



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΎΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Optimization methodology for dynamic applications utilizing tree data structures in embedded systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΟΥ

Θωμά Ν. Παπαστεργίου

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Μάρτιος 2016



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΎΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Optimization methodology for dynamic applications utilizing tree data structures in embedded systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΟΥ

Θωμά Ν. Παπαστεργίου

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την Μαρτίου 2016.

.....

Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής

.....

Κιαμάλ Πεκμεσζή
Καθηγητής

.....

Γεώργιος Γκούμας
Λέκτορας

Αθήνα, Μάρτιος 2016

.....

Θωμάς Ν. Παπαστεργίου

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

Copyright © Θωμάς Ν. Παπαστεργίου, 2016

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.

Table of Contents

Σύντομη Περίληψη.....	vii
Abstract.....	viii
Acknowledgments.....	ix
Αναλυτική Περίληψη.....	x
List of figures and tables.....	xxvi
1 The Dynamic Data Type Refinement (DDTR) Methodology.....	28
1.1 Introduction.....	28
1.2 The DDTR Methodology.....	30
1.3 Limitations of the original implementation and the new DDT library.....	31
1.4 DDT Exploration Stage.....	35
1.5 Pareto Optimal Exploration Stage.....	36
1.6 Experimental Results.....	36
2 The impact of cache in modern embedded systems.....	40
2.1 Why cache is used in modern embedded systems.....	41
2.2 Designing cache-friendly data structures.....	42
2.2.1 Clustering Technique.....	42
2.2.2 Coloring Technique.....	43
2.2.3 Compression Technique.....	44
2.3 Effect of the cache memory on the DDTR Methodology.....	45
3 Methodology for creating cache-friendly tree data structures.....	48
3.1 The HAT-trie data structure.....	49
3.2 The Ternary Tree.....	51
3.3 Patricia trie.....	52
3.4 Cache-Friendly tree implementations.....	54
3.4.1 Underlying theory.....	54
3.4.2 Challenges and solutions.....	58
4 Experimental setup and results.....	63
4.1 Platforms and benchmarks.....	63
4.2 Experimental results – synthetic datasets.....	64

4.3	Experimental results – real world datasets	68
4.3.1	IP dataset	68
4.3.2	Dictionary datasets	69
4.4	Overhead of the cache-friendly implementations	71
5	Conclusion and future work	75
5.1	Summary	75
5.2	Future work	75
References	77

Σύντομη Περίληψη

Με την πάροδο των χρόνων, ολοένα και περισσότερες εφαρμογές που εκτελούνταν αποκλειστικά σε υπολογιστικά συστήματα υψηλών προδιαγραφών (HPC) υλοποιούνται για ενσωματωμένα συστήματα. Τα ενσωματωμένα συστήματα, χάρη στην ραγδαία πρόοδο της τεχνολογίας, είναι πλέον σε θέση να εκτελούν πολυσύνθετες και απαιτητικές εφαρμογές που τις περισσότερες φορές βασίζονται σε μεγάλες δυναμικές δομές δεδομένων για να επιτελέσουν την λειτουργία για την οποία έχουν σχεδιαστεί. Ο σχεδιασμός των σημαντικών δομών δεδομένων των εφαρμογών, σε ένα μεγάλο βαθμό, καθορίζει και την απόδοση, καθώς και τις απαιτήσεις ολοκλήρωσης της εφαρμογής. Η διαδικασία επιλογής της σωστής δομής δεδομένων δεν είναι ούτε εύκολη, ούτε προφανής. Ο σχεδιαστής της εφαρμογής καλείται να λάβει υπόψη πολλές παραμέτρους που εξαρτώνται απ' την συσκευή που θα κληθεί να εκτελέσει την εφαρμογή. Ωστόσο, κάθε συσκευή συνήθως έχει τις δικές της απαιτήσεις. Μια μεθοδολογία ονόματι Dynamic Data Type Refinement (DDTR) αναπτύχθηκε ώστε να βοηθήσει τον σχεδιαστή να αξιολογήσει διαφορετικούς συνδυασμούς δομών δεδομένων με έναν τρόπο αποτελεσματικό και όσο πιο αυτοματοποιημένο γίνεται. Προσφέρει βελτιστοποιήσεις, κυρίως στον τομέα των λιστών και των πινάκων, που βασίζονται στα χαρακτηριστικά της εφαρμογής και τον τρόπο με τον οποίο αυτή προσπελάζει τα δεδομένα. Σε αυτήν την εργασία, διάφορες πτυχές της παραπάνω μεθοδολογίας επεκτείνονται: κατ' αρχάς ενσωματώνουμε δένδρικές δομές δεδομένων ώστε να εμπλουτίσουμε την ήδη υπάρχουσα συλλογή και να καταστήσουμε την μεθοδολογία κατάλληλη για ένα μεγαλύτερο εύρος σύγχρονων εφαρμογών. Έπειτα, παραθέτουμε κάποιες υλοποιήσεις δομών δεδομένων που λαμβάνουν υπόψη χαρακτηριστικά της συσκευής ώστε να επιτυγχάνουν καλύτερη απόδοση. Η επεκταμένη μεθοδολογία αξιολογείται μέσω διαφόρων κατασκευασμένων και πραγματικών αρχείων εισόδου που εκτελούνται στην πλατφόρμα Myriad και Freescale, όπου επιτυγχάνουμε βελτιώσεις της τάξης του 30%. Επιπροσθέτως, Pareto βέλτιστες υλοποιήσεις δομών δεδομένων που δεν ήταν διαθέσιμες με την προηγούμενη μεθοδολογία, είναι πλέον εφικτό να ανιχνευθούν.

Λέξεις Κλειδιά: μεθοδολογία DDTR, δένδρα, βελτιστοποίηση δυναμικών δομών δεδομένων, υλοποιήσεις που λαμβάνουν υπόψη την κρυφή μνήμη, ενσωματωμένα συστήματα

Abstract

Applications that were previously executed in High Performance Computers (HPC) systems are increasingly implemented in embedded devices. Modern embedded systems are now capable of executing complex and demanding applications that are usually based on large dynamic data structures. The design of the critical data structures of the applications, in a large extent, determines the performance and the memory requirements of the whole system. The work of selecting the correct data structure for an application is not an easy or obvious one. Depending on the platform of interest, different requirements may need to be satisfied. The Dynamic Data Structure Refinement methodology was originally developed to help the designer evaluate different data structure selections in an effective and automatic manner. It provides optimizations, mainly in list and array data structures, which are based on the application's features and access patterns. In this work, various aspects of the methodology are extended: first, we integrate radix tree optimizations to enrich the existing collection and make the methodology compatible with a larger group of modern applications. Then, we provide a set of platform-aware data structure implementations, for performing optimizations based on the hardware features. The extended methodology is evaluated using a wide set of synthetic and real-world benchmarks on the Myriad and Freescale platforms, in which we achieved a performance and memory trade-offs up to 30%. Additionally, Pareto optimal data structure implementations that were not available by the previous methodology, are now identified with the extended one.

Keywords: Dynamic Data Type Refinement (DDTR) methodology, radix tree, dynamic data structure optimization, cache-friendly implementations, embedded systems

Acknowledgments

The work described in this thesis was carried out at the Microprocessors Laboratory and Digital Systems Lab of the School of Electrical and Computer Engineering of the National Technical University of Athens. I would like to thank my supervisors, Prof. Dimitrios Soudris and Dr. Lazaros Papadopoulos for their trust they showed me. I am extremely grateful to both of them for their guidance, support and invaluable research experience, which were of paramount importance to the completion of this work.

Finally, I would also like to thank my closest friends and my family for the love and support they have showed me throughout the duration of this thesis and my studies in general.

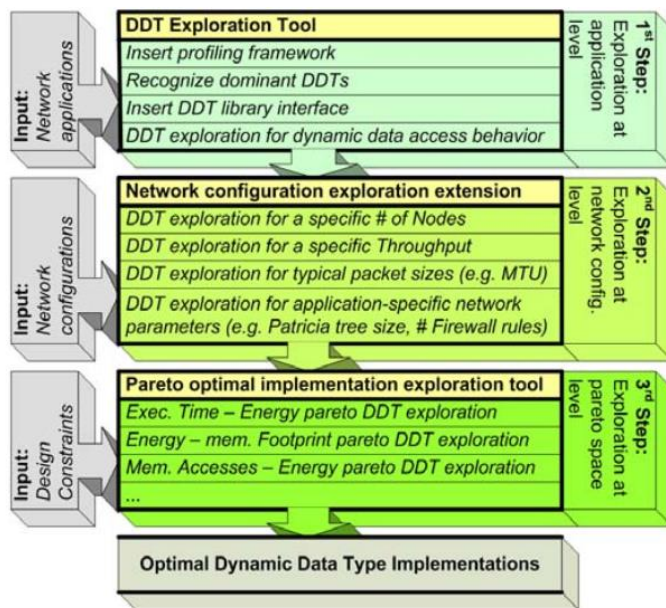
Εισαγωγή στην DDTR Μεθοδολογία

Καθώς περνάνε τα χρόνια, οι σύγχρονες εφαρμογές δικτύων και πολυμέσων γίνονται ολοένα πιο πολύπλοκες και απαιτητικές και χρειάζονται περισσότερη απόδοση, φαινόμενο που οδηγεί σε μεγαλύτερη κατανάλωση μνήμης και κατ' επέκταση, ενέργειας. Επίσης, οι εφαρμογές αυτές συνήθως χαρακτηρίζονται από μεγάλο βαθμό αλληλεπίδρασης με τον χρήστη. Επομένως, η συμπεριφορά τους καθίσταται δυναμική και καθορίζεται από εξωγενείς παράγοντες (όπως το πόσο «φορτωμένο» είναι ένα δίκτυο ή τις κινήσεις του παίκτη σε ένα παιχνίδι).

Ως αποτέλεσμα, η χρήση δυναμικών δομών δεδομένων (ΔΔ) είναι επιβεβλημένη, ώστε να μπορεί το πρόγραμμα να αντεπεξέλθει σε ένα μεγάλο εύρος διαφορετικών συνθηκών εκτέλεσης, κάτι που δεν θα ήταν σε θέση να πραγματοποιήσει αν χρησιμοποιούσε σταθερό αριθμό πόρων. Φυσικά, ένα δυναμικό πρόγραμμα είναι πολύ πιο δύσκολο να σχεδιαστεί και να βελτιστοποιηθεί από ένα στατικό, αφού στην δεύτερη περίπτωση όλοι οι παράμετροι λειτουργίας είναι καθορισμένοι και οι πόροι της εφαρμογής είναι ακριβώς όσοι χρειάζονται. Αντιθέτως, στην περίπτωση των δυναμικών προγραμμάτων, κάποια άστοχη εκτίμηση του προγραμματιστή στην επιλογή των δυναμικών ΔΔ είναι εύκολο να οδηγήσει σε σπατάλη πόρων ή χειρότερη απόδοση από την αναμενόμενη και αυτό μπορεί να έχει σημαντικές παρενέργειες (μικρή διάρκεια μπαταρίας, μη ικανοποίηση του χρήστη και άλλα).

Η επιλογή της κατάλληλης ΔΔ δεν είναι εύκολη υπόθεση, ακριβώς επειδή διαφορετικές συσκευές μπορεί να έχουν διαφορετικές ανάγκες, ακόμα και για το ίδιο πρόγραμμα. Για παράδειγμα, ένας υπολογιστής συχνά έχει άφθονη μνήμη, σε αντίθεση με ένα κινητό τηλέφωνο. Ένας προγραμματιστής επομένως θα πρέπει χειροκίνητα να αλλάζει το πρόγραμμα του για να το προσαρμόσει σε διαφορετικές πλατφόρμες, μια διαδικασία επίπονη και ευαίσθητη σε σφάλματα. Για τον λόγο αυτό αναπτύχθηκε μια μεθοδολογία που ονομάζεται Dynamic Data Type Refinement Methodology, ή εν συντομία, DDTR. Η μεθοδολογία αυτή αποτελείται από τρία διακριτά βήματα και αρχικά είχε σχεδιαστεί για τις ανάγκες δικτυακών εφαρμογών. Τα βήματα αυτά είναι:

- Στο πρώτο στάδιο, το πρόγραμμα εκτελείται για κάποια συγκεκριμένα δεδομένα εισόδου (benchmarks). Κατά την εκτέλεση, μελετάται η συμπεριφορά των δυναμικών ΔΔ, που έχουν σημανθεί και αντικατασταθεί με τις αντίστοιχες δομές που υπάρχουν σε μια βιβλιοθήκη. Για κάθε ΔΔ μπορεί να υπάρχουν πολλές υλοποιήσεις στην βιβλιοθήκη, επομένως δοκιμάζονται όλοι οι δυνατοί συνδυασμοί και για κάθε συνδυασμό τα αποτελέσματα (όπως κατανάλωση ενέργειας, μνήμης, ταχύτητα εκτέλεσης) αποθηκεύονται.
- Στο δεύτερο στάδιο, επειδή ακριβώς η μεθοδολογία αυτή είχε αρχικά αναπτυχθεί για δικτυακές εφαρμογές, εισάγονται οι παράμετροι του δικτύου. Αυτές έχουν να



Εικόνα 1: Βήματα της DDTR μεθοδολογίας

κάνουν με τον αριθμό των κόμβων του δικτύου και το μέγεθος των πακέτων και μπορούν να επηρεάσουν σημαντικά τα αποτελέσματα της αναζήτησης.

- Στο τρίτο και τελευταίο στάδιο, τα αποτελέσματα της αναζήτησης παρουσιάζονται μέσω ενός γραφικού περιβάλλοντος (Graphical User Interface - GUI) στον σχεδιαστή, μαζί με τις Pareto – βέλτιστες επιλογές. Από εκεί μπορεί να επιλέξει τον συνδυασμό με την λιγότερη ενέργεια για παράδειγμα.

Τις πιο πολλές φορές δεν υπάρχει λύση που να είναι βέλτιστη σε κάθε τομέα. Για παράδειγμα, η λύση με την λιγότερη ενέργεια δεν είναι αναγκαίο να έχει και την καλύτερη ταχύτητα εκτέλεσης. Με το παραπάνω εργαλείο μειώνεται ο χρόνος που χρειάζεται να αφιερώσει ο σχεδιαστής, αφού μεγάλο κομμάτι της δουλείας αναλαμβάνεται από το εργαλείο και επίσης πετυχαίνεται και σημαντικό κέρδος (σε ορισμένες δικτυακές εφαρμογές περίπου 80% λιγότερη ενέργεια και 20% καλύτερη ταχύτητα) ακριβώς επειδή ο προγραμματιστής μπορεί να μην έχει λάβει υπόψη όλες τις παραμέτρους του προγράμματος.

Ωστόσο υπάρχουν και συγκεκριμένοι περιορισμοί στο παραπάνω εργαλείο. Αυτοί έχουν αφενός να κάνουν με το γεγονός ότι υποστηρίζεται σχετικά περιορισμένος αριθμός δυναμικών ΔΔ που σαφώς περιορίζει και το πλήθος των εφαρμογών που μπορεί να υποστηρίξει (πχ με πίνακες κατακερματισμού και δένδρα). Αφετέρου, η βιβλιοθήκη που χρησιμοποιείται (Matisse profiling tool) δεν χρησιμοποιεί τα πλεονεκτήματα των αντικειμενοστραφών γλωσσών προγραμματισμού πράγμα που κάνει ιδιαίτερα δύσκολη την επέκτασή της. Τέλος, για μεγάλες εφαρμογές η όλη διαδικασία είναι από πολύ αργή έως μη εφαρμόσιμη, αφού ο χρήστης δεν μπορεί να επιλέξει μόνο συγκεκριμένους συνδυασμούς ΔΔ ώστε να μειώσει το χώρο αναζήτησης.

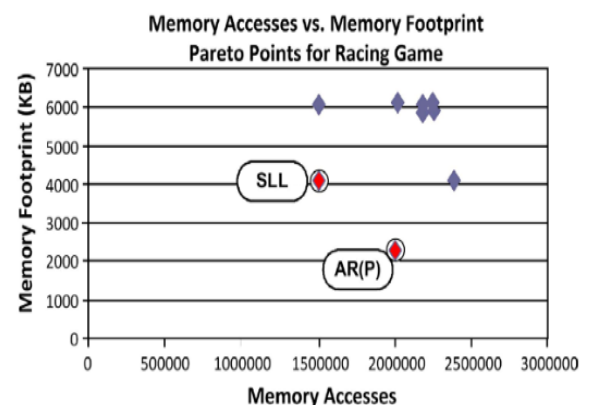
Για να αντιμετωπιστούν ορισμένα απ’ τα προβλήματα, αναπτύχθηκε μια βελτιωμένη DDTR μεθοδολογία, που χρησιμοποιούσε ένα νέο εργαλείο βιβλιοθήκης που εισάγει την έννοια της αφηρημένης ΔΔ. Η αφηρημένη ΔΔ είναι ένα επίπεδο αφαίρεσης ανάμεσα στην εφαρμογή και τα δεδομένα και περιέχει τις μεθόδους που χρησιμοποιούνται για να προσπελαστούν τα δεδομένα. Αυτό κάνει την διαδικασία πιο εύκολη, αφού πλέον μπορεί να εφαρμοστεί για οποιαδήποτε εφαρμογή (και όχι μόνο για δικτυακές) που συμμορφώνεται με τη διεπαφή της βιβλιοθήκης. Για κάθε ΔΔ υπάρχουν οι βασικές λειτουργίες όπως εισαγωγή, διαγραφή, προσπέλαση και τροποποίηση. Αυτή η αφαίρεση καθιστά εύκολη την εναλλαγή

μεταξύ διαφορετικών ΔΔ χωρίς την ανάγκη να αλλάξει ο κώδικας της εφαρμογής και επίσης ευνοεί την ύπαρξη πολλών υλοποιήσεων για μια ΔΔ. Οι δομές που υποστηρίζει η νέα βιβλιοθήκη είναι:

- *Στοίβα*: ένας σωρός αντικειμένων που προσθέτουμε και αφαιρούμε απ' την κορυφή.
- *Ουρά*: συλλογή αντικειμένων που προστίθενται απ' το ένα άκρο και αφαιρούνται απ' το άλλο.
- *Ουρά με δύο άκρα (deque)*: συλλογή αντικειμένων που προστίθενται και αφαιρούνται από οποιαδήποτε άκρο.
- *Μη ταξινομημένη λίστα*.
- *Ταξινομημένη λίστα*: λίστα με τα στοιχεία ταξινομημένα με βάση κάποιο κριτήριο.
- *Πίνακας κατακερματισμού*: χρησιμοποιεί συναρτήσεις κατακερματισμού για να εντοπίσει ένα στοιχείο σε σταθερό χρόνο.
- *Σύνολο (σετ)*: μη ταξινομημένη συλλογή αντικειμένων χωρίς επαναλήψεις.
- *Πολυσύνολο (multiset)*: σύνολο όπου επιτρέπονται οι επαναλήψεις
- *Δένδρο*: μη γραμμικές δομές που χρησιμοποιούνται για να δηλώσουν ιεραρχία. Κάθε κόμβος στο δένδρο έχει παιδιά και έναν πατέρα (εκτός απ' τον πρώτο).

Από τις παραπάνω ΔΔ το προηγούμενο εργαλείο υποστήριζε μόνο μη ταξινομημένες λίστες. Κάθε δομή έχει τις ανάλογες αφηρημένες λειτουργίες (πχ η στοίβα έχει push/pop) και αυτές υλοποιούνται με διαφορετικό τρόπο ανάλογα την περίπτωση. Μια στοίβα μπορεί να υλοποιείται μέσω μιας συνδεδεμένης λίστας είτε με πίνακα για παράδειγμα, αλλά η λειτουργία της σε αφηρημένο επίπεδο δεν αλλάζει. Πέρα από τη βιβλιοθήκη, αναπτύχθηκε και μια διεπαφή που επιτρέπει στον χρήστη να επιλέξει τους συνδυασμούς των υλοποιήσεων που τον ενδιαφέρουν ώστε να επιτύχει ταχύτερη εξερεύνηση.

Τα βήματα παραμένουν παρόμοια. Αρχικά οι ΔΔ ανιχνεύονται είτε χειροκίνητα είτε αυτόματα (εάν το πρόγραμμα τηρεί τη διεπαφή της βιβλιοθήκης). Έπειτα απομονώνονται αυτές που παίζουν σημαντικό ρόλο για την εκτέλεση του προγράμματος και γίνεται η εξερεύνηση με τους διάφορους συνδυασμούς και αποθηκεύονται τα αποτελέσματα. Τέλος, οι Pareto – βέλτιστοι συνδυασμοί επιλέγονται και παρουσιάζονται στον σχεδιαστή (Σχήμα 2). Η μεγαλύτερη ποικιλία δομών μπορεί να οδηγήσει σε νέους συνδυασμούς που δεν φαινόταν με το παλαιότερο εργαλείο. Το στάδιο με τις παραμέτρους του δικτύου λείπει, αφού πλέον αναφερόμαστε σε γενικού σκοπού εφαρμογές. Αναλόγως την εφαρμογή, μπορούμε να πετύχουμε μεγάλες βελτιώσεις.



Σχήμα 2: Pareto-βέλτιστοι συνδυασμοί για ένα παιχνίδι

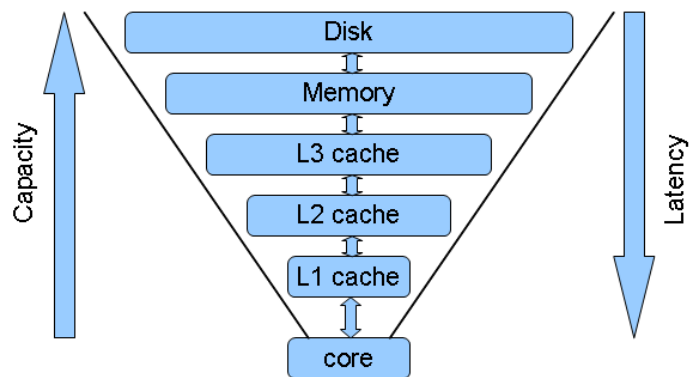
Κρυφή μνήμη και τρόποι αξιοποίησης της

Πέρα από τις ίδιες τις εφαρμογές, το υλικό των ενσωματωμένων συστημάτων στα οποία αυτές εκτελούνται γίνονται όλο και πιο πολύπλοκα. Πλέον ένα μεγάλο ποσοστό διαθέτει σύστημα μνήμης με πολλά επίπεδα και κρυφή (ή λανθάνουσα) μνήμη (KM - cache memory), όπως ένας προσωπικός υπολογιστής (Σχήμα 3). Αυτό εισάγει νέες παραμέτρους που η DDTR μεθοδολογία δεν λάμβανε υπόψη.

Η KM αποτελεί ένα (ή περισσότερα επίπεδα μικρής και γρήγορης μνήμης) και παρεμβάλλεται μεταξύ του επεξεργαστή και της κύριας μνήμης. Χάρη στο μικρό μέγεθος και την απόσταση από τον επεξεργαστή, προσφέρει ταχύτερη ανάγνωση/εγγραφή από την κύρια μνήμη (έως και εκατοντάδες κύκλους) και δύναται να επιταχύνει σημαντικά όσες εφαρμογές κάνουν σωστή αξιοποίηση της. Το πρόβλημα είναι πως ο προγραμματιστής δεν έχει συνήθως άμεση πρόσβαση στην KM και δεν μπορεί να καθορίσει ρητά πια στοιχεία θα μπουν στην KM. Όταν ένα κομμάτι μνήμης προσπελάζεται, αυτομάτως έρχεται στην KM. Επομένως για να εκμεταλλευτούμε αυτό το γεγονός, οι εφαρμογές καλό είναι να παρουσιάζουν μια εύκολα προβλέψιμη συμπεριφορά όσον αφορά τις προσπελάσεις στη μνήμη ή να επαναχρησιμοποιούν τα ίδια δεδομένα. Με μια τέτοια συμπεριφορά είναι εύκολο το μηχάνημα να προβλέψει τί θα χρησιμοποιηθεί στη συνέχεια ώστε να το φέρει έγκαιρα στη KM.

Οι δομές δεδομένων που χρησιμοποιούν δείκτες τείνουν να έχουν γενικά κακή συμπεριφορά στην KM. Αυτό συμβαίνει γιατί τα επιμέρους κομμάτια μνήμης που συνδέονται με δείκτες βρίσκονται διάσπαρτα στο χώρο διευθύνσεων. Άρα φέρνοντας ένα απ' τα κομμάτια στην KM είναι δύσκολο να φιλοξενεί πολλούς κόμβους της δομής. Οι πρώτες γλώσσες (όπως οι αρχικές εκδόσεις της Fortran) δεν υποστήριζαν δείκτες επομένως αυτό το πρόβλημα ήταν λιγότερο αισθητό. Καθώς όμως οι δυναμικές δομές δεδομένων κυριαρχούν, όλο και περισσότερες γλώσσες τους ενσωμάτωσαν (πχ C, Pascal). Υπάρχουν διάφορες τεχνικές για να χρησιμοποιήσουμε αποτελεσματικά την KM σε εφαρμογές με δείκτες και αναλύονται παρακάτω:

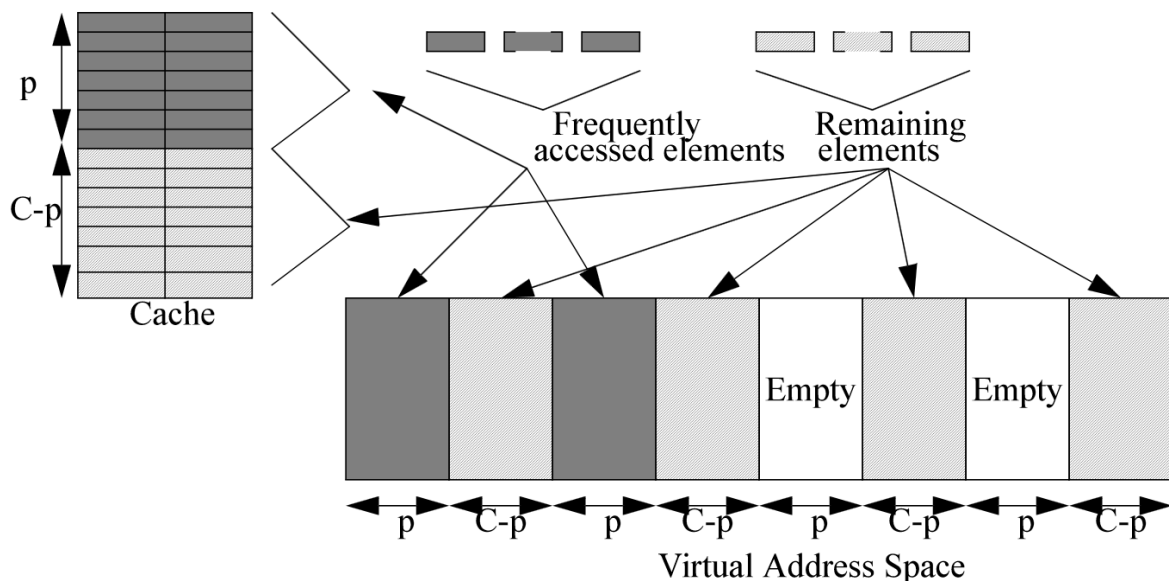
- **Ομαδοποίηση:** Το βασικό σκεπτικό εδώ είναι να ομαδοποιήσουμε πολλά στοιχεία μιας δομής δεδομένων που αναμένεται να προσπελαστούν σειριακά ή σε κοντινές χρονικές στιγμές. Για παράδειγμα, σε μια συνδεδεμένη λίστα αν ένας κόμβος προσπελαστεί υπάρχει μεγάλη πιθανότητα να προσπελαστεί και ο γειτονικός του. Όταν ένα κομμάτι μνήμης έρχεται στην KM έρχεται συνήθως σε κομμάτια των 64



Σχήμα 3: Ιεραρχία μνήμης σε ένα σύγχρονο επεξεργαστή

bytes ή παραπάνω, ενώ ένας κόμβος μπορεί να είναι πολύ μικρότερος. Επομένως σε ένα τέτοιο κομμάτι μπορούν να χωράνε πολλοί κόμβοι και να γλιτώσουμε πολλές άστοχες προσπελάσεις στη ΚΜ (και κατ' επέκταση χρόνο και ενέργεια). Για να συμβεί αυτό πρέπει να δεσμεύουμε μνήμη για πολλά στοιχεία τη φορά (ένα πίνακα στοιχείων για παράδειγμα) αφού η μνήμη που δεσμεύεται με τη malloc είναι συνεχόμενη. Αν χρησιμοποιούμε διαφορετική κλήση malloc για κάθε στοιχείο δεν υπάρχει εγγύηση ότι γειτονικά στοιχεία θα βρεθούν σε γειτονικές θέσεις μνήμης. Ακόμα και αν ένας κόμβος είναι μεγαλύτερος από ένα μπλοκ μνήμης που μεταφέρεται στην ΚΜ και πάλι αξίζει γειτονικά να στοιχεία να είναι σε συνεχόμενες θέσεις μνήμης, επειδή διευκολύνεται η πρόβλεψη που κάνει αυτόματα το σύστημα.

