



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Ακ. έτος 2009-2010, 8ο εξάμηνο, Σχολή ΗΜ&ΜΥ

4η ΕΡΓΑΣΙΑ

Τελική Ημερομηνία Παράδοσης: 18 Ιουλίου 2010 (δεν θα δοθεί παράταση)

1. Εισαγωγή

Αντικείμενο της άσκησης αυτής είναι η μελέτη της επίδρασης που έχουν στην απόδοση των προγραμμάτων διάφορες τεχνικές βελτιστοποίησης κώδικα που στοχεύουν στην αξιοποίηση της cache.

Για τους σκοπούς της άσκησης θα χρησιμοποιήσετε τον προσομοιωτή Simics με τρόπο παρόμοιο με την 3^η άσκηση. Ο κώδικας που θα αξιολογήσετε είναι ο πολλαπλασιασμός δύο τετραγωνικών πινάκων A και B, τις διάφορες εκδόσεις του οποίου θα πρέπει να γράψετε σε γλώσσα C και να τις μεταγλωττίσετε ώστε να μπορούν να προσομοιωθούν στον Simics.

2. Περιβάλλον Προσομοίωσης

2.1 Ανάπτυξη και μεταγλώττιση κώδικα

Καλείστε να αναπτύξετε κώδικα σε γλώσσα C με βάση διαφορετικές τεχνικές βελτιστοποίησης που θα αναφερθούν παρακάτω. Καθώς για την προσομοίωση θα χρησιμοποιηθεί ένα Linux-x86 target (tango), η μεταγλώττιση του κώδικά σας θα πραγματοποιείται σε ένα host x86 μηχάνημα χρησιμοποιώντας τον gcc. Για όλες τις εκδόσεις θα χρησιμοποιήσετε το **-O1** optimization flag του gcc.

```
host$ gcc -O1 -o name_executable source_file.c
```

ΠΡΟΣΟΧΗ: Το target μηχάνημα είναι 32-bit. Επομένως, σε περίπτωση που το host μηχάνημα στο οποίο πραγματοποιείται η μεταγλώττιση είναι 64-bit, θα πρέπει να χρησιμοποιήσετε το **-m32** flag του gcc, προκειμένου να παραχθεί εκτελέσιμο για 32-bit μηχάνημα.

2.2 Προσομοίωση

Η προσομοίωση θα εκτελεστεί σε ένα tango μηχάνημα. Μπορείτε να χρησιμοποιήσετε κάποιο checkpoint που έχετε αποθηκεύσει από την 3^η άσκηση ή να δημιουργήσετε κάποιο καινούριο ακολουθώντας τις οδηγίες εκείνης της άσκησης. Στο checkpoint αυτό θα μεταφέρετε τα εκτελέσιμα που θα αναπτύξετε χρησιμοποιώντας τον τρόπο που επίσης έχει περιγραφεί στην 3^η άσκηση (mount

/host κτλ).

Όπως φαίνεται και στον κώδικα του Παραρτήματος Α, οι εφαρμογές που θα αναπτύξετε χρησιμοποιούν **magic breakpoints**, τα οποία σηματοδοτούν την αρχή και το τέλος της περιοχής που μας ενδιαφέρει. Πιο συγκεκριμένα, η μεθοδολογία που θα ακολουθήσετε είναι η εξής :

1. Ανάπτυξη και μεταγλώττιση του κώδικα στο host μηχανήμα
2. Εκκίνηση του simics σε –stall mode και φόρτωση του κατάλληλου checkpoint
3. Ρύθμιση του simics
 - a. simics> magic-break-enable
 - b. simics> dstc-disable
4. Εκτέλεση του εκτελέσιμου στην κονσόλα του target.
5. Στο πρώτο σημείο διακοπής της εκτέλεσης, φορτώνετε την ιεραρχία της μνήμης, η οποία δίνεται στο Παράρτημα Β και συνεχίζετε την εκτέλεση (αφού λάβετε τα αναγκαία στατιστικά).
 - a. simics> run-command-file cache-hierarchy.simics
 - b. simics> c
6. Το δεύτερο σημείο διακοπής της εκτέλεσης σηματοδοτεί το τέλος της προσομοίωσης. Επομένως συγκεντρώνετε τα στατιστικά που μας ενδιαφέρουν και προχωράτε στην επόμενη περίπτωση.

