Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Συλλογη Σκουπιδιων σε Συστηματα Πιστοποιημενου Κωδικα

Διπλωματικη Εργασια

του

Ευθυμίου Γ. Βερβαινιώτη

Επιβλέπων: Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Εργαστηριο Τεχνολογιας Λογισμικου

Αθήνα, Νοέμβριος 2011

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Τεχνολογιας Λογισμικου

Συλλογη Σκουπιδιων σε Συστηματα Πιστοποιημενου Κωδικα

Διπλωματικη Εργασια

του

Ευθυμίου Γ. Βερβαινιώτη

Επιβλέπων: Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 9η Νοεμβρίου 2011.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νικόλαος Παπασπύρου	Κωνσταντίνος Σαγώνας	Νεκτάριος Κοζύρης
Επίκ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2011

(Υπογραφή)

.....

Ευθυμιος Γ. Βερβαινιωτης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Τεχνολογιας Λογισμικου

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή κ. Νικόλαο Παπασπύρου για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να την εκπονήσω. Επίσης ευχαριστώ ιδιαίτερα τον υποψήφιο διδάκτορα Άγγελο Μανουσαρίδη για την καθοδήγησή του και την εξαιρετική συνεργασία του. Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου για την καθοδήγησή τους, καθώς και για την ηθική και υλική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Ευθύμιος Βερβαινιώτης
Αθήνα, 9 Νοεμβρίου 2011

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-3-11, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Νοέμβριος 2011.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περίληψη

Η διαχείριση της μνήμης που έχει εκχωρηθεί σε ένα πρόγραμμα αποτελεί μια σοβαρή επιβάρυνση για τον προγραμματιστή, όταν αυτή γίνεται χειροκίνητα. Σε αυτήν την περίπτωση ο ίδιος ο προγραμματιστής είναι υπεύθυνος για την παρακολούθηση του χρόνου ζωής των αντικειμένων, των οποίων τη μνήμη πρέπει να αποδεσμεύει όταν αυτά δεν χρειάζονται πια. Αυτό έχει ως αποτέλεσμα ο κώδικας που απαιτείται για την δουλειά αυτή να δυσκολεύει την υλοποίηση, την συντήρηση, ακόμα και τη σχεδίαση των προγραμμάτων, ενώ συχνά μπορεί να οδηγήσει σε σφάλματα, όπως αιωρούμενες αναφορές και διαρροές μνήμης. Μια λύση στο πρόβλημα αυτό έρχεται να δώσει η συλλογή σκουπιδιών, η οποία απαλλάσσει τον προγραμματιστή από το καθήκον να παρακολουθεί αυτός τα αντικείμενα και να τα αποδεσμεύει όταν δεν είναι πια χρήσιμα (σκουπίδια). Η διαδικασία αυτή πραγματοποιείται κατά την ώρα εκτέλεσής ενός προγράμματος από το ίδιο το πρόγραμμα, το οποίο φέρει κώδικα που διαπιστώνει πότε ένα αντικείμενο δεν είναι χρήσιμο και ανάλογα φροντίζει να αποδεσμεύσει αυτόματα την μνήμη που έχει εκχωρηθεί σε αυτό. Η συλλογή σκουπιδιών χρησιμοποιείται ευρέως στις συναρτησιακές γλώσσες προγραμματισμού, καθώς σε αυτές τις γλώσσες είναι δύσκολη η ρητή αποδέσμευσή μνήμης.

Σκοπός της εργασίας αυτής είναι η συγκριτική μελέτη διάφορων αλγορίθμων συλλογής σκουπιδιών, κατάλληλων για χρήση σε ένα σύστημα μεταγλώττισης πιστοποιημένων εκτελέσιμων (certified binaries). Γίνεται εκτενής παρουσίαση των αλγορίθμων και αναφορά στα πλεονεκτήματα και μειονεκτήματα του καθενός. Οι αλγόριθμοι που αναφέρονται υλοποιήθηκαν σε C για το NFlint, ένα σύστημα μεταγλώττισης πιστοποιημένων εκτελέσιμων που έχει υλοποιηθεί στο Εργαστήριο Λογισμικού του ΕΜΠ.

Λέξεις Κλειδιά

Συλλογή σκουπιδιών, σωρός, αυτόματη αποδέσμευση μνήμης, διαρροή μνήμης, σημείωση και σάρωση, σημείωση χωρίς σάρωση, σημείωση με συμπύκνωση, διακοπή και αντιγραφή, γενετική συλλογή σκουπιδιών.

Abstract

Manual management of the memory allocated to a program can be a strenuous task for the programmer. In this case, the programmer is responsible for determining the lifetime of all objects, whose memory must be explicitly deallocated when they will not be needed any more. This complicates the design, implementation and maintenance of programs and potentially leads to certain types of bugs such as dangling pointers and memory leaks. An attractive solution to this problem is garbage collection, which frees the programmer from the task of manually examining the liveness of objects and freeing them when they are not going to be used any more. Garbage collection does that automatically at runtime. The program carries code that is able to determine if an object is useless (garbage) or not, and accordingly deallocate the memory used by it. Garbage collection is widely used in functional programming languages, for which manual memory deallocation is in general difficult.

The purpose of this diploma dissertation is the comparative study of several garbage collection algorithms, appropriate for use in a system for compiling certified binaries. We thoroughly present the algorithms and refer to the advantages and disadvantages of each one. The algorithms under study have been implemented in C for NFlint, a system for compiling certified binaries that has been developed at the Software Engineering Laboratory of NTUA.

Keywords

Garbage collection, heap, automatic memory deallocation, memory leak, mark & sweep, mark & don't sweep, mark & compact, stop & copy, semi-space, generational garbage collector.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Η αυτόματη συλλογή σκουπιδιών, ή αλλιώς αυτόματη ανάκληση μνήμης, ή απλά συλλογή σκουπιδιών είναι ένας τομέας που έχει απασχολήσει την επιστήμη των υπολογιστών από τα τέλη της δεκαετίας του 1950. Πιο συγκεκριμένα την περίοδο αυτή ο John McCarthy ήταν από τους πρώτους που ασχολήθηκαν με το θέμα αυτό, με σκοπό να επιλύσει προβλήματα στη Lisp. Με το πέρασμα του χρόνου έχει γίνει σημαντική έρευνα και πρόοδος πάνω σε τεχνικές συλλογής σκουπιδιών, ενώ ο τομέας αυτός παραμένει ενεργός μέχρι και σήμερα. Η τεχνική αυτή είναι ιδιαίτερα δημοφιλής στις συναρτησιακές γλώσσες προγραμματισμού (π.χ. Haskell, Erlang, ML) αλλά χρησιμοποιείται και σε άλλες γλώσσες υψηλού επιπέδου (π.χ. Java).

1.1 Αντικείμενο της εργασίας

Σκοπός της εργασίας αυτής είναι η υλοποίηση και παρουσίαση έξι αλγορίθμων συλλογής σκουπιδιών, οι οποίοι θα χρησιμοποιηθούν για το σύστημα πιστοποιημένων εκτελέσιμων NFlint. Αυτό αποτελείται, μεταξύ άλλων, από μία συναρτησιακή γλώσσα, στα πλαίσια του προγραμματισμού με αποδείξεις, η οποία δεν έχει ενσωματωμένο συλλέκτη σκουπιδιών, γεγονός που οδηγεί σε διαρροές μνήμης.

Επίσης γίνεται ένα συγκριτικό τεστ των αλγορίθμων αυτών σε πραγματικές συνθήκες. Πιο συγκεκριμένα καθένας από τους αλγόριθμους ενσωματώνεται σε συγκριτικά προγράμματα (benchmarks) και δοκιμάζεται στην πράξη. Μελετώνται οι απαιτήσεις του σε χώρο, καθώς και η επιβάρυνση που προκαλεί στην εκτέλεση πρόγραμμα. Τα τεστ αυτά είναι πολυάριθμα και για διαφορετικά μεγέθη του σωρού, αποσκοπώντας στην ανάδειξη του καλύτερου αλγορίθμου, ο οποίος θα χρησιμοποιηθεί από την υλοποίηση του NFlint.

1.2 Οργάνωση της εργασίας

Η εργασία αυτή αποτελείται από εννέα κεφάλαια:

- Στο **Κεφάλαιο 2** δίνεται το θεωρητικό υπόβαθρο των βασικών αρχών που σχετίζονται με τη *συλλογή σκουπιδιών*. Αρχικά περιγράφεται η χειροκίνητη διαχείριση μνήμης και τα

προβλήματα που μπορεί να προκύψουν από αυτήν. Στη συνέχεια περιγράφεται η λειτουργία της συλλογής σκουπιδιών, τα προβλήματα που επιλύει και γίνεται μια ανάλυση των διαφόρων πολιτικών συλλογής σκουπιδιών που υπάρχουν. Τέλος δίνεται μια εκτίμηση του κόστους της.

- Στο **Κεφάλαιο 3** αρχικά περιγράφεται το σύστημα πιστοποιημένου κώδικα *NFlint* και στη συνέχεια καθορίζονται οι περιορισμοί αλλά και οι απαιτήσεις όσον αφορά τη συλλογή σκουπιδιών.

- Στο **Κεφάλαιο 4** ξεκινάει η υλοποίηση της συλλογής σκουπιδιών. Δίνεται η *διεπαφή* (*interface*) της υλοποίησης της NFlint μέσω της οποίας γίνεται η επικοινωνία με τον συλλέκτη σκουπιδιών. Καθορίζεται δηλαδή η δομή που πρέπει να έχει ένας συλλέκτης σκουπιδιών για να λειτουργήσει σε αυτό το περιβάλλον.

- Στο **Κεφάλαιο 5** υλοποιούνται δύο μη–μετακινουντες (non–moving) αλγόριθμοι συλλογής σκουπιδιών: Ο αλγόριθμος *σημείωσης σάρωσης* (*mark and sweep*) και ο αλγόριθμος *σημείωσης χωρίς σάρωση* (*mark and don't sweep*).

- Στο **Κεφάλαιο 6** παρουσιάζονται δύο μετακινούντες (moving) αλγόριθμοι συλλογής σκουπιδιών: Ο αλγόριθμος *σημείωσης συμπύκνωσης* (*mark and compact*) και ο αλγόριθμος *διακοπής αντιγραφής* (*stop and copy*).

- Στο **Κεφάλαιο 7** υλοποιούνται δύο *γενετικοί* (*generational*) αλγόριθμοι συλλογής σκουπιδιών. Ο πρώτος χρησιμοποιεί τους αλγόριθμους διακοπής αντιγραφής και σημείωσης συμπύκνωσης, ενώ ο δεύτερος χρησιμοποιεί αποκλειστικά τον αλγόριθμο διακοπής αντιγραφής.

- Στο **Κεφάλαιο 8** παρουσιάζονται και αναλύονται τα αποτελέσματα και οι επιδόσεις των παραπάνω αλγορίθμων στα συγκριτικά τεστ.

- Τέλος, το **Κεφάλαιο 9** αποτελεί τον επίλογο αυτής της εργασίας, όπου παρουσιάζονται κάποια χρήσιμα συμπεράσματα.

Κεφάλαιο 2

Συλλογή Σκουπιδιών

Στην ενότητα αυτή παρουσιάζονται κάποιες βασικές έννοιες της διαχείρισης της μνήμης και ειδικότερα της συλλογής σκουπιδιών. Γίνεται μια σύντομη αναφορά στις διαφορετικές φιλοσοφίες των αλγόριθμων συλλογής σκουπιδιών. Αναλύονται οι ανάγκες που οδήγησαν στην ανάπτυξή της, τα προβλήματα που αυτή επιλύει, καθώς και το κόστος της.

2.1 Γενικά

Με τον όρο *συλλογή σκουπιδιών* (*garbage collection*) εννοούμε την αυτόματη αποδέσμευση της μνήμης που έχει εκχωρηθεί σε ένα αντικείμενο, όταν αυτό δεν πρόκειται να χρησιμοποιηθεί ξανά.

Ως *αντικείμενο* (*object/record*) ορίζεται ένα μεμονωμένο κατοχυρωμένο κομμάτι δεδομένων στο σωρό (π.χ. οποιαδήποτε δομή δεδομένων).

Ο *σωρός* (*heap*) είναι μια μεγάλη περιοχή ελεύθερης μνήμης, από την οποία γίνονται δυναμικές (κατά τον χρόνο εκτέλεσης δηλαδή) δεσμεύσεις μνήμης για τα αντικείμενα (π.χ. δομές δεδομένων) ενός προγράμματος. Με τον όρο *πρόγραμμα*[1] αναφερόμαστε σε μια συγκεκριμένη ακολουθία εντολών τις οποίες πρέπει να εκτελέσει ένας υπολογιστής για να παραγάγει το επιθυμητό για το χρήστη αποτέλεσμα.

Η *κατανομή σωρού* είναι ένας από τους τρεις τρόπους με τους οποίους η μνήμη ενός συστήματος/υπολογιστή μπορεί να κατανεμηθεί σε ένα πρόγραμμα. Οι άλλοι δύο είναι η *στατική κατανομή*, όταν δηλαδή οι διευθύνσεις μνήμης των δεδομένων ενός προγράμματος είναι γνωστές από το στάδιο της μεταγλώττισης και δεν αλλάζουν κατά την εκτέλεσή του, και η *κατανομή στοίβας* (*stack*), όπου αποθηκεύονται οι τοπικές μεταβλητές κάθε διαδικασίας και η ιεραρχία των καλούμενων υποπρογραμμάτων.

Η κατανομή στοίβας χαρακτηρίζεται από την φιλοσοφία LIFO (*Last In First Out*), δηλαδή το τελευταίο αντικείμενο που εισήχθη στη στοίβα είναι αυτό που μπορεί να προσπελαστεί πρώτο. Όταν ένα υποπρόγραμμα ολοκληρωθεί, τότε ο χώρος που χρησιμοποιούσε στη στοίβα

[1] Στη συλλογή σκουπιδιών το πρόγραμμα συχνά αναφέρεται και ως ''*μεταλλάκτης*'' (mutator), καθώς το μόνο που κάνει, από τη σκοπιά ενός συλλέκτη σκουπιδιών, είναι να μεταβάλλει τον τρόπο με τον οποίο συνδέονται μεταξύ τους τα αντικείμενα στη μνήμη.

ελευθερώνεται. Τα αντικείμενα που τυχόν έχουν δημιουργηθεί από το υποπρόγραμμα και έχουν αποθηκευτεί στη στοίβα (π.χ. τοπικές μεταβλητές) παύουν να υπάρχουν, αφού έχουν την ίδια διάρκεια ζωής με αυτό.

2.2 Διαχείριση μνήμης

Σε αντίθεση με τη στοίβα, ο σωρός δεν έχει αυτούς τους περιορισμούς. Τα αντικείμενα που βρίσκονται στο σωρό μπορούν να έχουν μεγαλύτερη διάρκεια ζωής από τα υποπρογράμματα που τα δημιούργησαν, ενώ μπορούν να προσπελαστούν με οποιαδήποτε σειρά. Η προσπέλαση αυτή είναι αναγκαίο να γίνεται με έμμεσο τρόπο, δηλαδή με την χρήση *δεικτών* (*pointers*) αφού η ακριβής διεύθυνση των αντικειμένων στην μνήμη γίνεται γνωστή μόνο κατά το χρόνο εκτέλεσης (run–time) του προγράμματος, ενώ ακόμη και για το ίδιο πρόγραμμα μπορεί να διαφέρει από εκτέλεση σε εκτέλεση.

Η διαδικασία με την οποία πραγματοποιείται *ανάθεση* (*allocation*) μνήμης σε ένα αντικείμενο (π.χ. κάποια δομή δεδομένων) είναι αρκετά απλή. Αρχικά ο προγραμματιστής ζητάει ένα τμήμα μνήμης με μέγεθος ίσο με αυτό του αντικειμένου. Η υλοποίηση της γλώσσας έχει ήδη αρχικοποιήσει ένα τμήμα της μνήμης και το χρησιμοποιεί για σωρό. Εφόσον βρεθεί ένα κομμάτι συνεχόμενης μνήμης αυτού του μεγέθους στο σωρό, τότε το κομμάτι αυτό κατοχυρώνεται και επιστρέφεται ένας δείκτης στην αρχή του. Με τον δείκτη αυτόν μπορεί πλέον το πρόγραμμα να προσπελαύνει τη νέα θέση μνήμης. Αν απαιτηθεί επιπλέον εκχώρηση μνήμης για τη δημιουργία ενός νέου αντικειμένου, τότε ο προγραμματιστής ζητάει ξανά ένα τμήμα μνήμης και η όλη διαδικασία επαναλαμβάνεται.

Γίνεται φανερή λοιπόν η ανάγκη *διαχείρισης του σωρού*, καθώς το μέγεθός του είναι περιορισμένο, και μπορεί εύκολα να εξαντληθεί αν ζητείται συνέχεια μνήμη και δεν λαμβάνεται μέριμνα για την απελευθέρωση των αντικειμένων που είναι άχρηστα (σκουπίδια).

Η ευθύνη αυτή βάραινε αρχικά τον προγραμματιστή, ο οποίος ήταν υπεύθυνος να παρακολουθεί τον κύκλο ζωής των αντικειμένων και να απελευθερώνει χειροκίνητα τη μνήμη που είχε εκχωρηθεί σε αυτά. Όπως αποδείχτηκε όμως σε αρκετές περιπτώσεις η μέθοδος αυτή (γνωστή και ως *ρητή αποδέσμευση μνήμης*) δεν ήταν αποτελεσματική, ενώ συχνά αποτελούσε πηγή σφαλμάτων (bugs). Πιο συγκεκριμένα, ο προγραμματιστής από τη μεριά του έπρεπε να γράψει περισσότερο κώδικα για τη διαχείριση του σωρού με αποτέλεσμα ο κώδικας να γίνεται δυσανάγνωστος και να δυσκολεύεται η επεκτασιμότητά του, αφού ο κώδικας ενός υποπρογράμματος ίσως απαιτούσε αλλαγές για να συνεργαστεί με κάποιο άλλο υποπρόγραμμα. Επίσης ο προγραμματιστής περνούσε αρκετό από τον χρόνο της ανάπτυξης μιας εφαρμογής στην εύρεση και στη διόρθωση σφαλμάτων, που σχετίζονταν με τη διαχείριση της μνήμης.

2.3 Πιθανά σφάλματα από τη ρητή αποδέσμευση μνήμης

Τα σφάλματα αυτά μπορούν να διακριθούν σε τρεις κατηγορίες:

- *Αιωρούμενες αναφορές* (*Dangling pointers*). Υπάρχει περίπτωση να αποδεσμευτούν τμήματα μνήμης, ενώ υπάρχουν ακόμη δείκτες που δείχνουν σε αυτά. Αν κάποιος από

αυτούς τους δείκτες χρησιμοποιηθεί στη συνέχεια για την προσπέλαση ενός τμήματος της μνήμης που έχει αποδεσμευτεί, τότε τα αποτελέσματα θα είναι απρόβλεπτα. Μπορεί το τμήμα αυτό να έχει εκχωρηθεί για κάποια άλλη χρήση εκείνη τη στιγμή και είτε τα δεδομένα του να είναι διαφορετικά, είτε να πειραχτούν δεδομένα που δεν θα έπρεπε.

- *Διαρροές μνήμης* (*Memory leaks*). Αν διαγραφούν ή αλλάξουν όλες οι αναφορές (δείκτες) σε ένα αντικείμενο, τότε το αντικείμενο θα γίνει απρόσιτο. Το πρόγραμμα δεν θα μπορεί να προσπελάσει το αντικείμενο, αφού δε θα υπάρχει κάποιος δείκτης σε αυτό. Το αντικείμενο θα έχει πλέον χαθεί και δεν υπάρχει περίπτωση απελευθέρωσής του παρά μόνο με τον τερματισμό του προγράμματος. Αν και αυτή η περίπτωση δεν επηρεάζει άμεσα την εκτέλεση του προγράμματος, μπορεί να οδηγήσει σε εξάντληση του διαθέσιμου χώρου, αφού το απρόσιτο αντικείμενο συνεχίζει να καταλαμβάνει χώρο στο σωρό.

- *Διπλή αποδέσμευση μνήμης* (*Double free bug*). Σε αυτήν την περίπτωση, επιχειρείται η αποδέσμευση τμήματος μνήμης που έχει ήδη αποδεσμευτεί. Τα αποτελέσματα μπορεί να είναι καταστροφικά αν έχει τύχει το συγκεκριμένο τμήμα μνήμης να έχει εκχωρηθεί εκ νέου μεταξύ των δύο αποδεσμεύσεων.

Τέτοια σφάλματα είναι δύσκολο να αποφευχθούν και ακόμη πιο δύσκολο να εντοπιστούν. Για παράδειγμα μπορεί μια αιωρούμενη αναφορά να περάσει απαρατήρητη αν το τμήμα της μνήμης στο οποίο δείχνει δεν έχει ανατεθεί σε καινούριο αντικείμενο. Ή μπορεί οι διαρροές μνήμης που έχει ένα πρόγραμμα να είναι μικρές και να μην εξαντληθεί η μνήμη. Αν όμως το ίδιο πρόγραμμα τρέξει σε υπολογιστή με λιγότερη μνήμη, τότε αυτή, λόγω των διαρροών, ίσως να μην επαρκεί για να ολοκληρωθεί η εκτέλεση του προγράμματος. Όλα αυτά οδηγούν στο συμπέρασμα πως ο κώδικας που χρειάζεται για την δουλειά αυτή δυσκολεύει την υλοποίηση, την συντήρηση, ακόμα και τη σχεδίαση των προγραμμάτων.

2.4 Αυτόματη διαχείριση μνήμης

Η *συλλογή σκουπιδιών* επιχειρεί να δώσει μια λύση στα προβλήματα αυτά, απαλλάσσοντας τον προγραμματιστή από το καθήκον της χειροκίνητης αποδέσμευσης μνήμης. Στην ιδανική περίπτωση, ένας *συλλέκτης*, δηλαδή ο κώδικας του προγράμματος που πραγματοποιεί την συλλογή των σκουπιδιών, θα πρέπει να είναι σε θέση να εντοπίζει όλα τα άχρηστα αντικείμενα στο σωρό και να ανακτά την μνήμη που έχει εκχωρηθεί σε αυτά.

Στην πραγματικότητα όμως, δεν είναι εύκολο για τον συλλέκτη να εντοπίσει όλα τα άχρηστα αντικείμενα. Αυτό συμβαίνει επειδή δεν είναι πάντα σαφές πότε ένα αντικείμενο είναι *ζωντανό* (*live*) ή *σκουπίδι* (*garbage*). Η απλούστερη απάντηση για το πότε ένα αντικείμενο είναι ζωντανό, είναι όταν υπάρχουν δείκτες προς αυτό. Με άλλα λόγια, όταν ένα αντικείμενο δεν έχει δείκτες προς αυτό, έχει γίνει δηλαδή *απρόσιτο*, τότε είναι σίγουρα σκουπίδι. Το πρόβλημα με την προσέγγιση αυτή, είναι ότι στην περίπτωση κατά την οποία δύο ή περισσότερα αντικείμενα συνδέονται κυκλικά μεταξύ τους, αλλά η κυκλική αυτή λίστα αντικειμένων δεν

είναι προσιτή από κανέναν άλλο δείκτη στο πρόγραμμα, τα αντικείμενα αυτά, αν και σκουπίδια, δεν μπορούν να συλλεχθούν, αφού υπάρχει δείκτης προς καθένα από αυτά. Ένας πιο πλήρης ορισμός, του πότε ένα αντικείμενο είναι ζωντανό, είναι όταν το αντικείμενο αυτό μπορεί να προσπελαστεί ακολουθώντας μια ''αλυσίδα'' έγκυρων δεικτών στο σωρό η οποία ξεκινάει από τις λεγόμενες *ρίζες* (*roots*) του προγράμματος. Οι ρίζες αυτές πρέπει να βρίσκονται έξω από τον σωρό. Ειδικότερα, ως *ρίζες* ορίζονται οι τιμές εκείνες στις οποίες έχει άμεση πρόσβαση ένα πρόγραμμα. Οι τιμές αυτές βρίσκονται στους καταχωρητές (registers), στη στοίβα του προγράμματος (π.χ. προσωρινές μεταβλητές) και στις ολικές (global) μεταβλητές. Προφανώς τα υπόλοιπα αντικείμενα, τα οποία δεν μπορούν να προσπελαστούν από καμία από τις παραπάνω αλυσίδες δεικτών, είναι σκουπίδια.

Υπάρχουν όμως περιπτώσεις, στις οποίες δείκτες που βρίσκονται στις ρίζες, δείχνουν σε αντικείμενα που δεν πρόκειται να ξαναχρησιμοποιηθούν. Τέτοιοι δείκτες είτε έχουν ξεχαστεί από τον προγραμματιστή, είτε, για λόγους εξοικονόμησης χρόνου, δεν έχουν καθαριστεί από τον μεταγλωττιστή (π.χ. μετά από το τέλος μιας διαδικασίας). Ακόμη υπάρχει η πιθανότητα μια λέξη στο σωρό ή στις ρίζες, η οποία δεν είναι έγκυρος δείκτης, να τύχει να έχει μια τιμή που να μοιάζει με δείκτη στο σωρό.

2.5 Απαιτήσεις από την υλοποίηση της γλώσσας

Έτσι λοιπόν για να μπορέσει να λειτουργήσει η συλλογή σκουπιδιών υπάρχουν κάποιες απαιτήσεις, τις οποίες οφείλει να ικανοποιεί ο μεταγλωττιστής (compiler) ή η γλώσσα[DMH92].

Πρώτον πρέπει όλα τα ζωντανά/χρήσιμα αντικείμενα στο σωρό να είναι ανά πάσα στιγμή *προσιτά* (*reachable*). Δηλαδή να υπάρχουν πάντα δείκτες σε αυτά. Αυτή η απαίτηση απαγορεύει στον προγραμματιστή να ''κρύβει'' τους δείκτες και να αποκαθιστά μετά την τιμή τους. Για παράδειγμα αν ο προγραμματιστής πάρει την αποκλειστική διάζευξη (xor) ενός δείκτη με έναν αριθμό (αριθμητική δεικτών), ή αν αποθηκεύσει έναν δείκτη σε κάποιο αρχείο και μετά τον διαγράψει, με σκοπό, και στις δύο περιπτώσεις, να αποκαταστήσει την τιμή του δείκτη αργότερα, τότε ο συλλέκτης σκουπιδιών μη βρίσκοντας δείκτη προς το αντικείμενο αυτό θα το θεωρήσει άχρηστο και θα αποδεσμεύσει τη μνήμη του. Όταν στη συνέχεια ο προγραμματιστής αποκαταστήσει την τιμή του δείκτη, τότε αυτός θα δείχνει σε τμήμα μνήμης που έχει αποδεσμευτεί (αιωρούμενη αναφορά).

Δεύτερον θα πρέπει το πρόγραμμα να μην επιτρέπει δείκτες σε τυχαίες θέσεις μνήμης, ακόμα και αν αυτές οι θέσεις μνήμης βρίσκονται εντός της δικαιοδοσίας του. Θα πρέπει το πρόγραμμα να χρησιμοποιεί μόνο τους δείκτες που έχουν επιστραφεί από την κλήση για ανάθεση νέας μνήμης, οι οποίοι (συνήθως) δείχνουν στην αρχή του τμήματος της μνήμης που κατοχυρώθηκε. Αυτό είναι απαραίτητο γιατί συχνά οι συλλέκτες σκουπιδιών χρησιμοποιούν επιπλέον βοηθητικά πεδία στα τμήματα μνήμης που κατοχυρώνονται κατά την ανάθεση μνήμης και μόνο με τους δείκτες αυτούς μπορούν να τα προσπελάσουν.

Τρίτον πρέπει ο μεταγλωττιστής να ελαχιστοποιεί κατά το δυνατόν τους δείκτες σε άχρηστα αντικείμενα. Εδώ χρειάζεται και η συμβολή του προγραμματιστή, καθώς ο μεταγλωττιστής δεν μπορεί πάντα να ξέρει πότε ένα αντικείμενο είναι άχρηστο. Σε διαφορετική περίπτωση

πρέπει να αποδεχτούμε το γεγονός ότι κάποια σκουπίδια θα περάσουν απαρατήρητα, αφού θα υπάρχουν δείκτες σε αυτά.

Τέταρτον πρέπει ο μεταγλωττιστής να διαχωρίζει τους δείκτες από τις τιμές του προγράμματος, έτσι ώστε να εξαλειφθεί η πιθανότητα μια λέξη στο σωρό να μοιάζει με έγκυρο δείκτη. Στην περίπτωση αυτή η συλλογή σκουπιδιών λέγεται *ακριβής*[2] (*precise*).

Πέμπτον η εκχώρηση επιπλέον μνήμης θα πρέπει να γίνεται από τον ίδιο τον συλλέκτη σκουπιδιών και όχι, λόγου χάρη, από κάποια κλήση συστήματος, καθώς μόνο ο συλλέκτης μπορεί να ξέρει ποια τμήματα του σωρού έχουν ελευθερωθεί από προηγούμενες συλλογές. Αυτό μπορεί να γίνει είτε άμεσα, δηλαδή ο προγραμματιστής να αντικαταστήσει οποιαδήποτε κλήση για ανάθεση μνήμης με μία κλήση στην σχετική συνάρτηση ανάθεσης μνήμης του συλλέκτη, διαγράφοντας παράλληλα τις τυχόν κλήσεις για ρητή αποδέσμευση μνήμης, είτε έμμεσα από τον μεταγλωττιστή. Άμεση συνέπεια του προηγουμένου είναι ότι κάθε αλγόριθμος συλλογής σκουπιδιών πρέπει να έχει ένα τμήμα κώδικα (συνήθως μία συνάρτηση) που να παρακολουθεί και να εκχωρεί την ελεύθερη μνήμη στο σωρό, όταν αυτή ζητηθεί (*allocator*).

Έκτον μπορεί, ανάλογα με το είδος του αλγόριθμου[3] συλλογής σκουπιδιών που πρόκειται να υλοποιηθεί, να χρειάζονται κάποιες επιπλέον μετατροπές στον κώδικα που παράγει ο μεταγλωττιστής, όπως για παράδειγμα να εκτελούνται κάποιες συγκεκριμένες λειτουργίες σε ενημερώσεις – διαγραφές δεικτών ή να διασφαλίζεται το αναλλοίωτο της κατάστασης του σωρού όση ώρα θα εκτελείται ένα κομμάτι του αλγορίθμου.

2.6 Πολιτικές συλλογής σκουπιδιών

Με την ικανοποίηση των παραπάνω απαιτήσεων, ο συλλέκτης σκουπιδιών είναι σε θέση να εκχωρεί για τη δημιουργία νέων αντικειμένων τμήματα από την διαθέσιμη μνήμη, όταν του ζητηθεί, αλλά και να διαχωρίζει τα χρήσιμα αντικείμενα του προγράμματος από τα άχρηστα.

2.6.1 Τρόποι εντοπισμού των σκουπιδιών

Οι τρόποι με τους οποίους εντοπίζονται τα σκουπίδια στο σωρό είναι δύο. Ο πρώτος τρόπος είναι ο έμμεσος τρόπος, γνωστός και ως *ανιχνευτικός* (*tracing*). Τα σκουπίδια δεν συλλέγονται τη στιγμή που δημιουργούνται, αλλά συσσωρεύονται στο σωρό. Η συλλογή των σκουπιδιών στους συλλέκτες που ακολουθούν αυτήν την προσέγγιση γίνεται όταν η διαθέσιμη μνήμη εξαντλείται ή κινδυνεύει να εξαντληθεί. Κατά την κλήση ενός τέτοιου συλλέκτη σκουπιδιών αρχικά εξετάζονται οι ρίζες του προγράμματος για έγκυρους δείκτες στο σωρό. Αφού βρεθεί ότι κάποια από τις ρίζες του προγράμματος είναι δείκτης σε κάποιο αντικείμενο στο σωρό, τότε εξετάζεται το αντικείμενο αυτό για δείκτες στο σωρό και αναδρομικά ή επαναληπτικά εξετάζονται όλα τα αντικείμενα στο σωρό που έχουν άλλα αντικείμενα να δείχνουν σε αυτά.

[2] Υπάρχει και η συντηρητική (conservative) συλλογή σύμφωνα με την οποία, ό,τι δείχνει στον σωρό θεωρείται έγκυρος δείκτης και η θέση μνήμης στην οποία δείχνει διατηρείται από τον συλλέκτη, αν και μπορεί να είναι σκουπίδι. Αυτό μπορεί σε κάποιες περιπτώσεις να προκαλέσει προβλήματα και συχνά αποφεύγεται.

[3] Συνήθως μόνο οι αυξητικοί και οι ταυτόχρονοι (βλ. Πολιτικές συλλογής σκουπιδιών) συλλέκτες έχουν τέτοιες απαιτήσεις.

Αυτό γίνεται για όλες τις ρίζες. Έτσι έχουμε αλυσίδες χρήσιμων αντικειμένων οι οποίες ξεκινάνε πάντα από κάποια ρίζα του προγράμματος. Όλα τα άλλα αντικείμενα, αυτά δηλαδή που δεν ανήκουν σε κάποια αλυσίδα, είναι σκουπίδια και πρέπει να ανακτηθούν. Η διαδικασία ανάκτησης της μνήμης που αυτά καταλαμβάνουν, πρέπει να ακολουθήσει άμεσα προκειμένου να ελευθερωθεί χώρος για να συνεχίσουν να ικανοποιούνται οι απαιτήσεις του προγράμματος για ανάθεση μνήμης. Βασικότερο πλεονέκτημα του τρόπου αυτού είναι ότι μπορεί να χειριστεί τις κυκλικές δομές δεδομένων. Από την άλλη στα μειονεκτήματά του συγκαταλέγεται το κόστος σε χρόνο που απαιτείται για να προσδιοριστούν τα ζωντανά αντικείμενα και να συλλεχθούν τα σκουπίδια, το οποίο μπορεί να οδηγήσει σε σύντομες παύσεις, καθώς κατά το στάδιο αυτό διακόπτεται η εκτέλεση του προγράμματος.

Ο δεύτερος τρόπος είναι πιο άμεσος από τον πρώτο και συλλέγει τα σκουπίδια τη στιγμή που αυτά δημιουργούνται. Απαιτεί να καταγράφονται σε κάθε αντικείμενο στο σωρό όλες οι αναφορές στο αντικείμενο αυτό, τόσο από άλλα αντικείμενα στο σωρό, όσο και από τις ρίζες του προγράμματος[Col60]. Με αυτήν την *καταμέτρηση των αναφορών* (*reference counting*) είναι δυνατόν να εξεταστεί αν το αντικείμενο είναι ζωντανό ή όχι. Αν ο μετρητής αναφορών ενός αντικειμένου γίνει μηδέν, τότε δεν υπάρχουν αναφορές σε αυτό. Το αντικείμενο είναι πλέον απρόσιτο και άρα αποτελεί σκουπίδι. Επιπλέον πρέπει να μειωθεί κατά ένα και ο μετρητής αναφορών όλων των αντικειμένων στα οποία αυτό έδειχνε. Αυτή η προσέγγιση απαιτεί από τον μεταγλωττιστή να ενημερώνει τους μετρητές αναφορών για κάθε αλλαγή στους δείκτες. Για παράδειγμα αν ένας δείκτης έδειχνε σε ένα αντικείμενο και στη συνέχεια δείχνει σε ένα άλλο, τότε κατά την αλλαγή της τιμής του δείκτη πρέπει αντίστοιχα να μειωθεί κατά ένα ο μετρητής αναφορών του πρώτου αντικειμένου και να αυξηθεί κατά ένα ο μετρητής αναφορών του δεύτερου. Κάθε αντικείμενο του οποίου ο μετρητής αναφορών μηδενίστηκε, μπορεί να ανακτηθεί εκείνη τη στιγμή. Δεν χρειάζεται κάποια κλήση του συλλέκτη σκουπιδιών. Η συλλογή στην ουσία παρεμβάλλεται στην εκτέλεση του προγράμματος. Πρόκειται δηλαδή και για αυξητικό αλγόριθμο συλλογής, όπως θα δούμε παρακάτω. Βασικότερο μειονέκτημα του τρόπου αυτού είναι η αδυναμία του να χειριστεί κυκλικές δομές δεδομένων[McB63]. Σε αντίθεση όμως με τον πρώτο τρόπο, το κόστος για την ενημέρωση των μετρητών αναφοράς αλλά και για την ανάκτηση των σκουπιδιών μοιράζεται ομοιόμορφα στο πρόγραμμα και προκαλεί πολύ μικρότερες παύσεις στην εκτέλεση του προγράμματος. Παρόλα αυτά, πολλοί προγραμματιστές αποφεύγουν αυτόν τον τρόπο συλλογής σκουπιδιών γιατί, πέρα από την αδυναμία του στο χειρισμό κυκλικών δομών δεδομένων, εξαρτάται άμεσα από την υλοποίηση του μεταγλωττιστή για την συνεχή ενημέρωση του μετρητή αναφορών σε κάθε αντικείμενο. Επιπροσθέτως το κόστος για τις ενημερώσεις – ανακτήσεις αυτές, αν και ισοκατανέμεται στο πρόγραμμα, παραμένει αρκετά υψηλό.

