατά με POSIX), μπορεί να υπάρξει πρόβλημα με την μεταφορά από πλατφόρμα με σειριακή συνοχικότητα (Sequential Consistency) σε πλατφόρμα με πιο ελεύθερο μοντέλο μνήμης (Relaxed Memory Model) το οποίο επιτρέπει περισσότερα race conditions. Τέλος, αν η χρήση τους δεν γίνει με φειδώ, τότε το εν λόγω παράλληλο πρόγραμμα μπορεί να είναι πιο αργό συγκριτικά με το αντίστοιχο σειριακό λόγο του επιπρόσθετου κόστους εκτέλεσης (overhead) των παραπάνω εργαλείων.

Ένα από τα βασικά ελαττώματα του παράλληλου προγραμματισμού είναι η έλλειψη μηχανισμών αφαιρετικότητας και σύνθεσης. Αφαιρετικότητα είναι η δυνατότητα να απλοποιήσουμε ένα σύστημα στο απλό μοντέλο των εισόδων και εξόδων χωρίς να μας ενδιαφέρει το εσωτερικό. Σύνθεση είναι η δυνατότητα να συνδυάσουμε δύο αφαιρετικά μοντέλα δημιουργώντας ένα καινούργιο αφαιρετικό μοντέλο. Για παράδειγμα ας φανταστούμε μία λίστα στην οποία θέλουμε να μπορούμε να εισάγουμε, να αφαιρούμε και να ψάχνουμε κόμβους. Εάν δύο threads προσπαθούν να εισάγουν και να διαγράψουν σε διπλανές θέσεις ταυτόχρονα, τότε ανάλογα την υλοποίηση, είναι πιθανό να χαθούν κόμβοι. Χρειάζεται λοιπόν να γνωρίζουμε και να τροποποιήσουμε την υλοποίηση της λίστας για να υποστηρίζει παράλληλη επεξεργασία. Τώρα ας φανταστούμε δύο τέτοιες λίστες και μία καινούργια διαδικασία που μετακινεί έναν κόμβο από την μία στην άλλη. Ακόμα και αν οι λίστες επιτρέπουν παράλληλη επεξεργασία, η υλοποίηση της μετακίνησης ως αφαίρεση ενός κόμβου από την μία λίστα και εισαγωγή του στην άλλη είναι ελαττωματική. Ας φανταστούμε το εξής σενάριο. Ο κόμβος Κ βρίσκεται

στο τέλος της λίστας Β. Το thread 1 θέλει να τον μετακινήσει στην αρχή της λίστας Α, ενώ το thread 2 θέλει να ψάξει στις δύο λίστες για το αν υπάρχει ο κόμβος Κ. Το thread 2 διασχίζει την λίστα Α και δεν βρίσκει το Κ, οπότε ετοιμάζεται να ψάξει στην λίστα Β. Εκείνη την στιγμή το thread 1 αφαιρεί τον κόμβο Κ από το τέλος της λίστας Β, και τον τοποθετεί στην αρχή της λίστας Α. Το thread συνεχίζει την αναζήτηση στην λίστα Β και δεν βρίσκει τον κόμβο Κ παρότι ο κόμβος υπάρχει σε μία από τις δύο λίστες. Τέτοιας φύσης φαινόμενα δεν συμβαίνουν μόνο στις λίστες αλλά είναι γενικότερα προβλήματα του παράλληλου προγραμματισμού.

Το 1993 προτάθηκε για πρώτη φορά ένα abstruction για τον παράλληλο προγραμματισμό, το Transactional Memory. Ο Herlihy και ο Moss [Herl93] πρότειναν ένα Transactional Memory σύστημα υλοποιημένο σε hardware και ο Stone και λοιποί [Ston93] πρότειναν ατομική επεξεργασία σε πολλά δεδομένα με υποστήριξη από software. Αυτές οι δύο δημοσιεύσεις θα είναι αρχή ενός νέου πεδίου έρευνας τόσο σε hardware όσο και σε software που θα μετατρέψουν το Transactional Memory σε ένα εργαλείο με πολλές δυνατότητες και πιθανές πρακτικές εφαρμογές.

Στην καρδιά του Transactional Memory (TM) υπάρχει το transaction, ένα τμήμα κώδικα που το TM φροντίζει να εκτελεστεί ατομικά σε σχέση με όλα τα άλλα transactions, παρεμβαίνοντας στην πρόσβαση του προγράμματος στην μνήμη και στην ροή εκτέλεσης του κώδικα. Με την εξασφάλιση ατομικότητας ανεξάρτητα του περιεχομένου του transaction, μπορούμε να θεωρήσουμε ότι όλα τα transactions εκτελούνται σειριακά (παρότι τρέχουν παράλληλα), έστω με κάποια σειρά. Έτσι κάθε transaction είναι πλέον ένα αφαιρετικό μοντέλο. Καθώς το Transaction είναι ανεξάρτητο του κώδικά που εκτελεί μπορούμε να συνθέσουμε δύο Transactions που εκτελούνται σειριακά μεταξύ τους, σε ένα μεγαλύτερο Transaction.

Βασική ιδέα στο TM είναι το ”all or nothing”. Όταν ένα transaction παραβιάζει την ατομικότητα ενός άλλου (και αυτομάτως το δεύτερο παραβιάζει την ατομικότητα του πρώτου), οι αλλαγές στην μνήμη που προκαλεί ένα από τα δύο transactions πρέπει να ακυρωθούν σαν το transaction που θα ακυρωθεί να μην είχε ξεκινήσει ποτέ. Άρα κάθε transaction που ξεκινάει έχει μία από δύο μοίρες, είτε θα ολοκληρωθεί (commit) και όλες οι αλλαγές στην μνήμη θα είναι εμφανείς είτε θα ματαιωθεί (abort) και καμία αλλαγή στην μνήμη δεν θα γίνει εμφανής. Συνήθως τα transactions που ματαιώνονται εκτελούνται ξανά.

Παρότι πολύ χρήσιμο εργαλείο, το TM δεν έχει γίνει ευρέως διαδεδομένο [Cacs08]. Η υλοποίησή του σε hardware εκτός από ακριβή, το κάνει εξαρτώμενο από τον εκάστοτε κατασκευαστή, και έρχεται συνήθως με μία σειρά από περιορισμούς στο μέγεθος του transaction. Από την άλλη η υλοποίησή του σε software παρότι μπορεί να προσφέρει περισσότερες δυνατότητες και δεν έχει τους ίδιους περιορισμούς, παράγει πιο αργά προγράμματα λόγω του overhead του TM.

Αυτό το overhead προκαλείται από τον επιπρόσθετο κώδικα σε κάθε πρόσβαση στην μνήμη, την ανάγκη για ιστορικό όλων των προσβάσεων, και του ελέγχου για παραβίαση της ατομικότητας. Κάθε TM σύστημα χρησιμοποιεί μία σειρά από μηχανισμούς για να εκτελεί τις παραπάνω λειτουργίες. Τα περισσότερα TMs διατηρούν μία σειρά από πληροφορίες σε λίστες τις οποίες πρέπει να διασχίσουν πάνω από μία φορές κατά την εκτέλεση ενός Transaction. Αρκετά TMs χρησιμοποιούν Bloom Filters[Bloo70]. Τα Bloom Filters υλοποιημένα σε software είναι αργά, ενώ πολλές φορές οδηγούν το TM σε λάθος αποφάσεις λόγω της μη ντετερμινιστικότητάς τους. Οι μηχανισμοί αυτοί, ή παρόμοιοί τους, δεν έχουν σταθερή υπολογιστικά συμπεριφορά και αυξάνουν το overhead των TMs συχνά με απρόβλεπτες μορφές.

Τα περισσότερα TM συστήματα που έχουν προταθεί, έχουν σχεδιαστεί την περίοδο μεταξύ 2006 και 2009. Από τότε, τα σύγχρονα υπολογιστικά συστήματα έχουν βελτιωθεί σε τομείς τους οποίους δεν μπορούσαν να λάβουν υπόψιν πριν πάνω από μία δεκαετία. Μία βασική αλλαγή, είναι η μεταφορά από 32 σε 64 bits. Αυτή η αλλαγή ελευθέρωσε το address space και επέτρεψε την αύξηση της μνήμης πέρα από τα 4GB. Όπως θα δούμε και στο StandawayTM , αυτή η αλλαγή μας επιτρέπει να παρακολουθούμε έως και 32 threads. Μία άλλη αλλαγή είναι βελτιώσεις στις ταχύτητες πρόσβασης των caches. Καθώς τα δεδομένα μπορούν να μεταφερθούν σε πολύ λίγους κύκλους από την μία cache στην άλλη, η τοπικότητα των δεδομένων αλλά και των metadata του TM ανά thread δεν είναι πλέον τόσο σημαντική. Τέλος, τα σύγχρονα υπολογιστικά συστήματα μπορούν να εκτελούν ταυτόχρονα πολλές

ατομικές εντολές. Σε παλαιότερα συστήματα, οι ατομικές εντολές απέτρεπαν (ή πιο σωστά καθυστερούσαν) την πρόσβαση στις caches μέχρι την ολοκλήρωσή τους. Στους νέους επεξεργαστές, οι ατομικές εντολές έχουν απεμπλακεί μεταξύ τους, και μπορούν να εκτελεστούν όχι μόνο ταυτόχρονα με άλλες προσβάσεις στην μνήμη, αλλά και ταυτόχρονα με άλλες ατομικές εντολές σε διαφορετικά cache lines.

Το StandawayTM εκμεταλλεύεται τα παραπάνω χαρακτηριστικά για να μειώσει το overhead του TM συστήματος. Χρησιμοποιεί ένα στατικά κατανεμημένο τρόπο αποθήκευσης ορισμένων metadata με σταθερό χρόνο πρόσβασης και ελέγχου της παραβίασης της ορθότητας, ενώ σημαντικό μέρος της υλοποίησης του συγχρονισμού αποτελούν οι ατομικές εντολές.

Για τον έλεγχο της βελτίωσης της ταχύτητας του StandawayTM, το συγκρίνουμε με το TM σύστημα TL2[Dice06] πάνω σε μία σειρά από benchmarks με το όνομα STAMP[Minh08]. Το TL2 έχει εδραιωθεί ως το μέτρο σύγκρισης των STMs και παρότι από το 2006 όταν προτάθηκε έχει γίνει αρκετή έρευνα πάνω στα STMs, το TL2 παραμένει ένα από τα πιο γρήγορα. Το STAMP προτάθηκε το 2008 για την αξιολόγηση των TM συστημάτων. Αποτελείται από 8 benchmarks με διαφορετικά χαρακτηριστικά που προσομοιώνουν πραγματικές εφαρμογές. Για την αξιολόγηση του StandawayTM έπρεπε να κάνουμε τις απαραίτητες αλλαγές στο STAMP.

Οι μετρήσεις που πήραμε παρότι σε μερικές περιπτώσεις δεν ήταν αυτές που περιμέναμε, μας απέδειξαν ότι το StandawayTM έχει προοπτικές και οι βασικές αρχές που ακολουθήσαμε δεν ήταν εσφαλμένες. Σίγουρα χρειάζεται περαιτέρω μελέτη πάνω στα διάφορα φαινόμενα μνήμης που επηρεάζουν την απόδοση καθώς και στις πολιτικές για αποφυγή του livelock στο οποίο το StandawayTM είναι επιρρεπές. Στο τέλος αυτής της διπλωματικής παρουσιάζουμε κάποιες προτάσεις οι οποίες πιστεύουμε ότι θα βελτιώσουν την αποδοτικότητα και σταθερότητα του StandawayTM και θα μπορέσουν να το κάνουν ένα από τα state of the art TM συστήματα.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

Ένα Transaction είναι μία σειρά από γεγονότα που φαίνονται αδιαχώριστα και στιγμιαία σε έναν εξωτερικό παρατηρητή, συγκεκριμένα σε ένα άλλο Transaction [Harr10]. Ένα Transaction μπορεί να πετύχει ή να αποτύχει. Αν πετύχει τότε λέμε ότι ολοκληρώνεται (commit) ενώ αν αποτύχει λέμε ότι ματαιώνεται (abort). Υπάρχουν κάποια συγκεκριμένα κριτήρια που μας ενδιαφέρουν:

- Ατομικότητα (atomicity): Είτε θα εκτελεστούν πλήρως όλα τα επιμέρους γεγονότα, είτε δεν θα εκτελεστεί κανένα. Δεν επιτρέπεται κάποιο γεγονός να εκτελεστεί εν μέρη και το Transaction να κάνει commit. Αν ένα Transaction κάνει abort δεν επιτρέπεται κάποιο γεγονός να αφήσει πίσω του σημάδια της εκτέλεσής του.

- Συνοχικότητα (consistency): Θεωρώντας κάποιες επιτρεπτές καταστάσεις εντός του προγράμματος, κάθε Transaction που ξεκινάει από επιτρεπτή κατάσταση και με σειριακή εκτέλεση καταλήγει σε επιτρεπτή κατάσταση, πρέπει να καταλήγει σε επιτρεπτή κατάσταση και σε παράλληλη εκτέλεση. Για παράδειγμα αν μεταφέρουμε χρήματα από έναν λογαριασμό σε άλλο, τότε το συνολικό άθροισμα χρημάτων όλων των λογαριασμών θα πρέπει να είναι σταθερό. Προφανώς ένα Transaction που κάνει abort δεν επηρεάζει το consistency καθώς δεν αφήνει πίσω σημάδια της εκτέλεσής του.

- Απομόνωση (isolation): Η εκτέλεση ενός Transaction δεν πρέπει να επηρεάζει τα υπόλοιπα Transactions που τρέχουν ταυτόχρονα.

- Αντοχικότητα (durability): Τα αποτελέσματα των γεγονότων ενός committed Transaction είναι μόνιμα και εμφανή στα Transactions που θα το ακολουθήσουν.

- Σειριοποιητότητα (serializability): Τα αποτελέσματα των εκτελέσεων των Transactions μπορούν να αποδοθούν σε μία επιτρεπτή σειρά εκτέλεσής του.

2.1 Διεπαφή

Στην πιο απλή μορφή του ένα σύστημα TM επικοινωνεί με το πρόγραμμα με την εξής διεπαφή (interface):

```
void Tx_begin();
bool Tx_commit();
void Tx_abort();
T Tx_read(T *address);
void Tx_write(T *address, T value);
```

Το Tx_begin ορίζει την αρχή του Transaction για κάθε thread. Το Tx_commit ορίζει το τέλος του Transaction και προσπαθεί να το κάνει commit. Αν δεν καταφέρει να κάνει commit τότε το Transaction θα γίνει abort. Πολλές φορές η διαδικασία του abort φαίνεται και στον προγραμματιστή μέσω του Tx_abort. Το Tx_read επιστρέφει τα δεδομένα τύπου T που βλέπει το Transaction στην θέση address

(όχι απαραίτητα το περιεχόμενο της θέσης μνήμης address αλλά την τιμή που εξασφαλίζει την μη παραβίαση των παραπάνω κριτηρίων). Το Tx_write γράφει στην εικόνα της μνήμης που αντιστοιχεί στο Transaction στην θέση address την τιμή value (πάλι με τρόπο τέτοιο ώστε να μην παραβιάζονται τα παραπάνω κριτήρια).

Παρακάτω βλέπουμε ένα απλό παράδειγμα αύξησης ενός μετρητή κατά ένα:

```
1 long counter = 0;
2 ...
3 do{
4     Tx_begin();
5     int temp = Tx_read(&counter);
6     Tx_write(&counter, temp + 1);
7 }while (!Tx_commit());
```

2.2 Γενικές Σχεδιαστικές Επιλογές

Με βάση την παραπάνω διεπαφή (interface) μπορούν να σχεδιαστούν πολλά διαφορετικά TM συστήματα, με διαφορετικά χαρακτηριστικά. Παρουσιάζουμε τους βασικούς ανεξάρτητους άξονες σχεδιαστικών επιλογών.

2.2.1 Χειρισμός Concurrency (Concurrency Control)

Ένα TM πρέπει να χειρίζεται τον συγχρονισμό των δεδομένων ώστε όλα τα Transactions να βλέπουν πάντα επιτρεπτή κατάσταση της μνήμης. Ορίζουμε τα παρακάτω συμβάντα:

- Μία σύγκρουση (conflict) συμβαίνει (occurs) όταν δύο Transactions εκτελούν συγκρουόμενες ενέργειες πάνω στα ίδια δεδομένα. Δηλαδή, δύο ταυτόχρονες εγγραφές, η εγγραφή από το ένα Transaction και ανάγνωση από το άλλο.

- Ένα conflict εντοπίζεται (detected) όταν το TM σύστημα αποφασίζει ότι έχει συμβεί ένα τέτοιο conflict.

- Το conflict λύνεται (resolved) όταν το TM εκτελέσει τις κατάλληλες ενέργειες για να αποφύγει το conflict: είτε να καθυστερήσει την εκτέλεση ενός εκ των δύο Transactions, είτε να κάνει abort τουλάχιστον το ένα εκ των δύο Transactions.

Με βάση τα παραπάνω ο πρώτος άξονας σχεδιαστικών επιλογών αφορά τον χρόνο που τα τρία γεγονότα εκτελούνται. Η σειρά παραμένει πάντα ίδια.

- Με απαισιόδοξο (pessimistic) concurrency control την στιγμή που συμβαίνει ένα conflict το TM το εντοπίζει και το επιλύει.

- Με αισιόδοξο (optimistic) concurrency control τα conflicts εντοπίζονται και επιλύονται την ώρα που το Transaction κάνει commit.

- Με υβριδικό (hybrid) concurrency control τα conflicts εντοπίζονται την στιγμή που συμβαίνουν αλλά η επίλυσή τους γίνεται την ώρα του commit.