2.3 Ιεραρχία μνήμης

Η ιεραρχία μνήμης δίνεται στο Παράρτημα Β. Όπως μπορείτε να παρατηρήσετε τα penalties των προσβάσεων στη μνήμη έχουν οριστεί ίσα με 0. Η επιλογή αυτή έγινε προκειμένου να μειωθεί ο απαιτούμενος χρόνος προσομοίωσης.

Επίσης, για την συγκεκριμένη άσκηση δε μας ενδιαφέρει η instruction cache, για αυτό και δεν αλλάζουμε το instruction-fetch-mode του simics ούτε απενεργοποιούμε το istc.

2.4 Μοντέλο απόδοσης

Καθώς οι caches έχουν μηδενικούς χρόνους πρόσβασης, ο αριθμός των κύκλων που δίνει ο simics για την εκτέλεση της περιοχής μας ενδιαφέρει δεν είναι σωστός. Για αυτό τον λόγο απαιτείται ένα μοντέλο απόδοσης, το οποίο θα προσεγγίζει με μεγαλύτερη ακρίβεια τον πραγματικό αριθμό απαιτούμενων κύκλων.

Το μοντέλο του simics για τις x86 αρχιτεκτονικές είναι ένας in-order επεξεργαστής με IPC=1. Εμείς θεωρούμε ότι δεν υπάρχουν instruction misses και ότι οι εντολές πρόσβασης στην data cache δεν προκαλούν καθυστέρηση εφόσον είναι hits. Κάθε miss στην L1 στοιχίζει 10 κύκλους και κάθε miss στη L2 200 κύκλους αντίστοιχα.

Επομένως, ο συνολικός αριθμός των κύκλων μπορεί να προσεγγιστεί ως εξής :

$$\text{Cycles} = \text{Instructions} + \text{L1_misses} * \text{L1_penalty} + \text{L2_misses} * \text{L2_penalty}$$

3. Τεχνικές Βελτιστοποίησης

3.1 Αρχική έκδοση

Αρχικά, σας δίνεται (Παράρτημα Α) μια απλοϊκή, μη-βελτιστοποιημένη έκδοση του αλγορίθμου πολλαπλασιασμού τετραγωνικών πινάκων, όπως αυτή που παρουσιάζεται στη συνέχεια:

```
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        for(k=0; k<N; k++)
            C[i][j] += A[i][k]*B[k][j];
```

Προσομοιώστε την παραπάνω απλοϊκή έκδοση του αλγορίθμου, θεωρώντας πίνακες διάστασης 256x256. Καταγράψτε τον απαιτούμενο χρόνο εκτέλεσης (αριθμός κύκλων) καθώς και τα miss rates στην L1 και στην L2.

3.2 Loop interchange

Ο αρχικός αλγόριθμος δεν είναι βελτιστοποιημένος ως προς την χωρική τοπικότητα των αναφορών και συνεπώς δεν κάνει την καλύτερη δυνατή αξιοποίηση της cache. Για αυτό το λόγο, καλείστε να εφαρμόσετε την τεχνική της *αναδιάταξης βρόχων* προκειμένου να πετύχετε καλύτερη τοπικότητα στην cache, η οποία ευελπιστείτε να οδηγήσει τελικά και σε καλύτερους χρόνους εκτέλεσης.