- **Χρωματισμός:** Με αυτήν την τεχνική προσπαθούμε να διαχωρίσουμε τα δεδομένα που ξέρουμε ότι είναι συχνά χρησιμοποιούμενα – και επομένως θέλουμε να υπάρχουν συνέχεια στην ΚΜ – από αυτά που χρησιμοποιούνται περιστασιακά και τυγχάνει να εκτοπίζουν εκείνα που θέλουμε να κρατήσουμε. Πρακτικά χωρίζουμε την ΚΜ σε δυο περιοχές και αναλόγως σε ποια περιοχή θέλουμε να μπει κάθε στοιχείο, φροντίζουμε κατά την δέσμευση του να τοποθετείται στις σωστές διευθύνσεις μνήμης (που εν τέλει θα καταλήξουν στην αντίστοιχη περιοχή της ΚΜ) αφήνοντας ενδεχομένως κενά στη μνήμη όπως φαίνεται στο Σχήμα 4. Πρόκειται για έναν «εικονικό» διαχωρισμό της ΚΜ μέσα από τον κώδικα, χωρίς να επηρεάζουμε

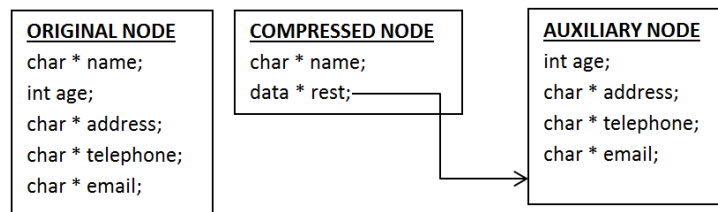


Σχήμα 4: Τεχνική χρωματισμού για διαχωρισμό των δεδομένων

κάπως το σύστημα. Με αυτόν τον τρόπο ελαχιστοποιούμε τον ανταγωνισμό ανάμεσα στα συχνά και στα σπανίως χρησιμοποιούμενα τμήματα της μνήμης.

- **Συμπύεση:** Η συμπύεση είναι μια αρκετά εύκολη τεχνική που παρ' όλα αυτά μπορεί να αποβεί πολύ αποδοτική. Πολλές φορές ένα στοιχείο μιας ΔΔ περιέχει πολλά πεδία και από αυτά μόνο ένα υποσύνολο εξετάζεται, κατά την αναζήτηση ενός στοιχείου ή στη διάρκεια κάποιας άλλης συχνής λειτουργίας. Για παράδειγμα, ένα

στοιχείο μπορεί να αντιπροσωπεύει έναν πελάτη και να έχει ως επιπρόσθετες πληροφορίες το όνομα, την ηλικία, την διεύθυνση και το τηλέφωνο, αλλά ο πελάτης να αναζητείται πάντα με βάση το όνομα. Σε αυτή την περίπτωση μπορούμε να απομονώσουμε τα στοιχεία τα οποία δεν μας ενδιαφέρουν άμεσα (πχ την ηλικία) κατά την αναζήτηση και να τα βάλουμε σε μια δευτερεύουσα δομή που συνδέεται με έναν δείκτη όπως στο Σχήμα 5. Έτσι «κρύβουμε» τον χώρο που χρειάζονται πίσω από το χώρο που απαιτείται για την προσθήκη ενός επιπλέον δείκτη. Καθώς κάθε κόμβος τώρα είναι μικρότερος, μπορούμε να πακετάρουμε περισσότερους κόμβους στον ίδιο χώρο και να πετύχουμε ακόμα καλύτερη απόδοση με τις προαναφερθείσες τεχνικές, αφού με την τεχνική της ομαδοποίησης στο ίδιο μπλοκ της ΚΜ θα έχουμε περισσότερους κόμβους και άρα όταν ένα μπλοκ έρχεται στην ΚΜ θα έρχονται πολλά στοιχεία μαζί του.



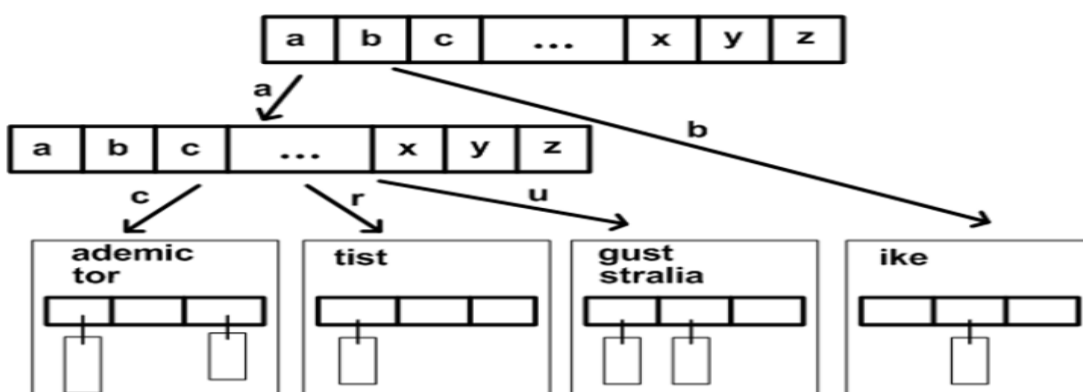
Σχήμα 5: Παράδειγμα συμπίεσης για έναν κόμβο που περιέχει τα στοιχεία ενός πελάτη

Προσθήκες στην DDTR Μεθοδολογία

Η DDTR μεθοδολογία δεν λαμβάνει, μέχρι τώρα, υπόψη το σύστημα στο οποίο εκτελείται η εφαρμογή και τα χαρακτηριστικά του, πράγμα που μπορεί να αλλάξει ριζικά τη συμπεριφορά της εφαρμογής ανάλογα με την επιλογή των δομών. Εώς τώρα σαν βάση θεωρούνταν ένα σύστημα με κύρια μνήμη μόνο, πράγμα που σχεδόν ποτέ δεν συμβαίνει. Αυτό μπορεί να αλλοιώσει τη σημασία κάποιων στατιστικών δεδομένων. Για παράδειγμα, ένα πρόγραμμα με 1000 προσπελάσεις δεδομένων στην κύρια μνήμη θεωρείται λιγότερο αποδοτικό απ' ό,τι ένα πρόγραμμα με 10000 προσπελάσεις στην κρυφή μνήμη (KM) αφού αυτές κοστίζουν πολύ λιγότερο σε κύκλους λειτουργίας. Αντί για προσπελάσεις δεδομένων, είναι καλύτερο να μετράμε τις «πραγματικές» προσπελάσεις στη μνήμη.

Το πρώτο βήμα είναι να εμπλουτίσουμε την DDTR μεθοδολογία και με άλλες ΔΔ, όπως τα δένδρα. Τα δένδρα προτιμούνται σε πλήθος εφαρμογών, όπως για την αναπαράσταση λεξικών, την επεξεργασία κειμένου, τη συμπίεση και αλλού, κυρίως χάρη στην καλή απόδοση όσον αφορά τη μέση περίπτωση. Υπάρχουν διάφοροι τύποι δένδρων με διαφορετικά χαρακτηριστικά. Στην παρούσα εργασία ασχολούμαστε με τρία είδη και πιο συγκεκριμένα:

HAT-trie. Αυτή η δομή δεδομένων προέκυψε ως μια προσπάθεια να ελαττωθούν οι σημαντικές απαιτήσεις σε μνήμη που έχουν οι περισσότεροι τύποι δένδρων, μέσω της μείωσης του μεγέθους τους. Βασίζεται στην ιδέα του burst-trie που μπορεί να μειώσει τον αριθμό των κόμβων ενός δένδρου έως και 80% με πολύ μικρό αρνητικό αντίκτυπο στην ταχύτητά του. Η βασική φιλοσοφία είναι πως στα φύλλα του δένδρου υπάρχουν κάποιοι κουβάδες (buckets) που περιέχουν συμβολοσειρές με κοινό πρόθεμα. Αν ένας κουβάς γίνει



Σχήμα 6: Δομή HAT-trie με τους κόμβους και τα buckets που είναι πίνακες κατακερματισμού

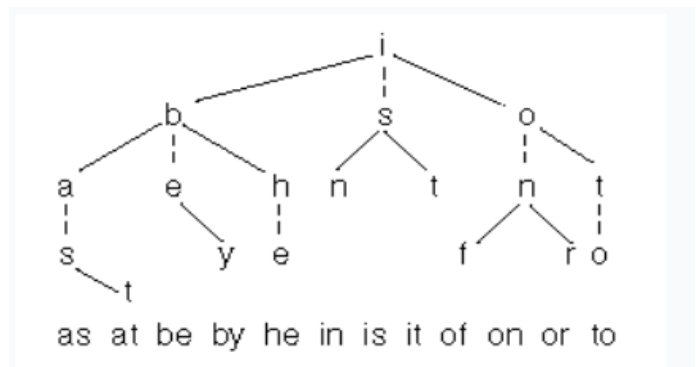
πολύ μεγάλος τότε «σπάει» και χωρίζεται σε μικρότερους κουβάδες που έχουν σαν γονείς νέους κόμβους του δένδρου. Κάθε κουβάς εσωτερικά υλοποιείται ως μια συνδεδεμένη λίστα με στοιχεία τις συμβολοσειρές.

Αυτό καθιστά το burst-trie μη αποδοτικό αφού η συνδεδεμένη λίστα δεν έχει καλή συμπεριφορά στην KM. Το HAT-trie είναι μια πιο KM-φιλική εκδοχή που μειώνει το

κόστος αναζήτησης σε ένα κουβά με το να οργανώνει τις επιμέρους συμβολοσειρές ως έναν πίνακα κατακερματισμού. Η αναζήτηση είναι πολύ πιο γρήγορη, αφού σε κάθε θέση του πίνακα οι συμβολοσειρές αποθηκεύονται συνεχόμενα και άρα μπορεί να αυξηθεί το μέγεθος του κουβά και να μειωθούν ακόμα παραπάνω οι κόμβοι του δένδρου.

Τριαδικό δένδρο. Το τριαδικό δένδρο είναι μια ΔΔ παρόμοια με το δυαδικό δένδρο. Η κύρια διαφορά είναι πως κάθε κόμβος έχει το πολύ τρία παιδιά αντί για δύο. Κάθε κόμβος επομένως φιλοξενεί έναν χαρακτήρα, τρεις δείκτες που ακολουθούνται αν ο τρέχων χαρακτήρας είναι μικρότερος, μεγαλύτερος ή ίσος με τον χαρακτήρα στον κόμβο, και μια σημείωση για το εάν ο κόμβος αυτός αποτελεί το τέλος μιας αποθηκευμένης συμβολοσειράς.

Τα κύρια πλεονεκτήματα του τριαδικού δένδρου είναι η απλότητα του και η καλή μέση περίπτωση λειτουργίας – κόστος $O(\log n)$. Επίσης χρησιμοποιεί γενικά λιγότερο χώρο για την αποθήκευση των συμβολοσειρών, αφού όσες συμβολοσειρές έχουν ίδια προθέματα μπορούν να χρησιμοποιούν κοινούς κόμβους. Στο δυαδικό δένδρο, κάθε κόμβος έχει αποθηκευμένη ολόκληρη την συμβολοσειρά, πράγμα που αυξάνει τις απαιτήσεις σε μνήμη. Η δομή του τριαδικού δένδρου το καθιστά κατάλληλο για εφαρμογές αυτόματης διόρθωσης/συμπλήρωσης και ελέγχου ορθογραφίας που δεν είναι εύκολο να πραγματοποιηθούν με άλλες δομές.



Σχήμα 7: Παράδειγμα τριαδικού δένδρου για συγκεκριμένες λέξεις

Patricia trie. Το Patricia trie σε κάθε κόμβο έχει αποθηκευμένη ολόκληρη την συμβολοσειρά μαζί με έναν ακέραιο που σημαδεύει ένα συγκεκριμένο bit της συμβολοσειράς. Όταν θέλουμε να ελέγξουμε αν μια συμβολοσειρά είναι αποθηκευμένη στον κόμβο πρώτα ελέγχουμε μόνο το συγκεκριμένο bit. Αν είναι το ίδιο τότε ελέγχουμε την υπόλοιπη συμβολοσειρά. Εάν δεν ταυτίζεται τότε ακολουθούμε το δεξί μονοπάτι αν το bit είναι 1 και το αριστερό αν είναι 0 και επαναλαμβάνουμε την διαδικασία στον επόμενο κόμβο. Σαν δομή είναι πιο πολύπλοκη αφού απαιτεί την ύπαρξη ενός σταθερού κόμβου – αναφοράς στην κορυφή του δένδρου και οι δείκτες μπορούν να δείχνουν σε κόμβους υψηλότερου επιπέδου. Το Patricia trie είναι κατάλληλο για εφαρμογές με IP διευθύνσεις ή πολύ μεγάλες συμβολοσειρές, αφού μπορεί να γλιτώσει πολλούς ελέγχους μιας και συνήθως μας ενδιαφέρουν μόνο συγκεκριμένα bits. Η εισαγωγή και η διαγραφή του κόμβου είναι πιο περίπλοκη ωστόσο.

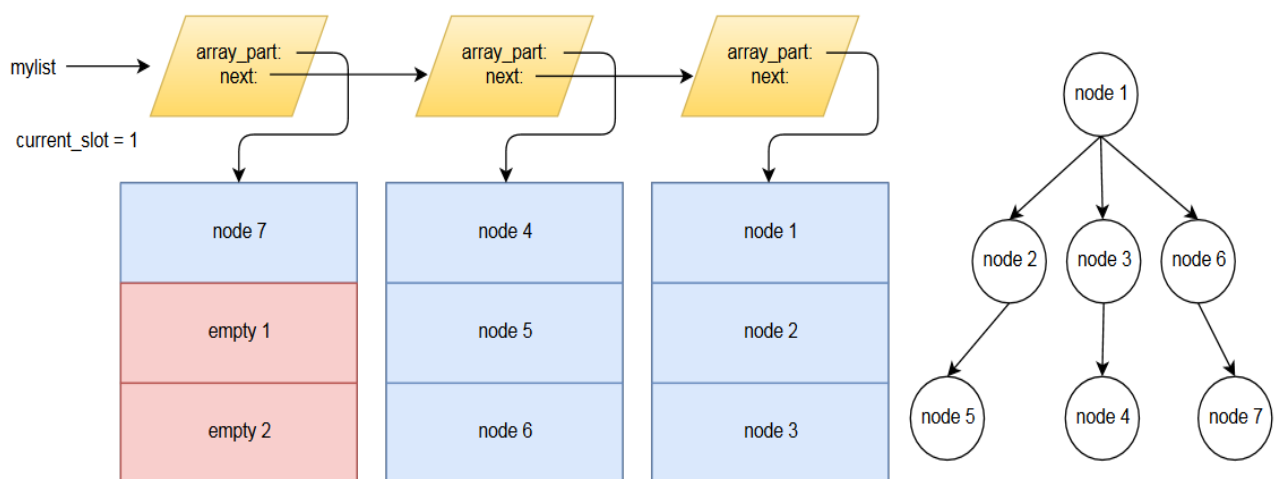
Γενικώς, τα δένδρα δεν έχουν πολύ καλή συμπεριφορά όσον αφορά την ΚΜ, όπως και οι πιο πολλές δομές με δείκτες εν γένει. Κάποιες τροποποιήσεις, όπως η εισαγωγή ενός πίνακα κατακερματισμού στην περίπτωση του HAT-trie, μπορούν να βελτιώσουν σημαντικά την απόδοση. Δημιουργήσαμε εκδοχές του τριαδικού δένδρου και του Patricia

τις με καλύτερη συμπεριφορά εφαρμόζοντας μια κοινή μεθοδολογία βελτιστοποίησης και στη συνέχεια αξιολογήσαμε την απόδοσή τους συγκριτικά με τις παλαιότερες δομές.

Η βασική ιδέα είναι να έχουμε συσχετισμένους κόμβους σε γειτονικές θέσεις μνήμης, ιδανικά στο ίδιο μπλοκ μνήμης που μεταφέρεται στην ΚΜ. Έτσι δεν θα χρειάζεται να έρχεται καινούργιο μπλοκ για κάθε κόμβο, αφού πλέον πολλοί κόμβοι θα είναι «πακεταρισμένοι» στο ίδιο μπλοκ. Οι λιγότερες προσβάσεις στην κύρια μνήμη έχουν σαν αποτέλεσμα την ταχύτερη εκτέλεση του προγράμματος και λιγότερες απαιτήσεις σε ενέργεια.

Το κύριο πρόβλημα που πρέπει να αντιμετωπιστεί είναι το γεγονός πως η δέσμευση κάθε κόμβου του δένδρου γίνεται ξεχωριστά (με εντολή στο λειτουργικό σύστημα) και στη συνέχεια αυτά τα διάσπαρτα κομμάτια μνήμης συνδέονται με δείκτες. Οι ασυνέχειες της μνήμης της δομής δημιουργούν πρόβλημα στα συστήματα πρόβλεψης που έχουν όλα τα σύγχρονα συστήματα για να προβλέπουν το μοτίβο των προσπελάσεων στη μνήμη ώστε να φέρνουν τα κρίσιμα κομμάτια μνήμης στην ΚΜ εγκαίρως.

Για να επιλύσουμε το πρόβλημα πρέπει να έχουμε μια συλλογή από κόμβους σε ένα πίνακα (που έχουν δεσμευτεί σε συνεχόμενες θέσεις μνήμης) και να τους διαμοιράζουμε αναλόγως τις ανάγκες της εφαρμογής. Έτσι, με μια κλήση συστήματος (malloc) θα έχουμε πολλούς κόμβους διαθέσιμους. Πλέον η δέσμευση ενός κόμβου του δένδρου θα ικανοποιείται από μια δική μας συνάρτηση που θα κοιτάει αν υπάρχει διαθέσιμος κόμβος προς διάθεση από αυτούς που έχουμε ήδη δεσμεύσει και αν όχι, θα αναλαμβάνει να καλέσει την malloc και να δημιουργήσει μια καινούργια ομάδα κόμβων.



Σχήμα 8: Κόμβοι δένδρου και ο τρόπος με τον οποίον έχουν δεσμευθεί

Οι διευθύνσεις μνήμης των επιμέρους πινάκων των κόμβων αποθηκεύονται σε μια δευτερεύουσα δομή δεδομένων που έχει την μορφή της συνδεδεμένης λίστας ώστε να έχουμε πρόσβαση στα κομμάτια της μνήμης που έχουμε δεσμεύσει για όταν χρειαστεί να τα αποδεσμεύσουμε (Σχήμα 8). Το μέγεθος της λίστας μικραίνει καθώς το μέγεθος των πινάκων αυξάνεται και συνήθως είναι αμελητέο σε σύγκριση με το μέγεθος όλου του δένδρου. Το μέγεθος του πίνακα είναι στην ευχέρεια του σχεδιαστή να το καθορίσει. Εμείς

επιλέξαμε να έχουμε περίπου 50 κόμβους ανά πίνακα. Όσοι περισσότεροι οι κόμβοι τόσο λιγότερες οι κλήσεις συστήματος που χρειάζονται, αλλά αυξάνεται η πιθανότητα να έχουμε σπατάλη μνήμης.

Αυτό που θέλουμε στη συνέχεια είναι να έχουμε τους κόμβους-παιδιά σε γειτονικές θέσεις μνήμης με τον γονιό. Κατά την προσπέλαση ενός δυαδικού δένδρου για παράδειγμα, υπάρχει 50% πιθανότητα να επιλεγεί ένα απ' τα δύο παιδιά ενός κόμβους καθώς διασχίζεται ένα μονοπάτι. Αν και τα δυο παιδιά είναι σε διπλανές θέσεις μνήμης (ή στο ίδιο μπλοκ) μπορούμε επομένως να αποφύγουμε τουλάχιστον τις μισές αστοχίες στο επίπεδο της KM. Στην καλύτερη περίπτωση που έχουμε γονέα και παιδιά στο ίδιο μπλοκ μνήμης το κέρδος μπορεί να είναι πολύ μεγαλύτερο.

Για να υλοποιηθεί το παραπάνω ωστόσο δεν αρκεί να έχουμε μια ομάδα κόμβων που έχουμε δεσμεύσει ομαδικά. Πρέπει και οι γειτονικοί κόμβοι στο πίνακα να αντιστοιχούν σε γειτονικούς κόμβους στο δέντρο. Αυτό εισάγει πολλές προκλήσεις. Για παράδειγμα, δεν υπάρχει απλός και αποδοτικός τρόπος να εξασφαλίσουμε ότι πατέρας και παιδί θα είναι σε γειτονικούς κόμβους. Καθώς διαμοιράζουμε τους κόμβους που έχουμε στην διάθεση μας, δεν ξέρουμε αν θα πρέπει να κρατήσουμε κάποιον γιατί κάποιος κόμβος μπορεί να έχει στο μέλλον παιδιά (ή πόσα θα είναι αυτά). Αν θέλαμε αυτό να το λάβουμε υπόψη, σίγουρα θα εισάγαμε μεγάλη πολυπλοκότητα που θα ελαχιστοποιούσε τα οφέλη της μεθόδου. Χρειάζεται μια γρήγορη απόφαση.

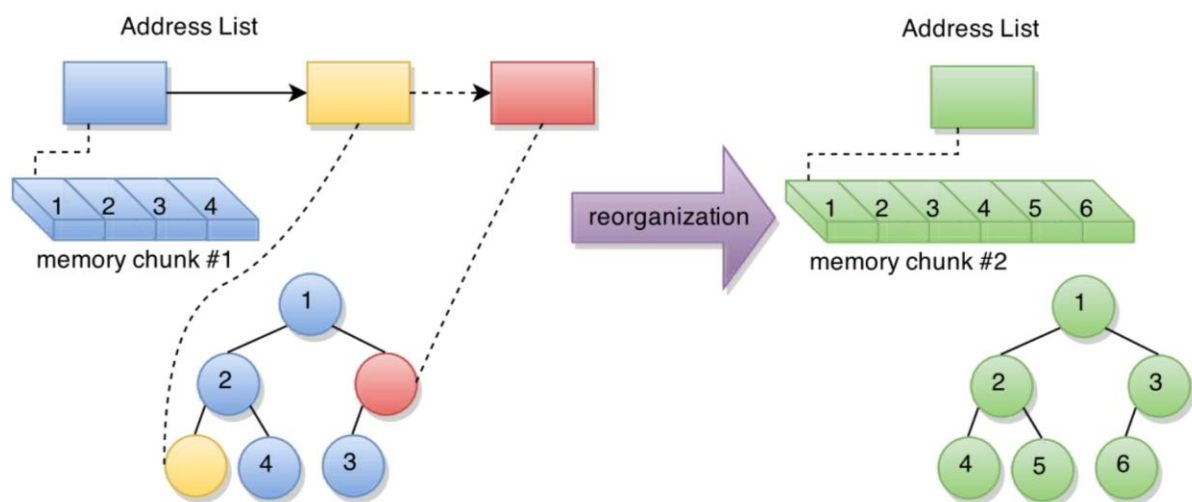
Επιπροσθέτως, το δένδρο είναι μια δυναμική δομή δεδομένων και η μορφολογία του αλλάζει διαρκώς κατά την εκτέλεση ενός προγράμματος. Ακόμα και αν είχαμε κάποιον τρόπο να εξασφαλίσουμε ότι κατά το διαμοιρασμό των κόμβων, παιδί και γονέας θα είναι σε γειτονικές θέσεις μνήμης, δεν έχουμε καμία εξασφάλιση ότι οι δύο αυτοί κόμβοι θα συνεχίσουν να έχουν την σχέση γονέα-παιδιού μέχρι το τέλος του προγράμματος. Μπορεί νέοι κόμβοι να εισαχθούν στην συνέχεια, ή κάποιιοι να διαγραφούν και νέες σχέσεις μεταξύ των κόμβων να δημιουργηθούν.

Τέλος, η διαγραφή των κόμβων εισάγει ένα ακόμα ζήτημα: πώς θα απελευθερώσουμε τη μνήμη αυτή. Καθώς οι κόμβοι έχουν δεσμευτεί ομαδικά πρέπει να αποδεσμευτούν και ομαδικά πράγμα που δεν είναι εφικτό όσο έστω και κάποιιοι είναι σε χρήση. Το να κρατάμε επιπλέον στοιχεία για κενούς κόμβους και να τους επαναχρησιμοποιούμε δεν είναι εφικτό (επιπλέον έλεγχοι και θέσεις μνήμης) ούτε συναφές με τη φιλοσοφία μας αφού δεν θέλουμε απλά να ξαναχρησιμοποιούμε τους κόμβους, αλλά να τους βάζουμε και στις σωστές θέσεις μνήμης.

Για να λύσουμε αυτά τα προβλήματα πρέπει αρχικά να συμβιβαστούμε με το γεγονός πως η αλληλουχία των κόμβων και οι σχέσεις μεταξύ τους θα αλλάζουν συνεχώς κατά την εκτέλεση του προγράμματος. Με αυτό υπόψη, μια απλή λύση είναι να γίνεται αναδιοργάνωση του δένδρου ανά τακτά διαστήματα ώστε αυτό να αποκτά την επιθυμητή δομή. Η αναδιοργάνωση δουλεύει ως εξής. Όταν ληφθεί η απόφαση για αναδιοργάνωση δεσμεύεται ένα συνεχόμενο κομμάτι μνήμης για όλους τους κόμβους. Στη συνέχεια το δένδρο διασχίζεται απ' την κορυφή και οι κόμβοι του δένδρου αντιστοιχίζονται σε κόμβους του μεγάλου συνεχόμενου πίνακα σύμφωνα με την επιθυμητή ιδιότητα (π.χ γονέας-παιδιά

σε συνεχόμενες θέσεις μνήμης). Η διαδικασία επαναλαμβάνεται αναδρομικά για όλους τους κόμβους μέχρι να καλυφθεί όλο το δένδρο. Στη συνέχεια αναβαθμίζουμε και τους δείκτες ώστε να δείχνουν στους κόμβους της νέας δομής και όχι της παλιάς. Τέλος μένει να απελευθερώσουμε τις θέσεις μνήμης των παλιών κόμβων – ένας απ’ τους λόγους που χρησιμοποιούμε μια λίστα για να τους κρατάμε – αφού δεν χρειάζεται να κρατάμε δυο αντίγραφα του δένδρου. Στο τέλος της αναδιοργάνωσης στη λίστα με τις χρησιμοποιούμενες θέσεις μνήμης απλά βάζουμε το νέο (μεγάλο) κομμάτι μνήμης που χρησιμοποιούμε.