Η παραπάνω ταξινόμηση κρύβει άλλον έναν άξονα. Ένα conflict μπορεί να εντοπιστεί είτε πρόωρα (tentative conflict detection), δηλαδή την ώρα που συμβαίνει, είτε στα πρώτα βήματα του commit (committing conflict detection). Αν έχουμε tentatice conflict detection, τότε θα έχουμε pessimistic concurrency control με άμεση επίλυση του conflict ή hybrid concurrency control με επίλυση του conflict στο commit. Αν έχουμε committing conflict detection, τότε έχουμε υποχρεωτικά optimistic concurrency control.

2.2.2 Διαχείριση Έκδοσης (Version Managment)

Καθώς ένα Transaction μπορεί να αποτύχει και δεν πρέπει να αφήσει πίσω δείγματα της εκτέλεσής του, το TM πρέπει να φροντίσει να έχει την εικόνα της μνήμης πριν και μετά την εκτέλεσή του.

- Στο Ενθουσιώδης (Eager) Version Managment οι εγγραφές ενός Transaction γίνονται απευθείας στην μνήμη ενώ οι παλιές τιμές αποθηκεύονται σε δομές αναίρεσης (undo-logs), ώστε η εικόνα της μνήμης να διορθωθεί σε abort. Με αυτήν την προσέγγιση συνήθως έχουμε και pessimistic concurrency control για να μπορέσουμε να διατηρήσουμε τα κριτήρια που προαναφέραμε.

- Στο Οκνηρό (Lazy) Version Managment οι εγγραφές ενός Transaction γίνονται σε μία δομή εγγραφών (redo-log) τοπική (local) για το Transaction. Οι αναγνώσεις σε θέσεις που έχουν εγγραφεί από το Transaction γίνονται από το redo-log. Όταν έρθει η ώρα για commit και δεν υπάρχει conflict τότε οι εγγραφές περνάνε στην μνήμη, ενώ αν γίνει abort τότε απλά το redo-log πετιέται.

2.2.3 Βαθμός Λεπτομέρειας (Granularity)

Για να μπορεί να παρακολουθεί πολλές θέσεις μνήμης γρήγορα, πολλές φορές ένα TM κάνει βελτιστοποιήσεις και χρησιμοποιεί μεθόδους που δεν είναι αντιστρέψιμες. Εξαιτίας αυτού εντοπίζει conflicts που δεν υπάρχουν στην πραγματικότητα. Ένα παράδειγμα είναι η χρήση Bloom Filters τα οποία όταν κορέσουν, οι περισσότερες αναζητήσεις σε αυτά επιστρέφουν θετικό αποτέλεσμα. Ένα άλλο παράδειγμα είναι η θεώρηση του cache line ως το ελάχιστο μέγεθος πρόσβασης. Εγγραφές σε διπλανές θέσης μνήμης από διαφορετικά threads στο ίδιο cache line προκαλούν conflict.

2.3 Software Transactional Memory

Από αυτό το σημείο και έπειτα, οι θεμελιώδης διαφορές μεταξύ STM και HTM δεν μας επιτρέπουν να συνεχίσουμε την ανάλυση από κοινού. Θα επικεντρωθούμε μόνο στο STM και θα αναλύσουμε τις σχεδιαστικές επιλογές που το αφορούν.

2.3.1 Metadata

Για να μπορεί ένα TM να αναγνωρίζει πότε έχει συμβεί ένα conflict, πρέπει να έχει αποθηκευμένες πληροφορίες σχετικές με το ιστορικό των προσβάσεων μνήμης. Αυτές οι πληροφορίες μπορούν τα αφορούν:

- Αντικείμενα (objects): Το TM φροντίζει να ελέγχει την πρόσβαση με βάση το κάθε αντικείμενο. Οι πληροφορίες σχετικές με το ποιο thread έχει εκτελέσει κάποιο access στο αντικείμενο αποθηκεύονται συνήθως στην επικεφαλίδα (header) του. Ταυτόχρονες προσβάσεις σε διαφορετικά στοιχεία (fields) του ίδιου αντικειμένου από διαφορετικά threads οδηγούν σε εσφαλμένα conflicts.

- Θέσεις Μνήμης (memory addresses): Το TM φροντίζει να ελέγχει την πρόσβαση σε κομμάτια (blocks) της μνήμης με συγκεκριμένες διευθύνσεις. Τα blocks είναι συγκεκριμένου μεγέθους, συνήθως όσο ένα word ή ένα cache line. Τυπικά, το TM κρατάει κάποια metadata συγκεκριμένου μεγέθους, και χρησιμοποιεί κάποια συνάρτηση μεταφοράς (hash function) για να συνδέει τα metadata με τις διευθύνσεις μνήμης.

Οι διαφορές μεταξύ object-based και word-based TM δεν επηρεάζουν ιδιαίτερα τις περισσότερες σχεδιαστικές επιλογές. Όπου γίνεται λόγος για διευθύνσεις μνήμης, μπορεί εύκολα να γίνει αντιστοιχία και για αντικείμενα. Για λόγους απλότητας και συντομίας, αναφερόμαστε κυρίως στις διευθύνσεις μνήμης.

2.3.2 Undo-logs και Redo-logs

Η εκτέλεση ενός Transaction είναι εκ φύσεως υποθετική (speculative execution). Το εκάστοτε Transaction εκτελείτε απομονωμένο από τα υπόλοιπα, και η τελική του μοίρα (αν θα γίνει committed ή aborted) καθορίζει το αν οι επιδράσεις του στην μνήμη πρέπει να γίνουν εμφανείς στα υπόλοιπα Transactions.

- Τα TMs με Eager Version Managment αποθηκεύουν τις νέες τιμές απευθείας στην μνήμη. Σε περίπτωση που το Transaction αποτύχει, οι παλιές τιμές πρέπει να γραφτούν στην μνήμη, για να αναιρεθούν οι επιδράσεις του Transaction. Τα Undo-Logs αποθηκεύουν τα ζευγάρια των παλαιών τιμών και των διευθύνσεών τους, μαζί με άλλες πιθανώς χρήσιμες πληροφορίες.

- Τα TMs με Lazy Version Managment αποφεύγουν να αλλάξουν την μνήμη προτού είναι βέβαια ότι το εκάστοτε Transaction θα ολοκληρωθεί επιτυχώς. Οι αλλαγές που θέλουν να κάνουν στην μνήμη αποθηκεύονται προσωρινά σε Redo-Logs παρόμοιας μορφής με τα undo-logs. Τα Transactions που διαβάζουν μεταβλητές που έχουν γράψει, θα πρέπει να ψάξουν στα redo-logs για να βρουν την πιο πρόσφατη τιμή της μεταβλητής αυτής.

2.3.3 Read Sets και Write Sets

Όπως προαναφέραμε, για να μπορεί ένα TM να αναγνωρίζει πότε έχει συμβεί ένα conflict, πρέπει να έχει αποθηκευμένες πληροφορίες σχετικές με το ιστορικό των προσβάσεων μνήμης. Το ιστορικό των αναγνώσεων (read set) και των εγγραφών (write set) συνήθως είναι χωριστά. Μερικά TMs έχουν κοινά sets για όλα τα threads, ενώ άλλα έχουν διαφορετικά sets για κάθε thread. Σε μερικές περιπτώσεις, το write set είναι το ίδιο με το undo ή redo log.

Ένα set μπορεί να περιέχει τριών ειδών πληροφορίες.

- Τις διευθύνσεις των θέσεων μνήμης που έγιναν accessed.

- Χρονοσφραγίδες (time-stamps) των θέσεων μνήμης που έγιναν accessed. Αντί το TM να κρατάει δεδομένα για το ποιες διευθύνσεις έγιναν access, κρατάει τον χρόνο στον οποίο έγινε η εγγραφή τους, σε σχέση με ένα κοινό ρολόι (global clock). Εάν κάποια θέση μνήμης που κάνει access το Transaction, έχει γραφτεί από ένα άλλο thread, τότε η χρονοσφραγίδα της θέσης αυτής θα έχει μία μεταγενέστερη τιμή από το global clock την στιγμή που ξεκινάει το Transaction. Οι χρονοσφραγίδα της κάθε θέσης ενημερώνεται με την εγγραφή στην θέση αυτή.

- Υπογραφές (signatures) των διευθύνσεων μνήμης που έχουμε κάνει access. Τα signatures αποθηκεύονται σε Bloom-Filters[Bloo70], ένα σχήμα που επιτρέπει την γρήγορη αναζήτηση ύπαρξης μίας διεύθυνσης αλλά επιτρέπει λανθασμένα θετικά αποτελέσματα (false positives).

Τα παραπάνω sets συνδυάζονται με διάφορους τρόπους σε κάθε TM για να πετύχουν το ζητούμενο αποτέλεσμα.

Κεφάλαιο 3

StandawayTM

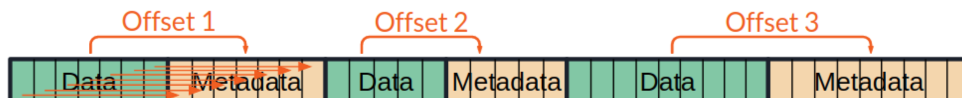
Με το StandawayTM προτείνουμε μία νέα μέθοδο τοποθέτησης των metadata και ελέγχου της ορθότητας της ατομικότητας των transactions. Σκοπός μας είναι να μειώσουμε το overhead του TM συστήματος θυσιάζοντας όμως αρκετή μνήμη. Το StandawayTM παρότι αρκετά απλό στην βασική υλοποίηση μπορεί να περιέχει αρκετά επιπρόσθετα features όπως θα δούμε παρακάτω.

Το StandawayTM προήλθε από δύο παρατηρήσεις. (i) Τα reads και writes που περνάνε μέσα από το TM είναι πολύ ακριβά και (ii) τα σύγχρονα υπολογιστικά συστήματα έχουν πολύ μνήμη. Το δεύτερο δεν χρειάζεται κάποια επεξήγηση. Για το πρώτο πρέπει να δούμε τις εξής περιπτώσεις:

- Το TM κρατάει logs με τις θέσεις μνήμης που έχει διαβάσει ή γράψει. Για να ελέγξουμε αν έχει γράψει κάποιος άλλος στις θέσεις μνήμης που έχουμε κάνει access, θα πρέπει να ψάξουμε τα logs. Αυτή η διαδικασία είναι γραμμική και πρέπει να γίνει για πολλές εγγραφές. Αν κρατάμε χρονοσφραγίδες (time-stamps) των προσβάσεων τότε μπορούμε να κοιτάμε μόνο δύο logs αντί για τα logs όλων των threads όμως το πρόβλημα παραμένει.

- Για να αποφύγουμε την γραμμικότητα αυτή μπορούμε να κρατάμε τις θέσεις μνήμης σε Bloom Filters, όμως τα Bloom Filters έχουν αργή υλοποίηση σε software ενώ η απώλεια πληροφορίας διακατέχει τον κίνδυνο τα μεγάλα transaction να ακυρώνονται χωρίς να έχουν πραγματικό conflict (false positives).

Για να ελαχιστοποιήσουμε τον χρόνο πρόσβασης στα logs, δημιουργήσαμε κατανεμημένα logs που βρίσκονται σε συγκεκριμένη απόσταση από τα δεδομένα που θέλουμε να κάνουμε access. Για παράδειγμα, αν η θέση των metadata για την θέση `A[i]` ενός πίνακα χαρακτήρων είναι `&A[i]+offset`, τότε η θέση των metadata για την θέση `A[i+1]` είναι `&A[i+1]+offset`. Αυτή η τοποθέτηση μας περιορίζει στο μέγεθος των metadata σε σχέση με το Granularity.



Στόχος: Make common case fast. Οι σχεδιαστικές μας επιλογές βασίστηκαν στο γεγονός ότι στα περισσότερα benchmarks τα περισσότερα transaction ολοκληρώνονται επιτυχώς. Έτσι πήραμε ως γενική περίπτωση την ολοκλήρωση του Transaction χωρίς προβλήματα. Επίσης όσο πιο μικρό το overhead του TM, τόσο λιγότερες οι πιθανότητες να έχουμε conflict. Το παραπάνω ισχύει σε περίπτωση που ανάμεσα στα transactions υπάρχει μη-transactional κώδικας, κάτι που σχεδόν πάντα συμβαίνει. Επίσης δόθηκε μεγάλη σημασία στην απλότητα του σχεδιασμού και της υλοποίησης και την αποφυγή optimizations κατά περίπτωση και λοιπών σχημάτων που ”διορθώνουν” λάθη του σχεδιασμού. Αυτό προέρχεται από ιδία φιλοσοφία.

Granularity. Στην υλοποίηση που παρουσιάζουμε έχουμε τους εξής περιορισμούς. Ο μέγιστος αριθμός από threads είναι 32 ενώ το Granularity είναι 8 bytes. Έτσι η διεύθυνση των metadata είναι πάντα στρογγυλοποιημένη σε 8 byte alignment προς τα κάτω (τρία τελευταία bits 0). Τα παραπάνω ισχύουν για 64-bit συστήματα.

Records. Για κάθε 8 bytes (word) που κάνει access το TM, κρατάμε μία εγγραφή μορφής bitmask 8 bytes. Η εγγραφή αυτή είναι κοινή (shared) μεταξύ όλων των threads. Τα πρώτα 32 bits κρατάνε τις ταυτότητες των threads που έχουν διαβάσει από το εν λόγω word (read record) ενώ τα επόμενα 32 bits κρατάνε τις ταυτότητες των threads που έχουν γράψει στο εν λόγο word (write record). Οι ταυτότητες είναι bitmasks με το bit στην θέση του thread ίσο με 1 και όλα τα άλλα bits 0. Για παράδειγμα η ταυτότητα του thread 2 (με αρχή το 0) είναι 00000000 00000000 00000000 00000100, ενώ η εγγραφή ανάγνωσης 00000001 01000000 00000010 00000101 σημαίνει ότι έχουν διαβάσει από το word τα threads 0, 2, 9, 22 και 24. Για κάθε thread και για κάθε word αφιερώνουμε 2 bits και έτσι προκύπτει ο περιορισμός των 32 threads στα 64-bits συστήματα. Όλα τα records πρέπει να αρχικοποιηθούν στο 0 πριν ξεκινήσουν τα transactions.

Conflict Detection. Με κάθε πρόσβαση σε ένα word έχουμε άμεσα και σε σταθερό χρόνο πρόσβαση στα records και μπορούμε να ελέγξουμε για conflict. Έτσι επιλέγουμε να εντοπίζουμε τα conflicts tentatively, δηλαδή την ώρα που συμβαίνουν. Όταν πάμε να γράψουμε σε ένα word, ελέγχουμε το read record του και αν περιέχει την ταυτότητα κάποιου άλλου thread τότε έχουμε conflict. Όταν πάμε να διαβάσουμε από μία θέση, ελέγχουμε το write record και αν περιέχει την ταυτότητα κάποιου άλλου thread έχουμε πάλι conflict. Για λόγους απλότητας επιλέγουμε να θεωρούμε κάθε write και ως read. Όταν κάνουμε εγγραφή σε ένα word, προσθέτουμε την Ταυτότητά μας και στο write και στο read record. Αυτή η επιλογή έγινε για λόγους απλότητας και ταχύτητας του κώδικα στην γενική περίπτωση.

Version Management. Εφόσον δεν υπάρχει conflict, είναι η ώρα να ολοκληρώσουμε την ανάγνωση ή την εγγραφή. Επιλέγουμε Eager Version Managment, δηλαδή άμεση εγγραφή στην μνήμη. Ο λόγος είναι ο εξής: Σε περίπτωση που έχουμε Lazy Version Managment, πρέπει να αποθηκεύουμε τις νέες τιμές σε redo-log καθώς δεν έχουμε χώρο να το κάνουμε στα records. Τα Transactions που πάνε να διαβάσουν words που έχουν τα ίδια προηγουμένως γράψει, θα πρέπει να αναζητήσουν τις νέες τιμές στα redo-logs, μία ακριβή διαδικασία. Έχοντας Eager Version Managment μπορούμε να αποθηκεύουμε απευθείας και να διαβάζουμε άμεσα από το ίδιο το word. Κρατάμε έτσι ένα undo-log για κάθε thread με τις παλιές τιμές. Ένα word προστίθεται στο undo-log, αν πριν την εν λόγω εγγραφή το write record που του αντιστοιχεί ήταν κενό (όλα μηδενικά).

Concurrency Control. Αυτή η προσέγγιση μας αναγκάζει σε Pessimistic Concurrency Control. Καθώς τα Transactions δουλεύουν στην κανονική μνήμη και όχι σε αντίγραφό της, οι εγγραφές σε words που έχουν διαβαστεί από τρίτο transaction, και οι αναγνώσεις από words που έχει γράψει τρίτο Transaction πρέπει να οδηγούν το αρχικό transaction σε abort. Αυτό μας δημιουργεί και το αναλλοίωτο ότι το write record θα έχει το πολύ ένα bit ίσο με 0, δηλαδή μόνο ένα thread μπορεί να γράφει σε ένα word την φορά. (Παρακάτω θα δούμε ότι παραβιάζεται για σύντομο χρονικό διάστημα το παραπάνω αναλλοίωτο, αλλά αυτό δεν επηρεάσει την ορθότητα.) Όταν ένα Transaction κάνει abort, τότε διατρέχουμε το undo-log και γράφουμε τα παλιά words στην θέση των καινούργιων.