2.6.2 Τρόποι συλλογής των σκουπιδιών

Εκτός από άμεσους και έμμεσους, οι συλλέκτες σκουπιδιών μπορούν να κατηγοριοποιηθούν ανάλογα με την πολιτική που εφαρμόζουν για την συλλογή των σκουπιδιών (αφού δηλαδή έχει καθοριστεί ποια αντικείμενα είναι ζωντανά και ποια όχι). Οι κατηγορίες αυτές είναι δύο.

Στην πρώτη ανήκουν οι *μη–μετακινούντες* (*non–moving*) συλλέκτες, δηλαδή εκείνοι που δεν μετακινούν κανένα αντικείμενο. Οι συλλέκτες αυτοί απλά αποδεσμεύουν μόνο τα άχρηστα αντικείμενα στο σωρό, ενώ αφήνουν τα ζωντανά αντικείμενα στη θέση τους ως έχουν. Η θέση που καταλάμβανε ένα σκουπίδι και που πλέον έχει ελευθερωθεί πρέπει να καταγράφεται, έτσι ώστε να χρησιμοποιηθεί για κάποια μελλοντική ανάθεση μνήμης. Η ανάθεση αυτή μπορεί να γίνει με δύο τρόπους: Είτε εκχωρώντας το πρώτο επαρκές σε μέγεθος κομμάτι ελεύθερης μνήμης που θα βρεθεί (first–fit), είτε ψάχνοντας για ένα κομμάτι που να έχει το ίδιο, ή όσο το δυνατόν πλησιέστερο μέγεθος με αυτό που έχει ζητηθεί (best–fit).

Στη δεύτερη ανήκουν οι *μετακινούντες* (*moving*) συλλέκτες, οι οποίοι μετακινούν ή αντιγράφουν τα ζωντανά αντικείμενα σε μια περιοχή μνήμης. Τα ζωντανά αντικείμενα μετακινούνται ή αντιγράφονται το ένα μετά το άλλο χωρίς κενά μεταξύ τους, δημιουργώντας έναν συμπυκνωμένο χώρο ζωντανών αντικειμένων στο σωρό. Σε μια τέτοια περίπτωση είναι αναγκαία η ενημέρωση των δεικτών, οι οποίοι δείχνουν στα αντικείμενα που μετακινήθηκαν. Λόγω του κόστους της ενημέρωσης αυτής, αλλά και λόγω της επιβάρυνσης της μετακίνησης ή της αντιγραφής των αντικειμένων, οι συλλέκτες αυτοί φαίνεται να είναι ιδιαίτερα χρονοβόροι σε σύγκριση με τους συλλέκτες που δεν μετακινούν τίποτα. Στην πραγματικότητα όμως οι συλλέκτες αυτοί παρουσιάζουν πλεονεκτήματα τόσο στη φάση της συλλογής σκουπιδιών, όσο και στην εκτέλεση του προγράμματος γενικότερα.

Πιο συγκεκριμένα δεν απαιτούν καμία ενέργεια για την αποδέσμευσή του χώρου που καταλαμβάνουν τα άχρηστα αντικείμενα. Αφού όλα τα αντικείμενα έχουν μετακινηθεί, τότε ολόκληρος ο χώρος που απομένει, ο οποίος είναι συνεχόμενος, μπορεί να θεωρείται ελεύθερος. Αντίθετα οι μη–μετακινούντες συλλέκτες θα πρέπει με κάποιο τρόπο να επισκέπτονται κάθε σκουπίδι και να καταγράφουν τη θέση του και τη μνήμη που αυτό καταλαμβάνει.

Επίσης για τον ίδιο λόγο, η ανάθεση μνήμης γίνεται πολύ εύκολα και γρήγορα (απλά αυξάνοντας έναν δείκτη στην αρχή του ελεύθερου χώρου κάθε φορά), επειδή ο ελεύθερος χώρος στο σωρό είναι συνεχόμενος, σε αντίθεση με τους μη–μετακινουντες συλλέκτες οι οποίοι για την ανάθεση απαιτείται να εξετάζουν πίνακες ή λίστες που περιέχουν τα τμήματα της μνήμης που είναι ελεύθερα, μέχρι να βρουν ένα με το κατάλληλο μέγεθος. Άμεση συνέπεια των παραπάνω είναι ότι οι μη–μετακινούντες συλλέκτες, αν εκτελούνται για αρκετή ώρα, αφήνουν τον σωρό *κατακερματισμένο* (*fragmented*). Δημιουργούνται δηλαδή μικρά τμήματα ελεύθερης μνήμης διασκορπισμένα στο σωρό, τα οποία περιβάλλονται από ζωντανά τμήματα μνήμης. Αυτό έχει σαν αποτέλεσμα, κυρίως στα προγράμματα με αντικείμενα μεταβλητού μήκους, ο συνολικός ελεύθερος χώρος στο σωρό να επαρκεί για την ανάθεση μνήμης σε ένα αντικείμενο σχετικά μεγάλου μεγέθους, αλλά να μην υπάρχει επιμέρους ελεύθερο κομμάτι αυτού του μεγέθους. Οι μετακινούντες συλλέκτες σκουπιδιών έχουν λύσει αυτό το πρόβλημα, αφού εφαρμόζουν *συμπύκνωση* των αντικειμένων.

Ακόμη, ανάλογα με τον τρόπο που γίνεται η πρόσβαση στα ζωντανά αντικείμενα τα οποία μετακινούνται, είναι δυνατόν ένα αντικείμενο που δείχνει σε ένα άλλο να τοποθετηθεί ‘‘κοντά’’ του, αυξάνοντας έτσι την *τοπικότητα*[4] (*locality*) των αντικειμένων, σε αντίθεση με τους μη–

[4]Στα συστήματα με κρυφή μνήμη (cache) ή και με εικονική (virtual) μνήμη, η αύξηση της τοπικότητας των αντικειμένων οδηγεί σε σημαντική μείωση στον χρόνο προσπέλασης των αντικειμένων αυτών.

μετακινούντες συλλέκτες, οι οποίοι αφήνουν τα αντικείμενα διασκορπισμένα στο σωρό.

2.6.3 Αλληλεπίδραση με το κυρίως πρόγραμμα

Οι συλλέκτες σκουπιδιών μπορούν τέλος να διαχωριστούν ανάλογα με τον τρόπο εκτέλεσής τους και την αλληλεπίδρασή τους με το πρόγραμμα. Οι βασικές κατηγορίες στις οποίες χωρίζονται οι συλλέκτες είναι τρεις:

Η πρώτη κατηγορία περιλαμβάνει τους συλλέκτες που ''*σταματούν τον κόσμο*'' (*stop–the–world*). Αυτοί οι συλλέκτες σταματούν προσωρινά την εκτέλεση του προγράμματος για να καθορίσουν ποια από τα αντικείμενα του σωρού είναι σκουπίδια και στη συνέχεια τα καθαρίζουν. Μόλις τελειώσει η διαδικασία της συλλογής, το πρόγραμμα συνεχίζει τη λειτουργία του. Η φιλοσοφία αυτή εισάγει μεγάλες παύσεις στην εκτέλεση του προγράμματος, οι οποίες παρουσιάζονται απρόοπτα και δεν είναι επιθυμητές σε διαδραστικά προγράμματα ή προγράμματα πραγματικού χρόνου.

Την δεύτερη κατηγορία αποτελούν οι *αυξητικοί* (*incremental*) συλλέκτες. Οι συλλέκτες αυτοί προσπαθούν να μειώσουν το χρόνο που σταματάει η εκτέλεση του προγράμματος. Αυτό το πετυχαίνουν διαχωρίζοντας τις διαδικασίες που πρέπει να εκτελεστούν σε διακριτά στάδια και παρεμβάλλοντας εν συνεχεία τα στάδια αυτά σε διαφορετικά σημεία του προγράμματος.

Στην τρίτη κατηγορία ανήκουν οι *παράλληλοι ή ταυτόχρονοι* (*parallel* or *concurrent*) συλλέκτες, οι οποίοι εκτελούνται παράλληλα με το πρόγραμμα σε διαφορετικά νήματα εκτέλεσης, εκμεταλλευόμενοι τα συστήματα με περισσότερους του ενός επεξεργαστές. Οι συλλέκτες αυτοί προκαλούν τις μικρότερες δυνατές παύσεις στην εκτέλεση του προγράμματος. Οι παύσεις αυτές συνήθως γίνονται όταν εξετάζονται οι ρίζες του προγράμματος για δείκτες στο σωρό.

2.6.4 Συνδυαστικές τεχνικές

Οι *συμβατικοί* (*conventional*) συλλέκτες σκουπιδιών χρησιμοποιούν έναν αλγόριθμο σε όλη τη διάρκεια εκτέλεσης του προγράμματος. Σε κάθε κύκλο συλλέγονται όλα τα σκουπίδια στο σωρό. Η αποδοτικότητα ενός τέτοιου συλλέκτη ταυτίζεται με την αποδοτικότητα του αλγορίθμου που χρησιμοποιεί.

Οι *υβριδικοί* (*hybrid*) συλλέκτες σκουπιδιών συνδυάζουν δύο ή και περισσότερους αλγορίθμους για τη συλλογή των σκουπιδιών. Ο συνδυασμός αυτός γίνεται στην προσπάθεια να βελτιωθεί η αποδοτικότητα του συλλέκτη. Η ιδέα είναι να καλυφθούν τα μειονεκτήματα του ενός αλγορίθμου, από τα πλεονεκτήματα του άλλου.

Τέλος οι *γενετικοί* (*generational*) συλλέκτες σκουπιδιών χωρίζουν τον σωρό σε τμήματα (γενιές) και σε κάθε κύκλο επικεντρώνονται στη συλλογή των σκουπιδιών όχι σε όλο το σωρό, αλλά σε ένα τμήμα του, το οποίο περιέχει τα πιο πρόσφατα αντικείμενα, καθώς σύμφωνα με μελέτες αυτά είναι περισσότερο πιθανό να καταλήξουν σκουπίδια. Περιοδικά ελέγχεται και ολόκληρος ο σωρός. Συχνά οι συλλέκτες αυτοί χρησιμοποιούν περισσότερους από έναν αλγόριθμους συλλογής σκουπιδιών. Προκαλούν, τον περισσότερο καιρό, μικρότερες παύσεις στην εκτέλεση του προγράμματος, καθώς συλλέγουν συχνότερα μόνο τα μικρότερα τμήματα του σωρού.

2.7 Κόστος της συλλογής σκουπιδιών

Αφού παρουσιάστηκαν όλες οι πολιτικές συλλογής σκουπιδιών, κρίνεται χρήσιμο να γίνει μια σύντομη αναφορά και στο κόστος της συλλογής σκουπιδιών.

Ανεξάρτητα από την πολιτική που ακολουθείται, η συλλογή σκουπιδιών προσθέτει μία επιβάρυνση στο πρόγραμμα, στην προσπάθειά της να καθορίσει ποια αντικείμενα χρειάζονται και ποια όχι και στη συνέχεια να καθαρίσει τα σκουπίδια. Το κόστος αυτό δεν είναι εύκολο να υπολογιστεί στη θεωρία, αφού εξαρτάται από πολλούς παράγοντες. Για παράδειγμα, δεν μπορούμε να ξέρουμε πότε θα χρειαστεί να επέμβει ο συλλέκτης σκουπιδιών. Ανάλογα με το πρόγραμμα, ή την είσοδο του προγράμματος, ο συλλέκτης μπορεί να κληθεί σε διαφορετικά σημεία από εκτέλεση σε εκτέλεση, με αποτέλεσμα να μην γνωρίζουμε σε τι κατάσταση θα είναι ο σωρός εκείνη τη στιγμή, πόσα αντικείμενα θα είναι ζωντανά και τον τρόπο που αυτά θα είναι συνδεδεμένα μεταξύ τους. Με την ίδια λογική δεν μπορούμε να ξέρουμε εκ των προτέρων ούτε πόσες φορές θα εκτελεστεί ο συλλέκτης.

Σημαντικό ρόλο στο κόστος παίζει και η φιλοσοφία του συλλέκτη σκουπιδιών. Είδαμε ότι οι μετακινούντες αλγόριθμοι έχουν γενικά μικρότερο κόστος σε χρόνο από τους μη–μετακινούντες. Οι συλλέκτες που ''σταματούν τον κόσμο'' οδηγούν σε τυχαίες παύσεις της εκτέλεσης του προγράμματος, γεγονός που τους καθιστά ακατάλληλους για διαδραστικά ή προγράμματα πραγματικού χρόνου. Από την άλλη μεριά η ολική τους επιβάρυνση σε χρόνο είναι συνήθως μικρότερη από τη συνολική επιβάρυνση των αυξητικών και ταυτόχρονων συλλεκτών με αποτέλεσμα να είναι ιδανικοί για μη διαδραστικές εφαρμογές, στις οποίες ενδιαφερόμαστε μόνο για τον συνολικό χρόνο που θα χρειαστεί το πρόγραμμα για να ολοκληρώσει την εκτέλεσή του. Άλλοι αλγόριθμοι (π.χ. μετρητές αναφορών) εισάγουν επιπλέον κόστος σε λειτουργίες του προγράμματος όπως η ανάγνωση – εγγραφή δεικτών.

Η πληρότητα του σωρού, δηλαδή το ποσοστό του σωρού που καταλαμβάνουν τα ζωντανά αντικείμενα, μπορεί να επηρεάσει την αποδοτικότητα των διαφόρων συλλεκτών. Όπως θα δούμε και στη συνέχεια πολλοί από αυτούς έχουν κόστος ανάλογο των ζωντανών αντικειμένων στο σωρό. Είναι επιθυμητό να γνωρίζει ο προγραμματιστής πως μεταβάλλονται οι επιδόσεις των συλλεκτών για διάφορα ποσοστά πληρότητας του σωρού.

Επίσης στο κόστος της συλλογής σκουπιδιών πρέπει να συνυπολογιστεί και το κόστος της ανάθεσης μνήμης. Όταν ο ελεύθερος χώρος είναι συνεχής τότε είναι πολύ γρηγορότερη η διαδικασία της ανάθεσης μνήμης, σε σχέση με την ανάθεση μνήμης σε κατακερματισμένο σωρό, όπου πρέπει να εντοπίζονται κάθε φορά τα ελεύθερα κομμάτια μνήμης. Η πολιτική ανάθεσης μνήμης στους μη–μετακινούντες αλγόριθμους (first–fit ή best–fit) έχει κι αυτή τη δική της επίπτωση στην απόδοση του προγράμματος τόσο σε χρόνο, αλλά και σε χώρο. Ένας συλλέκτης που είναι αποτελεσματικός στη συλλογή, αλλά έχει υψηλό κόστος ανάθεσης δεν μπορεί να θεωρηθεί αποδοτικός.

Δεν είναι όμως μόνο το κόστος σε χρόνο. Υπάρχει και το κόστος σε χώρο, το οποίο είναι και αυτό με τη σειρά του πολύ σημαντικό. Μπορεί ο εκάστοτε συλλέκτης σκουπιδιών να χρειάζεται επιπλέον μνήμη για να μπορέσει να πραγματοποιήσει τις λειτουργίες του, όπως για παράδειγμα επιπλέον χώρο στο σωρό για να μετακινήσει κάποια αντικείμενα, ή στοίβα

προκειμένου να προσπελάσει αναδρομικά κάποια αντικείμενα. Οι απαιτήσεις σε χώρο ενός συλλέκτη είναι πιο εύκολο να μελετηθούν ενώ συχνά γίνονται συμβιβασμοί μεταξύ χώρου και χρόνου. Δηλαδή μπορεί ένας συλλέκτης να είναι ταχύτερος από έναν άλλον, αλλά να χρειάζεται περισσότερη μνήμη για να δουλέψει, όπως συμβαίνει για παράδειγμα με τους μετακινούντες συλλέκτες η οποίοι αν και είναι ταχύτεροι συνήθως απαιτούν περισσότερο χώρο από τους μη–μετακινούντες.

Τέλος, στα παραπάνω κόστη πρέπει να συμπεριληφθεί και το κόστος υλοποίησης του αλγορίθμου συλλογής σκουπιδιών. Ανεξάρτητα από την πολιτική που θα επιλεχθεί, θα πρέπει να γραφεί κώδικας τόσο για τη φάση της συλλογής των άχρηστων αντικειμένων, όσο και για τη φάση της ανάθεσης μνήμης. Ο κώδικας αυτός θα αποτελεί μέρος του προγράμματος και θα πρέπει και αυτός να ελεγχθεί για τυχόν λάθη. Τα τελευταία χρόνια αυτό γίνεται πιο δύσκολο, καθώς επινοούνται ολοένα και πιο περίπλοκοι αλγόριθμοι συλλογής σκουπιδιών. Μπορεί ακόμη να απαιτούνται αλλαγές στην υλοποίηση της γλώσσας ή στον μεταγλωττιστή προκειμένου να δουλέψει η συλλογή σκουπιδιών. Η μεταφερσιμότητα του συλλέκτη σκουπιδιών, η δυνατότητα δηλαδή να χρησιμοποιηθεί ο ίδιος συλλέκτης σε συστήματα με άλλη αρχιτεκτονική, σε διαφορετικούς μεταγλωττιστές και σε διαφορετικά προγράμματα, είναι άλλος ένας παράγοντας που επηρεάζει το κόστος υλοποίησης και πρέπει να απασχολήσει τον προγραμματιστή.

Δυστυχώς όλα τα παραπάνω δεν αποτελούν ανεξάρτητες παραμέτρους και συχνά ο προγραμματιστής θα κληθεί να θυσιάσει κάποια από αυτά για να επιτύχει άλλα, ανάλογα με τις προτεραιότητες κάθε προγράμματος/εφαρμογής.

2.8 Συμπεράσματα

Η συλλογή σκουπιδιών αποτελεί ένα χρήσιμο εργαλείο στα χέρια του προγραμματιστή, ο οποίος πρέπει να επιλέξει ποια πολιτική συλλογής σκουπιδιών ταιριάζει καλύτερα στην εφαρμογή που θέλει να αναπτύξει. Μπορεί όμως και να επιλέξει να χειριστεί τη μνήμη χειροκίνητα. Η συλλογή σκουπιδιών δεν είναι υποχρεωτική, ούτε είναι ταχύτερη από την ρητή διαχείριση της μνήμης, απλά σε πολλές περιπτώσεις προσφέρει μια διευκόλυνση στον προγραμματιστή. Μελέτες έχουν δείξει ότι όταν η ρητή αποδέσμευση μνήμης υλοποιείται σωστά από τον προγραμματιστή, είναι, τις περισσότερες φορές, ταχύτερη από κάθε είδους συλλέκτη σκουπιδιών. Είναι επίσης πιο προβλέψιμη στη λειτουργία της και συνεπώς πιο εύκολο να καθοριστεί το κόστος της. Μικρά προγράμματα καθώς και προγράμματα με ξεκάθαρες απαιτήσεις σε διαχείριση μνήμης μπορούν να υλοποιηθούν με μικρότερο υπολογιστικό κόστος αν χρησιμοποιηθεί ρητή αποδέσμευσή μνήμης. Ο προγραμματιστής είναι αυτός που τελικά θα κρίνει αν είναι απαραίτητη η συλλογή σκουπιδιών καθώς και τον κατάλληλο αλγόριθμο συλλογής σκουπιδιών, ή αν θα είναι προτιμότερο να διαχειριστεί ο ίδιος τη μνήμη χειροκίνητα, κάνοντας πιθανώς κάποιους συμβιβασμούς και λαμβάνοντας κάθε φορά υπ' όψιν του τις απαιτήσεις αλλά και τους περιορισμούς που θα έχει η εφαρμογή.

Κεφάλαιο 3

Πιστοποιημένος Κώδικας

Ένα *πιστοποιημένο εκτελέσιμο* (*certified binary*) δίνει μία τιμή μαζί με μία απόδειξη, ότι η τιμή αυτή ικανοποιεί κάποιες συγκεκριμένες προδιαγραφές[SSTP02]. Με άλλα λόγια κάθε εκτελέσιμο φέρει, πέρα από τον κώδικά του και μια σειρά από αποδείξεις της ασφάλειας και της μερικής ορθότητάς του. Η ενσωματωμένη απόδειξη ελέγχεται με μηχανιστικό και αποδοτικό τρόπο πριν την εκτέλεση του πιστοποιημένου εκτελέσιμου, από κάποιο αξιόπιστο εργαλείο απόδειξης θεωρημάτων, και εφόσον βρεθεί να είναι έγκυρη (valid), τότε το εκτελέσιμο είναι, πέρα από κάθε αμφιβολία, συνεπές και μπορεί να εκτελεστεί χωρίς να χρειάζονται επιπλέον έλεγχοι κατά το χρόνο εκτέλεσής, που να διασφαλίζουν την ορθότητα και την ασφάλειά του, οι οποίοι θα πρόσθεταν μια επιπλέον επιβάρυνση στο χρόνο εκτέλεσής του.

Έτσι δεν έχει σημασία αν ο παραλήπτης εμπιστεύεται τον δημιουργό του εκτελέσιμου, αφού με αυτόν τον τρόπο μπορεί να διαπιστώσει αν αυτό πληροί τις ζητούμενες προδιαγραφές. Αυτό είναι πολύ σημαντικό σε περιπτώσεις που ο δημιουργός του εκτελέσιμου είναι άγνωστος, ή δεν είναι αξιόπιστος. Χαρακτηριστικό παράδειγμα είναι τα κατανεμημένα συστήματα υπολογισμού, τα οποία ανταλλάσσουν εκτελέσιμο κώδικα χωρίς να εμπιστεύονται απαραίτητα το ένα το άλλο.

3.1 Το σύστημα πιστοποιημένων εκτελέσιμων NFlint

Το σύστημα αυτό έχει αναπτυχθεί στο εργαστήριο Τεχνολογίας Λογισμικού του Εθνικού Μετσόβιου Πολυτεχνείου και αποτελεί μια επέκταση του Flint, το οποίο αναπτύχθηκε στο πανεπιστήμιο του Yale. Το NFlint παρέχει έναν μεταγλωττιστή από μια συναρτησιακή γλώσσα σε γλώσσα μηχανής. Σε συνδυασμό με ένα ισχυρό σύστημα τύπων δίνει την δυνατότητα απόδειξης διάφορων ιδιοτήτων του προγράμματος.

Για τον σκοπό αυτό, η διεπαφή του με τον χρήστη χωρίζεται σε δύο επιμέρους τμήματα: τη γλώσσα τύπων (type language) και τη γλώσσα υπολογισμών (computation language). Η πρώτη περιλαμβάνει τους κανόνες των τύπων και σε αυτήν περιγράφονται οι τύποι των δομικών μονάδων της γλώσσας υπολογισμών. Σε αυτήν επίσης γράφονται οι αποδείξεις για τις ιδιότητες του προγράμματος. Για την υλοποίηση του NFlint έχει χρησιμοποιηθεί το Calculus of Inductive Constructions (CIC) και συγκεκριμένα η υλοποίησή του στο σύστημα COQ. Στη γλώσσα υπολογισμών αναπαριστώνται οι διάφορες υπολογιστικές εντολές και κατά συνέπεια

όλη η λειτουργικότητα των παραγόμενων προγραμμάτων. Οι δύο γλώσσες συνδέονται μεταξύ τους με την χρήση singleton types. Ένας όρος στην γλώσσα υπολογισμών (το πρόγραμμα) μαζί με έναν όρο στην γλώσσα τύπων (την απόδειξη) αποτελούν ένα πιστοποιημένο εκτελέσιμο (certified binary).

3.1.1 Μεταγλώττιση με αποδείξεις

Το πιστοποιημένο εκτελέσιμο στην αρχική γλώσσα πρέπει να μεταγλωττιστεί σε γλώσσα μηχανής έτσι ώστε να μπορεί να εκτελεστεί από τον υπολογιστή. Η μεταγλώττιση αυτή γίνεται σε διάφορα στάδια, κάθε ένα από τα οποία είναι μια ενδιάμεση γλώσσα με τους δικούς της κανόνες τύπων. Το αρχικό πρόγραμμα, μετασχηματιζόμενο σε αυτές τις γλώσσες διατηρεί τις ιδιότητές που είχε στην αρχική, όπως αποδεικνύεται από τις κατάλληλες αποδείξεις οι οποίες επίσης μεταγλωττίζονται από τον μεταγλωττιστή NFlint. Τόσο οι κανόνες τύπων όσο και οι αποδείξεις είναι στην ίδια γλώσσα τύπων, το CIC.

Από την αρχική γλώσσα, το πρόγραμμα επιδέχεται τους παρακάτω μετασχηματισμούς, εκφραζομένους από αντίστοιχες ενδιάμεσες γλώσσες:

- Continuation Passing Style (CPS).
- Closure conversion.
- Hoisting.
- Typed Assembly Language (TAL).

Η τελευταία γλώσσα είναι μια γλώσσα μηχανής που χρησιμοποιεί το σύστημα τύπων για απόδειξη ιδιοτήτων. Αυτή η γλώσσα, η οποία είναι και το τελευταίο στάδιο στην διαδικασία της μεταγλώττισης που παρέχει πιστοποίηση, έχει τις εξής ιδιότητες:

- Δεν είναι σχετιζόμενη με κάποια συγκεκριμένη αρχιτεκτονική υλικού.
- Θεωρεί ότι υπάρχουν άπειροι καταχωρητές.
- Παρέχει εντολή ανάθεσης μνήμης.
- **Δεν** παρέχει εντολή αποδέσμευσης μνήμης.
- **Δεν** παρέχει συλλογή σκουπιδιών.

Ένα πρόγραμμα στην ουσία είναι:

- ένα σύνολο συναρτήσεων (μια ακολουθία εντολών).
- μια ένδειξη για την αρχική συνάρτηση (νούμερο συνάρτησης).
- μια ένδειξη για την τελική συνάρτηση (νούμερο συνάρτησης).

και οι εντολές του περιλαμβάνουν εντολές επεξεργασίας, εντολές άλματος και την εντολή τερματισμού.

Η εκτέλεση ξεκινάει από την αρχική συνάρτηση. Λόγω της φύσεως του προγράμματος (CPS), όλες οι συναρτήσεις αποτελούνται από μια διαδοχική σειρά εντολών επεξεργασίας και καταλήγουν σε μια εντολή άλματος. Μόνη εξαίρεση αποτελεί η συνάρτηση τερματισμού όπου τελειώνει στην εντολή τερματισμού (halt). Αυτό σημαίνει ότι ανά πάσα στιγμή υπάρχει ακριβώς μια ενεργή κλήση συνάρτησης στην στοίβα. Οι συναρτήσεις αυτές ονομάζονται συναρτήσεις ουράς (tail functions).

3.1.2 Το σύστημα LLVM

Το πρόγραμμα για να έχει κάποια χρησιμότητα, πρέπει να μεταγλωττιστεί από αυτήν την ιδεατή γλώσσα μηχανής στην γλώσσα μηχανής του υπολογιστή στον οποίον πρέπει να εκτελεστεί. Για τον σκοπό αυτό έχει χρησιμοποιηθεί το σύστημα LLVM. Το σύστημα αυτό μεταγλωττίζει ένα πρόγραμμα από μία επίσης ιδεατή γλώσσα μηχανής σε εκτελέσιμο αρχείο για διάφορες αρχιτεκτονικές. Σε αυτό το στάδιο, που είναι και το τελευταίο του μεταγλωττισμού, το πρόγραμμα που είναι ήδη σε TAL μεταγλωττίζεται στην ενδιάμεση γλώσσα του LLVM. Κατόπιν το LLVMτο μεταγλωττίζει σε γλώσσα μηχανής.

Αν και το LLVM παρέχει ένα σύστημα τύπων, είναι πολύ πιο απλό από το CIC και για αυτό θεωρούμε το στάδιο αυτό μη πιστοποιημένο. Γίνεται η ελάχιστη χρήση του συστήματος τύπων του που χρειάζεται για να θεωρούνται οι όροι σωστοί στο LLVM.

Η διαχείριση μνήμης δεν αντιμετωπίζεται από το LLVM και είναι αντικείμενο αυτής της εργασίας. Σε αυτό το στάδιο, που είναι και αυτό που ενδιαφέρει την συλλογή σκουπιδιών, στο πρόγραμμα εμφανίζονται οι παρακάτω τιμές (values):

- Σταθερές (boolean, integer).
- Δείκτες σε συναρτήσεις.
- Δείκτες σε tuples.

Η υλοποίηση του μεταγλωττιστή έχει γίνει συγκεκριμένα για αρχιτεκτονική Intel 32 bit. Οι σταθερές έχουν υλοποιηθεί με μία λέξη μεγέθους 32 bit, με το boolean να αντιστοιχεί στις τιμές 0/1. Οι δείκτες τόσο σε συναρτήσεις, όσο και σε tuples έχουν εύρος 32 bit και δείχνουν μόνο σε διευθύνσεις που αντιστοιχούν σε λέξεις, λόγω της φύσεως της αρχιτεκτονικής. Οι tuples είναι σύνολα διατεταγμένων τιμών, τα οποία χρησιμοποιούνται από το πρόγραμμα. Το μέγεθός τους είναι μεταβλητό. Μετά την αρχικοποίησή τους όμως δεν αλλάζουν. Όλα τα αντικείμενα του προγράμματος μπορεί να θεωρηθούν ότι είναι tuples. Μιας και όλες οι τιμές έχουν μέγεθος 32 bit (είτε είναι σταθερές είτε είναι δείκτες), μία tuple με N θέσεις απαιτεί $32 \times N$ bits μνήμης.

Για τον διαχωρισμό των δεικτών από τις σταθερές, οι τελευταίες ολισθαίνονται αριστερά κατά ένα bit και το λιγότερο σημαντικό ψηφίο τους (LSB) τίθεται ίσο με 1. Επειδή όλες οι λέξεις είναι ευθυγραμμισμένες (alligned) ανά λέξη (word), όλοι οι δείκτες έχουν τα δύο λιγότερο σημαντικά bit τους μηδενικά—αφού μία λέξη ισούται με τέσσερα bytes (ή ακέραιο

πολλαπλάσιο αυτών). Ως εκ τούτου το τελευταίο bit μίας τιμής φανερώνει εάν αυτή είναι σταθερά η δείκτης. Αναφορικά με τους δείκτες σε συναρτήσεις, αυτοί δείχνουν στη στοίβα, ενώ οι δείκτες στις tuples δείχνουν στο σωρό. Αυτό καθιστά εύκολο τον διαχωρισμό των δεικτών, αφού οι πρώτοι έχουν διαφορετικό εύρος διευθύνσεων από τους δεύτερους.

Το σύστημα αυτό χρησιμοποιεί κατανομή σωρού για την δημιουργία των tuples. Όπως και στις περισσότερες συναρτησιακές γλώσσες, έτσι και εδώ, τα αντικείμενα που δημιουργούνται χρειάζεται να έχουν χρόνο ζωής μεγαλύτερο από τα υποπρογράμματα (π.χ. διαδικασίες/συναρτήσεις) που τα δημιούργησαν. Αν συνδυάσουμε αυτά με το γεγονός ότι δεν υπάρχει ενσωματωμένος συλλέκτης σκουπιδιών, ούτε παρέχεται στον προγραμματιστή η δυνατότητα να αποδεσμεύσει τη μνήμη χειροκίνητα, οδηγούμαστε με βεβαιότητα στο συμπέρασμα, ότι αργά ή γρήγορα η διαθέσιμη μνήμη θα εξαντληθεί.

3.2 Συλλογή σκουπιδιών στο NFlint

Η υλοποίηση ενός συλλέκτη σκουπιδιών για το σύστημα αυτό αποτελεί μονόδρομο. Ο συλλέκτης που θα δημιουργηθεί πρέπει να είναι ασφαλής, δηλαδή να μην απελευθερώνει ποτέ την μνήμη ενός ''ζωντανού'' αντικειμένου, και αποτελεσματικός, δηλαδή να αποτρέπει όσο το δυνατόν καλύτερα τις διαρροές μνήμης. Ο συλλέκτης αυτός θα τοποθετηθεί στο ενδιάμεσο στάδιο της μεταγλώττισης των προγραμμάτων, αφού δηλαδή έχει παραχθεί ο ενδιάμεσος κώδικας και πριν την παραγωγή του τελικού κώδικα.

Όπως είδαμε στο προηγούμενο κεφάλαιο ο συλλέκτης σκουπιδιών έχει κάποιες απαιτήσεις από τον μεταγλωττιστή για να δουλέψει. Ο αρχικός σχεδιασμός του NFlint δεν περιελάμβανε συλλέκτη σκουπιδιών. Αναπόφευκτα χρειάστηκε να γίνουν κάποιες αλλαγές στον παραγόμενο από τον μεταγλωττιστή ενδιάμεσο κώδικα για να υποστηριχθεί η συλλογή σκουπιδιών. Ο μεταγλωττιστής πλέον ικανοποιεί τις περισσότερες από τις απαιτήσεις αυτές. Ειδικότερα:

1. Απαγορεύει την αριθμητική δεικτών. Έτσι ο προγραμματιστής δεν μπορεί να ''κρύψει'' τους δείκτες με αποτέλεσμα τα ζωντανά αντικείμενα (tuples) να είναι πάντα προσιτά.

2. Απαγορεύει δείκτες σε τυχαίες θέσεις μνήμης, ακόμα κι αν αυτές βρίσκονται εντός της δικαιοδοσίας του προγράμματος. Όλοι οι δείκτες δείχνουν στην αρχή μιας tuple.

3. Εξασφαλίζει ότι κάθε ανάθεση μνήμης πραγματοποιείται από την συνάρτηση ανάθεσης μνήμης (allocator) του συλλέκτη σκουπιδιών, η οποία επιστρέφει έναν δείκτη στην αρχή της νεοδημιουργηθείσας tuple.

4. Διασφαλίζει, μέσω του τρόπου που περιγράφηκε προηγουμένως με χρήση του lsb ότι μία τυχαία λέξη στο σωρό δε θα μοιάζει με έναν έγκυρο δείκτη.

Τα παραπάνω θεωρούνται **δεδομένα** από τους αλγόριθμους συλλογής σκουπιδιών που θα υλοποιηθούν στη συνέχεια και συνιστούν τις **απαιτήσεις** που αυτοί έχουν από τη γλώσσα, οι οποίες, αν δεν ικανοποιηθούν, καθιστούν αδύνατη την συλλογή σκουπιδιών.

Επιπλέον ο μεταγλωττιστής προσφέρει, πέρα από τις παραπάνω απαιτήσεις, και κάποιες πρόσθετες διευκολύνσεις:

- Ελαχιστοποιεί κατά το δυνατόν τους δείκτες σε άχρηστα αντικείμενα (tuples). Χρησιμοποιεί πρόθυμη αποτίμηση στις εκφράσεις και καθαρίζει τους δείκτες μόλις αυτοί δεν χρειάζονται πια.

- Μειώνει το πλήθος των ριζών. Πιο συγκεκριμένα οι αναδρομικές συναρτήσεις πραγματοποιούν αναδρομή ουράς (tail recursion), το οποίο σημαίνει ότι σε κάθε αναδρομική κλήση μιας συνάρτησης, υπάρχει μόνο ένα εγγράφημα δραστηριοποίησης (activation record) μειώνοντας έτσι το μέγεθος της στοίβας. Μικρότερη στοίβα συνεπάγεται λιγότερες ρίζες.

- Δεν επιτρέπει σε μια νέα tuple να δείχνει σε μία παλαιότερη. Αυτό εξασφαλίζει ότι δεν θα υπάρχουν κυκλικά συνδεδεμένες tuples

Οι διευκολύνσεις αυτές μπορούν να βοηθήσουν στην υλοποίηση και την αποτελεσματικότητα κάποιων αλγορίθμων, ωστόσο δεν αποτελούν απαιτήσεις χωρίς την ικανοποίηση των οποίων δεν θα ήταν δυνατή η συλλογή σκουπιδιών. Για τον λόγο αυτό δεν θεωρούνται δεδομένες από τους αλγόριθμους.

Δυστυχώς όμως υπήρχαν και άλλες απαιτήσεις, οι οποίες δεν στάθηκε δυνατό να ικανοποιηθούν. Πιο συγκεκριμένα, ο μεταγλωττιστής αυτός, στην παρούσα του μορφή, δεν υποστηρίζει την παραγωγή επιπλέον κώδικα για την παρακολούθηση των εγγραφών – αναγνώσεων σε δείκτες (π.χ. write barriers) ή κώδικα για τον συγχρονισμό με συλλέκτη που τρέχει παράλληλα με το πρόγραμμα σε διαφορετικό νήμα εκτέλεσης (thread). Η προσπάθεια μετατροπής του ώστε να υποστηρίξει την παραγωγή τέτοιου κώδικα, ο οποίος απαιτείται από κάποιους αλγόριθμους συλλογής σκουπιδιών, και να τον παρεμβάλει στο πρόγραμμα κρίθηκε ασύμφορη και εγκαταλείφθηκε. Άμεση συνέπεια των παραπάνω είναι ότι, στην παρούσα φάση, δεν είναι δυνατή η υλοποίηση των ταυτόχρονων και αυξητικών συλλεκτών. Οι αλγόριθμοι μέτρησης αναφορών είναι επίσης αδύνατον να υλοποιηθούν. Αυτό στην ουσία μας αφήνει μόνο τους ανιχνευτικούς αλγόριθμους που ‘‘σταματούν τον κόσμο’’. Και οι έξι αλγόριθμοι που θα υλοποιηθούν στη συνέχεια ανήκουν σε αυτήν την κατηγορία. Ανάμεσα στους αλγόριθμους αυτούς υπάρχουν μετακινούντες (moving) και μη–μετακινούντες (non–moving), καθώς οι πολιτικές αυτές δεν επηρεάζονται από τους παραπάνω περιορισμούς.

