Σοφράς Νίκος, 1115201200168
Τσιατούρας Βαγγέλης, 1115201200185
Ιανουάριος 7, 2018

ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ
ΤΕΛΙΚΗ ΑΝΑΦΟΡΑ

Τμήμα Πληροφορικής & Τηλεπικοινωνιών
Εθνικό Καποδιστριακό Πανεπιστήμιο Αθηνών

Περιεχόμενα

Γενική Περιγραφή

Η εφαρμογή υλοποιεί ένα trie, το οποίο διαχειρίζεται n-grams. Αρχικά τα n-grams εισάγονται στη δομή από το αρχείο init. Στη συνέχεια εφαρμόζονται οι πράξεις Query(Q), Add(A) και Delete(D) από το αρχείο work. Να σημειωθεί ότι στα static αρχεία η δομή υποστηρίζει μόνο πράξεις Query. Η εφαρμογή έχει υλοποιηθεί σε βελτιστοποιημένες δομές δεδομένων όπως το Bloom Filter και το Linear Hashing σε πολυνηματικό περιβάλλον.

Μια σημαντική παραδοχή που λήφθηκε κατά την υλοποίηση της εφαρμογής είναι ότι η κάθε λέξη ενός ngram αποθηκεύεται είτε σε στατικό πίνακα χαρακτήρων μεγέθους 20, είτε σε δυναμικό πίνακα χαρακτήρων ίσο με το μέγεθος της λέξης. Έτσι οι περισσότερες λέξεις αποθηκεύονται σε διαδοχικές θέσεις πίνακα ώστε να επιτυγχάνεται μικρό cache latency χωρίς να γίνεται μεγάλη σπατάλη χώρου.

Διαδικασία Αναζήτησης Κόμβων

Για την αναζήτηση κόμβων στην δομή του trie υλοποιήθηκε ένας αλγόριθμος που είναι βασισμένος στην αναζήτηση κατά βάθος (DFS) και στην δυαδική αναζήτηση. Πιο συγκεκριμένα, εφαρμόζεται αναζήτηση κατά βάθος ξεκινώντας από το root node. Στην συνέχεια για να επιλεχθεί ο επόμενος κόμβος, εφαρμόζεται δυαδική αναζήτηση πάνω στο πίνακα απογόνων του εκάστοτε κόμβου που βρίσκεται εκείνη τη στιγμή ο αλγόριθμος. Αξίζει να σημειωθεί ότι οι απόγονοι του κάθε κόμβου αποθηκεύονται ταξινομημένοι έτσι ώστε η δυαδική αναζήτηση να είναι λειτουργική.

Bloom Filter

Αυτή η δομή χρησιμοποιείται για την αποφυγή διπλότυπων αποτελεσμάτων. Είναι μια πιθανοτική δομή που βασίζεται στον κατακερματισμό και είναι ιδιαίτερα αποδοτική από άποψη ταχύτητας και χώρου.

Η υλοποίηση αυτής της δομής βασίζεται στους τύπους $p = \left(1 - \mathrm{e}^{-(\frac{m}{n}ln2)\frac{n}{m}}\right)^{\frac{m}{n}ln2}$ και $m = -\frac{nlnp}{(ln2)^2}$ για τον υπολογισμό της πιθανότητας ύπαρξης false positive σε ένα query, όπου *m* είναι το μέγεθος του πίνακα, *n* είναι η αναμενόμενη είσοδος και *p* είναι η πιθανότητα για false positive.

Χρησιμοποιούνται 5 συναρτήσεις κατακερματισμού. Για τις 4 πρώτες χρησιμοποιούμε την murmur3 με διαφορετικό seed, ενώ για την $5^{η}$ χρησιμοποιούμε τη

βελτιστοποίηση Kirsch-Mitzenmacher με τις τιμές της $1^{ης}$ και της $2^{ης}$ συνάρτησης. Το μέγεθος του πίνακα μπορεί να μεγαλώσει ανά Query αν χρειαστεί, το νέο μέγεθος θα διατηρηθεί στην υπόλοιπη διάρκεια της εκτέλεσης εφόσον δεν υπάρξει ανάγκη για μεγαλύτερο πίνακα. Η εφαρμογή προβλέπει ότι το αποτέλεσμα από ένα Query θα είναι περίπου το 70% από το μέγεθος του Query ή μικρότερο. Έτσι έχοντας ένα κατώφλι πιθανότητας 0.00001% μπορούμε να ρυθμίσουμε το μέγεθος του πίνακα ώστε να αποφευχθούν τα false positives.

Σε ότι αφορά τον πολυνηματισμό, αρχικά, δημιουργούνται *n* bloom filters, όπου *n* ο αριθμός των worker threads, μέσω της δομής BFStorage. Κάθε φορά που ένα worker thread εκτελεί ένα query η BFStorage αναθέτει ένα Bloom Filter στο συγκεκριμένο thread. Το Bloom Filter ενδέχεται να μεγαλώσει ώστε να προσαρμοστεί στις απαιτήσεις του ερωτήματος. Όταν εκτελεστεί το ερώτημα επιστρέφεται το Bloom Filter στη δομή BFStorage και γίνεται διαθέσιμο για επόμενη χρήση. Όταν τελειώσουν τα ερωτήματα από όλες τις ριπές η δομή BFStorage απελευθερώνει τη μνήμη για τα *n* Bloom Filters.

Linear Hashing

Στο πρώτο επίπεδο του trie (δηλαδή για τους απογόνους του root node) εφαρμόζεται Linear Hashing. Αρχικά το hash table αποτελείται από 32 buckets τα οποία έχουν μέγεθος 8, αργότερα η δομή προσαρμόζεται με βάση την είσοδό της. Οι κόμβοι είναι ταξινομημένοι εσωτερικά σε κάθε bucket και εφαρμόζεται δυαδική αναζήτηση για την εύρεση ενός από αυτούς. Κατά την διαγραφή ή μετακίνηση ενός κόμβου γίνεται αριστερή ολίσθηση των γειτονικών δεξιά κόμβων ώστε να καλυφθεί το κενό και να διατηρηθεί η ταξινόμηση. Αξίζει να σημειωθεί ότι δεν χρησιμοποιείται load factor. Σαν συνάρτηση κατακερματισμού χρησιμοποιείται η murmur3 με το remainder operator (%) για να προσαρμοστεί στον τρέχοντα γύρο. Μια σημαντική βελτιστοποίηση που δέχθηκε αυτή η δομή ήταν η χρήση της δεξιάς ολίσθησης (<<) έναντι της ύψωσης σε δύναμη του 2 (συνάρτηση pow στο math.h). *Βλέπε σελίδα 9.*

Top-k Ngrams

Η λειτουργικότητα για αυτό το μέρος της εφαρμογής είναι υλοποιημένη στο ngramcounter.c/.h. Αρχικά τα αποτελέσματα από κάθε query εισάγονται σε ένα στατικό hash table μεγέθους 39999 μαζί με ένα μετρητή που μετράει πόσες φορές εμφανίζεται το αποτέλεσμα στη ριπή. Όταν πάει να εισαχθεί ένα ngram που υπάρχει ήδη τότε απλά

αυξάνεται ο μετρητής του. Όταν τελειώσει η ριπή τα ζευγάρια ngram και counter αντιγράφονται σε ένα πίνακα. Στη συνέχεια καλείται η quickselect, η οποία έχει μέση πολυπλοκότητα *O(n)* και βρίσκει τη θέση που θα υπάρχει το k-οστό στοιχείο και ταξινομεί εν μέρει τον πίνακα. Τέλος, καλείται η quicksort η οποία ταξινομεί τον πίνακα από την πρώτη θέση μέχρι τη θέση *k* με μέση πολυπλοκότητα *O(klogk)* και τυπώνονται τα αποτελέσματα της ριπής με φθίνουσα σειρά. Μόλις τελειώσει η εκτύπωση διαγράφεται ο πίνακας και τα δεδομένα από το στατικό hash table ώστε να είναι έτοιμο για την επόμενη ριπή. Έτσι, με αυτή τη δομή επιτυγχάνεται μέση πολυπλοκότητα *O(n)* όταν το *n* είναι αρκετά μεγαλύτερο από το *k* και *O(klogk)* όταν το *k* είναι ίδιας τάξης με το *n.*