Read-logs & Undo-logs. Τα records περιέχουν bitmasks για το αν ένα thread έχει κάνει ή όχι κάποιας μορφής access στο word. Όταν το transaction ολοκληρωθεί, είτε κάνει abort, είτε commit, τα records που έχει κάνει access πρέπει να καθαρίσουν, δηλαδή να αφαιρεθεί η ταυτότητα του thread από αυτά. Για να γνωρίζει το TM ποια records πρέπει να καθαρίσουν, κάθε πρώτη ανάγνωση αποθηκεύεται σε ένα read-log. Για να γνωρίζουμε τα records που έχουμε γράψει αρκεί να δούμε το undo-log. Στο commit το transaction περνάει από όλες τις θέσεις του read-log και undo-log και αφαιρεί την ταυτότητα από τα αντίστοιχα records. Στο abort προτού αφαιρέσει την Ταυτότητα από ένα record, διορθώνει την τιμή του word σε αυτήν πριν την εκτέλεση του aborting transaction.

TM Operations. Συνοψίζοντας όλα τα παραπάνω:

- **Tx_read**: Προσθέτουμε την Ταυτότητά μας στο read record. Αν το write record είναι κενό ή περιέχει μόνο την Ταυτότητά μας, διαβάζουμε την τιμή από την μνήμη, αλλιώς αφαιρούμε την Ταυτότητά μας από το read record και κάνουμε abort. Αν το read record δεν περιείχε την ταυτότητά μας, προσθέτουμε το record στο read-log.

- **Tx_write**: Προσθέτουμε την Ταυτότητά μας στο read και write record. Αν το read record ήταν

κενό ή περιείχε μόνο την Ταυτότητά μας, κάνουμε την εγγραφή, αλλιώς αφαιρούμε την Ταυτότητά μας από τα records και κάνουμε abort. Αν το write record ήταν κενό, προσθέτουμε την παλιά τιμή του word στο undo-log.

- **Tx_begin**: Δεν χρειάζεται να κάνουμε κάτι.
- **Tx_commit**: Διατρέχουμε το read-log και undo-log και αφαιρούμε την ταυτότητά μας από τα records.
- **Tx_abort**: Διατρέχουμε το undo-log και γράφουμε τις παλιές τιμές των words. Διατρέχουμε το read-log και undo-log και αφαιρούμε την ταυτότητά μας από τα records.

Δυστυχώς η θέση των records στην μνήμη εξαρτάται από την θέση του εκάστοτε word και είναι hard coded at compile time. Αυτό πρέπει να γίνει είτε από τον προγραμματιστή είτε από τον compiler. Η αλλαγή του compiler για να εκτελεί αυτήν την διαδικασία είναι κάτι πέρα από τα πλαίσια αυτής της διπλωματικής οπότε η υλοποίηση που παρουσιάζουμε απαιτεί από την προγραμματιστή να δημιουργεί και να αρχικοποιεί τα records. Έτσι έχουμε ένα ελαφρώς διαφορετικό interface από αυτό που παρουσιάσαμε στο κεφάλαιο 2.2 καθώς περνάμε το offset του record από το αντίστοιχο word ως παράμετρο:

```
void Tx_begin();
bool Tx_commit();
void Tx_abort();
T Tx_read(T *address, int offset);
void Tx_write(T *address, T value, int offset);
```

Με βάση το νέο interface, το παράδειγμα αύξησης μετρητή κατά ένα γίνεται:

```
1  struct {
2      long data = 0;
3      long metadata = 0;
4  } counter;
5  ...
6  do {
7      Tx_begin();
8      int temp = Tx_read(&counter.data, 8);
9      Tx_write(&counter.data, temp + 1, 8);
10 } while(!Tx_commit());
```

Το StandawayTM υποστηρίζει δυναμική μνήμη εντός transaction. Η υλοποίηση είναι απλή από την πλευρά του StandawayTM αλλά θέλει προσοχή στην χρήση από την πλευρά του προγραμματιστή. Τα allocation της μνήμης γίνονται στο χρόνο εκτέλεσης και, σε περίπτωση abort ελευθερώνονται αφού διορθωθούν τα words στις παλιές τους τιμές. Τα deallocation γίνονται σε περίπτωση commit στο τέλος της εκτέλεσης του. Αυτή η προσέγγιση είναι δυνατή καθώς όλα τα transactions βλέπουν μία consistent εικόνα της μνήμης και έχουμε Eager Version Managment.

3.1 Δομή Tx_Thread

Κάθε thread έχει μερικές προσωπικές πληροφορίες (thread local data). Στις παραπάνω συναρτήσεις περνάμε και έναν δείκτη στην δομή `Tx_Thread` που κρατάει αυτές τις πληροφορίες. Το TM είναι σχεδιασμένο για 64 bit όπου το μέγεθος των long είναι 8 bytes και το μέγεθος των int είναι 4 bytes. Η δομή έχει την εξής μορφή:

```c
typedef struct {
    unsigned int t_id; //Ο αριθμός του thread
    record_t mask; //Η Ταυτότητα του thread
    read_write_record_t read_long_mask; //Η ταυτότητα του thread σε long μορφή
    read_write_record_t read_write_mask; //Η ταυτότητα του thread για read και write records μα

    //undo-log
    int undo_set_pos;
    int undo_set_max;
    Tx_undo_set_t* undo_set;

    //read-log
    int read_set_pos;
    int read_set_max;
    meta_data_t ** read_set;

    jmp_buf jump_buffer; //Για επανάληψη μετά από abort

    //Διαχείριση allocation μνήμης
    int malloc_set_pos;
    int malloc_set_max;
    void ** malloc_set;

    //Διαχείριση deallocation μνήμης
    int free_set_pos;
    int free_set_max;
    void ** free_set;

    //Στατιστικά για aborts και commits
    unsigned long aborts;
    unsigned long commits;
}Tx_Thread;
```

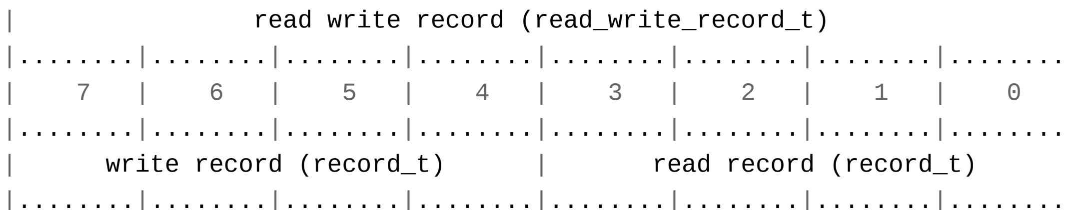
Με τις εξής συνοδευτικές δομές:

```c
typedef unsigned int record_t; //read ή write record
typedef unsigned long read_write_record_t; //read και write record μαζί

//read και write record
typedef struct {
    record_t write_record;
    record_t read_record;
}__attribute__((aligned((8)))) __attribute__((packed)) meta_data_mask_t;

//Δομή για αναφορά τόσο στα read και write records μαζί όσο και στο καθένα χωριστά
typedef union {
    meta_data_mask_t records;
    read_write_record_t bundle;
} __attribute__((aligned((8)))) __attribute__((packed)) meta_data_t;

```

```c
//Κομμάτι του undo-log μαζί με τα undo_set_pos και undo_set_max
typedef struct {
    uintptr_t address;
    meta_data_t *metadata;
    unsigned long value;
}Tx_undo_set_t;
```

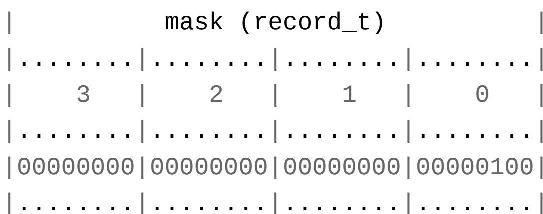
Στην ορθή λειτουργία του StandawayTM παίζει ρόλο ο συγχρονισμός με την χρήση ατομικών εντολών οι οποίες σηματοδοτούν και ορίζουν την σειρά με την οποία γίνονται οι αναγνώσεις και οι εγγραφές. Όπως θα δούμε παρακάτω, ο χρόνος που γίνεται μία ανάγνωση ή μία εγγραφή ορίζεται από τον χρόνο που εκτελείται η αλλαγή του αντίστοιχου record ακόμα και αν το ίδιο το word γίνεται accessed σε επόμενο χρόνο.

Ένας από τους λόγους που ο μέγιστος αριθμός threads είναι 32, είναι η χρήση atomic primitives για την διαχείρισή των records. Τα atomic primitives μπορούν να αναφερθούν το μέγιστο σε μεταβλητές μεγέθους 64 bits. Παρακάτω βλέπουμε την εικόνα ενός read και write record μαζί στην μνήμη.

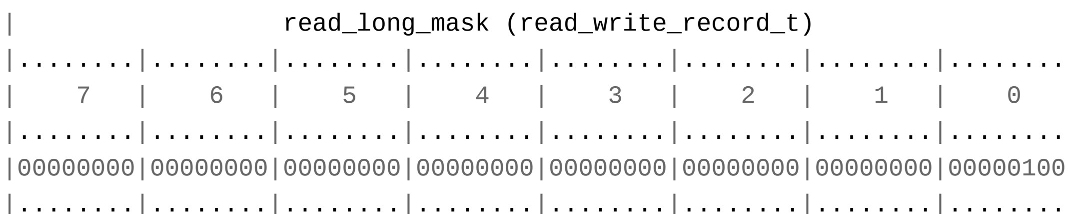


Μερικές από τις παραπάνω δομές δημιουργήθηκαν για να μπορούμε να αναφερόμαστε ταυτόχρονα στα δύο records μαζί αλλά και στο καθένα ξεχωριστά. Συγκεκριμένα το `meta_data_t.bundle` αφορά το `read write record`. Το `meta_data_t.records.write_record` αφορά το `write record` και `meta_data_t.records.read_record` το αφορά το `read record`. Η δομές `meta_data_t` και `meta_data_mask_t` έχουν μέγεθος ίσο με το `read write record`. Καθώς η αναφορά στα records γίνεται τόσο μέσα από το `meta_data_mask_t` αλλά και μέσα από το `record_t` μας απασχολεί η τοποθέτηση τους στην μνήμη. Οι δομές που φτιάχνουμε είναι `packed` δηλαδή δεν υπάρχει padding ανάμεσα στα δύο records, ενώ για αποφυγή Unaligned Memory Access Error (SIGBUS) όλα τα records έχουν alignment 8.

Υπάρχουν τρεις ταυτότητες που μας ενδιαφέρουν για να μπορούμε να διαχειριζόμαστε τα records με μία μόνο ατομική εντολή. Η βασική Ταυτότητα είναι η `mask`. Για παράδειγμα για το thread 2 (με αρχή το 0) είναι:



Η Ταυτότητα `read_long_mask` μας επιτρέπει να έχουμε πρόσβαση και στα δύο records αλλάζοντας μόνο το read records και για το thread 2 θα είναι η εξής:



Τέλος, η Ταυτότητα read_write_mask μας επιτρέπει να επηρεάζουμε και τις δύο μάσκες ταυτόχρονα. Υπενθυμίζουμε ότι κάθε write μετράει και ως read και για αυτό δεν έχουμε write_long_mask. Για το thread 2 (με αρχή το 0) το read_write_mask θα είναι το εξής:

```
|                    read_write_mask (read_write_record_t)                      |
|........|........|........|........|........|........|........|........|
|    7   |    6   |    5   |    4   |    3   |    2   |    1   |    0   |
|........|........|........|........|........|........|........|........|
|00000000|00000000|00000000|00000100|00000000|00000000|00000000|00000100|
|........|........|........|........|........|........|........|........|
```

Όλα τα logs (read-log, undo-log, malloc-log, free-log) αποτελούνται από έναν πίνακα (_set) από δομές που περιέχουν τα δεδομένα που ενδιαφέρουν το κάθε log και δύο αριθμούς. Ο πρώτος αριθμός (_pos) δείχνει την πρώτη κενή θέση του πίνακα και ο δεύτερος (_max) κρατάει το μέγεθος του πίνακα. Εάν ο πίνακας γεμίσει τότε ειδικές συναρτήσεις φροντίζουν να αυξήσουν την χωρητικότητα του εκάστοτε πίνακα. Για παράδειγμα το undo_set είναι ένας πίνακας από δομές Tx_undo_set_t. Όπως προαναφέραμε, το undo-log πρέπει να κρατάει την παλιά τιμή του word (unsigned long value;). Επίσης κρατάει την διεύθυνση του word (uintptr_t address;) καθώς και την διεύθυνση των αντίστοιχων records (meta_data_t *metadata;).

Στην αρχή του κάθε thread, η δομή αυτή πρέπει να αρχικοποιηθεί, και στο τέλος του thread οι αντίστοιχες δομές πρέπει να καταστραφούν. Ο παρακάτω κώδικας κάνει ακριβώς αυτό.

```
1  void Tx_init_thread(Tx_Thread* Self, unsigned int t_id){
2      Self->t_id = t_id;
3      //initialize masks
4      Self->mask = 1u << t_id;
5      meta_data_t mt;
6      mt.bundle = 0;
7      mt.records.read_record = Self->mask;
8      Self->read_long_mask = mt.bundle;
9      mt.records.write_record = Self->mask;
10     Self->read_write_mask = mt.bundle;
11     //initialize read log
12     Self->read_set_max = 1024;
13     Self->read_set_pos = 0;
14     Self->read_set = malloc(sizeof(meta_data_t *) * Self->read_set_max);
15     assert(Self->read_set);
16     //initialize undo log
17     Self->undo_set_max = 128;
18     Self->undo_set_pos = 0;
19     Self->undo_set = malloc(sizeof(Tx_undo_set_t) * Self->undo_set_max);
20     assert(Self->undo_set);
21     //initialize malloc log
22     Self->malloc_set_max = 32;
23     Self->malloc_set_pos = 0;
24     Self->malloc_set = malloc(sizeof(void*) * Self->malloc_set_max);
25     assert(Self->malloc_set);
26     //initialize free log
27     Self->free_set_max = 32;
28     Self->free_set_pos = 0;
29     Self->free_set = malloc(sizeof(void*) * Self->free_set_max);
30     assert(Self->free_set);
```

```
    //initialize statistics
    Self->aborts = 0;
    Self->commits = 0;
}
```

Η μεταβλητή t_id περιέχει τον αριθμό του thread που καλεί την Tx_init_thread. Η t_id είναι ένας αριθμός από το 0 ως το 31. Στις γραμμές 4 με 10 βλέπουμε την δημιουργία των Ταυτοτήτων με βάση αυτό το id. Κάθε id πρέπει να είναι μοναδικό.

```
void Tx_finalize_thread(Tx_Thread* Self){
    free(Self->read_set);
    free(Self->undo_set);
    free(Self->malloc_set);
    free(Self->free_set);
    atomic_fetch_add(total_aborts, Self->aborts);
    atomic_fetch_add(total_commits, Self->commits);
}
```

Οι μεταβλητές total_aborts και total_commits είναι global μεταβλητές αρχηκοποιημένες στο 0. Στο τέλος της εκτέλεσης έχουν αποθηκευμένα τα συνολικά aborts και commits όλων των threads.

3.2 Ανάγνωση

Παρακάτω βλέπουμε τον κώδικα για ανάγνωση από το thread με Tx_Thread Self ενός unsigned int, μεγέθους όσο μισό word στην θέση address με record σε απόσταση από την αρχή του word ίση με offset.

```
#define LONG_PTR_MASK ~0b111ul

unsigned int Tx_read_32u(Tx_Thread* Self,
                         unsigned int const *address_typed,
                         uintptr_t offset){
    uintptr_t address = (uintptr_t) address_typed;
    uintptr_t aligned_address = address & LONG_PTR_MASK;
    meta_data_t *meta_data = (meta_data_t*) (aligned_address + offset);

    meta_data_t prev_mask;
    prev_mask.bundle = atomic_fetch_or_explicit(&meta_data->bundle,
                                                Self->read_long_mask,
                                                memory_order_acquire);
    if((prev_mask.records.write_record == 0) ||
       (prev_mask.records.read_record & Self->mask) == Self->mask){
        if((prev_mask.records.read_record & Self->mask) == 0){
            Self->read_set[Self->read_set_pos++] = meta_data;
            if(Self->read_set_pos == Self->read_set_max)
                expand_read_set(Self);
        }
        return *address_typed;
    }
    atomic_fetch_and_explicit(&meta_data->bundle,
```

```
24                                      ~Self->read_long_mask,
25                                      memory_order_relaxed);
26      Tx_abort(Self);
27  }
```

Για να μπορέσουμε να κάνουμε αριθμητικές πράξεις με τους δείκτες, κάνουμε cast σε variable τύπου `uintptr_t`. Σύμφωνα με το C standard για το `uintptr_t`:

> an unsigned integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer.

Έτσι η μεταβλητή `address` περιέχει την διεύθυνση της μνήμης σε μορφή που επιτρέπει αριθμητικές πράξεις, ενώ η μεταβλητή `aligned_address` περιέχει την διεύθυνση του word που περιέχει την μεταβλητή που θέλουμε να διαβάσουμε. Η μεταβλητή `meta_data` είναι δείκτης στα records.

Στην γραμμή 11 υπάρχει το atomic primitive `atomic_fetch_or_explicit`. Σύμφωνα με το C reference manual:

> Atomically replaces the value pointed by obj with the result of bitwise OR between the old value of obj and arg, and returns the value obj held previously. The operation is read-modify-write operation.

Η εκτέλεση αυτής της ατομικής εντολής σηματοδοτεί χρονικά την εκτέλεση του Tx_read. Με άλλα λόγια, η σειρά εκτέλεσης του Tx_read σε σχέση με τις εντολές των υπόλοιπον Transactions είναι ισοδύναμη με την σειρά εκτέλεσης της ατομικής εντολής σε σχέση με τις αντίστοιχες ατομικές εντολές. Έτσι όλα τα Tx_reads και όπως θα δούμε παρακάτω τα Tx_writes είναι μεταξύ τους διατεταγμένα. Η εντολή στην γραμμή 11 προσθέτει την Ταυτότητα του thread στο read record, ενώ επιστρέφει στο `prev_mask` την τιμή του read και write record. Σύμφωνα με το C reference manual:

> A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.

