HY255 Εργαστήριο Λογισμικού – L23

Performance – Sample Profiling

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

Περί Απόδοσης

- Η καλή/υψηλή απόδοση των συστημάτων είναι ένα βασικός στόχος του σχεδιασμού τους σε όλη την ιστορία των υπολογιστών και ακόμη και σήμερα
  - Εμείς στο HY255 ενδιαφερόμαστε πρωτίστως για την ορθότητα
  - Θεωρώντας ότι τα συστήματα που σχεδιάζουμε είναι σωστά, η απόδοση, ακόμη και σήμερα, είναι ίσως η πιο σημαντική διάσταση για την χρήση τους
  - Δεδομένων των περιορισμών που αντιμετωπίζουμε σήμερα (power and energy limitations, αύξηση του όγκου της δουλειάς που πρέπει να γίνει από υπολογιστές και πολλούς τεχνολογικούς περιορισμούς) η απόδοση σήμερα γίνεται ακόμη πιο σημαντική και για πιο πολλά συστήματα και περιοχές εφαρμογών
- Τι είναι απόδοση; Το πόσο γρήγορα μπορούμε να κάνουμε μια "δουλειά"
  - Σε πολλά προγράμματα αυτό μεταφράζεται στον χρόνο εκτέλεσης του προγράμματος
- Πως μπορούμε να μετρήσουμε την απόδοση ενός συστήματος;
  - Απλά
    - t1 = get-timer-value
    - run program
    - t2 = get-timer-value
  - Ο χρόνος εκτέλεσης είναι το t2-t1 και όσο τον μειώνουμε για το ίδιο input το πρόγραμμά μας γίνεται αποδοτικότερο
  - Άρα είναι σχετικά εύκολο (επί της αρχής) να πούμε αν ένα πρόγραμμα είναι πιο γρήγορο από ένα άλλο

Πως βελτιώνουμε την απόδοση ενός συστήματος;

- Για να μπορούμε να βελτιώσουμε κάτι (οτιδήποτε) πρέπει πάντοτε να μπορούμε να το μετρήσουμε
  - Η μέτρηση είναι στη βάση όλων των επιστημών
  - Χωρίς μέτρηση δεν μπορούμε να συγκρίνουμε
  - Χωρίς σύγκριση δεν μπορούμε να εκτιμήσουμε το καλό και το καλύτερο
  - Χωρίς εκτίμηση δεν μπορούμε να χτίσουμε πάνω σε καλές λύσεις και τεχνικές και να κάνουμε συγκεκριμένη πρόοδο
- Για να βελτιώσουμε την απόδοση ενός συστήματος πρέπει
  - 1ο βήμα: Πάντα να καταλαβαίνουμε που ξοδεύει τον περισσότερο χρόνο του
  - 2ο βήμα: Να βελτιώσουμε τα τμήματά του που συνεισφέρουν περισσότερο στην συνολική καθυστέρηση
- Δεν έχει νόημα να βελτιώνουμε τμήματα ενός προγράμματος που έχουν μικρό αντίκτυπο στην απόδοσή του, όσο πολύπλοκος ή απλοϊκός και να είναι ο σχεδιασμός τους
- Αυτό το εκφράζει με απλό τρόπο ένα εμπειρικός κανόνας: ο νόμος του Amdahl

Amdahl's Law

- Έστω ένα πρόγραμμα και ο αντίστοιχος χρόνος εκτέλεσής του

```
int main(void){
        f1();
        f2();
}
```