Πολυνηματισμός

Στα πλαίσια της εφαρμογής υλοποιήθηκε ένας generic job scheduler και ένα thread pool. Ο scheduler αποτελείται από μια ουρά από εργασίες (Jobs) και έχει αναπτυχθεί με τέτοιο τρόπο ώστε να μπορεί να λειτουργήσει και σε οποιαδήποτε εφαρμογή που χρειάζεται παραλληλία. Το thread pool αποτελείται από worker threads τα οποία δημιουργούνται στην αρχή του προγράμματος και περιμένουν να τους ανατεθεί μια εργασία από τον scheduler. Ο τελευταίος, δέχεται jobs μέσω της συνάρτησης submit_scheduler και τα τοποθετεί στο τέλος της ουράς. Μόλις υπάρχει κάποια εργασία διαθέσιμη στην ουρά, “ξυπνάει” ένα worker thread και αναλαμβάνει το πέρας της. Όταν τελειώσει, επιστρέφει στο thread pool μέχρι να του ανατεθεί μια νέα εργασία.

Επομένως το main thread είναι υπεύθυνο για την διαχείριση του job scheduler. Αρχικά, διαβάζει τα αρχεία εισόδου και αποθηκεύει τα Queries σε μια δομή (QueryList). Στη συνέχεια όταν εκτελεστεί η πράξη F, δηλαδή το τέλος της ριπής, εξάγει τα Queries από την QueryList και τα αναθέτει στον job scheduler. Δηλαδή κάθε εργασία έχει σαν παράμετρο ένα Query. Να σημειωθεί ότι την χρονική στιγμή που το main thread υποβάλλει όλα τα Queries στον scheduler, τίθεται σε αναμονή έως ότου όλα τα Queries εκτελεστούν.

Από την άλλη, τα worker threads βρίσκονται σε αναμονή όσο δεν τους έχει ανατεθεί κάποια εργασία. Κάθε worker thread θα αναλάβει ξεχωριστό Query και είναι υπεύθυνο να αποθηκεύσει τα αποτελέσματα σε συγκεκριμένη θέση του «πίνακα» της δομής QueryResults. Τέλος, αφού έχει ολοκληρώσει τη παρούσα εργασία, θα αναλάβει νέα εργασία ή θα πέσει σε ύπνωση αν δεν υπάρχει κάποια εργασία διαθέσιμη.

Static/compress

Όταν η εφαρμογή λειτουργεί σε static mode πρέπει να συμπιέσει τους κόμβους με έναν μόνο απόγονο. Για να πραγματοποιηθεί αυτό, καλείται η συνάρτηση compress_trie όταν τελειώσει η αρχικοποίηση. Αρχικά προσπελαύνει το trie κατά βάθος και σε πρώτη φάση ελέγχει αν έχουν έναν απόγονο μετρώντας το πλήθος αυτών των κόμβων και τελικά τους συγχωνεύει σε έναν. Οι λέξεις των συμπιεσμένων κόμβων αποθηκεύονται δυναμικά στον ίδιο πίνακα διαδοχικά η μία μετά την άλλη. Για να γίνεται η προσπέλαση των συμπιεσμένων λέξεων χρησιμοποιείται ένας πίνακας μεγέθους όσο και το πλήθος των λέξεων που έχουν συμπιεστεί. Ο πίνακας αυτός διατηρεί το μήκος της κάθε λέξης και αν είναι τερματική ή όχι. Παρατηρήθηκε ότι η συμπίεση των κόμβων προσέφερε μεγάλη βελτίωση όχι μόνο στο χώρο, αλλά και στον χρόνο εκτέλεσης στα μεγάλα datasets.

Query Results

Η συγκεκριμένη δομή είναι υπεύθυνη για την προσωρινή αποθήκευση και εκτύπωση των αποτελεσμάτων στο εσωτερικό μιας ριπής. Αποτελείται από έναν πίνακα από οριζόντιες γραμμές στις οποίες αποθηκεύονται τα αποτελέσματα του κάθε ερωτήματος, ένα ερώτημα ανά γραμμή. Για παράδειγμα, μόλις το worker thread που έχει αναλάβει το πρώτο ερώτημα μίας ριπής τελειώσει, θα αποθηκεύσει τα αποτελέσματα στην πρώτη γραμμή αυτής της δομής. Με αυτόν τον τρόπο εξασφαλίζεται η σωστή σειρά των αποτελεσμάτων. Όταν όλα τα ερωτήματα της ριπής έχουν απαντηθεί, θα ξυπνήσει το main thread για να τυπώσει τα αποτελέσματα και να καθαρίσει τη δομή για την επόμενη ριπή. Το πλεονέκτημα αυτής της δομής είναι ότι υπάρχει ταυτόχρονη πρόσβαση στις γραμμές του πίνακα και έτσι μπορούν πολλά worker threads να αποθηκεύουν τα αποτελέσματα τους ταυτόχρονα.

Versioning

Εξαιτίας του πολυνηματισμού, όταν η εφαρμογή λειτουργεί σε dynamic mode προέκυψε το πρόβλημα της ασυνέπειας σωστών αποτελεσμάτων, καθώς μεταξύ των Queries πραγματοποιούνται πράξεις Add και Delete. Πιο συγκεκριμένα, πρώτα εκτελούνται οι πράξεις Add και Delete και τελευταία τα Queries. Ο τρόπος με τον οποίο λύθηκε το πρόβλημα αυτό ήταν να εφαρμοστεί versioning στους κόμβους του trie. Αυτό πραγματοποιήθηκε προσθέτοντας 3 πεδία σε κάθε κόμβο τα οποία περιγράφουν σε ποια «έκδοση» του trie αυτοί οι κόμβοι εισήχθησαν, διεγράφησαν και ετικετοποιήθηκαν σαν

τερματικοί. Οι συναρτήσεις Insert και Delete είναι υπεύθυνες για την εκχώρηση αυτών των μεταβλητών με βάση την «έκδοση» στην οποία κλήθηκαν την προκειμένη στιγμή για ένα ngram. Στην περίπτωση όπου εκτελείται ένα Query, για την εύρεση των κόμβων που πρέπει να εκτυπωθούν, πρέπει να ισχύει η συνθήκη: *NodeAppendVersion <= QueryVersion && NodeDeleteVersion > QueryVersion*. Για την περίπτωση όπου ένας κόμβος εισήχθη κατά την αρχικοποίηση του trie ο παραπάνω κανόνας δεν ισχύει, αφού το NodeDeleteVersion αρχικοποιείται σε -1 επομένως η συνθήκη θα μετατραπεί ως εξής: *NodeAppendVersion <= QueryVersion && (NodeDeleteVersion > QueryVersion || NodeDeleteVersion == -1).* Τέλος κρίθηκε αναγκαίο να υπάρχει γνώση σε ποια έκδοση ένας κόμβος ετικετοποιήθηκε σαν τερματικός. Πιο συγκεκριμένα, στις περιπτώσεις όπου γίνονται δυο Append τα οποία έχουν ανάμεσά τους ένα Query, υπάρχει περίπτωση κατά την εκτέλεσή του να προκύψουν λανθασμένα αποτελέσματα. Για παράδειγμα:

- A the dog is brown (“brown” τελική κατάσταση)
- Q the dog is brown and big (Λανθασμένη εκτύπωση “the dog” και “the dog is brown”)
- A the dog (“brown” “dog” τελικές καταστάσεις)

Εάν η συνάρτηση Query δεν λάβει υπόψιν την version στην οποία ετικετοποιήθηκε η λέξη “dog”, θα τυπωθούν λάθος αποτελέσματα.

