

Σχεδιασμός Ενσωματωμένων Συστημάτων

Φιλιππόπουλος Ορφέας
Παπαρρηγόπουλος Θοδωρής

el18082
el18040

2η Εργαστηριακή Άσκηση

1. Εισαγωγή – Σκοπός της Άσκησης

Ο σκοπός της άσκησης είναι να βελτιστοποιηθούν οι δυναμικές δομές δεδομένων του Deficit Round Robin (DRR) και του Dijkstra, με χρήση της μεθοδολογίας «Βελτιστοποίησης Δυναμικών Δομών Δεδομένων» - Dynamic Data Type Refinement (DDTR).

Θα βελτιστοποιήσουμε τις δομές δεδομένων ως προς:

- Τα memory accesses
- Το memory footprint

2. Deficit Round Robin

2.1 Περιγραφή Διαδικασίας

Χρησιμοποιώντας τον ήδη υλοποιημένο αλγόριθμο DRR στα drr.h και drr.c, θα κάνουμε την βελτιστοποίηση χρησιμοποιώντας Single Linked List, Dynamic Linked List, Dynamic Array.

Τα βήματα που κάναμε είναι τα εξής:

αρχικά build-ραμε τα αντίστοιχα εκτελέσιμα με την εντολή:

```
$ gcc drr.c -o drr -pthread -lcdsl -D{CONFIG_CL} -D{CONFIG_PK}  
-L../synch_implementations -I../sync_implementations (Δείτε αντίστοιχο Makefile)
```

Στη συνέχεια με το **python_run_valgrind_mem_accesses.py** script τρέξαμε τη παρακάτω εντολή (για τα memory accesses) για κάθε ένα διαφορετικό configuration:

```
$ valgrind --log-file="mem_accesses_log_{config_cl}_{config_pk}.txt" --tool=lackey --trace-mem=yes ./drr_{config_cl}_{config_pk}
```

Ομοίως με το **python_run_valgrind_mem_footprint.py** script τρέξαμε την παρακάτω εντολή (για το memory footprint) για κάθε διαφορετικό configuration:

```
$ valgrind --tool=massif ./drr_{config_cl}_{config_pk}
```

Τέλος με το **preprocces.py** script τρέξαμε τις παρακάτω εντολές, και.. πολύ άλλη python, (για να πάρουμε τα αποτελέσματα των παραπάνω) και φέραμε τα αποτελέσματα σε μικρότερα αρχεία (για παράδειγμα το παρακάτω cat έπαιρνε αρκετή ώρα) στο φάκελο **outputs_short_version**.

```
$ cat mem_accesses_log_{config_cl}_{config_pk}.txt | grep 'I|L' | wc -l  
$ ms_print massif.out.XXXX > mem_footprint_log_{config_cl}_{config_pk}.txt
```

Για λόγους πληρότητας με το **plot.py** script πλοτάρουμε τα αποτελέσματα και αποθηκεύσαμε το figure που προέκυψε στο φάκελο plots.

2.2 Αποτελέσματα

Μετά τη συνοπτική παρουσίαση των βημάτων που εκτελέσαμε, παρακάτω παραθέτουμε τα αποτελέσματα που προέκυψαν:

Client List	Packet List	Memory Accesses	Memory Footprint (KB)
SLL	SLL	69350950	798.8
SLL	DLL	70021992	980.3
SLL	DYN_ARR	470472311	1111.0
DLL	SLL	69363614	823.0
DLL	DLL	70034109	983.3
DLL	DYN_ARR	470486898	1128.0
DYN_ARR	SLL	69884637	760.2
DYN_ARR	DLL	70562550	928.5
DYN_ARR	DYN_ARR	471195063	1075.0