Το κόστος σε χρόνο θεωρήθηκε σημαντικότερο από το κόστος σε χώρο, κι έτσι οι έγιναν κάποιες υποχωρήσεις εις βάρος του χώρου που απαιτούν οι αλγόριθμοι με σκοπό να επιτευχθεί καλύτερος χρόνο εκτέλεσης. Χαρακτηριστικά παραδείγματα τέτοιων υποχωρήσεων είναι:

- Η χρήση στοίβας από όλους τους αλγορίθμους για την αναδρομική διάσχιση των αντικειμένων.

- Η επιλογή της first–fit πολιτικής ανάθεσης μνήμης αντί της best–fit στους μη–μετακινούντες αλγόριθμους.

- Αντιγραφή ή μετακίνηση των αντικειμένων με τρόπο που να αυξάνει την τοπικότητά τους στους μετακινούντες αλγόριθμους.

Οι αλγόριθμοι αυτοί καλύπτουν τις απαιτήσεις του συστήματος NFlint, καθώς αυτό δεν προορίζεται για τη δημιουργία διαδραστικών προγραμμάτων ή εφαρμογών πραγματικού χρόνου.

Κεφάλαιο 4

Υλοποίηση Συλλογής Σκουπιδιών

Μετά τις αναγκαίες τροποποιήσεις στην παραγωγή του ενδιάμεσου κώδικα, είναι εφικτή η σύνδεση του τελικού εκτελέσιμου με κάποιον συλλέκτη σκουπιδιών. Παρακάτω παρουσιάζονται οι προδιαγραφές που πρέπει να πληρούνται από την υλοποίηση ενός αλγορίθμου συλλογής σκουπιδιών, προκειμένου να μπορεί ο αλγόριθμος αυτός να συνεργαστεί με τον κώδικα που παράγει ο μεταγλωττιστής. Αντίστροφα οποιαδήποτε υλοποίηση γλώσσας ικανοποιεί τις προδιαγραφές αυτές, μαζί με τις απαιτήσεις που τέθηκαν στο προηγούμενο κεφάλαιο, μπορεί να χρησιμοποιήσει τους αλγόριθμους αυτούς για τη συλλογή σκουπιδιών. Το σύνολο των κοινώς αποδεκτών προδιαγραφών αυτών, που επιτρέπει στα δύο μέρη (πρόγραμμα – συλλέκτης σκουπιδιών) να επικοινωνούν μεταξύ τους αποτελεί την *διεπαφή* (*interface*) της συλλογής σκουπιδιών. Γίνεται φανερό ότι ένας από τους βασικούς σκοπούς της διεπαφής αυτής είναι να ορίζει και να διατυπώνει το σύνολο των λειτουργιών – υπηρεσιών που παρέχει ο συλλέκτης σκουπιδιών στο πρόγραμμα και αντίστροφα, χωρίς να επιτρέπει πρόσβαση στον κώδικα που υλοποιεί αυτές τις υπηρεσίες. Η διεπαφή είναι στην ουσία ένα ''συμβόλαιο κλήσης'' μεταξύ καλούντος και καλούμενου και διαχωρίζει την προγραμματιστική υλοποίηση κάποιων υπηρεσιών από τη χρήση τους.

4.1 Παρουσίαση της διεπαφής συλλογής σκουπιδιών

Στη συνέχεια θα δούμε τον τρόπο με τον οποίο το κυρίως πρόγραμμα επικοινωνεί με τον συλλέκτη σκουπιδιών, έχει δηλαδή πρόσβαση στις λειτουργίες που αυτός παρέχει (π.χ. ανάθεση μνήμης, συλλογή σκουπιδιών) αλλά και πώς ο συλλέκτης σκουπιδιών επικοινωνεί με το πρόγραμμα (π.χ. για την εξέταση των ριζών).

Η διεπαφή της συλλογής σκουπιδιών έχει υλοποιημένες κάποιες συναρτήσεις, τις οποίες παρέχει στο πρόγραμμα, μέσω των οποίων αυτό αποκτά πρόσβαση στις λειτουργίες του συλλέκτη. Το πρόγραμμα δηλαδή δεν επικοινωνεί άμεσα με τον συλλέκτη σκουπιδιών, αλλά έμμεσα. Πρώτα καλεί τις συναρτήσεις που παρέχει η διεπαφή και μετά αυτές με τη σειρά τους καλούν τις συναρτήσεις του συλλέκτη. Αυτές οι συναρτήσεις μπορούν να θεωρηθούν ως *συναρτήσεις–περιτύλιγμα* (*wrapper functions*), αφού ο κύριος σκοπός τους είναι να καλέσουν τις συναρτήσεις του συλλέκτη σκουπιδιών, οι οποίες θα κάνουν την πραγματική δουλειά. Οι

συναρτήσεις αυτές είναι τρεις:

`int32_t *gc_alloc(int32_t size)`. Αυτή την συνάρτηση πρέπει να καλεί το πρόγραμμα όταν χρειάζεται επιπλέον μνήμη (π.χ. για την δημιουργία μιας tuple). Το όρισμα της συνάρτησης αυτής είναι ένας ακέραιος αριθμός (`size`), ο οποίος αντιστοιχεί στο μέγεθος σε λέξεις (1 λέξη = 32 bit) της μνήμης στο σωρό που χρειάζεται να δεσμευθεί. Αφού εξακριβωθεί ότι ο αριθμός αυτός είναι μεγαλύτερος του μηδενός, τότε καλείται με το ίδιο όρισμα η συνάρτηση ανάθεσης μνήμης του συλλέκτη σκουπιδιών. Αν αυτή επιστρέψει οποιαδήποτε τιμή πέρα από `NULL`, τότε η κλήση της θεωρείται επιτυχής και η συγκεκριμένη τιμή (δείκτης) επιστρέφεται στο πρόγραμμα. Αν επιστρέψει `NULL` τότε σημαίνει πως η διαθέσιμη μνήμη στο σωρό έχει εξαντληθεί. Ακολουθεί άμεσα η κλήση της συνάρτησης του συλλέκτη που είναι υπεύθυνη για την συλλογή των σκουπιδιών. Μόλις επιστρέψει η συνάρτηση αυτή, σημαίνει ότι η συλλογή σκουπιδιών ολοκληρώθηκε και όσα σκουπίδια βρέθηκαν, έχουν ελευθερωθεί. Για να διαπιστωθεί αν η μνήμη που ελευθερώθηκε επαρκεί για την ανάθεση που απέτυχε προηγουμένως, επιχειρείται ξανά κλήση της συνάρτησης ανάθεσης μνήμης του συλλέκτη. Αν αποτύχει (επιστρέψει `NULL`) και πάλι, τότε ενημερώνεται ο χρήστης, με το κατάλληλο μήνυμα, ότι η διαθέσιμη μνήμη δεν επαρκεί για την ολοκλήρωση του προγράμματος και η λειτουργία του προγράμματος τερματίζεται. Διαφορετικά επιστρέφεται στο πρόγραμμα η τιμή (δείκτης) που επέστρεψε η συνάρτηση ανάθεσης μνήμης του συλλέκτη.

`void gc_init(void)`. Η συνάρτηση αυτή είναι αρχικά υπεύθυνη για την δημιουργία του σωρού και την αρχικοποίηση ενός δείκτη που δείχνει στην αρχή του, ενώ στη συνέχεια καλεί την συνάρτηση αρχικοποίησης του συλλέκτη σκουπιδιών. Ο σωρός που δημιουργείται έχει μέγεθος που δίνεται από την μεταβλητή `HEAP_SPACE_SIZE`, η τιμή της οποίας μπορεί να καθοριστεί πριν από τη μεταγλώττιση της διεπαφής. Το μέγεθος του σωρού παραμένει αμετάβλητο κατά την εκτέλεση του προγράμματος. Αν και θα μπορούσε κάθε αλγόριθμος να μεταβάλλει το μέγεθος του σωρού, ανάλογα με τις ανάγκες του, εντούτοις αποφασίστηκε κάθε αλγόριθμος να έχει και να διαχειρίζεται ένα προκαθορισμένο μέγεθος σωρού. Με αυτόν τον τρόπο μπορεί να εξακριβωθεί στη συνέχεια ποιος αλγόριθμος διαχειρίζεται καλύτερα ένα δεδομένο μέγεθος σωρού και πως επηρεάζεται η συμπεριφορά των αλγορίθμων για διάφορα μεγέθη του σωρού, κοινά όμως για όλους τους αλγόριθμους.

`void gc_final(void)`. Αυτή η συνάρτηση (συνάρτηση ολοκλήρωσης) καλείται ακριβώς πριν ολοκληρωθεί η λειτουργία του προγράμματος. Φροντίζει για την αποδέσμευση της μνήμης που δεσμεύτηκε για τη δημιουργία του σωρού. Επίσης καλεί την αντίστοιχη συνάρτηση ολοκλήρωσης του συλλέκτη. Η συνάρτηση αυτή είναι κατάλληλη και για την προβολή στατιστικών σχετικά με την συλλογή σκουπιδιών.

Μια τέτοια έμμεση πρόσβαση στις συναρτήσεις του συλλέκτη προσφέρει κάποια πλεονεκτήματα. Αρχικά, διευκολύνεται ο έλεγχος των λαθών στον αλγόριθμο συλλογής σκουπιδιών, αφού μπορεί πολύ εύκολα, μέσω της διεπαφής αυτής, να προσδιοριστεί ποια συνάρτηση προκαλεί

κάποιο σφάλμα, χωρίς τροποποίηση του κώδικα του συλλέκτη. Ακόμη, η διεπαφή εκτελεί κάποιους ελέγχους κατά το χρόνο εκτέλεσης του προγράμματος, οι οποίοι είναι απαραίτητοι και που, χωρίς αυτήν, θα έπρεπε να γίνονται από τον συλλέκτη, με αποτέλεσμα να χρειαζόταν παραπάνω κώδικας για την υλοποίηση κάθε αλγόριθμου. Επίσης, η διεπαφή έχει αναλάβει εξ' ολοκλήρου την επικοινωνία με τον χρήστη, η οποία γίνεται σε περιβάλλον γραμμής εντολών. Αν απαιτηθεί η επικοινωνία να γίνει σε γραφικό περιβάλλον για παράδειγμα, τότε δεν χρειάζεται να γίνει καμία τροποποίηση στον συλλέκτη σκουπιδιών, παρά μόνο στη διεπαφή.

Αναλυτικά, όλος ο κώδικας της υλοποίησης της διεπαφής αυτής παρουσιάζεται στο Παράρτημα 2.

4.1.1 Υπηρεσίες που παρέχει ο συλλέκτης

Σύμφωνα με τα παραπάνω κάθε συλλέκτης σκουπιδιών οφείλει να παρέχει μια σειρά από υπηρεσίες, τις οποίες περιμένει από αυτόν η συγκεκριμένη διεπαφή. Ειδικότερα κάθε αλγόριθμος συλλογής σκουπιδιών πρέπει να παρέχει υλοποίηση για τις ακόλουθες συναρτήσεις:

`int32_t *gc_algo_alloc(int32_t size)`. Πρόκειται για τη συνάρτηση που είναι υπεύθυνη για την δυναμική ανάθεση μνήμης στο σωρό (allocator). Δέχεται ως όρισμα έναν ακέραιο αριθμό, ο οποίος είναι εξασφαλισμένα μεγαλύτερος του μηδενός και αντιπροσωπεύει το μέγεθος σε λέξεις (1 λέξη = 32 bit) της μνήμης στο σωρό που χρειάζεται να δεσμευθεί (`size`). Ο χώρος αυτός θα χρησιμοποιηθεί για τη δημιουργία μιας tuple. Η συνάρτηση αυτή, αφού βρει ή δημιουργήσει ένα ενιαίο τμήμα ελεύθερης μνήμης στο σωρό με το ζητούμενο μέγεθος, πρέπει να επιστρέψει έναν δείκτη στην αρχή του τμήματος αυτού. Το τμήμα αυτό κατοχυρώνεται και παύει να θεωρείται πια ελεύθερο. Ο δείκτης που επιστρέφεται θα είναι ο μόνος τρόπος για το πρόγραμμα να προσπελάσει αυτό το τμήμα στο σωρό. Όλοι οι δείκτες στο σωρό, θα πρέπει να είναι δείκτες που έχουν επιστραφεί από αυτήν τη συνάρτηση, αφού μόνο από αυτήν θα γίνεται η ανάθεση μνήμης στο σωρό. Σε περίπτωση που δεν βρεθεί ένα τέτοιο τμήμα, κατάλληλο για ανάθεση, αυτό σημαίνει πως είναι αδύνατον να ικανοποιηθεί η απαίτηση του προγράμματος για νέα μνήμη. Τότε η συνάρτηση αυτή θεωρείται ότι έχει αποτύχει και πρέπει να επιστρέψει την τιμή `NULL`[1].

`void gc_algo_collect(void)`. Με την κλήση αυτής της συνάρτησης ενεργοποιείται ο συλλέκτης σκουπιδιών. Δεν δέχεται καμία παράμετρο, ούτε επιστρέφει κάποια τιμή. Η συνάρτηση αυτή καλείται όταν έχει εξαντληθεί η διαθέσιμη μνήμη στο σωρό ή, ισοδύναμα, όταν αποτύχει μία κλήση στη συνάρτηση ανάθεσης. Δουλειά της είναι να βρίσκει τα σκουπίδια στο σωρό και να απελευθερώνει τη μνήμη που έχει εκχωρηθεί σε αυτά. Ο τρόπος που το πετυχαίνει αυτό, ποικίλει ανάλογα με τον αλγόριθμο συλλογής σκουπιδιών που χρησιμοποιείται. Τα τμήματα μνήμης που ελευθερώνονται από τη συνάρτηση αυτή, μπορούν στη συνέχεια να επαναχρησιμοποιηθούν από τη συνάρτηση ανάθεσης μνήμης.

[1] Δεν προτείνεται η συνάρτηση ανάθεσης ή οποιαδήποτε άλλη συνάρτηση του συλλέκτη σκουπιδιών να τροποποιήσει με οποιονδήποτε τρόπο (π.χ. `malloc()`, `calloc()`, `realloc()`) του μεγέθους του σωρού κατά το χρόνο εκτέλεσης του προγράμματος.

`void gc_algo_init(void)`. Αυτή η συνάρτηση χρησιμοποιείται για την αρχικοποίηση του συλλέκτη σκουπιδιών. Κάθε αλγόριθμος συλλογής σκουπιδιών απαιτεί την δική του εξειδικευμένη αρχικοποίηση, ανάλογα με τις ανάγκες του. Σκοπός της συνάρτησης αυτής είναι να αρχικοποιηθούν οι διάφορες μεταβλητές που χρησιμοποιούνται από τον εκάστοτε αλγόριθμο συλλογής σκουπιδιών με τιμές, οι οποίες γίνονται γνωστές μόνο στον χρόνο εκτέλεσης του προγράμματος (π.χ. αρχική διεύθυνση του σωρού). Η κλήση της γίνεται μόνο στην αρχή κάθε προγράμματος.

`void gc_algo_final(void)`. Αυτή είναι η συνάρτηση ολοκλήρωσης του συλλέκτη σκουπιδιών. Καλείται μόνο κατά την ολοκλήρωση της εκτέλεσης του προγράμματος. Είναι χρήσιμη για τυχόν καθάρισμα που θέλει να κάνει ο συλλέκτης, αλλά και για την προβολή στατιστικών, όπως αριθμός των φορών που κλήθηκε ο συλλέκτης, πόσο χρόνο έκανε κ.λ.π.

Οι συναρτήσεις αυτές δεν χρειάζεται να επικοινωνούν με το χρήστη. Όλη την απαραίτητη επικοινωνία (π.χ. ειδοποίηση για την εξάντληση της μνήμης) την έχει αναλάβει η διεπαφή της συλλογής σκουπιδιών. Εξαιρέσεις αποτελούν μηνύματα προς το χρήστη, τα οποία μπορούν να χρησιμοποιηθούν για την προβολή στατιστικών ή για σκοπούς εκσφαλμάτωσης (debugging) του συλλέκτη. Τέλος η εκτέλεση του προγράμματος δεν αναμένεται να τερματιστεί σε καμία από τις παραπάνω συναρτήσεις. Σε περίπτωση που κάποιος αλγόριθμος συλλογής σκουπιδιών δεν μπορεί να συνεχίσει τη λειτουργία του και είναι απαραίτητος ο τερματισμός του προγράμματος, τότε αυτός είναι υπεύθυνος για την ενημέρωση του χρήστη με το κατάλληλο μήνυμα. Αν συμβαίνει αυτό, θα πρέπει να αναφέρεται ρητά στις προδιαγραφές του αλγορίθμου.

4.1.2 Υπηρεσίες που παρέχει η διεπαφή

Από τη μεριά της η διεπαφή παρέχει και αυτή κάποιες υπηρεσίες στον συλλέκτη σκουπιδιών, οι οποίες τον βοηθούν να έχει πρόσβαση σε δεδομένα του προγράμματος, που είναι γνωστά μόνο κατά τον χρόνο εκτέλεσης. Πιο συγκεκριμένα παρέχονται οι ακόλουθες συναρτήσεις:

`int32_t get_heap_size(void)`. Η συνάρτηση αυτή, όταν καλείται από τον συλλέκτη, επιστρέφει το μέγεθος του σωρού σε λέξεις. Το μέγεθος αυτό είναι στατικό και δεν αλλάζει κατά τον χρόνο εκτέλεσης του προγράμματος.

`int32_t *heap_space_ptr(void)`. Με την κλήση αυτής της συνάρτησης επιστρέφεται ένας δείκτης στην αρχή του σωρού. Ο δείκτης αυτός συνήθως ζητείται από την συνάρτηση αρχικοποίησης του κάθε συλλέκτη, αλλά μπορεί να μπορεί να χρησιμοποιηθεί πρακτικά σε οποιοδήποτε σημείο του, αφού ο χώρος που καταλαμβάνει ο σωρός είναι σταθερός και η αρχή του δεν πρόκειται να αλλάξει στον χρόνο εκτέλεσης του προγράμματος.

`void gc_walk_roots(void (*walk_root)(int32_t *))`. Αυτή η συνάρτηση χρησιμοποιείται από τον συλλέκτη για να διατρέξει τις ρίζες του προγράμματος. Δέχεται ως όρισμα μία δεύτερη συνάρτηση, την οποία καλεί για κάθε ρίζα, περνώντας της ως παράμετρο ένα

δείκτη στη ρίζα αυτή. Απαραίτητη προϋπόθεση για τη δεύτερη συνάρτηση (συνάρτηση–όρισμα) είναι να δέχεται ως όρισμα έναν δείκτη. Το γεγονός ότι τα ορίσματα είναι δείκτες στις ρίζες επιτρέπει στον συλλέκτη να μεταβάλει, σε περίπτωση που χρειάζεται, την τιμή των ριζών του προγράμματος.

Αφού παρουσιάστηκαν όλες οι συναρτήσεις της διεπαφής της συλλογής σκουπιδιών, θα ήταν χρήσιμο να δούμε ένα παράδειγμα του τι συμβαίνει κατά τη διάρκεια εκτέλεσης ενός προγράμματος που χρησιμοποιεί συλλογή σκουπιδιών, κάνοντας χρήση της συγκεκριμένης διεπαφής.

Τυπικά λοιπόν, όταν ξεκινάει ένα πρόγραμμα καλείται η συνάρτηση αρχικοποίησης της διεπαφής, η οποία δημιουργεί τον σωρό και αρχικοποιεί έναν δείκτη στην αρχή του. Στη συνέχεια καλείται η συνάρτηση αρχικοποίησης του συλλέκτη σκουπιδιών, η οποία ενημερώνεται για το μέγεθος και την αρχική διεύθυνση του σωρού, μέσω των συναρτήσεων που παρέχει η διεπαφή. Μετά ξεκινάει η εκτέλεση του προγράμματος, του οποίου όλες οι ανάγκες για δυναμική εκχώρηση μνήμης στο σωρό εξυπηρετούνται από τη συνάρτηση ανάθεσης μνήμης του συλλέκτη (μέσω της διεπαφής). Αν η συνάρτηση ανάθεσης του συλλέκτη κάποια στιγμή αποτύχει (επιστρέψει `NULL` δηλαδή), αυτό σημαίνει πως έχει εξαντληθεί ο διαθέσιμος χώρος στο σωρό. Αμέσως μετά την αποτυχημένη κλήση της συνάρτησης ανάθεσης ακολουθεί κλήση στον συλλέκτη σκουπιδιών. Αυτός, χρησιμοποιώντας τη συνάρτηση της διεπαφής που διατρέχει τις ρίζες, θα κατασκευάσει τις ‘‘αλυσίδες’’ των προσιτών αντικειμένων που ξεκινάνε από αυτές και έτσι θα μπορέσει να διαπιστώσει ποια αντικείμενα είναι ζωντανά και ποια είναι σκουπίδια απελευθερώνοντας τη μνήμη που καταλαμβάνουν τα δεύτερα. Μετά από την κλήση στη συνάρτηση συλλογής σκουπιδιών, ακολουθεί ξανά κλήση στη συνάρτηση ανάθεσης μνήμης του συλλέκτη προκειμένου να εξακριβωθεί αν η μνήμη που ελευθερώθηκε επαρκεί για να ικανοποιηθεί η ανάθεση μνήμης που απέτυχε προηγουμένως. Αν αποτύχει και πάλι η ανάθεση μνήμης, τότε το πρόγραμμα εμφανίζει μήνυμα στο χρήστη ότι δεν επαρκεί η μνήμη και τερματίζει την λειτουργία του. Σε διαφορετική περίπτωση η εκτέλεση του προγράμματος συνεχίζεται μέχρι είτε να αποτύχει πάλι η συνάρτηση ανάθεσης, οπότε και ακολουθεί ξανά κλήση στο συλλέκτη σκουπιδιών για να ελευθερωθεί μνήμη κ.ο.κ., είτε μέχρι να ολοκληρωθεί φυσιολογικά η εκτέλεσή του. Προφανώς αν το πρόγραμμα δεν έχει απαιτήσεις για δυναμική εκχώρηση μνήμης στο σωρό, δεν θα κληθεί ποτέ η συνάρτηση ανάθεσης μνήμης και κατά συνέπεια δε θα ενεργοποιηθεί ποτέ ο συλλέκτης σκουπιδιών. Κατά τον ομαλό τερματισμό του προγράμματος καλείται η συνάρτηση ολοκλήρωσης της διεπαφής, η οποία στη συνέχεια καλεί τη συνάρτηση ολοκλήρωσης του συλλέκτη σκουπιδιών.

4.2 Υλοποίηση των αντικειμένων

Όπως είδαμε σε προηγούμενο κεφάλαιο όλα αντικείμενα (tuples) στην NFlint έχουν μεταβλητό μήκος. Επίσης δεν υπάρχει δεδομένο που να μην είναι tuple. Ακόμη και ένας ακέραιος θεωρείται tuple με μέγεθος ίσο με 1. Για να μπορέσει να λειτουργήσει αποτελεσματικά ένας συλλέκτης σκουπιδιών πρέπει να έχει πληροφορίες για το χώρο που καταλαμβάνει η κάθε tuple. Μία λύση είναι να προστεθεί μία επιπλέον λέξη (word) σε κάθε tuple, στην οποία θα

αποθηκεύεται το μέγεθος της. Αν και τυπικά τα bits που απαιτούνται για να περιγραφεί το μέγεθος μιας tuple είναι πολύ λιγότερα από αυτά που έχει μια λέξη, ωστόσο αυτή είναι η μικρότερη δυνατή επιβάρυνση προκειμένου να έχουμε όλες τις tuples στο σωρό ευθυγραμμισμένες ανά λέξη. Με αυτό τον τρόπο, κάθε tuple αποκτά μια επικεφαλίδα, η οποία προσδιορίζει το μέγεθός της σε λέξεις. Η επικεφαλίδα χρειάζεται, όπως και τα υπόλοιπα δεδομένα του προγράμματος, ολίσθηση προς τα αριστερά, και να τεθεί το λιγότερο σημαντικό bit της ίσο με 1. Απαιτείται ολίσθηση 2 θέσεων καθώς το επιπλέον αυτό bit θα χρησιμοποιηθεί από τους αλγόριθμους σκουπιδιών. Συνεπώς το μέγιστο μέγεθος μιας tuple είναι 2^{30} bits. Η επικεφαλίδα κατασκευάζεται στην συνάρτηση ανάθεσης μνήμης του συλλέκτη σκουπιδιών. Έτσι λοιπόν αν ζητηθεί μνήμη με μέγεθος 3 λέξεων, τότε η συνάρτηση ανάθεσης πρέπει να θεωρεί ότι απαιτείται εκχώρηση τμήματος μνήμης με μέγεθος $3 + 1 = 4$ λέξεις. Το πρόγραμμα δεν πρέπει να έχει επίγνωση της επικεφαλίδας αυτής και δεν θα μπορεί να την προσπελάσει. Μόνο ο συλλέκτης σκουπιδιών θα μπορεί να κάνει κάτι τέτοιο. Ο επιστεφόμενος δείκτης στο καινούριο τμήμα μνήμης που κατοχυρώθηκε θα πρέπει να δείχνει στην αμέσως επόμενη λέξη μετά την επικεφαλίδα. Τα bits που περισσεύουν από την επικεφαλίδα (αυτά δηλαδή που δεν χρησιμοποιούνται για την καταγραφή του μεγέθους της) μπορούν να χρησιμοποιηθούν από τον συλλέκτη για την αποθήκευση χρήσιμων πληροφοριών, όπως για παράδειγμα να διαπιστώνει αν έχει επισκεφθεί ήδη μία tuple. Όλοι οι αλγόριθμοι που θα παρουσιαστούν στη συνέχεια ακολουθούν την παραπάνω τακτική.

4.3 Υλοποίηση των αλγορίθμων

Η γλώσσα στην οποία υλοποιήθηκε η παραπάνω διεπαφή αλλά και οι αλγόριθμοι συλλογής σκουπιδιών που θα παρουσιαστούν στη συνέχεια είναι η C. Πρόκειται για μία προστακτική γλώσσα προγραμματισμού υψηλού επιπέδου που επιτρέπει τη χρήση δεικτών. Στους αλγόριθμους έχει γίνει η υπόθεση ότι το μέγεθος των λέξεων και των δεικτών είναι 32 bit. Κάθε αλγόριθμος έχει τοποθετηθεί στο δικό του αρχείο πηγαίου κώδικα (source code) και η επιλογή του γίνεται στατικά κατά τον χρόνο μεταγλώττισης του προγράμματος. Τέλος να σημειωθεί ότι οι αλγόριθμοι αυτοί έχουν σχεδιαστεί ώστε να μπορούν να λειτουργούν σε οποιαδήποτε γλώσσα χρησιμοποιεί την εν λόγω διεπαφή και ταυτόχρονα ικανοποιεί και τις απαιτήσεις που παρουσιάστηκαν στο προηγούμενο κεφάλαιο.

Αναλυτικά ο κώδικας κάθε αλγορίθμου παρουσιάζεται στο Παράρτημα 1, στο τέλος της εργασίας αυτής.

Κεφάλαιο 5

Μη-Μετακινούντες Αλγόριθμοι

Όπως είδαμε σε προηγούμενο κεφάλαιο, οι μη–μετακινούντες (non–moving) αλγόριθμοι συλλογής σκουπιδιών είναι αυτοί που δεν μετακινούν κανένα αντικείμενο κατά τη διαδικασία της συλλογής. Όλα τα ζωντανά αντικείμενα παραμένουν στη θέση τους. Η ανάθεση μνήμης σε αυτούς τους αλγορίθμους γίνεται ψάχνοντας τις θέσεις που κάποτε άνηκαν σε άχρηστα αντικείμενα και που τώρα είναι ελεύθερες.

Στη συνέχεια του κεφαλαίου αυτού θα παρουσιαστούν δύο τέτοιοι αλγόριθμοι: Ο αλγόριθμος *σημείωσης σάρωσης* (*mark and sweep*) και ο αλγόριθμος *σημείωσης χωρίς σάρωση* (*mark and don't sweep*). Και οι δύο είναι ανιχνευτικοί (tracing) αλγόριθμοι που ''σταματούν τον κόσμο''. Αναλυτικά τα χαρακτηριστικά των ανιχνευτικών αλγορίθμων και των αλγορίθμων που ''σταματούν τον κόσμο'' παρουσιάζονται στο κεφάλαιο 2.

5.1 Ο αλγόριθμος Σημείωσης Σάρωσης

5.1.1 Περιγραφή

Ο αλγόριθμος αυτός είναι ο πρώτος ανιχνευτικός αλγόριθμος που χρησιμοποιήθηκε για τη συλλογή σκουπιδιών. Παρουσιάστηκε από τον John McCarthy[McC60] το 1960 και θεωρείται ένας από τους κλασσικότερους αλγόριθμους συλλογής σκουπιδιών, ενώ κάποιες παραλλαγές του παραμένουν ιδιαίτερα δημοφιλείς ακόμη και σήμερα. Η βασική ιδέα του συγκεκριμένου αλγορίθμου είναι αρκετά απλή.

Κάθε αντικείμενο (στην προκειμένη περίπτωση κάθε tuple) έχει μια σημαία (flag), η οποία δείχνει αν το αντικείμενο αυτό είναι ζωντανό ή όχι. Η σημαία αυτή χρησιμοποιείται μόνο από τον συλλέκτη σκουπιδιών. Αν η σημαία είναι καθαρή, αυτό σημαίνει πως το αντικείμενο είναι σκουπίδι, και πρέπει να ανακτηθεί η μνήμη που έχει εκχωρηθεί σε αυτό. Σε διαφορετική περίπτωση το αντικείμενο θεωρείται ζωντανό και διατηρείται από τον συλλέκτη. Στην πράξη χρησιμοποιείται ένα bit. Χωρίς βλάβη της γενικότητας μπορούμε να θεωρήσουμε ότι αν η τιμή του είναι 1, τότε η tuple είναι ζωντανή, αλλιώς, αν η τιμή του είναι 0, είναι σκουπίδι[1]. Το bit αυτό ονομάζεται και bit σήμανσης (mark bit).

[1]Η σημασία του 0 και του 1 μπορεί να είναι και η αντίστροφη, ανάλογα με την υλοποίηση.

Algorithm 1 Mark & Sweep

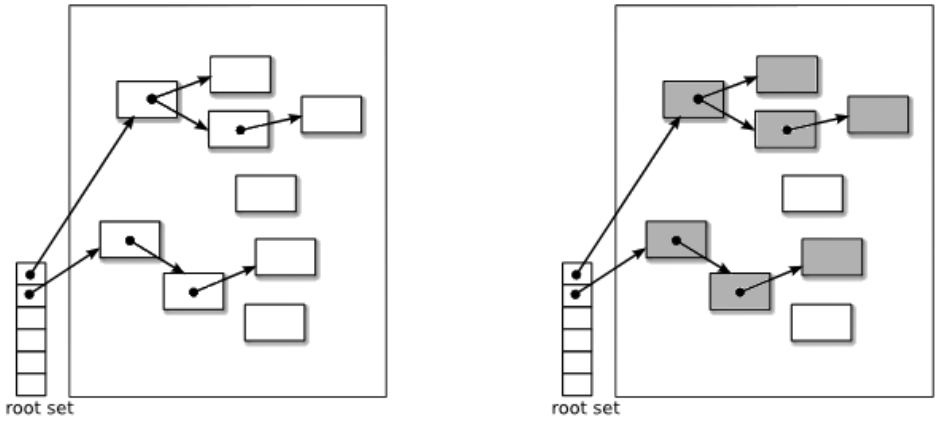
```
for each root r do
  if r is heap pointer then
    mark(r)
sweep()
```

where:

```
mark(object x):
if x is not marked then
  x.mark_bit ← marked
  for every field f_i of x do
    if x.f_i is heap pointer then
      mark(x.f_i)
```

```
sweep():
for each object p in heap do
  if p is marked then
    p.marked_bit ← not marked
  else put p in freelist
```

Σε όλη τη διάρκεια της εκτέλεσης του προγράμματος καμία tuple δεν είναι σημειωμένη, εκτός από τη στιγμή που εκτελείται ο συλλέκτης σκουπιδιών. Η λειτουργία του συλλέκτη σημείωσης σάρωσης έχει, όπως υποδηλώνει και το όνομά του, δύο στάδια. Στο πρώτο στάδιο, ο συλλέκτης *σημειώνει* (*mark*) ποιες tuples είναι ζωντανές χρησιμοποιώντας το συγκεκριμένο bit τους ως σημαία. Η διαδικασία ξεκινάει από τις ρίζες. Σκοπός του συλλέκτη είναι να επισκεφτεί και να σημειώσει όλες τις tuples που είναι προσιτές από τις ρίζες του προγράμματος. Έτσι δημιουργούνται αλυσίδες από ζωντανές tuples, οι οποίες μπορούμε να θεωρήσουμε ότι σχηματίζουν έναν κατευθυνόμενο γράφο, του οποίου οι ρίζες είναι οι ρίζες του προγράμματος. Για να διασχίσουμε αυτόν το γράφο μπορούμε να χρησιμοποιήσουμε οποιονδήποτε αλγόριθμο διάσχισης γράφων όπως για παράδειγμα διάσχιση κατά βάθος (Depth–First–Search) ή διάσχιση κατά πλάτος (Breadth–First–Search). Στη συγκεκριμένη υλοποίηση χρησιμοποιήθηκε η διάσχιση κατά βάθος (DFS), σύμφωνα με την οποία αν μία ρίζα δείχνει σε κάποια tuple στο σωρό, τότε ο συλλέκτης επισκέπτεται αυτήν την tuple. Αν δεν είναι ήδη σημειωμένη, την σημειώνει και στη συνέχεια την εξετάζει για δείκτες σε άλλες tuples. Αν βρεθούν τέτοιοι δείκτες, τότε ο συλλέκτης με αναδρομή ακολουθεί κάθε δείκτη στο σωρό, σημειώνει την αντίστοιχη tuple και εν συνεχεία την εξετάζει και αυτήν για δείκτες στο σωρό. Αν η tuple είναι ήδη σημειωμένη, αυτό σημαίνει ότι ο συλλέκτης την έχει ήδη επισκεφθεί στο παρόν στάδιο σημείωσης και έχει σημειώσει όλες τις tuples που είναι προσιτές από εκείνη. Σε αυτήν την περίπτωση ο συλλέκτης την αφήνει και συνεχίζει από το σημείο που ήταν πριν την επισκεφτεί. Αυτό εξασφαλίζει δύο πράγματα: Πρώτον ότι το στάδιο της σημείωσης θα τερματιστεί, ακόμη και στην περίπτωση που ο αλγόριθμος συναντήσει κυκλικές δομές δεδομένων και δεύτερον ότι ο τερματισμός του σταδίου αυτού θα γίνει μόνο όταν έχουν σημειωθεί όλα τα προσιτά από τις ρίζες αντικείμενα. Η διαδικασία αυτή επαναλαμβάνεται για όλες τις ρίζες του προγράμματος. Αφού ολοκληρωθεί το στάδιο αυτό, όσες tuples δεν έχουν σημειωθεί σημαίνει ότι είναι απρόσιτες από τις ρίζες του προγράμματος και άρα αποτελούν σκουπίδια. Αμέσως ξεκινάει



(α′) Πριν (β′) Μετά

Σχήμα 5.1: Ο σωρός πριν και μετά το στάδιο της σημείωσης

το δεύτερο στάδιο του συλλέκτη, το στάδιο της *σάρωσης* (*sweep*), το οποίο αποδεσμεύει την μνήμη των σκουπιδιών. Στο στάδιο αυτό ο συλλέκτης χρειάζεται να διατρέξει ολόκληρο το σωρό και να εξετάσει μια προς μια τις tuples που αυτός περιέχει. Όσες tuples δεν είναι σημειωμένες θεωρούνται ως ελεύθερες και τοποθετούνται σε μια συνδεδεμένη λίστα (freelist). Η λίστα αυτή, μετά την ολοκλήρωση του δεύτερου σταδίου, θα περιέχει όλα τα τμήματα μνήμης που είναι ελεύθερα και θα χρησιμοποιηθεί στη συνέχεια από την συνάρτηση ανάθεσης μνήμης του συλλέκτη. Ο συλλέκτης αφήνει στη θέση της κάθε σημειωμένη tuple που συναντάει ενώ παράλληλα καθαρίζει το bit σήμανσής της, προετοιμάζοντας έτσι τον σωρό για να ακολουθήσει το επόμενο στάδιο σημείωσης.