Η αναδιοργάνωση εισάγει κάποια μειονεκτήματα όπως για παράδειγμα ότι στιγμιαία σταματάει η ροή του προγράμματος για να αντιγραφεί το δένδρο. Επίσης, η ίδια η αντιγραφή έχει κάποιο κόστος αφού χρειάζονται επιπλέον κύκλοι και υπολογιστική ισχύ για να πραγματοποιηθεί. Ακόμα, το αποτύπωμα στη μνήμη του υπολογιστή, έστω και προσωρινά – όσο διαρκεί η αναδιοργάνωση – αυξάνεται, αφού πρακτικά χρειάζεται διπλάσιος χώρος για τις δυο εκδόσεις του δένδρου. Για κάποιες εφαρμογές αυτό μπορεί να μην είναι πρόβλημα, ενώ για άλλες να είναι. Για να μειωθεί η επίπτωση στη μνήμη θα μπορούσε να αναζητηθεί μια σταδιακή αντιγραφή του δένδρου, ένα κομμάτι τη φορά, ενδεχομένως με κάποια επιπλέον επιβάρυνση στην πολυπλοκότητα, ωστόσο δεν έχουμε εστιάσει σε τέτοιες τεχνικές. Αυτό που έχει σημασία είναι το τελικό αποτέλεσμα να αντισταθμίσει τα μειονεκτήματα. Το δένδρο μετά την αναδιοργάνωση θα επιδεικνύει καλύτερη συμπεριφορά και ταχύτερη εκτέλεση των λειτουργιών του χάρη στη καλύτερη χρησιμοποίηση της ΚΜ. Επαφίεται στον προγραμματιστή να αξιολογήσει αν η εφαρμογή επιδέχεται μια επιβάρυνση (στιγμιαία) στη μνήμη για να έχει καλύτερη απόδοση.



Σχήμα 9: Πριν και μετά την αναδιοργάνωση του δένδρου

Με αυτά τα μειονεκτήματα υπόψη, το επόμενο στάδιο είναι να αποφασιστεί σε ποιες στιγμές θα γίνεται η αναδιοργάνωση. Γενικά, δεν είναι επιθυμητό να συμβαίνει πολύ συχνά γιατί το κόστος του να αναδιοργανώνεται το δένδρο θα υπερκαλύψει το κέρδος απ’ την πιο αποδοτική δομή. Η αναδιοργάνωση θα πρέπει να συμβαίνει εφόσον το δένδρο έχει υποστεί επαρκείς τροποποιήσεις από την τελευταία φορά που πραγματοποιήθηκε, ώστε να έχουμε

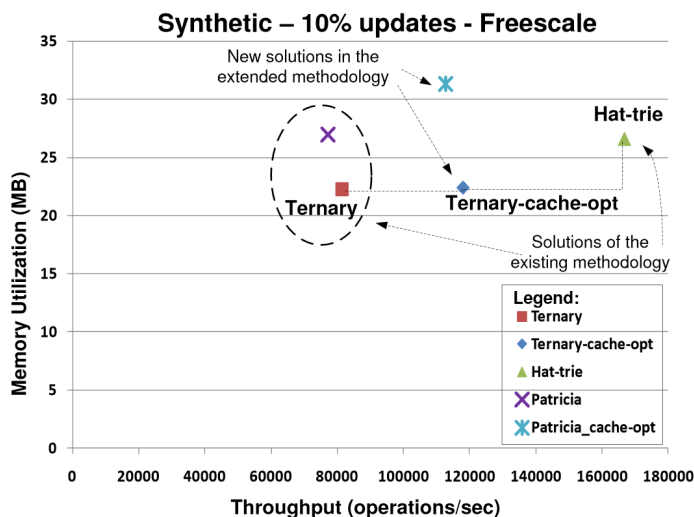
μεγάλη πιθανότητα να έχει απολέσει ένα μεγάλο κομμάτι την καλή του συμπεριφορά. Οι τροποποιήσεις μπορεί να είναι είτε εισαγωγές, είτε διαγραφές κόμβων. Ένας μικρός αριθμός τροποποιήσεων (για παράδειγμα μόνο 1000 σε ένα δένδρο εκατομμυρίων κόμβων) δεν πρόκειται να επηρεάσει ιδιαίτερα τη μέση περίπτωση. Στα πειράματα μας, το δένδρο αναδιοργανώνεται όταν οι αλλαγές που έχει υποστεί είναι ίσες με το μέγεθος που είχε κατά την τελευταία αναδιοργάνωση.

Πειραματικά αποτελέσματα

Μετά την εφαρμογή των βελτιστοποιήσεων που περιγράφηκαν προηγουμένως, χρειάζεται κάποιος πειραματισμός ώστε να αξιολογήσουμε τα αποτελέσματα τους στην πράξη και να επαληθεύσουμε τα θεωρητικά αποτελέσματα. Οι δομές δεδομένων που χρησιμοποιήσαμε στα πειράματα είναι το τριαδικό δένδρο, το Patricia trie και το HAT-trie. Ο στόχος είναι να μελετήσουμε συστήματα που έχουν διαφορετική ιεραρχία μνήμης, αφού κάποια συστήματα διαθέτουν KM ενώ άλλα όχι. Περνάμε επομένως από μια προσέγγιση ανεξάρτητη του συστήματος (παλιά DDTR) σε μια που λαμβάνει το σύστημα υπόψη.

Το υλικό που χρησιμοποιήθηκε είναι το ενσωματωμένο σύστημα Freescale i.MX6, με 4 πυρήνες που διαθέτει δύο επίπεδα KM και 1GB RAM, και η Myriad πλακέτα που υποστηρίζει 8 πυρήνες και 1MB κοινή κύρια μνήμη. Η Myriad πλακέτα δεν έχει σύστημα KM και επίσης έχει πολύ μικρότερη χωρητικότητα μνήμης.

Για την αξιολόγηση των αποτελεσμάτων χρησιμοποιήθηκαν ορισμένα κατασκευασμένα αρχεία ελέγχου αλλά και κάποια από πραγματικές εφαρμογές. Τα κατασκευασμένα (synthetic) αρχεία αποτελούνται από 10 εκατομμύρια πράξεις σε συμβολοσειρές-αριθμούς (εισαγωγή στοιχείου, διαγραφή ή εντοπισμός). Τα αρχεία πραγματικών εφαρμογών αποτελούνται από ένα σύνολο IP διευθύνσεων (κωδικοποιημένων



Εικόνα 10: Αποτελέσματα για ένα κατασκευασμένο αρχείο εισόδου που περιέχει μικρό αριθμό τροποποιήσεων στον Freescale

σαν αριθμοί για την προστασία την ανωνυμίας) που συνδέθηκαν στους servers για το Παγκόσμιο Κύπελλο 1998. Αποτελούνται από 3 εκατομμύρια διευθύνσεις από τις οποίες μόνο 4% είναι μοναδικές. Επιπροσθέτως, χρησιμοποιούνται και τα δεδομένα από ένα λεξικό. Στην περίπτωση της Myriad, λόγω του περιορισμένου χώρου μνήμης πρέπει να πάρουμε ένα υποσύνολο αυτών των δεδομένων αφού χρειάζεται να έχουμε μικρότερα δένδρα. Οι μετρικές που καταγράφουμε είναι ο αριθμός των εντολών ανά δευτερόλεπτο και το αποτύπωμα του προγράμματος στη μνήμη. Παρακάτω αναφέρονται ενδεικτικά κάποια αποτελέσματα για κάθε σύστημα.

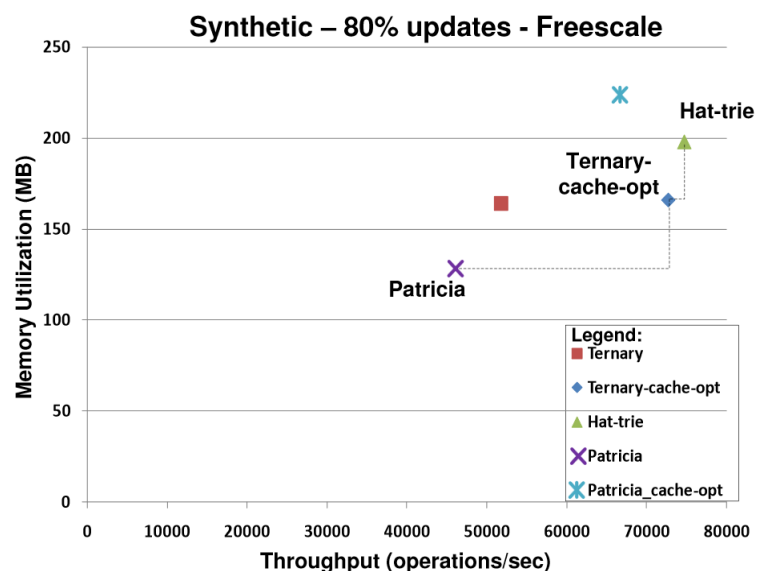
Στα κατασκευασμένα αρχεία με μικρό αριθμό τροποποιήσεων, παρατηρούμε ότι τα αποτελέσματα της βελτιστοποίησης είναι αισθητά (Εικόνα 10). Αυτό είναι αναμενόμενο, αφού το δένδρο χρειάζεται να αναδιοργανωθεί λιγότερες φορές και μεταξύ των

αναδιοργανώσεων έχουμε περισσότερες προσπελάσεις πράγμα που μεγιστοποιεί τα οφέλη της βελτιστοποίησης. Οι εντολές/ δευτερόλεπτο αυξάνονται έως και 30-35%. Η δομή HAT-trie έχει πολύ καλύτερη απόδοση πάντως, πράγμα αναμενόμενο αφού βασίζεται σε πίνακες κατακερματισμού για την γρήγορη αναζήτηση δεδομένων και στην συγκεκριμένη περίπτωση οι αναζητήσεις κυριαρχούν.

Όσον αφορά τη μνήμη του προγράμματος, θα περίμενε κανείς οι βελτιστοποιημένες εκδόσεις να έχουν διπλάσια μνήμη (έστω και προσωρινά) από τις απλές γιατί χρειάζεται να αντιγραφεί το δένδρο κατά την αναδιοργάνωση. Ωστόσο αυτό δεν συμβαίνει και έχει να κάνει με το πότε συνέβη η τελευταία αναδιοργάνωση. Αν έχει περάσει μεγάλο διάστημα χωρίς αναδιοργάνωση, τότε το δένδρο μεγαλώνει και το κόστος της «κρύβεται». Γι' αυτό οι βελτιστοποιημένες εκδόσεις (ιδιαίτερα στην περίπτωση του τριαδικού δένδρου) έχουν αντίστοιχες απαιτήσεις μνήμης. Σε απόλυτες τιμές το Patricia trie χρειάζεται παραπάνω μνήμη γιατί το τριαδικό δένδρο εκμεταλλεύεται κοινά προθέματα στις συμβολοσειρές ώστε να επαναχρησιμοποιεί κόμβους, σε αντίθεση με το Patricia trie που πρέπει να αποθηκεύσει όλη τη συμβολοσειρά. Κάτι αντί-στοιχο συμβαίνει και στην περίπτωση του HAT-trie.

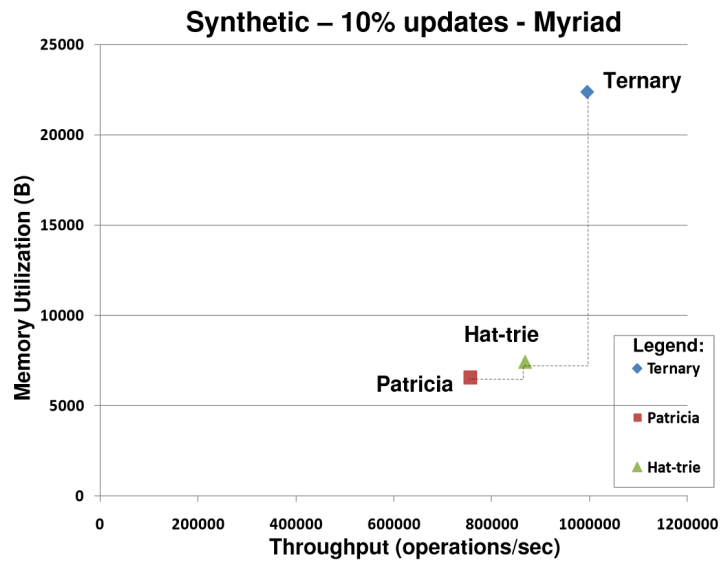
Αν αυξήσουμε τον αριθμό των updates στο 80% (Εικόνα 11), τότε θα έχουμε ένα μεγαλύτερο δένδρο και περισσότερες εισαγωγές ή διαγραφές δεδομένων επομένως τόσο περισσότερες φορές θα χρειάζεται να γίνει η αναδιοργάνωση του στις βελτιστοποιημένες εκδόσεις. Η υλοποίηση με το HAT-trie συνεχίζει να έχει την καλύτερη απόδοση, αλλά αυτή τη φορά η διαφορά με τις άλλες εκδόσεις έχει ελαχιστοποιηθεί. Η κύρια αιτία είναι πως οι γρήγορες αναζητήσεις που προσφέρει αυτή η δομή έχουν όλο και λιγότερη βαρύτητα καθώς ο αριθμός των μετατροπών μεταβάλλεται. Επίσης η συγκεκριμένη τεχνική έχει σημαντική επιβάρυνση όταν υπάρχουν πολλές αλλαγές, επειδή χρειάζεται να τροποποιηθούν οι κόμβοι και να ξαναμοιραστούν τα στοιχεία μεταξύ τους. Η βέλτιστη υλοποίηση με το τριαδικό δένδρο μπορεί να πλησιάσει σε ταχύτητα το HAT-trie απαιτώντας ταυτόχρονα λιγότερη μνήμη. Όσον αφορά τη μνήμη, παρατηρούμε μια αντίστοιχη σχέση με την προηγούμενη περίπτωση.

Τα αντίστοιχα αποτελέσματα για την Myriad πλατφόρμα φαίνονται παρακάτω (Εικόνες 12 και 13). Στην Myriad η σημαντική διαφοροποίηση είναι πως δεν υπάρχει KM επομένως η αναδιοργάνωση του δένδρου δεν έχει κανένα νόημα. Η Myriad έχει πολύ μικρή



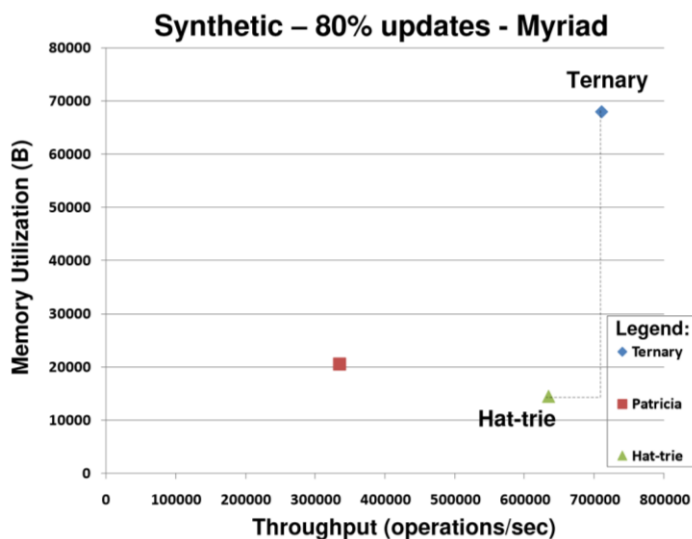
Εικόνα 11: Αποτελέσματα για ένα κατασκευασμένο αρχείο εισόδου που περιέχει μεγάλο αριθμό τροποποιήσεων στον Freescale

μνήμη (της τάξης του 1MB) πράγμα που σημαίνει πως το δένδρο θα είναι αναγκαστικά πολύ μικρότερο. Στο πρώτο πείραμα που οι αναζητήσεις κυριαρχούν το τριαδικό δένδρο παρουσιάζει καλύτερη ταχύτητα από το HAT-trie, ωστόσο υστερεί σε θέματα μνήμης (πίσω και από το Patricia trie). Ο κύριος λόγος για το τελευταίο είναι πως σε μικρό αριθμό δεδομένων, η πιθανότητα να υπάρχουν κοινά προθέματα είναι επίσης ελάχιστη, επομένως μεγάλο ποσοστό των κόμβων του τριαδικού δένδρου δεν επαναχρησιμοποιείται. Το πλήγμα στην απόδοση του HAT-trie είναι άρρηκτα συνδεδεμένο με την έλλειψη KM μιας και είναι μια δομή σχεδιασμένη για την καλή αξιοποίηση της.



Εικόνα 12: Αποτελέσματα για ένα κατασκευασμένο αρχείο εισόδου που περιέχει μικρό αριθμό τροποποιήσεων στην Myriad

Στον παρακάτω πίνακα (Πίνακας 14) φαίνονται ο χρόνος εκτέλεσης και η επιβάρυνση των βελτιστοποιημένων δομών για όλα τα αρχεία ελέγχου. Το κόστος αυτό σχετίζεται με την αναδιοργάνωση του δένδρου που αναλύθηκε προηγουμένως. Πιο συγκεκριμένα το execution time overhead είναι το ποσοστό του χρόνου εκτέλεσης που καταλαμβάνεται από την αναδιοργάνωση, ενώ το memory footprint είναι η επιπλέον κατανάλωση μνήμης όταν ο νέος πίνακας δεσμεύεται ώστε να αντιγραφεί το παλιό δένδρο με έναν πιο αποτελεσματικό τρόπο.



Εικόνα 13: Αποτελέσματα για ένα κατασκευασμένο αρχείο εισόδου που περιέχει μεγάλο αριθμό τροποποιήσεων στην Myriad

Όσον αφορά το επιπλέον κόστος εκτέλεσης, αυτό σχετίζεται στένα με τη στιγμή που θα γίνει η αναδιοργάνωση. Για παρά-δειγμα αν η τελευταία ανα-διοργάνωση συνέβη προς στη λήξη του προγράμματος αυτός ο χρόνος θα είναι σημαντικός, επειδή τότε το δένδρο ενδε-χομένως θα είναι μεγαλύτερο (άρα περισσότερα δεδομένα χρειάζεται να αντιγραφούν) και επιπλέον θα υπάρχει λίγος

	Execution time overhead		Memory size overhead (MB)	
	Ternary cache-opt	Patricia cache-opt	Ternary cache-opt	Patricia cache-opt
Synthetic 10% upd.	5.7%	5.38%	22	28
Synthetic 80% upd.	20%	18.4%	138	168
IP dataset	4.6%	3.6%	3.7	2.6
Dict. 100% upd. alph. order	35%	15.3%	14.3	18.5
Dict. 100% upd. random order	36%	20.3%	14.3	18.5
Dict. 35% upd. random order	20%	12.1%	16.2	32

Πίνακας 14: Επιβάρυνση στο χρόνο εκτέλεσης και στη μνήμη για διάφορα αρχεία εισόδου

διαθέσιμος χρόνος ώστε να πραγματοποιηθούν αρκετές λειτουργίες για να οφεληθούμε απ' την νέα δομή. Γενικά στα αρχεία που έχουν πολλές αναζητήσεις (synthetic 10% και IP που έχει 3%) το κόστος αυτό είναι πολύ μικρό, της τάξης του 5% ενώ όσο αυξάνονται οι τροποποιήσεις στο δένδρο ανεβαίνει μέχρι και 20%. Το επιπλέον κόστος σε μνήμη συχνά δεν είναι πρόβλημα μιας και είναι στιγμιαίο.

Συνοψίζοντας, οι προσθήκες στην DDTR μεθοδολογία με την ενσωμάτωση των φιλικών ως προς την KM δομών δεδομένων που περιγράφηκαν, επεκτείνει την μεθοδολογία καθιστώντας την ικανή να ανταπεξέλθει και σε περιορισμούς ως προς το ίδιο το υλικό. Σε πολλές περιπτώσεις κάποιες Pareto – βέλτιστες λύσεις δεν θα ήταν ανιχνεύσιμες με το προηγούμενο εργαλείο. Η συλλογή των ΔΔ εμπλουτίζεται, δίνοντας στον σχεδιαστή περισσότερες δυνατότητες και ευελιξία.

Τέλος, οι βελτιστοποιημένες ΔΔ μέσω της αναδιοργάνωσης μπορεί να είναι μια καλή εναλλακτική στις κλασικές ΔΔ. Αν το επιπλέον κόστος στη μνήμη και η καθυστέρηση κατά την αναδιοργάνωση δεν είναι σημαντικός περιορισμός, τότε μπορεί να επιτευχθεί μεγάλη απόδοση μέσα απ' την καλύτερη αξιοποίηση της KM του συστήματος.

List of figures and tables

Εικόνα 1: Βήματα της DDTR μεθοδολογίας.....	xi
Εικόνα 2: Pareto-βέλτιστοι συνδυασμοί για ένα παιχνίδι	xii
Σχήμα 3: Ιεραρχία μνήμης σε ένα σύγχρονο επεξεργαστή	xiii
Σχήμα 4: Τεχνική χρωματισμού για διαχωρισμό των δεδομένων.....	xiv
Σχήμα 5: Παράδειγμα συμπίεσης για έναν κόμβο που περιέχει τα στοιχεία ενός πελάτη....	xv
Σχήμα 6: Δομή HAT-trie με τους κόμβους και τα buckets που είναι πίνακες κατακερματισμού	xvi
Σχήμα 7: Παράδειγμα τριαδικού δένδρου για συγκεκριμένες λέξεις.....	xvii
Σχήμα 8: Κόμβοι δένδρου και ο τρόπος με τον οποίο έχουν δεσμευθεί	xviii
Σχήμα 9: Πριν και μετά την αναδιοργάνωση του δένδρου	xx
Εικόνα 10: Αποτελέσματα για έναν κατασκευασμένο αρχείο εισόδου που περιέχει μικρό αριθμό τροποποιήσεων στον Freescale	xxii
Εικόνα 11: Αποτελέσματα για έναν κατασκευασμένο αρχείο εισόδου που περιέχει μεγάλο αριθμό τροποποιήσεων στον Freescale	xxiii
Εικόνα 12: Αποτελέσματα για έναν κατασκευασμένο αρχείο εισόδου που περιέχει μικρό αριθμό τροποποιήσεων στην Myriad	xxiv
Εικόνα 13: Αποτελέσματα για έναν κατασκευασμένο αρχείο εισόδου που περιέχει μεγάλο αριθμό τροποποιήσεων στον Myriad	xxiv
Πίνακας 14: Επιβάρυνση στο χρόνο εκτέλεσης και στη μνήμη για διάφορα αρχεία εισόδου	xxv
Figure 1.1: 3-step DDTR Methodology flow	30
Figure 1.2: Performance vs Energy Pareto points and optimal points	31
Table 1.3: Used ADTs and their implementations in the new library.....	31
Figure 1.4: New library design	31
Figure 1.5: GUI for the new library	31
Figure 1.6: DDT exploration methodology	31
Figure 1.7: All stages of DDT methodology	31
Figure 1.8: Memory accesses vs memory footprint for 2D race game	31
Figure 1.9: Performance vs Energy for 2D race game	31
Figure 1.10: Memory accesses vs memory footprint for Astar algorithm	31
Figure 1.11: Performance vs Energy for Astar algorithm	31
Figure 1.12: Performance vs memory footprint for WFQ algorithm	31
Figure 1.13: Performance vs Energy for WFQ algorithm	31
Table 1.14: Matisse and new library comparison.....	31
Figure 2.1: Memory hierarchy and latency-capacity trade-offs in a typical modern processor	41
Figure 2.2: A linked list and how it is stored in cache	42
Figure 2.3: Example of coloring with 2-way cache with C cache lines	43
Figure 2.4: Original node and the resulting structure after hot/cold area splitting	44

Figure 2.5: Hot/cold splitting	45
Figure 2.6: Dual core processor with shared L2 cache	45
Figure 2.7: Example of overall performance comparison between indiscriminate and selective prefetching algorithms for some benchmarks. The prefetch memory overhead, memory stalls and instruction cost are shown.	46
Table 3.1: Existing DDTR Methodology characteristics and limitations	48
Figure 3.2: A typical HAT-trie. The array of characters is the trie node and the boxes at the last level represent the buckets. If the box is dashed then it is a pure bucket.	50
Figure 3.3: Binary vs ternary tree for the same vocabulary	52
Figure 3.4: searching for a specific string in the Patricia trie.....	52
Figure 3.5: the Patricia trie before and after the subsequent insertion of nodes	53
Figure 3.6: Deletion of a node in a Patricia trie	55
Figure 3.7: Custom allocator	55
Figure 3.8: A tree (right) and its custom allocation pattern through custom_malloc.....	56
Figure 3.9: ideal allocation for a simple binary tree.....	57
Figure 3.10: The reorganization function and its result	59
Figure 3.11: Pseudocode for the reorganization with pointer fixing.....	60
Figure 3.12: list before and after the reorganization	61
Figure 4.1: Throughput vs memory utilization of the synthetic benchmark with 10% updates in the Freescale board.....	64
Figure 4.2: Throughput vs memory utilization of the synthetic benchmark with 80% updates in the Freescale board.....	65
Figure 4.3: Throughput vs memory utilization of the synthetic benchmark with 10% updates in the Myriad board	66
Figure 4.4: Throughput vs memory utilization of the synthetic benchmark with 80% updates in the Myriad board	67
Figure 4.5: Throughput vs memory utilization of the IP benchmark in the Freescale board	68
Figure 4.6: Throughput vs memory utilization of the IP benchmark in the Myriad board ...	69
Figure 4.7: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in alphabetical order in the Freescale board.....	70
Figure 4.8: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in random order in the Freescale board.....	70
Figure 4.9: Throughput vs memory utilization of the dictionary benchmark with 35% updates and strings inserted in random order in the Freescale board.....	71
Figure 4.10: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in alphabetical order in the Myriad board	72
Figure 4.11: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in random order in the Myriad board	72
Figure 4.12: Throughput vs memory utilization of the dictionary benchmark with 35% updates and strings inserted in random order in the Myriad board	73
Table 4.13: Execution time and memory footprint overhead of cache conscious implementation on Freescale board.....	74

CHAPTER 1

The Dynamic Data Type Refinement (DDTR) methodology

1.1 Introduction

Modern applications in various fields, such as multimedia and networking, are becoming more and more complex and demanding. These applications often require high performance which yields high memory consumption. Additionally, there is an increase in the dynamism of the access pattern due to the high degree of interaction with the environment (such as in the case of the wireless networks) [1].

There are many factors that contribute to the application's dynamic behavior. In the case of the wireless networks for example, depending on the network traffic, there may be more packets to process with different sizes and arrival times. The packages must be stored and then processed. In gaming applications, the actions of the user, that are considered unpredictable, constantly change the game's state and as a result, the resources needed by the system as far as memory and computational power are concerned.

As a corollary, these types of applications do not allow for static memory allocation during the compile time, before the program is going to be executed. This would be very convenient because it is simpler, as the variables are bound to specific memory addresses throughout the execution of the program, and is efficient without additional overhead for dynamic allocation. On the other hand, it is very limiting and almost never suitable for modern applications, because it is inflexible and cannot react to changes. Furthermore, it usually requires allocating in advance enough memory to cover the worst condition that the algorithm may face and this fact, apart from being difficult to predict for complex applications, can also lead to wasted memory space and suboptimal usage.