  - Έστω ότι η f1() παίρνει 95sec και η f2() παίρνει 5sec

0 sec <----------f1()----------> 95 sec <----f2()----> 100 sec

  - Έστω ότι βρούμε έναν εξαιρετικό και ιδιοφυέστατο τρόπο να βελτιώσουμε την f2() κατά 1000 φορές, οπότε πρακτικά η f2() θα εκτελείται σε μηδενικό (πολύ μικρό) χρόνο
  - Πόσο πιο γρήγορο μπορεί να γίνει το σύστημά μας;
  - Το πολύ να κερδίσουμε 5%, άρα όσο και να προσπαθήσουμε δεν θα δούμε ποτέ μια σημαντική διαφορά στον συνολικό χρόνο εκτέλεσης
- **Amdahl's Law (παράφραση): Ασχολούμαστε πάντοτε (ξοδεύουμε τον χρόνο μας και την ενέργειά μας) στα μέρη ενός συστήματος που είναι πιο αργά και επηρεάζουν την συνολική απόδοση του συστήματος**
  - Στην παραπάνω περίπτωση, ξεκάθαρα πρέπει να αφιερώσουμε την προσπάθειά μας στην f1()
- Σημείωση: Όσο απλός και αν ακούγεται αυτός ο κανόνας πάρα πολλές φορές παρασυρόμαστε και ασχολούμαστε με τα τμήματα ενός συστήματος που καταλαβαίνουμε περισσότερο και όχι με αυτά που είναι τα πιο αργά/βαριά
- Αυτό αποτελεί μεγάλο πρόβλημα, γιατί εν τέλη, μετά από πολλή προσπάθεια και χρόνο, συνειδητοποιούμε ότι τα τμήματα που βελτιώνουμε δεν οδηγούν σε αισθητή συνολική βελτίωση του συστήματος
- Άρα: έχουμε πάντα το νόμο του Amdahl κατά νου και προσπαθούμε να καταλάβουμε ποια τμήματα ενός προγράμματος είναι τα "βαρύτερα"

Πως βρίσκουμε που ξοδεύει τον χρόνο του ένα σύστημα;

- Είναι αυτονόητο ή εύκολο να βρούμε που ξοδεύει ένα σύστημα τον χρόνο του;
  - **Συνήθως όχι**
- Πως μπορούμε να βρίσκουμε που πηγαίνει ο χρόνος;
- Ας συνεχίσουμε να χρησιμοποιούμε την απλή (και πραγματική) μέθοδο με τους timers που αναφέραμε

```
int main(void){
        t1 = get_timer();
        f1();
        t2 = get_timer();
        f2();
        t3 = get_timer();
        printf("f1=%d sec, f2=%d sec", t2-t1, t3-t2);
        return;
}
```

  - Η f1() παίρνει t2-t1 χρόνο και η f2() παίρνει t3-t2 χρόνο
- Τι γίνεται καθώς θέλουμε να μετρήσουμε σε πιο μεγάλη λεπτομέρεια τον χρόνο κάθε τμήματος ενός προγράμματος που χρησιμοποιεί nested συναρτήσεις, βιβλιοθήκες, κ.ο.κ.;
  - Θα πρέπει να προσθέτουμε αρκετό νέο κώδικα για να μετρούμε κάθε τμήμα του κώδικα χωριστά και στη συνέχεια να αποδίδουμε τον χρόνο στο σωστό υποσύστημα
- Ακόμη χειρότερα: Τι θα γίνει αν η f1() καλεί την f2();
  - Αν μετρήσουμε την f1() συνολικά, θα μετρήσουμε και μέρος της f2()
- Αυτές οι δυσκολίες μας κάνουμε να σκεφτούμε αν υπάρχει άλλος τρόπος να μετρήσουμε που πηγαίνει ο χρόνος του προγράμματος ➔ **Sample profiling**
  - Η μέθοδος με τους timers εξακολουθεί να είναι σημαντική και χρήσιμη και μάλιστα μπορεί να μας βοηθήσει σε θέματα που δεν δουλεύουν άλλες μέθοδοι (π.χ. στον υπολογισμό του latency/καθυστέρησης μηνυμάτων)
  - Ωστόσο απαιτεί προσπάθεια, προσοχή και λεπτομερή γνώση του συστήματος με το οποίο ασχολούμαστε