5.1.2 Ανάθεση μνήμης

Η ανάθεση μνήμης στον αλγόριθμο σημείωσης σάρωσης πραγματοποιείται με τη βοήθεια της λίστας των ελεύθερων τμημάτων που έχει κατασκευαστεί στο στάδιο της σάρωσης. Αυτό είναι απαραίτητο επειδή τα τμήματα ελεύθερης μνήμης είναι διασκορπισμένα στο σωρό. Σκοπός της συνάρτησης ανάθεσης μνήμης είναι να διατρέξει τη συνδεδεμένη αυτή λίστα και να βρει ένα τμήμα μνήμης στο σωρό κατάλληλο για ανάθεση. Οι τρόποι με τους οποίους μπορεί να πραγματοποιηθεί το παραπάνω είναι δύο:

- Ο πρώτος τρόπος, ο οποίος είναι γνωστός και ως *πρώτο–ταίριασμα* (*first–fit*), απαιτεί να γίνει η διάσχιση της λίστας μέχρι να βρεθεί το πρώτο τμήμα ελεύθερης μνήμης με μέγεθος μεγαλύτερο η ίσο από το ζητούμενο. Αυτό έχει σαν αποτέλεσμα να είναι γρηγορότερη η ανάθεση, αλλά μπορεί να οδηγήσει στη διάσπαση ενός τμήματος ελεύθερης μνήμης στο σωρό, με μέγεθος πολύ μεγαλύτερο του απαιτουμένου, ενώ στη συνέχεια της λίστας ίσως υπάρχουν τμήματα με το ζητούμενο μέγεθος, τα οποία θα ήταν καταλληλότερα για τη συγκεκριμένη ανάθεση. Αυτή η πολιτική ανάθεσης μνήμης έχει ως συνέπεια τον μεγαλύτερο κατακερματισμό του σωρού, καθώς έχει την τάση να διασπά τα μεγάλα τμήματα ελεύθερης μνήμης σε μικρότερα, δημιουργώντας έτσι την πιθανότητα,

ενώ υπήρχε πριν από κάποια διάσπαση ένα τμήμα ελεύθερης μνήμης με επαρκές μέγεθος, μετά από αυτήν να μην μπορεί να βρεθεί άλλο με το κατάλληλο μέγεθος.

- Σύμφωνα με τον δεύτερο τρόπο, ο οποίος είναι γνωστός ως *καλύτερο–ταίριασμα* (*best–fit*) η λίστα διασχίζεται μέχρι να βρεθεί ένα τμήμα ελεύθερης μνήμης στο σωρό που να έχει μέγεθος ίσο με το ζητούμενο. Αν κάτι τέτοιο δεν είναι εφικτό, τότε παραχωρείται το τμήμα με το πλησιέστερο στο ζητούμενο μέγεθος. Αυτή η πολιτική ανάθεσης μνήμης αν και μειώνει τον κατακερματισμό του σωρού, καθώς δεν διασπά τα μεγαλύτερα τμήματα ελεύθερης μνήμης, παρά μόνο όταν είναι πραγματικά απαραίτητο, έχει μεγάλο κόστος εκτέλεσης, αφού συχνά απαιτείται η εξέταση ολόκληρης της λίστας των ελευθέρων τμημάτων.

Προφανώς σε μία γλώσσα της οποίας όλα τα αντικείμενα έχουν το ίδιο μέγεθος οι δύο αυτοί τρόποι ταυτίζονται. Η συνάρτηση ανάθεσης μνήμης, ανεξάρτητα από την πολιτική που θα ακολουθηθεί, είναι υπεύθυνη για τη διατήρηση της λίστας ελεύθερων τμημάτων (freelist). Όταν εκχωρείται ένα τμήμα μνήμης θα πρέπει να το αφαιρεί από την συγκεκριμένη λίστα. Επίσης είναι υπεύθυνη για το σχηματισμό της επικεφαλίδας κάθε τμήματος που παραχωρεί. Στη συγκεκριμένη περίπτωση η επικεφαλίδα αυτή είναι η επικεφαλίδα της tuple, αφού στο NFlint όλες οι αναθέσεις μνήμης γίνονται μόνο για τη δημιουργία των tuples. Το bit σήμανσης της κάθε tuple θα πρέπει να είναι καθαρό (0). Μόνο ο συλλέκτης σκουπιδιών θα μπορεί να αποφασίσει αν χρειάζεται να σημειωθεί μία tuple ή όχι. Αν δεν είναι δυνατόν να βρεθεί ένα τμήμα μνήμης στο σωρό κατάλληλο για ανάθεση, τότε πρέπει να κληθεί ο συλλέκτης σκουπιδιών, για να ανανεωθεί η λίστα.

5.1.3 Κόστος εκτέλεσης

Ο αλγόριθμος σημείωσης σάρωσης έχει δύο στάδια. Κάθε στάδιο κάνει και ένα πέρασμα στο σωρό.

Το κόστος του σταδίου της σημείωσης (mark) είναι ανάλογό μόνο του αριθμού των ζωντανών αντικειμένων στο σωρό. Όσο αυξάνεται το πλήθος των ζωντανών αντικειμένων, τόσο αυξάνεται το κόστος του συλλέκτη για να τα σημειώσει όλα. Δεν επηρεάζεται δηλαδή από το μέγεθος του σωρού, μόνο από την πληρότητά του.

Το κόστος του σταδίου της σάρωσης (sweep) είναι ανάλογο του συνολικού μεγέθους του σωρού και όχι της πληρότητάς του, αφού πρέπει να σαρωθεί ολόκληρος για να βρεθούν τα σημειωμένα αντικείμενα.

Επίσης στο κόστος του αλγορίθμου πρέπει να συνυπολογιστεί και το κόστος της ανάθεσης μνήμης. Το κόστος αυτό εξαρτάται από τις απαιτήσεις σε μνήμη του προγράμματος και δεν είναι εύκολο να υπολογιστεί. Η επιλογή της πολιτικής ανάθεσης μνήμης (first–fit ή best–fit) επηρεάζει σημαντικά την απόδοση του προγράμματος. Συνήθως η πολιτική first–fit οδηγεί γρηγορότερα στην εύρεση κατάλληλου τμήματος μνήμης για ανάθεση, αλλά αυξάνει τον κατακερματισμό του σωρού, οδηγώντας σε περισσότερες κλήσεις του συλλέκτη σκουπιδιών.

Τέλος όσον αφορά το κόστος του αλγορίθμου σημείωσης σάρωσης σε χώρο, δεν υπάρχουν κάποιες πρόσθετες απαιτήσεις. Ο αλγόριθμος αυτός μπορεί να διαχειριστεί ολόκληρο το

μέγεθος του σωρού που έχει στη διάθεσή του. Χρειάζεται όμως στοίβα για την αναδρομική διάσχιση των αντικειμένων. Το μέγεθος της στοίβας αυτής χρειάζεται να είναι τόσο μεγάλο όσο και το μεγαλύτερο μονοπάτι των προσιτών από τις ρίζες αντικειμένων[2].

5.1.4 Υλοποίηση του αλγορίθμου

Στην παρούσα υλοποίηση ο αλγόριθμος χρησιμοποιεί την πολιτική πρώτου–ταιριάσματος για την ανάθεση μνήμης. Ακόμη διασχίζει τα αντικείμενα αναδρομικά χρησιμοποιώντας τον αλγόριθμο διάσχισης κατά βάθος. Επίσης προσπαθεί να ενώνει διαδοχικά τμήματα ελεύθερης μνήμης, σχηματίζοντας ένα τμήμα μεγαλύτερου μεγέθους με σκοπό να ελαττώσει τον κατακερματισμό του σωρού. Τέλος η συνδεδεμένη λίστα με τα ελεύθερα αντικείμενα αποθηκεύεται στον ίδιο το σωρό, χωρίς να απαιτείται επιπλέον χώρος. Αυτό καθίσταται εύκολο, αφού κάθε tuple έχει μέγεθος τουλάχιστον δύο λέξεων—μία για την επικεφαλίδα της και μία τουλάχιστον για το δεδομένο που αυτή αποθηκεύει. Η πρώτη λέξη χρησιμοποιείται ως ένδειξη του μεγέθους του ελεύθερου τμήματος και στην δεύτερη αποθηκεύεται ένας δείκτης στην επόμενη ελεύθερη tuple. Ως bit σήμανσης χρησιμοποιείται το δεύτερο λιγότερο σημαντικό ψηφίο (lsb) της επικεφαλίδας μιας tuple, αφού το πρώτο χρησιμοποιείται για να εξασφαλιστεί ότι η λέξη αυτή δεν είναι δείκτης στο σωρό. Άρα το μέγεθος στην επικεφαλίδα μιας tuple χρειάζεται ολίσθηση προς τα αριστερά κατά 2 θέσεις. Για την ανάκτησή του προφανώς απαιτείται ολίσθηση προς τα δεξιά πάλι κατά 2 θέσεις.

Αφού παρουσιάστηκαν όλες οι λεπτομέρειες του αλγόριθμου σημείωσης σάρωσης, θα παρουσιαστεί συνοπτικά και η υλοποίησή των απαιτούμενων, από τη διεπαφή συλλογής σκουπιδιών, συναρτήσεων που αυτός παρέχει.

Στη συνάρτηση `int32_t *gc_algo_alloc(int32_t size)` ο αλγόριθμος διατρέχει την λίστα ελεύθερων τμημάτων, ψάχνοντας για ένα τμήμα με μέγεθος ίσο με `size + 1` (Χρειαζόμαστε μια επιπλέον λέξη για τη δημιουργία της επικεφαλίδας της tuple). Αυτό που χρειάζεται για τη διάσχιση της λίστας είναι ένας δείκτης στο πρώτο στοιχείο της. Αυτό με τη σειρά του δείχνει στο επόμενο κ.ο.κ.. Έτσι αφού κατά τη διάσχιση της λίστας βρεθεί το πρώτο τμήμα με μέγεθος μεγαλύτερο η ίσο αυτού που απαιτείται, διακρίνουμε δύο περιπτώσεις.

- Το τμήμα να έχει μέγεθος ίσο με το ζητούμενο. Στην περίπτωση αυτή η συνάρτηση ανάθεσης θα παραχωρήσει ολόκληρο το τμήμα. Αρχικά κατασκευάζει την επικεφαλίδα στην πρώτη λέξη του τμήματος. Η επικεφαλίδα περιέχει την τιμή του μεγέθους της tuple ολισθημένη αριστερά 2 κατά θέσεις και το λιγότερο σημαντικό bit της είναι ίσο με 1. Ο δείκτης που έδειχνε σε αυτήν, τώρα δείχνει στην επόμενη ελεύθερη tuple (αν υπάρχει). Η συνάρτηση επιστρέφει δείκτη στη δεύτερη λέξη του τμήματος μνήμης που κατοχυρώθηκε.

- Το τμήμα να έχει μέγεθος μεγαλύτερο από το ζητούμενο. Στην περίπτωση αυτή το τμήμα διασπάται σε δύο κομμάτια. Το πρώτο κομμάτι έχει μέγεθος ίσο με το ζητούμενο, οπότε

[2]Στα σημερινά συστήματα που χρησιμοποιούν εικονική μνήμη αυτό δεν είναι πρόβλημα, επειδή δεσμεύεται ένας πολύ μεγάλος χώρος για τη στοίβα του προγράμματος.

και πραγματοποιείται η εκχώρηση της μνήμης σύμφωνα με την πρώτη περίπτωση. Το δεύτερο κομμάτι έχει το μέγεθος που απομένει. Επειδή όμως δεν διαθέτει την κατάλληλη επικεφαλίδα, η συνάρτηση ανάθεσης μνήμης φροντίζει να του φτιάξει μία με το νέο του μέγεθος. Αν το μέγεθος αυτό είναι μεγαλύτερο ή ίσο του 2, τότε το κομμάτι αυτό τοποθετείται στη συνδεδεμένη λίστα για να χρησιμοποιηθεί σε μελλοντική ανάθεση μνήμης. Αν το μέγεθός του είναι ίσο με 1, τότε αυτό δεν επαρκεί για να συνδεθεί στη λίστα, αφού απαιτείται μέγεθος τουλάχιστον 2 λέξεων—μία για το μέγεθός του και μία για τον δείκτη στο επόμενο στοιχείο της λίστας. Στην περίπτωση αυτή το κομμάτι αυτό αφήνεται εκτός λίστας με την ελπίδα ότι θα ενωθεί αργότερα με κάποιο από τα δύο γειτονικά του κομμάτια, σχηματίζοντας ένα μεγαλύτερο τμήμα ελεύθερης μνήμης.

Αν δεν υπάρχει τμήμα μνήμης με μέγεθος μεγαλύτερο ή ίσο του απαιτούμενου, τότε η συνάρτηση ανάθεσης μνήμης επιστρέφει `NULL`.

Στη συνάρτηση `void gc_algo_collect(void)`, γίνεται η κλήση του συλλέκτη σκουπιδιών. Αρχικά διατρέχονται οι ρίζες με χρήση της συνάρτησης `gc_walk_roots()` που παρέχει η διεπαφή, με παράμετρο την συνάρτηση `void applyMark(int32_t *ptr)`, η οποία θα εφαρμοστεί σε κάθε ρίζα. Σκοπός της τελευταίας είναι αρχικά να ελέγχει αν αυτή η ρίζα είναι δείκτης στο σωρό και σε περίπτωση που είναι θα καλεί την συνάρτηση `mark(int32_t *x)` η οποία θα σημειώσει αναδρομικά, με χρήση του αλγόριθμου διάσχισης κατά βάθος (DFS) όλες τις προσιτές tuples από την συγκεκριμένη ρίζα. Αν συναντήσει μία ήδη σημειωμένη tuple, την αγνοεί. Αυτό θα έχει σαν αποτέλεσμα να σημειωθούν τελικά όλες οι ζωντανές tuples στο σωρό. Έπειτα καλείται η συνάρτηση `sweep(int32_t* l)`, η οποία στη συγκεκριμένη υλοποίηση, είναι μία αναδρομική συνάρτηση που επιστρέφει κάθε φορά έναν δείκτη στην επόμενη μη σημειωμένη tuple. Αυτοί οι δείκτες θα χρησιμοποιηθούν για τη δημιουργία της συνδεδεμένης λίστας με τα ελεύθερα τμήματα. Ως παράμετρο δέχεται έναν δείκτη στην επικεφαλίδα μιας tuple, από την οποία συνεχίζει τη σάρωση του σωρού. Η πρώτη κλήση της `sweep()` έχει ως παράμετρο την πρώτη tuple στο σωρό και όταν επιστρέψει θα δώσει έναν δείκτη στην αρχή της συνδεδεμένης λίστας. Στη συνέχεια σε κάθε αναδρομική κλήση της η `sweep()` διασχίζει το σωρό καθαρίζοντας το bit σήμανσης κάθε σημειωμένης tuple που συναντά, μέχρι να βρει μία μη σημειωμένη (εφόσον υπάρχει). Τότε ελέγχει μήπως είναι και οι αμέσως επόμενές της μη σημειωμένες. Όσες συνεχόμενες μη σημειωμένες tuples βρει τις ενώνει (merge) με αυτήν σχηματίζοντας ένα μεγαλύτερο τμήμα ελεύθερης μνήμης. Παράλληλα φτιάχνει για το τμήμα αυτό καινούρια επικεφαλίδα με το νέο του μέγεθος. Αν δεν σταθεί δυνατόν να γίνει κάποια συνένωση τότε το νέο μέγεθος ταυτίζεται με το παλιό. Αν το συνολικό μέγεθος του τμήματος αυτού είναι ίσο με 1, τότε το τμήμα αυτό αγνοείται με την ελπίδα να εκμεταλλευτεί σε κάποια επόμενη κλήση της. Ο έλεγχος για συνεχόμενες μη σημειωμένες tuples σταματάει μόλις βρεθεί μία σημειωμένη tuple. Αυτή θα περαστεί ως παράμετρος στην επόμενη αναδρομική κλήση της `sweep()`, με την οποία θα συνεχιστεί η διάσχιση του σωρού. Αν κατά τη διάσχιση η `sweep()` φτάσει στο τέλος του σωρού, χωρίς να βρεθεί μη σημειωμένη tuple, τότε επιστρέφει `NULL`. Όταν επιστρέψει μια `sweep()`, η `sweep()` που την κάλεσε αποθηκεύει τον επιστρεφόμενο δείκτη στη δεύτερη λέξη της μη σημειωμένης tuple που έχει βρει και επιστρέφει τελικά έναν δείκτη σε αυτήν. Δεν έχει σημασία αν δεν είχε εξαντληθεί η λίστα αυτή πριν την κλήση του

συλλέκτη σκουπιδιών, αφού η `sweep()` θα φτιάξει καινούρια με όλα τα ελεύθερα τμήματα μνήμης στο σωρό.

Η συνάρτηση αρχικοποίησης του συλλέκτη `void gc_algo_init(void)`, αρχικά ενημερώνεται για το μέγεθος και την αρχική διεύθυνση του σωρού, μέσω των συναρτήσεων της διεπαφής. Στη συνέχεια κατασκευάζει τη λίστα ελευθέρων τμημάτων η οποία περιέχει μόνο ένα τμήμα ελεύθερης μνήμης. Το τμήμα αυτό είναι ολόκληρος ο σωρός, αφού δεν έχει γίνει καμία ανάθεση μνήμης ακόμα.

Τέλος, η συνάρτηση ολοκλήρωσης του αλγορίθμου `void gc_algo_final(void)` δεν χρειάζεται να κάνει κάποια δουλειά και το σώμα της παρέμεινε κενό.

5.1.5 Πλεονεκτήματα – Μειονεκτήματα

Βασικότερο πλεονέκτημα του αλγορίθμου σημείωσης σάρωσης είναι ότι μπορεί να διαχειριστεί τις κυκλικές δομές δεδομένων. Επίσης δεν απαιτείται η παραγωγή επιπλέον κώδικα από τον μεταγλωττιστή που να παρακολουθεί τις αναγνώσεις – εγγραφές δεικτών. Επιπλέον ο αλγόριθμος αυτός μπορεί να διαχειριστεί ολόκληρο το μέγεθος του σωρού, το οποίο έχει στη διάθεσή του. Μπορεί να συνεργαστεί και με άλλους αλγόριθμους συλλογής σκουπιδιών, ενώ μπορεί και να μετατραπεί ώστε να μην χρησιμοποιεί στοίβα για τη διάσχιση των αντικειμένων, με επιπλέον επιβάρυνση όμως στο κόστος εκτέλεσής του.

Από την άλλη, το μεγαλύτερο μειονέκτημα του αλγορίθμου αυτού είναι ότι αφήνει τον σωρό κατακερματισμένο (fragmented) διασκορπίζοντας τις tuples σε κάθε σημείο του. Αυτό όχι μόνο αυξάνει τις διαρροές μνήμης στο πρόγραμμα, αλλά προκαλεί και απώλεια της τοπικότητας των αντικειμένων. Σε ένα σύστημα με εικονική μνήμη αυτό μπορεί να οδηγήσει σε συνεχείς εναλλαγές (swapping) των σελίδων μεταξύ της πρωτεύουσας και δευτερεύουσας μνήμης. Τέλος ο αλγόριθμος αυτός προκαλεί παύσεις στην εκτέλεση του προγράμματος, οι οποίες είναι ανάλογες τόσο του μεγέθους του σωρού, όσο και της πληρότητάς του, καθώς απαιτεί σε κάθε εκτέλεσή του να πραγματοποιούνται δύο διασχίσεις του σωρού, χωρίς να υπάρχει η δυνατότητα διακοπής από το πρόγραμμα. Το γεγονός αυτό καθιστά τον αλγόριθμο μη αποδοτικό για χρήση σε αλληλεπιδραστικές εφαρμογές ή εφαρμογές πραγματικού χρόνου.

Παρά τους περιορισμούς αυτούς όμως, ο αλγόριθμος σημείωσης σάρωσης παραμένει ένας αποτελεσματικός αλγόριθμος, ειδικά σε εφαρμογές και συστήματα, στα οποία σημαντικότερο μέλημα είναι ο συνολικός χρόνος εκτέλεσης.

5.2 Ο αλγόριθμος Σημείωσης χωρίς Σάρωση

5.2.1 Περιγραφή

Ο αλγόριθμος σημείωσης χωρίς σάρωση είναι ένας αλγόριθμος που βασίζεται στον αλγόριθμο σημείωσης σάρωσης. Σκοπός του είναι να προσπαθήσει να μειώσει τις μεγάλες παύσεις στην εκτέλεση του προγράμματος που συνήθως προκαλεί ο αλγόριθμος σημείωσης σάρωσης. Η βασική ιδέα είναι να αναβληθεί η σάρωση του σωρού, μέχρι την επόμενη ανάθεση μνήμης[Hug82]. Η μεθοδολογία αυτή είναι γνωστή και ως οκνηρή σάρωση (lazy sweeping). Το

κόστος της σάρωσης το πληρώνει η συνάρτηση ανάθεσης μνήμης, η οποία σε κάθε κλήση της είναι αναγκασμένη να σαρώνει ένα τμήμα του σωρού, μέχρι να βρεθεί κομμάτι κατάλληλο για ανάθεση. Επειδή όμως η σάρωση του σωρού γίνεται τμηματικά, το κόστος αυτό ισοκατανέμεται στο πρόγραμμα.

Σε κάθε κύκλο συλλογής σκουπιδιών, όπως γίνεται φανερό από το όνομα του αλγόριθμου, υπάρχει μόνο ένα στάδιο, αυτό της σημείωσης (mark) των αντικειμένων (tuples). Το στάδιο αυτό είναι παρόμοιο με εκείνο του αλγορίθμου σημείωσης σάρωσης. Και εδώ υπάρχει η ανάγκη για ένα bit σήμανσης σε κάθε tuple, ανάλογα με την τιμή του οποίου ο συλλέκτης θα μπορέσει να προσδιορίσει αν η tuple είναι ζωντανή ή σκουπίδι. Σε αντίθεση όμως με τον αλγόριθμο σημείωσης σάρωσης, στον αλγόριθμο αυτό η σημασία της τιμής του bit σήμανσης αλλάζει από τον έναν κύκλο συλλογής σκουπιδιών στον άλλον. Κατά τα γνωστά, πριν από την εκτέλεση του σταδίου της σήμανσης, καμία tuple δεν είναι σημειωμένη. Ο συλλέκτης, ξεκινώντας από τις ρίζες, διασχίζει αναδρομικά, κάνοντας χρήση του αλγορίθμου διάσχισης κατά βάθος[3], όλες τις tuples που είναι προσιτές από τις ρίζες σημειώνοντάς τες, κάνοντας χρήση του bit σήμανσης που αυτές διαθέτουν. Η τιμή που θα πάρει το bit σήμανσης είναι αυτή που αντιστοιχεί εκείνη τη στιγμή στη σημασία του σημειωμένου (marked). Όπως και στον αλγόριθμο σημείωσης σάρωσης, έτσι κι εδώ σε περίπτωση που ο συλλέκτης βρει μία ήδη σημειωμένη tuple την αφήνει και συνεχίζει από το σημείο που ήταν πριν την επισκεφτεί, εξασφαλίζοντας αφενός ότι το στάδιο της σημείωσης θα τερματιστεί, ακόμη και στην περίπτωση κυκλικών δομών δεδομένων, αφετέρου ότι ο τερματισμός του σταδίου αυτού θα γίνει μόνο όταν έχουν σημειωθεί όλες οι προσιτές από τις ρίζες tuples, οι οποίες όμως παραμένουν σημειωμένες ακόμα και κατά την εκτέλεση του προγράμματος.

Algorithm 2 Mark & don't Sweep

```
marked ← not marked
for each root r do
  if r is heap pointer then
    mark(r)

where:
mark(object x):
if x is not marked then
  x.mark_bit ← marked
  for every field f_i of x do
    if x.f_i is heap pointer then
      mark(x.f_i)
```

Όταν ολοκληρωθεί το στάδιο αυτό τότε όσες tuples δεν έχουν σημειωθεί αποτελούν σκουπίδια και θεωρούνται στη συνέχεια ως τμήματα ελεύθερου χώρου, τα οποία μπορούν να χρησιμοποιηθούν από τη συνάρτηση ανάθεσης μνήμης.

5.2.2 Ανάθεση μνήμης

Η συνάρτηση ανάθεσης μνήμης του αλγορίθμου σημείωσης χωρίς σάρωση είναι η πιο πολύπλοκη συνάρτηση ανάθεσης μνήμης που θα παρουσιαστεί σε αυτήν την εργασία. Αυτό συμβαίνει επειδή η συνάρτηση αυτή είναι υπεύθυνη να εντοπίσει τα τμήματα ελεύθερης μνήμης,

[3]Και εδώ μπορεί να χρησιμοποιηθεί οποιοσδήποτε αλγόριθμος διάσχισης γράφων

τα οποία είναι διασκορπισμένα στο σωρό, χωρίς να έχει κάποια βοήθεια (π.χ. μια συνδεδεμένη λίστα). Για να το πετύχει αυτό, σαρώνει το σωρό μέχρι να βρεθεί ένα ελεύθερο τμήμα κατάλληλου μεγέθους για ανάθεση. Στην περίπτωση αυτή η πολιτική πρώτου–ταιριάσματος αποτελεί μονόδρομο, αφού η πολιτική καλύτερου–ταιριάσματος μπορεί να επιβαρύνει σημαντικά το ήδη υψηλό κόστος της ανάθεσης μνήμης. Παρακάτω θα εξηγηθεί ο λόγος που συμβαίνει αυτό.

Η συνάρτηση ξεκινάει από την αρχή του σωρού, εξετάζοντας μία προς μία τις tuples που βρίσκονται σε αυτόν, ψάχνοντας να βρει μια tuple κατάλληλη για την ανάθεση. Χρησιμοποιεί έναν βοηθητικό δείκτη, η τιμή του οποίου διατηρείται από κλήση σε κλήση, για να ξέρει σε ποια tuple βρίσκεται. Ο δείκτης αυτός ενημερώνεται κάθε φορά που η συνάρτηση ανάθεσης επισκέπτεται μια tuple και τοποθετείται να δείχνει στην αμέσως επόμενη από αυτήν. Όσες σημειωμένες tuples βρει τις αφήνει ως έχουν. Όσες μη σημειωμένες tuples βρει, των οποίων όμως το μέγεθος δεν επαρκεί για την ανάθεση τις σημειώνει! Αυτές οι tuples, αν και ήταν ελεύθερες, θεωρούνται πλέον ως σημειωμένες και δεν μπορούν να χρησιμοποιηθούν για ανάθεση. Αν βρει μια tuple που να μην είναι σημειωμένη, και το μέγεθός της επαρκεί, τότε την σημειώνει και την χρησιμοποιεί για την ανάθεση. Ο βοηθητικός δείκτης δείχνει πάντα στην επόμενη tuple. Αυτό είναι πολύ σημαντικό για να ξέρει η συνάρτηση από πού να συνεχίσει το ψάξιμο στην επόμενη κλήση της. Εξίσου σημαντική είναι η παρατήρηση ότι με αυτόν τον τρόπο κάθε tuple που βρίσκεται πριν από αυτή που δείχνει ο βοηθητικός δείκτης είναι σημειωμένη. Αν ο δείκτης αυτός φτάσει στο τέλος του σωρού, τότε δεν υπάρχει άλλη tuple κατάλληλη για ανάθεση. Σε μια τέτοια περίπτωση όλες οι tuples στο σωρό, αφού βρίσκονται σε προηγούμενες διευθύνσεις από αυτές που δείχνει ο βοηθητικός δείκτης, θα είναι σημειωμένες. Μια τέτοια στρατηγική αν και φαίνεται ότι σπαταλάει τον ελεύθερο χώρο στο σωρό, στην πραγματικότητα είναι απαραίτητη προκειμένου να εξασφαλιστεί ότι όταν η συνάρτηση ανάθεσης μνήμης αποτύχει, τότε όλες οι tuples στο σωρό θα είναι σημειωμένες[4]. Αντιστρέφοντας τη σημασία του bit σήμανσης στον αλγόριθμο, τότε αυτομάτως όλες οι tuples στο σωρό γίνονται μη σημειωμένες! Αμέσως ακολουθεί το στάδιο της σημείωσης του συλλέκτη σκουπιδιών, το οποίο θα βρει έναν καθαρό σωρό και θα σημειώσει όσες tuples είναι ζωντανές. Το στάδιο αυτό δεν θα είχε νόημα να πραγματοποιηθεί αν υπήρχαν ήδη σημειωμένες tuples στο σωρό. Ο βοηθητικός δείκτης θα δείχνει πάλι στην αρχή του σωρού, έτοιμος για την επόμενη κλήση στην συνάρτηση ανάθεσης μνήμης.

5.2.3 Κόστος εκτέλεσης

Ο αλγόριθμος σημείωσης χωρίς σάρωση έχει ένα στάδιο, αυτό της σημείωσης. Κατά το στάδιο αυτό πραγματοποιεί ένα πέρασμα στο σωρό. Το πέρασμα αυτό είναι ανάλογο μόνο των ζωντανών αντικειμένων (tuples) στο σωρό, καθώς μόνο αυτά σημειώνονται.

Ο αλγόριθμος αυτός είναι πολύ γρήγορος στην συλλογή σκουπιδιών, υποφέρει ωστόσο στον τομέα της ανάθεσης μνήμης. Αν και η μεθοδολογία που ακολουθείται στην ανάθεση μνήμης βελτιώνει λίγο την κατάσταση, εντούτοις αυτή εξαρτάται από το μέγεθος και, κυρίως, από την πληρότητα του σωρού. Σε έναν σχετικά άδειο σωρό, η ανάθεση μνήμης γίνεται αρκετά

[4] Αυτό δε θα ήταν εύκολο να γίνει αν είχε ακολουθηθεί η πολιτική καλύτερου–ταιριάσματος.

γρήγορα. Όσο αυξάνεται η πληρότητα του σωρού όμως, η αποδοτικότητα της συνάρτησης ανάθεσης μειώνεται αισθητά.

Από την πλευρά του κόστους σε χώρο, δεν υπάρχουν κάποιες πρόσθετες απαιτήσεις. Και αυτός ο αλγόριθμος μπορεί να διαχειριστεί ολόκληρο το μέγεθος του σωρού που έχει στη διάθεσή του. Χρειάζεται όμως στοίβα για την αναδρομική διάσχιση των αντικειμένων, το μέγεθος της οποίας χρειάζεται να είναι όσο και το μέγιστο μονοπάτι του γράφου που σχηματίζουν τα προσιτά από τις ρίζες αντικείμενα.

5.2.4 Υλοποίηση του αλγορίθμου

Σύμφωνα με τα παραπάνω λοιπόν, στην παρούσα υλοποίηση ο αλγόριθμος αυτός χρησιμοποιεί την πολιτική πρώτου–ταιριάσματος για την ανάθεση μνήμης. Επίσης χρησιμοποιεί τον αλγόριθμο διάσχισης κατά βάθος για να διασχίζει τα αντικείμενα(tuples) αναδρομικά. Επίσης προσπαθεί και αυτός να ενώνει διαδοχικά τμήματα ελεύθερης μνήμης, σχηματίζοντας ένα τμήμα μεγαλύτερου μεγέθους με σκοπό να ελαττώσει τον κατακερματισμό του σωρού. Ως bit σήμανσης χρησιμοποιείται το δεύτερο λιγότερο σημαντικό ψηφίο (lsb) της επικεφαλίδας μιας tuple, αφού το πρώτο χρησιμοποιείται για να εξασφαλιστεί ότι η λέξη αυτή δεν είναι δείκτης στο σωρό.

Ακολουθεί μία συνοπτική περιγραφή της υλοποίησης των συναρτήσεων που απαιτεί η διεπαφή της συλλογής σκουπιδιών.

Στη συνάρτηση `int32_t *gc_algo_alloc(int32_t size)` διατρέχεται ο σωρός μέχρι να βρεθεί ένα μη σημειωμένο (ελεύθερο) τμήμα μνήμης με μέγεθος μεγαλύτερο ή ίσο του `size + 1`. Γίνεται χρήση ενός βοηθητικού δείκτη, ο οποίος για κάθε tuple που επισκέπτεται η συνάρτηση ανάθεσης μνήμης, τοποθετείται να δείχνει στην επόμενή της. Αρχικά ο δείκτης αυτός δείχνει στην αρχή της πρώτης tuple στο σωρό. Όσο καλείται η συνάρτηση ανάθεσης, εκείνος διατρέχει στη σειρά όλες τις tuples του σωρού. Έτσι η συνάρτηση μπορεί να συνεχίζει το ψάξιμο από το σημείο που είχε μείνει την τελευταία φορά που καλέστηκε. Αν ο δείκτης δείχνει στο τέλος του σωρού, αυτό σημαίνει ότι δεν υπάρχουν άλλες tuples να εξεταστούν και η συνάρτηση ανάθεσης επιστρέφει `NULL`. Αλλιώς, κάθε σημειωμένη tuple που συναντά, την αφήνει ως έχει. Αντίθετα για κάθε μη σημειωμένη tuple που συναντά εξετάζει αν το μέγεθός της είναι κατάλληλο για την ανάθεση. Αν δεν είναι εξετάζει μήπως είναι και οι επόμενές της μη σημειωμένες, έτσι ώστε να δημιουργήσει μια tuple μεγαλύτερου μεγέθους. Συνενώνει όσες διαδοχικές μη σημειωμένες tuples βρει μέχρι να δημιουργηθεί μία καταλλήλου μεγέθους. Αν το τελικό μέγεθος της tuple μετά και τις τυχόν συνενώσεις πάλι δεν επαρκεί, τότε την σημειώνει και προχωράει στην επόμενη. Όταν τελικά (με οποιοδήποτε τρόπο) βρεθεί ένα τμήμα ελεύθερης μνήμης με το απαιτούμενο μέγεθος, τότε αυτό χρησιμοποιείται για την ανάθεση. Και εδώ διακρίνουμε δύο περιπτώσεις:

- Το τμήμα να έχει μέγεθος ίσο με το ζητούμενο. Στην περίπτωση αυτή η συνάρτηση ανάθεσης θα παραχωρήσει ολόκληρο το τμήμα. Αρχικά κατασκευάζει την επικεφαλίδα στην πρώτη λέξη του τμήματος. Η επικεφαλίδα περιέχει την τιμή του μεγέθους της tuple ολισθημένη αριστερά 2 κατά θέσεις και το λιγότερο σημαντικό bit της είναι ίσο

με 1. Επίσης η επικεφαλίδα σε αυτήν την περίπτωση όμως σημειώνεται. Η συνάρτηση επιστρέφει δείκτη στη δεύτερη λέξη του τμήματος μνήμης που κατοχυρώθηκε.

- Το τμήμα να έχει μέγεθος μεγαλύτερο από το ζητούμενο. Στην περίπτωση αυτή το τμήμα διασπάται σε δύο κομμάτια. Το πρώτο κομμάτι έχει μέγεθος ίσο με το απαιτούμενο, οπότε και πραγματοποιείται η εκχώρηση της μνήμης σύμφωνα με την πρώτη περίπτωση. Το δεύτερο κομμάτι έχει το μέγεθος που απομένει. Επειδή όμως δεν διαθέτει την κατάλληλη επικεφαλίδα, η συνάρτηση ανάθεσης μνήμης φροντίζει να του φτιάξει μία με το νέο του μέγεθος. Το κομμάτι αυτό παραμένει προς το παρόν μη σημειωμένο.

Ο βοηθητικός δείκτης ενημερώνεται ώστε να δείχνει πάντα στην επόμενη tuple.

Η συνάρτηση `void gc_algo_collect(void)` αρχικά εναλλάσσει την σημασία του bit σήμανσης (δηλαδή αν το 0 είχε τη σημασία του σημειωμένου, τώρα έχει τη σημασία του μη σημειωμένου και αντίστροφα). Ο λόγος που προβαίνει στην συγκεκριμένη ενέργεια είναι ότι η κλήση της γίνεται όταν έχει αποτύχει η συνάρτηση ανάθεσης μνήμης. Αυτό έχει σαν αποτέλεσμα, στον συγκεκριμένο αλγόριθμο, εκείνη τη στιγμή να είναι όλες οι tuples του σωρού σημειωμένες. Με την εναλλαγή αυτή, γίνονται αυτόματα μη σημειωμένες. Στη συνέχεια κάνοντας χρήση της συνάρτησης `gc_walk_roots()`, εφαρμόζει σε κάθε ρίζα την συνάρτηση `void applyMark(int32_t *ptr)` η οποία ελέγχει αν η ρίζα είναι δείκτης στο σωρό και σε περίπτωση που είναι καλεί την συνάρτηση `mark(int32_t *x)` για να σημειώσει αναδρομικά κάνοντας χρήση του αλγόριθμου DFS όλες τις προσιτές tuples από την εκάστοτε ρίζα. Αν η συνάρτηση `mark()` συναντήσει μία ήδη σημειωμένη tuple, την αγνοεί. Αφού σημειωθούν όλες οι ζωντανές tuples στο σωρό, ο συλλέκτης σκουπιδιών ξαναβάζει τον βοηθητικό δείκτη της συνάρτησης ανάθεσης μνήμης να δείχνει στην αρχή του σωρού.