Energy consumption and performance are the two most important metrics of the efficiency of any embedded system. Especially in a large category of popular embedded systems, such as handheld devices, energy consumption becomes more and more crucial as the amount of processing power is usually satisfactory, but the duration of the battery (most of the times) is not. Lower energy consumption from the memory system also leaves much room for other improvements, such as bigger screens and more powerful processors, signal receivers and more functionality. The memory subsystem is closely related to the energy consumption because the memory must be constantly powered in order to keep the data (leakage energy). More memory will require more power, all other things being equal. Also, the faster the memory (SRAM), the higher the cost. Even the distance from the processor affects memory consumption. Finally, depending on the memory, the existence of a cache can radically change the performance and energy footprint of the program. A cache hit will

generally reduce the cost of the access a lot while a cache miss will greatly increase it (further if subsequent misses occur in the next levels).

To avoid the issues of wasted memory and inflexibility of a static allocation strategy, the vast majority of modern applications use Dynamic Data Types (DDT) and dynamic allocation and management [2]. These structures can be allocated and deallocated at runtime, making it easier to follow the fluctuations of the storage requirements throughout the life of the program and at the same time, using only as much memory as needed. The most characteristic example of a DDT is the linked list.

Although the dynamism is a common ground of all DDT, there is still a great variety of dynamic structures, each one designed to cover different needs. In addition to linked lists, singly and doubly linked, there are also trees, arrays, hash tables, queues, stacks and more. Many data structures can be utilized for the same purpose, but each one of them has distinct advantages and disadvantages, like more efficient storage or lower search and update times. A dynamic array (similar to C++ Standard Template Library) is better for fast random accesses, but generally requires more space than a list or a tree, as it needs to be large enough (because frequently changing the size can be expensive or impossible), whereas the other two can easily change their size according to memory size requirements. The existence of the aforementioned trade-offs makes the correct selection of the appropriate DDT crucial for the application's runtime characteristics and it is usually determined by the requirements of the program and utilized algorithms. There is no general solution that is optimal for every possible case, so each application must be examined separately.

Selecting the correct dynamic data structure for a specific application is not always intuitive. The application may have different requirements according to the platform that runs. A desktop computer has more memory to spare than a cellphone device. The requirements may even vary among different cellphones. Also, most algorithms use by default specific data structures (such as priority queues) for performance and the programmer will need to run his own benchmarks and study each algorithm separately to change some of the code to make it suitable for the specific application. It is obvious that this is an error-prone and tedious procedure that requires a lot of time.

In this diploma thesis, we begin by presenting the DDTR methodology in the first chapter. Its original implementation and its basic are explained as well as some additional work that aimed at improving it. In the second chapter, we analyze some features of the hardware of the embedded systems (cache memory) that play a major role in the selection of the data structure and the application's overall behavior, but nevertheless are ignored by the DDTR methodology. We present some techniques for improving the cache behavior of the data structures. In the third chapter, a methodology is presented for giving some tree data structures improved cache characteristics and the challenges that occur in the process. Finally, the modified data structures are evaluated through a wide set of benchmarks and the results are analyzed. In the last chapter, some conclusions are drawn as well as ideas for further work in this direction.

1.2 The DDTR Methodology

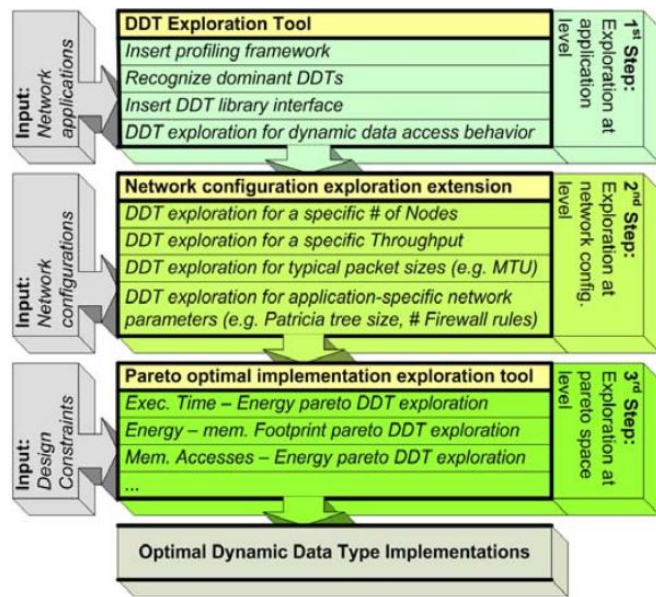


Figure 1.1:3-step DDTR Methodology flow

A systematic, step by step methodology is needed to help the designer make the correct decisions by presenting the trade-off between various DDTs in a given application. This methodology is the Dynamic Data Type Refinement (DDTR) Methodology and was initially developed for the systematic exploration of networking applications [3]. It consists of three distinct steps applied to each application (Figure 1.1).

The first step is the exploration at the application level based on the application dynamic data access behavior. Each DDT of interest in the application is marked for profiling and then the application is run for a typical representative input. The most active data structures are revealed by this process and the source code is modified to link each dominant data structure with a custom C++ DDT Library. At this first implementation, the DDT library was comprised of 10 different DDTs and is described in [4]. This procedure does not alter the functionality of the application. The typical functions such as insert, delete, modify are supported. Then the dominant DDTs are automatically changed and all combinations are used while the application runs with the trace input. Then, the combinations that yielded the lowest energy consumption, shortest execution time, fewest memory accesses and lowest memory footprint are kept.

For the second step the difference of various network configurations are taken into account. The network parameters are extracted from the data in the traces. The most important parameters for networks applications are usually the number of nodes, the throughput and the typical packet sizes used (e.g. MTU packet size). Other parameters are more related to the application like the Radix Tree size which is important for the IPv4 routing application [5] and greatly affects the exploration. This stage requires input traces that are relevant to the network configuration. The combinations from the previous step are taken and each one of them is simulated for all different configurations. Then again the best results are selected.

In the third and final step, the designer is provided with a Pareto-optimal set represented by a Pareto curve (Figure 1.2) after the results of the log files from the previous steps are processed. Instead of one solution the designer has many choices of combinations

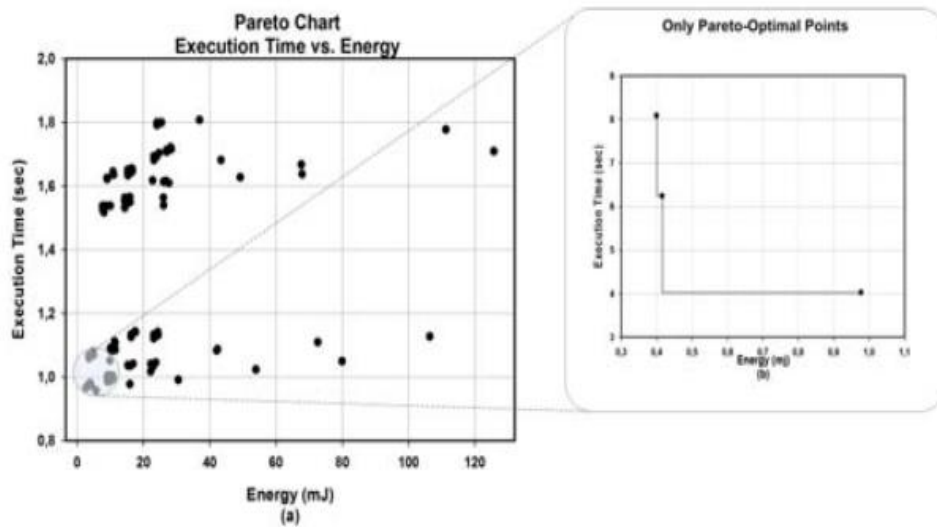


Figure 1.2: Performance vs Energy Pareto points and optimal points

that yield different trade-offs (one may have better energy performance while another faster execution). From this graphical view it is easy to select the appropriate method that satisfies the application's needs.

This novel methodology and the supporting automation framework led to acceleration of the execution time by 20% in average and reduction of energy consumption by 80% on selected applications. Those applications consist of a routing application, a context switching algorithm, a firewall application and the Deficit Round Robin scheduling application. Those applications were taken from the NetBench Benchmarking suite [5]. Another advantage that does not appear in the figures is, of course, the reduction of the design time and the fact that the methodology does not require any changes as far as the hardware and the application's functionality are concerned.

1.3 Limitations of the original implementation and the new DDT library

The exploration in the aforementioned methodology is supported by a DDT library that is called Matisse profiling tool. Its main disadvantages are the low flexibility and the limited DDT support that do not allow the application of this methodology to complex applications with complicated dynamic behavior. Modern multimedia and network applications utilize many advanced data structures, such as sorted lists and hash tables, which are not supported by the Matisse tool. Additionally, this tool does not distinguish between DDT functionality and DDT implementation, nor exploited the advantages of the object oriented programming. As a result, the extension of the library and the combinations of DDTs in more complex structures is rendered extremely difficult. Also, limitations in the design make the integration with the application difficult because each object was treated as

an array of basic data types (like integer). For larger applications, applying the tool is infeasible and time consuming. Finally, the user is unable to reduce the exploration space by selecting specific combinations of DDTs and therefore the exploration time is prohibitive for complex programs as every possible combination must be investigated.

Therefore a superior DDT exploration methodology was developed based on a new DDT library tool [6]. This new library follows a different approach as it introduces the concept of Abstract Data Types (ADT). The ADT is an abstraction layer between the application and the data and contains the methods that are used to access and modify the data. The exploration is easier and therefore the methodology can be applied to various domains, apart from networking applications because the application can ignore the underlying implementation and just conform to the interface to serve its requests. The common operations for every data type are:

- Insert: addition of an element in the data structure.
- Remove: deletion of an element from the data structure.
- Get: returns the element's value without modifying it.
- Set: modify an element to have the desired value.

The abstract operations make it easy to swap between different data types without the need to change the source code. Additionally, we can have many implementations for each abstract data type and the DDTs are now the concrete instances of the ADTs. Below we present the supported ADTs of the new library:

- ✓ *Stack*: LIFO (Last In First Out). A pile of objects that can be pushed or popped from the top of the data structure.
- ✓ *Queue*: a collection of objects that can be inserted from the one end and removed from the other end.
- ✓ *Deque*: an extension of the queue where objects can be inserted and removed from either end (double-ended queue).
- ✓ *Unsorted List*: a collection of elements where each one (except the first) has a unique predecessor and (except the last) a unique successor. In the unsorted list the elements are not in any particular order.
- ✓ *Sorted List*: like the unsorted list but the elements are sorted based on some criterion, such as the value or the lexicographic order.
- ✓ *Hash Table*: uses hashing techniques to locate elements in relatively constant time in a structure by extracting its locations from the unique key that is assigned to every element.
- ✓ *Set*: an unordered collection of elements (without repetitions), chosen from a pool (base set).
- ✓ *Multiset*: like the set, but without the restraint of each element appearing at most once.
- ✓ *Tree*: non-linear structures that are generally used to represent a hierarchy. Each element in the tree (node) has some children and one parent (except the first).

From the aforementioned list the Matisse tool supported only the Unsorted List DDT. Each of the ADT can be realized by different DDTs as shown in Table 1.3. Depending on the selected DDT each operation is implemented in a different way. For example the ADT for the queue supports the classic operations of push and pop. How those operations are going to be implemented is dependent on the decision whether an array or a linked list will be used for the queue. This object oriented approach makes it easy to extend the list and insert new implementations.

ADT	Data structures (ADT implementation variations)
Stack	Stack As LinkedList Stack As Array
Queue	Queue As LinkedList Queue As Array
Deque	Deque As LinkedList Deque As Array
Unsorted List	List As Array List As Array Embedded SLL LinkedList DLL LinkedList SLL LinkedList with Roving Pointer DLL LinkedList with Roving Pointer SLL LinkedList with Arrays DLL LinkedList with Arrays SLL LinkedList with Arrays and Roving Pointer DLL LinkedList with Arrays and Roving Pointer
SortedList	List As Array List As Array Embedded SLL LinkedList DLL LinkedList SLL LinkedList with Roving Pointer DLL LinkedList with Roving Pointer SLL LinkedList with Arrays DLL LinkedList with Arrays SLL LinkedList with Arrays and Roving Pointer DLL LinkedList with Arrays and Roving Pointer

Table 1.3: Used ADT and their implementations in the new library

Another issue of the Matisse tool that the new library attempts to solve is the combination of the DDTs to form some complex data structures. By taking advantage of the object oriented programming it is possible to create new DDTs, such as arrays of lists and arrays of lists of arrays and so on. Therefore, there is a multilevel implementation that can be expanded and made as complex as necessary with the only restraint being the usefulness of these new DDTs. For simplicity and effectiveness, only a two-level implementation is examined as shown in Table 1.3. Examples of two-level DDTs are the SLL LinkedList with Arrays and Roving Pointer. The roving pointer provides an optimization for sequential. The Matisse library tool has a more monolithic design. The supported DDTs cannot be combined to form multi-layered new DDTs. Also the fact the Matisse handles application's object as arrays of basic data types requires decomposing every class of the application. The overall design is summarized in Figure 1.4.

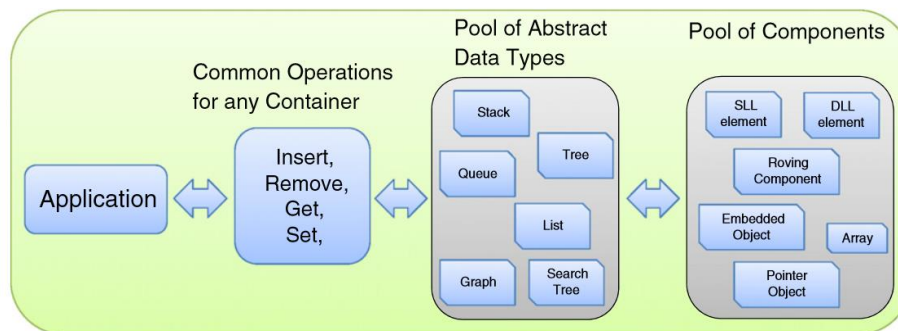


Figure 1.4: New library design

Apart from the new library, a GUI was developed for the easier tracking of the exploration process and the convenience of the designer. The DDTs that are present in the application are spotted and then presented to the designer alongside alternative implementations for them. For example, in Table 1.3 we can see that there are 10 possible implementations for the unsorted list and 2 for the queue. So if an application used these two ADTs the user will view the window that is depicted in Figure 1.5. Then some combinations can be omitted if the designer desires to reduce the exploration space. The output is a file that contains the Pareto points that represent the application's behavior and a file with the optimal Pareto points.

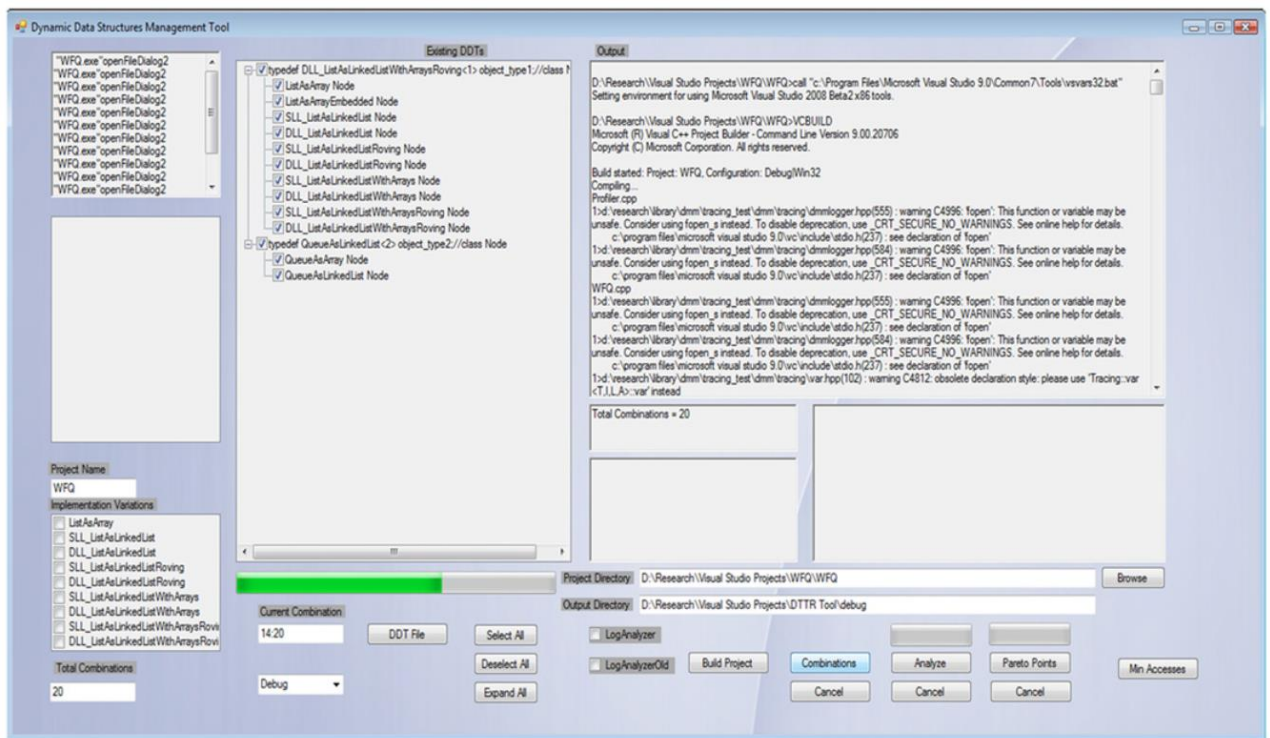


Figure 1.5: The GUI for the new library

The tool [7] also keeps a record of the memory requirements of every DDT in the application and logs all memory accesses whether they are reads or writes. This allows for the computation of the energy requirements. For the experiments, an external memory is used and the knowledge of the accesses to this memory is sufficient to extract the energy consumption by using the appropriate energy model.

The methodology for this new tool is more general than the previous one which was constrained to networking applications only, although the basic principles remain the same. It consists of two stages. At first the DDT exploration takes place, where each DDT is evaluated and different combinations of DDTs are tried and profiled while the results are logged. In the next stage, the optimal DDTs are selected through a Pareto optimal exploration, while at the same time the designer's constraints are satisfied. This flow is shown in Figure 1.6. The automatic exploration methodology's aim is to present the suitable information for the designer to select the best for each case DDTs in application, much like

the original methodology presented at the beginning of the chapter. But, there are critical differences that will become clear in the following sections.

1.4 DDT Exploration Stage

In this first stage, the input is the source code of the application under investigation and the result is the profiling information. The library code must be plugged into the source code. There are two ways of achieving this. Either the programmer must manually swap the operations that are related to dynamic data storage and retrieval with the corresponding ones from the library, or the programmer has nothing to do because the code is already STL compliant. As the library uses the STL interface, no manual intervention is required. This is very important because some applications have huge and complex code that can pose many difficulties to modify. But if the STL interface is followed, the process is fully automated in contrast with the previous tool that did not support any automation. Also, as more DDTs are supported, a wider range of applications are suited to be optimized.

After the DDTs are identified, either manually or automatically, they need to be evaluated. The need arises from the fact that the application may (and usually does) contain many data structures and types that do not play a major role as far as the program's performance, energy consumption or memory are concerned. Therefore, there is no reason for the exploration to consider these DDTs because of the additional exploration cost that they will incur without providing any benefits. The methodology will check every possible combination of DDTs and DDTs without any impact and this will just prolong the process. To counter the issue, only the DDTs that are relevant are identified and considered for the exploration process. To find the dominant DDTs, the number of accesses for each individual DDT and the number of objects that it hosts are used.

The last step consists of the exploration itself. The designer is presented by the possible options and selects what combinations (all or specific to reduce the exploration time) will be investigated through a GUI. Then the tool automatically evaluates these combinations. In this methodology only specific DDT are explored for each DDT existent in the source code, whereas in the previous one all DDTs were explored by default. For example, if the DDT is Queue, according to the Table 1.3, only two cases will be considered. This further reduces the required time.

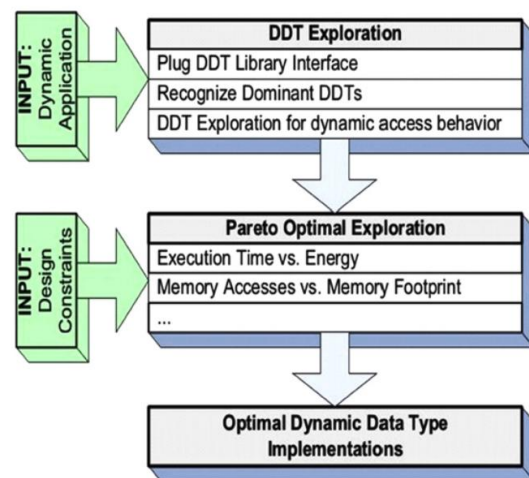


Figure 1.6: The DDT exploration methodology

The output of this stage is a file that contains all relevant information and has to do with memory and energy requirements as well as performance for each combination.

1.5 Pareto Optimal Exploration Stage

After the first stage has finished and produced the output file, in the second stage, this file is processed by a script and the optimal Pareto points are produced taking into account the design constraints. This stage is fully automated as well and creates some Pareto charts. Each Pareto point corresponds to a different configuration (i.e. combination of DDTs). The bigger variety of DDTs compared to the Matisse tool makes the procedure more complete and can reveal Pareto points that were previously unreachable. The whole process is displayed in Figure 1.7. The final output are the Pareto graphs that will help the designer decide what combinations are better suited to the application's needs. In contrast with the previous methodology the network configuration layer (Figure 1.1) is absent as the target group is no longer only network applications.

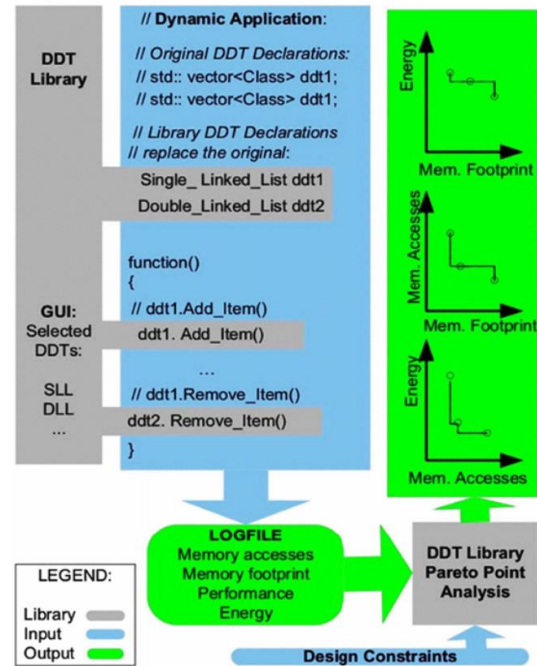


Figure 1.7: All stages of DDT methodology

1.6 Experimental Results

The DDT exploration methodology was performed on various multimedia applications. The memory hierarchy was an external MICRON SDRAM [8] at 266MHz with a size of 8MB. The data from the profiling (such as memory access) were fed to the MICRON energy model to produce the energy requirements of the application, while the execution time was calculated by operating system instructions. The platform used was an Intel Pentium4 3.2GHz with 1GB RAM.

The benchmarks selected were a 2D racing game, the Astar algorithm [9], a tile game (Comboling) and a 3D environment builder (Simblob) [10], as well as, the Dijkstra [11] algorithm from Mibench suite [12] and a Weighted Fair Queuing (WFQ) algorithm [13] – a modified version of Deficit Round Robin (DRR) – taken from Netbench [5].

For the 2D racing game, like a normal racing game, the environment changes and move (in different speed) according to the car movements and speed. The environmental objects are stored in an unsorted list and are constantly upgraded. According to Table 1.3 there are 10 implementations for the unsorted list ADT. By examining different options we get the following graphs (Figures 1.8 and 1.9):

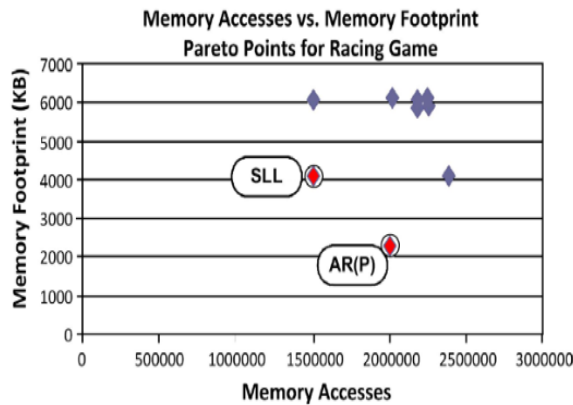


Figure 1.8: Memory accesses vs memory footprint for 2D race game

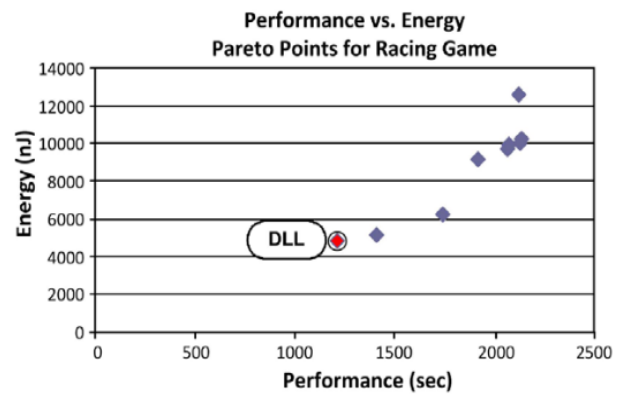


Figure 1.9: Performance vs Energy for 2D race game

As far as the smallest memory footprint is concerned we can conclude that the Array with Pointers – AR(P) DDT wins, while on the other hand the Double Linked List (DLL) has the best performance and the lowest energy at the same time. The optimal Pareto points are the ones marked. In Figure 1.8 there are two choices because no choice is best in both cases (axes), while in the second graph (Figure 1.9) there is a selection that optimizes both criteria. The usage of AR(P) lowers the memory footprint by 44% compared to the original program and the DLL lowers the energy consumption by 6%.

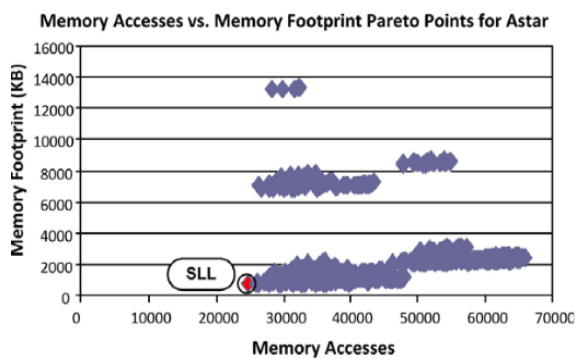


Figure 1.10: Memory accesses vs memory footprint for Astar algorithm

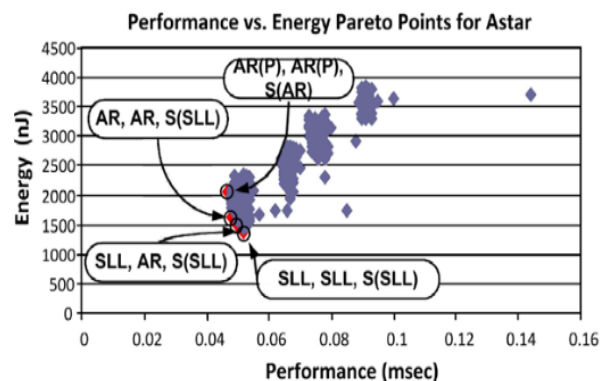


Figure 1.11: Performance vs Energy for Astar algorithm

In the case of the Astar algorithm we can get similar graphs. The Astar algorithm [9] is a pathfinding algorithm that uses a heuristic to prune the search space and is frequently used in many multimedia applications, such as video games. The DDTs that are prevalent in the algorithm are 1) a sorted list (according to a specific criterion) that contains the nodes to be expanded 2) an unsorted list for the nodes that are closed 3) an unsorted list for storing the

neighbors of each node (the successors). The behavior of different DDTs can be shown in the following graphs (Figures 1.10 and 1.11).