2.2.1 Σχολιασμός αποτελεσμάτων

Παρατηρούμε ότι σε κάθε περίπτωση δομής **client list** καθώς πηγαίνουμε **SLL → DLL → DYN_ARR** στη δομή **packet list** το **memory footprint** αυξάνεται με τρόπο παρόμοιο με το γραμμικό. Από αυτό καταλαβαίνουμε πως **καθοριστικό ρόλο** όσον αφορά το **memory footprint** διαδραματίζει το **packet list** και η **δομή** που χρησιμοποιούμε σε αυτό (και όχι τόσο το **client list**). Επιπροσθέτως, όσον αφορά τις **προσβάσεις στη μνήμη** και στις 3 περιπτώσεις του **client list** παρατηρούμε πολύ παρόμοια **αποτελέσματα** (καθώς και μεταξύ του **SLL και DLL στο packet list**) ενώ παρατηρούμε πως σε κάθε περίπτωση όταν χρησιμοποιούμε **DYN_ARR** για **packet list** τα **memory accesses** **αυξάνονται** κατά **μία τάξη μεγέθους**.

Παρακάτω παρουσιάζουμε και με γραφικό τρόπο τα αποτελέσματα που προέκυψαν:

2.2.2 Καλύτερο configuration για Memory Footprint

Από τα παραπάνω αποτελέσματα, στη περίπτωση που ήταν **blocker το μέγεθος της μνήμης** τότε θα επιλέγαμε το configuration:

Client List: Dynamic array

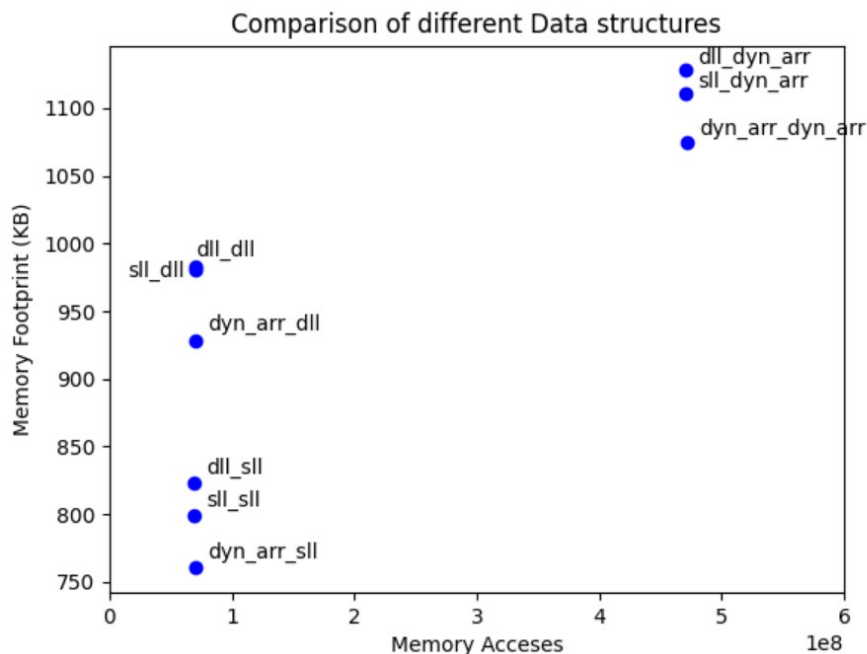
Packet List: Single Linked List

2.2.3 Καλύτερο configuration για Memory Accesses

Αντιθέτως άμα δεν είχαμε θέμα από μέγεθος μνήμης και θέλαμε να επιταχύνουμε την εκτέλεση του DRR με λιγότερες προσβάσεις στη μνήμη, τότε θα επιλέγαμε το configuration:

Client List: Single Linked List

Packet List: Single Linked List



3 Dijkstra

3.1 Περιγραφή Διαδικασίας

Χρησιμοποιώντας τον ήδη υλοποιημένο αλγόριθμο Dijkstra στο dijkstra.c, θα κάνουμε την βελτιστοποίηση χρησιμοποιώντας Single Linked List, Dynamic Linked List, Dynamic Array.