Στη συνάρτηση αρχικοποίησης του συλλέκτη `void gc_algo_init(void)` γίνονται οι απαραίτητες αρχικοποιήσεις για το μέγεθος και την αρχική διεύθυνση του σωρού. Στη συνέχεια φτιάχνεται μια εικονική επικεφαλίδα tuple στην αρχή του σωρού. Έτσι ο σωρός μπορεί να θεωρηθεί ως μια μεγάλη tuple η οποία θα μπορεί να χρησιμοποιηθεί από τη συνάρτηση ανάθεσης μνήμης στη συνέχεια.

Τέλος, η συνάρτηση ολοκλήρωσης του αλγορίθμου `void gc_algo_final(void)` δεν χρειάζεται να κάνει κάποια δουλειά και το σώμα της παρέμεινε κενό.

5.2.5 Πλεονεκτήματα – Μειονεκτήματα

Ο αλγόριθμος σημείωσης χωρίς σάρωση έχει αρκετά κοινά με τον αλγόριθμο σημείωσης σάρωσης. Στα πλεονεκτήματά του συγκαταλέγεται το γεγονός ότι και αυτός μπορεί να διαχειριστεί τις κυκλικές δομές δεδομένων. Επίσης ούτε εδώ απαιτείται η παραγωγή επιπλέον κώδικα από τον μεταγλωττιστή που να παρακολουθεί τις αναγνώσεις – εγγραφές δεικτών. Ο αλγόριθμος σημείωσης χωρίς σάρωση μπορεί να διαχειριστεί ολόκληρο το μέγεθος του σωρού, το οποίο έχει στη διάθεσή του. Επιπλέον προσφέρει ένα πολύ γρηγορότερο στάδιο συλλογής σκουπιδιών, αφού δεν έχει στάδιο σάρωσης του σωρού. Για το λόγο αυτό οι παύσεις που προκαλούνται στο πρόγραμμα είναι συνήθως πού μικρότερες από αυτές του αλγόριθμου σημείωσης σάρωσης.

Η συνάρτηση ανάθεσης μνήμης του αλγορίθμου αυτού προκαλεί μεγαλύτερη επιβάρυνση στην εκτέλεση του προγράμματος, αν και αυτή ισοκατανέμεται καλύτερα στην διάρκεια εκτέλεσης του προγράμματος, καθώς συνήθως σαρώνεται μόνο ένα μικρό κομμάτι του σωρού σε κάθε αίτηση για εκχώρηση νέας μνήμης. Θα μπορούσαμε να πούμε ότι ο αλγόριθμος αυτός πραγματοποιεί, με την ευρεία έννοια του όρου, ένα είδος αυξητικής συλλογής σκουπιδιών.

Επίσης και αυτός υποφέρει από το φαινόμενο του κατακερματισμού του σωρού. Μετά από μερικές εκτελέσεις του οι tuples καταλήγουν να είναι διασκορπισμένες σε κάθε σημείο του σωρού. Αυτό όχι μόνο αυξάνει τις διαρροές μνήμης στο πρόγραμμα, αλλά προκαλεί και απώλεια της τοπικότητας των αντικειμένων. Σε ένα σύστημα με εικονική μνήμη αυτό μπορεί να οδηγήσει σε συνεχείς εναλλαγές (swapping) των σελίδων μεταξύ της πρωτεύουσας και δευτερεύουσας μνήμης.

Κεφάλαιο 6

Μετακινούντες Αλγόριθμοι

Η δεύτερη κατηγορία αλγορίθμων συλλογής σκουπιδιών της εργασίας αυτής, είναι οι μετακινούντες (moving) αλγόριθμοι. Κύριο χαρακτηριστικό αυτών των αλγορίθμων είναι η ικανότητά τους να μετακινούν ή να αντιγράφουν τα ζωντανά αντικείμενα από μία περιοχή του σωρού σε μία άλλη. Τα αντικείμενα τοποθετούνται το ένα μετά το άλλο, χωρίς κενά. Η τοποθέτηση μπορεί να γίνει είτε διατηρώντας την αρχική διάταξη των αντικειμένων, είτε λαμβάνοντας υπ' όψιν τον τρόπο που αυτά συνδέονται, είτε τυχαία. Ανεξάρτητα από τον τρόπο της τοποθέτησης των αντικειμένων, όταν ολοκληρωθεί η διαδικασία ο υπόλοιπος χώρος στο σωρό είναι ελεύθερος χώρος. Ο χώρος αυτός είναι μάλιστα και συνεχόμενος, γεγονός που διευκολύνει την ανάθεση μνήμης σε σχέση με τους μη–μετακινούντες αλγόριθμους.

Στο κεφάλαιο αυτό θα ακολουθήσει η παρουσίαση δύο τέτοιων αλγορίθμων, του αλγορίθμου *σημείωσης συμπύκνωσης* (*mark and compact*) και του αλγορίθμου *διακοπής αντιγραφής* (*stop and copy, semispace*). Πρόκειται περί δύο ανιχνευτικών (tracing) αλγορίθμων που ''σταματούν τον κόσμο''. Αναλυτικά τα χαρακτηριστικά των ανιχνευτικών αλγορίθμων και των αλγορίθμων που ''σταματούν τον κόσμο'' παρουσιάζονται στο κεφάλαιο 2.

6.1 Ο αλγόριθμος Σημείωσης Συμπύκνωσης

6.1.1 Περιγραφή

Ο αλγόριθμος αυτός αποτελεί στην ουσία μια παραλλαγή του αλγόριθμου σημείωσης σάρωσης. Στόχος του είναι να αντιμετωπίσει το πρόβλημα του κατακερματισμού του σωρού. Αυτό το πετυχαίνει μετακινώντας τις ζωντανές tuples στη μία άκρη του, τοποθετώντας τες διαδοχικά την μία μετά την άλλη, και τον ελεύθερο χώρο στην άλλη[HW67]. Η αρχική διάταξη των tuples διατηρείται. Δηλαδή αν μια tuple ήταν πριν από μία άλλη στο σωρό, θα συνεχίσει να είναι και μετά την εκτέλεση του συλλέκτη. Και σε αυτόν τον αλγόριθμο απαιτείται η χρήση ενός bit σήμανσης, έτσι ώστε να μπορούν να διαχωρίζονται οι ζωντανές tuples από τα σκουπίδια.

Κρίνοντας από το όνομά του η μόνη φανερή διαφορά από τον αλγόριθμο σημείωσης σάρωσης είναι ότι το στάδιο της σάρωσης έχει αντικατασταθεί από το στάδιο της συμπύκνωσης. Παρόλα αυτά αλγόριθμος σημείωσης συμπύκνωσης έχει στην ουσία τρία στάδια[CN83]. Στο πρώτο στάδιο πραγματοποιείται κατά τα γνωστά η σημείωση των ζωντανών αντικειμένων.

Πριν από το στάδιο αυτό καμία tuple δεν πρέπει είναι σημειωμένη. Ο αλγόριθμος σημειώνει (mark) όλες τις προσιτές από τις ρίζες tuples κάνοντας και αυτός χρήση κάποιου αλγόριθμου διάσχισης γράφων (συνήθως διάσχιση κατά βάθος). Και εδώ σε περίπτωση που βρεθεί μία ήδη σημειωμένη tuple ο συλλέκτης την αφήνει και συνεχίζει από το σημείο που ήταν πριν την επίσκεψή της. Υπάρχει όμως μια θεμελιώδης διαφορά στο στάδιο της σημείωσης μεταξύ των δύο αλγορίθμων. Στο στάδιο αυτό ο αλγόριθμος σημείωσης συμπύκνωσης όσους δείκτες στο σωρό βρει τους βάζει να δείχνουν σε έναν πίνακα με διευθύνσεις. Ο πίνακας αυτός είναι γνωστός και ως *πίνακας μεταθέσεων* (*relocation table*), επειδή θα περιέχει όλες τις καινούργιες διευθύνσεις των tuples μετά την μετακίνησή τους. Με την ολοκλήρωση αυτού του σταδίου δύο πράγματα γίνονται εμφανή. Πρώτον έχουν σημειωθεί όλες οι ζωντανές tuples και δεύτερον δεν υπάρχει πια δείκτης που να δείχνει στο σωρό. Όλοι οι δείκτες δείχνουν πλέον στον πίνακα μεταθέσεων. Στη συνέχεια ακολουθεί το δεύτερο στάδιο, αυτό της συμπύκνωσης (compact). Στο στάδιο αυτό εξετάζεται γραμμικά όλος ο σωρός για να βρεθούν οι μη σημειωμένες tuples. Κάθε μη σημειωμένη tuple που βρίσκεται, θεωρείται ως ελεύθερος χώρος και αγνοείται από τον συλλέκτη. Κάθε σημειωμένη tuple που συναντά ο συλλέκτης, της καθαρίζει αρχικά το bit σήμανσής της και στη συνέχεια, σε περίπτωση που υπάρχει ελεύθερος χώρος πριν από αυτήν, την μετακινεί έτσι ώστε όλος ο ελεύθερος χώρος που υπήρχε πάνω από τη συγκεκριμένη tuple να βρίσκεται πλέον κάτω από αυτή. Ο ελεύθερος χώρος δεν χάνεται, απλά μεταφέρεται. Άμεση συνέπεια είναι ότι αρκεί να βρεθεί μια μη σημειωμένη tuple στο σωρό για να μετακινηθούν όλες οι επόμενες της. Τέλος ο συλλέκτης αποθηκεύει την διεύθυνση της τρέχουσας tuple, ασχέτως αν την μετακίνησε ή όχι, στον πίνακα μεταθέσεων. Αφού γίνει η διάσχιση ολόκληρου του σωρού, όλες οι tuples έχουν μετακινηθεί στο πάνω μέρος του και όλος ο ελεύθερος χώρος στο κάτω.

Algorithm 3 Mark & Compact

```
for each root r do
  if r is heap pointer then
    mark(r)
    r ← pointer to reloc. table
compact()
fixPointers()

where:
mark(object x):
if x is not marked then
  x.mark_bit ← marked
  for every field f_i of x do
    if x.f_i is heap pointer then
      mark(x.f_i)
      x.f_i ← pointer to reloc. table
```

```
compact():
for each object p in heap do
  if p is marked then
    p.marked_bit ← not marked
    if free space exists before p then
      slide p up to eliminate free space
    reloc. table ← address of p
  else p is free space
```

```
fixPointers():
for each field f_i of every object p in heap do
  if p.f_i is reloc. table pointer then
    p.f_i ← correct address from reloc. table
for each root r do
  if r is reloc. table pointer then
    r ← correct address from reloc. table
```

Ο ελεύθερος χώρος σχηματίζει ένα μεγάλο τμήμα συνεχόμενης μνήμης, το οποίο όπως θα δούμε στη συνέχεια επιταχύνει σημαντικά την διαδικασία ανάθεσης μνήμης. Το πρόβλημα με αυτήν την μετακίνηση είναι ότι όσες tuples μετακινήθηκαν, βρίσκονται τώρα σε διαφορετική διεύθυνση στο σωρό και οι δείκτες που έδειχναν σε αυτές δεν μπορούν πλέον να χρησιμοποιηθούν για την προσπέλασή τους διότι δείχνουν ακόμα στην παλιά τους διεύθυνση. Τη λύση στο πρόβλημα αυτό έρχεται να δώσει το τρίτο στάδιο του αλγορίθμου σημείωσης συμπύκνωσης, που είναι υπεύθυνο για την ενημέρωση των δεικτών με τις καινούργιες διευθύνσεις. Το έδαφος για το στάδιο αυτό έχει ήδη προετοιμαστεί από το στάδιο της σημείωσης, κατά το οποίο όλοι οι δείκτες στο σωρό τροποποιήθηκαν ώστε να δείχνουν στον πίνακα μεταθέσεων. Στο στάδιο αυτό σαρώνεται πάλι ο σωρός, αλλά και οι ρίζες του προγράμματος, και όσοι δείκτες βρεθούν να δείχνουν στον πίνακα μεταθέσεων, ενημερώνονται με τη νέα διεύθυνση, η οποία αντιστοιχεί στις tuples που πρέπει πραγματικά να δείχνουν. Ο πίνακας μεταθέσεων, όπως αναφέρθηκε και προηγουμένως, έχει ήδη ενημερωθεί από το δεύτερο στάδιο, αυτό της συμπύκνωσης.

6.1.2 Ανάθεση μνήμης

Η ανάθεση μνήμης στον αλγόριθμο σημείωσης συμπύκνωσης πραγματοποιείται εύκολα και γρήγορα, αφού όλος ο ελεύθερος χώρος είναι συνεχόμενος (contiguous). Δεν χρειάζεται σάρωμα του σωρού ή χρήση κάποιας βοηθητικής δομής δεδομένων (π.χ. συνδεδεμένη λίστα). Το μόνο που χρειάζεται είναι ένας δείκτης στην αρχή του ελεύθερου χώρου. Αν ο ελεύθερος χώρος επαρκεί για την ανάθεση, τότε παραχωρείται ένα τμήμα μνήμης με το ζητούμενο μέγεθος στο πρόγραμμα και ο δείκτης απλά αυξάνεται για να δείχνει πάλι στην αρχή του εναπομένοντα ελεύθερου χώρου.

6.1.3 Ο πίνακας μεταθέσεων

Όταν μετακινείται μία tuple, όλοι οι δείκτες που έδειχναν σε αυτήν γίνονται αυτόματα αιωρούμενες αναφορές. Όλοι αυτοί οι δείκτες, το πλήθος των οποίων δεν είναι γνωστό εκ των προτέρων, πρέπει να βρεθούν και να ενημερωθούν με τη νέα διεύθυνση της tuple στο σωρό. Η νέα διεύθυνση γίνεται γνωστή μόνο τη στιγμή που θα μετακινηθεί η tuple. Μια σκέψη θα ήταν να τοποθετούσαμε στην παλιά της διεύθυνση, μία ταμπέλα με τη νέα της διεύθυνσή. Έτσι όσοι δείκτες έδειχναν σε αυτήν, θα ενημερωθούν για τη νέα της διεύθυνση. Αυτή η προσέγγιση όμως δε θα δουλέψει στην πράξη καθώς υπάρχει μεγάλη πιθανότητα μια άλλη tuple να μετακινηθεί στην θέση αυτή και να σβηστεί η ταμπέλα. Πρέπει να βρεθεί ένας αποδοτικός τρόπος ώστε όλοι οι δείκτες προς τις tuples που μετακινήθηκαν να ενημερώνονται για τις νέες τους διευθύνσεις.

Σαφώς το να σαρώνεται ολόκληρος ο σωρός κάθε φορά που μετακινείται μια tuple και να ενημερώνονται οι δείκτες που δείχνουν σε αυτήν είναι ασύμφορο. Επίσης δε συμφέρει να διατηρούμε τόσες συνδεδεμένες λίστες όσες και οι tuples στο σωρό και σε κάθε λίστα να έχουμε τη διεύθυνση όλων των δεικτών που δείχνουν σε καθεμιά τους, επειδή κάθε φορά που θα μετακινείται μια tuple θα πρέπει να σαρωθούν όλες οι λίστες, μέχρι να βρεθεί η λίστα που αντιστοιχεί στην συγκεκριμένη tuple και η οποία θα περιέχει όλες τις διευθύνσεις των δεικτών

που δείχνουν σε αυτήν. Ενώ στη συνέχεια θα πρέπει να διασχίσουμε τις λίστες αυτές και να αντικαθιστούμε την παλιά τιμή των δεικτών με την καινούρια. Εκτός του αυξημένου κόστους εκτέλεσης, κάτι τέτοιο θα απαιτούσε και αρκετό χώρο[1].

Ο πίνακας μεταθέσεων μπορεί να κάνει την ίδια δουλειά με το μισό κόστος σε χώρο. Πιο αναλυτικά το ελάχιστο μέγεθος μιας tuple είναι 2 λέξεις. Για έναν σωρό με μέγεθος $2N$ λέξεων, το πλήθος των tuples είναι το πολύ N. N tuples σημαίνουν και N διευθύνσεις. Αφού κάθε διεύθυνση έχει μέγεθος 1 λέξης, ένας πίνακας μεγέθους N λέξεων μπορεί να αποθηκεύσει όλες τις απαραίτητες διευθύνσεις. Πώς όμως θα ενημερωθούν οι δείκτες που δείχνουν σε tuples που έχουν μετακινηθεί; Κάθε δείκτης στο σωρό, θα αντιστοιχεί σε μία συγκεκριμένη θέση του πίνακα αυτού. Το εύρος διευθύνσεων του σωρού παραμένει $2N$ και αυτές πρέπει να απεικονιστούν σε έναν πίνακα N θέσεων. Για να γίνει κάτι τέτοιο, θα πρέπει σε μια θέση του πίνακα να αντιστοιχούν 2 διευθύνσεις του σωρού. Πρέπει βέβαια να αποκλειστεί το ενδεχόμενο 2 διαφορετικές tuples να αντιστοιχούν στην ίδια θέση του πίνακα. Την λειτουργία αυτή έχει αναλάβει μία συνάρτηση (με τη μαθηματική έννοια του όρου) απεικόνισης, η οποία απεικονίζει κάθε διεύθυνση του σωρού, ο οποίος έχει μέγεθος $2N$ σε μία θέση του πίνακα, ο οποίος έχει μέγεθος N, σύμφωνα με τον τύπο:

$$\textit{θέση πίνακα} = (\textit{διεύθυνση σωρού} \ / \ 8) \ \textbf{mod} \ N$$

Με απλά μαθηματικά αποδεικνύεται ότι μόνο δύο διαδοχικές διευθύνσεις στο σωρό θα βρεθούν στην ίδια θέση του πίνακα. Αυτό όμως δεν αποτελεί πρόβλημα αφού είναι αδύνατο δύο tuples, με ελάχιστο μέγεθος 2 λέξεις η κάθε μία, να καταλαμβάνουν διαδοχικές θέσεις μνήμης στο σωρό.

Ας δούμε τώρα τη χρήση του πίνακα αυτού. Αρχικά όλοι οι δείκτες που δείχνουν σε μία tuple η οποία θα μετακινηθεί, πρέπει να τροποποιηθούν ώστε δείχνουν στην κατάλληλη θέση του πίνακα μεταθέσεως. Η θέση αυτή υπολογίζεται με χρήση της παραπάνω συνάρτησης απεικόνισης. Επειδή δεν είναι εκ των προτέρων γνωστό αν θα μετακινηθεί μια tuple, εφαρμόζουμε την τακτική αυτή σε κάθε δείκτη στο σωρό που συναντάμε. Αυτό θέλει ένα επιπλέον πέρασμα του σωρού. Για να αποφύγουμε το επιπλέον πέρασμα εκμεταλλευόμαστε το γεγονός ότι το στάδιο της σημείωσης κάνει έτσι κι αλλιώς ένα πέρασμα από όλες τις ζωντανές tuples. Παράλληλα με το πέρασμα αυτό λοιπόν, κάνουμε την απαραίτητη τροποποίηση σε όλους τους δείκτες στο σωρό με σχεδόν μηδενική επιβάρυνση στο χρόνο εκτέλεσης. Προφανώς τα σκουπίδια δεν μας ενδιαφέρουν.

Στη συνέχεια πρέπει να μπουν στον πίνακα οι πραγματικές διευθύνσεις που έχουν οι tuples. Αυτό γίνεται κατά το στάδιο της συμπύκνωσης. Κατά την διάσχιση του σωρού ο συλλέκτης χρησιμοποιεί την αρχική διεύθυνση κάθε σημειωμένης tuple που συναντάει για να βρει τη θέση του πίνακα που της αντιστοιχεί, και στη συνέχεια αποθηκεύει την τελική διεύθυνσή της

[1] Στη χειρότερη περίπτωση ένας σωρός με μέγεθος $2N$ μπορεί να έχει το πολύ N tuples. Για κάθε tuple θέλουμε μία λίστα, καθεμιά από τις οποίες θα έχει όσα στοιχεία όσα και οι δείκτες που θα δείχνουν στην tuple που αυτή αντιστοιχεί. Έτσι απαιτείται χώρος τουλάχιστον $2N$ για να αποθηκευτούν οι λίστες αυτές, αφού οι δείκτες προς τις tuples μπορεί να είναι και εκτός σωρού (π.χ. στις ρίζες).

σε αυτή τη θέση. Σε περίπτωση που δεν μετακινηθεί η tuple, η τελική της διεύθυνση συμπίπτει με την αρχική.

Αφού ολοκληρωθεί το στάδιο της συμπύκνωσης, ο πίνακας έχει όλες τις τρέχουσες διευθύνσεις των tuples στο σωρό. Με ένα τρίτο πέρασμα στο σωρό, διασχίζονται γραμμικά όλες οι, συμπυκνωμένες πλέον, ζωντανές tuples. Κάθε δείκτης στον πίνακα, δείχνει σε μία συγκεκριμένη θέση του. Σε αυτή τη θέση βρίσκεται η σωστή διεύθυνση της tuple που ζητάμε.

6.1.4 Κόστος εκτέλεσης

Ο αλγόριθμος σημείωσης συμπύκνωσης πραγματοποιεί τρία περάσματα στο σωρό. Το πρώτο πέρασμα γίνεται στο στάδιο της σημείωσης και είναι ανάλογο μόνο των ζωντανών αντικειμένων, αφού μόνο αυτά σημειώνονται.

Το δεύτερο στάδιο, αυτό της συμπύκνωσης είναι ανάλογο του μεγέθους του σωρού, καθώς σαρώνεται ολόκληρος. Στο κόστος της διάσχισης του σωρού πρέπει να προστεθεί και το κόστος της μετακίνησης των tuples.

Το τρίτο στάδιο πραγματοποιεί και αυτό ένα πέρασμα στο σωρό. Το πέρασμα αυτό είναι ανάλογο μόνο των ζωντανών αντικειμένων. Δεν χρειάζεται η διάσχιση ολόκληρου του σωρού αφού τη στιγμή που γίνεται έχουν φύγει όλα τα σκουπίδια.

Ο αλγόριθμος αυτός, πέρα από την στοίβα που απαιτεί για την αναδρομική διάσχιση των αντικειμένων, χρειάζεται χώρο για να αποθηκεύσει τον πίνακα μεταθέσεων. Για σωρό μεγέθους $2N$ λέξεων, χρειάζεται N επιπλέον λέξεις για τον πίνακα αυτόν. Ισοδύναμα χρησιμοποιεί το $1/3$ του συνολικού μεγέθους του σωρού για τον πίνακα και με τα υπόλοιπα $2/3$ ικανοποιεί τις απαιτήσεις του προγράμματος σε μνήμη.

Τέλος το κόστος ανάθεσης μνήμης είναι το μικρότερο δυνατό, καθώς η ελεύθερη μνήμη είναι συνεχόμενη. Αυτό αντισταθμίζει κάπως το κόστος των τριών περασμάτων στο σωρό, ελαττώνοντας το συνολικό κόστος εκτέλεσης.

6.1.5 Υλοποίηση του αλγορίθμου

Και αυτός ο αλγόριθμος χρησιμοποιεί ως bit σήμανσης το δεύτερο λιγότερο σημαντικό ψηφίο (lsb) της επικεφαλίδας μιας tuple, αφού το πρώτο χρησιμοποιείται για να εξασφαλιστεί ότι η λέξη αυτή δεν είναι δείκτης στο σωρό. Επίσης δεν επιτρέπει να υπάρχει κενός χώρος ανάμεσα στις tuples. Η υλοποίησή του έχει γίνει στα πλαίσια που ορίζει η διεπαφή συλλογής σκουπιδιών.

Η συνάρτηση `int32_t *gc_algo_alloc(int32_t size)` πραγματοποιεί την ανάθεση μνήμης. Χρησιμοποιεί έναν δείκτη στην αρχή του ελεύθερου χώρου, ο οποίος είναι συνεχόμενος. Αν ο χώρος αυτός επαρκεί για την ανάθεση μιας tuple με μέγεθος `size + 1` τότε κατοχυρώνει ένα κομμάτι με το μέγεθος αυτό στην αρχή του χώρου, φτιάχνει την κατάλληλη επικεφαλίδα (το κομμάτι θεωρείται μη σημειωμένο), και επιστρέφει έναν δείκτη στην δεύτερη λέξη του τμήματος που μόλις κατοχυρώθηκε. Ο δείκτης αυξάνεται και δείχνει στην αρχή του ελεύθερου χώρου που ακολουθεί αμέσως μετά το κομμάτι αυτό. Αν ο διαθέσιμος χώρος δεν επαρκεί για την ανάθεση, η συνάρτηση επιστρέφει `NULL`.

Η συνάρτηση `void gc_algo_collect(void)` σηματοδοτεί την κλήση του συλλέκτη σκουπιδιών. Αρχικά ο συλλέκτης διατρέχει όλες τις ρίζες του προγράμματος με τη συνάρτηση `gc_walk_roots()`, εφαρμόζοντας σε κάθε μία την συνάρτηση `applyMark(int32_t *ptr)` η οποία ελέγχει αν η ρίζα είναι δείκτης στο σωρό και σε περίπτωση που είναι κάνει δύο πράγματα. Πρώτον καλεί την συνάρτηση `mark(int32_t *x)` για να σημειώσει αναδρομικά κάνοντας χρήση του αλγόριθμου DFS όλες τις προσιτές tuples από την εκάστοτε ρίζα. Δεύτερον με χρήση της συνάρτησης απεικόνισης, βάζει τη ρίζα να δείχνει στην κατάλληλη θέση του πίνακα μεταθέσεων. Κατά την αναδρομική κλήση της η `mark()` κάνει το ίδιο με κάθε δείκτη στο σωρό που βρίσκει, ενώ παράλληλα αγνοεί όσες ήδη σημειωμένες tuples βρει. Στη συνέχεια καλείται η συνάρτηση `compact()`, η οποία είναι υπεύθυνη για την μετακίνηση των ζωντανών tuples στην αρχή του σωρού. Η συνάρτηση αυτή ξεκινάει μία γραμμική διάσχιση του σωρού από την αρχή του. Σε κάθε σημειωμένη tuple που συναντά καθαρίζει το bit σήμανσής της και αν υπάρχει ελεύθερος χώρος πριν από αυτήν, την μετακινεί προς τα πάνω τόσες θέσεις όσες και το μέγεθος του ελεύθερου χώρου. Ασχέτως με το αν την μετακινήσει τελικά η όχι, ενημερώνει τον πίνακα μεταθέσεων για την τρέχουσα θέση της, και συνεχίζει την ίδια διαδικασία και στην επόμενη. Αφού ολοκληρωθεί το στάδιο αυτό, τότε καλείται η συνάρτηση `fixPointers()`, η οποία πραγματοποιεί και αυτή με τη σειρά της ένα πέρασμα στο σωρό φτιάχνοντας τους δείκτες που έδειχναν στον πίνακα μεταθέσεων και τους κάνει να δείχνουν στις νέες διευθύνσεις των tuples. Το ίδιο κάνει και για όλες τις ρίζες.

Η συνάρτηση αρχικοποίησης του συλλέκτη `void gc_algo_init(void)` κάνει τις απαραίτητες αρχικοποιήσεις για το ολικό μέγεθος και την αρχική διεύθυνση του σωρού. Από το ολικό μέγεθος δεσμεύει το 1/3 για χρήση στον πίνακα μεταθέσεων και αρχικοποιεί και έναν δείκτη στην αρχή του. Με βάση αυτόν τον δείκτη θα γίνεται η προσπέλαση του πίνακα. Τα υπόλοιπα 2/3 τα χρησιμοποιεί ως σωρό.

Τέλος, η συνάρτηση ολοκλήρωσης του αλγορίθμου `void gc_algo_final(void)` δεν χρειάστηκε να κάνει κάποια δουλειά και το σώμα της παρέμεινε κενό.

6.1.6 Πλεονεκτήματα – Μειονεκτήματα

Βασικότερο πλεονέκτημα του αλγορίθμου σημείωσης συμπύκνωσης είναι ότι αποτρέπει τον κατακερματισμό του σωρού. Μάλιστα το καταφέρνει αυτό θυσιάζοντας μόνο το 1/3 του σωρού, χρησιμοποιώντας τα υπόλοιπα 2/3 ως ωφέλιμο μέγεθος σωρού, σε αντίθεση με άλλους μετακινούντες αλγορίθμους, όπως για παράδειγμα τον αλγόριθμο διακοπής αντιγραφής ο οποίος μπορεί να χρησιμοποιήσει μόνο το 1/2 του μεγέθους του σωρού ως ωφέλιμο. Άμεση συνέπεια της εξάλειψης του κατακερματισμού είναι ότι ο ελεύθερος χώρος είναι συνεχόμενος και η ανάθεση μνήμης γίνεται πολύ γρήγορα. Επιπλέον, τα μακρόβια αντικείμενα, τα οποία καταλήγουν να βρίσκονται στην αρχή του σωρού, είναι απίθανο να μετακινηθούν σε κάποια από τις επόμενες κλήσεις του αλγορίθμου. Στα πλεονεκτήματά του συγκαταλέγεται και το γεγονός ότι μπορεί να διαχειριστεί τις κυκλικές δομές δεδομένων. Επίσης δεν απαιτεί την παραγωγή επιπλέον κώδικα από τον μεταγλωττιστή ο οποίος να παρακολουθεί τις αναγνώσεις – εγγραφές δεικτών.

Από την άλλη τα τρία περάσματα που κάνει στον σωρό προσθέτουν μια ιδιαίτερη επιβάρυνση στην εκτέλεση του προγράμματος και ιδιαίτερα στις παύσεις που προκαλούνται, καθώς δεν υπάρχει τρόπος να διακοπούν τα περάσματα αυτά. Αυτό τον καθιστά απαγορευτικό για διαδραστικές εφαρμογές ή εφαρμογές πραγματικού χρόνου. Τέλος αντίθετα με τους δύο μη–μετακινούντες αλγόριθμους που παρουσιάστηκαν στο προηγούμενο κεφάλαιο, ο αλγόριθμος αυτός δεν μπορεί να χρησιμοποιήσει για αναθέσεις μνήμης ολόκληρο το σωρό που έχει στη διάθεσή του.

6.2 Ο αλγόριθμος Διακοπής Αντιγραφής

6.2.1 Περιγραφή

Ο αλγόριθμος αυτός παρουσιάστηκε πρώτη φορά από τον Marvin L. Minsky[Min63] το 1963 και αποτελεί την τρίτη κλασσική μέθοδο συλλογής σκουπιδιών (μετά την καταμέτρηση αναφορών και τη σημείωση σάρωση). Παραλλαγές του αλγορίθμου αυτού χρησιμοποιούνται ακόμη και σήμερα, με δημοφιλέστερη αυτή που παρουσίασε ο C.J. Cheney[Che70] το 1970.

Κύριο γνώρισμα του αλγορίθμου διακοπής αντιγραφής είναι ότι τα ζωντανά αντικείμενα (tuples) αντιγράφονται διαδοχικά από μία περιοχή του σωρού, σε μία άλλη. Με τον τρόπο αυτό αποτρέπεται ο κατακερματισμός του σωρού.

Ο τρόπος με τον οποίο γίνεται αυτή η αντιγραφή είναι ο ακόλουθος. Συνήθως ο σωρός χωρίζεται σε δύο περιοχές, ίσου μεγέθους (semispace). Μόνο μία από τις δύο αυτές περιοχές μπορεί να είναι ενεργή ανά πάσα στιγμή. Η ενεργή περιοχή χρησιμοποιείται για να ικανοποιήσει τις ανάγκες του προγράμματος σε μνήμη. Όταν εξαντληθεί ο διαθέσιμος χώρος στην ενεργή περιοχή, τότε καλείται ο συλλέκτης σκουπιδιών, ο οποίος κάνοντας χρήση ενός αλγορίθμου διάσχισης γράφων, διασχίζει όλες τις προσιτές από τις ρίζες tuples. Κάθε tuple που επισκέπτεται ο συλλέκτης την αντιγράφει από την ενεργή περιοχή (from–space) του σωρού στην ανενεργή (to–space). Η ανενεργή περιοχή του σωρού αρχικά δεν περιέχει τίποτα και θεωρείται ελεύθερος ενιαίος χώρος. Η τοποθέτηση των tuples στην ανενεργή περιοχή του σωρού γίνεται σειριακά, χωρίς δηλαδή να υπάρχουν κενά μεταξύ τους. Επειδή οι δύο περιοχές του σωρού έχουν το ίδιο μέγεθος, είναι βέβαιο ότι οι tuples της ενεργής περιοχής θα χωρέσουν στην ανενεργή. Όπως είναι φυσικό, η διεύθυνση των tuples μετά την αντιγραφή αλλάζει, με αποτέλεσμα όλοι οι δείκτες σε αυτές γίνονται μετέωρες αναφορές. Για το λόγο αυτό, σε κάθε tuple που αντιγράφεται, ο αλγόριθμος τοποθετεί στην παλιά της διεύθυνση, η οποία βρίσκεται στην ενεργή περιοχή του σωρού, έναν δείκτη, ο οποίος δείχνει στη νέα της διεύθυνση, στην ανενεργή περιοχή του σωρού. Ο δείκτης αυτός ονομάζεται *δείκτης προωθήσεως* (*forwarding pointer*). Η δουλειά του δείκτη προωθήσεως είναι διπλή. Πρώτον καθιστά δυνατή την ενημέρωση των δεικτών που δείχνουν στην παλιά διεύθυνση της tuple και δεύτερον φανερώνει ότι η συγκεκριμένη tuple έχει ήδη επισκεφθεί από τον αλγόριθμο. Η ενημέρωση των δεικτών δεν απαιτεί επιπλέον πέρασμα του σωρού, καθώς μπορεί να ενσωματωθεί στο πρώτο πέρασμα, αφού στη διάσχιση του γράφου που σχηματίζουν οι ζωντανές tuples, ο συλλέκτης ακολουθεί όλους τους δείκτες που δείχνουν στο σωρό. Κάθε τέτοιος δείκτης που δείχνει σε μία

tuple που θα μετακινηθεί μπορεί να ενημερωθεί αμέσως μόλις αυτή μετακινηθεί. Αν ο δείκτης δείχνει σε tuple που έχει ήδη μετακινηθεί, τότε ενημερώνεται από τον δείκτη προωθήσεως που υπάρχει στην παλιά της διεύθυνση. Αυτό εγγυάται τόσο ότι θα ενημερωθούν όλοι οι δείκτες, όσο και ότι θα τερματιστεί η διαδικασία της αντιγραφής ακόμη και στην παρουσία κυκλικών δομών δεδομένων. Με την ολοκλήρωση της διαδικασίας της αντιγραφής όλες οι ζωντανές tuples έχουν αντιγραφεί στην ανενεργή περιοχή του σωρού. Επιπλέον αν ο συνολικός χώρος που καταλαμβάνουν είναι μικρότερος από το μέγεθος της περιοχής αυτής, τότε υπάρχει και (συνεχόμενος) ελεύθερος χώρος στο τέλος του σωρού. Ό,τι περιέχει η ενεργή περιοχή πλέον θεωρείται σκουπίδι. Στο σημείο αυτό ο αλγόριθμος καθιστά την ανενεργή περιοχή του σωρού ενεργή και την μέχρι πρότινος ενεργή την καθιστά ανενεργή. Έτσι με μια απλή αντιστροφή του ρόλου των περιοχών συλλέγονται όλα τα σκουπίδια στο σωρό.

Algorithm 4 Stop & Copy – Semispace

```
for each root r do
  if r is heap1 pointer then
    r ← copy(r)
swap(heap1, heap2)
```

where:

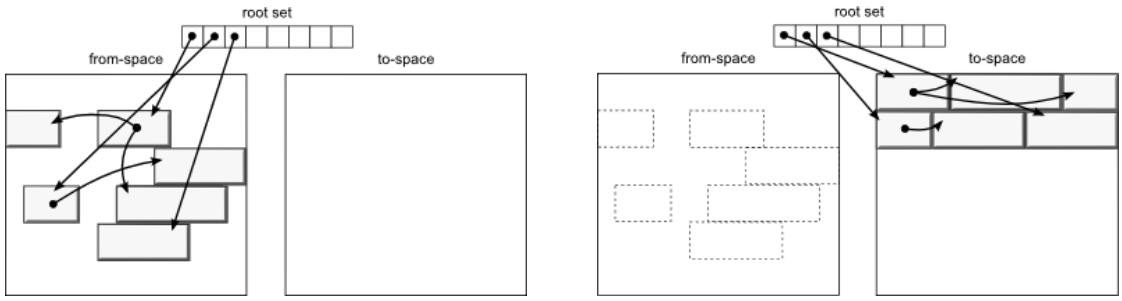
```
copy(object x):
if x is visited then
  return x.fpointer
else x.visited_bit ← true
  for every field f_i in x do
    move x.f_i from heap1 to heap2
  x.fpointer ← address of x' in heap2
  for every field f_i of x do
    if x.f_i is heap1 pointer then
      x.f_i ← copy(x.f_i)
return x.fpointer
```

6.2.2 Ανάθεση μνήμης

Ο ελεύθερος χώρος είναι πάντα συνεχόμενος (contiguous) στον αλγόριθμο διακοπής αντιγραφής. Αυτό απαιτεί απλά τη χρήση ενός δείκτη στην αρχή του ελεύθερου χώρου, χωρίς την ανάγκη για βοηθητικές δομές δεδομένων, γεγονός που καθιστά την ανάθεση μνήμης ταχύτατη. Όλη η μνήμη παραχωρείται κάθε φορά από την ενεργή περιοχή του σωρού. Αν ο ελεύθερος χώρος της περιοχής αυτής επαρκεί για την ανάθεση, τότε παραχωρείται ένα τμήμα μνήμης με το ζητούμενο μέγεθος στο πρόγραμμα και ο δείκτης αυξάνεται έτσι ώστε να δείχνει πάλι στην αρχή του εναπομένοντα ελεύθερου χώρου.