Μετρήσεις & Στατιστικά

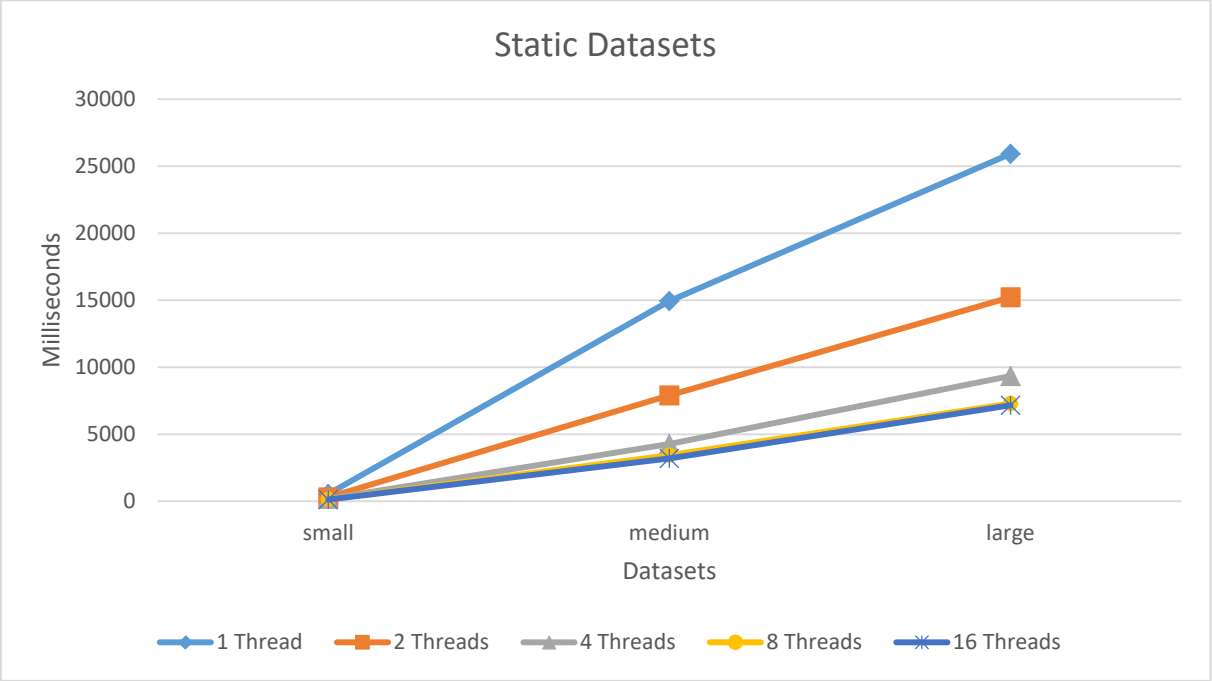
Στη παρακάτω ενότητα ακολουθούν μετρήσεις όσον αφορά χρόνους εκτέλεσης της εφαρμογής, συγκριτικά στατιστικά και μετρήσεις κατανάλωσης μνήμης. Για την εξαγωγή των αποτελεσμάτων χρησιμοποιήθηκαν τα εξής προγράμματα:

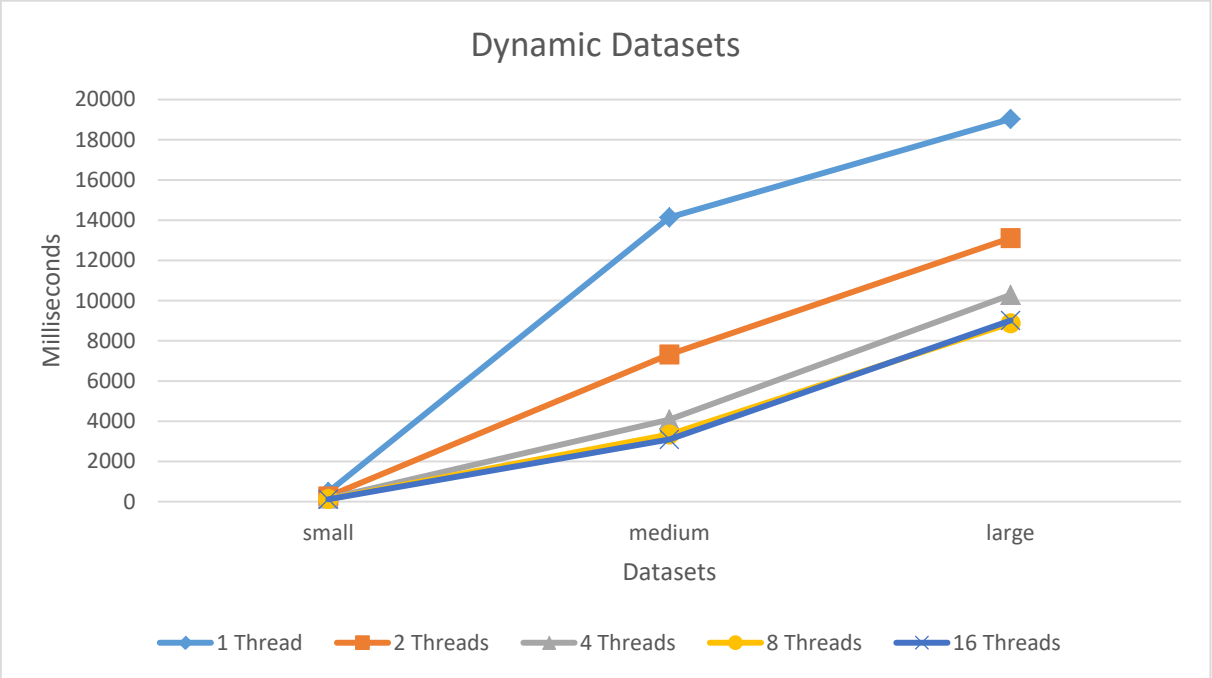
- Valgrind
- Kcachegrind
- Massif
- Massif-Visualizer

Οι προδιαγραφές του μηχανήματος στο οποίο έγιναν οι δοκιμές και οι μετρήσεις για την εφαρμογή είναι:

- CPU: Intel Core i7 6700k @ 4 GHz
- RAM: 16 GB DDR4 @ 3000 MHz

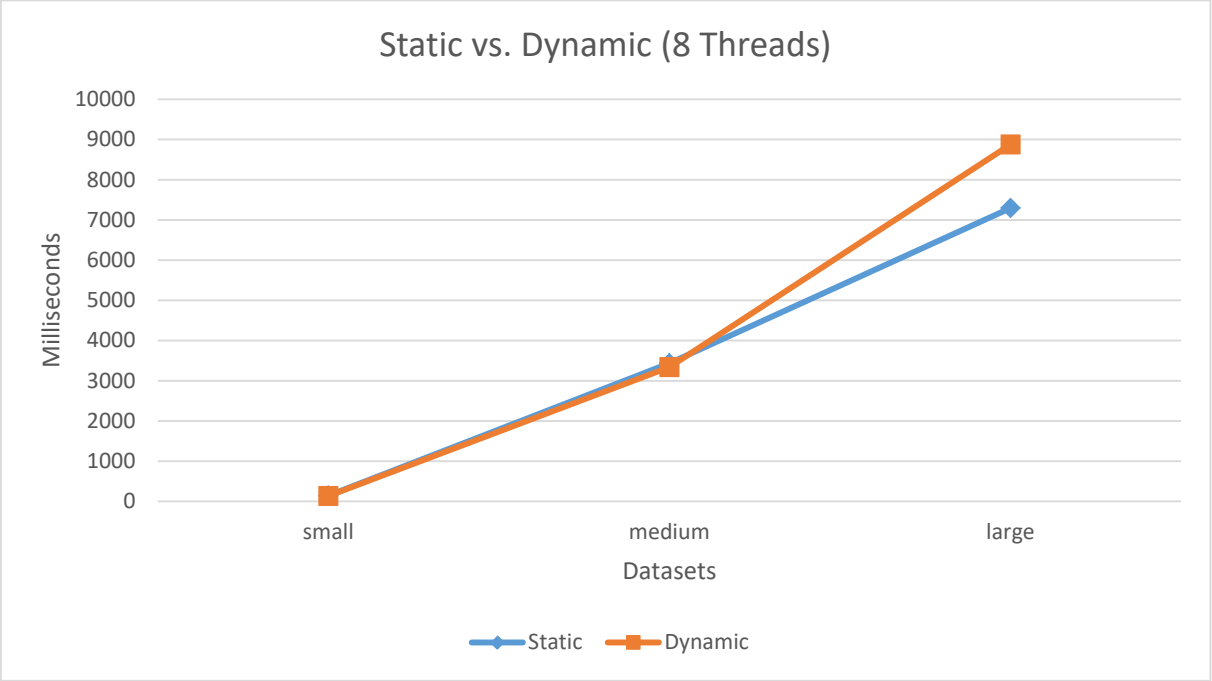
Χρόνοι εκτέλεσης





Με βάση τα παραπάνω διαγράμματα καταλήξαμε στο συμπέρασμα ότι η εφαρμογή με 8 Threads έχει την καλύτερη επίδοση από άποψη ταχύτητας εκτέλεσης. Επίσης με περισσότερα Threads δεν παρατηρήθηκε βελτίωση.

Static vs. Dynamic

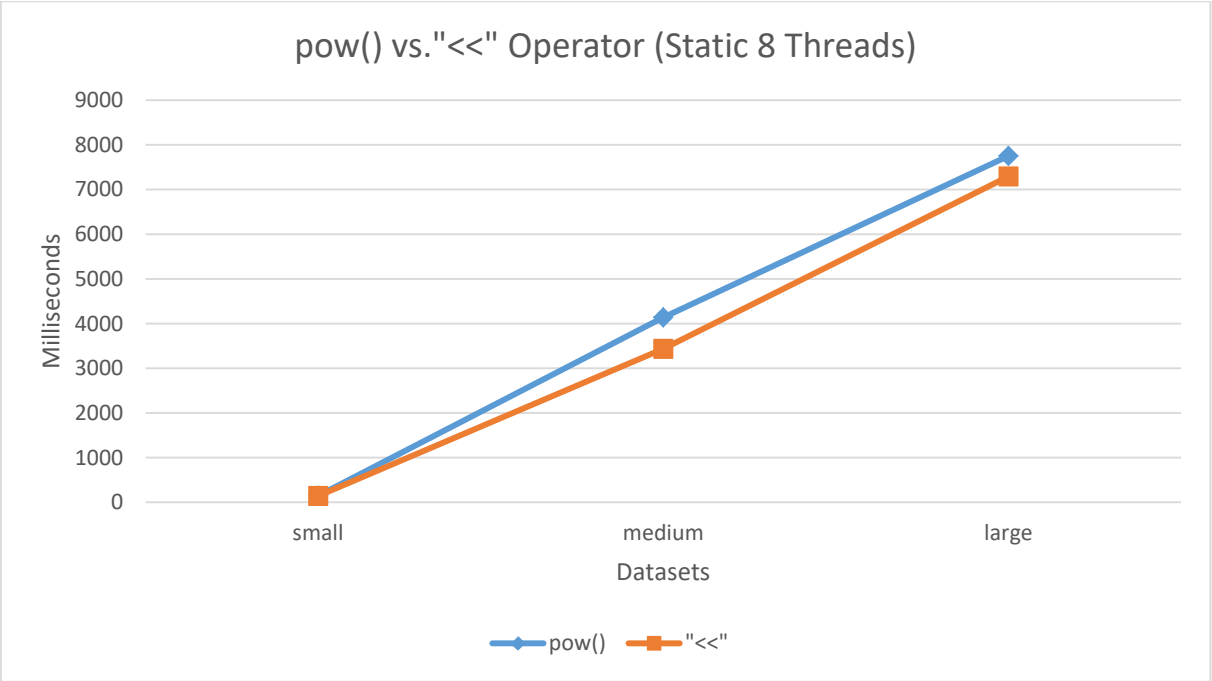


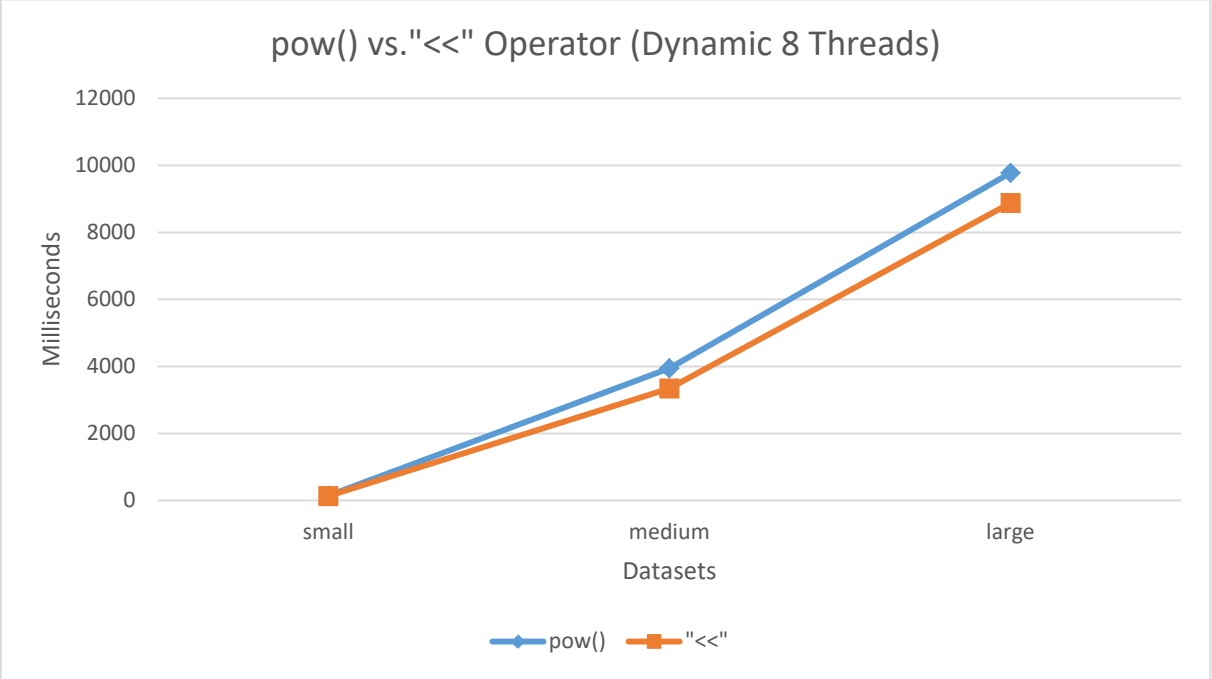
Στο παραπάνω διάγραμμα παρατηρούμε ότι οι χρόνοι εκτέλεσης είτε η εφαρμογή εκτελείται σε static είτε σε dynamic mode, για small και medium datasets είναι παρόμοιοι. Κάτι τέτοιο όμως δεν ισχύει στο large dataset. Όταν η εφαρμογή εκτελείται σε static mode καταφέρνει ταυτόχρονα μικρότερη κατανάλωση μνήμης και ταχύτερο (κατά 18%) χρόνο εκτέλεσης.