Τα βήματα που κάναμε είναι τα εξής:

Το πρώτο βήμα που κάναμε ήταν να τρέξουμε dijkstra.c και να καταγράψουμε τα αποτελέσματα (dijkstra.txt).

Μετά, τροποποιήσαμε το κώδικα προκειμένου να αξιοποιήσουμε τη βιβλιοθήκη DDTR για να εξάγουμε τα ζητούμενα αποτελέσματα. Αφού το κάναμε αυτό (**dijkstra_dynamic.c**), προκειμένου να αποφανθούμε για το εάν έχει γίνει σωστή η εισαγωγή, τρέξαμε το κώδικα για κάθε ένα configuration που ζητείται και αποθηκεύσαμε τα αποτελέσματα στα αρχεία:

dijkstra_dynamic_sll.txt

dijkstra_dynamic_dll.txt

dijkstra_dynamic_dyn_arr.txt

αντίστοιχα.

Τα αποτελέσματα τα ελέχθηκαν και χειροκίνητα για την ορθή λειτουργία του προγράμματος. Επιπροσθέτως στο φάκελο dijkstra_with_results μπορείτε να βρείτε τους τροποποιημένους κώδικες οι οποίοι γράφουν τα αποτελέσματα σε αρχείο, για να γράφουμε αυτόματα τα αποτελέσματα στα παραπάνω .txt.

Στη συνέχεια ξανα-build-ραμε (αφού βγάλαμε το κώδικα που γράφει τα αποτελέσματα στο αντίστοιχο αρχείο .txt) τα αντίστοιχα εκτελέσιμα με την εντολή:

```
$ gcc dunamic_dijkstra.c -o dynamic_dijkstra_{CONFIG} -pthread -lcdsl -D{CONFIG} -L../synch_implementations -I../sync_implementations
```

Τα βάλουμε σε ένα Makefile για να είναι πιο αυτοματοποιημένα, πριν τα είχαμε τρέξει στο όλα terminal.

Ακολουθως με το **python_run_valgrind_mem_accesess.py** script τρέξαμε τη παρακάτω εντολή (για τα memory accesses) για κάθε ένα διαφορετικό configuration:

```
$ valgrind --log-file="mem_accesses_log_{config}.txt" --tool=lackey --trace-mem=yes  
./dijkstra_dynamic_{config} input.dat
```

Ομοίως με το **python_run_valgrind_mem_footprint.py** script τρέξαμε την παρακάτω εντολή (για το memory footprint) για κάθε διαφορετικό configuration:

```
$ valgrind --tool=massif ./dijkstra_dynamic_{config}
```

Τέλος με το **preprocces.py** script τρέξαμε τις παρακάτω εντολές, για να πάρουμε τα αποτελέσματα των παραπάνω και φέραμε τα αποτελέσματα σε μικρότερα αρχεία για ένα είδος chache (επειδή κάποιες εντολές έπερναν αρκετή ώρα) στο φάκελο **outputs_short_version**.

```
$ cat mem_accesses_log_{config}.txt | grep 'T\| L' | wc -l  
$ ms_print massif.out.XXXX > mem_footprint_log_{config}.txt
```

Για λόγους πληρότητας με το **plot.py** script πλοτάρουμε τα αποτελέσματα και αποθηκεύσαμε το figure που προέκυψε στο φάκελο plots.

3.2 Αποτελέσματα

Data structure	Memory Accesses	Memory Footprint (KB)
SLL	102215081	710.7
DLL	102393534	944.7
DYN_ARR	149446873	363.6

Παρακάτω παρουσιάζουμε και με γραφικό τρόπο τα αποτελέσματα που προέκυψαν:

3.2.1 Καλύτερο configuration για Memory Accesses

Παρατηρούμε ότι όσον αφορά στον αριθμό προσβάσεων στη μνήμη το καλύτερο configuration είναι το **Single Linked List**.

3.2.2 Καλύτερο configuration για Memory Footprint

Αντιθέτως όσον αφορά το memory footprint το καλύτερο configuration είναι (με διαφορά) το **Dynamic Array**.