This time there is a Pareto optimal solution for the memory accesses and memory footprint and many solutions for the performance and energy. There are triplets because combinations of DDTs are tried as there are 3 distinct data structures in the source code of the algorithm. So for example the (AR(P),AR(P),S(AR)) that is the most energy efficient would mean AR(P) for the first and the second list and S(AR) for the third. If the best performance is selected, a gain of 10% over the original code can be achieved.

Finally, the Weighted Fair Queueing algorithm is a common packet sharing algorithm that allows a number of flows to share the same link. It is implemented with switching devices. It is an approximation of the generalized processor sharing (GPS) scheme, where every flow with a non-empty queue, at any time, is served simultaneously and the bandwidth is equally distributed to each flow. The most important data structures are 1)the class Packets which encapsulate the information of the packets to be scheduled in queues, therefore has a corresponding ADT of Queue and 2)the class Nodes that creates nodes where the packets are stored and scheduled, with the corresponding ADT of Unsorted List. As a result, there are 2 possibilities for (1): Queue as Array and Queue as Linked List and 10 possibilities for (2).The Pareto optimal points appear in Figures 1.12 & 1.13. So for example, if for the class Packets we select Q(AR) – Queue as Array and for the class Nodes SLL – Single Linked List we achieve 22% speedup compared to the original implementation.

To sum up, the library can be evaluated based on the experimental results. In the case of the 2D game it was easy to plug in because the game itself used the STL interface. On the other hand, the user of the Matisse tool must perform some code modifications to insert the required interface for the tool to run. Also, the Queue, Stack and Sorted List ADTs are not supported by the Matisse tool and therefore some of the applications – such as the WFQ – would be ineligible for full exploration as far as all the data structures are concerned and the corresponding Pareto optimal points would be unobtainable. In the following table (Table 1.14) the differences and similarities of the two tools are summarized.

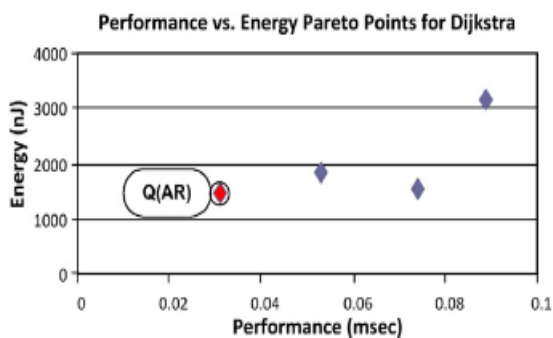


Figure 1.12: Memory accesses vs memory footprint for WFQ algorithm

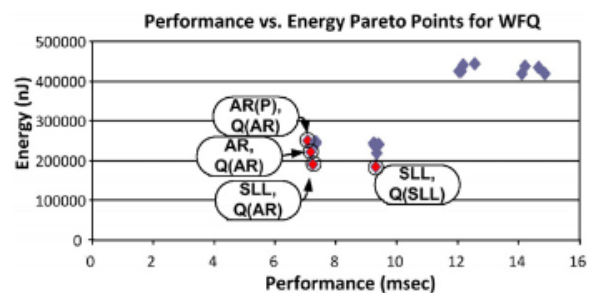


Fig. 19. Performance vs. energy consumption Pareto points the WFQ.

Figure 1.13: Performance vs Energy for WFQ algorithm

Feature	Matisse profiling tool	New DDT library
Abstract data type (ADT)	Not supported	Supported
DDT implementations provided	Limited (Unsorted Lists only)	Extended (Sorted Lists, Trees, Sets, and more)
Extension of the DDT library	Not easy	Easy (due to the object oriented design of the library and the ADT support)
Combination of DDTs in more complex ones	Supported	Supported, but easier (due to the object oriented design of the library)
Integration in complex applications	Not easy (each object is handled as an array of basic data types)	Easy (the actual object is inserted in the DDT)
Interface	No STL compliant	STL compliant
Selection of specific DDT combinations to be evaluated	Not Supported	Supported
GUI	Not provided	Provides a GUI for simple use
Benchmarks evaluated	NetBench	NetBench, MiBench, ALPBench, etc.
Benchmarks code size	A few KB	Tens of MB
Application domains	Network applications only	Modern network, multimedia applications, games, etc.

Table 1.14. Matisse and new library comparison

CHAPTER 2

The impact of cache in modern embedded systems

The DDTR methodology was thoroughly described in the previous chapter, both the original implementation and the new library that improved it. The general idea is to run an application for a different combination of data structures and profile it. That way an exploration is performed and the best combination is selected based on different criteria, such as performance-memory trade-offs. However, there are some very obvious limitations and facts that were overlooked.

At first, consumer electronic devices relied on non-programmable circuits for their operation (either streaming or something else) and that made them simpler. However, recent demands on flexibility moved the balance towards programmable components to increase their versatility. This fact makes the embedded systems more and more unpredictable, because to cope with current needs their architecture has to become more complex [14]. For example, it is not rare, even for common embedded systems, to use parallel architectures and sophisticated memory hierarchies.

The computational power of modern embedded systems is constantly increasing and that is a trend that will continue in the foreseeable future. Nowadays, demanding applications that, just a few years ago, were executed exclusively in High Performance Computer (HPC) systems, invade the field of embedded and handheld systems creating new requirements. Characteristic examples are embedded servers and multicore heterogeneous architectures that integrate both embedded cores and FPGA programmable logic [15] that execute complex applications and require high processing power. Database applications and streaming are common ground for HPC. Their connection is the large amount of data that they need to store and process.

The term High Performance Embedded Computing (HPEC) now refers to embedded devices with huge computational capabilities that are used mostly in aerospace and military applications [16]. Their advantage is their energy efficiency that makes them competitive in the market over large power systems. The optimization of data structures is not only related to the application's access pattern, but also with the underlying hardware specification where the application is executed. Some embedded systems have caches or scratchpads and complicated memory hierarchy that are specifically designed to take advantage of particular characteristics of a program. Others are multicore systems that aim to increase throughput. This introduces huge challenges for various reasons that will be made clearly in the rest of the chapter. Many modern embedded systems resemble more and more the general purpose computers that we are all familiar with, although they may lack some of their capabilities depending on the embedded system needs (such as connectivity, I/O devices and more).

2.1 Why cache is used in modern embedded systems

Caches in modern embedded system are a common practice to improve performance and energy. There are many components of a program that are sequential or frequently used and allow the exploitation of a fast memory, such as cache. The program's code is, most of the times, executed line by line and if some jumps in the code occur, they are usually either short or infrequent compared to the total instructions that are executed. Therefore, the code's instructions are usually stored in a special instruction cache for faster instruction fetch. Data re-usage is also fairly common. To get advantage of the data spatial and temporal locality it is beneficial to use a data cache.

Apart from utilizing some internal properties of the application (such as locality) a cache is also required from a practical point of view. There is a gap between the processor's and the memory's speed that is constantly increasing. Microprocessor's performance has improved over 60% each year for almost two decades, while memory access time has decreased by less than 10% per year. That means that the processor can compute data faster that they can arrive from the main memory. This leads to wasted power as there are intervals where the processor is just idle, waiting for the data. The cache is the link between the fast processor and the slower memory. As the cache is smaller (order of magnitudes) than the main memory it is much faster. The cache's speed depends on the technology used and on its size. The smaller the cache, the simpler the subsystem, that is used to search for a block, and therefore faster. Also, a small cache can be created by high-tech expensive materials that further boost access speed, but would be very cost-inefficient for larger memories. Finally, as the caches are closer to the processor the data need to transfer only a fraction of the distance to the main memory. All these factors can make a cache access 10-200 times faster than a main memory and tens of millions time faster than a disk access. There are many levels of cache in modern computer processors usually 2 or 3 named L1, L2 and L3 cache respectively. The higher the level, the bigger the cache, but the access is slower (Figure 2.1).

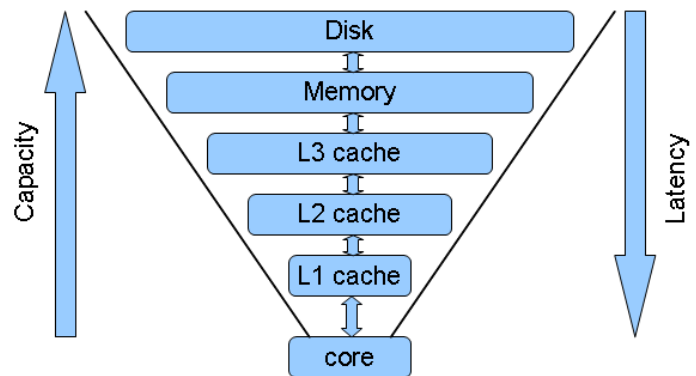


Figure 2.1: Memory hierarchy and latency-capacity trade-offs in a typical modern processor

An L1 cache can usually rival the processor's speed. If the data is found in the cache, a cache hit happens and if not, a cache miss. In the case of the cache miss, the next level in the memory hierarchy is examined and so on. Cache misses can be particularly costly especially deeper in the memory hierarchy. Apparently the cache is a very important component of every modern processing system that deal with data intensive or general purpose applications (such as multimedia or networking or games). Programmers however, generally have no administrative control over these caches therefore it is important for the

programs and the data structures to display a cache-friendly behavior. This is usually done by making the memory accesses more predictable or regular.

2.2 Designing cache-friendly data structures

From a software perspective, programming languages has also evolved. In the beginning, some primitive languages, such as Fortran and Algol that were used for scientific operations, did not support pointers and stored their data in arrays instead. Subsequent languages (C, Pascal) supported pointers and new data structures were created. Programs that make extensive use of pointers became popular and unsurprisingly, the techniques used for data manipulation in arrays were not as effective for pointer-manipulating programs [17]. Traditionally, pointer-based data structures were designed as if memory accesses costs were uniform. Reference locality can be improved either by changing a program's data access pattern or its data organization and layout.

Here some techniques for improving cache performance will be analyzed. Some of those techniques [18] will be used in the subsequent chapters.

2.2.1 Clustering Technique

The clustering's main purpose is to put together data that are likely to be accessed contemporaneously in the same cache block (the memory unit that is transferred between the cache and the main memory). This way, when something is fetched in the cache, it will bring with it other elements that are going to be used soon. This technique resembles an implicit prefetching.

For example clustering can be used for a linked list data structure that is usually accessed sequentially. If each node is small enough so that multiple nodes can be fitted in a cache block (usually 64 bytes) then we can allocate multiple nodes at once with the same malloc (as an array) instead of each one separately. When allocating an array, all cells are located in adjacent memory addresses. On the other hand if we allocate each node separately, adjacent nodes could end up in distant memory addresses. This is illustrated in Figure 2.2. Every line of the 2D array represents a cache block, so each cache bock has space for 4 list nodes. If we allocate each node separately there is no guarantee that they will end up in the same block. If they end up in the same cache block however, as depicted in the second array, when we access one of the nodes, the rest of them will also be in the cache so we can reap huge benefits in the case of sequential access. The

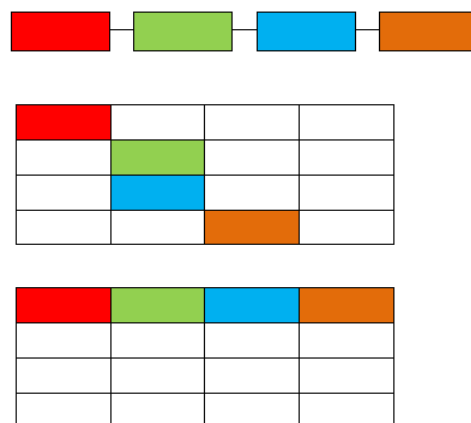


Figure 2.2: A linked list and how it is stored in cache

clustering technique that is best depends on the access pattern. The above scheme assumed sequential access. If the manner that the data are accessed is known in advance, it is highly probable that a good clustering scheme can be extracted.

2.2.2 Coloring Technique

Like clustering, coloring is another technique that aims at the improvement of the program’s cache behavior. Coloring attempts to reduce conflict misses. Most caches typically have finite associativity and that means only a limited number of elements can be mapped to the same cache block without competing with others. The idea is to give specific addresses to the elements that we wish to keep together in cache.

For simplicity, we assume a 2-way associative cache, so each cache line has enough space for two cache blocks. A cache with C cache lines is partitioned into 2 regions (if we use 2 different colors). The one region will contain p cache lines, while the other will contain $C-p$ cache lines as shown in the Figure 2.3. The elements that we know (or suspect)

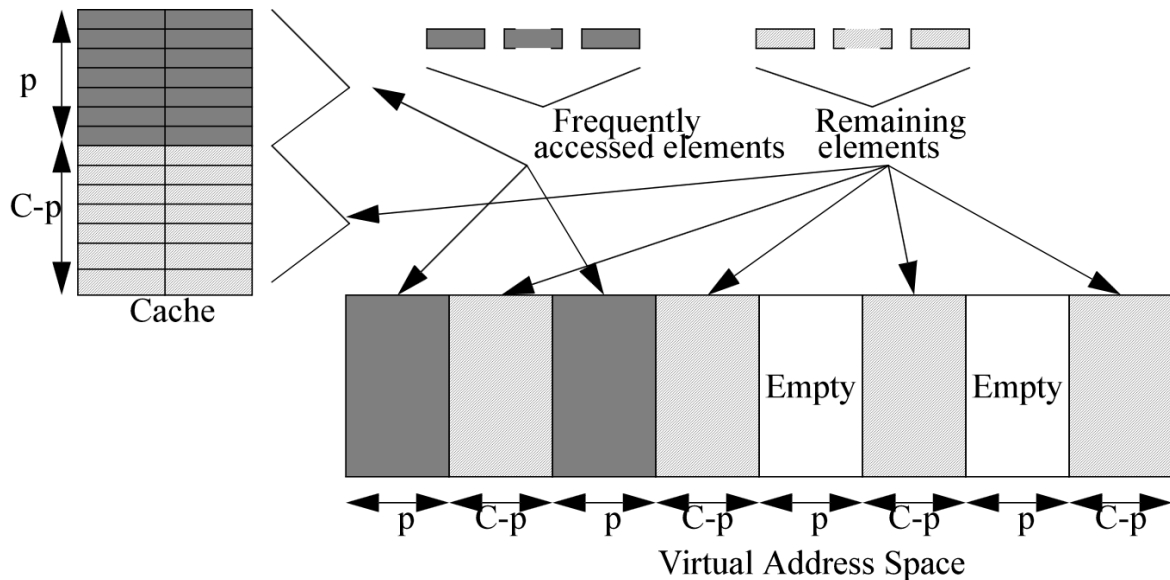


Figure 2.3: Example of coloring with 2-way cache with C cache lines

that are frequently accessed are desired in a separate area that the remaining elements that are infrequently accessed. The reason being that, if both types of elements share the same areas of the cache some infrequent data, when they are ultimately accessed, will force the frequent data out. Therefore, instead of paying the cache miss cost for the infrequent data – that are very unlikely to be in cache as between two accesses many other elements are expected to be fetched – we have to pay additional cache misses for the frequent data that are pushed out of the cache. These are the conflict misses that the coloring technique attempts to counter.

To achieve that “logical” separation of the cache into two pieces, we need to allow specific gaps in the virtual address space. Those gaps represent the regions that are mapped to the $C-p$ cache piece. Those memory can be utilized to accommodate the infrequent

elements or (if not enough are allocated to fill all the gaps) can be left empty. To summarize, the idea is to use specific address ranges for the frequent elements and the rest for everything else.

2.2.3 Compression Technique

Compression is a relatively easy-to-use technique that nevertheless can yield significant benefits. Compressing a data structure allows more elements to “fit” in the same cache block. A direct effect is the reduction of capacity and conflict misses. Also, the program’s memory footprint is shrunk. The downside is the requirement of more processing operations in order to make sense of the compressed information, or in other words, decompress. Some techniques may include data encoding, such as key compression, or structure encoding techniques, such as pointer elimination and hot/cold splitting.

The pointer elimination replaces the pointers in a data structure with integer offsets. So for example in a heap that is implemented as an array, instead of actual pointers to the correct element of the array, it is better to have just the corresponding offset. An integer needs less space to be stored than a pointer – in a 64bit system, as it is often the case, it requires 8 bytes while a 4 bytes for an integer is most of the times sufficient except from extremely huge data structures. The amount of memory that is saved can be significant especially if the system needs many bytes for pointers and the pointers are a considerable portion of the node’s memory footprint.

In many cases of data structures, during searches (lookup operations) only a small portion of the actual memory that the node takes up is examined. A node can be filled with additional information. For example, the node contains the contact information of the person (name, address and so on). We know that the search is based on the name only, and if the person exists in the database, then we retrieve the rest of the information for further processing. Hot/cold splitting is basically the isolation of the node’s elements that are accessed during a favorite operation, i.e. an operation that is fairly common and frequently executed, from the rest of the elements in the node. The separation is done through a pointer. The modified (or compressed node) keeps the relevant fields, while the rest of the fields are stored in a separate data structure that is accessed through a pointer. An example is shown in Figure 2.4. This kind of compression increases the structure’s memory footprint, as the same fields continue to be stored but with the added overhead of an additional pointer per node. However, this increase is not harmful, as the new structure has better cache properties. By utilizing other

techniques, such as clustering (that is explained previously), it is possible to fit more compressed nodes in the same cache line. If those nodes represent the nodes of a linked list that is accessed sequentially until a specific

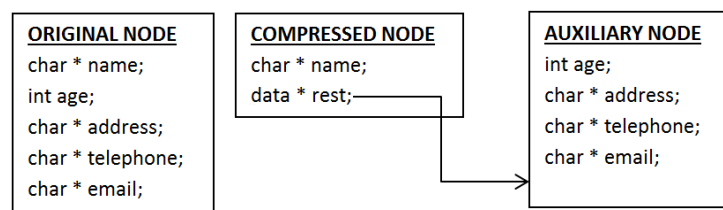


Figure 2.4: Original node and the resulting structure after hot/cold area splitting

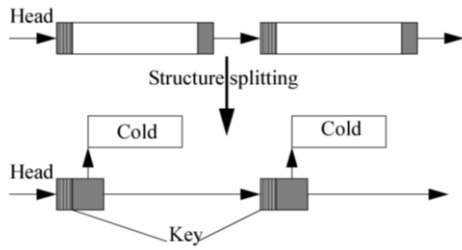


Figure 2.5: Hot/cold splitting

name is found, instead of fitting only one node per cache line we can now fit more, depending on the total size of the information that we wish to hide. So after each node is fetched the next nodes will be as well and that will contribute to fewer misses and faster search. In Figure 2.5 the cold and the hot areas are shown in the general case.

2.3 Effect of the cache memory on the DDTR Methodology

The DDTR is a platform independent methodology and none of the steps described in Chapter 1 takes hardware specifications into consideration, as there are no relevant parameters that refer to the underlying system that the application is executed. However there are many hardware parameters that greatly affect the performance of the system and are mentioned here.

The existence (or not) of cache memory can make some metrics, such as the number of data accesses that were used in typical DDTR, misleading when the number of memory accesses is evaluated – for example, let us consider a system with some sort of cache memory. Not every memory access is the same. There are accesses that refer to data in the cache and others that go deeper in the memory hierarchy. As a result, a program with 1000 accesses does not mean that it has worse memory behavior than one with 10000, especially if we consider that all these accesses could be cache misses and therefore incur huge overhead. If we consider that each cache miss can be hundreds of times slower than a cache hit, the program with 10000 accesses can outperform the one with the lower number under specific circumstances.

Apart from the cache, the DDTR Methodology also ignores other aspects of the hardware that nevertheless impact the program’s behavior. Increasing the processor’s speed indefinitely and shrinking its area is not viable anymore thanks to the increase in the heat it produces, so the next trend is to fit many processors instead of a powerful one. Many applications nowadays are designed to take advantage of multicore systems with concurrent data structures that allow simultaneous operations by many cores. The number of CPUs that the program needs to run can greatly affect the performance and the results. Usually each CPU has its own L1 cache and other parts of the memory hierarchy (like the L2 cache and the main memory) are shared with the rest of the cores, as shown in Figure 2.6. Maximizing the usage of the resources provided by the multicore systems

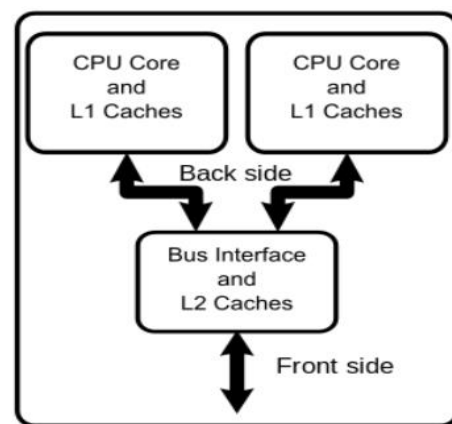


Figure 2.6: Dual core processor with shared L2 cache

require adjustments both to the operating system support and to existing application software. Increasing the number of cores poses new challenges as locking techniques must be carefully consider in order to keep the data consistent without impeding the accesses and creating bottlenecks that will negatively impact the program’s performance.

Modern processors also have sophisticated prefetching mechanisms [19]. Data prefetching is a data access latency hiding technique, which decouples and overlaps data transfers and computation. As CPU usually stalls on a cache miss, in an effort to reduce this idle time, data prefetching attempts to predict future data accesses and initiates a data fetch so that the requested data will be closer to the processor before it is requested. A data prefetching strategy has to consider many issues in order to mask latency efficiently. Apart from predicting future accesses accurately, it must be also poised to bring that data to the cache in time. There are many strategies for data prediction. Some use recent history of data accesses from which patterns are recognized and are analyzed in [20] and [21]. Others [22] use the compiler and user-provided hints, or examine the loop’s behavior, or even running a helper thread ahead of actual execution of an application to predict cache misses [23]. Some data structures and operations are easy to predict and that fact can be exploited by an efficient prefetcher to drastically reduce the number of cache misses and therefore to improve the application’s performance and energy consumption. For example, it would be beneficial to traverse an array sequentially while searching for an element than randomly access distant elements, as is the case in binary search as the next element is easily predicted and therefore can be fetched in time for the access. For relatively small number of elements that can yield better results than binary search although the later has lower complexity. This is another aspect of the hardware that the DDTR Methodology was not designed to take into consideration but could skew some results.

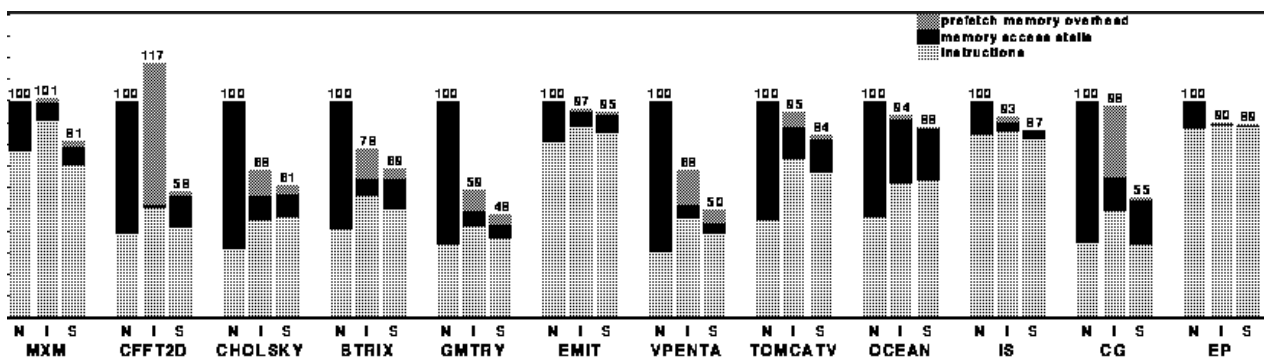


Figure 2.7: Example of overall performance comparison between indiscriminate and selective prefetching algorithms (N = no prefetching, I= indiscriminate prefetching and S = selective prefetching) for some benchmarks. The prefetch memory overhead, memory stalls and instruction cost are shown.

Based on the aforementioned observations, we conclude that the use of data accesses to evaluate the performance of a data structure can yield misleading results. Therefore, in this work instead of data accesses, we utilize “real” memory accesses and present data structure implementations that exploit hardware specifications in order to improve the performance of the application under optimization.

To sum things up, the DDTR methodology presented in Chapter 1 supports only a limited number of simple data structures (mainly list and array based). More complex data structures, like trees, were never evaluated, but still these data structures are fairly common in many modern applications. Secondly, the DDTR approach was developed under the assumption that data structure implementation constraints are related only to the application dynamic behavior. Therefore it contains only platform independent data structure implementations. They are built at high levels of abstraction and consider a plane memory hierarchy that is almost never the case. As a result, memory specifications, such as caches are ignored. The evaluation of the results is done by benchmarks on a x86 processor and not on different platforms to get a taste of potential differences in performance according to the hardware. Finally, the profiling component of the DDTR methodology logs only execution time and memory utilization. Other important metrics such as throughput and latency of operations are not available.

CHAPTER 3

Methodology for creating cache-friendly tree data structures

There are many ways to optimize the data structures used by an application and it is a common problem that it is extensively studied in literature such as in [1], [24], [25]. However these works mainly focus on static data allocation techniques that happen during compile-time. For example one could use a scratchpad, a piece of fast memory that resembles a cache, with the difference that the programmer has complete control over which elements are stored there. So if we wish for specific data structures to remain in the cache we can keep them in the scratchpad by using special instructions during the compilation of the program. The DDTR Methodology has some serious limitations (Table 3.1) and in this work we try to counter some of them. We aim at dynamic applications, where the access pattern may change at runtime along with the application’s behavior.

	Existing DDTR	New DDTR
Library of Data Structures	list, arrays	list, arrays, radix trees
Platform-awareness	Platform-independent implementations	Cache-conscious implementations
Metrics	Exec. time, memory	Exec. time, memory, throughput, latency
Evaluation	x86	ARM-based, Myriad embedded systems



Table 3.1: Existing DDTR Methodology characteristics and limitations

The first step is to enrich the DDTR with new data structures. Instead of only arrays and lists we evaluate the performance for some tree implementations that are fairly common or have good performance. Several radix tree data structures have been proposed to optimize the performance or the energy requirements of different applications. Tries are fast tree-based data structures for managing strings in-memory, but are space – intensive. They word “trie” comes from “retrieval”. Alternatively, they are named radix trees. There are many implementations of tries and are preferred for their reasonable worst case performance and are valuable for applications, such as data mining, dictionary and text processing, pattern matching and even compression. Below is an overview of the tree data structures that are added to our library. The three data structures that are described are:

1. **HAT-trie**. Based on the burst-trie, this is a fast and cache friendly combination of tree and hash table.
2. **Ternary tree**. A classic tree for strings.
3. **Patricia trie**. A specialized tree used for IP addresses.

After the aforementioned trees are explained, we attempt to give the less cache friendly implementations of ternary and patricia trie some cache conscious characteristics in order to evaluate how those characteristics impacted the different metrics that we use, as well as explain some of the problems that may occur.

3.1 The HAT-trie data structure

The HAT-trie data structure is introduced in [26]. Most tries need large amounts of memory space and a lot of effort has been put into limiting their size. The most successful procedure for reducing the size of a trie structure and achieve satisfactory has been the burst-trie. The burst-trie is an in-memory string data structure that can reduce the number of nodes maintained in the trie by as much as 80% with little to no cost as far as access speed is concerned. This is managed by selectively collapsing chains of trie nodes into small buckets of strings that share a common prefix. When a bucket becomes big enough, it bursts into smaller buckets parented by a new trie-node. The bucket is usually implemented as linked lists with move-to-front on access [27].

