Προηγμένα Θέματα Αρχιτεκτονικής Υπολογιστών

Άσκηση 4η

Δοκιμάκης Βύρων Α.Μ: 03107127

Α.3.1)

Τα αποτελέσματα της προσομοίωσης του αρχικού κώδικα που μας δίνεται φαίνονται παρακάτω:

Metrics	ijk
L1 Data Miss Rate	2,44%
L2 Miss Rate	1,49%
Number Of Cycles	472095621

Α.3.2)

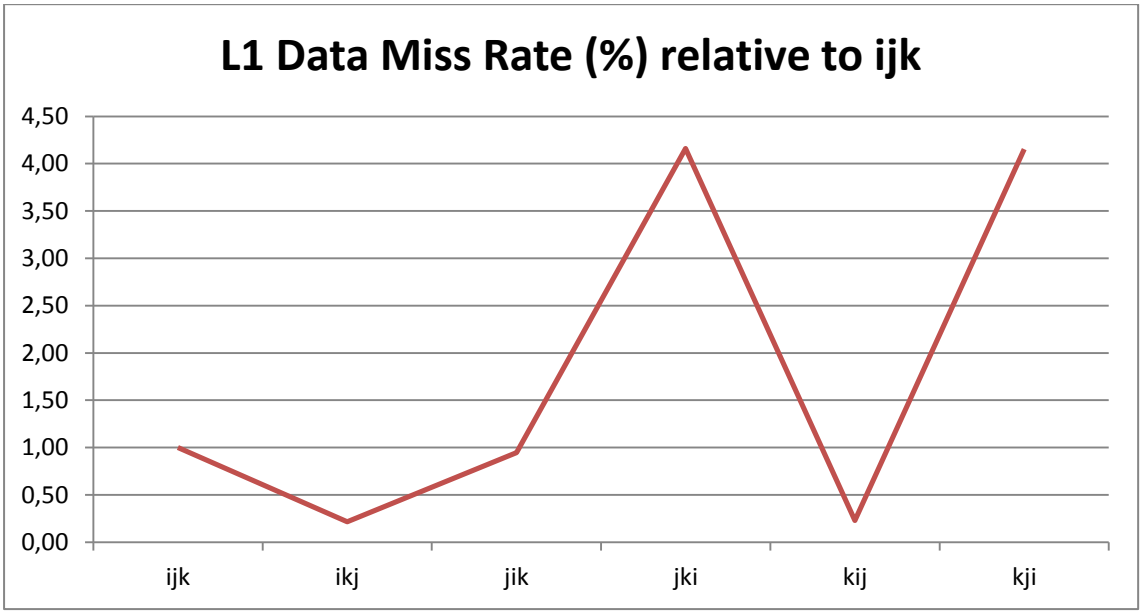
1. Τα αποτελέσματα των προσομοιώσεων όλων των δυνατών αναδιατάξεων των βρόχων φαίνονται παρακάτω:

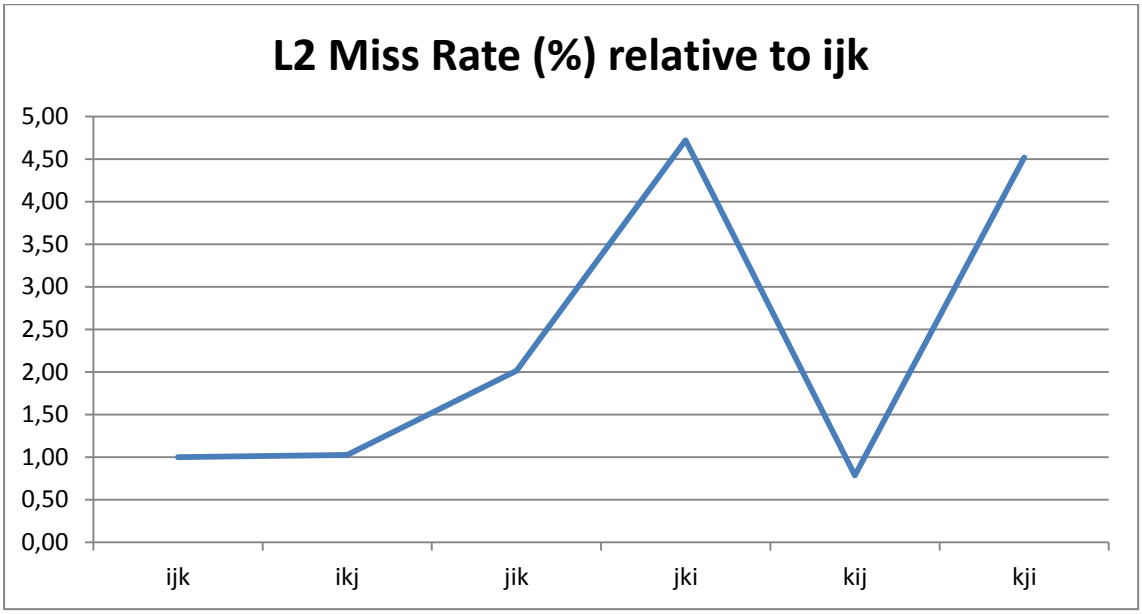
L1 Data Miss Rate (%)						
Interchange	ijk	ikj	jik	jki	kij	kji
miss rate	2,44	0,53	2,30	10,13	0,56	10,12