1. Δοκιμάστε όλες τις διαφορετικές αναδιατάξεις που μπορείτε να κάνετε στον αρχικό κώδικα. Καταγράψτε τη συμπεριφορά της κάθε μιας, ως προς το χρόνο εκτέλεσης και τα miss rates στις L1D και L2 caches.
2. Υπάρχει κάποια συσχέτιση ανάμεσα στους παρατηρούμενους χρόνους εκτέλεσης για τις διάφορες εκδόσεις και τα αντίστοιχα miss rates που μετρήσατε; Σχολιάστε σχετικά.
3. Πώς αλλάζει η κάθε αναδιάταξη την απόδοση του απλοϊκού αλγορίθμου; Πώς εξηγείται αυτό σε σχέση με τα διαφορετικά access patterns που συνεπάγεται η κάθε αναδιάταξη, και την τοπικότητα που επιτυγχάνει θεωρητικά το κάθε access pattern;
4. Τι speedup δίνει η καλύτερη αναδιάταξη σε σχέση με την απλοϊκή έκδοση;

3.3 Cache blocking

Βασικός στόχος της τεχνικής αυτής είναι η βελτίωση της χρονικής τοπικότητας των αναφορών. Η γενική ιδέα του *cache blocking* έγκειται στον διαχωρισμό του χώρου επαναλήψεων ενός loop (loop iteration space) σε μικρότερους υποχώρους, έτσι ώστε το σύνολο δεδομένων (working set) που επεξεργάζεται ο κάθε υποχώρος να χωρά σε κάποιο επίπεδο κρυφής μνήμης, και να μπορεί συνεπώς να επαναχρησιμοποιηθεί εκεί στο μέγιστο δυνατό βαθμό, προτού εκτοπιστεί.

Στην περίπτωση που έχουμε να κάνουμε με πίνακες, ο διαχωρισμός του χώρου επαναλήψεων ενός ή περισσότερων loops οδηγεί στο διαχωρισμό των πινάκων σε μικρότερους υποπίνακες (ή blocks). Έτσι μπορούμε να εφαρμόσουμε τον αρχικό αλγόριθμο διαδοχικά πάνω στους επιμέρους υποπίνακες, και αν αυτοί συναθροιστικά χωράνε σε κάποιο επίπεδο κρυφής μνήμης, τότε μπορούμε να επιτύχουμε υψηλότερα επίπεδα επαναχρησιμοποίησης των στοιχείων τους και των στοιχείων του προβλήματος συνολικά.

1. Χρησιμοποιήστε τον κώδικα του πολλαπλασιασμού πινάκων που καταλήξατε στο 3.2 και υλοποιήστε μια έκδοση blocking, όπου θα διαχωρίσετε τον χώρο επαναλήψεων των **τριών**

loops.

2. Προσομοιώστε την έκδοση που υλοποιήσατε για διαστάσεις τετραγωνικών blocks από 8 έως 128 με βήμα 8. Παρουσιάστε σε διαγράμματα τη μεταβολή του χρόνου εκτέλεσης, καθώς και των miss rates για τις L1D και L2 caches, σε σχέση με το μέγεθος του block.
3. Ποιο είναι το overhead της εφαρμογής του blocking; Παρουσιάστε σε ένα διάγραμμα, για όλα τα μεγέθη blocks, τη μεταβολή του συνολικού αριθμού εντολών της blocked έκδοσης σε σχέση με τον αριθμό εντολών της αρχικής interchanged. Χρησιμοποιήστε το συνολικό αριθμό εντολών που εκτελούνται στην περιοχή ενδιαφέροντος, δηλαδή το άθροισμα user και supervisor instructions.
4. Πώς εξηγείται η μεταβολή των miss rates καθώς μεταβάλλεται το μέγεθος του block, για τα δεδομένα μεγέθη των L1D και L2 caches που έχουμε θεωρήσει; Αποτυπώνεται σε μεταβολή στον χρόνο εκτέλεσης, και αν ναι πώς;
5. Τι speedup δίνει η cache-blocked έκδοση σε σχέση με την απλοϊκή έκδοση;

3.4 Υλοποίηση στο host μηχανήμα

Σε αυτό το κομμάτι της άσκησης θα πραγματοποιήσετε την αντίστοιχη μελέτη στο πραγματικό host μηχανήμα. Ο πυρήνας του Linux παρέχει πληροφορίες για τον επεξεργαστή και την ιεραρχία μνήμης του μηχανήματος σας. Πιο συγκεκριμένα, όσον αφορά τη μνήμη θα πρέπει να κοιτάξετε στο path :

```
host$ ls /sys/devices/system/cpu/cpu*/cache/index*
```

Στους φακέλους αυτούς βρίσκονται αρχεία που παρέχουν πληροφορίες για τις caches του κάθε cpu, όπως τύπο, μέγεθος, associativity κτλ. Περισσότερες πληροφορίες μπορείτε να βρείτε στο section 5.3 του <http://people.redhat.com/drepper/cpumemory.pdf> .