Optimization in Linear Hashing

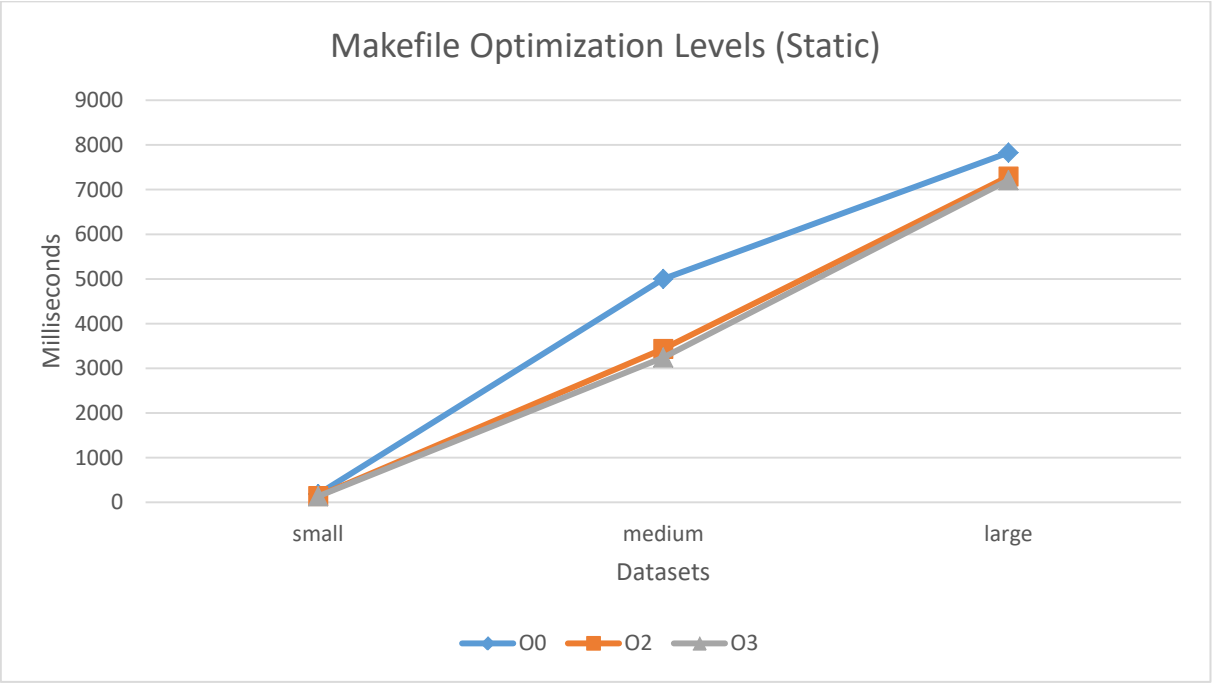
Η υλοποίηση της εφαρμογής αρχικά χρησιμοποιούσε την pow() από την math.h για τον υπολογισμό ύψωσης σε δύναμη στον αλγόριθμο του γραμμικού κατακερματισμού για την εύρεση ενός κάδου. Τελικά η συνάρτηση pow() αντικαταστάθηκε με πράξεις ολίσθησης (<<), καθώς είναι πιο αποδοτικές από άποψης ταχύτητας. Ακολουθούν συγκριτικά διαγράμματα μεταξύ των δύο υλοποιήσεων.

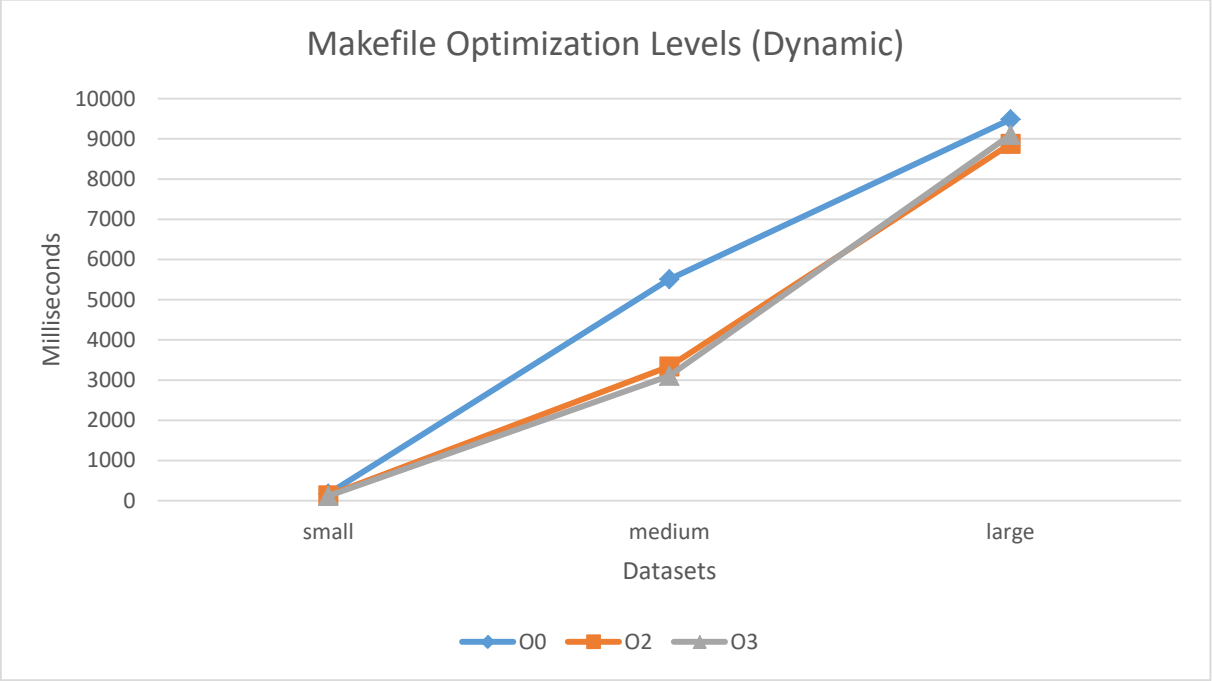
Datasets	Speed Up
Large Dynamic	10%
Large Static	6%
Medium Dynamic	15%
Medium Static	17%


pow() vs."<<" Operator (Static 8 Threads)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Milliseconds
small
medium
large
Datasets
pow()
"<<"


pow() vs."<<" Operator (Dynamic 8 Threads)
12000
10000
8000
6000
4000
2000
0
Milliseconds
small
medium
large
Datasets
pow()
"<<"