Although fast, a burst-trie is not cache-conscious. Like many other data structures it assumes equal cost of every memory access and is efficient in this setting, but that is not always the case. Depending on whether a memory access is served by the cache or by the main memory, the cost varies greatly. Although the trie is a memory intensive data structure, each node tends to be small in size and that gives it some cache-conscious properties, because it improves the probability that frequently accessed trie paths will reside within the cache. The most frequently used trie nodes, such as the top levels of the tree, are much more likely to stay in cache because they have a high probability (the closer to the root of the tree, the higher) to be accessed. The problem with the burst-trie, however, is that the buckets are represented as linked lists, which are inherently cache inefficient. To understand why that happens we must examine the way a linked list is created. In most cases, each node is allocated separately and linked with pointers, and that causes subsequent nodes to not be in subsequent memory addresses. If they were, it would be easier to predict the next access as a list tends to be accessed sequentially and two nodes in the same block would both result in the same cache block. This pointer chasing problem hinders the effectiveness of hardware prefetchers that attempt to reduce the number of cache misses by preloading data into the cache.

The HAT-trie is basically a cache-conscious variant of the burst-trie that takes advantage of the cache hierarchy used in modern processors. To address the bottlenecks of the burst-trie, the HAT-trie must reduce significantly the cost of tree traversal – the number of tree nodes that are created – and more importantly, the cost of searching a bucket. To achieve this, it has to swap the linked lists with a new, more suitable data structure, such as large cache-conscious arrays. Therefore the buckets are now structured as cache-conscious hash tables. The advantage of using hashing instead of a simple array, is that the buckets can

scale more efficiently, further reducing the number of trie nodes that are needed (and therefore the number of memory accesses). This is the basis of the HAT-trie and their most important difference with the burst-trie.

The HAT-trie is built and searched in a top-down manner. The query for our needs is considered a string. The pointer that corresponds to the first character of the query is followed to the appropriate trie node. Traversing the tree “consumes” the current character from the query. For the following character this procedure will repeat until either a bucket is accessed, or the whole query is consumed. In the first case, the rest of query is hashed (based on length for example) by a fast bitwise hash function and searched in the bucket in the correct slot. The slot contains strings and each string is examined until the first mismatch and then the next string is checked (and so on) until either the target string is found or not. The bucket does not contain duplicates. If the whole query is consumed and a bucket has not been reached, then each trie node has a Boolean variable that serves as an end-of-string flag and indicates if the search is successful or not.

An insertion is much like a common search. If the element is already present nothing happens. Otherwise, the query is consumed until a bucket is found where it is inserted in a similar manner to the hash table insertion (hashing the rest of the string and inserting it in

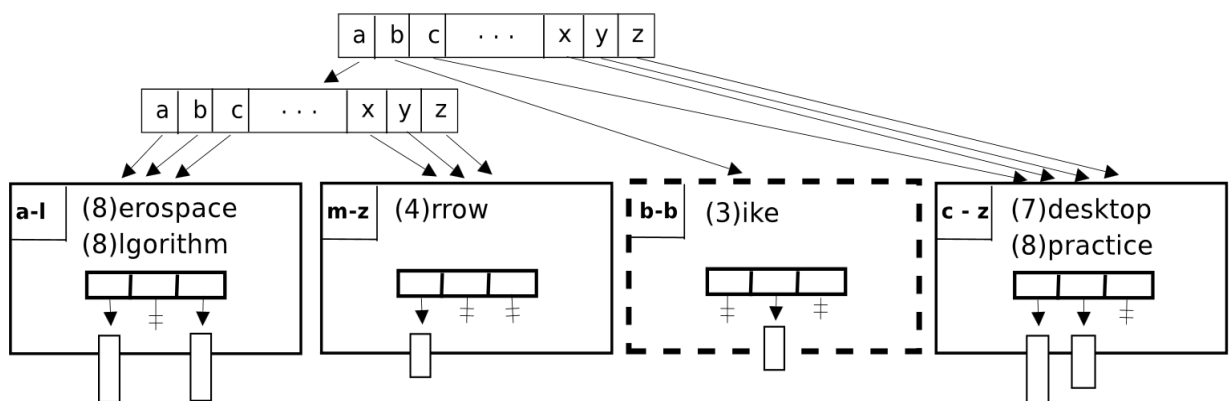


Figure 3.2: A typical HAT-trie. The array of characters is the trie node and the boxes at the last level represent the buckets. If the box is dashed then it is a pure bucket.

the correct section of the hash table data structure). When a bucket is full, – contains a specific number of strings or has surpassed a typical size – it splits. Then a new parent node is created and the strings of the old bucket are distributed in the new buckets according to their lead character that is removed. There are two types of buckets, hybrid and pure. The hybrid buckets are buckets indexed by many characters in the trie node, whereas the pure buckets are only indexed by one (the difference is more obvious in Figure 3.2). The buckets are split by using B-trie splitting. On split, a pure bucket is converted into a hybrid by creating a new parent node with all pointers assigned to it. The old node is pushed up as a grandparent. To split a hybrid node, first a good split point is found (that achieves as equal distribution as possible). The process is to count the number of strings. Then, based on that information, a character is selected as a good split-point.

The bucket consists of $n+1$ word-length pointers which are either empty or point to their respective slot entries. The first pointer is reserved for some additional information such as the character range, the bucket type and the number of strings. Each pointer points to a “slot” that is an array of strings. Using larger buckets can save up some space (by reducing the number of nodes) but the access time for each bucket is increased as there are more elements to be checked per slot. However, large buckets are also more probable to remain in cache and if a bucket is cached then the search is very cache efficient as the strings are processed sequentially. Many small buckets (and therefore many nodes) can lead similar strings to be located in different buckets and destroy cache performance. The performance of HAT-trie in benchmarks [26] is up to 80% faster than a regular burst-trie, while simultaneously can reduce the space consumption by as much as 70%.

3.2 The Ternary Tree

The ternary tree [28] is a tree data structure (trie) that is similar to the classic binary tree. The main difference is that each node has at most three children. So each node contains a character and three pointers (greater, equal, less). The ternary tree (or ternary search tree or sometimes prefix tree) has the ability of incremental string search. The query (string to search) is consumed one character at a time when the character that is stored to the node is the same as the leading character at that time of the query under examination. If it is not then the leading character is not consumed. In any case, a recursive procedure occurs, where we follow the appropriate pointer according to whether the character of the query is greater, less or equal to the character of the node. Also, each node can be either marked or unmarked, based on the existence of an inserted string that ends at that node. If we reach a null pointer and the query is consumed, then the query is not inserted into the ternary tree. Likewise, if the query is consumed but we have reached a node that is unmarked, then the string is not found.

The ternary tree has some advantages over other data structures thanks to their simplicity and good average-case running time. All the common operation, such as lookup, insertion and delete cost $O(\log n)$ in the average case and $O(n)$ in the worst case (if a chain is formed instead of a tree). Also, they tend to use less space for storage at the cost of speed. It is slower than the prefix tree, but better suited for large sets of data thanks to their space efficiency. Common applications for the ternary trees include spell-checking, near neighbor checking (similarity search) and auto-completion of text, some of which are not possible with other data structures (such as hash tables). Figure 3.3 shows the difference in the structure of a ternary and a binary tree for the same vocabulary. The binary tree stores the whole string inside the node therefore its size can be considerably bigger even though there are only two children per node. So, there may be fewer nodes overall, but each node could be considerably larger and there is no data reuse (in the case of common prefixes).

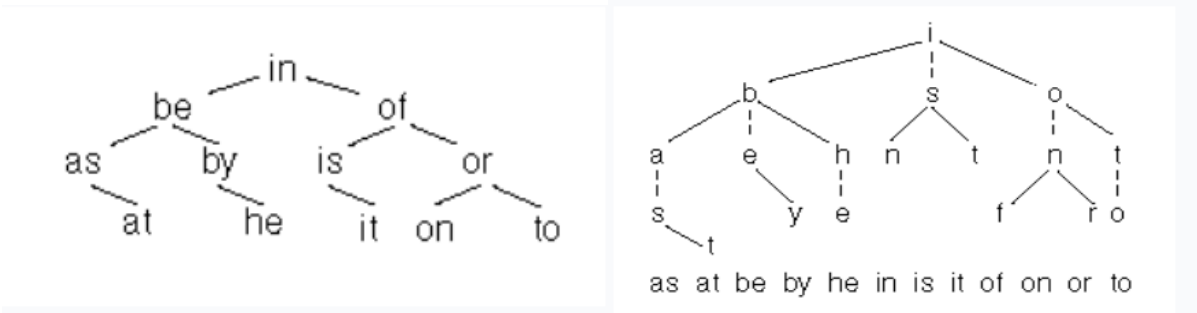


Figure 3.3: Binary vs ternary tree for the same vocabulary

3.3 Patricia trie

The Patricia trie is a combination of radix and crit-bit tree. Contrary to the Ternary's tree nodes, Patricia nodes store a complete string, alongside an integer that marks a specific bit of the string. As a result, instead of examining the whole string, only the critical bit is examined. Insertions are more complex than the ternary tree, since the crit-bit must be examined before inserting the node to the correct position in the tree. The trie is more complex as well: there is a node that serves as a default entry at the top of the tree that is not modified. Also, no null pointers exist and every pointer must point to a node.

The basic Patricia trie implementation that we used can be found at [29]. This Patricia trie was originally designed for IP addresses. The procedure of searching for a specific string, like all trees, begins from the root (Figure 3.4). The red bits (crit-bits) are checked and the red lines are followed. At each node, the crit-bit is compared. If it is the same, then the string-to-search is compared with the string that is stored in the node. If it is different, the action depends on the value of the crit-bit. If the bit is set, then the right pointer is followed, otherwise the left one is followed. This procedure repeats recursively until either we arrive at a node with a crit bit number smaller or equal to the previous node, or a match is found.

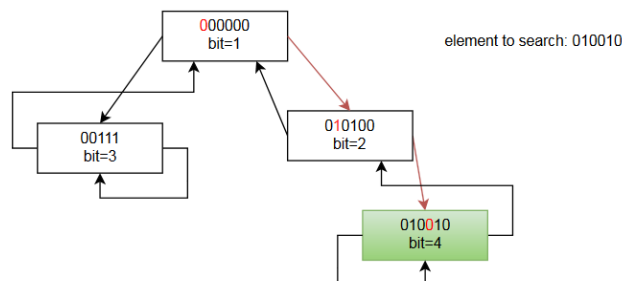


Figure 3.4: searching for a specific string in the Patricia trie

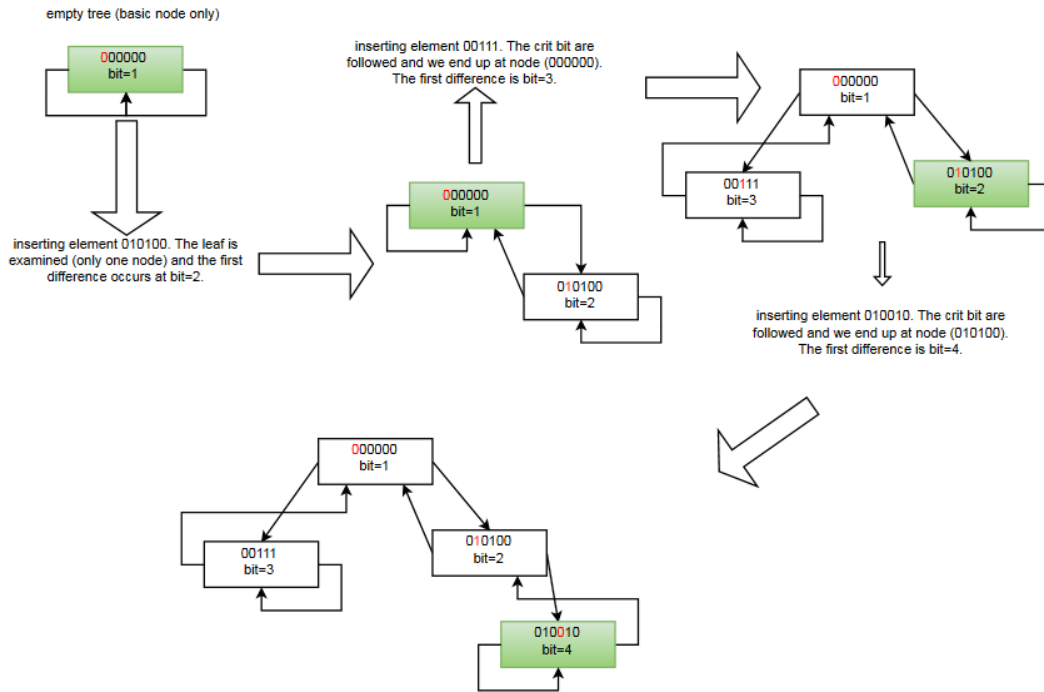


Figure 3.5: the Patricia trie before and after the subsequent insertion of nodes

The subsequent insertions and the trie structure that lead to the example of Figure 3.4 is shown in Figure 3.5. To insert a new element that is not present in the data structure, we begin from the root and compare the crit-bits. Again if it is set we follow the right pointer (else the left) until we reach a leaf. At that point we find the first bit between the leaf's string and the target string that differs. This number will decide how close to the top the new node will be inserted.

For node removal we have to search for the target node while at the same time keeping track of the parent node and the grandparent. We point the grandparent to the correct nodes and then delete the target's data and copy in its parent's data but not the bit value. The aforementioned procedures can be better understood through the illustration of

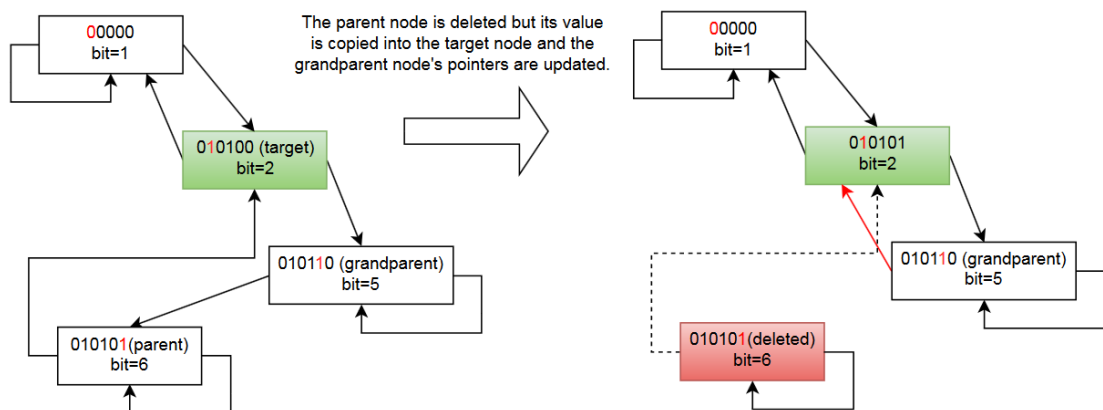


Figure 3.6: Deletion of a node in a Patricia trie

Figure 3.6.

3.4 Cache-Friendly tree implementations

3.4.1 Underlying theory

The problem with the tree data structures in general, is that they not particularly cache friendly. This is also true for most pointer-based data structures. Some modifications however are poised to give them some good cache properties. An example is the HAT-trie that uses hash tables and array slots in the buckets to improve the cache behavior, instead of using linked lists. Programmers typically allocate a data structure's elements with little concern for memory hierarchy. Often the resulting layout interacts poorly with the program's data access patterns, causing unnecessary cache misses and reducing performance. We created some cache-friendly implementations for the Ternary and Patricia trie data structures to add in the DDTR library extension. They are based in the clustering and compression techniques that have been previously described in Chapter 2. The compression technique enables elements to be clustered to the same cache block by separating the node fields (i.e. structure's data) in those which are accessed frequently and those which are not. Then we cluster these reduced nodes together in the same cache line.

	Existing DDTR	New DDTR
Library of Data Structures	list, arrays	list, arrays, radix trees
Platform-awareness	Platform-independent implementations	Cache-conscious implementations
Metrics	Exec. time, memory	Exec. time, memory, throughput, latency
Evaluation	x86	ARM-based, Myriad embedded systems



Based on this idea, in our cache conscious implementations, we cluster adjacent tree nodes in the same cache line. Thus when a node is fetched into the cache, more nodes are fetched that are probable to be accessed in the near future. Even if the adjacent nodes are generally big enough to not fit in the same cache line, we can still get great improvements just by allocating them sequentially in memory. Modern prefetchers are smart enough to predict such behavior and in case of a cache miss, to fetch instead of the target memory block, the adjacent blocks as well.

The most important problem that we try to counter is the fact that separate allocation of nodes causes the whole tree to be located in random memory addresses and that pose problems to the operating system that tries to predict what will be accessed next. Therefore, to maximize locality, adjacent tree nodes (nodes that are connected with a relation child-parent) are allocated in adjacent positions in an array, in an effort to reduce cache misses.

Therefore we need to use a custom memory allocator, similar to `malloc`, that will perform local clustering and replace the operating system's default allocator. The result of this allocator (*custom_malloc*) is that the tree nodes will be not randomly distributed in memory, but concentrated in small pools that are basically arrays of nodes. The pseudocode for the custom allocator is presented in Figure 3.7.

```

struct Node * custom_malloc (list *mylist){
    if (current_slot < MALLOC_SIZE) {
        return &(alloc_array[current_slot++]);
    }
    else {
        current_slot = 0;
        alloc_array = malloc( sizeof (struct Node) * MALLOC_SIZE);
        push(mylist, &(alloc_array[0]), MALLOC_SIZE);
        return &(alloc_array[current_slot++]);
    }
}

```

Figure 3.7: Custom allocator

This is the C function that implements the custom allocator, named `custom_malloc`. The return value of this function is a pointer to the desired node of the target data structure (*struct Node*). The parameter named `MALLOC_SIZE` represents the number of nodes that can fit in a pool. As explained before, there is an array of nodes and this parameter basically decides how big this array is going to be. The importance of this value and its effect is discussed in detail below. The data structure *mylist* is a singly linked list that is used to keep track of the node arrays that we have allocated for reasons that will be soon explained. So, when the programmer desires to allocate a node and invokes this function one of the following will happen:

- I. If the previously used node array is not full yet – from the `MALLOC_SIZE` nodes that fit in the array only some are used – the request will be satisfied by this array. The parameter *current_slot* is a global variable that marks how many array cells are occupied by actual data. If this number is smaller than `MALLOC_SIZE` then there is still room in the array for one more node. As the nodes are pre-allocated in the array, a pointer to the appropriate array cell is returned and the *current_slot* parameter is increased in order to have the correct value for the next invocation of the function.
- II. The other possibility is that the aforementioned array is already filled therefore we cannot get any more nodes. In this case, we need to allocate a new array with `MALLOC_SIZE` number of nodes. The allocation is done in the ordinary way by invoking the default system allocator (*malloc*). After the new array is allocated we need to put it in the list (*mylist*) through a custom push function with the other arrays that are used to accommodate the other

nodes of the data structure that have been inserted so far. The *current_slot* variable is set equal to zero as no nodes from the new array are used. Finally, the function returns the pointer to the first element of the array while increasing the *current_slot* to indicate that one node is used.

The custom singly linked list (*mylist*) that is used to keep a record of the arrays that are used is a normal list. Its size depends on the size of the data structure and on *MALLOC_SIZE*. As the later indicates how many nodes are contained in each array, the bigger this value the less arrays will be needed to supply the nodes of the data structures. Similarly, the bigger the data structure the more arrays will be needed and as a result, the longer the linked list will be. Increasing the value of *MALLOC_SIZE* can lead to smaller lists and less system calls for allocation of the arrays, therefore faster to traverse, but can waste memory if some nodes are left unused. The bigger the nodes the most obvious is the problem. For a relatively big data structure the wasted memory is negligible. The

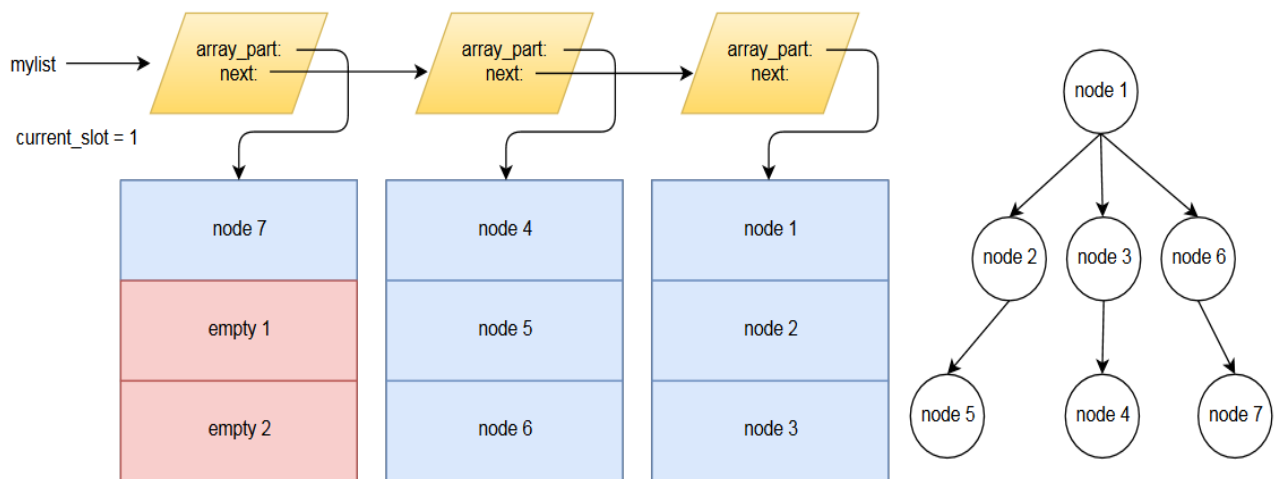


Figure 3.8: A tree (right) and its custom allocation pattern through *custom_malloc*

programmer can use the value that better suits the application's needs. For our experiments a typical value is 50 nodes per array.

In Figure 3.8, a tree and its allocation pattern through *custom_malloc* is shown. The *mylist* points to a SLL (Singly Linked List) with arrays as elements. Each array contains 3 nodes, so in this case we assume *MALLOC_SIZE*=3. The nodes of the tree are the elements of the array. After the first 3 nodes are allocated and the array is exhausted, a new SLL node with an array with 3 more nodes is inserted in the list and so on. At the stage depicted here, 7 nodes are used and therefore the last array has 2 empty nodes – nodes that have yet to be used in the tree (filled with red).

One important question to answer is how this new configuration affects the tree's cache behavior in theory. In the case of a simple ordinary tree, such as a binary tree, after visiting a node there is a 50% chance that each of the two children will be visited next – the amortized chance of checking none of the children – because the node contains the desired

item – is negligible as in every search that will happen at most once, when the desired node is found. At the same time, it is highly probable that these children will be nodes allocated in a completely different time from their parent and as a result, their address in the memory space will be completely random and different from his, assuming that the default allocator of the system is used. If the cache is relatively small or congested with many elements, it is highly unlikely that the children will be already in cache therefore each time a pointer is followed a cache miss is guaranteed. In the best-case scenario the processor can execute some other independent instructions to mask the latency induced by the cache miss. But most of the times, while a tree is traversed in search for a specific node, only one field of the node is examined with a simple instruction (like an integer comparison) and then the appropriate path is selected. So if the node is not in cache the processor has to stall.

By grouping nodes in an array we can have many nodes in the same cache line or, if that is not possible, at least to adjacent cache lines. That can help the processor predict the access pattern. In the ideal case let us assume that the parent and both children can fit in a cache line and are allocated through a custom allocator scheme, like the one described above, occupying adjacent positions in the array. Figure 3.9 offers a visual representation of this case. The nodes are separated in groups of 3 and assigned a specific color and number (group1 are the 3 red nodes for example). The 3 nodes with the same color have a parent-children relationship and more specifically parent – left child – right child and they are allocated in that way. We assume that each array of the custom allocator can accommodate 6 elements so 2 groups at once. Therefore there are 3 arrays – 2 of them consist of the groups 1&3 and 4&5 respectively. The other one has only the group 2 and 3 empty nodes that are yet to be used. For simplicity we also assume that the cache line is sufficiently large to contain 3 nodes so that the same colors are in the same cache line.

With the above optimistic assumptions as a given it is possible to examine this structure’s behavior from a theoretical point of view. In the worst case there will be one cache miss every two nodes because the parent and the two children are in the same cache line and it is not possible to get a miss when going from a node of a particular color to

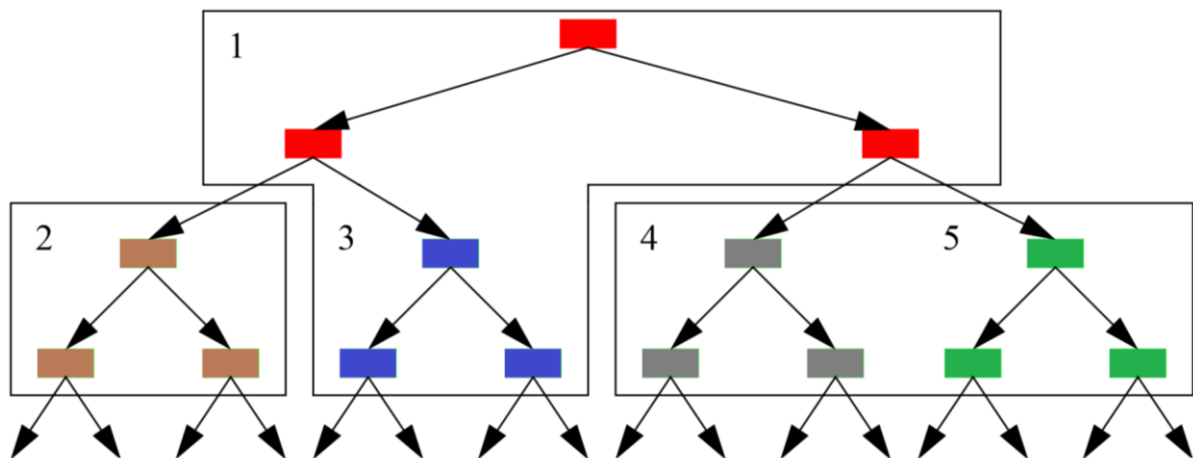


Figure 3.9: ideal allocation for a simple binary tree

another one of the same color. So, in place of two misses we get only one. That is a 50% miss reduction assuming the above configuration, which is a significant improvement.

3.4.2 Challenges and solutions

The allocation strategy that is described so far has three major drawbacks that are easily revealed once it is carefully examined and must be considered:

- I. The most important problem is how to ensure such an ideal configuration as the one depicted in Figure 3.9. If we serve the allocations request from nodes pre-allocated as part of an array, we cannot guarantee that the child and the parent will be in adjacent cells. The insertion requests are random and if we allocate a node at a given time, the next allocated node is most often than not a distant node without ties to the previous one. To counter this issue, one could argue that we should leave the adjacent nodes empty once we allocate a parent so that there will be space for its children once they are allocated. But this approach gives rise to new problems. When the time to allocate a child comes, how will we determine in what position of the SLL is the array that contains the parent? This requires additional bookkeeping information, such as one additional pointer per node to the SLL element that this node is stored. But adding pointers to the nodes will increase their size and reduce the spatial locality as fewer nodes will be feasible to be packed together in the same cache line. Another option is to search, at the insertion of the node, the SLL to find the parent instead of keeping a pointer from the parent to the list. That way we get rid of the additional memory but we introduce a performance overhead as the list can be considerably big and will be searched at every insertion. The additional time can quickly add up and destroy any benefit from the improved cache behavior.
- II. A tree data structure is a dynamic data structure and its morphology (the relations between the nodes) is bound to constantly change as new elements are added and removed. Even the perfect configuration of Figure 3.9 cannot be maintained as when new nodes are added, because after the new insertions new parent – children relations are created that no longer correspond to adjacent cells in the array. This problem is the memory fragmentation and is inherent to every dynamic data structures.
- III. No matter what allocation strategy is followed, the problem that persists is how to deal with deletion operations. If a node is an element of an array what happens when it is deleted? An array cell cannot be deleted; only have its value erased. Having an array of pointers instead of an array of nodes introduce additional overhead and serves little purpose as the main idea is to

have neighboring nodes in adjacent array cells. The cell that represents a deleted node can be reused in the next insertion request, but that brings forth the issue of the additional bookkeeping of case I. A “free list” will be required with pointers to the array cells that are empty. This is not desirable because it will increase the program’s memory footprint for keeping the additional list.