Στις προσομοιώσεις που εκτελέσατε παραπάνω, ο κάθε πίνακας είχε διπλάσιο μέγεθος από τη προσομοιωμένη L2 cache. Βρίσκοντας λοιπόν το μέγεθος της L2 cache του μηχανήματος σας μπορείτε να υπολογίσετε το μέγεθος του κάθε τετραγωνικού πίνακα. Για παράδειγμα αν η L2 cache έχει μέγεθος 6M, τότε ο κάθε πίνακας θα πρέπει να έχει μέγεθος περίπου 12M, το οποίο μας δίνει διάσταση 1773x1773 ή 1774x1774. Σε αυτή την περίπτωση λοιπόν επιλέγουμε μια διάσταση ίση με την πιο κοντινή δύναμη του 2, δηλαδή 2048x2048.

Το σημείο που πρέπει να προσέξετε σε σχέση με τον κώδικα που εκτελέσατε στις προσομοιώσεις αφορά τη χρήση των magic breakpoints. Προφανώς, στο host μηχανήμα δεν υπάρχει κάτι αντίστοιχο. Για αυτό το λόγο, θα πρέπει να αντικαταστήσετε τις κλήσεις των breakpoints με κλήσεις της `gettimeofday`. Με αυτό τον τρόπο, θα μπορέσετε να μετρήσετε την επίδοση του κομματιού του κώδικα που μας ενδιαφέρει.

1. Προσομοιώστε την έκδοση που υλοποιήσατε στο 3.3 για διαστάσεις τετραγωνικών blocks από 8 έως 512 για μεγέθη ίσα με δυνάμεις του 2 (8, 16, 32, ... , 512). Παρουσιάστε σε διαγράμματα τη μεταβολή του χρόνου εκτέλεσης σε σχέση με το μέγεθος του block.
2. Υπολογίστε όπως και στο 3.3 το speedup της cache-blocked έκδοσης.
3. Συγκρίνετε τα συμπεράσματα με αυτά που εξάγατε στο 3.3. Είναι όμοια; (**hint:** Στην προσομοίωση χρησιμοποιείτε ένα “απλό” σύστημα, το οποίο μοντελοποιεί έναν in-order cpu με $CPI=1$, χωρίς prefetching, branch prediction, κτλ. Αντίθετα το host μηχανήμα σας, περιλαμβάνει διάφορα optimizations. Ποια από αυτά θα μπορούσαν να επηρεάσουν την απόδοση του συγκεκριμένου κώδικα;)

4. Συνολική Εκτίμηση

Για όλες τις παραπάνω διαδοχικές βελτιστοποιήσεις, κάντε μια ποσοτική εκτίμηση της συνεισφοράς κάθε τεχνικής στην τελική απόδοση του αλγορίθμου, καθώς και μια σύντομη ποιοτική εκτίμηση της συνεισφοράς τους συνοψίζοντας τα βασικότερα συμπεράσματα των προηγούμενων ερωτημάτων.

Παραδοτέο της άσκησης θα είναι ένα ηλεκτρονικό κείμενο (pdf, doc ή odt) που θα περιέχει την αναφορά με τα διαγράμματα και τα συμπεράσματά σας, καθώς και τον κώδικα που υλοποιήσατε. Στο ηλεκτρονικό κείμενο να αναφέρετε στην αρχή τα στοιχεία σας (Όνομα, Επώνυμο, ΑΜ).

Η άσκηση θα παραδοθεί μόνο ηλεκτρονικά στην ιστοσελίδα:

<http://www.cslab.ece.ntua.gr/courses/advcomparch/submit>.