Makefile Optimization

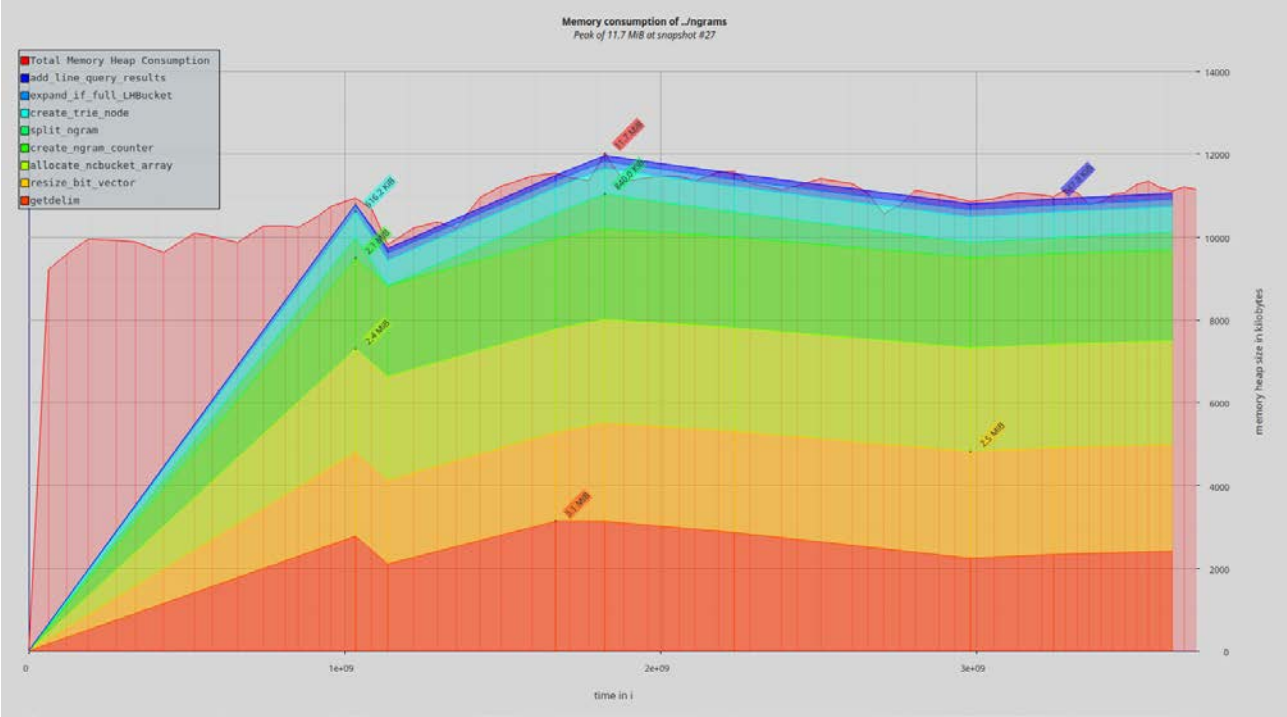




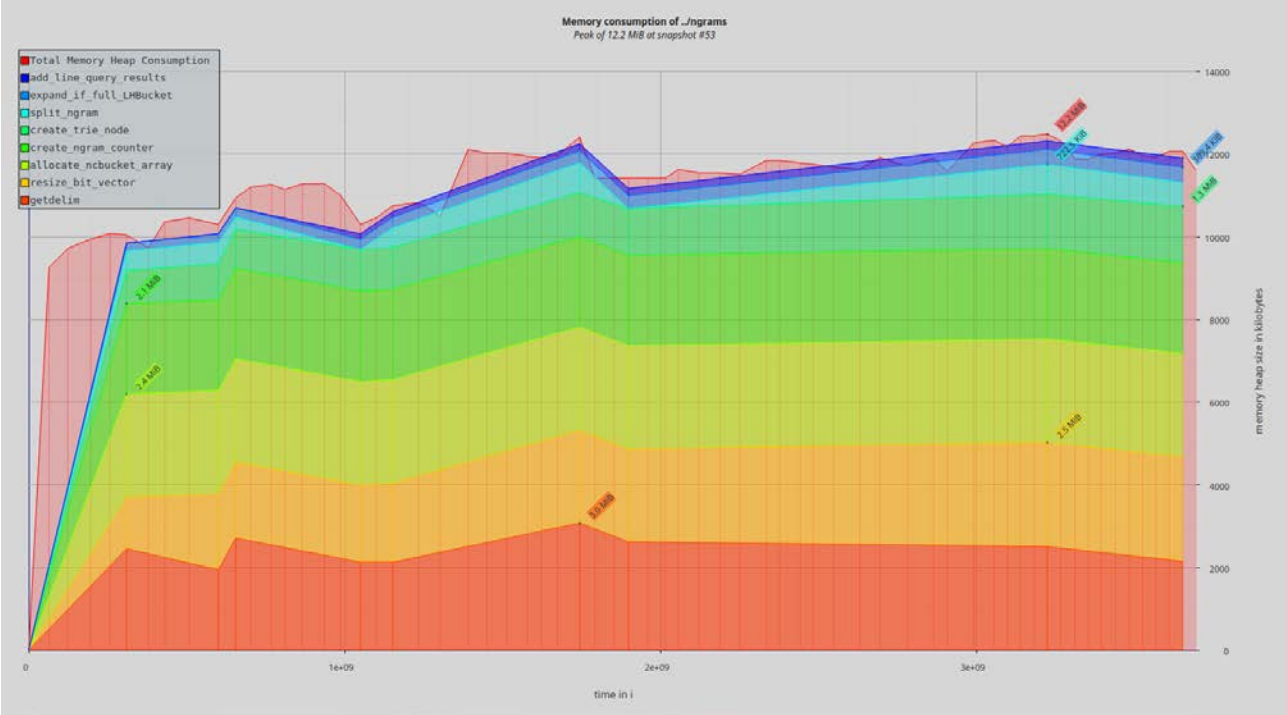
Για την μεταγλώττιση της εφαρμογής χρησιμοποιήθηκαν optimization modes του gcc. Συγκριτικά το Optimization Level 2 παρείχε καλύτερους χρόνους εκτέλεσης σε σχέση με τα άλλα modes.