6.2.3 Κόστος εκτέλεσης

Ο αλγόριθμος διακοπής αντιγραφής έχει ένα στάδιο. Σε αυτό το στάδιο πραγματοποιεί ένα πέρασμα στο σωρό, το οποίο είναι ανάλογο μόνο των ζωντανών tuples. Σε κάθε πέρασμα οι tuples αντιγράφονται από τη μία περιοχή της μνήμης στην άλλη, οπότε στο παραπάνω κόστος πρέπει να συνυπολογιστεί και το κόστος της αντιγραφής, το οποίο προκαλεί μία



(α΄) Πριν (β΄) Μετά

Σχήμα 6.1: Ο σωρός πριν και μετά την εκτέλεση του αλγορίθμου διακοπής αντιγραφής

πρόσθετη επιβάρυνση. Συνεπώς κόστος του αλγορίθμου αυξάνεται κι άλλο με την αύξηση της πληρότητας του σωρού, αφού η επιβάρυνση που προκαλεί η αντιγραφή των tuples είναι πολύ μεγαλύτερη από αυτή που θα προκαλούσε μια απλή σημείωσή τους.

Επιπλέον ο αλγόριθμος αυτός, πέρα από την στοίβα που απαιτεί για την αναδρομική διάσχιση των tuples, χωρίζει και τον σωρό σε δύο ισομεγέθη τμήματα εκ των οποίων μόνο το ένα μπορεί να χρησιμοποιηθεί κάθε φορά.

Τέλος το κόστος ανάθεσης μνήμης είναι το μικρότερο δυνατό, καθώς η ελεύθερη μνήμη είναι συνεχόμενη. Αυτό σε συνδυασμό με το μοναδικό πέρασμα που κάνει στο σωρό, τον καθιστά έναν από τους γρηγορότερους αλγόριθμους για τη συλλογή σκουπιδιών, παρόλο που θυσιάζει το μισό μέγεθος του σωρού για να μπορέσει να λειτουργήσει.

6.2.4 Υλοποίηση του αλγορίθμου

Στην παρούσα υλοποίηση ο χρησιμοποιείται ο αλγόριθμος διάσχισης κατά βάθος (DFS) για τη διάσχιση όλων των προσιτών από τις ρίζες αντικειμένων (tuples). Αυτή η επιλογή έγινε επειδή με αυτό τον τρόπο αντικείμενα που δείχνουν το ένα στο άλλο αντιγράφονται σε γειτονικές θέσεις με αποτέλεσμα την καλύτερη τοπικότητά τους. Ο δείκτης προωθήσεως αποθηκεύεται στην παλιά διεύθυνση και συγκεκριμένα στην επικεφαλίδα μιας tuple που μόλις μετακινήθηκε.

Ακολουθεί μια συνοπτική περιγραφή της υλοποίησης των συναρτήσεων του συλλέκτη διακοπής αντιγραφής που απαιτεί από αυτόν η διεπαφή συλλογής σκουπιδιών.

Η συνάρτηση `int32_t *gc_algo_alloc(int32_t size)` πραγματοποιεί την ανάθεση μνήμης. Χρησιμοποιεί έναν βοηθητικό δείκτη στην αρχή του ελεύθερου χώρου της ενεργής πάντα περιοχής του σωρού. Αν ο χώρος αυτός, ο οποίος είναι συνεχόμενος, επαρκεί για την ανάθεση μιας tuple με μέγεθος `size + 1` τότε κατοχυρώνει ένα κομμάτι με το μέγεθος αυτό στην αρχή του χώρου, φτιάχνει την κατάλληλη επικεφαλίδα με το μέγεθός του, και επιστρέφει έναν δείκτη στην δεύτερη λέξη του τμήματος που μόλις κατοχυρώθηκε. Ο βοηθητικός δείκτης αυξάνεται και δείχνει στην αρχή του ελεύθερου χώρου που απομένει μετά την ανάθεση. Σε περίπτωση που ο διαθέσιμος χώρος δεν επαρκεί για την ανάθεση, η συνάρτηση επιστρέφει `NULL`.

Η συνάρτηση `void gc_algo_collect(void)` ενεργοποιεί την συλλογή σκουπιδιών. Αρ-

χικά ο συλλέκτης διατρέχει όλες τις ρίζες του προγράμματος κάνοντας χρήση της συνάρτησης `gc_walk_roots()`, εφαρμόζοντας σε κάθε μία την συνάρτηση `copyTuple(int32_t *ptr)`. Η συνάρτηση αυτή επιστρέφει έναν δείκτη στην ανενεργή περιοχή του σωρού, ο οποίος είναι η νέα διεύθυνση της tuple που έχει μετακινηθεί. Ξεκινάει από τις ρίζες και διασχίζει αναδρομικά κατά βάθος τις ζωντανές tuples, οι οποίες βρίσκονται στην ενεργή περιοχή του σωρού, ακολουθώντας κάθε δείκτη στην ενεργή περιοχη που αυτές έχουν. Όταν επιστρέψει η συνάρτηση, οι δείκτες αυτοί ενημερώνονται για τη νέα διεύθυνση της tuple στην οποία έδειχναν. Αν η συνάρτηση συναντήσει μια tuple που έχει μετακινηθεί, τότε το μόνο που κάνει είναι να επιστρέψει τον δείκτη προωθήσεως που έχει τοποθετηθεί στην επικεφαλίδα της. Οι δείκτες προωθήσεως δείχνουν στην ανενεργή περιοχή του σωρού, οπότε δεν υπάρχει κίνδυνος να μπερδευτούν με τους υπόλοιπους. Αν η συνάρτηση συναντήσει μια tuple που δεν έχει μετακινηθεί ακόμα, την αντιγράφει στην πρώτη διαθέσιμη θέση της ανενεργής περιοχής. Η διεύθυνση της ανενεργής περιοχής στην οποία αντιγράφεται, αποτελεί τη νέα διεύθυνση της tuple και αποθηκεύεται στην επικεφαλίδα της πίσω στην ενεργή περιοχή του σωρού. Αναδρομικά θα κάνει το ίδιο με όσους δείκτες έχει η tuple αυτή, και τελικά θα επιστρέψει τον δείκτη προωθήσεως της.

Η συνάρτηση αρχικοποίησης του συλλέκτη `void gc_algo_init(void)` κάνει τις απαραίτητες αρχικοποιήσεις για το ολικό μέγεθος και την αρχική διεύθυνση του σωρού. Στη συνέχεια χωρίζει το σωρό σε δύο ισομεγέθη τμήματα, και ορίζει το πρώτο ως ενεργό και το δεύτερο ως μη ενεργό. Επίσης αρχικοποιεί έναν δείκτη στην αρχή κάθε τμήματος.

Τέλος, η συνάρτηση ολοκλήρωσης του αλγορίθμου `void gc_algo_final(void)` δεν χρειάστηκε να κάνει κάποια δουλειά και το σώμα της παρέμεινε κενό.

6.2.5 Πλεονεκτήματα – Μειονεκτήματα

Το κυριότερο πλεονέκτημα του αλγορίθμου διακοπής αντιγραφής είναι το γεγονός ότι δεν αφήνει τον σωρό κατακερματισμένο. Επιπλέον, ο τρόπος που το καταφέρνει αυτό είναι βέλτιστος, αφού πραγματοποιεί μόνο ένα πέρασμα στο σωρό, το κόστος του οποίου μάλιστα εξαρτάται μόνο από το πλήθος των ζωντανών tuples[App87]. Αυτό τον καθιστά έναν από τους γρηγορότερους αλγόριθμους για τη συλλογή σκουπιδιών. Μπορεί επίσης να χειριστεί κυκλικές δομές δεδομένων, ενώ δεν απαιτεί την παραγωγή κώδικα από την μεταγλωττιστή για την παρακολούθηση εγγραφών – αναγνώσεων σε δείκτες.

Το προφανές μειονέκτημά του είναι ότι από τον σωρό που του έχει διατεθεί, μπορεί να χρησιμοποιήσει μόνο τον μισό κάθε φορά. Αυτό σημαίνει πως για ένα πρόγραμμα που απαιτεί N λέξεις μνήμης, απαιτείται σωρός μεγέθους $2N$. Αυτό όμως επηρεάζει λιγότερο τα σημερινά συστήματα, που έχουν πολύ χαμηλότερο κόστος σε φυσική μνήμη σε σχέση με παλαιότερα.

Κεφάλαιο 7

Γενετικοί Αλγόριθμοι

Οι αλγόριθμοι που έχουν παρουσιαστεί ως τώρα έχουν ένα κοινό χαρακτηριστικό· σε κάθε κύκλο συλλογής σκουπιδιών ασχολούνται με όλα τα ζωντανά αντικείμενα στο σωρό. Αυτό δεν είναι απαραίτητα μειονέκτημα, αφού σε γενικές γραμμές είναι καλό για ένα συλλέκτη να ελευθερώνει όση περισσότερη μνήμη μπορεί. Υπάρχουν όμως αντικείμενα, τα οποία έχουν επιβιώσει από έναν μεγάλο αριθμό συλλογών, και πρόκειται να επιβιώσουν για αρκετές ακόμα. Αυτά τα αντικείμενα εξετάζονται σε κάθε κύκλο συλλογής σκουπιδιών, σημειώνονται ως ζωντανά και στη χειρότερη περίπτωση μετακινούνται συνεχώς από τη μία περιοχή του σωρού στην άλλη. Αυτό επιφέρει μια σημαντική επιβάρυνση στην εκτέλεση του συλλέκτη. Επίσης οι περισσότεροι αλγόριθμοι προκαλούν ακανόνιστες παύσεις στην εκτέλεση του προγράμματος, οι οποίες δεν είναι επιθυμητές.

7.1 Περιγραφή

Στην προσπάθειά τους να μειώσουν το χρόνο που δαπανάται στη συλλογή σκουπιδιών, υπάρχουν αλγόριθμοι οι οποίοι επιχειρούν να απομονώσουν τα μακρόβια αντικείμενα σε μία περιοχή του σωρού και να επικεντρωθούν στη συλλογή των πιο καινούριων. Έχει διαπιστωθεί μάλιστα ότι τα περισσότερα αντικείμενα που εκχωρούνται δυναμικά γίνονται σκουπίδια (πεθαίνουν) γρήγορα[Sha88].

Αυτό το πετυχαίνουν ακολουθώντας μια *γενετική* (*generational*) τεχνική[LH83]. Ο σωρός χωρίζεται σε τμήματα, τα οποία αντιπροσωπεύουν τις γενιές (generations) των αντικειμένων που περιέχουν. Ανάλογα με την παλαιότητά του, το αντικείμενο κατατάσσεται στην ανάλογη γενιά του σωρού. Κάθε επόμενη γενιά θα πρέπει να είναι εκθετικά μεγαλύτερη σε μέγεθος από την προηγούμενή της. Όλα τα καινούρια αντικείμενα τοποθετούνται στην πρώτη γενιά, ενώ όσο παλιώνουν προάγονται (μετακινούνται) σε επόμενη γενιά. Υπάρχουν αρκετοί τρόποι για να καθορίσουμε πως παλιώνει ένα αντικείμενο[WM89], όπως για παράδειγμα όταν περνάει ένα προκαθορισμένο διάστημα χρόνου, όταν συμπληρώνεται ένας προκαθορισμένος αριθμός αναθέσεων, κ.α.. Συνήθως όμως ένα αντικείμενο παλιώνει κάθε φορά που επιβιώνει από κάποιους κύκλους (συχνά έναν) συλλογής σκουπιδιών.

Επίσης είναι συνηθισμένο οι περιοχές στις οποίες χωρίζεται ο σωρός να είναι δύο. Η πρώτη

περιοχή (''βρεφοκομείο'') χρησιμοποιείται για τα καινούρια αντικείμενα. Στη δεύτερη περιοχή (''γηροκομείο'') θα μετακινούνται όσα αντικείμενα επιβιώσουν από έναν αριθμό συλλογών (συχνά μία). Η μετακίνηση αυτή πρέπει να συνοδεύεται και από την ενημέρωση των δεικτών που δείχνουν στα αντικείμενα αυτά.

Ο αλγόριθμος γενετικής συλλογής σκουπιδιών θα επιχειρεί αρχικά να συλλέξει μόνο την πρώτη περιοχή του σωρού. Η περιοχή αυτή είναι εκθετικά μικρότερη από την επόμενή της, γεγονός που οδηγεί σε έναν πολύ πιο γρήγορο κύκλο συλλογής σκουπιδιών, από αυτόν που θα χρειαζόταν για να συλλεχθεί ολόκληρος ο σωρός. Ο κύκλος αυτός επιταχύνεται ακόμη περισσότερο, καθώς τα νέα αντικείμενα έχουν πολλές πιθανότητες να καταλήξουν σκουπίδια πριν μετακινηθούν στην επόμενη γενιά. Έτσι ένας αλγόριθμος συλλογής σκουπιδιών με κόστος ανάλογο μόνο των ζωντανών αντικειμένων θα ήταν πολύ αποτελεσματικός, αφού τα ζωντανά αντικείμενα δεν αναμένεται να είναι πολλά. Συνήθως για τη συλλογή της πρώτης γενιάς χρησιμοποιείται αλγόριθμος διακοπής αντιγραφής. Κάθε συλλέκτης βέβαια μπορεί να χρησιμοποιεί διαφορετικό αλγόριθμο, η συνδυασμό αλγορίθμων για τη συλλογή των γενιών.

Ειδική μέριμνα πρέπει να ληφθεί για τυχόν αντικείμενα σε μεγαλύτερες γενιές, που δείχνουν σε μικρότερες[1]. Για να τα εντοπίσει ο συλλέκτης πρέπει να διατρέξει ολόκληρο το σωρό, γεγονός που θα επιβάρυνε σημαντικά την αποδοτικότητά του. Εναλλακτικά πρέπει ο μεταγλωττιστής να εισάγει κώδικα που να παρακολουθεί τις εγγραφές σε τέτοιους δείκτες και να τους παρέχει εν συνεχεία στο πρόγραμμα. Ευτυχώς όμως στις περισσότερες γλώσσες και στις περισσότερες προγραμματιστικές τεχνικές τέτοιοι δείκτες είναι σπάνιοι, καθώς τα αντικείμενα αρχικοποιούνται τη στιγμή της δημιουργίας τους.

Αν δεν ελευθερωθεί αρκετός χώρος, ή αν πραγματοποιηθεί ένας προκαθορισμένος αριθμός συλλογών στην πρώτη γενιά, τότε ο συλλέκτης θα συλλέξει την επόμενη γενιά. Στη γενική περίπτωση ο συλλέκτης θα πρέπει να μπορεί να εξετάζει ολόκληρο το σωρό, αν και κάτι τέτοιο θα γίνεται σπάνια. Με άλλα λόγια όσο πιο μεγάλη είναι μια γενιά τόσο πιο σπάνια θα εξετάζεται για σκουπίδια. Η προσέγγιση αυτή επιλύει το πρόβλημα της συνεχούς εξέτασης των μακρόβιων αντικειμένων που τέθηκε στην αρχή του παρόντος κεφαλαίου.

Σε αυτό το κεφάλαιο θα παρουσιαστούν δύο απλοί γενετικοί αλγόριθμοι, οι οποίοι χωρίζουν το σωρό σε δύο γενιές. Ο πρώτος χρησιμοποιεί τον αλγόριθμο διακοπής αντιγραφής για τη συλλογή της πρώτης γενιάς και τον αλγόριθμο σημείωσης συμπύκνωσης για τη συλλογή ολόκληρου του σωρού. Ο δεύτερος χρησιμοποιεί τον αλγόριθμο διακοπής αντιγραφής και για τη συλλογή της πρώτης γενιάς και για τη συλλογή ολόκληρου του σωρού.

7.2 Ανάθεση μνήμης

Και για τους δύο γενετικούς αλγόριθμους που θα παρουσιαστούν στη συνέχεια ο ελεύθερος χώρος είναι πάντα συνεχόμενος. Αυτό απαιτεί απλά τη χρήση ενός δείκτη στην αρχή του ελεύθερου χώρου, χωρίς την ανάγκη για βοηθητικές δομές δεδομένων, γεγονός που καθιστά την ανάθεση μνήμης πολύ γρήγορη. Όλη η μνήμη παραχωρείται από την πρώτη γενιά του

[1]Στην NFlint δεν επιτρέπονται τέτοιοι δείκτες, οπότε δε χρειάζεται να ληφθεί κανένα μέτρο. Είναι εξασφαλισμένο ότι κανένα αντικείμενο δε θα δείχνει σε νεότερό του.

σωρού. Αν ο ελεύθερος χώρος της γενιάς αυτής επαρκεί για την ανάθεση, τότε παραχωρείται ένα τμήμα μνήμης με το ζητούμενο μέγεθος στο πρόγραμμα και ο δείκτης αυξάνεται έτσι ώστε να δείχνει πάλι στην αρχή του εναπομένοντα ελεύθερου χώρου.

7.3 Ο Γενετικός αλγόριθμος 1

Ο αλγόριθμος αυτός χωρίζει τον σωρό σε δύο γενιές. Η δεύτερη γενιά έχει τετραπλάσιο μέγεθος από την πρώτη. Στην πρώτη γενιά αποθηκεύονται τα νέα αντικείμενα. Όταν γεμίσει αυτή η γενιά, τότε με χρήση του αλγορίθμου διακοπής αντιγραφής αντιγράφονται όλες οι ζωντανές tuples (όσες δηλαδή επιβίωσαν από τη συλλογή αυτή) στη δεύτερη γενιά. Για να συμβεί αυτό πρέπει ο ελεύθερος χώρος στην δεύτερη γενιά να είναι μεγαλύτερος ή ίσος με το μέγεθος της πρώτης γενιάς, έτσι ώστε να εξασφαλιστεί ότι ακόμη και στη χειρότερη περίπτωση, οι tuples που θα μετακινηθούν θα χωρέσουν στον προορισμό τους. Σε διαφορετική περίπτωση συλλέγεται ολόκληρος ο σωρός με χρήση του αλγόριθμου σημείωσης συμπύκνωσης[San91]. Η χρήση του αλγορίθμου σημείωσης συμπύκνωσης απαιτεί δύο πράγματα: Πρώτον ότι δεν θα υπάρχει κενό μεταξύ των tuples σε κανένα σημείο του σωρού. Αυτή η απαίτηση όμως δεν μπορεί να ικανοποιηθεί, αφού μεταξύ της γενιάς 1 και της γενιάς 2 είναι πολύ πιθανό να υπάρχει ελεύθερος χώρος. Για να λυθεί το πρόβλημα αυτό, κατασκευάζεται μια εικονική tuple στον ελεύθερο αυτό χώρο, με μέγεθος ίσο με αυτό του χώρου. Η tuple αυτή δεν επηρεάζει την εκτέλεση του προγράμματος, αφού είναι απρόσιτη από το πρόγραμμα και απλά ο αλγόριθμος σημείωσης συμπύκνωσης θα την θεωρήσει ως σκουπίδι. Ο αλγόριθμος διακοπής αντιγραφής θα την αγνοήσει εντελώς, θεωρώντας το χώρο που καταλαμβάνει ελεύθερο. Δεύτερον σημαίνει ότι θα πρέπει να δημιουργηθεί ένας πίνακας μεταθέσεων με μέγεθος 1/3 του σωρού. Έτσι πριν χωριστεί σε γενιές ο σωρός, δεσμεύεται πρώτα το 1/3 του μεγέθους του για τον πίνακα αυτόν. Αν η διαθέσιμη μνήμη εξαντλείται, υπάρχει η πιθανότητα η γενιά 2 να ξεχειλίσει από τα ζωντανά αντικείμενα και μερικά να καταλάβουν χώρο και στη γενιά 1. Σε τέτοια περίπτωση ο γενετικός αλγόριθμος εκφυλίζεται σε αλγόριθμο σημείωσης συμπύκνωσης (μέχρι τουλάχιστον να ελευθερωθεί αρκετός χώρος στη γενιά 2, οπότε και επανέρχεται στη γενετική μέθοδο συλλογής). Ο ελεύθερος χώρος, όσο μικρός και αν είναι παραμένει κάτω από τις tuples.

Algorithm 5 Generational Garbage Collection Algorithm 1

if free space in generation2 $\geq$ size of generation1 **then**
  stop & copy generation1 $\rightarrow$ generation2
else mark & compact the whole heap

7.3.1 Κόστος εκτέλεσης

Οι συλλογές σκουπιδιών που πραγματοποιεί ο συγκεκριμένος γενετικός αλγόριθμος διακρίνονται σε μικρές (minor) και μεγάλες (major) ανάλογα με το ποιο κομμάτι του σωρού συλλέγεται κάθε φορά. Στις μικρές συλλογές συλλέγεται μόνο η πρώτη γενιά του σωρού με τον αλγόριθμο διακοπής αντιγραφής. Το κόστος της συλλογής αυτής είναι ανάλογο μόνο του

αριθμού των ζωντανών tuples που βρίσκονται σε εκείνη τη γενιά. Δεδομένου ότι σε αυτήν υπάρχουν μόνο νέες tuples που δεν έχουν επιβιώσει από κανέναν κύκλο συλλογής σκουπιδιών, και δεχόμενοι ότι τα περισσότερα αντικείμενα που εκχωρούνται δυναμικά ζουν λίγο, τότε ο αριθμός των ζωντανών tuples στη γενιά αυτή δεν αναμένεται να είναι μεγάλος. Συνεπώς το κόστος μιας μικρής συλλογής είναι συνήθως πολύ μικρό.

Στις μεγάλες συλλογές συλλέγεται ολόκληρος ο σωρός. Το κόστος μιας τέτοιας συλλογής είναι πού μεγαλύτερο από εκείνο μιας μικρής. Η χρήση του αλγορίθμου σημείωσης συμπύκνωσης απαιτεί να γίνουν τρία περάσματα στο σωρό. Τα πρώτο και το τρίτο πέρασμα έχουν κόστος ανάλογο του αριθμού των ζωντανών αντικειμένων. Το δεύτερο πέρασμα είναι ανάλογο του μεγέθους του σωρού. Πάντως στη γενική περίπτωση το κόστος εκτέλεσης μιας μεγάλης συλλογής είναι μικρότερο από εκείνο του αλγορίθμου σημείωσης συμπύκνωσης καθώς οι tuples της γενιάς 2 έχουν επιβιώσει τουλάχιστον από μία συλλογή σκουπιδιών, και είναι πιθανό να επιβιώσουν από αρκετές ακόμα. Έτσι αν και σαρώνεται όλος ο σωρός, τα αντικείμενα που χρειάζονται μετακίνηση είναι συνήθως λιγότερα, καθώς αυτά της γενιάς 2 είναι λιγότερο πιθανό να χρειαστεί να μετακινηθούν.

Ευτυχώς, οι μεγάλες συλλογές δεν είναι συχνές. Στη χειρότερη περίπτωση γίνεται μία μεγάλη συλλογή για κάθε τέσσερις μικρές. Στην πράξη όμως οι μεγάλες συλλογές γίνονται πολύ σπανιότερα. Αυτό μειώνει το κόστος εκτέλεσης του γενετικού αλγορίθμου, όπως επίσης και την διάρκεια των παύσεων που προκαλούνται στο πρόγραμμα.

Όσον αφορά τις απαιτήσεις σε χώρο, ο γενετικός αλγόριθμος 1 έχει τις ίδιες απαιτήσεις με τον αλγόριθμο σημείωσης συμπύκνωσης. Το 1/3 του σωρού δεσμεύεται για τη δημιουργία του πίνακα μεταθέσεων. Τα υπόλοιπα 2/3 χρησιμοποιούνται από τον αλγόριθμο ως ωφέλιμα.

7.3.2 Υλοποίηση

Στην υλοποίηση του αλγορίθμου αυτού θεωρούνται δεδομένες όλες οι απαιτήσεις των δύο επί μέρους αλγορίθμων που χρησιμοποιούνται, του αλγόριθμου διακοπής αντιγραφής και του αλγόριθμου σημείωσης συμπύκνωσης (π.χ. η ανάγκη για bit σήμανσης). Δεν απαιτείται βέβαια να χωριστεί ο σωρός στα δύο για να λειτουργήσει ο αλγόριθμος διακοπής αντιγραφής, αφού ο χώρος προορισμού (to–space) είναι ο ελεύθερος χώρος της γενιάς 2. Παρακάτω παρουσιάζεται η υλοποίηση του αλγορίθμου.

Η συνάρτηση `int32_t *gc_algo_alloc(int32_t size)` πραγματοποιεί την ανάθεση μνήμης. Χρησιμοποιεί έναν βοηθητικό δείκτη στην αρχή του ελεύθερου χώρου της γενιάς 1. Αν ο χώρος αυτός, ο οποίος είναι συνεχόμενος, επαρκεί για την ανάθεση μιας tuple με μέγεθος `size + 1` τότε κατοχυρώνει ένα κομμάτι με το μέγεθος αυτό στην αρχή του χώρου, φτιάχνει την κατάλληλη επικεφαλίδα με το μέγεθός του, και επιστρέφει έναν δείκτη στην δεύτερη λέξη του τμήματος που μόλις κατοχυρώθηκε. Ο βοηθητικός δείκτης αυξάνεται και δείχνει στην αρχή του ελεύθερου χώρου που απομένει μετά την ανάθεση. Σε περίπτωση που ο διαθέσιμος χώρος δεν επαρκεί για την ανάθεση, η συνάρτηση επιστρέφει `NULL`.

Η συνάρτηση `void gc_algo_collect(void)` ενεργοποιεί την συλλογή σκουπιδιών. Αρχικά ο συλλέκτης εξετάζει αν υπάρχει αρκετός χώρος στην γενιά 2 για να χωρέσουν οι ζωντανές

tuples που βρίσκονται στη γενιά 1. Στη χειρότερη περίπτωση (αν όλες είναι ζωντανές) απαιτείται ελεύθερος χώρος μεγαλύτερος ή ίσος του συνολικού μεγέθους της γενιάς 1. Αν επαρκεί ο χώρος, τότε εφαρμόζεται κατά τα γνωστά ο αλγόριθμος διακοπής αντιγραφής, ο οποίος συλλέγει μόνο την γενιά 1. Ασχολείται μόνο με δείκτες που δείχνουν στη γενιά 1, αγνοώντας όλους τους υπόλοιπους. Αναδρομικά αντιγράφει όλες τις tuples της γενιάς 1 στον ελεύθερο χώρο της γενιάς 2. Αν ο ελεύθερος χώρος της γενιάς 2 δεν επαρκεί, τότε εφαρμόζεται κατά τα γνωστά ο αλγόριθμος σημείωσης συμπύκνωσης σε όλο το σωρό. Αν μετά την εφαρμογή του υπάρχει ελεύθερος χώρος μεταξύ της γενιάς 2 και της γενιάς 1, τότε αυτός καλύπτεται από μία εικονική (dummy) tuple.

Η συνάρτηση αρχικοποίησης του συλλέκτη `void gc_algo_init(void)` κάνει τις απαραίτητες αρχικοποιήσεις για το ολικό μέγεθος και την αρχική διεύθυνση του σωρού. Στη συνέχεια δεσμεύει το 1/3 του μεγέθους του σωρού για τη δημιουργία του πίνακα μεταθέσεων και αρχικοποιεί έναν δείκτη στην αρχή του. Από τα 2/3 του σωρού που απομένουν, δίνει το 1/5 στην γενιά 1 και τα 4/5 στη γενιά 2. Τέλος αρχικοποιεί έναν δείκτη στην αρχή κάθε γενιάς.

Τέλος, η συνάρτηση ολοκλήρωσης του αλγορίθμου `void gc_algo_final(void)` δεν χρειάστηκε να κάνει κάποια δουλειά και το σώμα της παρέμεινε κενό.

7.4 Ο Γενετικός αλγόριθμος 2

Ο αλγόριθμος αυτός είναι παρόμοιος με τον γενετικό αλγόριθμο 1. Και εδώ για την συλλογή της πρώτης γενιάς χρησιμοποιείται ο αλγόριθμος διακοπής αντιγραφής. Η μόνη τους διαφορά είναι ότι στον γενετικό αλγόριθμο 2 χρησιμοποιείται ο ίδιος αλγόριθμος και στην περίπτωση που χρειαστεί να συλλεχθούν τα σκουπίδια ολόκληρου του σωρού. Σαν αποτέλεσμα δε χρειάζεται ο πίνακας μεταθέσεων που χρησιμοποιείται στον προηγούμενο αλγόριθμο. Χρειάζεται όμως να δεσμευθεί ο μισός σωρός για να μπορέσει να λειτουργήσει ο αλγόριθμος διακοπής αντιγραφής. Συνεπώς ο σωρός, πριν χωριστεί σε γενιές, χωρίζεται στα δύο. Το πρώτο μισό είναι η ενεργή περιοχή του, και το δεύτερο η ανενεργή. Στη συνέχεια κάθε περιοχή χωρίζεται σε δύο γενιές. Και εδώ η δεύτερη έχει το τετραπλάσιο μέγεθος από την πρώτη. Η πρώτη γενιά χρησιμοποιείται για τις αναθέσεις μνήμης, ενώ στη δεύτερη προάγονται οι παλαιότερες tuples. Επίσης ο αλγόριθμος αυτός δεν επηρεάζεται από την ύπαρξη ελεύθερου χώρου μεταξύ της πρώτης και της δεύτερης γενιάς (αρκεί οι 2 γενιές να είναι συνεχόμενες).

Κατά τα άλλα η βασική ιδέα παραμένει η ίδια. Όταν γεμίσει η γενιά 1, τότε ο συλλέκτης επιχειρεί να συλλέξει μόνο αυτήν. Αν ο ελεύθερος χώρος της γενιάς 2 επαρκεί, τότε η συλλογή είναι επιτυχής, αλλιώς προχωράει σε συλλογή ολόκληρου του σωρού. Θεωρεί την γενιά 1 και τη γενιά 2 ως μια ενιαία ενεργή περιοχή του σωρού, της οποίας τις ζωντανές tuples αντιγράφει στην ανενεργή περιοχή του σωρού. Όταν ολοκληρωθεί η αντιγραφή και ενημερωθούν όλοι οι δείκτες (βλ. προηγούμενο κεφάλαιο), εναλλάσσονται η ενεργή περιοχή με την ανενεργή περιοχή και συνεχίζεται η εκτέλεση του προγράμματος. Σε περίπτωση που η γενιά 2 δεν επαρκεί για την αποθήκευση των ζωντανών tuples, ο γενετικός αλγόριθμος 2 εκφυλίζεται σε αλγόριθμο διακοπής αντιγραφής (μέχρι τουλάχιστον να ελευθερωθεί αρκετός χώρος στην γενιά 2, οπότε

και συνεχίζεται η γενετική τεχνική συλλογής).

Algorithm 6 Generational Garbage Collection Algorithm 2

if free space in generation2 $\geq$ size of generation1 **then**
  stop & copy generation1 $\rightarrow$ generation2
else stop & copy the whole heap

7.4.1 Κόστος εκτέλεσης

Και αυτός ο αλγόριθμος πραγματοποιεί μεγάλες και μικρές συλλογές. Το κόστος της μικρής συλλογής είναι ανάλογο μόνο του αριθμού των ζωντανών tuples στην πρώτη γενιά, ο οποίος, όπως εξηγήθηκε προηγουμένως, δεν αναμένεται να είναι μεγάλος.

Στη μεγάλη συλλογή χρησιμοποιείται επίσης ο αλγόριθμος διακοπής αντιγραφής, μόνο που τώρα συλλέγει ολόκληρο το σωρό. Το κόστος της συλλογής αυτής είναι πάλι ανάλογο μόνο του ζωντανού αριθμού των αντικειμένων στο σωρό, παραμένει όμως πολύ μεγαλύτερο από το κόστος μιας μικρής συλλογής.

Όπως και σε κάθε γενετικό αλγόριθμο, έτσι και σε αυτόν μια μεγάλη συλλογή πραγματοποιείται λιγότερο συχνά από μία μικρή. Αν και στη χειρότερη περίπτωση γίνεται μία μεγάλη συλλογή για κάθε τέσσερις μικρές, στην πράξη οι μεγάλες συλλογές γίνονται πολύ σπανιότερα. Αυτό μειώνει το κόστος εκτέλεσης του γενετικού αλγορίθμου, όπως επίσης και την διάρκεια των παύσεων που προκαλούνται στο πρόγραμμα.

Από τη μεριά του κόστους σε χώρο, ο γενετικός αλγόριθμος 2 απαιτεί την κατάτμηση του σωρού σε δύο ισομεγέθη τμήματα. Το πρώτο τμήμα θα χρησιμοποιηθεί για την ενεργή περιοχή και το δεύτερο για την ανενεργή. Έτσι μόνο 1/2 του σωρού μπορεί ανά πάσα στιγμή να χρησιμοποιηθεί από τον αλγόριθμο ως ωφέλιμο.

7.4.2 Υλοποίηση

Ο αλγόριθμος αυτός μοιράζεται τις ίδιες απαιτήσεις με τον αλγόριθμο διακοπής αντιγραφής (π.χ. χρήση bit σήμανσης). Η υλοποίησή του μοιάζει αρκετά με την υλοποίηση του αλγορίθμου γενετικής συλλογής 1. Ειδικότερα:

Η συνάρτηση `int32_t *gc_algo_alloc(int32_t size)` παραμένει ίδια με την αντίστοιχη συνάρτηση του γενετικού αλγόριθμου 1.

Η συνάρτηση `void gc_algo_collect(void)` είναι υπεύθυνη για τη συλλογή των σκουπιδιών. Αρχικά επιχειρεί μία συλλογή μόνο της γενιάς 1 (της ενεργής περιοχής) όπως ακριβώς στον γενετικό αλγόριθμο 1. Αν αυτό δε σταθεί δυνατό, λόγω ανεπάρκειας του ελεύθερου χώρου της γενιάς 1, τότε προχωρεί σε συλλογή ολόκληρου του σωρού με χρήση πάλι του αλγόριθμου διακοπής αντιγραφής. Οι ζωντανές tuples και των δύο γενιών της ενεργής περιοχής αντιγράφονται στην ανενεργή περιοχή του σωρού. Πρακτικά μπορεί να χρησιμοποιηθεί η ίδια συνάρτηση και για τις δύο περιπτώσεις συλλογών, αφού το μόνο που αλλάζει είναι οι δείκτες πηγής (source) και προορισμού (destination). Έτσι έχει προστεθεί ένα νέο όρισμα στη συνάρτηση, ένας ακέραιος αριθμός, ανάλογα με την τιμή του οποίου καθορίζεται και ποια

συλλογή θα πραγματοποιήσει ο αλγόριθμος. Η επικεφαλίδα της συνάρτησης είναι σε αυτήν την περίπτωση `int32_t *copyTuple(int32_t *l, int32_t vl)`. Αν η παράμετρος `vl` έχει την τιμή 1, τότε πραγματοποιείται συλλογή μόνο της γενιάς 1. Αν έχει την τιμή 2 πραγματοποιείται συλλογή ολόκληρου του σωρού. Η ανενεργή περιοχή έχει και αυτή δυο γενιές με μεγέθη που ταυτίζονται με τα μεγέθη των δύο γενιών της ενεργής περιοχής. Μετά από κάθε τέτοια συλλογή, ο γενετικός αλγόριθμος 2 φροντίζει ώστε η ανενεργή περιοχή του σωρού να γίνει ενεργή και η ενεργή περιοχή ανενεργή.

Η συνάρτηση `void gc_algo_init(void)` ενημερώνεται για το ολικό μέγεθος και την αρχική διεύθυνση του σωρού κάνοντας χρήση των συναρτήσεων που παρέχει η διεπαφή συλλογής σκουπιδιών. Στη συνέχεια ο σωρός διαιρείται σε 2 ισομεγέθη τμήματα. Το πρώτο τμήμα καθιερώνεται ως η ενεργή περιοχή του σωρού και το δεύτερο ως η ανενεργή. Κάθε περιοχή στη συνέχεια διαιρείται σε δύο γενιές. Η δεύτερη έχει τέσσερις φορές το μέγεθος της πρώτης. Χρησιμοποιούνται τέσσερις δείκτες. Ένας δείχνει στην αρχή της ενεργής περιοχής, ένας στην ανενεργή, ένας στην πρώτη γενιά της ενεργής περιοχής και ένας στη δεύτερη. Δεν απαιτούνται δείκτες στις γενιές της ανενεργής περιοχής του σωρού. Γνωρίζοντας την αρχική της διεύθυνση και το μέγεθος κάθε γενιάς μπορούμε να χρησιμοποιήσουμε τους αντίστοιχους δείκτες της ενεργής περιοχής.

Τέλος, η συνάρτηση `void gc_algo_final(void)` είναι επίσης η ίδια με αυτήν του γενετικού αλγόριθμου 1.