*Δουλέψτε ατομικά. Έχει ιδιαίτερη αξία για την κατανόηση του μαθήματος να κάνετε μόνοι σας την εργασία.
Μην προσπαθήσετε να την αντιγράψετε απλά από άλλους συμφοιτητές σας.*

ΠΑΡΑΡΤΗΜΑ Α

```
#include <stdio.h>
#include <stdlib.h>

#define __MAGIC_CASSERT(p) do {
    typedef int __check_magic_argument[(p) ? 1 : -1];
} while (0)
#define MAGIC(n) do {
    __MAGIC_CASSERT(! (n));
    __asm__ __volatile__ ("xchg %bx,%bx");
} while (0)
#define MAGIC_BREAKPOINT MAGIC(0)

inline int min(int a, int b){
    if(a<=b) return a;
    else return b;
}

void init_matrix(float **mat, int n) {
    unsigned int i,j;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            mat[i][j] = (float)(i+j);
}

int main(int argc, char **argv){
    float **A,**B,**C;
    int i,j,k;
    int N;

    N=atoi(argv[1]);

    A=(float**)malloc(N*sizeof(float*));
    for(i=0; i<N; i++)
        A[i]=(float*)malloc(N*sizeof(float));

    B=(float**)malloc(N*sizeof(float*));
    for(i=0; i<N; i++)
        B[i]=(float*)malloc(N*sizeof(float));

    C=(float**)malloc(N*sizeof(float*));
    for(i=0; i<N; i++)
        C[i]=(float*)malloc(N*sizeof(float));

    fprintf(stderr, "Initializing matrices...\n");
    init_matrix(A, N);
    init_matrix(B, N);
    init_matrix(C, N);

    MAGIC_BREAKPOINT;

    for(i=0; i<N; i++) {
        for(j=0; j<N; j++)
            for(k=0; k<N; k++)
                C[i][j] += A[i][k]*B[k][j];
    }

    MAGIC_BREAKPOINT;

    return 0;
}
```

ΠΑΡΑΡΤΗΜΑ Β

```
# Transaction staller for memory
@staller = pre_conf_object('staller', 'trans-staller')
@staller.stall_time = 0

# l2 cache: 128Kb Write-back
@l2c = pre_conf_object('l2c', 'g-cache')
@l2c.cpus = conf.cpu0
@l2c.config_line_number = 1024
@l2c.config_line_size = 128
@l2c.config_assoc = 4
@l2c.config_replacement_policy = 'lru'
@l2c.penalty_read = 0
@l2c.penalty_write = 0
@l2c.timing_model = staller

# instruction cache: 32Kb
@ic = pre_conf_object('ic', 'g-cache')
@ic.cpus = conf.cpu0
@ic.config_line_number = 512
@ic.config_line_size = 64
@ic.config_assoc = 2
@ic.config_replacement_policy = 'lru'
@ic.penalty_read = 0
@ic.penalty_write = 0
@ic.timing_model = l2c

# data cache: 32Kb Write-through
@dc = pre_conf_object('dc', 'g-cache')
@dc.cpus = conf.cpu0
@dc.config_line_number = 512
@dc.config_line_size = 64
@dc.config_assoc = 2
@dc.config_replacement_policy = 'lru'
@dc.penalty_read = 0
@dc.penalty_write = 0
@dc.timing_model = l2c

# transaction splitter for instruction cache
@ts_i = pre_conf_object('ts_i', 'trans-splitter')
@ts_i.cache = ic
@ts_i.timing_model = ic
@ts_i.next_cache_line_size = 64

# transaction splitter for data cache
@ts_d = pre_conf_object('ts_d', 'trans-splitter')
@ts_d.cache = dc
@ts_d.timing_model = dc
@ts_d.next_cache_line_size = 64

# instruction-data splitter
@id = pre_conf_object('id', 'id-splitter')
@id.ibranch = ts_i
@id.dbranch = ts_d

@SIM_add_configuration([staller, l2c, ic, dc, ts_i, ts_d, id], None)
@conf.phys_mem0.timing_model = conf.id
```