Memory Consumption

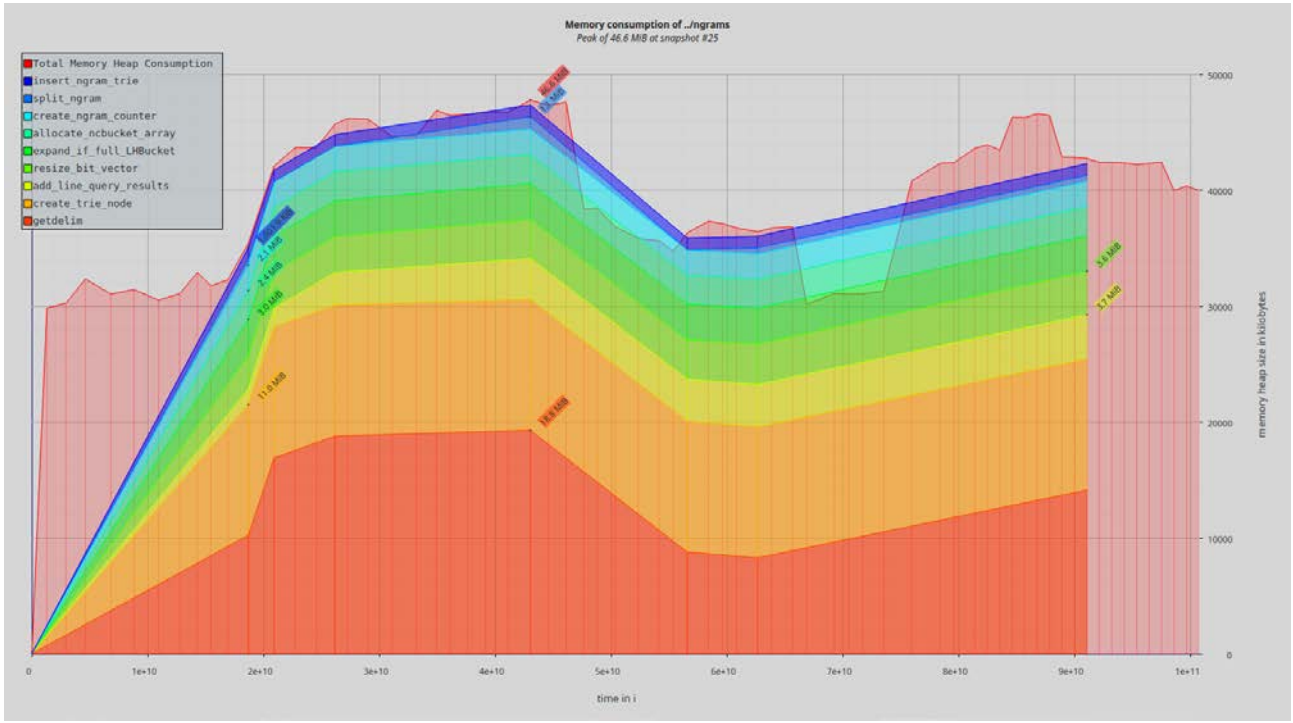
Small Static Dataset



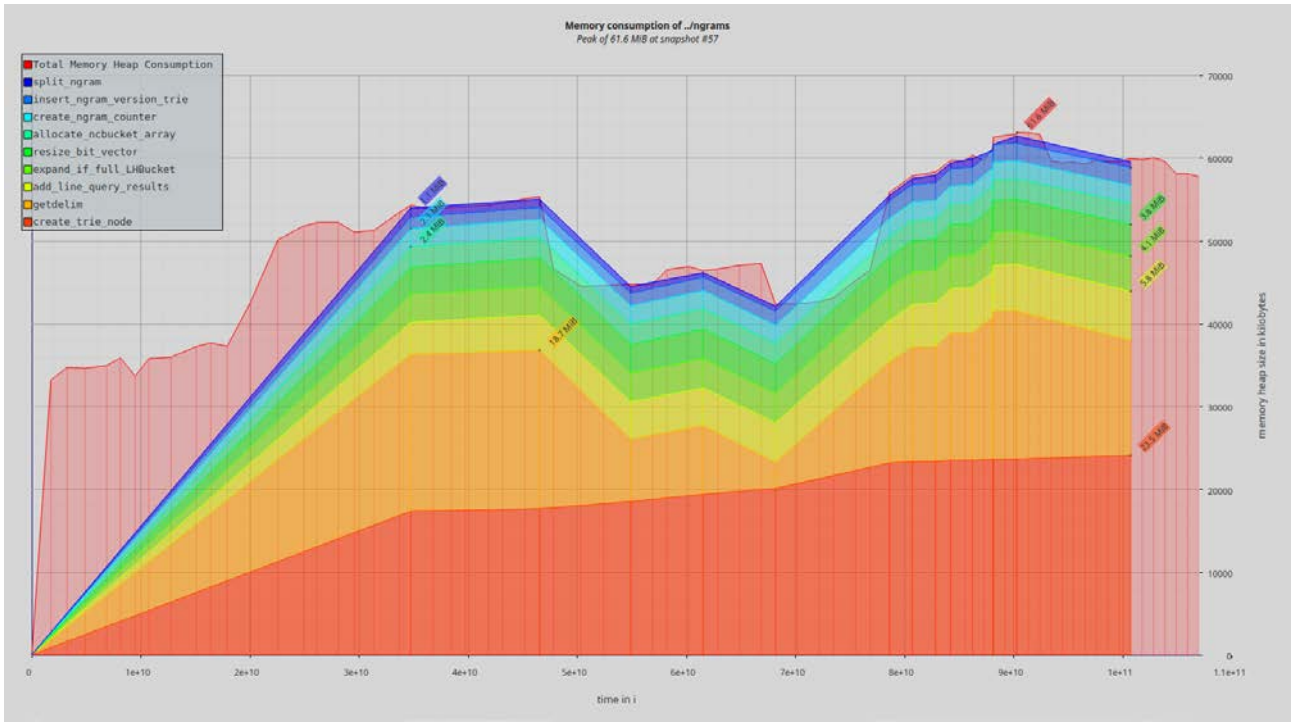
Small Dynamic Dataset



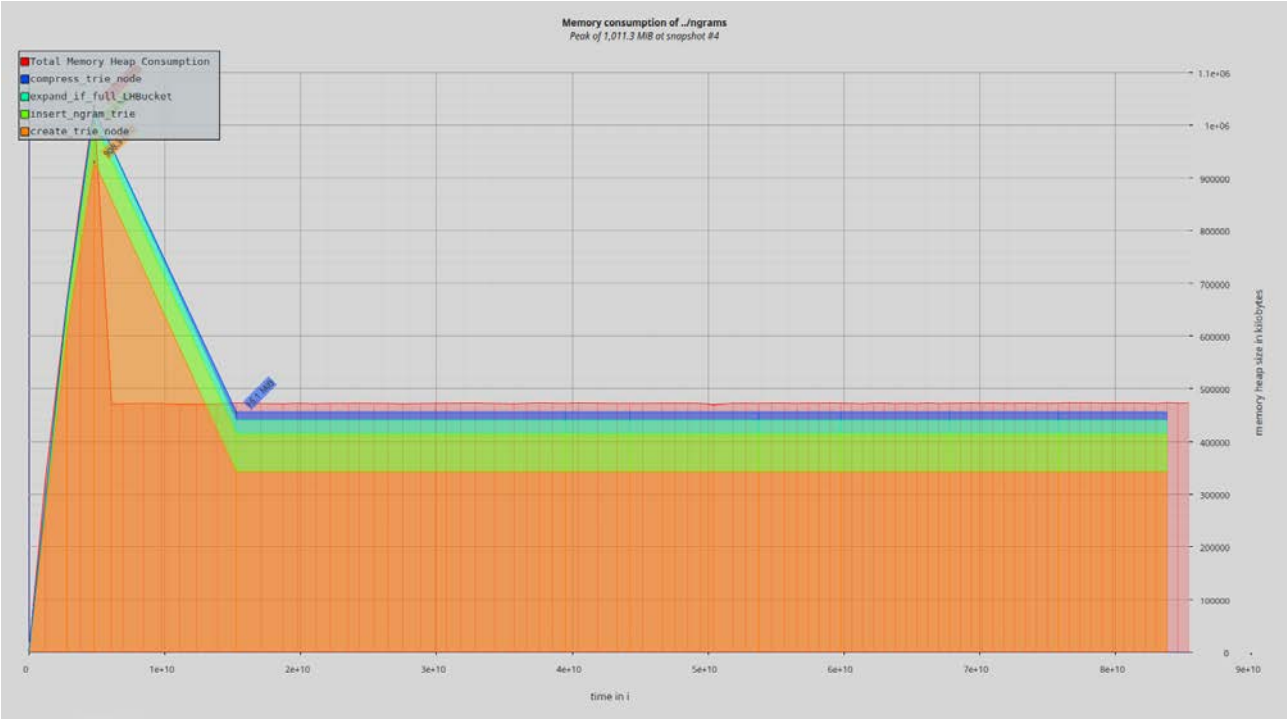
Medium Static Dataset



Medium Dynamic Dataset



Large Static Dataset



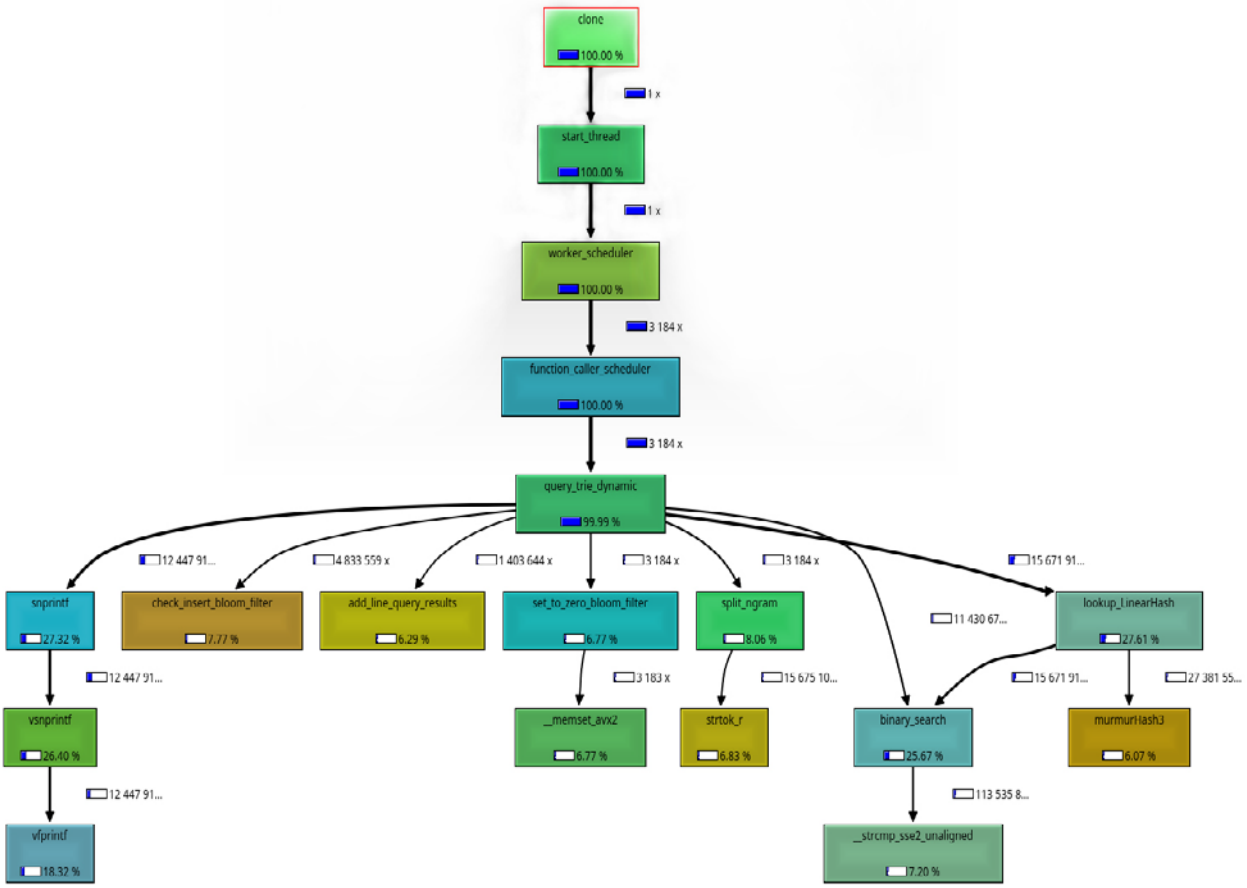
Large Dynamic Dataset

Παρατηρούμε ότι στα small και medium datasets το static mode καταλαμβάνει περίπου 25% λιγότερη μνήμη σε σχέση με το dynamic. Τα αποτελέσματα στο large dataset είναι εντυπωσιακά. Η μέγιστη κατανάλωση μνήμης στο static mode είναι κατά 47% μικρότερη. Επιπλέον, στο γράφημα