Μεταφράζοντας, κανένα από τα reads ή writes σε αυτό το thread δεν μπορεί να τοποθετηθεί πριν από αυτό το load. Με άλλα λόγια, το read στο `*address_typed` θα γίνει μετά από την ατομική εντολή. Σημειώνουμε ότι η τιμή που επιστρέφει η ατομική εντολή είναι η αμέσως προηγούμενη του or που εκτελείται.

Ελέγχουμε το write record (την προηγούμενη τιμή του). Αν είναι ίσο με κενό ή έχουμε ξαναγράψει την θέση μνήμης, τότε η εγγραφή θα επιτύχει. Σημειώνουμε για τον έλεγχο της γραμμής 16: άλλα threads μπορεί να έχουν αλλάξει το record και να μην έχουν προλάβει να το επιστρέψουν στην προηγούμενη του μορφή πριν κάνουν abort. Έτσι ελέγχουμε μόνο το bit της Ταυτότητας και όχι για ισότητα του όλου record και της Ταυτότητας. Έπειτα, ελέγχουμε αν το read record πριν από το or είχε μέσα του την Ταυτότητά μας. Αν δεν την είχε, τότε είναι πρώτη φορά που αυτό το Transaction κάνει access το word, οπότε προσθέτουμε το record στο read log. Τέλος επιστρέφουμε την τιμή της μεταβλητής με διεύθυνση που δόθηκε.

Σε περίπτωση που κάποιος άλλος έχει γράψει την θέση που θέλουμε να διαβάσουμε, ο έλεγχος θα αποτύχει. Σε αυτήν την περίπτωση, αφαιρούμε την Ταυτότητά μας από το read record (την οποία μόλις βάλαμε) και κάνουμε abort. Καθώς πρέπει να κατασκευάσουμε διαφορετικές συναρτήσεις για κάθε τύπο μεταβλητής, κατασκευάσαμε ένα Template το οποίο χρησιμοποιούμε για να παράγουμε κώδικα:

```
#define READ_TEMPLATE(name, type) \
        type Tx_read_##name(Tx_Thread* Self, \
                            type const *address_typed, \
                            uintptr_t offset){ \
            uintptr_t address = (uintptr_t) address_typed; \
            uintptr_t aligned_address = address & LONG_PTR_MASK; \
            meta_data_t *meta_data = (meta_data_t*) (aligned_address + offset); \
            meta_data_t prev_mask; \
            prev_mask.bundle = atomic_fetch_or_explicit(&meta_data->bundle,
                                                        Self->read_long_mask,
                                                        memory_order_acquire); \
            if((prev_mask.records.write_record == 0) || \
               (prev_mask.records.read_record & Self->mask) == Self->mask){ \
                if((prev_mask.records.read_record & Self->mask) == 0){ \
                    Self->read_set[Self->read_set_pos++] = meta_data; \
                    if(Self->read_set_pos == Self->read_set_max) \
                        expand_read_set(Self); \
                } \
                return *address_typed;\
            } \
            atomic_fetch_and_explicit(&meta_data->bundle, \
                                      ~Self->read_long_mask, \
                                      memory_order_relaxed); \
            Tx_abort(Self); \
            return 0; \
        }

READ_TEMPLATE(8, char)
READ_TEMPLATE(8u, unsigned char)
READ_TEMPLATE(16, short)
READ_TEMPLATE(16u, unsigned short)
READ_TEMPLATE(32, int)
READ_TEMPLATE(32u, unsigned int)
READ_TEMPLATE(64, long)
READ_TEMPLATE(64u, unsigned long)
READ_TEMPLATE(f, float)
READ_TEMPLATE(d, double)
READ_TEMPLATE(p, void *)
```

Ενώ για την κλήση των συναρτήσεων χρησιμοποιήσαμε το _Generic controlling expression όπως φαίνεται παρακάτω:

```
#define Tx_read(Self, Address, offset) _Generic((Address), \
        long *: Tx_read_64, \
        unsigned long *: Tx_read_64u, \
        int *: Tx_read_32, \
        unsigned int *: Tx_read_32u, \
        short *: Tx_read_16, \
        unsigned short *: Tx_read_16u, \
        char *: Tx_read_8, \
```

```
        unsigned char *: Tx_read_8u, \
        float *: Tx_read_f, \
        double *: Tx_read_d, \
        default : Tx_read_p \
    )(Self, Address, (uintptr_t) offset)

char Tx_read_8(Tx_Thread* Self, const char *address, uintptr_t offset);
short Tx_read_16(Tx_Thread* Self, const short *address, uintptr_t offset);
int Tx_read_32(Tx_Thread* Self, const int *address, uintptr_t offset);
long Tx_read_64(Tx_Thread* Self, const long *address, uintptr_t offset);
unsigned char Tx_read_8u
    (Tx_Thread* Self, const unsigned char *address, uintptr_t offset);
unsigned short Tx_read_16u
    (Tx_Thread* Self, const unsigned short *address, uintptr_t offset);
unsigned int Tx_read_32u
    (Tx_Thread* Self, const unsigned int *address, uintptr_t offset);
unsigned long Tx_read_64u
    (Tx_Thread* Self, const unsigned long *address, uintptr_t offset);
float Tx_read_f(Tx_Thread* Self, const float *address, uintptr_t offset);
double Tx_read_d(Tx_Thread* Self, const double *address, uintptr_t offset);
void* Tx_read_p(Tx_Thread* Self, void * const *address, uintptr_t offset);
```

3.3 Εγγραφή

Ομοίως βλέπουμε τον κώδικα για εγγραφή του thread με `Tx_Thread Self` ενός `unsigned int`, μεγέθους όσο μισό word στην θέση `address` με record σε απόσταση από την αρχή του word ίση με `offset`.

```
#define LONG_PTR_MASK ~0b111ul

void Tx_write_32u(Tx_Thread *Self,
                  unsigned int* address_typed,
                  unsigned int value,
                  uintptr_t offset) {
    uintptr_t address = (uintptr_t) address_typed;
    uintptr_t aligned_address = address & LONG_PTR_MASK;
    meta_data_t *meta_data = (meta_data_t*) (aligned_address + offset);

    meta_data_t prev_mask;
    prev_mask.bundle = atomic_fetch_or_explicit(&meta_data->bundle,
                                                Self->read_write_mask,
                                                memory_order_acquire);

    if ((prev_mask.records.write_record & Self->mask) == Self->mask ||
        (prev_mask.records.read_record & ~Self->mask) == 0) {
        if ((prev_mask.records.write_record & Self->mask) == 0) {
            Self->undo_set[Self->undo_set_pos].address = aligned_address;
            Self->undo_set[Self->undo_set_pos].metadata = meta_data;
            Self->undo_set[Self->undo_set_pos].value =
                            *((unsigned long *) aligned_address);
```

```
23            if (++Self->undo_set_pos == Self->undo_set_max) expand_undo_set(Self);
24        }
25
26        *address_typed = value;
27        return;
28    }
29
30    atomic_fetch_and_explicit(&meta_data->bundle,
31                              ~Self->read_write_mask,
32                              memory_order_relaxed);
33    Tx_abort(Self);
34 }
```

Ομοίως με την Ανάγνωση, δημιουργούμε τους αντίστοιχους δείκτες και μεταβλητές για το word, τα records και την προηγούμενη τιμή των records. Για λόγους ευκολίας κάθε write θεωρείται και read. Στην γραμμή 12 υπάρχει το αντίστοιχο atomic primitive `atomic_fetch_or_explicit`. Η εκτέλεση αυτής της ατομικής εντολής σηματοδοτεί χρονικά την εκτέλεση του Tx_write. Με άλλα λόγια, η σειρά εκτέλεσης του Tx_write σε σχέση με τις εντολές των υπόλοιπον Transactions είναι ισοδύναμη με την σειρά εκτέλεσης της ατομικής εντολής σε σχέση με τις αντίστοιχες ατομικές εντολές. Έτσι, σε συνδυασμό με την αντίστοιχη ατομική εντολή στην σειρά 11 του Tx_read, όλα τα Tx_reads και τα Tx_writes είναι μεταξύ τους διατεταγμένα. Παρακάτω θα δούμε πώς το abort σχετίζεται με την σειρά αυτή.

Η εντολή στην γραμμή 12 προσθέτει την Ταυτότητα του thread στο read και write record, ενώ επιστρέφει στο `prev_mask` την προηγούμενη τιμή του read και write record.

Μεταφράζοντας, κανένα από τα reads ή writes σε αυτό το thread δεν μπορεί να τοποθετηθεί πριν από αυτό το load. Με άλλα λόγια, το write στο `*address_typed` θα γίνει μετά από την ατομική εντολή. Σημειώνουμε ότι η τιμή που επιστρέφει η ατομική εντολή είναι η αμέσως προηγούμενη του or που εκτελείται. Έτσι, τα Tx_read τα οποία δεν κάνουν abort, βλέπουν την τιμή που έχουν γράψει τα προηγούμενα writes, ενώ τα writes που δεν κάνουν abort αποτρέπουν όλα τα άλλα Tx_read από το να ολοκληρωθούν επιτυχώς.

Ελέγχουμε τα read και write records. Αν το write record (πριν το atomic or) περιέχει την Ταυτότητά μας, τότε έχουμε ξαναγράψει στο word και μπορούμε να γράψουμε ξανά. Σημειώνουμε ότι άλλα Transactions που τρέχουν ταυτόχρονα μπορούν να αλλάξουν το record αλλά θα κάνουν abort. Αν το read record έχει όλα τα άλλα bits 0, τότε πάλι μπορούμε να γράψουμε. Με άλλα λόγια μπορούμε να γράψουμε άμα δεν έχει διαβάσει κανείς άλλος το word ή αν έχουμε ήδη γράψει ξανά στο word. Αν είναι πρώτη φορά, προσθέτουμε τις αντίστοιχες πληροφορίες στο undo log. Τέλος, κάνουμε την εγγραφή στην μνήμη. Σημειώνουμε ότι κανένα άλλο Transaction δεν θα δει την νέα τιμή πριν αφαιρεθούν οι Ταυτότητές μας από τα records. Αν κάποιο άλλο Transaction έχει γράψει ή διαβάσει το word, η εγγραφή και κατά συνέπειά το Transaction θα αποτύχουν. Σε αυτήν την περίπτωση, αφαιρούμε την Ταυτότητά μας από το read και το write record (τις οποίες μόλις βάλαμε) και κάνουμε abort.

Καθώς πρέπει να κατασκευάσουμε διαφορετικές συναρτήσεις για κάθε τύπο μεταβλητής, κατασκευάσαμε ένα Template το οποίο χρησιμοποιούμε για να παράγουμε κώδικα:

```
1  #define WRITE_TEMPLATE(name, type) \
2          void Tx_write_##name(Tx_Thread *Self, \
3                               type* address_typed, \
4                               type value, \
5                               uintptr_t offset){ \
6              uintptr_t address = (uintptr_t) address_typed; \
7              uintptr_t aligned_address = address & LONG_PTR_MASK; \
8              meta_data_t *meta_data = (meta_data_t*) (aligned_address + offset); \
```

```
            meta_data_t prev_mask; \
            prev_mask.bundle = atomic_fetch_or_explicit(&meta_data->bundle, \
                                                        Self->read_write_mask, \
                                                        memory_order_acquire); \
            if((prev_mask.records.write_record & Self->mask) == Self->mask || \
               (prev_mask.records.read_record & ~Self->mask) == 0){ \
                if((prev_mask.records.write_record & Self->mask) == 0) { \
                    Self->undo_set[Self->undo_set_pos].address = aligned_address; \
                    Self->undo_set[Self->undo_set_pos].metadata = meta_data; \
                    Self->undo_set[Self->undo_set_pos].value =
                                  *((unsigned long*) aligned_address); \
                    if (++Self->undo_set_pos == Self->undo_set_max) \
                        expand_undo_set(Self); \
                } \
                *address_typed = value; \
                return; \
            } \
            atomic_fetch_and_explicit(&meta_data->bundle, \
                                      ~Self->read_write_mask, \
                                      memory_order_relaxed); \
            Tx_abort(Self); \
        }

WRITE_TEMPLATE(8, char)
WRITE_TEMPLATE(8u, unsigned char)
WRITE_TEMPLATE(16, short)
WRITE_TEMPLATE(16u, unsigned short)
WRITE_TEMPLATE(32, int)
WRITE_TEMPLATE(32u, unsigned int)
WRITE_TEMPLATE(64, long)
WRITE_TEMPLATE(64u, unsigned long)
WRITE_TEMPLATE(f, float)
WRITE_TEMPLATE(d, double)
WRITE_TEMPLATE(p, void *)
```

Ενώ για την κλήση των συναρτήσεων χρησιμοποιήσαμε το _Generic controlling expression όπως φαίνεται παρακάτω:

```
#define Tx_write(Self, Address, value, offset) _Generic((Address), \
        long *: Tx_write_64, \
        unsigned long *: Tx_write_64u, \
        int *: Tx_write_32, \
        unsigned int *: Tx_write_32u, \
        short *: Tx_write_16, \
        unsigned short *: Tx_write_16u, \
        char *: Tx_write_8, \
        unsigned char *: Tx_write_8u, \
        float *: Tx_write_f, \
        double *: Tx_write_d, \
        default: Tx_write_p \
    )(Self, Address, value, (uintptr_t) offset)
```

```
14
15 void Tx_write_8(Tx_Thread* Self, char *address, char value, uintptr_t offset);
16 void Tx_write_16(Tx_Thread* Self, short *address, short value, uintptr_t offset);
17 void Tx_write_32(Tx_Thread* Self, int *address, int value, uintptr_t offset);
18 void Tx_write_64(Tx_Thread* Self, long *address, long value, uintptr_t offset);
19 void Tx_write_8u
20     (Tx_Thread* Self, unsigned char *address, unsigned char value, uintptr_t offset);
21 void Tx_write_16u
22     (Tx_Thread* Self, unsigned short *address, unsigned short value, uintptr_t offset);
23 void Tx_write_32u
24     (Tx_Thread* Self, unsigned int *address, unsigned int value, uintptr_t offset);
25 void Tx_write_64u
26     (Tx_Thread* Self, unsigned long *address, unsigned long value, uintptr_t offset);
27 void Tx_write_f(Tx_Thread* Self, float *address, float value, uintptr_t offset);
28 void Tx_write_d(Tx_Thread* Self, double *address, double value, uintptr_t offset);
29 void Tx_write_p(Tx_Thread* Self, void* *address, void* value, uintptr_t offset);
```

3.4 Αρχή του Transaction

Ένα abort μπορεί να εκτελεστεί από οποιοδήποτε μέρος του Transaction και το Transaction πρέπει να επαναληφθεί. Έτσι η ροή εκτέλεσης του προγράμματος πρέπει στην αρχή του Transaction χωρίς να έχει επηρεαστεί η εσωτερική κατάσταση (internal state) του επεξεργαστή. Μία τέτοια λειτουργικότητα προσφέρει το ζευγάρι εντολών `setjmp()` και `longjmp`, σύμφωνο με POSIX.

Διαβάζοντας το manual:

SYNOPSIS

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

DESCRIPTION

[...] are used for performing "nonlocal gotos": transferring execution from one function to a predetermined location in another function. The `setjmp()` function dynamically establishes the target to which control will later be transferred, and `longjmp()` performs the transfer of execution.

The `setjmp()` function saves various information about the calling environment (usually, the stack pointer, the instruction pointer, possibly the values of other registers and the signal mask) in the buffer env for later use by `longjmp()`. In this case, `setjmp()` returns 0.

The `longjmp()` function uses the information saved in env to transfer control back to the point where `setjmp()` was called and to restore ("rewind") the stack to its state at the time of the `setjmp()` call. In addition, and depending on the implementation, the values of some other registers and the process signal mask may be restored to their state at the time of the `setjmp()` call.

Following a successful `longjmp()`, execution continues as if `setjmp()` had returned for a second time. This "fake" return can be distinguished from a true `setjmp()` call because the "fake" return returns the value provided in val. If the programmer mistakenly passes the value 0 in val, the "fake" return will instead return 1.

NOTE

If the function which called setjmp() returns before longjmp() is called, the behavior is undefined. Some kind of subtle or unsubtle chaos is sure to result.