7.5 Πλεονεκτήματα – Μειονεκτήματα

Τα πλεονεκτήματα των δύο αυτών αλγορίθμων είναι πολλά. Πρώτον η επιβάρυνση που προκαλούν στο πρόγραμμα κατά το στάδιο της συλλογής σκουπιδιών είναι μικρή, αφού στους περισσότερους κύκλους συλλογής σκουπιδιών πραγματοποιούν μόνο μικρές συλλογές. Δεν ασχολούνται συχνά με τα μακρόβια αντικείμενα, παρά μόνο όταν είναι απαραίτητο. Ειδικά στον γενετικό αλγόριθμο 1 τα αντικείμενα αυτά μετακινούνται σπάνια, σε αντίθεση με τον γενετικό αλγόριθμο 2 ο οποίος μετακινεί τα μακρόβια αντικείμενα σε κάθε μεγάλη συλλογή. Επιπροσθέτως ο ελεύθερος χώρος στο σωρό είναι συνεχόμενος, αφού χρησιμοποιείται η πολιτική μετακίνησης των αντικειμένων, αποτρέποντας έτσι τον κατακερματισμό του σωρού. Αυτό οδηγεί σε ταχύτατη ανάθεση μνήμης. Μπορούν επίσης να διαχειριστούν κυκλικές δομές δεδομένων, αρκεί όμως οι δείκτες παλαιοτέρων αντικειμένων σε νεότερα να παρέχονται από των συλλέκτη μαζί με τις ρίζες. Αυτό απαιτεί την παραγωγή επιπλέον κώδικα από τον μεταγλωττιστή.

Στον αντίποδα, υπάρχει αυξημένο κόστος σε χώρο για τις ανάγκες τους (1/3 του σωρού θυσιάζει ο πρώτος και 1/2 του σωρού ο δεύτερος). Επίσης οι αλγόριθμοι αυτοί δεν αποδίδουν καλά σε περιπτώσεις που η πληρότητα του σωρού είναι πολύ υψηλή. Σε αρκετές τέτοιες περιπτώσεις παρουσιάζουν απόδοση που είναι χειρότερη των συμβατικών αλγορίθμων.

Κεφάλαιο 8

Συγκριτική Μελέτη

Η επιβάρυνση που προκαλεί η συλλογή σκουπιδιών στο πρόγραμμα είναι δύσκολο να υπολογιστεί. Εξαρτάται από πολλούς παράγοντες, οι οποίοι μάλιστα μπορεί να διαφέρουν από πρόγραμμα σε πρόγραμμα. Η θεωρητική προσέγγιση του κόστους τους επικεντρώνεται μόνο στα χαρακτηριστικά του καθενός, ενώ είναι δυνατόν να προσδιοριστεί η αποδοτικότητα ενός αλγορίθμου στην βέλτιστη και στην χείριστη περίπτωση. Για τη μέση περίπτωση το μόνο που μπορούμε να πούμε με σιγουριά για το κόστος ενός αλγορίθμου συλλογής σκουπιδιών, είναι ότι βρίσκεται κάπου ανάμεσα. Δυστυχώς όμως αυτό δεν είναι αρκετό, αφού αλγόριθμοι με παρόμοια χαρακτηριστικά, μπορεί να έχουν διαφορετικές επιδόσεις, ανάλογα κάθε φορά με το περιβάλλον στο οποίο υλοποιούνται.

8.1 Αναζήτηση καταλληλότερου αλγορίθμου συλλογής σκουπιδιών στο περιβάλλον NFlint

Οι περιορισμοί που θέτει η υλοποίηση του NFlint για τη συλλογή σκουπιδιών, είχαν σαν αποτέλεσμα την απόρριψη κάποιων αλγορίθμων (π.χ. μετρητές αναφοράς). Τελικά αποφασίστηκε να υλοποιηθούν οι έξι αλγόριθμοι που παρουσιάστηκαν στα προηγούμενα κεφάλαια. Η θεωρητική προσέγγιση στο κόστος των αλγορίθμων αυτών άφηνε πολλά αναπάντητα ερωτήματα. Το κυριότερα από αυτά αφορούσαν το πώς για ένα δεδομένο μέγεθος σωρού, επηρεάζεται η απόδοση κάθε αλγορίθμου, όπως για παράδειγμα:

- Αξίζει στον αλγόριθμο διακοπής αντιγραφής να θυσιαστεί ο μισός σωρός από αυτόν που θα είχε στη διάθεσή του; Μήπως ο αλγόριθμος σημείωσης σάρωσης θα τα κατάφερνε καλύτερα για το ίδιο μέγεθος σωρού;

- Μεταξύ του αλγόριθμου σημείωσης σάρωσης και σημείωσης χωρίς σάρωση, ποιος είναι ο ταχύτερος; Πόσο επηρεάζει ο κατακερματισμός του σωρού την αποδοτικότητα των δύο αυτών αλγορίθμων;

- Ο αλγόριθμος σημείωσης συμπύκνωσης απαιτεί περισσότερο χώρο από τους αλγόριθμους σημείωσης σάρωσης και σημείωσης χωρίς σάρωση και λιγότερο από τον αλγόριθμο

διακοπής αντιγραφής. Είναι ταχύτερος από τους πρώτους; Πόσο πιο αργός είναι από τον δεύτερο; Αξίζει και εδώ το παραπάνω κόστος σε χώρο του αλγορίθμου διακοπής αντιγραφής;

– Πόσο αποδοτικοί είναι οι γενετικοί συλλέκτες σε σχέση με τους απλούς μετακινούντες αλγόριθμους, με τους οποίους έχουν τις ίδιες απαιτήσεις σε χώρο; Αξίζει και εδώ ο παραπάνω χώρος που απαιτεί ο γενετικός αλγόριθμος 2; Πόσο επηρεάζουν τον γενετικό αλγόριθμο 1 τα τρία περάσματα του σωρού που πραγματοποιεί σε κάθε μεγάλη συλλογή του;

Οι ερωτήσεις συνεχίζονται, καθιστώντας δύσκολη την επιλογή του κατάλληλου αλγορίθμου για το NFlint. Για μία πιο ολοκληρωμένη εικόνα της συμπεριφοράς του κάθε αλγορίθμου σε αυτό το περιβάλλον, κάθε αλγόριθμος θα δοκιμαστεί στη συνέχεια του κεφαλαίου σε συνθήκες πραγματικού χρόνου. Σε αυτό το σημείο τονίζεται ότι το κόστος σε χρόνο εκτέλεσης έχει θεωρηθεί σημαντικότερο από το κόστος σε χώρο. Αυτό έχει σαν αποτέλεσμα μεταξύ δύο αλγορίθμων να προτιμηθεί ο ταχύτερος, ασχέτως από τις απαιτήσεις που έχει για επιπλέον μνήμη. Προφανώς μεταξύ δυο αλγορίθμων με το ίδιο κόστος εκτέλεσης, θα προτιμηθεί αυτός με τις μικρότερες απαιτήσεις σε χώρο.

8.1.1 Τα benchmarks

Για τη μέτρηση της επίδοσης των αλγορίθμων χρησιμοποιήθηκαν δύο συγκριτικά προγράμματα (benchmarks), τα οποία έτρεξαν για διαφορετικά μεγέθη σωρού με σκοπό να παρατηρηθεί πως μεταβάλλεται η αποδοτικότητα του αλγορίθμου συλλογής σκουπιδιών για διαφορετικές εισόδους του προγράμματος και για διαφορετικά μεγέθη του σωρού.

Το πρώτο πρόγραμμα που χρησιμοποιήθηκε για τη σύγκριση των αλγορίθμων ήταν το πρόγραμμα tak. Πρόκειται για την υλοποίηση της συνάρτησης `Tak()`, την οποία ονόμασε έτσι ο John McCarthy προς τιμήν του δημιουργού της Ikuo Takeuchi. Η συνάρτηση αυτή είναι αναδρομική και κάνει χιλιάδες αναδρομικές κλήσεις. Η συνολική μνήμη που απαιτεί το πρόγραμμα για την εκτέλεσή του (για τις συγκεκριμένες εισόδους) είναι της τάξεως των 2 GB, αλλά η μνήμη που καταλαμβάνουν τα ζωντανά αντικείμενα στο σωρό, ανά πάσα στιγμή, είναι λιγότερη από 1,5 KB.

Το δεύτερο πρόγραμμα που χρησιμοποιήθηκε ήταν το nofib, το οποίο υπολογίζει αναδρομικά τους N πρώτους όρους της ακολουθίας Fibonacci. Η πολυπλοκότητα του προγράμματος αυτού είναι $O(2^n)$. Η συνολική μνήμη που απαιτεί το πρόγραμμα για την εκτέλεσή του (για τις συγκεκριμένες εισόδους) είναι της τάξεως του 1 GB, αλλά η μνήμη που καταλαμβάνουν τα ζωντανά αντικείμενα στο σωρό, ανά πάσα στιγμή, είναι περίπου 2 KB.

Και για τα δύο προγράμματα η συνολική απαιτούμενη μνήμη είναι αυτή για την οποία δεν χρειάζεται συλλέκτης σκουπιδιών (δεν γίνεται καμία συλλογή). Αντίθετα, με τη χρήση ενός συλλέκτη σκουπιδιών η ελάχιστη μνήμη που απαιτείται για την εκτέλεση του προγράμματος είναι τόση όσο και το μέγιστο μέγεθος των ζωντανών αντικειμένων στο σωρό ανά πάσα στιγμή. Αυτές οι δύο τιμές αποτελούν τα σενάρια καλύτερης και χειρότερης περίπτωσης αντίστοιχα για έναν συλλέκτη σκουπιδιών.

Κάθε αλγόριθμος ενσωματώθηκε στα συγκριτικά προγράμματα αυτά. Σε κάθε εκτέλεση όλοι οι αλγόριθμοι είχαν το ίδιο μέγεθος σωρού στη διάθεσή τους.

8.2 Αποτελέσματα μετρήσεων

Το σύστημα στο οποίο έγιναν τα συγκριτικά τεστ για όλους τους αλγορίθμους είχε 16 GB μνήμης, τον επεξεργαστή Intel Xeon E7340 στα 2.40GHz, ενώ το λειτουργικό σύστημα ήταν Linux 2.6.32-5 διανομή Debian 6.0.2 (squeeze). Τα αποτελέσματα μπορούν να θεωρηθούν σε μεγάλο βαθμό αξιόπιστα, αφού παρουσίασαν συνέπεια στις επαναλήψεις της εκτέλεσης των προγραμμάτων.

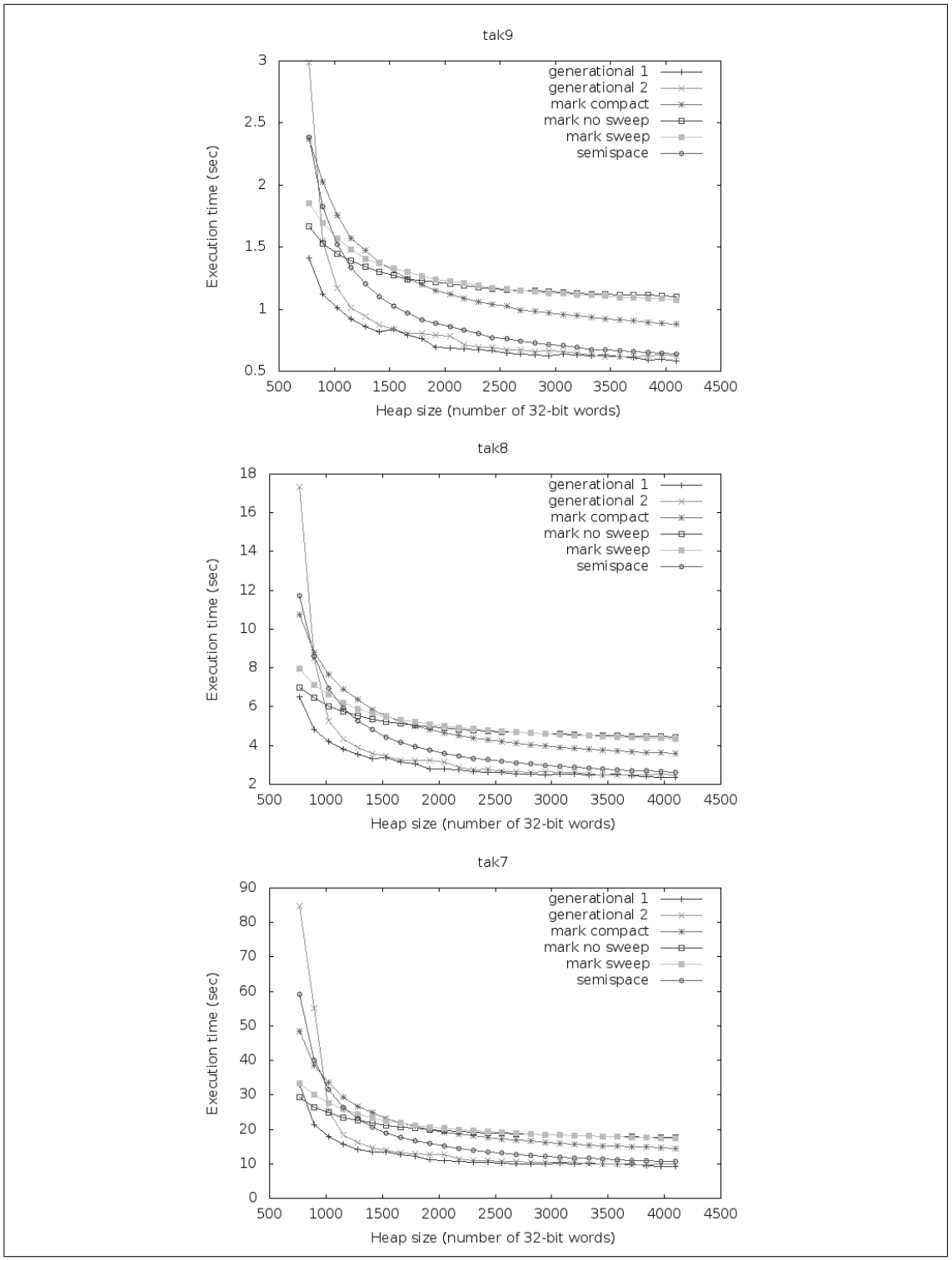
Παρακάτω παρουσιάζονται τα αποτελέσματα των μετρήσεων. Η αναπαράσταση των αποτελεσμάτων έχει γίνει με τη βοήθεια διαγραμμάτων. Κάθε διάγραμμα αντιστοιχεί σε μία σειρά μετρήσεων και περιέχει όλες τις γραφικές παραστάσεις των αλγορίθμων. Για κάθε πρόγραμμα διεξήχθησαν τρεις σειρές μετρήσεων, για διαφορετική είσοδο του προγράμματος η κάθε μία. Η είσοδος που δίνεται είναι τέτοια, ώστε σε κάθε σειρά μετρήσεων το πρόγραμμα να πραγματοποιεί περισσότερους υπολογισμούς από την προηγούμενη. Έτσι θα μπορέσουμε να παρατηρήσουμε και πως μεταβάλλεται η συμπεριφορά των αλγορίθμων συλλογής σκουπιδιών για διαφορετικές εισόδους του προγράμματος. Σε κάθε μέτρηση μετρήθηκε ο συνολικός χρόνος που έκανε το πρόγραμμα για να ολοκληρώσει τη λειτουργία του και ο αριθμός των συλλογών που πραγματοποιήθηκαν από το συλλέκτη σκουπιδιών, καθώς αυξανόταν το μέγεθος του σωρού.

Το εύρος των μεγεθών του σωρού που χρησιμοποιήθηκε είναι τέτοιο ώστε να μην προκαλέσει ακραία σενάρια, όπως π.χ. χειρότερης – καλύτερης περίπτωσης (worst – best case) στους αλγορίθμους.

8.2.1 Σύγκριση 1: Χρόνος εκτέλεσης για το πρόγραμμα tak

Στο σχήμα 8.1 φαίνονται τα αποτελέσματα και των τριών σειρών μετρήσεων για το συγκριτικό πρόγραμμα tak. Για μικρότερο αριθμό ως είσοδο, έχουμε μεγαλύτερο υπολογιστικό κόστος. (Ο μικρότερος χρόνος θεωρείται καλύτερος.)

Παρατηρούμε εύκολα, ότι όσο αυξάνεται το μέγεθος του σωρού, ελαττώνεται ο συνολικός χρόνος της εκτέλεσης του προγράμματος. Για μέγεθος σωρού κάτω από 1500 λέξεις, είμαστε κοντά σε σενάριο χειρότερης περίπτωσης για τους περισσότερους αλγορίθμους, αφού με εξαίρεση τον γενετικό αλγόριθμο 1, οι υπόλοιποι κατατάσσονται ανάλογα με το ποσοστό του σωρού που μπορούν να διαχειριστούν. Οι μη–μετακινούντες αλγόριθμοι τα καταφέρνουν πολύ καλά σε αυτό το διάστημα. Οι γενετικός αλγόριθμος 2 υστερεί σημαντικά, επειδή λόγω του περιορισμένου μεγέθους του σωρού, πραγματοποιεί πολύ συχνά μεγάλες συλλογές. Όσο αυξάνεται το μέγεθος του σωρού, παρατηρείται ραγδαία μείωση στο χρόνο του, πλησιάζοντας σε επιδόσεις τον γενετικό αλγόριθμο 1. Ο λόγος αυτής της συμπεριφοράς είναι ότι με την αύξηση αυτή η πρώτη γενιά γίνεται αρκετά μεγάλη σε μέγεθος, με αποτέλεσμα να χρειάζεται περισσότερος χρόνος για να εξαντληθεί ο χώρος της. Ο χρόνος αυτός δίνει την ευκαιρία σε



Σχήμα 8.1: Το πρόγραμμα tak – Μετρήσεις χρόνου εκτέλεσης σε συνάρτηση με το μέγεθος του σωρού.

πολλά ζωντανά αντικείμενα να γίνουν σκουπίδια και έτσι ο αλγόριθμος από ένα σημείο και μετά πραγματοποιεί κυρίως μικρές συλλογές. Με αυτόν τον τρόπο βλέπουμε πόσο σημαντικό είναι να συλλέγεται μόνο η πρώτη γενιά του σωρού και πως αυτό συμβάλει στην κατακόρυφη αύξηση της επίδοσης των γενετικών αλγορίθμων. Ο αλγόριθμος διακοπής αντιγραφής στο συγκεκριμένο παράδειγμα έχει συμπεριφορά παρόμοια με αυτή των γενετικών. Αυτό συμβαίνει επειδή στο πρόγραμμα αυτό δεν υπάρχουν μακρόβια αντικείμενα, με αποτέλεσμα να μην επιβαρύνεται για τη μετακίνησή τους. Οι συλλογές που πραγματοποιεί σε μία τέτοια περίπτωση, έχουν παρόμοιο κόστος με αυτές που κάνουν οι γενετικοί[1]. Ο αλγόριθμος σημείωσης συμπύκνωσης, αν και στην αρχή του διαγράμματος βρίσκεται πάνω από τους δύο μη–μετακινούντες αλγόριθμους, στη συνέχεια καταφέρνει να τους ξεπεράσει, παρά τα τρία περάσματα που πραγματοποιεί στο σωρό σε κάθε συλλογή του. Αποδεικνύεται πως ο κατακερματισμός του σωρού επηρεάζει τους δύο μη–μετακινούντες αλγόριθμους σε τέτοιο βαθμό, ώστε αν και μπορούν να διαχειριστούν ολόκληρο το σωρό που έχουν στη διάθεσή τους, απαιτούν τελικά τον περισσότερο χρόνο από όλους. Μεταξύ όλων των αλγορίθμων, ο αλγόριθμος σημείωσης χωρίς σάρωση, παρουσιάζει την μεγαλύτερη σταθερότητα, αφού δεν φαίνεται να επηρεάζεται σημαντικά ούτε από το μέγεθος ούτε από την πληρότητα του σωρού. Επίσης παρουσιάζει οριακά καλύτερη συμπεριφορά από τον αλγόριθμο σημείωσης σάρωσης για μικρότερα μεγέθη σωρού, αν και τελικά οι δύο αυτοί αλγόριθμοι φαίνεται να συγκλίνουν, με εκείνον της σημείωσης σάρωσης είναι οριακά καλύτερος όσο μεγαλώνει το μέγεθος του σωρού. Η αύξηση του υπολογιστικού κόστους του προγράμματος δεν φάνηκε να επηρεάζει σημαντικά τους αλγόριθμους, αφού αυξήθηκε μεν ο συνολικός χρόνος εκτέλεσης, η μορφή όμως της καμπύλης κάθε αλγορίθμου παραμένει η ίδια. Το μόνο που φαίνεται να αλλάζει είναι η αρχική κατάταξη των αλγορίθμων για μεγέθη σωρού κάτω από 1500 λέξεις. Αυτό είναι απολύτως αναμενόμενο, αφού παράλληλα με την αύξηση του υπολογιστικού κόστους του προγράμματος, αυξάνεται και η απαίτησή του σε μνήμη. Το γεγονός αυτό οδηγεί τους περισσότερους αλγορίθμους σε καταστάσεις χειρότερης περίπτωσης.

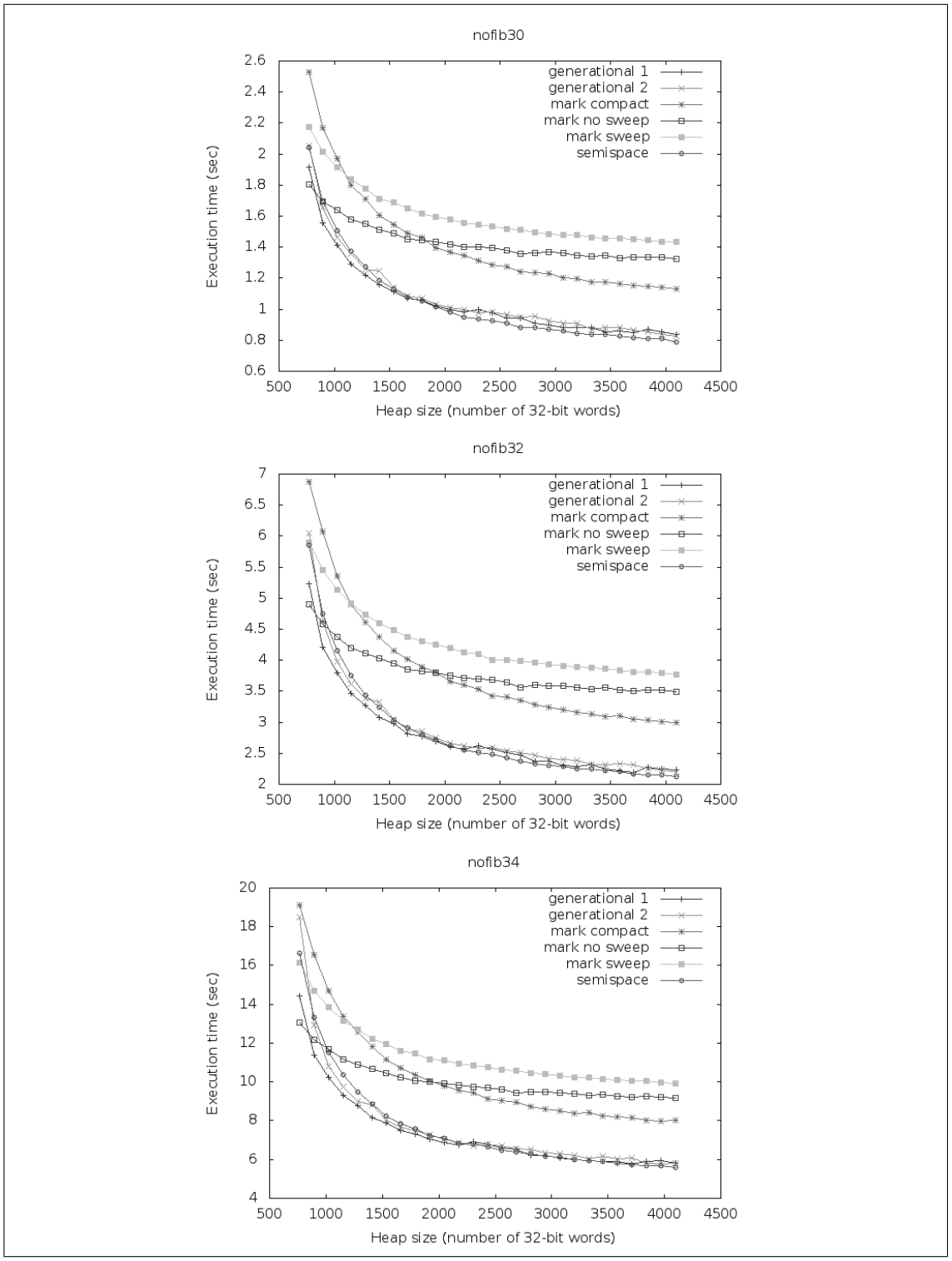
Νικητής, έστω και οριακά είναι ο γενετικός αλγόριθμος 1, αφού έχει τον καλύτερο χρόνο σε όλη τη διάρκεια των δοκιμών από όλους τους αλγορίθμους. Αν επίσης λάβουμε υπ' όψιν μας το γεγονός ότι χρειάζεται λιγότερη μνήμη από τον αλγόριθμο διακοπής αντιγραφής και από τον γενετικό αλγόριθμο 2, τότε η αποδοτικότητά του είναι στην ουσία πολύ μεγαλύτερη.

8.2.2 Σύγκριση 2: Χρόνος εκτέλεσης για το πρόγραμμα nofib

Στο σχήμα 8.2 που ακολουθεί παρουσιάζονται τα αποτελέσματα των μετρήσεων και για το δεύτερο πρόγραμμα, το nofib. Για μεγαλύτερο αριθμό εισόδου, έχουμε μεγαλύτερο υπολογιστικό κόστος. (Ο μικρότερος χρόνος θεωρείται καλύτερος.)

Εδώ παρατηρούνται λίγο διαφορετικά αποτελέσματα. Για μεγέθη σωρού κάτω από 1500 λέξεις, μεταξύ των δύο μη–μετακινούντων αλγορίθμων έχει καταφέρει να παρεμβληθεί ο γενετικός αλγόριθμος 1, γεγονός που φανερώνει ότι είμαστε μακριά από το σενάριο χειρότερης

[1]Σημειώνεται ότι και οι δύο γενετικοί αλγόριθμοι χρησιμοποιούν τον αλγόριθμο διακοπής αντιγραφής για τη συλλογή της πρώτης γενιάς



Σχήμα 8.2: Το πρόγραμμα nofib – Μετρήσεις χρόνου εκτέλεσης σε συνάρτηση με το μέγεθος του σωρού.

περίπτωσής του. Ο αλγόριθμος σημείωσης συμπύκνωσης δεν τα καταφέρνει τόσο καλά, γεγονός που σε συνδυασμό με το πόσο κοντά είναι ο αλγόριθμος διακοπής αντιγραφής σε εκείνον της σημείωσης σάρωσης φανερώνει ότι δεν έχουμε μεγάλη πληρότητα στο σωρό. Πάντως ο αλγόριθμος σημείωσης συμπύκνωσης τελικά καταφέρνει να εκμεταλλευτεί την έλλειψη του κατακερματισμού και να ξεπεράσει σε επιδόσεις τους δύο μη–μετακινούντες αλγόριθμους, οι οποίοι δείχνουν ακόμη μία φορά πόσο πολύ επηρεάζονται από το φαινόμενο του κατακερματισμού. Αξιοσημείωτη είναι για μία ακόμη φορά η σταθερότητα του αλγορίθμου σημείωσης χωρίς σάρωση, ο οποίος είναι σταθερά καλύτερος από τον αλγόριθμο σημείωσης σάρωσης. Και εδώ ο γενετικός αλγόριθμος 2 παρουσιάζει θεαματική μείωση του χρόνου εκτέλεσής του για μεγαλύτερα μεγέθη του σωρού. Ο γενετικός αλγόριθμος 1 παρουσιάζει την καλύτερη συμπεριφορά από όλους τους αλγορίθμους αν και φαίνεται να χάνει στο τέλος από τον αλγόριθμο διακοπής αντιγραφής, ο οποίος και σε αυτό το παράδειγμα επωφελείται σημαντικά από την έλλειψη μακρόβιων αντικειμένων. Ο γενετικός αλγόριθμος 2 συμπληρώνει την τριάδα, αφού αν και ξεκίνησε πέμπτος, κατέληξε πολύ γρήγορα να έχει παρεμφερείς επιδόσεις με τους άλλους δύο. Η αύξηση της υπολογιστικής πολυπλοκότητας στο συγκεκριμένο παράδειγμα δεν επηρεάζει τους αλγορίθμους. Όσο αυξάνεται το υπολογιστικό κόστος του προγράμματος, αυξάνεται ο συνολικός χρόνος εκτέλεσης, αλλά η σχετική θέση των αλγορίθμων στο διάγραμμα παραμένει σχεδόν ίδια, ακόμη και για μικρά μεγέθη σωρού.

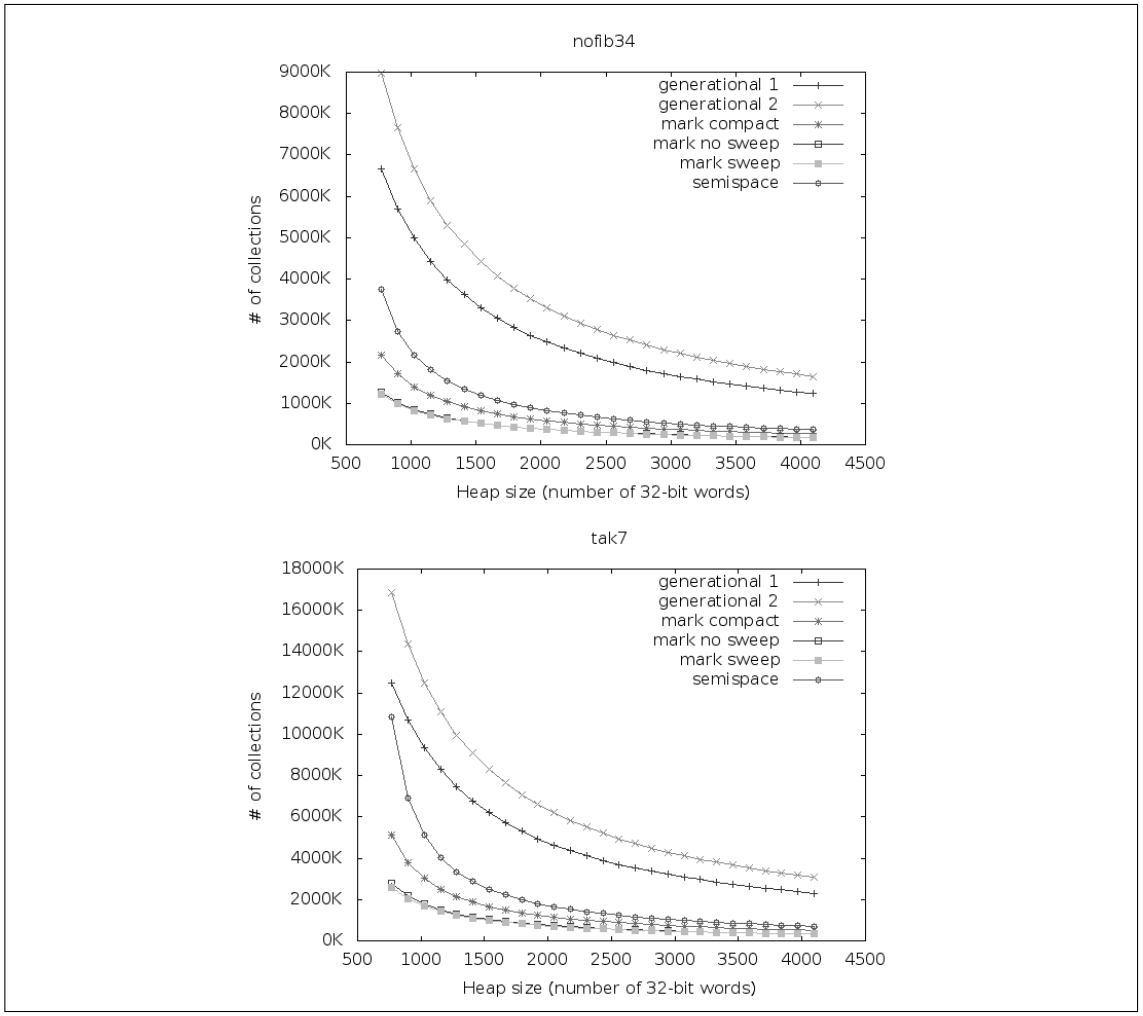
Καλύτερος αλγόριθμος θεωρείται και πάλι ο γενετικός αλγόριθμος σημείωσης συμπύκνωσης, αφού προηγείται όλων στα μικρά και μεσαία μεγέθη σωρού, χάνοντας μόνο στα μεγάλα μεγέθη από τον αλγόριθμο διακοπής αντιγραφής.

8.2.3 Σύγκριση 3: Αριθμός συλλογών που πραγματοποιούνται

Στο σχήμα 8.3 παρουσιάζεται διαγραμματικά ο αριθμός των συλλογών σκουπιδιών που πραγματοποιεί ο κάθε αλγόριθμος σε καθένα από τα δύο προηγούμενα προγράμματα. Ενδεικτικά χρησιμοποιήθηκαν για τα προγράμματα αυτά οι πιο απαιτητικές είσοδοι.

Από τα διαγράμματα του σχήματος αυτού οδηγούμαστε στο συμπέρασμα ότι ο αριθμός των συλλογών κάθε αλγορίθμου παρουσιάζει μία σχεδόν εκθετική μείωση σε σχέση με την αύξηση του μεγέθους του σωρού. Οι αλγόριθμοι με τις μεγαλύτερες συλλογές είναι ο γενετικός αλγόριθμος 2 με αμέσως επόμενο τον γενετικό αλγόριθμο 1. Ακολουθούν οι αλγόριθμοι διακοπής αντιγραφής, σημείωσης συμπύκνωσης, σημείωσης χωρίς σάρωση και τέλος ο αλγόριθμος σημείωσης σάρωσης, ο οποίος μαζί με εκείνον της σημείωσης χωρίς σάρωση, πραγματοποιεί τις λιγότερες συλλογές από όλους τους υπόλοιπους. Ωστόσο αυτοί οι δύο αλγόριθμοι έχουν μακράν τις χειρότερες επιδόσεις από τους υπόλοιπους. Αρκεί να παρατηρήσουμε ότι παρόλο που κάνουν λίγο λιγότερες συλλογές από τον αλγόριθμο σημείωσης συμπύκνωσης που απαιτεί τρία περάσματα στο σωρό, υστερούν αρκετά απέναντί του στο χρόνο εκτέλεσης. Άμεση συνέπεια του γεγονότος αυτού, είναι ότι αυτοί οι δύο αλγόριθμοι επιβαρύνονται σημαντικά από τον κατακερματισμό του σωρού και τη μεγαλύτερη ευθύνη για την επιβάρυνση αυτή φέρει η συνάρτηση ανάθεσης μνήμης.

Οι γενετικοί αλγόριθμοι έχουν εμφανώς το μεγαλύτερο πλήθος συλλογών, τουλάχιστον



Σχήμα 8.3: Προγράμματα nofib και tak – Πλήθος συλλογών σκουπιδιών σε συνάρτηση με το μέγεθος του σωρού.

Γενετικός αλγόριθμος 1 – tak7			
Μέγεθος σωρού	Συνολικές Συλλογές	Μεγάλες Συλλογές	Ποσοστό
768 λέξεις	12504842	2507821	20,05 %
1000 λέξεις	9539561	744703	7,81 %
2000 λέξεις	4751074	125142	2,63 %
3000 λέξεις	3155915	57822	1,83 %
4000 λέξεις	2366357	35256	1,49 %
Γενετικός αλγόριθμος 2 – tak7			
Μέγεθος σωρού	Συνολικές Συλλογές	Μεγάλες Συλλογές	Ποσοστό
768 λέξεις	16873187	14933144	88,5 %
1000 λέξεις	12768679	2917017	22,85 %
2000 λέξεις	6334744	238050	3,76 %
3000 λέξεις	4211444	98050	2,33 %
4000 λέξεις	3155915	57822	1,83 %

Πίνακας 8.1: Συλλογές σκουπιδιών στους γενετικούς αλγορίθμους

τετραπλάσιο(!) από τους υπόλοιπους. Παρόλα αυτά έρχονται πρώτοι στις μετρήσεις χρόνου εκτέλεσης. Αυτό το οφείλουν στο γεγονός ότι οι περισσότερες συλλογές συλλέγουν μόνο ένα μικρό κομμάτι του σωρού (γενιά 1). Στο σημείο αυτό έχει ενδιαφέρον να δούμε πόσες από τις συλλογές αυτές πραγματοποιούνται σε ολόκληρο το σωρό.