In order to solve these problems, we need to understand it is inevitable that the sequence of the nodes in a data structure and the relations between them are going to change as the application is executed. The memory fragmentation, that occurs when new nodes are inserted to the tree or existing nodes are deleted, cannot be avoided. With this in mind, a simple solution is to initiate a complete reorganization of the way the nodes of the tree are stored in memory [30].

The reorganization procedure works as following. When the decision to reorganize the data structure is taken, a new array of nodes is allocated that is as big as the tree itself. That means that the tree is actually represented as an array in a big chunk of continuous memory. The tree is then traversed and the nodes are stored in the desired way to achieve an ideal cache behavior. The parent and their children are put in adjacent cells. Then the same process is followed for each child of these children recursively. This way we can achieve the allocation scheme of Figure 3.9. The pseudocode and the resulting tree are presented in Figure 3.10. The number inside the tree nodes correspond to their array positions. As it is a binary tree (for simplicity) each triad (different color) has the parent and the children in adjacent nodes so that when the parent is fetched to the cache, the children are too. The *current_index* variable is a global parameter that is the offset of the next usable cell in the

```

function reorganize ( current_node):
    node_array[current_index++] = copy(current_node)
    for each child of current_node do:
        for each child_child of child do:
            reorganize(child_child)

```

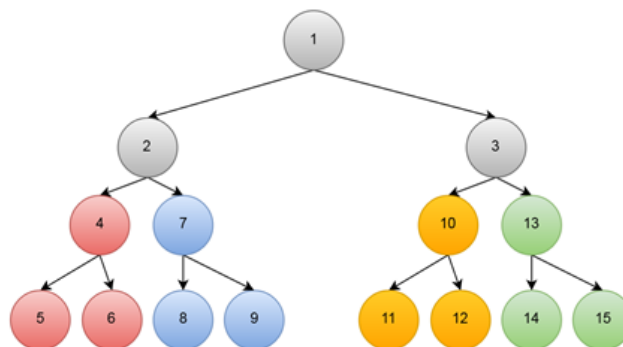


Figure 3.10: The reorganization function and its result

new array.

We also need to update the pointers. We copy the old node to the new array but we cannot copy the pointers, as they point to arrays in *mylist* structure instead of the new array that we allocated when the reorganization started. There are many ways to do the pointer update. A convenient way is to add an integer field in the node that shows the offset in the array of each node, but this increases the node size and can limit the advantages. We can update the pointers recursively as we know that the parent and the children will be in adjacent positions. The pseudocode is shown in Figure 3.11. The general idea is that we copy the current node to the correct new array position and then we do the same for its children, updating the pointers in the meantime. Then we call the function recursively for each child's child. The return value of the function is the array cell that the node resides.

Through this reorganization process, the functionality of the SLL data structure that we use to keep the arrays with the tree's nodes is revealed. Once the whole tree is copied in the new big array we need to deallocate its former memory to reduce the additional memory, as there is no reason to keep the tree and an outdated copy of it. The SLL keeps the memory chunks that are used for the tree so all there is to do is deallocate those chunks (arrays). Then the SLL is empty and the new big array that contains the whole reorganized tree is the sole element kept in it. As we add new nodes, new smaller arrays will be inserted in the list as explained in Figures 3.8 and 3.7.

```
function fix_pointers (node)
    retval = current_index
    parent = copy(node, node_array[current_index++])
    if (parent->right) {
        copy(parent->right, node_array[current_index])
        parent->right = &(node_array[current_index++])
    }
    if (parent->left) {
        ...
    }
    parent->right->right = fix_pointers(parent->right->right)
    parent->right->left = fix_pointers(parent->right->left)
    parent->left->right = fix_pointers(parent->left->right)
    parent->left->left = fix_pointers(parent->left->left)
return retval
```

Figure 3.11: Pseudocode for the reorganization with pointer fixing

The tree reorganization process is further illustrated with a simple example in Figure 3.12. Before the reorganization, in the list there is an array with 4 elements, named memory chunk #1 (leftover from a previous reorganization) and 2 more nodes with 1 element each (here we suppose that it is an array of one element – not so useful for cache friendly behavior but necessary to keep the example easy to follow). When the decision that a reorganization is needed a new array is allocated with size equal to the current number of nodes and then it is filled in the aforementioned manner. Finally the previous address-list's elements are deallocated and the new memory chunk is inserted.

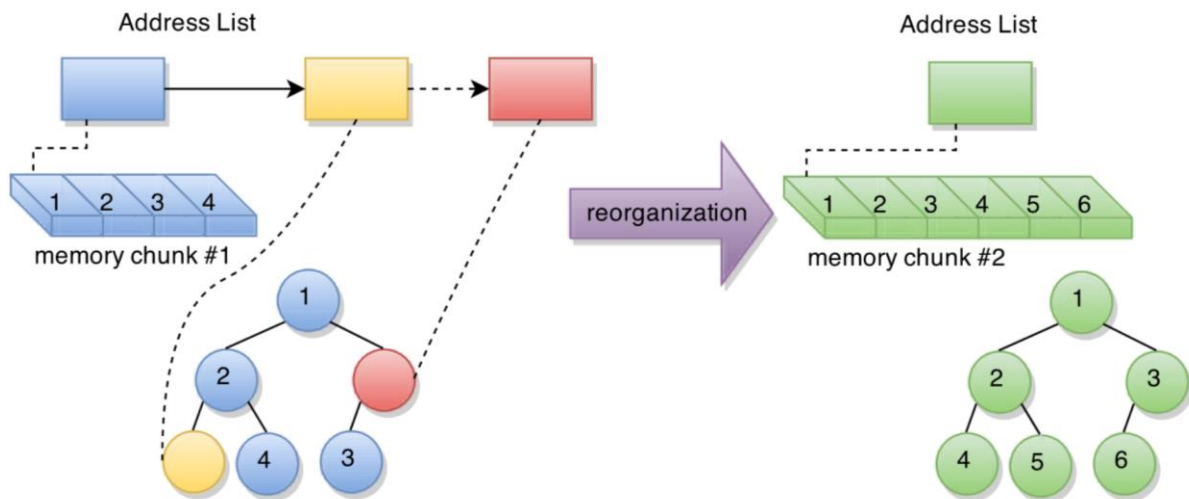


Figure 3.12: list before and after the reorganization

Although the reorganization can greatly improve the performance of an application based on trees, there are some obvious disadvantages that must be carefully considered. The most important is that during the reorganization the memory usage for the tree data structure is doubled and this can be observed as spikes in the application's memory footprint. Depending on the size of the tree, the time this peak in memory is going to last can vary as the whole tree needs to be copied. For some applications this may be negligible, but it can prove problematic especially if the application has tight memory requirements and no flexibility. Of course there are ways to counter this problem. For example the tree cache-friendly reorganization could be done incrementally, only a piece of the tree at a time. This will reduce the memory overhead by a large margin. But in the current work we have not focused on such techniques. Another disadvantage is that during this process the application basically “freezes” if it is executed on a single core machine as this core must execute the reorganization function and cannot serve any requests. However, in a multicore environment some of the functionality can be kept. A separate core can start copying the tree while the others continue to serve some requests (such as search) by using the old copy as the new tree is prepared. For most of the associated problems there are possible roundabouts to counter them, at least partially, at a cost of extra complexity of the related data structures and code. This performance – memory tradeoff can be evaluated by the programmer, who will ultimately decide if it is worthy for the specific application.


With the disadvantages in mind, the last important decision that we have to take is how often to perform the reorganization. Generally, we do not want it to happen too often because the cost of reorganizing the tree will surpass the performance gain from the better cache behavior. The reorganization should happen once the tree has been modified enough times. A modification is anything that can destroy the ideal structure of a reorganized tree such as insertions and deletions. If the tree has the correct properties (parent and children in adjacent memory addresses) as more and more modifications occur those properties will be less and less common to every part of the tree. A few modifications in a large tree, like 1000 insertions when the tree has millions nodes are unlikely to affect the performance greatly. So, the decision must be taken when the number of insertions/deletions reaches a specific coefficient with reference to the tree size. In our experiments we put that threshold at 1.0. That means we initiate a reorganization once the tree has doubled in size since the last time it was reorganized.

CHAPTER 4

Experimental setup and results

Once the optimizations described in Chapter 3 are implemented, some experimentation is needed to evaluate the actual results in real applications and corroborate them with the theoretical results. The evaluation of the extended DDTR is done by using two modern embedded chips and a set of real world benchmarks in addition to some synthetic ones. In the previous DDTR only one processor is used. The data structures that are used in the experiments are the ternary tree, the patricia trie and the HAT-trie and have

	Existing DDTR	New DDTR
Library of Data Structures	list, arrays	list, arrays, radix trees
Platform-awareness	Platform-independent implementations	Cache-conscious implementations
Metrics	Exec. time, memory	Exec. time, memory, throughput, latency
Evaluation	x86	ARM-based, Myriad embedded systems



been thoroughly described in Chapter 3.

4.1 Platforms and benchmarks

The goal is to evaluate the DDTR methodology with various systems that have completely different memory hierarchy. Depending on the processor, each system can display completely unique characteristics. For example some systems have cache memory while others do not. The existence or not of a cache could affect the decision for a specific data structure or make the optimization that were discussed earlier obsolete. This is the reason that we need to take into account the platform and move from a platform independent methodology to a platform dependent.

The first platform that was used is the Freescale i.MX6 [31] which is a 4-core ARM-based embedded chip. It contains two levels of cache memory and a 1GB DDR3 RAM. The second one is the Myriad chip designed by Movidius Ltd [32] and normally acts as a low-power co-processor in mobile devices, smartphones and wearable gadgets [33]. It integrates 8 VLIW cores (Very Long Instruction Word), which access a 1MB shared SRAM memory. In contrast with the Freescale, no cache exists between the cores and the memory and also the memory capacity is much lower.

The evaluation of the extended DDTR methodology was made through a variety of synthetic and real-world datasets. The synthetic benchmarks consist of some custom

testcases of 10M operations each. The operations are items (number strings) that need to be inserted, deleted or looked up in the tree data structures that are added to the DDTR library and want to examine. On the other hand, the real-world benchmarks are two sets that refer to IP addresses and dictionary entries. The IP dataset contains a portion of the IP addresses that made requests to servers for the 1998 World Cup and was taken from the Internet Traffic Archive [34]. It is composed of 3 million IP addresses, of which 4% are unique, so there are many duplicates. The dictionary datasets are taken from real dictionaries of various languages, utilized in the WinEdt text editor [35] and contains only unique string entries (a dictionary does not have duplicates), but in some experiments we have duplicated some entries. In the Myriad experiments, due to low memory – only 1MB – we need to restrain the size of the tree and therefore only a small part of the benchmark is executed. The metrics that are used to evaluate each case are the throughput and the memory footprint. The throughput is defined as the number of operations per second. Finally, some results are presented that relate to the memory overhead and the performance of the cache conscious implementations.

4.2 Experimental results – synthetic datasets

In this section we present and analyze the experimental results of the application of the extended DDTR methodology in various synthetic benchmarks. We performed two experiments using synthetic benchmarks in the Freescale board, which are presented in Figures 4.1 and 4.2. In the first one, the updates are 10% of the total operations, while in the

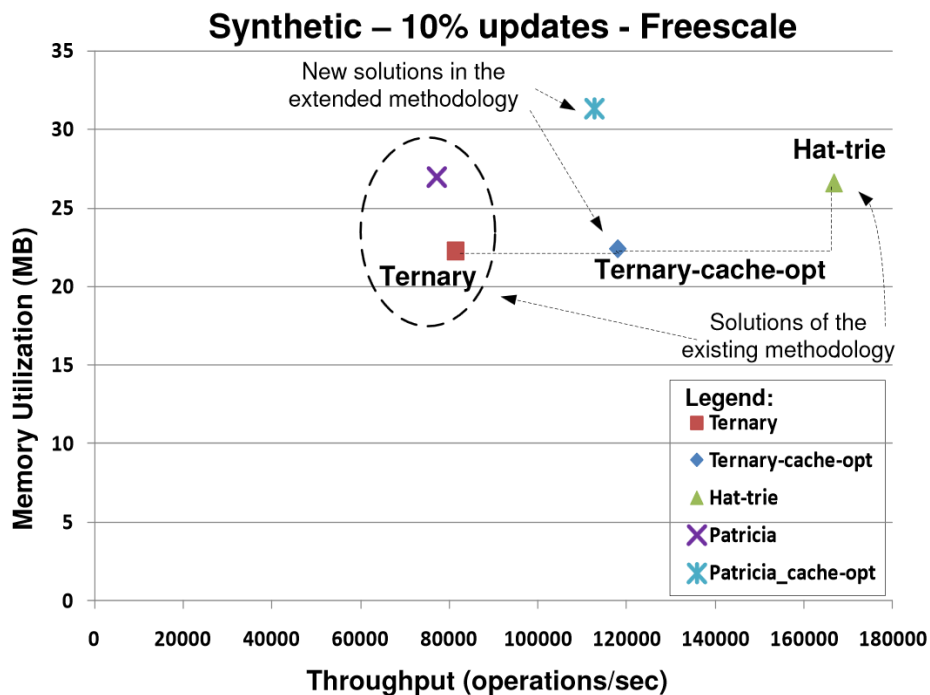


Figure 4.1: Throughput vs memory utilization of the synthetic benchmark with 10% updates in the Freescale board

second one they are 80%. As an update we define an insertion or a deletion. When an insertion is performed the algorithm tries to find the element and if it is not stored in the structure, then it adds it. Similarly, when a deletion is performed the element must already exist before it is deleted.

In the first experiment in Figure 4.1, there are three optimal implementations overall, namely the ternary tree (labeled as Ternary), the cache conscious ternary tree (labeled as Ternary-cache-opt) and the HAT-trie (labeled as Hat-trie). Unsurprisingly, the highest throughput is achieved by the HAT-trie implementation – almost 35% higher than the Patricia trie. The reason for this phenomenon is easy to explain: the HAT-trie provides a much faster lookup in comparison to the other data structures that we implemented because it acts as a hash table. There are only a few updates so the majority of the operations are lookups and that gives the HAT-trie an edge. Also, the fact that this benchmark has a small percentage of updates favors the cache-optimized implementations as the optimized ternary tree that outperforms the traditional ones. Both the optimized version of the ternary tree and the Patricia trie have higher throughput than their non-optimized counterparts.

As far as memory is concerned for the results of Figure 4.1, we observe that the optimized ternary version has comparable memory footprint with the non-optimized one. One would expect the memory to be twice as the reorganization requires twice the memory (temporarily). However, depending on the size of the tree that the last reorganization happened, the extra memory overhead may not exceed the memory of the program when the execution finished. This is also dependent on the type of the tree. For the patricia trie for example, the overhead is considerably higher because each node contains the whole string

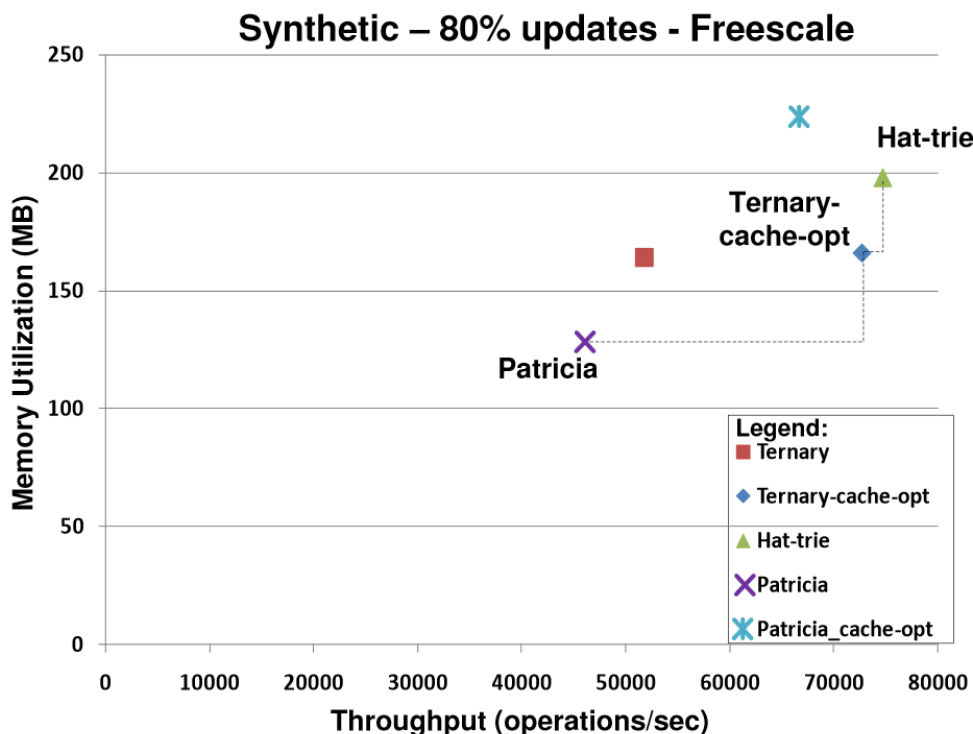


Figure 4.2: Throughput vs memory utilization of the synthetic benchmark with 80% updates in the Freescale board

and as a result, the existence of common substrings cannot reduce the number of nodes required. Additionally, the ternary implementations outperform the HAT-trie, because these implementations are built inherently with memory efficiency in mind. The patricia has however the best memory utilization.

In the second benchmark (Figure 4.2) most of the strings are unique and that causes the majority of the operations to be updates. As a consequence, the tree is bigger. The HAT-trie continues to have the best performance, although the gap with the optimized implementations from the previous experiment has closed. Compared to the original patricia implementation the HAT-trie's throughput is increased by 39%. The ternary optimized version can almost rival the HAT-trie while at the same time displaying better memory utilization. The main reason for this is that the advantage of the HAT-trie that is the fast lookups play a less important role here because many updates are required. The many updates will cause the buckets to burst more often (Chapter 3) and this leads to reduced throughput.

As far as memory is concerned, the same phenomenon is applied as in Figure 4.1 with the exception that the non-optimized Patricia implementation has the best memory footprint. Although in theory the ternary should have less memory than the Patricia, sometimes that is not the case, because the Ternary suffers if the strings does not have long common prefixes and / or are big enough as each character need a separate node with three pointers, while in the Patricia implementation the node has the whole string and two

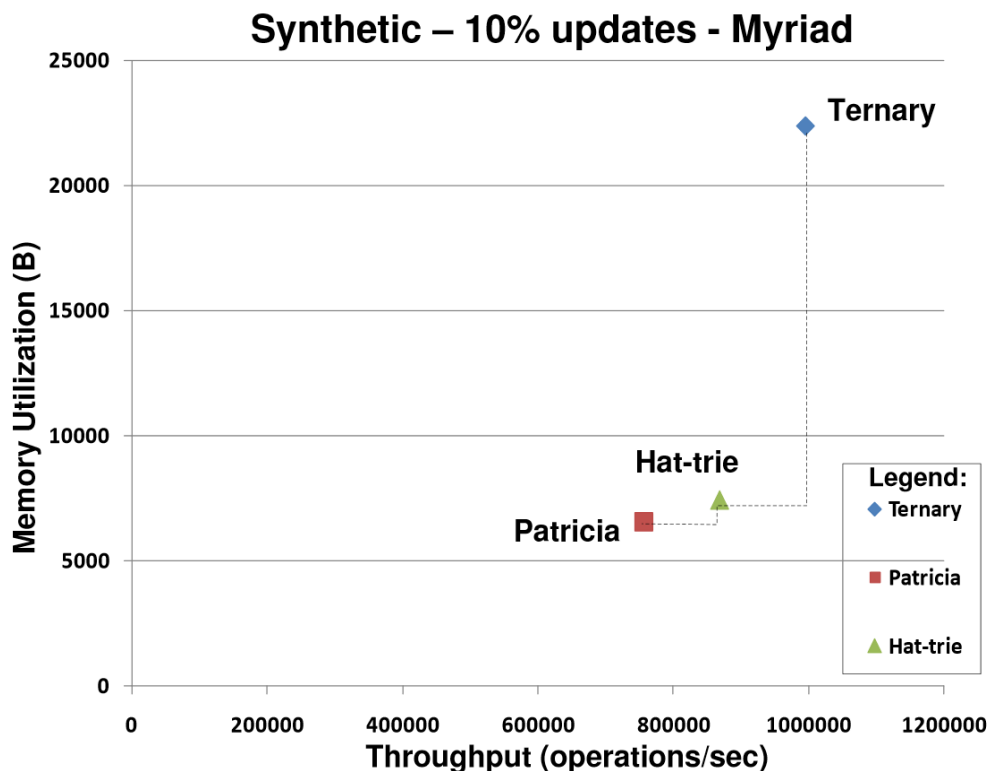


Figure 4.3: Throughput vs memory utilization of the synthetic benchmark with 10% updates in the Myriad board

pointers. So for the same string the space required by the Patricia node can be superior.

The corresponding evaluation of the synthetic benchmarks in the Myriad platform is presented in Figures 4.3 and 4.4. The cache conscious implementations are omitted in the Myriad experiments, due to the lack of cache memory in the Myriad chip. In the first experiment (Fig. 4.3), where the lookup operations are dominant, the ternary tree provides the highest throughput (22% higher in comparison with the Patricia implementation). However, the Patricia has the least memory requirements for reasons that have been explained previously, as the number of the strings is too small and as a result, the Ternary cannot take advantage of common prefixes (many strings are also unique). In the second experiment where the updates dominate and the trees have more nodes, the Ternary tree and the HAT-trie are the optimal implementations for throughput. An interesting observation is that the lack of cache memory in Myriad has caused the HAT-trie performance to fall behind the Ternary implementation, even though the HAT-trie basically behaves as a hash table (the bucket size is big compared to the 1MB memory of the platform, so there are basically no nodes). Therefore the results are a lot different compared to the Freescale board and the solutions that a designer would prefer can be different if the different platforms were not taken into account. This fact indicates the drawbacks of a platform-independent methodology, as the hardware and the specific input can make the same DDT behave differently.

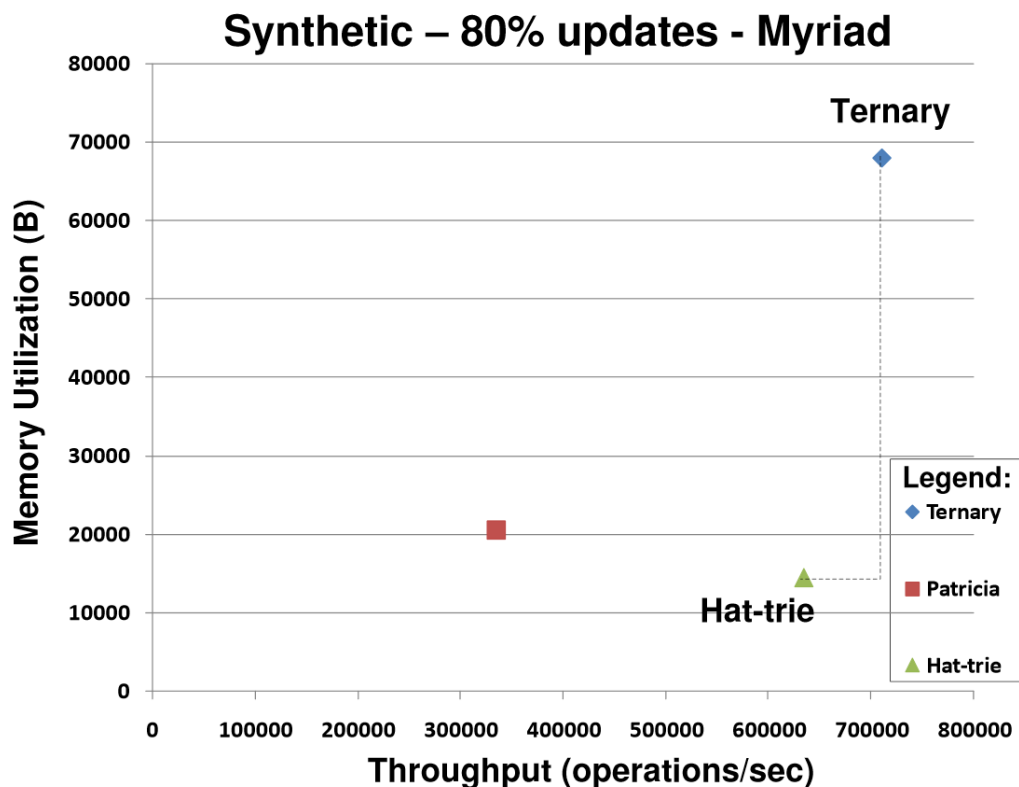


Figure 4.4: Throughput vs memory utilization of the synthetic benchmark with 80% updates in the Myriad board

4.3 Experimental results – real world datasets

4.3.1 IP dataset

The IP Dataset consists of 3 million IP addresses, in which 4% are unique and they represent the addresses that made requests during the World Cup 1998. A sole address usually does many requests to a site, hence the small percentage of unique addresses compared to the total requests. The procedure for each address is the following: first it is searched in the corresponding tree and, if it is not found, it is inserted. Each line has a number that corresponds to a client (real IP addresses are concealed to preserve users' anonymity).

The results on the Freescale chip are presented in Figure 4.5. The overwhelming percentage of lookup applications gives the HAT-trie a considerable edge over the rest of the implementations (30% in comparison with the patricia-cache-opt). We also observe that the optimized Ternary and Patricia versions lead to 12% and 17% better throughput over their non-optimized counterparts. The non-optimized and the optimized implementation have similar memory utilization as the tree reorganization does not happen very often, therefore the two versions' final memory consumption align. The Patricia implementations still has the lowest memory footprint.

With respect to the Myriad board, the corresponding results for the IP benchmark are shown in Figure 4.5. As in the synthetic benchmarks, the Ternary tree provides the highest

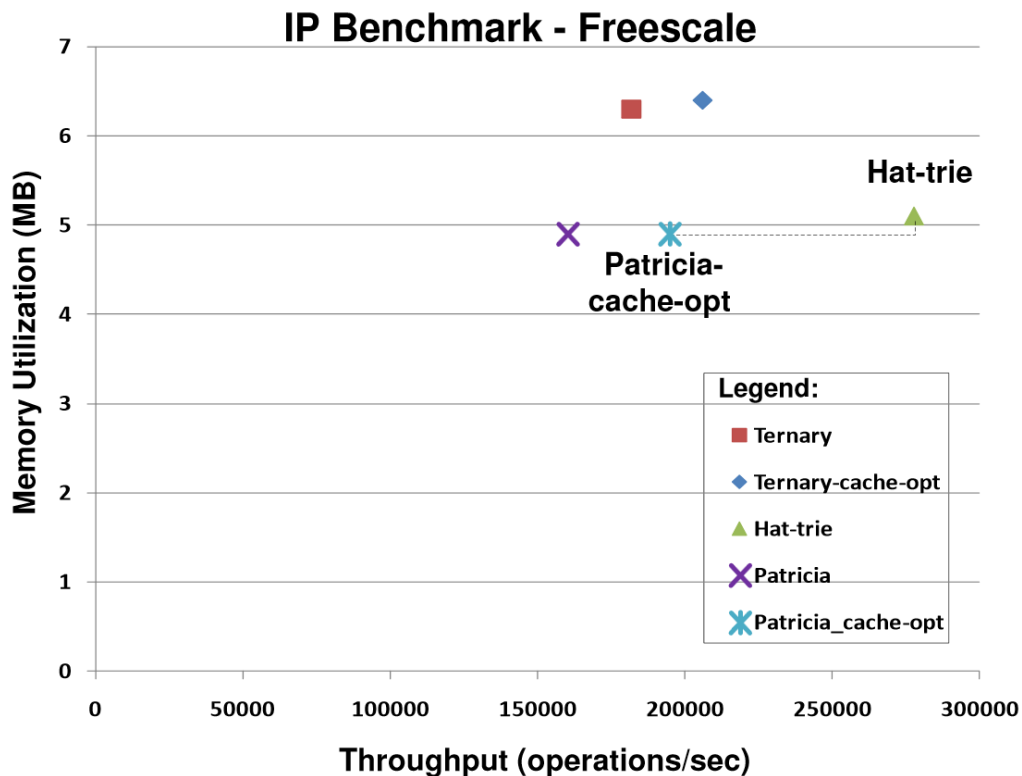


Figure 4.5: Throughput vs memory utilization of the IP benchmark in the Freescale board

throughput but requires the most memory. Again, this is explained by the small number of the different strings that substantially limits the possibility of having long prefixes and as a result, saving up space by reusing these nodes. The HAT-trie has the least memory requirements because it is basically a hash table and the elements are closely packed together. The Patricia implementation is worse in every aspect from the HAT-trie, but still beats the Ternary tree as far as memory requirements are concerned.