Μεταφράζοντας περιληπτικά, η συνάρτηση `setjmp` ορίζει ένα σημείο στο πρόγραμμα στο οποίο μπορούμε να πάμε με χρήση του `longjmp`. Η `setjmp` αποθηκεύει πληροφορίες σε έναν `jump buffer` τον οποίο περνάμε ως μεταβλητή στο `longjmp`. Η `setjmp` επιστρέφει σε δύο περιπτώσεις, όταν εκτελείτε, όπου επιστρέφει 0, και όταν καλείτε η αντίστοιχη `longjmp` όπου επιστρέφει την μη μηδενική τιμή που περάσαμε στην `longjmp`. Για να μπορέσουμε να χρησιμοποιήσουμε την `longjmp`, πρέπει η αντίστοιχη `setjmp` να βρίσκεται στην στοίβα εκτέλεσης (call stack), ή πιο απλά η συνάρτηση που έχει καλέσει την `setjmp` πρέπει να μην έχει επιστρέψει. Πέρα από την κλήση της `setjmp`, η αρχή ενός Transaction δεν απαιτεί κάποια άλλη ενέργεια. Το παραπάνω μας περιορίζει στον τρόπο που μπορούμε να ορίσουμε την συνάρτηση `Tx_begin`. Η μία προσέγγιση μας ωθεί στην χρήση Ορισμών (`#define`):

```
#define Tx_begin(x) setjmp(x->jump_buffer)
```

όπου `x` δείκτης στην `Tx_Thread` δομή του thread. Η εναλλακτική είναι η χρήση του προσδιοριστή (specifier keyword) `inline` στον ορισμό της συνάρτησης `Tx_begin` στο αντίστοιχο header file:

```
inline void Tx_begin(Tx_Thread *Self){
    setjmp(Self->jump_buffer);
}
```

3.5 Committing Transaction

Τα `Tx_write` γράφουν απευθείας στην μνήμη οπότε οι ενέργειες που πρέπει να πάρουμε έχουν να κάνουν με διόρθωση των records και καθαρισμό των logs. (Θα αναφερθούμε στην διαχείριση μνήμης σε επόμενο κεφάλαιο.)

```
1  int Tx_commit(Tx_Thread* Self) {
2      atomic_thread_fence(memory_order_seq_cst);
3      //clear read write records
4      for (int i = 0; i < Self->undo_set_pos; ++i) {
5          atomic_fetch_and_explicit(&(Self->undo_set[i].metadata->bundle),
6                                    ~Self->read_write_mask,
7                                    memory_order_relaxed);
8      }
9      //clear read records
10     for (int i = 0; i < Self->read_set_pos; ++i) {
11         atomic_fetch_and_explicit(&(Self->read_set[i]->records.read_record),
12                                   ~Self->mask,
13                                   memory_order_relaxed);
14     }
15     Self->read_set_pos = 0;
16     Self->undo_set_pos = 0;
17     Self->commits++;
18     return 1;
19 }
```

Προτού καθαρίσουμε τα records φροντίζουμε όλες οι εγγραφές να έχουν περαστεί στην μνήμη. Αυτό γίνεται με χρήση ενός φράχτη (fence) όπως βλέπουμε στην γραμμή 2. Έπειτα, διατρέχουμε

τις εγγραφές των undo log και read log και αφαιρούμε την Ταυτότητά μας από τα αντίστοιχα records. Τέλος καθαρίζουμε τα logs (γραμμές 15 και 16), και ενημερώνουμε τα στατιστικά του Thread (γραμμή 17).

3.6 Aborting Transaction

Σε αντίθεση με ένα επιτυχημένο Transaction, όταν ένα Transaction αποτύχει, οι αλλαγές στην μνήμη πρέπει να αναιρεθούν. Η υπόλοιπες διαδικασίες είναι παρόμοιες.

```
1  void Tx_abort(Tx_Thread* Self){
2      for (int i = 0; i < Self->undo_set_pos; ++i) {
3          *((unsigned long *) Self->undo_set[i].address) = Self->undo_set[i].value;
4          atomic_fetch_and_explicit(&(Self->undo_set[i].metadata->bundle),
5                                    ~Self->read_write_mask,
6                                    memory_order_release);
7      }
8      for (int i = 0; i < Self->read_set_pos; ++i) {
9          atomic_fetch_and_explicit(&(Self->read_set[i]->records.read_record),
10                                   ~Self->mask,
11                                   memory_order_relaxed);
12     }
13     Self->read_set_pos = 0;
14     Self->undo_set_pos = 0;
15     Self->aborts++;
16     longjmp(Self->jump_buffer, 1);
17 }
```

Για κάθε εγγραφή του undo log, επαναφέρουμε την μνήμη στην σωστή της τιμή (γραμμή 3), και αφαιρούμε από το αντίστοιχο record την Ταυτότητά μας. Σύμφωνα με το C reference:

> A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable (see Release-Acquire ordering below) and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.

Μεταφράζοντας, όλες οι εγγραφές στην μνήμη πριν από την ατομική εντολή είναι ορατές στα άλλα threads. Έτσι, τα υπόλοιπα threads θα δουν την διορθωμένη τιμή, πριν δουν το record χωρίς την Ταυτότητά μας. Η ατομική εντολή στην γραμμή 4, ορίζει την σειρά με την οποία γίνονται οι διορθώσεις στην μνήμη, σε σχέση με όλες τις αναγνώσεις και εγγραφές. Τα read records και τα logs επαναφέρονται όμοια με το αντίστοιχο `Tx_commit`, ομοίως και τα στατιστικά. Τέλος, το αποτυχημένο Transaction πρέπει να επαναληφθεί. Εκτελούμε longjmp στο `Tx_begin` και το Transaction αρχίσει εκ νέου.

3.7 Διαχείριση μνήμης

Η διαχείριση μνήμης είναι μία ιδιαίτερη διαδικασία στα TMs. Ο προγραμματιστής δεν μπορεί να χρησιμοποιήσει το interface που του δίνει το λειτουργικό. Η κατανομή (allocation) μνήμης πρέπει να αναιρείται σε περίπτωση abort και η απαλλαγή (deallocation) μνήμης πρέπει να ολοκληρώνεται φροντίζοντας τα threads να μην μπορούν να αναφερθούν στις αντίστοιχες θέσεις.

Το StandawayTM εκπληρώνει τις παραπάνω προϋποθέσεις, εμφολεύοντας (wrapping) τις αντίστοιχες κλίσεις προς την standard c library (malloc, free). Όταν ένα Transaction ολοκληρωθεί, ανάλογα με το αν επιτύχει η αποτύχει, ολοκληρώνεται η αντίστοιχη διαχείριση μνήμης. Όλα τα deallocations

μεταφέρονται στο τέλος των committing Transaction και όλα τα allocations ακυρώνονται στο τέλος των aborting Transactions.

Οι wrappers των αντίστοιχων κλίσεων αποθηκεύουν σε logs ιστορικό των θέσεων που έχουν γίνει allocated ή θα γίνουν deallocated.

```
void* Tx_malloc(Tx_Thread *Self, size_t size){
    void* res = malloc(size);
    if(!res) return NULL;
    Self->malloc_set[Self->malloc_set_pos++] = res;
    if(Self->malloc_set_pos == Self->malloc_set_max) expand_malloc_list(Self);
    return res;
}
```

```
void Tx_free(Tx_Thread *Self, void* ptr){
    Self->free_set[Self->free_set_pos++] = ptr;
    if(Self->free_set_pos == Self->free_set_max) expand_free_list(Self);
}
```

Ομοίως με τα logs, read και undo, τα free και malloc logs μπορούν να επεκταθούν, (αν και κάτι πολύ σπάνιο) με χρήση αντίστοιχων συναρτήσεων.

```
#define EXPAND_TEMPLATE(func) void expand_##func##_list(Tx_Thread* Self){ \
        Self->func##_set = realloc(Self->func##_set, \
                                   Self->func##_set_max * 2 * sizeof(void*)); \
        if(!Self->func##_set){ \
            fprintf(stderr, "run out of memory in %s_set\n", #func); \
            exit(-1); \
        } else \
            Self->func##_set_max = Self->func##_set_max * 2; \
}

EXPAND_TEMPLATE(malloc);
EXPAND_TEMPLATE(free);
```

Στο τέλος του `Tx_commit` υπάρχει κώδικας, ο οποίος ολοκληρώνει το deallocation της μνήμης διατρέχοντας το free log, και επαναφέρει τα malloc και free logs.

```
void Tx_commit(Tx_Thread* Self){
    ...
    for (int i = 0; i < Self->free_set_pos; ++i) {
        free(Self->free_set[i]);
    }
    Self->malloc_set_pos = 0;
    Self->free_set_pos = 0;
}
```

Ενώ στο τέλος του `Tx_abort` υπάρχει κώδικας που αναιρεί το allocation της μνήμης διατρέχοντας το malloc log, και ομοίως επαναφέρει τα malloc και free logs. Τελευταία εντολή στο `Tx_abort` παραμένει το longjmp καθώς ο κώδικας μετά το longjmp είναι απροσπέλαστος.

```
void Tx_abort(Tx_Thread* Self){
    ...
    for (int i = 0; i < Self->malloc_set_pos; ++i) {
        free(Self->malloc_set[i]);
    }
    Self->malloc_set_pos = 0;
    Self->free_set_pos = 0;
    longjmp(Self->jump_buffer, 1);
}
```

3.8 Livelock

Καθώς ολοκληρώνουμε την παρουσίαση του κώδικα και το StandawayTM έχει πάρει μορφή, αρχίζει να φαίνεται και μία βασική του αδυναμία. Έστω δύο Transactions. Το πρώτο διαβάζει την μεταβλητή A και γράφει μετά από λίγο γράφει στην μεταβλητή B. Το δεύτερο διαβάζει την μεταβλητή B και μετά από λίγο γράφει στην μεταβλητή A. Τα δύο αυτά Transactions θα κάνουν συνέχεια abort, το ένα εξαιτίας του άλλου, και καθώς θα επαναλαμβάνονται το πρόγραμμα παρότι δεν θα κολλήσει (deadlock, freeze) θα τρέχει αέναα σε κύκλο και θα παρουσιάζει πρόοδο. Αυτό το πρόβλημα μπορεί να σχηματιστεί αντίστοιχο με περισσότερα Transactions και περισσότερες μεταβλητές.

Για να λύσουμε αυτό το πρόβλημα, πρέπει να δώσουμε προτεραιότητα σε ένα εκ των Transactions. Δημιουργούμε λοιπόν ένα mutex, το οποίο μπορεί να έχει μόνο ένα Transaction σε όλο το πρόγραμμα κάθε στιγμή. Αν ένα Transaction έχει το mutex, τότε σε περίπτωση που θα έπρεπε να κάνει abort, επιμένει επαναλαμβάνοντας την προσπάθεια ανάγνωσης ή εγγραφής μετά από ελάχιστο χρόνο (για να μην επιβαρύνει πολύ τις caches). Έτσι, το νέο Tx_read έχει την εξής μορφή:

```
#define READ_TEMPLATE(name, type) \
        type Tx_read_##name(Tx_Thread* Self, \
                            type const *address_typed, \
                            uintptr_t offset){ \
            uintptr_t address = (uintptr_t) address_typed; \
            uintptr_t aligned_address = address & LONG_PTR_MASK; \
            meta_data_t *meta_data = (meta_data_t*) (aligned_address + offset); \
            meta_data_t prev_mask; \
        again: \
            prev_mask.bundle = atomic_fetch_or_explicit(&meta_data->bundle, \
                                                        Self->read_long_mask, \
                                                        memory_order_acquire); \
            if((prev_mask.records.write_record == 0) || \
               (prev_mask.records.write_record & Self->mask) == Self->mask){ \
                if((prev_mask.records.read_record & Self->mask) == 0){ \
                    Self->read_set[Self->read_set_pos++] = meta_data; \
                    if(Self->read_set_pos == Self->read_set_max) \
                        expand_read_set(Self); \
                } \
                return *address_typed;\
            } \
            atomic_fetch_and_explicit(&meta_data->bundle, \
                                      ~Self->read_long_mask, \
                                      memory_order_relaxed); \
            if ((Self->locked == true) || \
```

```
26                  (pthread_mutex_trylock(&persistent_lock) == 0)) { \
27                  Self->locked = true; \
28                  int k = WAIT_TIME; \
29                  do { k--; } while(k!=0); \
30                  goto again; \
31              } \
32              Tx_abort(Self); \
33              return 0; \
34          }
```

Και το νέο Tx_write, την εξής μορφή:

```
1   #define WRITE_TEMPLATE(name, type) \
2           void Tx_write_##name(Tx_Thread *Self, \
3                                type* address_typed, \
4                                type value, \
5                                uintptr_t offset){ \
6               uintptr_t address = (uintptr_t) address_typed; \
7               uintptr_t aligned_address = address & LONG_PTR_MASK; \
8               meta_data_t *meta_data = (meta_data_t*) (aligned_address + offset); \
9               meta_data_t prev_mask; \
10          again: \
11              prev_mask.bundle = atomic_fetch_or_explicit(&meta_data->bundle, \
12                                                          Self->read_write_mask, \
13                                                          memory_order_acquire); \
14              if((prev_mask.records.write_record & Self->mask) == Self->mask || \
15                 (prev_mask.records.read_record & ~Self->mask) == 0){ \
16                  if((prev_mask.records.write_record & Self->mask) == 0) { \
17                      Self->undo_set[Self->undo_set_pos].address = aligned_address; \
18                      Self->undo_set[Self->undo_set_pos].metadata = meta_data; \
19                      Self->undo_set[Self->undo_set_pos].value = \
20                                    *((unsigned long*) aligned_address); \
21                      if (++Self->undo_set_pos == Self->undo_set_max) \
22                          expand_undo_set(Self); \
23                  } \
24                  *address_typed = value; \
25                  return; \
26              } \
27              atomic_fetch_and_explicit(&meta_data->bundle, \
28                                        ~Self->read_write_mask, \
29                                        memory_order_relaxed); \
30              if ((Self->locked == true) || \
31                  (pthread_mutex_trylock(&persistent_lock) == 0)) { \
32                  Self->locked = true; \
33                  int k = 1000; \
34                  do { k--; } while(k!=0); \
35                  goto again; \
36              } \
37              Tx_abort(Self); \
38          }
```

Το mutex `persistent_lock` γίνεται initialized στην αρχή του προγράμματος, ενώ η ελευθέρωσή (unlock) του, γίνεται κάποια στιγμή μέσα στο Tx_commit. Η μεταβλητή `Self->locked` αποθηκεύει την πληροφορία του αν το συγκεκριμένο thread έχει γίνει persistent, δηλαδή αν το thread θα ολοκληρωθεί επιτυχώς. Το `Self->locked` γίνεται false στην αρχή του Transaction.

Με τα ίδια εργαλεία μπορούμε να κατασκευάσουμε και irreversible Transactions, δηλαδή Transactions που κάνουν πάντα commit. Φυσικά, μπορεί να τρέχει μόνο ένα irriversible Transaction κάθε στιγμή, και όλα τα άλλα κανονικά. Αυτή η μορφή Transaction είναι πολύ χρήσιμη σε περίπτωση που θέλουμε να επικοινωνήσουμε με το περιβάλλον ή να προκαλέσουμε κάποια πράξη που απλά η επιστροφή της μνήμης στην προηγούμενη κατάσταση δεν είναι αρκετή. Τέτοιες περιπτώσεις είναι συνήθως κλήσεις συστήματος και εγγραφές σε αρχεία, στο δίκτυο, ή στην οθόνη. Ο τρόπος που έχουμε δομήσει το StandawayTM μας επιτρέπει να διαλέξουμε να κάνουμε ένα Transaction irriversible μία οποιαδήποτε στιγμή (και όχι υποχρεωτικά από την αρχή του Transaction), όμως μία τέτοια πράξη μπορεί να αποτύχει.

Κεφάλαιο 4

Μεθοδολογία και Πειράματα

Το κομμάτι της αξιολόγησης μίας τέτοιας πρότασης, αφορά τρία μέρη. Την υλοποίηση της πρότασης, την αντιστοιχία της με ένα ισοδύναμο μοντέλο, και την σύγκριση των δύο υπό κοινή βάση.

4.1 StandawayTM

Το StandawayTM (https://github.com/volglizolic/PipinoSTMv3) έχει υλοποιηθεί σε C, μία γλώσσα με μερικές πολύ χρήσιμες λειτουργίες και ιδιότητες.

- **Low Level**: Το StandawayTM δίνει έμφαση στις επιδόσεις και η C είναι μία γλώσσα που μας επιτρέπει να καταλάβουμε εύκολα πως ο κώδικάς μας θα μεταφραστεί σε γλώσσα μηχανής και ποιο το κόστος κάθε εντολής. Επίσης η C μας επιτρέπει να δούμε την μνήμη ως έναν τεράστιο πίνακα, κάτι ιδιαίτερα χρήσιμο αν αναλογιστούμε τον τρόπο που το StandawayTM βλέπει τον κόσμο.

- **Αριθμητική δεικτών (pointer arithmetic)**: Τα records είναι τοποθετημένα σε θέσεις μνήμης σχετικές με τα words που τα αφορούν. Ο προγραμματιστής όμως δεν αναφέρεται πάντα σε words αλλά συχνά σε κομμάτια τους. Το StandawayTM αντιστοιχεί τις διευθύνσεις των μεταβλητών σε διευθύνσεις των words και των αντίστοιχων record τους. Αυτή η διαδικασία για να γίνει γρήγορα και αποδοτικά προϋποθέτει απλές πράξεις μεταξύ pointers.

- **Ευέλικτο Casting και Unions**: Το StandawayTM εναλλάσσεται μεταξύ αναφορών σε μεμονωμένα read records, write records και τον συνδυασμό τους ως read-write records. Αυτές οι αναφορές συχνά πρέπει να είναι ατομικές και να αφορούν μόνο μία θέση μνήμης. Η C μας επιτρέπει να είμαστε ευέλικτη στις αναφορές μας στην μνήμη, αλλά είναι και αρκετά αυστηρή ώστε να μας αποτρέπει από λάθη σε αναφορές μνήμης.

- **Atomic operations**: Το 2011 ορίστηκε το πρότυπο της C11, το οποίο όρισε για πρώτη φορά την έννοια του μοντέλου μνήμης (memory model) για παράλληλα συστήματα και της πολυνηματικής επεξεργασίας (multithreaded execution). Παρότι multithreaded προγράμματα γραμμένα σε C υπήρχαν πολύ πριν το 2011, η υλοποίησή τους βασίζοντας σε άλλα standards όπως το POSIX.