Ο πίνακας 8.1 εμφανίζει ενδεικτικά το πλήθος των συλλογών σκουπιδιών κάθε γενετικού αλγορίθμου για το πρόγραμμα tac και το ποσοστό των συνολικών συλλογών που αποτελούν οι μεγάλες. Τα αποτελέσματα είναι εντυπωσιακά και δικαιολογούν απολύτως την αύξηση των επιδόσεων των γενετικών αλγορίθμων στα διαγράμματα του χρόνου εκτέλεσης. Διπλασιάζοντας το μέγεθος του σωρού μειώνεται σημαντικά το ποσοστό των μεγάλων συλλογών και μάλιστα γίνεται μικρότερο από 2% για μεγέθη σωρού πάνω από 15 KB. Το πρόγραμμα tak με είσοδο 7 απαιτεί κάτι λιγότερο από 1,5 KB μνήμης για να τρέξει. Οπότε για δεκαπλάσιο μέγεθος σωρού από το ελάχιστο, οι δύο γενετικοί αλγόριθμοι κάνουν περίπου 2 μεγάλες συλλογές για κάθε 100.

Επιστρέφοντας στο σχολιασμό του σχήματος 8.2 καταλήγουμε στο συμπέρασμα ότι ο αριθμός των συλλογών που κάνει ένας αλγόριθμος δεν αποτελεί αξιόπιστο κριτήριο για να εκτιμηθεί η απόδοσή του.

8.3 Συμπεράσματα

Από τις μετρήσεις που διεξήχθησαν βγαίνει το συμπέρασμα ότι οι γενετικοί αλγόριθμοι έχουν εξαιρετικές επιδόσεις, όσο μεγαλώνει το μέγεθος του σωρού. Αντίθετα οι μη–μετακινούντες αλγόριθμοι φαίνεται να αποδίδουν καλά για μικρότερα μεγέθη σωρού. Όσον αφορά τους απλούς μετακινούντες αλγόριθμους, ο αλγόριθμος διακοπής αντιγραφής κερδίζει με διαφορά, παρόλο που το ωφέλιμο μέγεθος σωρού που έχει στη διάθεσή του είναι μικρότερο από εκείνο του αλγόριθμου σημείωσης συμπύκνωσης.

Ο γενετικός αλγόριθμος 1 αναδεικνύεται ως ο καταλληλότερος αλγόριθμος συλλογής

σκουπιδιών για το περιβάλλον του NFlint. Ο συνδυασμός των αλγορίθμων σημείωσης συμπύκνωσης και διακοπής αντιγραφής αποδείχτηκε ιδιαίτερα αποτελεσματικός. Τα τρία περάσματα που κάνει στο σωρό σε κάθε μεγάλη συλλογή του δεν τον εμπόδισαν να είναι καλύτερος σχεδόν σε κάθε μέτρηση. Η εξήγηση πίσω από αυτό είναι ότι επειδή έχει μειωμένες απαιτήσεις σε μνήμη σε σχέση με τον γενετικό αλγόριθμο 2, έχει μεγαλύτερο σωρό να διαθέσει για την πρώτη γενιά. Άρα αυτή γεμίζει πιο αργά, με αποτέλεσμα να προλαβαίνουν περισσότερα αντικείμενα να γίνουν σκουπίδια. Ακόμη και όταν κάνει μεγάλες (major) συλλογές, τα αντικείμενα που βρίσκονται στη δεύτερη γενιά έχουν ήδη επιβιώσει τουλάχιστον μία συλλογή σκουπιδιών (βλ. Κεφάλαιο 7). Επίσης η μετακίνηση των αντικειμένων σε κάθε μεγάλη συλλογή γίνεται λαμβάνοντας υπ' όψιν τη διάταξή τους στο σωρό. Αυτά έχουν σαν αποτέλεσμα στις αρχικές διευθύνσεις του σωρού να βρίσκονται τα μακροβιότερα αντικείμενα του προγράμματος. Τα αντικείμενα αυτά είναι απίθανο να χρειαστεί να μετακινηθούν, γεγονός που επιταχύνει περαιτέρω την συλλογή σκουπιδιών. Η συνάρτηση ανάθεσης μνήμης που έχει δεν τον επιβαρύνει καθόλου αφού είναι μία από τις ταχύτερες. Τέλος, όπως είναι ήδη γνωστό, αποτρέπει τον κατακερματισμό του σωρού, γεγονός που του επιτρέπει να εκτελείται αρκετή ώρα, χωρίς επιπτώσεις στην απόδοσή του.

Κεφάλαιο 9

Συμπεράσματα

Στην εργασία αυτή παρουσιάστηκαν οι βασικές ιδέες πίσω από κάποιους αλγόριθμους συλλογής σκουπιδιών, καθώς και προτάσεις για την υλοποίησή τους, με σκοπό την κατανόηση της έρευνας που έχει γίνει όλα αυτά τα χρόνια σε αυτό το πεδίο. Διαπιστώθηκε επίσης ότι ο θεωρητικός προσδιορισμός του κόστους της συλλογής σκουπιδιών χάνει συχνά την ουσία του, αφού εξαρτάται από πολλούς αστάθμητους παράγοντες.

Η συλλογή σκουπιδιών μπορεί, όπως αποδείχτηκε, να εφαρμοστεί επιτυχώς και με ασφάλεια και σε συστήματα πιστοποιημένου κώδικα, χωρίς σοβαρές επιπτώσεις στο κόστος εκτέλεσης των προγραμμάτων. Οι μετρήσεις δείχνουν ότι ακόμη και οι συλλέκτες που σταματούν τον κόσμο μπορούν να θεωρηθούν αποδοτικοί στα συστήματα αυτά, καθώς δεν υπάρχουν αυστηρές απαιτήσεις πραγματικού χρόνου σε τέτοια περιβάλλοντα.

Στο μέλλον αναμένονται ακόμη καλύτεροι αλγόριθμοι συλλογής σκουπιδιών. Ενδιαφέρουσες προοπτικές παρουσιάζουν οι γενετικοί αλγόριθμοι, αλλά και οι παράλληλοι συλλέκτες σκουπιδιών, αφού τα υπολογιστικά συστήματα γίνονται ολοένα και πιο ισχυρά και αποκτούν περισσότερους επεξεργαστές. Αυτό, σε συνδυασμό με την πρόοδο στον τομέα του πιστοποιημένου κώδικα, θα οδηγήσει τη μηχανική λογισμικού σε νέες κατευθύνσεις.

Παράρτημα 1

Κώδικας των αλγορίθμων συλλογής σκουπιδιών

Ο αλγόριθμος σημείωσης σαρωσης

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4
5  #include "gc_algo.h"
6
7  /* pointer to and size of heap space */
8  uint32_t heap_size   = 0;
9  int32_t  *heap_space = NULL;
10 int32_t  *heap_next  = NULL;
11
12 inline static int32_t isHeapPointer(int32_t *p)
13 {
14    return (!(*p & 0x3) && // is pointer
15            *p >= (int32_t) heap_space && *p < (int32_t) (heap_space + heap_size));
16 }
17
18 void markTuple(int32_t *l)
19 {
20    int32_t tuple_size;
21
22    if ((*(l - 1) >> 1) & 0x01) return;  // already marked
23    *(l - 1) += 2;  // MARK!
24    tuple_size = *(l - 1) >> 2;
25    while (tuple_size-- > 1)
26    {
27       if (isHeapPointer(l)) markTuple((int32_t*) *l);  // mark all tuples recursively
28       l++;
29    }
30 }
31
32 int32_t* sweep(int32_t* l)    // sweep returns pointer to next unmarked tuple (if any)
33 {
34    int32_t tuple_size, size, *p;
35
36    while (l < heap_space + heap_size)
37    {
38       tuple_size = *l >> 2;
39       if ((*l >> 1) & 0x01)   // if tuple is marked
40          *l -= 2;             // UNMARK!
41       else break;
42       l += tuple_size;
43    }
44    if (l >= heap_space + heap_size) return NULL; //end of heap!
45    p = l + tuple_size;
46    size = tuple_size;
47    while ((p < heap_space + heap_size) && !((*p >> 1) & 0x01)) // while tuples in heap are not
          marked
```

```
48      {
49          tuple_size = *p >> 2;
50          p +=  tuple_size;                   // examine next tuple
51          size += tuple_size;                 // merge consequtive unmarked tuples in heap
52      }
53      if (size == 1) return sweep(l + 1);     // size too small... ignore
54      *l = (size << 2) + 1;                   // size = total size of (consequtuve) unmarked tuples
55      *(l + 1) = (int32_t) sweep(p);          // points to next unmarked tuple!
56      return l + 1;
57  }
58
59  static void applyMark(int32_t *ptr)
60  {
61      if (isHeapPointer(ptr)) markTuple((int32_t *) *ptr);
62  }
63
64
65  void gc_algo_collect()
66  {
67      gc_walk_roots(applyMark);         // MARK
68      heap_next = sweep(heap_space);    // SWEEP
69  }
70
71  int32_t *gc_algo_alloc(int32_t size)
72  {
73      int32_t *l, tuple_size, *previous, i;
74
75      size++;
76      previous = (int32_t*) &heap_next;
77      l = (int32_t*) heap_next;
78      while (l)
79      {
80          tuple_size = *(l - 1) >> 2;
81          if ((i = tuple_size - size) >= 0) // fits?
82          {
83              if (i < 2)
84              {                             // remaining space too small to link to free list. leave
                      it!
85                  *previous = *l;           // fix free list pointers
86                  if (i) *(l + size - 1) = 5; // (i = 0 or 1) will be processed in future collections!
87              }
88              else
89              {                             // remaining space is enough to be used in future allocs
                      as a tulpe
90                  *(l + size - 1) = (i << 2) + 1;
91                  *(l + size) = *l;         // set size for remaining free space of tuple
92                  *previous = (int32_t) (l + size); // previous points to this (remaining tuple (
                        unused) space)
93              }
94              *(l - 1) = ((size) << 2) + 1;  // set size for newly created tuple
95              return l;                     // return address of newly created tuple
96          }
97          previous = l;
98          l = (int32_t*) *l;  // l = next tuple in free list
99      }
100     return NULL;  // out of mem!
101 }
102
103 void gc_algo_init()
104 {
105     heap_space = heap_space_ptr();
106     heap_next = heap_space + 1;             // points to first tuple in free space
107     heap_size = get_heap_size();
108     *(heap_space) = (heap_size << 2) + 1;  // whole heap size (unmarked)
109     *(heap_space + 1) = (int32_t) NULL;    // pointer to next free space
110 }
111
112 void gc_algo_final()
113 {
114 }
```

Ο αλγόριθμος σημείωσης χωρίς σάρωση

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4
5  #include "gc_algo.h"
6
7  /* pointer to and size of heap space */
8  uint32_t heap_size   = 0;
9  int32_t  heap_next   = 0;
10 int32_t  *heap_space = NULL;
11 /* this is used to mark/unmark tuples */
12 int32_t  mark_bit    = 0;
13
14 inline static int32_t isHeapPointer(int32_t *p)
15 {
16    return (!(*p & 0x3) && // is pointer
17           *p >= (int32_t) heap_space && *p < (int32_t) (heap_space + heap_size)); /
18 }
19
20 static void markTuple(int32_t *l)
21 {
22    int32_t tuple_size;
23
24    if (((*(l - 1) >> 1) & 0x01) == mark_bit) return;      // already marked as used
25    tuple_size = *(l - 1) >> 2;
26    *(l - 1) = (((tuple_size << 1) + mark_bit) << 1) + 1; // mark as used
27    while (tuple_size-- > 1)
28    {
29       if (isHeapPointer(l)) markTuple((int32_t*) *l);     // mark (as used) all reachable tuples
             recursively
30       l++;
31    }
32 }
33
34 static void applyMark(int32_t *ptr)
35 {
36    if (isHeapPointer(ptr)) markTuple((int32_t *) *ptr);
37 }
38
39 void gc_algo_collect()
40 {
41    mark_bit = !mark_bit;      // swap mark bit. Every marked tuple is now an enemy of the empire
          .. eeh unmarked
42    gc_walk_roots(applyMark); // mark unmarked tuples
43    heap_next = 0;
44 }
45
46 int32_t *gc_algo_alloc(int32_t size)
47 {
48    int32_t *l, *p, tuple_size, i;
49
50    size++;
51    l = heap_space + heap_next;          // everything before heap_next has been mark used!
52    while (l < heap_space + heap_size)
53    {
54       tuple_size = (*l >> 2);
55       if (((*l >> 1) & 0x01) != mark_bit)  // skip tuples marked as used
56       {
57          if ((i = tuple_size - size) >= 0) // fits?
58          {
59             *l = (((size << 1) + mark_bit) << 1) + 1;           // set size for newly created
                   tuple
60             if (i)                                              // tuple has remaining space
61                *(l + size) = (((i << 1) + !mark_bit) << 1) + 1; // set size. set remaining space
                      as unused
62             heap_next = (int32_t) (l + size - heap_space);      // continue next allocation from
                    here
63             return l + 1;                                       // return address of newly
                    created tuple
64          }
65          else
66          {
67             p = l + tuple_size;
68             if (p >= heap_space + heap_size)    // reached the end of the heap. return NULL
69                break;
70             if (((*p >> 1) & 0x01) != mark_bit) // found consequtive unmarked tuple
```

```
71              {
72                  tuple_size += (*p >> 2);
73                  *l = (((tuple_size << 1) + !mark_bit) << 1) + 1; // consolidate
74                  continue;                                        // check again if it fits
75              }
76              else                                                 // doesn't fit! Mark as USED!!!
77                  *l = (((tuple_size << 1) + mark_bit) << 1) + 1;  // It will be collected at next
                        collection!
78          }
79      }
80      l += tuple_size;    // examine next tuple
81  }
82  return NULL;    // out of mem!
83  }
84
85  void gc_algo_init()
86  {
87      heap_space = heap_space_ptr();
88      heap_size = get_heap_size();
89      *(heap_space) = (((heap_size << 1) + !mark_bit) << 1) + 1;  // whole heap size (unmarked)
90  }
91
92  void gc_algo_final()
93  {
94  }
```

Ο αλγόριθμος σημείωσης συμπύκνωσης

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdint.h>
4
5   #include "gc_algo.h"
6
7   /* size of each heap space */
8   uint32_t heap_size        = 0;
9   uint32_t reloc_table_size = 0;
10  /* pointers to heap spaces */
11  int32_t *heap_space       = NULL;
12  int32_t *reloc_table      = NULL;
13  /* next free element in heap_space (used by gc_algo_alloc) */
14  int32_t heap_next         = 0;
15
16  inline static int32_t isHeapPointer(int32_t *p)
17  {
18    return (!(*p & 0x3) && // is pointer
19            *p >= (int32_t) heap_space && *p < (int32_t) (heap_space + heap_size)); // points to
                heap?
20  }
21
22  inline static int32_t isTablePointer(int32_t *p) // is pointer to relocation table?
23  {
24    return (!(*p & 0x3) && // is pointer
25            *p >= (int32_t) reloc_table && *p < (int32_t) (reloc_table + reloc_table_size));
26  }
27
28  void markTuple(int32_t *l)
29  {
30      int32_t tuple_size;
31
32      if ((*(l - 1) >> 1) & 0x01) return;  // already marked
33      *(l - 1) += 2;  // MARK!
34      tuple_size = *(l - 1) >> 2;
35      while (tuple_size--- > 1)
36      {
37          if (isHeapPointer(l))
38          {
39              markTuple((int32_t*) *l);      // mark all reachable tuples recursively
40              *l = (int32_t) (reloc_table + (*l / 8) % reloc_table_size); // now points to reloc
                    table
41          }
42          l++;
43      }
44  }
```

```
45
46  void compact()
47  {
48     int32_t size, tuple_size, *l, *next;
49
50     next = NULL;
51     l = heap_space;
52     while (l < heap_space + heap_next)
53     {
54       tuple_size = *l >> 2;
55       if ((*l >> 1) & 0x01)  // if l is marked
56       {
57           *l -= 2;           // UNMARK
58           if (next)
59           {  // store new tuple address at relocation table
60              *(reloc_table + (int32_t) (((int32_t) (l + 1)) / 8) % reloc_table_size) = (int32_t)
                    (next + 1);
61              for (size = 0; size < tuple_size; size++) // move tuple
62                 *(next++) =  *(l + size);
63           }
64           else // store tuple address at relocation table
65              *(reloc_table + (int32_t) (((int32_t) (l + 1)) / 8) % reloc_table_size) = (int32_t)
                    (l + 1);
66       }
67       else if (!next) next = l; // reached first unmarked tuple in heap
68       l += tuple_size;
69     }
70     if (next) heap_next = (int32_t) (next - heap_space);
71     l = heap_space + 1;
72     for (size = 0; size < heap_next - 1; size++)     // fix pointers
73     {
74        if (isTablePointer(l)) *l = *((int32_t*) *l); // Get correct tuple address from relocation
              table!
75        l++;
76     }
77  }
78
79  static void applyMark(int32_t *ptr)
80  {
81     if (isHeapPointer(ptr))
82     {
83        markTuple((int32_t *) *ptr);
84        *ptr = (int32_t) (reloc_table + (*ptr / 8) % reloc_table_size); // root now points to
               reloc table
85     }
86  }
87
88  static void fixPointers(int32_t *ptr)
89  {
90     if (isTablePointer(ptr)) *ptr = *((int32_t*) *ptr); // fix root pointers pointing to
           relocation table
91  }
92
93  void gc_algo_collect()
94  {
95     gc_walk_roots(applyMark);    // MARK
96     compact();                   // COMPACT
97     gc_walk_roots(fixPointers); // root tuple pointers point to relocation table. Fix them!
98  }
99
100 int32_t *gc_algo_alloc(int32_t size)
101 {
102    if (heap_next + ++size <= heap_size) // fits?
103    {
104       heap_next += size;
105       *(heap_space + heap_next - size) = (size << 2) + 1; // set size
106       return heap_space + heap_next - size + 1; // return pointer to new tuple
107    }
108    else return NULL; // out of mem!
109 }
110
111 void gc_algo_init()
112 {
113    heap_space        = heap_space_ptr();
114    reloc_table_size  = get_heap_size() / 3;
115    if (get_heap_size() % 3) reloc_table_size++;
116    heap_size         = get_heap_size() - reloc_table_size;
117    reloc_table       = heap_space + heap_size;
118 }
```

```
119
120 void gc_algo_final()
121 {
122 }
```

Ο αλγοριθμος διακοπής αντιγραφής

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4
5  #include "gc_algo.h"
6
7  /* size of each heap space */
8  uint32_t heap_size    = 0;
9  /* pointers to heap spaces */
10 int32_t *heap_space1  = NULL;
11 int32_t *heap_space2  = NULL;
12 /* next free element in heap_space1 (used by gc_alloc) */
13 int32_t  heap_next    = 0;
14
15 inline static int32_t isHeapPointer(int32_t *p)
16 {
17    return (!(*p & 0x3) && // is pointer?
18           *p >= (int32_t) heap_space1 && *p < (int32_t) (heap_space1 + heap_size)); // points to
                heap?
19 }
20
21 int32_t* copyTuples(int32_t *l)
22 {
23    int32_t tuple_size, size, *p;
24
25    if ((*(l - 1) & 3) == 3)                        // tuple is already visited!
26       return (int32_t*) ((int32_t) *(l - 1) - 3); // return forwarding pointer
27    tuple_size = *(l - 1) >> 2;
28    p = heap_space2 + heap_next;
29    for (size = 0; size < tuple_size; size++)       // copy tuple from from-space (l) to to-space
          (p)
30       *(p + size) =  *(l + size - 1);
31    *(l - 1) = (int32_t) (p++ + 1) + 3;             // mark as visited! now contains new tuple
          addr + 3
32    heap_next += tuple_size;
33    while (tuple_size-- > 1)                        // scan and copy all (reachable) tuples
          recursively
34    {
35       if (isHeapPointer(p)) *p = (int32_t) copyTuples((int32_t*) *p); // update pointer
36       p++;
37    }
38    return p - size + 1;                            // size == tuple_size (from the for-loop)
39 }
40
41
42 static void applySemiSpace(int32_t *ptr)
43 {
44    if (isHeapPointer(ptr)) *ptr = (int32_t) copyTuples((int32_t *) *ptr);
45 }
46
47
48 void gc_algo_collect()
49 {
50    int32_t *tmp;
51
52    heap_next = 0;
53    gc_walk_roots(applySemiSpace); // start garbage collection
54    tmp = heap_space1;             // swap spaces
55    heap_space1 = heap_space2;
56    heap_space2 = tmp;
57 }
58
59 int32_t *gc_algo_alloc(int32_t size)
60 {
61    if (heap_next + ++size <= heap_size) // fits?
62    {
63       heap_next += size;
```

```
64          *(heap_space1 + heap_next - size) = (size << 2) + 1; // set size
65          return heap_space1 + heap_next - size + 1; // return pointer to new tuple
66      }
67      else return NULL; // out of mem!
68  }
69
70  void gc_algo_init()
71  {
72      heap_size   = get_heap_size() / 2;
73      heap_space1 = heap_space_ptr();
74      heap_space2 = heap_space1 + heap_size;
75  }
76
77  void gc_algo_final()
78  {
79  }
```

Γενετικός αλγόριθμος 1

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdint.h>
4
5   #include "gc_algo.h"
6
7   /* size of each heap space */
8   uint32_t heap_size        = 0;
9   int32_t  gen0_size        = 0;
10  int32_t  gen1_size        = 0;
11  uint32_t reloc_table_size = 0;
12  /* pointers to heap spaces */
13  int32_t *heap_space       = NULL;
14  int32_t *gen0             = NULL;
15  int32_t *gen1             = NULL;
16  int32_t *reloc_table      = NULL;
17  /* next free element in heap_spaces */
18  int32_t gen0_next         = 0;
19  int32_t gen1_next         = 0;
20  /* for statistics */
21  int32_t whole_heap_collections = 0;
22
23  inline static int32_t isGen0Pointer(int32_t *p)
24  {
25      return (!(*p & 0x3) && // is pointer
26              *p >= (int32_t) gen0 && *p < (int32_t) (gen0 + gen0_size)); // points to gen0?
27  }
28
29  inline static int32_t isHeapPointer(int32_t *p)
30  {
31      return (!(*p & 0x3) && // is pointer
32              *p >= (int32_t) heap_space && *p < (int32_t) (heap_space + heap_size)); //points to
                heap?
33  }
34
35  inline static int32_t isTablePointer(int32_t *p)  // is pointer to relocation table?
36  {
37      return (!(*p & 0x3) && // is pointer
38              *p >= (int32_t) reloc_table && *p < (int32_t) (reloc_table + reloc_table_size)); //
                points to table?
39  }
40
41  int32_t* copyTuple(int32_t *l)
42  {
43      int32_t tuple_size, size, *p;
44
45      if ((*(l - 1) & 3) == 3)                        // tuple is already visited!
46          return (int32_t*) ((int32_t) *(l - 1) - 3);  // return forwarding pointer
47      tuple_size = *(l - 1) >> 2;
48      p = gen1 + gen1_next;
49      for (size = 0; size < tuple_size; size++)       // copy tuple from from-space (l) to to-space
            (p)
50          *(p + size) =  *(l + size - 1);
51      *(l - 1) = (int32_t) (p++ + 1) + 3;             // mark as visited! now contains new tuple
            addr + 3
```

```
52      gen1_next += tuple_size;
53      while (tuple_size--- > 1)                       // scan and copy all reachable tuples
            recursively
54      {
55          if (isGen0Pointer(p)) *p = (int32_t) copyTuple((int32_t*) *p); // update pointer
56          p++;
57      }
58      return p - size + 1;                            // size == tuple_size (from for-loop)
59  }
60
61  void markTuple(int32_t *l)
62  {
63      int32_t tuple_size;
64
65      if ((*(l - 1) >> 1) & 0x01) return;  // already marked
66      *(l - 1) += 2; // MARK!
67      tuple_size = *(l - 1) >> 2;
68      while (tuple_size--- > 1)
69      {
70          if (isHeapPointer(l))
71          {
72              markTuple((int32_t*) *l);      // mark all reachable tuples recursively
73              *l = (int32_t) (reloc_table + (*l / 8) % reloc_table_size); // now points to the reloc
                    table
74          }
75          l++;
76      }
77  }
78
79  void compact()
80  {
81      int32_t tuple_size, *next, *l, size;
82
83      next = NULL;
84      l = gen1;
85      while (l < gen0 + gen0_next)
86      {
87          tuple_size = *l >> 2;
88          if ((*l >> 1) & 0x01)           // if l is marked
89          {
90              *l -= 2;                    // UNMARK!
91              gen1_next += tuple_size;
92              if (next)
93              {  // store new tuple address at relocation table
94                  *(reloc_table + (int32_t) (((int32_t) (l + 1)) / 8) % reloc_table_size) = (int32_t)
                        (next + 1);
95                  for (size = 0; size < tuple_size; size++) // move tuple
96                      *(next++) =  *(l + size);
97              }
98              else // store current tuple address at relocation table
99                  *(reloc_table + (int32_t) (((int32_t) (l + 1)) / 8) % reloc_table_size) = (int32_t)
                        (l + 1);
100         }
101         else if (!next) next = l; // reached first unmarked tuple in heap
102         l += tuple_size;
103     }
104     gen0_next = (gen1_next > gen1_size) ? gen1_next - gen1_size : 0; // set current gen0 size
105     l = heap_space + 1;
106     for (size = 0; size < gen1_next - 1; size++)
107     {
108         if (isTablePointer(l)) *l = *((int32_t*) *l); // Get correct tuple address from relocation
                table!
109         l++;
110     }
111 }
112
113 static void applySemiSpace(int32_t *ptr)  // only gen0
114 {
115     if (isGen0Pointer(ptr)) *ptr = (int32_t) copyTuple((int32_t *) *ptr);
116 }
117
118 static void applyMark(int32_t *ptr)       // whole heap
119 {
120     if (isHeapPointer(ptr))
121     {
122         markTuple((int32_t *) *ptr);
123         *ptr = (int32_t) (reloc_table + (*ptr / 8) % reloc_table_size);
124     }
125 }
```

```
126
127 static void applyPointerFix(int32_t *ptr) // only for roots
128 {
129    if (isTablePointer(ptr)) *ptr = *((int32_t*) *ptr);
130 }
131
132 void gc_algo_collect()
133 {
134    if (gen1_size - gen1_next >= gen0_next) //fits?
135    {
136       gc_walk_roots(applySemiSpace);     // collect gen0 with semispace
137       gen0_next = 0;
138    }
139    else
140    {
141       whole_heap_collections++;
142       gen1_next = 0;
143       gc_walk_roots(applyMark);          // collect whole heap with mark & compact
144       compact();
145       gc_walk_roots(applyPointerFix);    // fix the root pointers still pointing to reloc table
146    }
147    *(gen1 + gen1_next) = ((gen1_size - gen1_next) << 2) + 1;
148 }
149
150 int32_t *gc_algo_alloc(int32_t size)
151 {
152    if (gen0_next + ++size <= gen0_size) // fits?
153    {
154       gen0_next += size;
155       *(gen0 + gen0_next - size) = (size << 2) + 1; // set size
156       return gen0 + gen0_next - size + 1; // return pointer to new tuple
157    }
158    else return NULL; // out of mem!
159 }
160
161 void gc_algo_init()
162 {
163    reloc_table_size = get_heap_size() / 3;
164    if (get_heap_size() % 3) reloc_table_size++;
165    gen0_size        = (get_heap_size() - reloc_table_size) / 5;
166    gen1_size        = get_heap_size() - reloc_table_size - gen0_size;
167    heap_size        = gen0_size + gen1_size;
168    heap_space       = heap_space_ptr();
169    gen1             = heap_space;
170    gen0             = gen1 + gen1_size;
171    reloc_table      = gen0 + gen0_size;
172 }
173
174 void gc_algo_final()
175 {
176 }
```

Γενετικός αλγόριθμος 2

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <stdint.h>
 4
 5 #include "gc_algo.h"
 6
 7 /* size of each heap space */
 8 uint32_t heap_size        = 0;
 9 int32_t  gen0_size        = 0;
10 int32_t  gen1_size        = 0;
11 /* pointers to heap spaces */
12 int32_t *heap_space1      = NULL;
13 int32_t *heap_space2      = NULL;
14 int32_t *gen0             = NULL;
15 int32_t *gen1             = NULL;
16 /* next free element in heap_spaces */
17 int32_t gen0_next         = 0;
18 int32_t gen1_next         = 0;
19
20 inline static int32_t isGen0Pointer(int32_t *p)
```

```
21 {
22    return (!(*p & 0x3) && // is pointer
23            *p >= (int32_t) gen0 && *p < (int32_t) (gen0 + gen0_size));  // points to gen0?
24 }
25
26 inline static int32_t isHeapPointer(int32_t *p)
27 {
28    return (!(*p & 0x3) && // is pointer
29            *p >= (int32_t) heap_space1 && *p < (int32_t) (heap_space1 + heap_size)); //points to
                 heap?
30 }
31
32 int32_t* copyTuple(int32_t *l, int32_t vl)
33 {
34    int32_t tuple_size, size, *p;
35
36    if ((*(l - 1) & 3) == 3)                        // tuple is already visited!
37       return (int32_t*) ((int32_t) *(l - 1) - 3); // return forwarding pointer
38    tuple_size = *(l - 1) >> 2;
39    p = gen1_next + (vl == 1 ? heap_space1 : heap_space2);
40    for (size = 0; size < tuple_size; size++)       // copy tuple from from-space (l) to to-space
          (p)
41       *(p + size) =  *(l + size - 1);
42    *(l - 1) = (int32_t) (p++ + 1) + 3;             // mark as visited! now contains new tuple
          addr + 3
43    gen1_next += tuple_size;
44    while (tuple_size-- > 1)                        // scan and copy all reachable tuples
          recursively
45    {
46       if (vl == 1)
47       {
48          if (isGen0Pointer(p)) *p = (int32_t) copyTuple((int32_t*) *p, 1); // update pointer
49       }
50       else
51          if (isHeapPointer(p)) *p = (int32_t) copyTuple((int32_t*) *p, 2); // update pointer
52       p++;
53    }
54    return p - size + 1;                            //size == tuple_size (from for-loop)
55 }
56
57 static void applySemiSpace(int32_t *ptr)  // only gen0
58 {
59    if (isGen0Pointer(ptr)) *ptr = (int32_t) copyTuple((int32_t *) *ptr, 1);
60 }
61
62 static void applySemiSpace2(int32_t *ptr) // whole heap
63 {
64    if (isHeapPointer(ptr)) *ptr = (int32_t) copyTuple((int32_t *) *ptr, 2);
65 }
66
67 void gc_algo_collect()
68 {
69    int32_t *tmp;
70
71    if (gen1_size - gen1_next >= gen0_next) // fits?
72    {
73       gc_walk_roots(applySemiSpace);       // collect only gen0 with semispace
74       gen0_next = 0;
75    }
76    else
77    {
78       gen1_next   = 0;
79       gc_walk_roots(applySemiSpace2);      // collect the whole heap with semispace
80       tmp         = heap_space1;           // swap spaces
81       heap_space1 = heap_space2;
82       gen1        = heap_space2;
83       heap_space2 = tmp;
84       gen0        = gen1 + gen1_size;
85       gen0_next   = (gen1_next > gen1_size) ? gen1_next - gen1_size : 0;
86    }
87 }
88
89 int32_t *gc_algo_alloc(int32_t size)
90 {
91    if (gen0_next + ++size <= gen0_size) // fits?
92    {
93       gen0_next += size;
94       *(gen0 + gen0_next - size) = (size << 2) + 1; // set size
95       return gen0 + gen0_next - size + 1; // return pointer to new tuple
```

```
96      }
97      else return NULL; // out of mem!
98  }
99
100 void gc_algo_init()
101 {
102     heap_size   = get_heap_size() / 2;
103     heap_space1 = heap_space_ptr();
104     heap_space2 = heap_space1 + heap_size;
105     gen0_size   = (heap_size / 5);
106     gen1_size   = heap_size - gen0_size;
107     gen1        = heap_space1;
108     gen0        = gen1 + gen1_size;
109 }
110
111 void gc_algo_final()
112 {
113 }
```

Παράρτημα 2

Κώδικας της διεπαφής

gc_algo.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <time.h>
5  #include <sys/time.h>
6  #include <sys/resource.h>
7
8  #include "gc_algo.h"
9
10 /* heap space pointer */
11 static int32_t *heap_space;
12
13 /* runtime interface to LLVM for liveness analysis */
14 extern int32_t *stack_base;
15 extern uint32_t live_slots;
16 extern uint32_t live_regs;
17
18
19 void gc_init(void)
20 {
21   heap_space = calloc(HEAP_SPACE_SIZE, sizeof(int32_t));
22   collect_calls = 0;
23   alloc_calls = 0;
24   gc_algo_init();
25 }
26
27 int32_t *heap_space_ptr(void)
28 {
29   return heap_space;
30 }
31
32 int32_t get_heap_size(void)
33 {
34   return HEAP_SPACE_SIZE;
35 }
36
37 int32_t *gc_alloc(int32_t size)
38 {
39   int32_t *retval;
40
41   if (size == 0)
42     return NULL;
43
44   // Try to allocate resource
45   retval = gc_algo_alloc(size);
46
47   // allocation successful. get out of here
48   if ( retval != NULL )
49     return retval;
50
51   // not enough size! Trigger collection
52   gc_algo_collect();
```

```
53
54    // Try to allocate resource again
55    retval = gc_algo_alloc(size);
56
57    // allocation successful. get out of here
58    if (retval != NULL)
59      return retval;
60
61    // not enough size! give up and terminate program
62    printf("gc_out_of_mem!\n");
63    exit(1);
64  }
65
66  void gc_final(void)
67  {
68    gc_algo_final();
69  }
70
71  void gc_walk_roots(void (*walk_root)(int32_t *))
72  {
73    int i;
74    // examine live stack slots
75    for (i = 0; i < 31; i++)
76    {
77      int32_t *ptr = stack_base + 1 + i;
78      if ((live_slots >> i) & 0x1)
79      {
80        int32_t val = *ptr;
81        walk_root(ptr);
82      }
83    }
84
85    // examine live regs (dumped on stack)
86    for (i = 0; i < 8; i++)
87    {
88      int32_t *ptr = stack_base - 1 - i;
89      if ( (live_regs >> 4 * i) & 0xf )
90      {
91        int32_t val = *ptr;
92        walk_root(ptr);
93      }
94    }
95  }
```

gc_algo.h

```
1   #ifndef _GC_ALGO_H
2   #define _GC_ALGO_H
3
4   #include <stdint.h>
5
6   /*
7     Size of the heap space (number of 32-bit words)
8   */
9   #define HEAP_SPACE_SIZE (2048)
10
11  /*
12    ============================================================
13    The following functions must be provided by the algorithm
14    ============================================================
15  */
16
17  /*
18    called by gc_init
19    For algo-specific initialization
20  */
21  void gc_algo_init(void);
22
23  /*
24    called by gc_alloc
25    Returns a valid location on the heap, or null if there is not enough space
26  */
27  int32_t *gc_algo_alloc(int32_t size);
28
```

```
29 /*
30    called by gc_alloc when there is not enough space
31    Implements the collection algorithm
32 */
33 void gc_algo_collect(void);
34
35 /*
36    called by gc_final
37 */
38 void gc_algo_final(void);
39
40 /*
41   ============================================================
42    Garbage collector runtime interface (implemented by gc_algo)
43   ============================================================
44 */
45
46 /*
47    called by the binary on program start for memory allocation
48 */
49 void gc_init(void);
50
51 /*
52    called by the binary on allocation request
53 */
54 int32_t *gc_alloc(int32_t size);
55
56 /*
57    called by the binary prior to program exit ("final" function)
58 */
59 void gc_final(void);
60
61 /*
62   ============================================================================
63    The following functions are provided by gc_algo for the algorithms to use
64   ============================================================================
65 */
66
67 /*
68    retrieve a pointer to the beggining of the heap space
69 */
70 int32_t *heap_space_ptr(void);
71
72 /*
73    retrieves the size of the heap space
74 */
75 int32_t get_heap_size(void);
76
77 /*
78    iterates over the live roots. Argument is a function called for each root
79 */
80 void gc_walk_roots(void (*walk_root)(int32_t *));
81
82 /*
83     Debug wrapper
84 */
85
86 //#define DEBUG(format, args...) printf(format , ## args)
87 #define DEBUG(format, args...)
88
89 #endif /* _GC_ALGO_H */
```

Βιβλιογραφία

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[Che70] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, Oct. 1983.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically-typed-language. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 273–282, San Fransisco, California, USA, June 1992.

[Hug82] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software – Practice and Experience*, 12(11):1081–1084, November 1982.

[HW67] Bruce K. Haddon and William M. Waite. A compaction procedure for variable length storage elements. *Computer Journal*, 10:162–165, August 1967.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[McB63] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computations by machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[Min63] Marvin L. Minsky. A lisp garbage collector algorithm using serial and secondary storage. A.I. Memo 58, Project MAC, MIT, Cambridge, Massachusetts, 1963.

[San91] Patrick M. Sansom. Combining copying and compacting garbage collection. In *Functional Programming*, Glasgow, Scotland, UK, August 1991. Springer-Verlag.

[Sha88] Robert A. Shaw. Empirical analysis of a lisp system. phd thesis. Stanford University, Stanford, California, February 1988. Also appears as a Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory, 1988.

[SSTP02] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certifed binaries. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL 2002)*, pages 217–232, Portland, Oregon, USA, 2002.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the oportunistic garbage collector. In *ACM SIGPLAN 1989 Conference on Object Orientated Programming Systems, Languages and Applications. (OOPSLA '89)*, pages 23–35, New Olreans, Luisiana, USA, October 1989.