για το large dataset σε static mode η δεσμευμένη μνήμη μετά τη συμπίεση του trie μειώνεται από 1011 MB σε 470 MB.

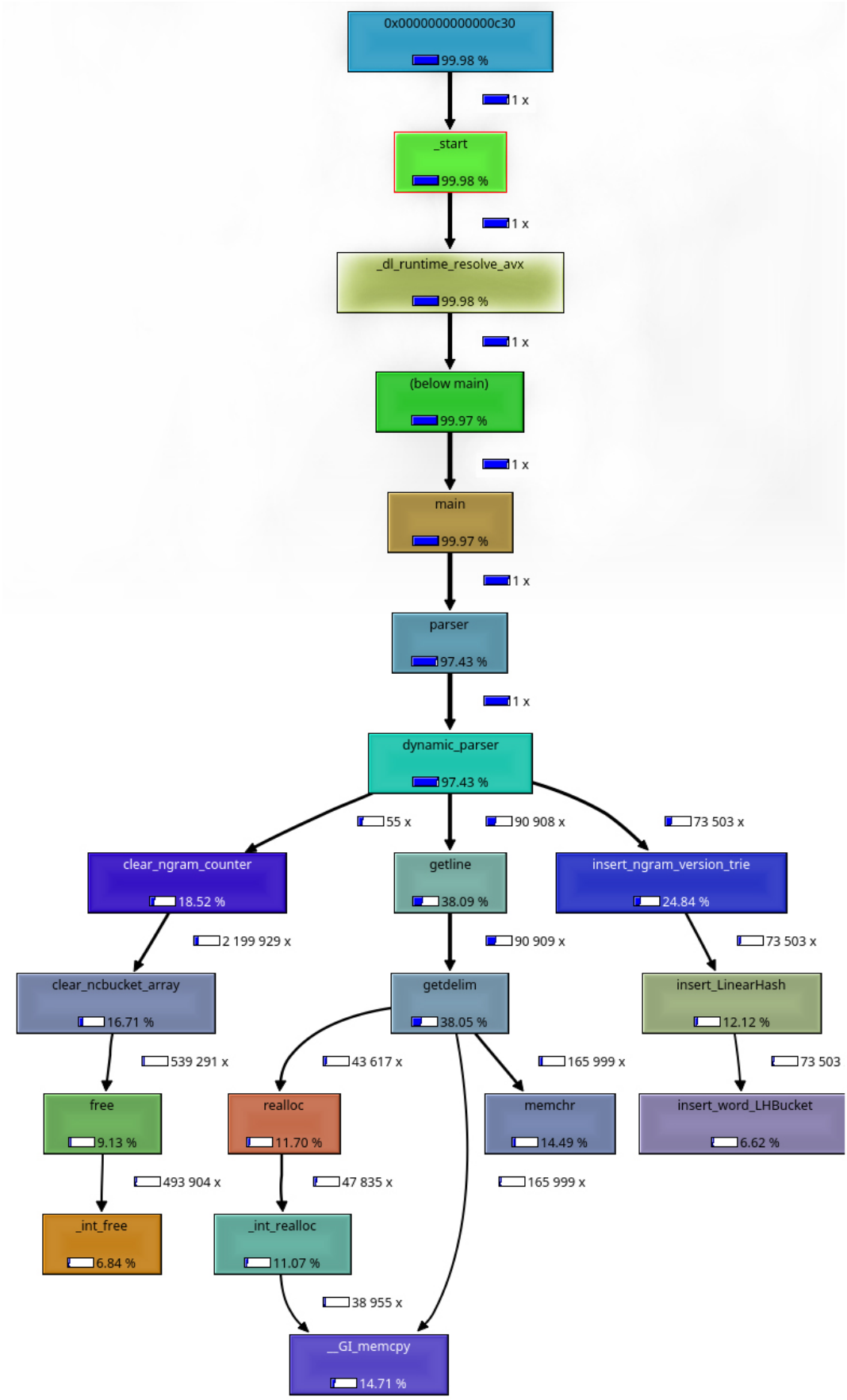
Call flow

Ακολουθούν διαγράμματα της ροής εκτέλεσης της εφαρμογής:

Worker Thread



Master Thread



Σημειώσεις

Για την ανάπτυξη της εφαρμογής χρησιμοποιήθηκε καθ’ όλη τη διάρκειά της version control (Git). Ο κώδικας της εφαρμογής βρίσκεται online στο repository.

Στα πρώτα στάδια της ανάπτυξης της εφαρμογής χρησιμοποιήθηκαν unit tests με το framework Check. Στην πορεία, επιλέξαμε να υλοποιούμε δικές μας tester συναρτήσεις σε κάθε αρχείο.