- **Compiler attributes**: Για την υλοποίηση χρησιμοποιούμε τον gcc gnu compiler, ο οποίος μας δίνει ένα interface για να χειριζόμαστε την τοποθέτηση των διάφορων δομών στην μνήμη μέσα από το keyword `__attribute__`. Η μία χρήση (`__attribute__((aligned((8)))))` αφορά την ευθυγράμμιση (alignment) των records στην μνήμη. Η χρήση αριθμητικής δεικτών εμπεριέχει τον κίνδυνο για αναφορά σε διευθύνσεις οι οποίες δεν είναι aligned κατά το μέγεθος του read-write record, κάτι που είτε θα οδηγούσε στον τερματισμό του προγράμματος, είτε σε πολύ αργή εκτέλεσή του. Η άλλη χρήση (`__attribute__((packed)))` αφορά τον ορισμό των records ως συνεχόμενες θέσης μνήμης, και την τοποθέτηση των records στην μνήμη μέσα στις δομές του κυρίως προγράμματος. Με την χρήση του `__attribute__((packed))` αποτρέπουμε τον compiler από την τοποθέτηση padding. Τα read και write records βρίσκονται σε διπλανές θέσεις ώστε να είναι δυνατή η αναφορά και στα δύο μαζί. Έτσι δεν πρέπει να έχουν padding

μεταξύ τους. Το πρόγραμμα στο οποίο θέλουμε να χρησιμοποιήσουμε το StandawayTM, ορίζει και αρχικοποιεί τα records. Καθώς τα records πρέπει να μηδενιστούν και θέλουμε να έχουμε ίδιο αριθμό words και records, αποφασίσαμε να κάνουμε χρήση του `__attribute__((packed))` για πιο γρήγορη υλοποίηση και να τοποθετήσουμε τα records μέσα στην εκάστοτε δομή που κάνουμε access transactionally.

4.2 Σύγκριση με TL2

Για να συγκρίνουμε το StandawayTM επιλέξαμε το TL2[Dice06] για διάφορους λόγους. Στην έρευνα για τα STM το TL2 έχει χρησιμοποιηθεί εκτενώς ως μέτρο σύγκρισης, οπότε ακολουθούμε αυτήν την προσέγγιση. Επίσης υπάρχει έτοιμη υλοποίηση του TL2 σε C, συμβατή με το STAMP [Minh08].

Το TL2 χρησιμοποιεί διαφορετικές πολιτικές (σχεδιαστικές επιλογές) από το StandawayTM όμως έρευνες[Dice07, Abba10] έχουν δείξει ότι η συμπεριφορά των benchmarks παραμένει παρόμοια ανεξάρτητα των πολιτικών που ακολουθούνται.

Εδώ πρέπει να σημειώσουμε ότι παρότι στην γενική περίπτωση διαφορετικές πολιτικές έχουν παρόμοιες επιδόσεις, η κάθε μία είναι πιο επιρρεπής ή όχι σε μερικές παθογένειες των TM συστημάτων. Παρακάτω θα δούμε πως αυτές οι παθογένειες επηρεάζουν την επίδοσή μας, και τι μπορούμε να κάνουμε για αυτό.

4.3 STAMP Μετροπρογράμματα

Η αξιολόγησή των TMs ήταν ανέκαθεν ένα σημείο ενδιαφέροντος για την ακαδημαϊκή κοινότητα. Οι πρώτες προσπάθειες είχαν την μορφή μετατροπής benchmarks για παράλληλη επεξεργασία, από βασισμένη σε κλειδώματα (lock-based) υλοποίηση σε αντίστοιχη βασισμένη σε Transactions. Ταυτόχρονα, δημιουργήθηκαν αρκετά micro-benchmarks απλών δομών με αντίστοιχη επεξεργασία πάνω τους όπως λίστες και δέντρα. Όμως καμία από τις δύο προσεγγίσεις δεν κατάφερε να προσομοιώσει τον τρόπο που ένας προγραμματιστής θα χρησιμοποιούσε το TM ως μέρος ενός μεγάλου προγράμματος.

Το 2008 δημοσιεύθηκε το STAMP[Minh08]. Ένα benchmark suite με σκοπό να ελέγχει την απόδοση των διάφορων TMs (STM και HTM) τόσο σε σχέση με σειριακή εκτέλεση όσο και σε σχέση με άλλα TMs. Από τότε το STAMP έχει επικρατήσει ως το standard benchmark για την αξιολόγησή τους. Το STAMP αποτελείται από οχτώ προγράμματα, πλήρως παραμετροποιήσιμα, με διάφορα memory access patterns και Transactions διαφορετικών μεγεθών.

Παρότι το STAMP μας δείχνει μία καλή εικόνα για την συγκριτική ταχύτητα των διαφόρων TMs, δεν μας δίνει επίγνωση των διάφορων μικροσυμβάντων και παθολογιών του εκάστοτε TM. Για αυτό, πρέπει να βασιστούμε στην δική μας διορατικότητα και εμπειρία, και να εξηγήσουμε διάφορες περιπτώσεις όπως θα δούμε σε επόμενα κεφάλαια.

Στο κεφάλαιο 3.1 μιλήσαμε για την διεπαφή επικοινωνίας μεταξύ του προγράμματος και ενός τυχαίου TM συστήματος. Στο κεφάλαιο 4 ορίσαμε την διεπαφή του StandawayTM και αναφέραμε ότι υπό αυτήν την υλοποίηση ο προγραμματιστής είναι υπεύθυνος για την διαχείριση μνήμης για τα records καθώς και την αρχικοποίηση τους. Για να μπορεί το STAMP να είναι συμβατό με το StandawayTM , αλλάξαμε κατάλληλα το STAMP. Παρατηρούμε ότι κομμάτι του StandawayTM βρίσκεται πλέον στο STAMP και μπορούμε να παρακολουθήσουμε τις επιπτώσεις της αρχικής διαχείρισης μνήμης στην επίδοση των benchmarks.

Στο STAMP παρατηρήσαμε τριών ειδών δεδομένα με transactional accesses. Απλές μεταβλητές, πίνακες και structs. Για κάθε μία ακολουθήσαμε μια ελαφρώς διαφορετική τακτική:

- **Απλές μεταβλητές**: Για την μεταβλητή x τύπου T δημιουργούμε ένα struct και ορίζουμε και την x με τον εξής τρόπο:

```c
typedef struct {
    T data;
    char padding[sizeof(read_write_record_t) - sizeof(T)];
    read_write_record_t metadata;
}__attribute__((packed)) __attribute__((aligned((8))))
                                x_bundle = {.metadata = 0};
#define x x_bundle.data
```

Με το record να έχει offset ίσο με 8.

- **Στατικοί Πίνακες**: Για τους στατικούς πίνακες ορίζουμε νέο struct παρόμοιο με τις απλές μεταβλητές και κατασκευάζουμε τους πίνακες από αυτή την μεταβλητή.
- **Δυναμικοί Πίνακες**: Για τον δυναμικό πίνακα A συνολικού μεγέθους _m_size αρχικά ορίζουμε το μέγεθος aligned κατά 8 προς τα πάνω m_size με or στα 3 τελευταία bits και προσθέτοντας ένα. Κάνουμε calloc τον πίνακα A με συνολικό μέγεθος 2*m_size. Το offset του πίνακα A είναι ίσο με m_size.
- **Structs**: Για τα structs βρίσκουμε το μέγεθός τους και προσθέτουμε ανάλογο padding για να γίνουν 8 bytes alighed. Έπειτα προσθέτουμε ίσο χώρο για metadata με το μέγεθος των δεδομένων συν το padding. Το struct είναι packed και aligned 8. Το offset υπολογίζεται ανά περίπτωση αλλά είναι ίσο με το μέγεθος των metadata. Για παράδειγμα ο κόμβος μίας διπλά συνδεδεμένης λίστας από integers είναι ο εξής:

```c
typedef struct node_t{
    struct node_t *next;
    int data;
    char padding[sizeof(read_write_record_t) - sizeof(data)];
    struct node_t *prev;
    read_write_record_t metadata[3];
}__attribute__((packed)) __attribute__((aligned((8))));
#define node_offset 3*8
```

4.4 Υπολογιστικό Σύστημα

Το υπολογιστικό σύστημα στο οποίο τρέξαμε τα πειράματά μας είναι το εξής:

- CPU: AMD Ryzen 7 1700
- RAM: 16GB DDR4 2667MHz
- OS: Fedora Linux Workstation 32

Ο AMD Ryzen 7 1700 έχει 8 υπολογιστικούς πυρήνες και 16 λογικούς με hyperthreading στα 3GHz. Συνολική L1 cache 768KB, συνολική L2 cache 4MB και Συνολική L3 cache 16MB.

Κεφάλαιο 5

Μετρήσεις και Αποτελέσματα

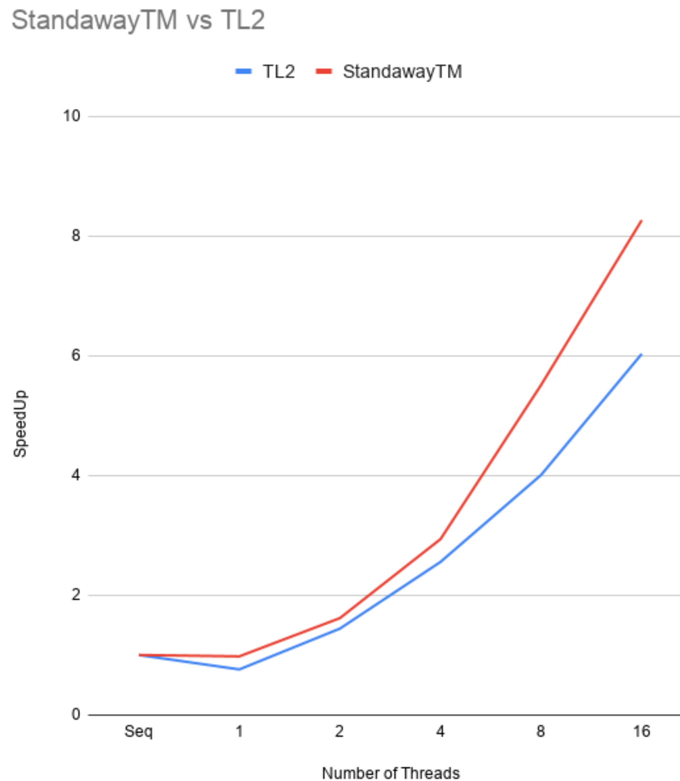
Για να τρέξουμε τις μετρήσεις πρέπει να τροποποιήσουμε αρκετά το STAMP. Δυστυχώς αυτή είναι μία χρονοβόρα διαδικασία και πολλές φορές μπορεί να οδηγήσει σε λάθη και δεν μπορέσαμε να μετατρέψουμε όλα τα benchmarks. Έτσι, από τα 8 benchmarks της σουίτας αυτής χρησιμοποιούμε μόνο τα 5. Συγκεκριμένα θα παρουσιάσουμε τα Genome, K-means, Labyrinth, Ssca2 και Vacation ενώ τα Bayes, Intruder και Yada δεν θα παρουσιαστούν καθώς προέκυψαν διάφορα προβλήματα.

Όπως αναφέρθηκε, το υπολογιστικό σύστημα που χρησιμοποιούμε είναι hyperthreaded. Έτσι στα τεστ με 16 threads, ανά δύο threads μοιράζονται επεξεργαστική ισχύ. Επίσης κάθε 4 cores αποτελούν ένα σύμπλεγμα με κοινή cache. Αυτές οι μικροαρχιτεκτονικές λεπτομέρειες παίζουν μεγάλο ρόλο στην απόδοση του StandawayTM, περισσότερο ίσως από άλλα TMs καθώς βασίζεται αρκετά σε atomic operations.

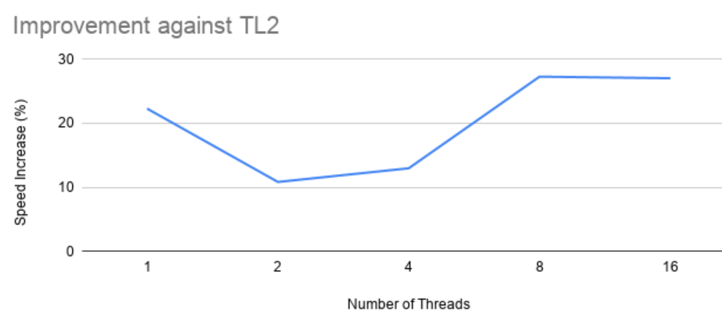
Παρακάτω παρουσιάζουμε δύο ειδών μετρήσεις. Οι πρώτες αφορούν την αύξηση της ταχύτητας (speed-up) συγκριτικά με την σειριακή εκτέλεση του benchmark χωρίς χρήση TM συστήματος με ένα thread. Οι δεύτερες αφορούν την βελτίωση του StandawayTMσε σύγκριση με το TL2.

5.1 Genome

Στο Genome βλέπουμε δύο χαρακτηριστικά που υποσχεθήκαμε για το StandawayTM. To StandawayTM είναι πιο γρήγορο και έχει μικρότερο impact στην σειριακή εκτέλεση.



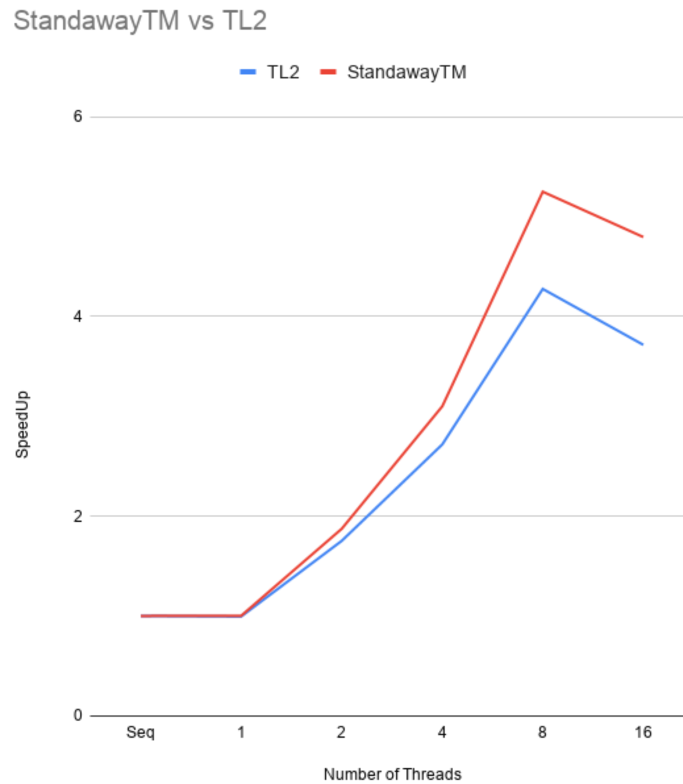
Σχήμα 5.1: Speedup of TL2 and StandawayTM in Genome



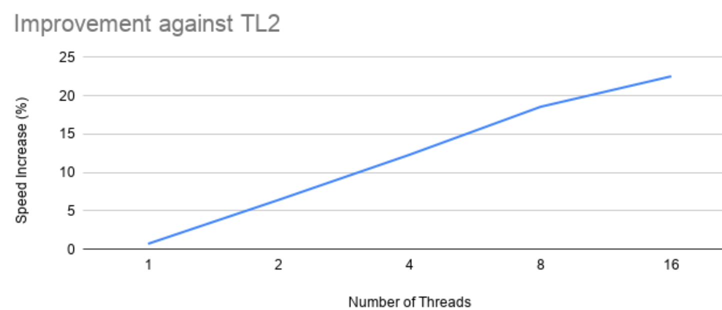
Σχήμα 5.2: Speed Improvement of StandawayTM compared to TL2 in Genome

5.2 Labyrinth

Στο Labyrinth βλέπουμε ότι το StandawayTM είναι πιο γρήγορο. Καθώς μεγάλο μέρος της εκτέλεσης στο Labyrinth είναι τοπικές πράξεις παρά Transactional accesses η εκτέλεση με ένα thread και TM είναι παρόμοια με την σειριακή. Επίσης η εκτέλεση στα 16 threads είναι πιο αργή από την αντίστοιχη στα 8 threads λόγο hyperthreading.



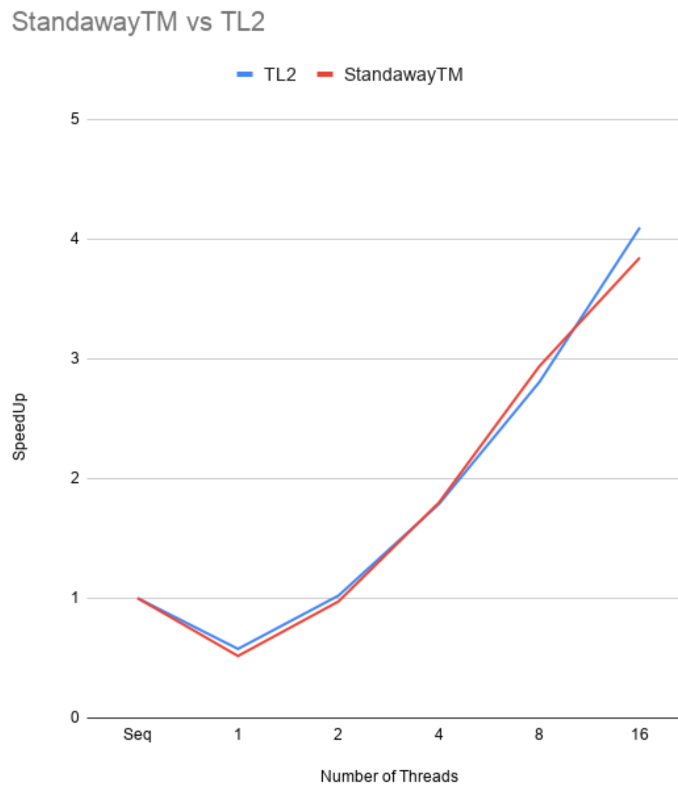
Σχήμα 5.3: Speedup of TL2 and StandawayTM in Labyrinth



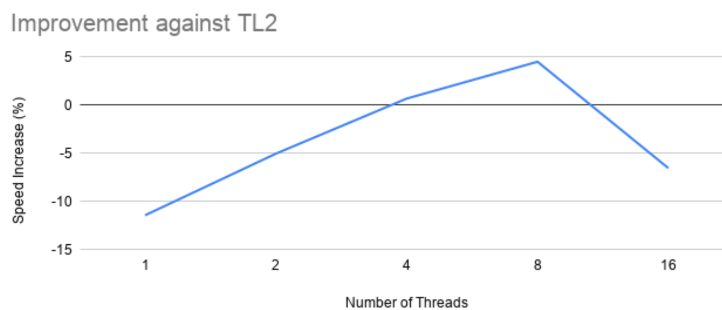
Σχήμα 5.4: Speed Improvement of StandawayTM compared to TL2 in Labyrinth

5.3 SSCA2

Στο ssca2 γίνεται εμφανές το πρώτο μειονέκτημα του StandawayTM. Το ssca2 εκμεταλλεύεται την τοπικότητα των δεδομένων. Με το StandawayTM μειώνουμε ουσιαστικά (effectively) τις caches στο μισό. Έτσι το StandawayTM δυσκολεύεται να διατηρήσει το προβάδισμά του μικρού overhead. Παρακάτω θα παρουσιάσουμε πιθανές λύσεις για αυτά τα προβλήματα όμως αυτές πάνε πέρα από τα πλαίσια της διπλωματικής αυτής.