Sample Profiling

- Βασική ιδέα
  - Κατά την εκτέλεση του προγράμματος **σε περιοδικά διαστήματα**
  - Κοιτάζουμε **ποιο** υποσύστημα του προγράμματος εκτελείται
  - Στο τέλος της εκτέλεσης κοιτάζουμε πόσες **φορές** εκτελέστηκε το κάθε υποσύστημα
  - Όποιο υποσύστημα εκτελέστηκε τις **πιο πολλές φορές**, αυτό είναι και το πιο ”βαρύ” (και το δεύτερο πιο βαρύ, κ.ο.κ.)
- Υλοποίηση
  - Περιοδικά, κατά την εκτέλεση του προγράμματος, εκτελείται ένας interrupt handler
  - Ο handler κοιτάζει τον program counter (PC)
  - Συμβουλεύεται τον symbol table του προγράμματος και βρίσκει σε ποια συνάρτηση ανήκει ο PC
  - Θεωρεί ότι αυτή η συνάρτηση εκτελέστηκε +1 φορά
  - Στο τέλος της εκτέλεσης του προγράμματος τυπώνει τους counters για όλες τις συναρτήσεις
  - Έτσι έχουμε μια εκτίμηση του πόσες φορές N_f εκτελέστηκε η κάθε συνάρτηση f
- Μετατροπή αριθμού δειγμάτων σε χρόνο
  - Μπορούμε να κάνουμε την υπόθεση ότι το ποσοστό των δειγμάτων της κάθε συνάρτησης αντιστοιχεί στον χρόνο που εκτελείται η κάθε συνάρτηση
  - Αυτό αποτελεί ”εκτίμηση” μια και ο αριθμός των δειγμάτων δεν αντιστοιχεί πραγματικά σε χρόνο
  - Με την υπόθεση αυτή μπορούμε να κάνουμε τον εξής υπολογισμός
  - Ο χρόνος T_f μιας συνάρτησης f είναι το ποσοστό του συνολικού χρόνου του προγράμματος με βάση το ποσοστό δειγμάτων N_f της f
  - Άρα αν ο συνολικός χρόνος εκτέλεσης του προγράμματος είναι T, τότε για κάθε συνάρτηση f

$$T_f = (N_f / N) * T$$

  - Έτσι μπορούμε να υπολογίσουμε τον χρόνο που παίρνει κάθε συνάρτηση κατά την εκτέλεση ενός προγράμματος

Χρήση του Sample Profiling

- Το εντυπωσιακό με το sample profiling είναι ότι είναι γενικό, μπορεί να εφαρμοστεί σε όλα σχεδόν τα συστήματα, και μας δίνει μια γρήγορη και εύκολη εκτίμηση το που πηγαίνει ο χρόνος του συστήματος
- Χρειάζεται να έχουμε έναν interrupt handler που είναι ίδιος για όλα τα προγράμματα
- Τον handler τον προσθέτει ο compiler στο πρόγραμμά μας όταν ορίζουμε την παράμετρο –p
- Στο τέλος της εκτέλεσης το αρχείο <name>.gprof είναι ένα text αρχείο που περιέχει τις τιμές των counters με τα samples κάθε συνάρτησης και την αναγωγή σε χρόνο
- Άρα κάνοντας απλά

```
$ gcc –ansi –pedantic –Wall –pg main.c
$ a.out
$ gprof a.out
```

#samples	%samples	%cumulative samples	function
38	48.7%	48.7%	f1
12	15.4%	64.1%	f2
9	11.5%	75.6%	f3
9	11.5%	87.2%	main
3	3.8%	91.0%	f4
3	3.8%	94.9%	printf
2	2.6%	97.4%	f5
1	1.3%	98.7%	f6
1	1.3%	100.0%	f7

- Στο τέλος της εκτέλεσης θα έχουμε την κατανομή του χρόνου του προγράμματός μας στις διάφορες συναρτήσεις
- Δοκιμάστε το στις ασκήσεις σας μέχρι τώρα
- Ερωτήσεις
  - Τι γίνεται αν τα samples είναι πολύ συχνά;
  - Τι γίνεται αν τα samples είναι πολύ αραιά;