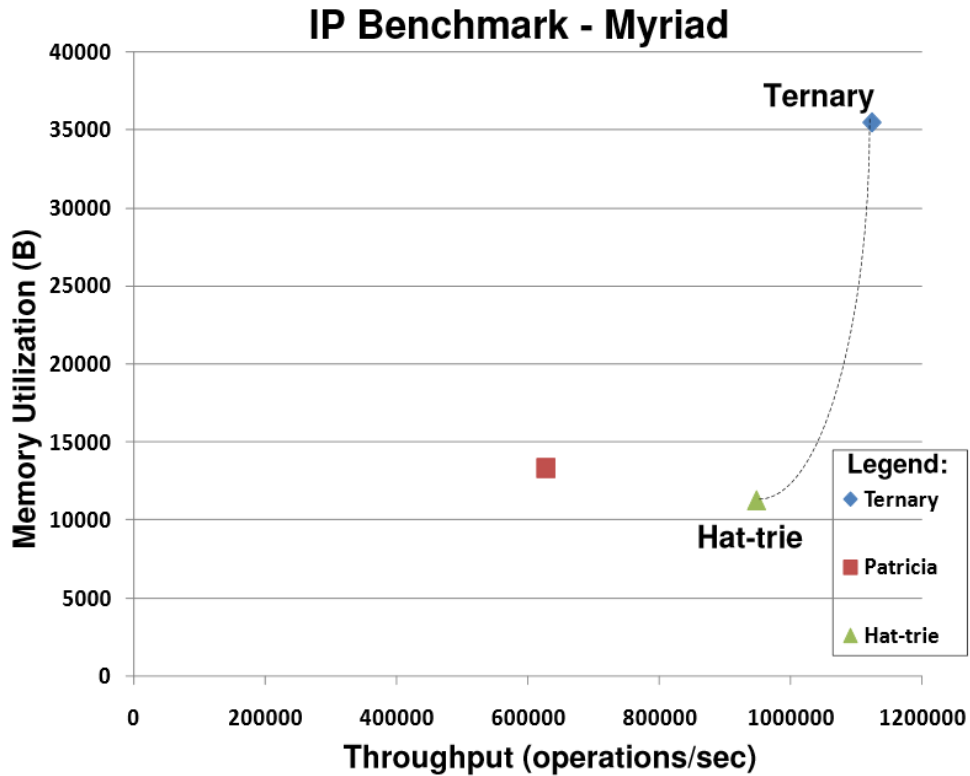


Figure 4.6: Throughput vs memory utilization of the IP benchmark in the Myriad board

4.3.2 Dictionary datasets

In this subsection, the experiments with the dictionary datasets are presented. The first experiment (Figure 4.7) contains only update operations (insertions) and the insertion requests are coming in an alphabetical order, like creating a dictionary from scratch. The optimal implementations in this case are the Ternary trie and the HAT-trie. The Ternary trie leads to 23% higher throughput in comparison to the HAT-trie, while the HAT-trie provides 18% lower memory consumption. It is noteworthy that the non-optimized implementations outperform the optimized ones.

Again, in Figure 4.8 all the operations are updates, but now the ordering is random.

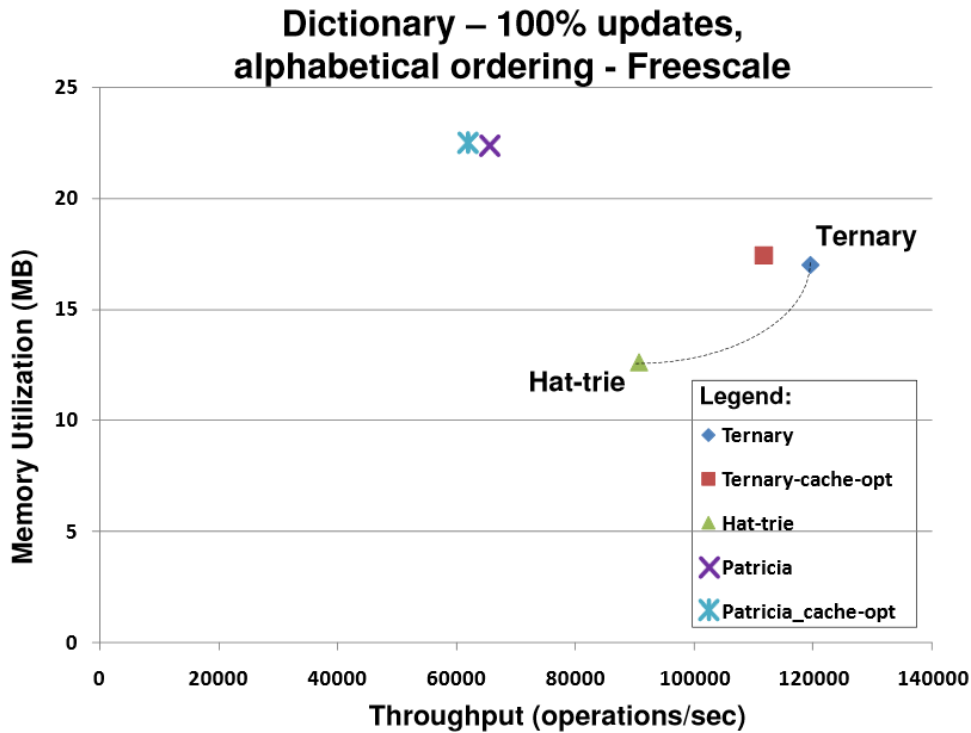


Figure 4.7: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in alphabetical order in the Freescale board

The memory consumptions remain the same as the previous experiment, but the performance is altered. Therefore, ordering matters only for the performance. When the dataset is sorted, consecutive words tend to have the same prefixes. So, the same path is

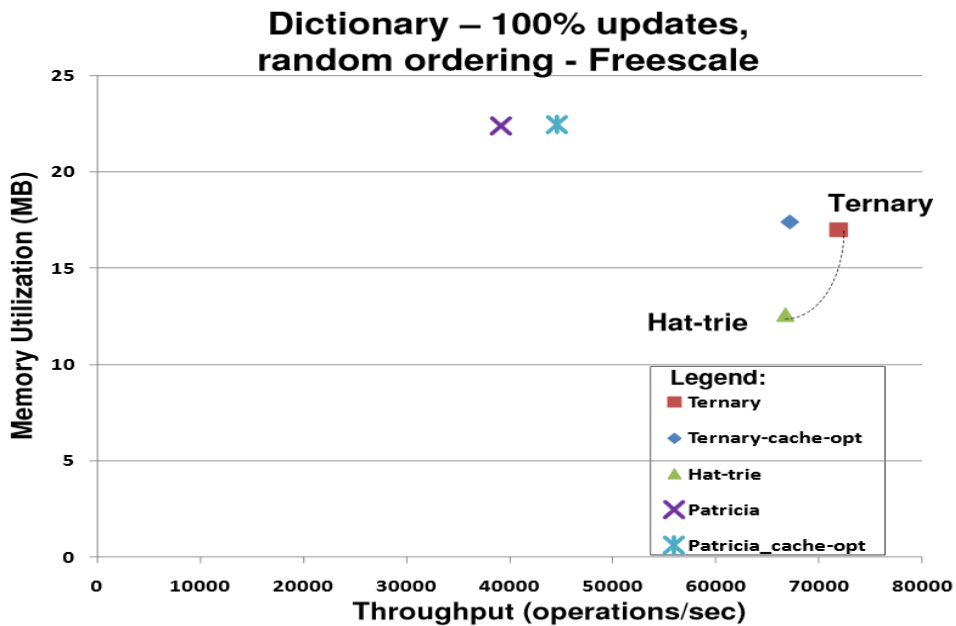


Figure 4.8: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in random order in the Freescale board

followed (up to a point) and, as this path was recently accessed, the corresponding nodes will be in the cache. In this case, the advantage of a cache conscious implementation is

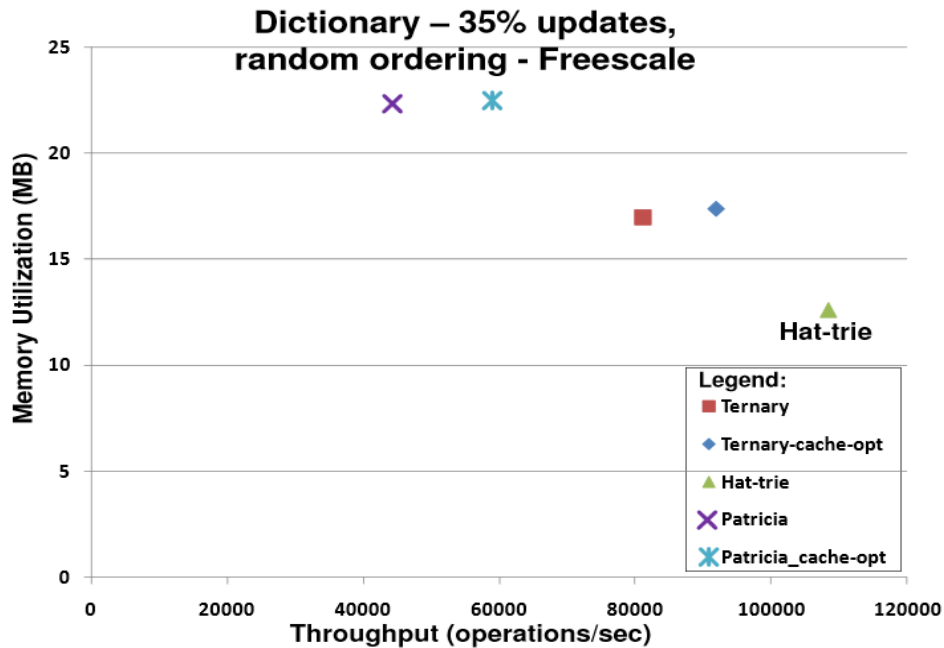


Figure 4.7: Throughput vs memory utilization of the dictionary benchmark with 35% updates and strings inserted in random order in the Freescale board

automatically limited, as the application already displays a good behavior. This phenomenon leads to superior performance (throughput). With random ordering, the patricia optimized implementation is better than the non-optimized while with alphabetical ordering the opposite is true.

Finally, in Figure 4.9 the random ordering is kept, but some entries have been duplicated in order to have some lookup-only operations instead of exclusively insertions. Once that happens, the HAT-trie achieves the best throughput while continuing to require the least memory and this is expected thanks to the fast lookup time – it is a hash table after all. Also the optimized implementations perform better than the non-optimized ones with very small memory overhead because the big number of the lookup operations cover the cost of the reorganization and provide faster execution.

The corresponding results for the Myriad board are presented in Figures 4.10, 4.11 and 4.12. As in the previous experiments, the HAT-trie and the Ternary tree provide the best throughput. However the high requirements of the ternary tree in memory continue. The reason, as explained before, is the small number of overlapping prefixes, thanks to the small number of different strings that can be stored to the limited Myriad memory, in combination with the fact that each character of the string is stored separately in its own node.

4.4 Overhead of the cache-friendly implementations

The following table (Table 4.13) presents the execution time and the overhead of the cache conscious implementation for all datasets. The overhead is closely related to the tree

reorganization process that is described in Chapter 3. More specifically the execution time overhead is the percentage of time that is spent for the tree reorganization during the execution of each benchmark. The memory footprint overhead is the memory spike that is observed when the new array is allocated for the old tree to be copied in a more efficient manner.

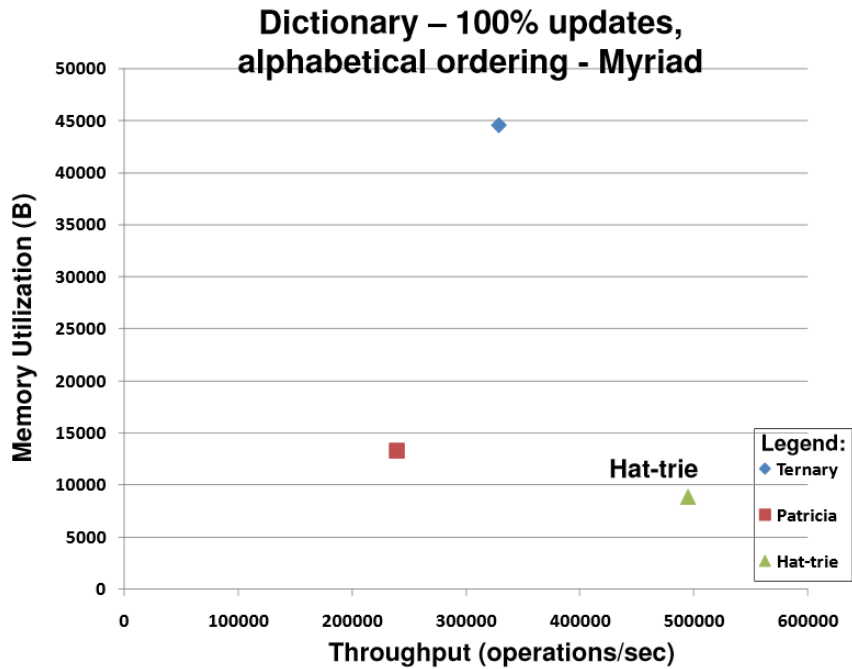


Figure 4.9: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in alphabetical order in the Myriad board

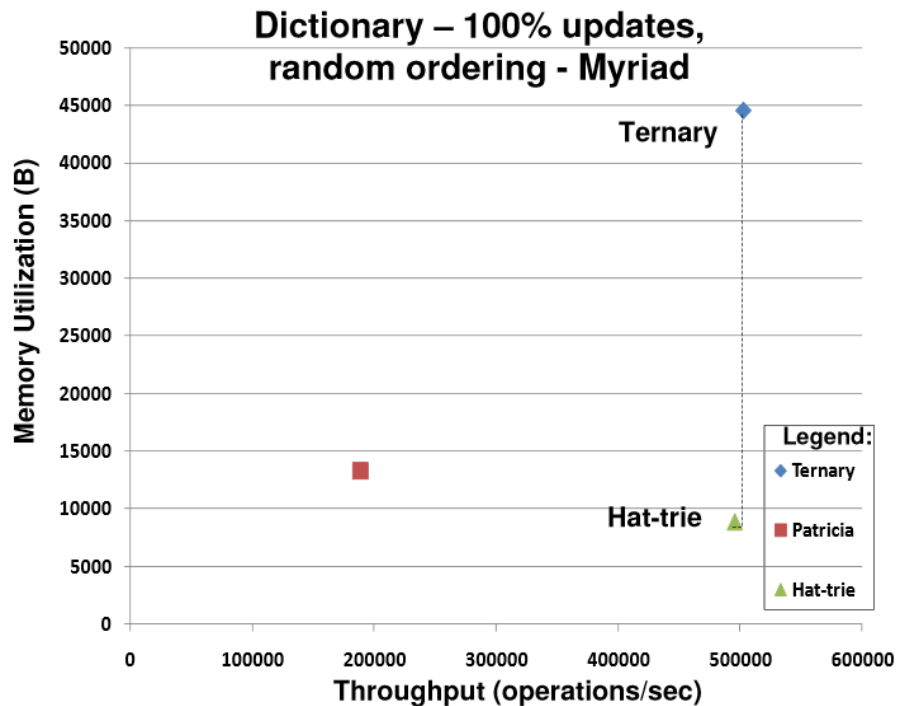


Figure 4.8: Throughput vs memory utilization of the dictionary benchmark with 100% updates and strings inserted in random order in the Myriad board

With respect to the performance overhead, it is largely dependent on the “timing” of the last reorganization. For instance, if the last reorganization took place near the end of the execution, the performance overhead will be high, as the tree is already as big as possible (it will not get much bigger in the remaining time) and therefore the amount of time that need to be invested in the reorganization is relatively big, while the advantages will be limited because there will not be sufficient time for enough operation to take place in the optimized data structure.

At an extent, this can be observed in the dictionary datasets. However in other benchmarks, such as the IP dataset, the performance overhead is very small. As far as the memory overhead is concerned, it obviously depends on the size of the tree that is copied. If the extra amount of memory that is needed is not a constraint, then this overhead can be ignored.

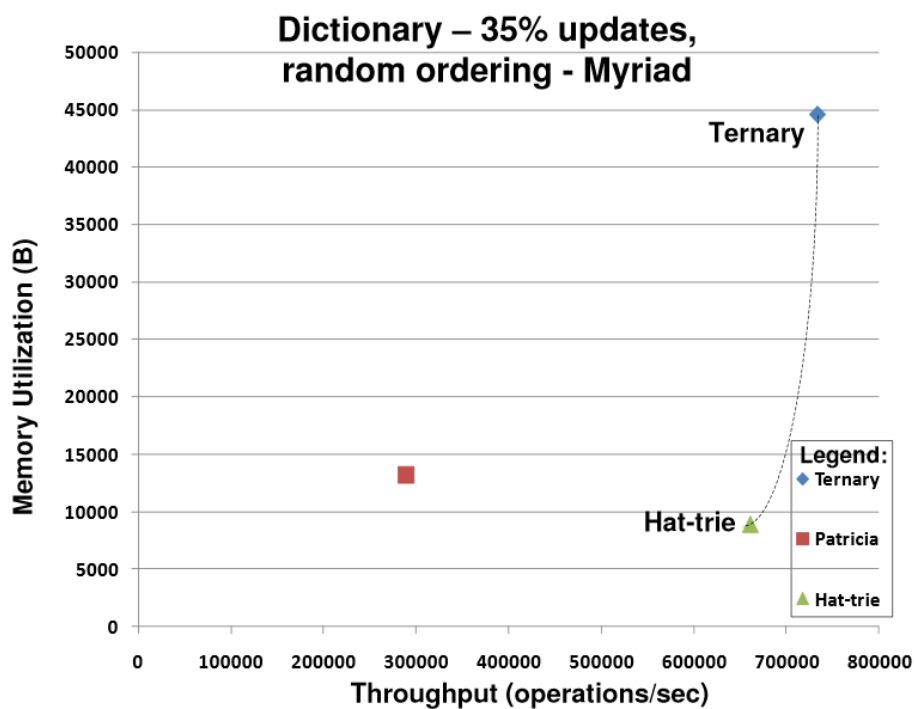


Figure 4.10: Throughput vs memory utilization of the dictionary benchmark with 35% updates and strings inserted in random order in the Myriad board

	Execution time overhead		Memory size overhead (MB)	
	Ternary cache-opt	Patricia cache-opt	Ternary cache-opt	Patricia cache-opt
Synthetic 10% upd.	5.7%	5.38%	22	28
Synthetic 80% upd.	20%	18.4%	138	168
IP dataset	4.6%	3.6%	3.7	2.6
Dict. 100% upd. alph. order	35%	15.3%	14.3	18.5
Dict. 100% upd. random order	36%	20.3%	14.3	18.5
Dict. 35% upd. random order	20%	12.1%	16.2	32

Table 4.11: Execution time and memory footprint overhead of cache conscious implementation on Freescale board

CHAPTER 5

Conclusion and future work

5.1 Summary

In this work, the original DDTR methodology, its foundations – as well as some of its improvements – and its limitations were presented. In an attempt to further improve the DDTR methodology, we expanded the DDTR library with some additional data structures (trees) that are common in many multimedia applications. A variety of trees were used for the same purpose, each implemented in a completely different manner in order to highlight their strong and weak points. Additionally, we presented a methodology to render these tree data structures into more cache-friendly ones and we tested the new modified DDTs with a collection of benchmarks.

There are apparently some general principles regarding the data structures behavior under different application requirements. For example, we noticed in all experiments that the HAT-trie performs well, when there is a large number of lookup operations. Also, in the dictionary benchmarks, the word ordering affects the performance. However, we noticed the different behavior of the same implementations between the two platforms with different memory hierarchies. Indeed, the existence (or not) of a cache plays a major role in the performance results of the data structure implementations.

The improvements made in the DDTR methodology by the integration of the cache-conscious implementations extended the methodology by making it able to adhere to hardware-related constraints such as the existence of a cache system. In many cases (such as the IP benchmarks and the synthetic ones), some Pareto points would not be “visible” with the previous DDTR methodology. In other words, the methodology is adapted, not only to the application constraints, but also to the hardware constraints and specifications. Thus, the set of data structure implementation solution increases, providing the developer with more flexibility.

To sum up, the cache conscious implementations through the data structure reorganization can be a valid alternative to the generic ones. If the performance and the memory overhead that it incurs is not a constraint, then such implementations can achieve high performance through effective cache utilization.

5.2 Future work

The basic techniques described here can be also used to improve other data structures that use pointers, such as singly linked lists. It is not a methodology that is tied to tree-like data structures, but some general principles that are followed. Apart from this, only

limited information about the hardware is taken into account. To be precise we are interested only in the existence of cache memory. But if more information is available, better decisions can be made. For example, if we have a detailed energy model for the system, we can juxtapose the energy cost of the reorganization with the energy saving from an improved cache performance in order to decide whether it is worthy (if memory is our main interest).

The next step for the DDTR methodology should be to consider more information about the application. There is a variety of profiling tools that produce massive amount of data about the program's behavior, like its access pattern. If the workload is known in advance or can be predicted, then the reorganization can be scheduled for specific moments, such as when many reads are about to happen. It is a waste to do the reorganization at the end of the program (as may very well be the case) because then we can never get its cost refunded through better performance for the remaining operations.

Finally, during the reorganization the program's execution must stop momentarily in order to reorganize the tree. If many cores are available there are partial solutions, like having another core reorganizing the tree while using the older copy. But if only one core is available, stopping the flow of the program can be prohibited, especially if the application has tight real-time constraints. It would be interesting if the reorganization could be achieved through an incremental manner, parts of the tree at each time in order to reduce the delay and make it suitable for more applications.

REFERENCES

- [1] L. Benini, A. Macii, E. Macii and M. Poncino, "Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation," *IEEE Design & Test of Computer*, pp. 74-85, 2000.
- [2] D. Wood, "Data Structures, Algorithms, and Performance," Addison-Wesley Longman Publishing Co., 1993.
- [3] S. M. A. Bartzas, G. Pouiklis, D. Atienza, F. Catthoor, D. Soudris and A. Thanailakis, "Dynamic data type refinement methodology for systematic performance – energy design exploration of network applications," 2006.
- [4] S. Mamagkakis, "Design of energy efficient wireless networks using dynamic data type refinement methodology," *Proc. of WWIC*, 2004.
- [5] G. Memik, "Netbench: A benchmarking suite for network processors," in *Proc. of ICCAD*, IEEE Press, 2001.
- [6] L. Papadopoulos, C. Mpaloukas and D. Soudris, "Exploration methodology of dynamic data structures in multimedia and network applications for embedded platforms," *Journal of System Architecture*, 2008.
- [7] C. Poucet, D. Atienza and F. Catthoor, "Template-Based Semi-Automatic Profiling of Multimedia Applications," in *The Proceedings of the International Conference on Multimedia and Expo*, 2006.
- [8] "Micron Technology," [Online]. Available: <https://www.micron.com/>.
- [9] "Astar," [Online]. Available: <https://webdocs.cs.ualberta.ca/~games/pathfind/>.
- [10] "Simblob," [Online]. Available: <http://sourceforge.net/projects/simblob/>.
- [11] "Dijkstra," [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra's_algorithm.
- [12] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the 4th IEEE Workshop Workload Characterization*, 2001.
- [13] "Weighted Fair Queueing (WFQ)," [Online]. Available: https://en.wikipedia.org/wiki/Weighted_fair_queueing.
- [14] M. Duranton, "The Challenges for High Performance Embedded Systems," *Digital*

System Design: Architectures, Methods and Tools, pp. 3-7, 2006.

- [15] "Intel Atom Processor E6x5C Series Product Preview Datasheet," Intel Corp., 2010. [Online]. Available: <http://www.arrow.com/offers/intel/e6xx/E6x5C%20Datasheet.pdf>.
- [16] W. Wolf, *High-performance embedded computing: architectures, applications and methodologies*, Morgan Kaufmann, 2007.
- [17] S. Perl and R. Sites, "Studies of Windows NT performance using dynamic execution traces," in *Second Symposium on Operating Systems Design and Implementation*, 1996.
- [18] T. Chilimbi, M. Hill and J. Larus, "Cache-conscious data structures".
- [19] Byna, Y. Chen and X.-H. Sun, "A Taxonomy of Data Prefetching Mechanisms," 2008.
- [20] Chen and Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, pp. 609-623, 1995.
- [21] F. Dahlgren, M. Dubois and P. Stenström, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," in *Proceedings of International Conference on Parallel Processing*, 1993.
- [22] Luk and Mowry, "Compiler-based Prefetching for Recursive Data Structures," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [23] Annavaram, Patel and Davidson, "Data Prefetching by Dependence Graph Precomputation," in *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.
- [24] S. Steinke, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. DATE Conference*, Washington, DC, 2002.
- [25] M. Leeman, "Automated dynamic memory data type implementation exploration and optimization," in *Proc. of the IEEE Computer Society Annual Symposium on VLSI*, Washington, DC, 2003.
- [26] N. Askitis and R. Sinha, "HAT-trie: A Cache-conscious Trie-based Data Structure for Strings," in *Proc. Thirtieth Australasian Computer Science Conference (ACSC)*, 2007.
- [27] Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, 1998.
- [28] "Ternary tree from wikipedia," [Online]. Available:

https://en.wikipedia.org/wiki/Ternary_tree.

- [29] M. Smart, "Patricia trie," [Online]. Available: https://chromium.googlesource.com/native_client/pnacl-llvm-testsuite/+/_upstream/master/MultiSource/Benchmarks/MiBench/network-patricia/patricia.c.
- [30] T. Papastergiou, L. Papadopoulos and D. Soudris, "Platform-aware Dynamic Data Type Refinement Methodology for Radix Tree Data Structures".
- [31] Freescale Semiconductor, "Freescale Semiconductor, i..MX 6Dual/6Quad Applications Processors for Industrial Products Datasheet," 2014. [Online]. Available: http://cache.freescale.com/files/32bit/doc/data_sheet/IMX6DQIEC.pdf.
- [32] "movidius website," [Online]. Available: www.movidius.com.
- [33] D. Moloney, "TOPS/W software programmable media processor," HotChips HC23, Stanford, 2011.
- [34] "The Internet Traffic Archive," [Online]. Available: <http://ita.ee.lbl.gov/>.
- [35] "WinEdt editor," [Online]. Available: <http://www.winedt.com/>.