Σχήμα 5.5: Speedup of TL2 and StandawayTM in ssca2

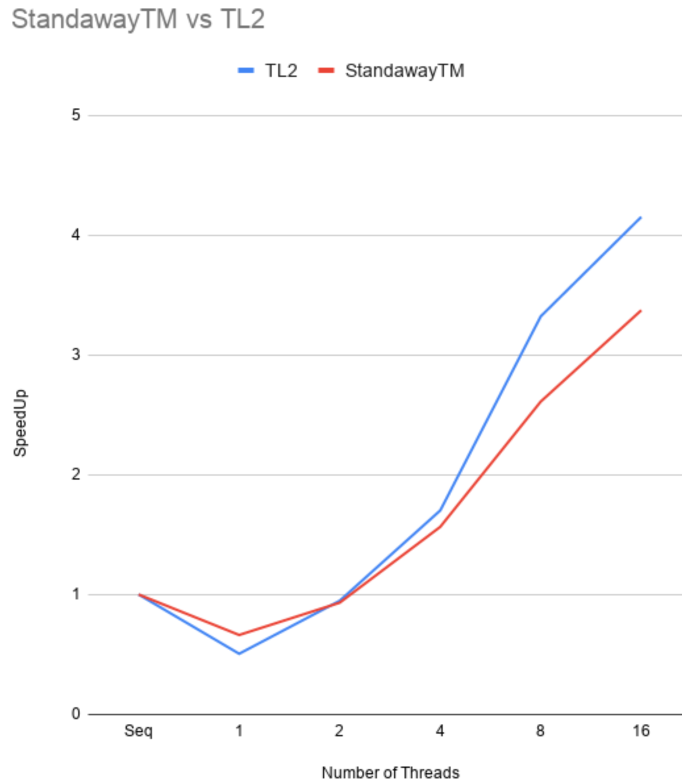


Σχήμα 5.6: Speed Improvement of StandawayTM compared to TL2 in ssca2

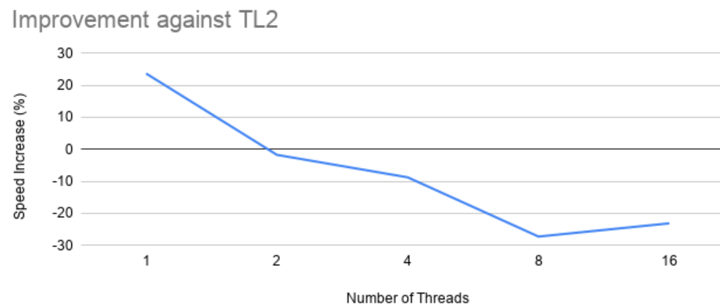
5.4 Vacation

Οι δημιουργοί του STAMP προτείνουν την εκτέλεσή του με συγκεκριμένες παραμέτρους για κάθε benchmark, αλλά στην περίπτωση του Vacation και του K-Means που θα δούμε παρακάτω προτείνονται δύο διαφορετικά σετ από παραμέτρους.

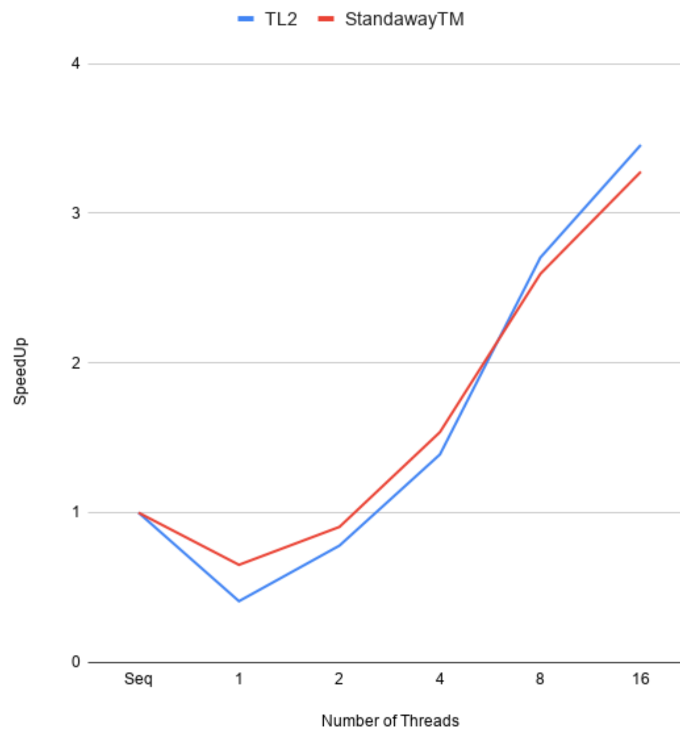
Και στα δύο βλέπουμε το δεύτερο μειονέκτημα του StandawayTM. Το StandawayTM είναι memory intensive και επιβαρύνει αρκετά το cache coherency πρωτόκολλο. Καθώς ο αριθμός των ατομικών εντολών αυξάνεται, ο επεξεργαστής πρέπει να φροντίσει ώστε όλα τα cores να βλέπουν την ίδια μνήμη, προκαλώντας κορεσμό στο δίκτυο των caches.



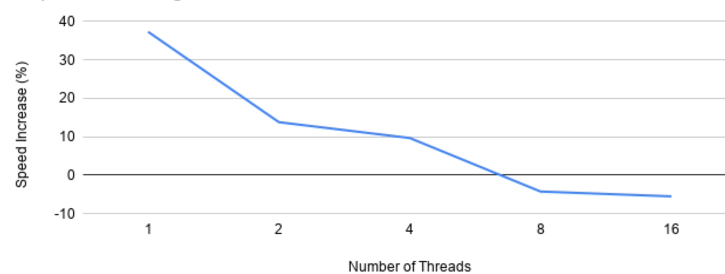
Σχήμα 5.7: Speedup of TL2 and StandawayTM in Vacation I



Σχήμα 5.8: Speed Improvement of StandawayTM compared to TL2 in Vacation I



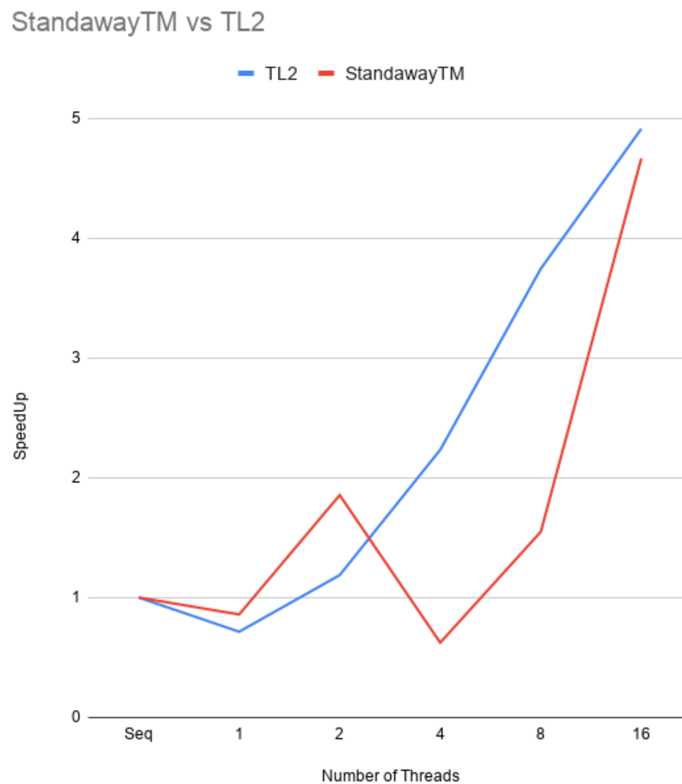
Σχήμα 5.9: Speedup of TL2 and StandawayTM in Vacation II



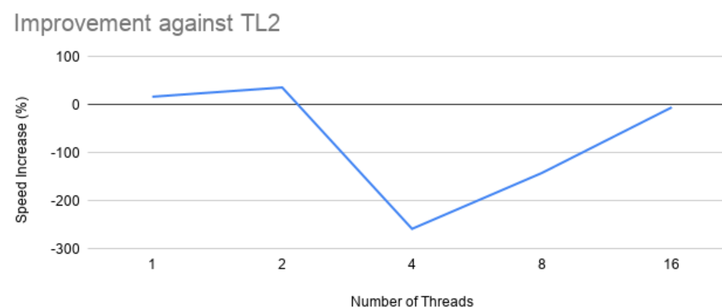
Σχήμα 5.10: Speed Improvement of StandawayTM compared to TL2 in Vacation II

5.5 K-Means

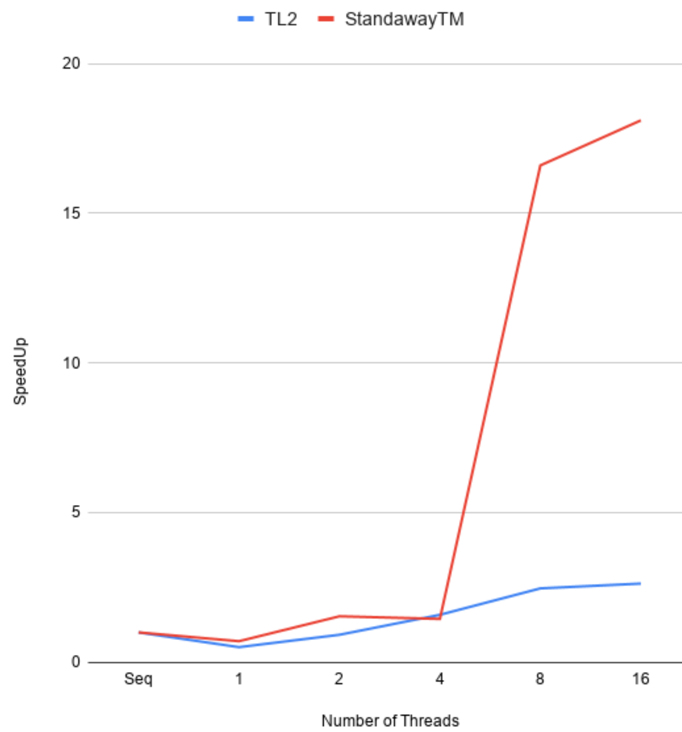
Στα K-Means βλέπουμε το τρίτο μειονέκτημα του StandawayTM. Το StandawayTM είναι ένα πολύ απλοϊκό TM. Καθώς πολλά Transactions συγκρούονται, το StandawayTM κατά κύριο λόγο τα αφήνει να ”τα βρούνε αναμεταξύ τους”. Αυτό σημαίνει ότι η απόδοση του StandawayTM είναι στενά συνδεδεμένη με το εκάστοτε πρόγραμμα αλλά και πολλές φορές διάφορους χρονισμούς και ”races”. Έτσι στο K-Means βλέπουμε ότι στα 4 (κυρίως) και 8 threads για λόγους που δύσκολα μπορούμε να ανακαλύψουμε, η απόδοση πέφτει κατακόρυφα ενώ μετά ανεβαίνει ξανά. Στην δεύτερη περίπτωση (σχήμα 5.13 και 5.14) στα 8 threads βλέπουμε speedup μεγαλύτερο του 8. Αυτό δεν πρέπει να μας προβληματίζει καθώς ο k-means δεν είναι ντετερμινιστικός αλγόριθμος και μπορεί να ολοκληρωθεί πρόωρα αν βρει το σωστό αποτέλεσμα.



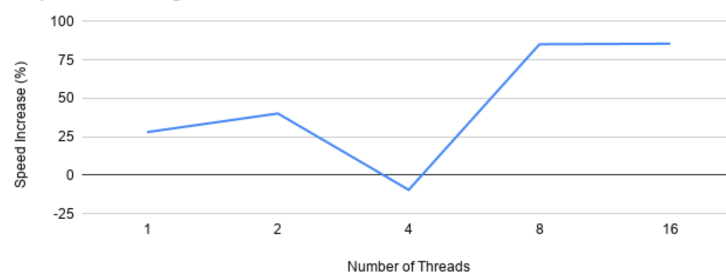
Σχήμα 5.11: Speedup of TL2 and StandawayTM in K-Means I



Σχήμα 5.12: Speed Improvement of StandawayTM compared to TL2 in K-Means I



Σχήμα 5.13: Speedup of TL2 and StandawayTM in K-Means II



Σχήμα 5.14: Speed Improvement of StandawayTM compared to TL2 in K-Means II

Κεφάλαιο 6

Σχετική Βιβλιογραφία

Τα STM είχαν μελετηθεί εκτενώς πριν από περίπου μία δεκαετία. Από τότε οι βελτιώσεις στις caches, η αύξηση των πυρήνων εντός ενός επεξεργαστή και οι βελτιώσεις στην ταχύτητα των ατομικών εντολών, έχουν ανοίξει νέα μονοπάτια και έχουν δημιουργήσει χώρο για νέα έρευνα η οποία δεν έχει ακόμα ολοκληρωθεί.

Η πρώτη αναφορά σε μηχανισμό που σχετίζεται με Transactional Memory κάνει ο David B. Lomet το 1977 [Lome77] όπου εμπνευσμένος από τις βάσεις δεδομένων προτείνει θεωρητικά την επέκταση των ατομικών εντολών σε πάνω από μία μεταβλητές. Για τα επόμενα χρόνια δεν υπήρξε κάποια νέα πρόταση μέχρι το 1993 όταν δύο νέα papers ανοίγουν τον δρόμο για τα Transactional Memory συστήματα. Ο Herlihy και ο Moss [Herl93] προτείνουν έναν μηχανισμό υλοποιημένο σε hardware ο οποίος επιτρέπει σε κάποιο thread να κλειδώνει θέσεις μνήμης και να αποτρέπει την πρόσβαση σε αυτές από τα άλλα threads μέχρι να ελευθερωθούν. Στο paper που παρουσιάζουν εισάγονται στην βιβλιογραφία οι βασικές έννοιες των Transaction και τυποποιείται η ορολογία που χρησιμοποιείται μέχρι σήμερα. Ο Stone και λοιποί, παρουσιάζουν την ίδια περίοδο ένα αντίστοιχο σύστημα βασισμένο σε Software το οποίο αποθηκεύει σε λίστες τις διευθύνσεις που θέλει να ”κρατήσει” (reserve) και κάνει τις εγγραφές αν έχει καταφέρει να κρατήσει όλες τις θέσεις που κάνει access. Το σύστημα αυτό, γίνεται η απαρχή των Software Transactional Memery συστημάτων. Το 2001 εισάγεται η έννοια της υποθετικής εκτέλεσης με την σύγχρονη μορφή της από τους Rajwar και Goodman [Rajw01], όπου διαφορετικά threads έχουν διαφορετική εικόνα της μνήμης.

Η πραγματική έκρηξη στα TM συστήματα, γίνεται όμως μερικά χρόνια μετά, μεταξύ του 2006 και 2009. Η ”επανάσταση” αυτή αρχίζει με την SUN, η οποία επενδύει στην τεχνολογία αυτή, ίσως και πρώιμα. Οι Dice και Shavit δημοσιεύουν μία από τις πιο εμπεριστατωμένες και ολοκληρωμένες μελέτες πάνω στα TMs [Dice07] και προτείνουν το δικό τους STM. Έναν χρόνο μετά, οι ίδιοι παρουσιάζουν ίσως το TL2 [Dice06] που έχει χρησιμοποιηεθεί ευρέως ως σύστημα αναφοράς και σύγκρισης για τα STM συστήματα. Ο Dice και ο Shavit αποδεικνύουν πως το Transaction μπορεί να γίνει ένα χρήσιμο εργαλείο, όχι μόνο ακαδημαϊκά αλλά και στην βιομηχανία. Στρέφοντας την προσοχή της ερευνητικής κοινότητας στο πεδίο αυτό, ακολουθεί μία περίοδος ωρίμανσης των TMs. Τα TMs αρχίζουν να εισάγονται ερευνητικά ως κομμάτια των ίδιων των γλωσσών προγραμματισμού με υποστήριξη από compilers και runtime συστήματα [AdlT06]. Ακολουθούν διάφορες προτάσεις για STMs τα οποία έχουν πολλές φορές καλύτερη απόδοση από το TL2, όπως το TinySTM[Rieg06] και το SwissTM [Drag09]. Το 2008 γράφεται και το βασικότερο benchmark πάνω στα TM συστήματα, το STAMP [Minh08]. Την ίδια περίοδο (2007), σχεδιάζεται και ο πρώτος επεξεργαστής που υποστηρίζει HTM, ο Rock (SPARC v9) από την Sun. Ο Rock παρουσίαζε αρκετά προβλήματα με την χρήση του HTM συστήματος, όπως αυξημένες θερμοκρασίες και χαμηλή απόδοση, και με την εξαγορά της Sun από την Oracle, το HTM σύστημα επέστρεψε σε ερευνητικό στάδιο. Το 2010 γράφεται η δεύτερη έκδοση του βιβλίου Transactional Memory (v2) [Harr10], το οποίο είναι και βασική πηγή έμπνευσης και πληροφορίας για την εργασία αυτή.