Πιθανά optimizations στο πρόγραμμά μας

- (1) Καλύτερο αλγόριθμο
  - Έστω ότι ψάχνουμε ένα συγκεκριμένο value σε ένα block μνήμης που περιέχει πολλά values
  - Αν κρατούμε τα values διατεταγμένα μέσα στο block, θα μπορούμε να κάνουμε κάποιας μορφής binary search που θα έχει σημαντικό αντίκτυπο
- (2) Καλύτερο σχεδιασμό για τον ίδιο αλγόριθμο
  - Το να ψάχνουμε για ένα value με binary search μπορεί να γίνει πάνω από ένα διατεταγμένο array ή με κάποιας μορφής tree index
  - Το διατεταγμένο array έχει πλεονεκτήματα όταν όλα τα values έχουν το ίδιο (και μικρό) μέγεθος
- (3) Καλύτερο κώδικα για τον ίδιο σχεδιασμό

Code Optimizations

- Αντικατάσταση κοινών υποεκφράσεων με το αποτέλεσμα

```
x = sqrt(a*a + b*b) + (sqrt(a*a + b*b)>0 ? ... : ...);
vs.
r = sqrt(a*a + b*b);
x = r + (r>0?...:...);
```

- ή

```
for(i=0;i<limits[username];i++){
     ...
}
vs.
n = limits[username]
for(i=0;i<n;i++){
     ...
}
```

- Αντικατάσταση ακριβών εκφράσεων με πιο φθηνές (strength reduction)

```
if (sqrt(a*a + b*b) < sqrt(c*c + d*d)) ...
vs.
if ( (a*a + b*b) < (c*c + d*d) ) ...
```

- Αφαιρούμε μικρά loops

```
for(i=0;i<3;i++){
    a[i] = b[i] + c[i];
}
vs.
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

Περισσότερα Code Optimizations

- Χρησιμοποιούμε μια cache για ακριβούς υπολογισμούς

```
for(i=0; i<A_SIZE; i++){
    x = A[i];
    r=sqrt(x);
}
vs.
for(i=0; i<A_SIZE; i++){
    x = A[i];
    if (! (r = lookup_sqrt_cache(x))) {
            r=sqrt(x);
            insert_sqrt_cache(x,r);
    }
}
```

- Γράφουμε έναν δικό μας allocator για τις συγκεκριμένες ανάγκες του προγράμματός μας
- Χρησιμοποιούμε προσεγγιστικές τιμές, πχ integers αντί για floats, εφόσον είναι εφικτό
- Αναδιοργανώνουμε τον κώδικά μας ώστε να έχει καλύτερη συμπεριφορά με τις caches και την μνήμη της CPU
- Χρησιμοποιούμε μια άλλη γλώσσα (πιθανώς πιο χαμηλού επιπέδου) που ταιριάζει καλύτερα στο σύστημά μας
- Αρκετά από αυτά τα optimizations τα κάνει πλέον ο compiler όταν ορίζουμε ένα υψηλότερο επίπεδο optimization με την παράμετρο –O[0,1,2,3]
  - gcc –Wall -ansi –pedantic –O[0,1,2,3] main.c
  - Το –O3 παράγει πλέον πολύ βελτιστοποιημένο κώδικα που ακόμη και ένας έμπειρος σχεδιαστής θα δυσκολευόταν να γράψει

What if…

- Έστω ότι κάναμε ότι μπορούσαμε να σκεφτούμε και το σύστημά μας εξακολουθεί να είναι αργό για την δουλειά που το θέλουμε
  - Αυτό συμβαίνει συχνά στις μέρες μας…
- Τι γίνεται τώρα; ➔ **Παραλληλισμός**
- Με κάποιο τρόπο το πρόγραμμά μας να μπορεί να τρέχει σε πολλούς επεξεργαστές σε ένα μικρό ή μεγάλο σύστημα
  - (1) ”Χαλάει” το βασικό assumption ότι ΟΛΕΣ οι εντολές του προγράμματός μας ΠΑΝΤΟΤΕ εκτελούνται η μία μετά την άλλη (**sequential programs**)
  - (2) Χρειαζόμαστε ένα μηχανισμό για να μπορεί ένα πρόγραμμα να χρησιμοποιεί πολλά cores ➔ **threads**

Reading

- Bryant Ch. 5.14
- Bartlett Ch.12