Το 2011 αρχίζει η δεύτερη επανάσταση των TMs με την IBM αυτήν την φορά στο προσκήνιο και δύο νέους επεξεργαστές, τον IBM BlueGene/Q (PowerPC A2) το 2011 και τον IBM zEnterprise EC12 το 2012. Τα επόμενα χρόνια οι γλώσσες προγραμματισμού αρχίζουν να προσθέτουν επίσημα τα Transactions, κυρίως ως επεκτάσεις. Ευρέως χρησιμοποιούμενοι compilers όπως ο gcc θα εισάγουν τα Transactions το 2014 παρότι η C δεν έχει οριστικοποιήσει ακόμα (2020) την επίσημη τυποποίηση των

Transactions. Διάφορα συστήματα STM υποστηρίζονται από C, C++, Java, Python, Haskel και άλλες γλώσσες. Το 2013 η Intel προσθέτει το δικό της HTM σύστημα, το TSX, στους Haswell επεξεργαστές της. Όμως ένα bug στο TSX θα καθυστερήσει την χρήση του μέχρι την επόμενη γενιά, την Broadwell. Από τότε τα HTMs αρχίζουν να μπαίνουν σε όλο και περισσότερες αρχιτεκτονικές.

Τα τελευταία χρόνια, η έρευνα έχει στραφεί σε εξειδίκευση των STMs σε διάφορους τομείς, όπως για συστήματα NUMA, για βάσεις δεδομένων, για ανάλυση γράφων ή ακόμα και για χρήση τους στα λειτουργικά συστήματα ή επικοινωνία με περιφερειακά. Δεν έχουμε βρει κάποια έρευνα τα τελευταία χρόνια πάνω σε STMs γενικής χρήσης όπως αυτό που προτείνουμε εμείς.

Κεφάλαιο 7

Συμπεράσματα και επόμενα βήματα

Στα πειραματικά μας αποτελέσματα αποδείξαμε ότι το StandawayTM έχει την δυνατότητα για βελτιωμένες επιδόσεις. Ο απλός σχεδιασμός και υλοποίησή του, μειώνουν αρκετά το overhead των Transactional accesses σε σύγκριση με τα προτεινόμενα TMs. Όμως τα φαινόμενα που διέπουν τον παράλληλο προγραμματισμό είναι αρκετά περίπλοκα και σε πολλές περιπτώσεις το StandawayTM δεν καταφέρνει να εκπληρώσει την υπόσχεσή του. Αυτό που αποδείξαμε με αυτήν την εργασία είναι ότι παρά το κόστος πάνω στο locality, την επιβάρυνση των caches, και των πολλών ατομικών εντολών, το StandawayTM μπορεί να είναι πιο γρήγορο από αντίστοιχα TMs που ξοδεύουν περισσότερο χρόνο σε τοπικά (thread local) δεδομένα.

Το StandawayTM βρίσκεται ακόμα στα πρώιμα στάδια της ζωής του. Πάνω στις βασικές αρχές του μπορούν να χτιστούν περαιτέρω σχήματα που φέρνουν την ωρίμανσή του, όπως είδαμε στο κεφάλαιο 3.7 και 3.8. Σίγουρα χρειάζεται πολύ μελέτη ακόμα έως ότου μπορέσει να είναι ένα ευρέως χρησιμοποιούμενο εργαλείο. Πρέπει σίγουρα να εξαλειφθούν οι ιδιαιτερότητες που είδαμε στο κεφάλαιο 5.5, και πρέπει να αυτοματοποιηθεί η διαδικασία απόδοσης μνήμης για τα records. Η ενσωμάτωση του σε compiler είναι μία καλή αρχή αλλά σίγουρα θέλει πολύ προσοχή και είναι ιδιαίτερα δύσκολη διαδικασία. Επίσης χρειάζεται η επίσημη (formal) απόδειξη της ορθότητάς του, πέρα από την διαισθητική και πειραματική. Παρακάτω παρουσιάζουμε μερικές ακόμα κατευθύνσεις που πιστεύουμε ότι θα οδηγήσουν στην ωρίμανσή του και στην πιθανή υιοθέτησή του από την ερευνητική κοινότητα αλλά και την βιομηχανία.

7.1 Memory Footprint

Το StandawayTM διπλασιάζει το footprint της μνήμης των Transactionaly accessed δεδομένων. Αυτό μπορεί να είναι αποδεκτό σε αρκετές περιπτώσεις, είτε λόγο μικρού όγκου τέτοιων δεδομένων, είτε λόγο αρκετής μνήμης. Υπάρχουν περιπτώσεις όμως, που τέτοια ελευθερία δεν είναι αποδεκτή και το memory footprint του StandawayTM πρέπει να περιοριστεί.

Αυτό μπορεί να γίνει αρχικά, αυξάνοντας το μέγεθος του word, αλλάζοντας ουσιαστικά το granularity. Προφανώς αυτή η αλλαγή θα αυξήσει τα false conflicts όμως ανάλογα το κατά πόσο τα δεδομένα γίνονται accessed σε clusters ή όχι, αυτή η επιλογή μπορεί να μειώσει το footprint χωρίς να πέσει η απόδοση. Υπάρχουν μάλιστα περιπτώσεις που η αύξηση του μεγέθους του word θα αυξήσει την απόδοση καθώς το πρόγραμμα θα έχει περισσότερη διαθέσιμη cache. Το κατάλληλο μέγεθος του word για μέγιστη απόδοση εξαρτάται από το εκάστοτε πρόγραμμα και τον αριθμό των threads. Αξίζει να σημειώσουμε ότι το μέγεθος του word δεν είναι απαραίτητο να παραμένει ίδιο καθ' όλη την διάρκεια του προγράμματος και κάθε access μπορεί να αφορά διαφορετικό μέγεθος word αρκεί όλα τα accesses να χτυπάνε το ίδιο record. Έτσι ολόκληρα structs μπορούν να έχουν ένα record που ορίζει την πρόσβασή τους ενώ άλλα structs μπορούν να έχουν ξεχωριστά records για κάθε πεδίο τους.

Επιπλέον και τα ίδια τα records μπορούν να γίνουν πιο μικρά. Δυστυχώς το μέγιστο μέγεθός τους περιορίζεται από το μέγιστο μέγεθος των παραμέτρων των ατομικών εντολών, όμως δεν υπάρχει αντίστοιχο κάτω όριο. Για εφαρμογές που τρέχουν με δύο threads αρκούν 4 bits για κάθε read και write record. Στην γενική περίπτωση για κάθε thread αρκούν 2 bits αλλά καθώς η πρόσβαση του επεξεργαστή στην μνήμη γίνεται με βάση το byte, καλό θα είναι ο αριθμός των bits να είναι δυνάμεις

του 2 ώστε μην γίνει πολύ περίπλοκη η πρόσβαση σε αυτά. Αυτή η βελτιστοποίηση είναι κάθετη με την αλλαγή του μεγέθους του word και μπορεί να συνυπάρξει μαζί της. Έτσι για ένα πρόγραμμα που τρέχει σε 4 threads και granularity τα 32 bytes (τέσσερα longs σε 64-bit μηχάνημα), το memory footprint αυξάνεται κατά 3.125% εξαιτίας του StandawayTM (Με όλα τα accesses να είναι Transactional).

Ένα πρόγραμμα δεν κάνει access όλα τα δεδομένα σε κάθε σημείο του. Καθώς το πρόγραμμα περνάει διάφορα στάδια και κάνει access διαφορετικές δομές, τα αχρησιμοποίητα records μπορούν να "ανακυκλωθούν". Αυτή η διαδικασία είναι ίσως αρκετά δύσκολο να αυτοματοποιηθεί από έναν compiler, όμως με την βοήθεια του προγραμματιστή μπορούμε να ορίσουμε περιοχές του προγράμματος στις οποίες δομές παύουν να γίνονται accessed Transactionally και τα records είναι ελεύθερα για να χρησιμοποιηθούν από άλλες δομές.

7.2 Αρχικοποίηση των records

Το StandawayTM αρχικοποιεί όλα τα records στο 0. Αυτή μπορεί να είναι μία αργή διαδικασία όμως πολλές φορές το λειτουργικό μηδενίζει τις σελίδες της μνήμης που προέρχονται από άλλες διεργασίες για λόγους ασφάλειας. Αυτή η διαδικασία γίνεται "on demand" και πολλές φορές η κλήση σε calloc μπορεί να είναι πιο αργή από κλήση σε malloc. Τα linux δεν μηδενίζουν όλες τις σελίδες για λόγους που φεύγουν πέρα από τα πλαίσια αυτής της διπλωματικής. Όμως πρέπει να παρατηρήσουμε το εξής. Οι σελίδες από άλλα processes είναι μηδενισμένες ενώ οι σελίδες που είχαν γίνει free από το ίδιο process περιέχουν "σκουπίδια". Όμως πριν ελευθερώσουμε ένα struct τα records που περιέχει έχουν μηδενιστεί καθώς κανένα άλλο thread δεν θα γραφεί ή θα διαβάσει από μία δομή πριν ελευθερωθεί. Έτσι μπορούμε να ανακυκλώσουμε δομές χωρίς να αρχικοποιήσουμε τα records αρκεί να γίνει με πολύ προσοχή.

7.3 Strong atomicity και Hardware

Τα STM "υποφέρουν" από ένα πρόβλημα, δεν μπορούν να αντιληφθούν conflicts από non-Transactional accesses. Παρότι ο προγραμματισμός όπου γίνονται ταυτόχρονα Transactional και non-Transactional accesses δεν φαίνεται να έχει κάποια ιδιαίτερη χρηστικότητα, και οφείλεται μάλλον σε λάθος του προγραμματιστή, το hardware μπορεί εύκολα να εντοπίζει τέτοια σφάλματα και να φροντίζει για την αποφυγή του εφόσον τα records είναι σε συγκεκριμένη απόσταση από τα words και το hardware ξέρει σε ποια memory pages γίνονται Transactions.

Σε επόμενο βήμα, παρατηρούμε ότι το ίδιο το StandawayTM μπορεί να υλοποιηθεί πολύ εύκολα σε hardware εκμεταλλευόμενο τις ίδιες τις caches. Αν και η υλοποίηση αυτή μπορεί να είναι πολύ ακριβή και να μην αξίζει η βελτίωση της απόδοσης.

7.4 Τοποθέτηση των records

Στην προσαρμογή του STAMP ώστε να είναι συμβατό με το StandawayTM , βρεθήκαμε μπροστά σε ένα δίλημμα το οποίο αποφασίσαμε να αγνοήσουμε. Η τοποθέτηση των records μπορεί να γίνει είτε αμέσως δίπλα στα words που αφορούν είτε σε κάποια άλλη θέση της μνήμης. Εάν τοποθετήσουμε το word και το record στο ίδιο cache line, τότε η πρόσβαση στο word μας φέρνει το cache line στην cache μας και επιτρέπει το γρήγορο access του record. Όμως σε κάθε access είτε read είτε write, θα γράψουμε στο record, και κατά συνέπεια θα κάνουμε invalidate τα αντίγραφα word στις άλλες caches ακόμα και αν δεν γράφουμε σε αυτά. Αν από την άλλη μεταφέρουμε τα records σε άλλα cache lines, τότε από την μία αυτά τα cache lines θα έχουν περισσότερη κίνηση και από την άλλη το word και το record θα είναι σε διαφορετικές cache lines και θα έχουμε μεγαλύτερο latency. Μία λογική βελτίωση θα ήταν να διαβάζουμε το record πριν το αλλάξουμε και αν υπάρχει ήδη η Ταυτότητά μας, ή πρόκειται να κάνουμε abort δεν χρειάζεται να την προσθέσουμε. Από την άλλη αυτή η βελτιστοποίηση κάνει το πρόγραμμα πιο αργό αν τα περισσότερα accesses γίνονται για πρώτη φορά στο εκάστοτε word και

πετυχαίνουν. Προφανώς όλα τα παραπάνω χρειάζονται αρκετή μελέτη και πειραματικές δοκιμές ώστε να αποφανθούμε για τα φαινόμενα που επικρατούν και την βέλτιστη τοποθέτηση των records.

7.5 Commit time false conflicts

Κατά την διάρκεια του commit, ένα Transaction πρέπει να αφαιρέσει τις Ταυτότητές του από τα records στα οποία έχει κάνει πρόσβαση. Σε αυτήν την διάρκεια, τα Transaction μπορούν να κάνουν conflict με το committing Transaction ενώ κάτι τέτοιο δεν χρειάζεται να συμβεί. Αυτό μπορεί να αποφευχθεί αν ο έλεγχος τον records αγνοεί τα committing Transactions με την χρήση ενός global record που περιέχει τις Ταυτότητές τους. Δυστυχώς, αν και η υλοποίησή του είναι σχετικά εύκολη, δεν είχαμε στον κατάλληλο χρόνο για τον έλεγχό της και την μελέτη των επιπτώσεών της στην επίδοση του StandawayTM.

7.6 Becoming rude

Σε περίπτωση πολλών abort, ένα thread θα προσπαθήσει να λάβει το global lock και να τρέξει ”αποφασιστικά” περιμένοντας όπου χρειαστεί τα άλλα threads να ελευθερώσουν την θέση που το ενδιαφέρει. Το thread που τρέχει ”αποφασιστικά” δεν κλειδώνει την θέση αυτή μέχρι να είναι σίγουρο ότι είναι ελεύθερη. Αυτό επιτρέπει σε άλλα threads να τρέχουν πάνω στην θέση αυτή για περισσότερο χρόνο αλλά εν τέλει το ”αποφασιστικό” thread θα πάρει την θέση. Μπορούμε να κάνουμε το εν λόγω thread, πιο ”αγενές” κλειδώνοντας την θέση που το ενδιαφέρει πριν ελευθερωθεί από τα υπόλοιπα threads. Από την άλλη, μπορούμε να κάνουμε το thread πιο ”ευγενικό” παίρνοντας το global lock μόνο για μία θέση μνήμης ώστε να ξεπεράσουμε ένα σημείο που προκαλεί livelock, αλλά αν υπάρχουν περισσότερα από ένα, η μοίρα μας είναι αβέβαιη. Προφανώς ανάλογα το πρόγραμμα υπάρχει βέλτιστη επιλογή αλλά αυτό μας οδηγεί σε profiling και αρκετή επιπλέον δουλειά που επίσης βγαίνει εκτός του πλαισίου αυτής της διπλωματικής.

7.7 Επίλογος

Παρότι ακόμα σε πρώιμα στάδια, το StandawayTM μπορεί να γίνει ένα ευρέως διαδεδομένο εργαλείο. Χρειάζεται ακόμα αρκετή μελέτη στα φαινόμενα που επηρεάζουν την απόδοσή του και αρκετός πειραματισμός για εύρεση των κατάλληλων παραμέτρων, όμως φαίνεται να μπορεί να ανταγωνιστεί και να ξεπεράσει τα σύγχρονα TMs.

Βιβλιογραφία

[Abba10] Gulfam Abbas and Naveed Asif, *Performance Tradeoffs in Software Transactional Memory*, Ph.D. thesis, Blekinge Institute of Technology, 2010.

[AdlT06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha and Tatiana Shpeisman, "Compiler and runtime support for efficient software transactional memory", in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 26–37, 2006.

[Bloo70] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, vol. 13, p. 422–426, July 1970.

[Cacs08] Călin Caşcaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras and Siddhartha Chatterjee, "Software Transactional Memory: Why Is It Only a Research Toy?", *ACM QUEUE*, vol. 6, pp. 46–58, September 2008.

[Dice06] Dave Dice, Ori Shalev and Nir Shavit, "Transactional Locking II", in *DISC 2006: Distributed Computing*, pp. 194–208, 2006.

[Dice07] Dave Dice and Nir Shavit, "Understanding Tradeoffs in Software Transactional Memory", in *International Symposium on Code Generation and Optimization*, 2007.

[Drag09] Aleksandar Dragojević, Rachid Guerraoui and Michał Kapałka, "Stretching transactional memory", in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 155–165, 2009.

[Harr10] Tim Harris, James Larus and Ravi Rajwar, *Transactional Memory, 2nd edition*, Morgan & Claypool Publishers, December 2010.

[Herl93] Maurice Herlihy and J. Eliot B. Moss, "Transactional memory: architectural support for lock-free data structures", in *ISCA*, p. 289–300, May 1993.

[Lome77] David B. Lomet, "Process structuring, synchronization, and recovery using atomic actions", in *ACM Conference on Language Design for Reliable Software*, 1977.

[Minh08] Chì Cao Minh, JaeWoong Chung, Christos Kozyrakis and Kunle Olukotun, "STAMP: Stanford Transactional Applications for MultiProcessing", in *2008 IEEE International Symposium on Workload Characterization*, September 2008.

[Rajw01] Ravi Rajwar and James R. Goodman, "Speculative lock elision: enabling highly concurrent multithreaded execution", in *34th International Symposium on Microarchitecture*, pp. 294–305, 2001.

[Rieg06] Torvald Riegel, Pascal Felber and Christof Fetzer, "A lazy snapshot algorithm with eager validation", in *20th International Symposium on Distributed Computing*, pp. 284–298, 2006.

[Ston93] Janice M. Stone, Harold S. Stone, Phil Heidelberger and John Turek, "Multiple reservations and the Oklahoma update", in *IEEE Parallel and Distributed Technology*, vol. 1, p. 58–71, November 1993.