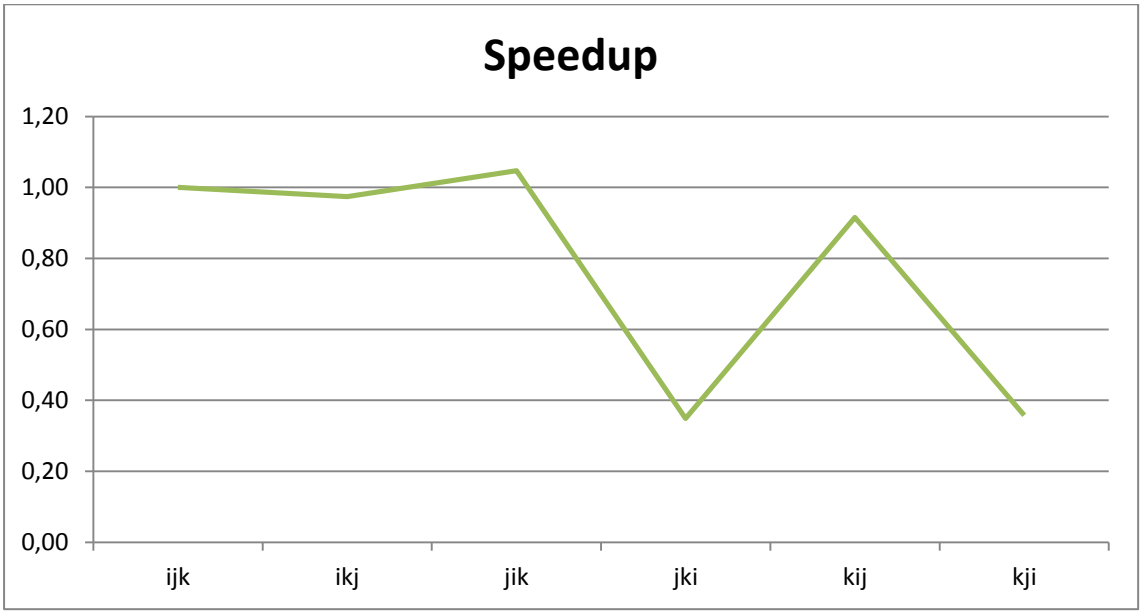
L2 Miss Rate (%)						
Interchange	ijk	ikj	jik	jki	kij	kji
miss rate	1,49	1,52	2,99	7,02	1,17	6,71

Number of Cycles						
Interchange	ijk	ikj	jik	jki	kij	kji
# of cycles	472095621	484515490	450693422	1354183131	515415001	1317482765

Για πιο καλή εποπτεία και πιο εύκολη εξαγωγή συμπερασμάτων, χρησιμοποιούμε τα παρακάτω διαγράμματα:







2. Όπως αναμέναμε και θεωρητικά, στις περιπτώσεις που τα miss rates τόσο της L1 όσο και της L2 cache αυξάνονται, η απόδοση (μετρούμενη με βάση τον χρόνο εκτέλεσης – αριθμό κύκλων) πέφτει . Η παραπάνω παρατήρηση επιβεβαιώνεται ξεκάθαρα από τις περιπτώσεις των αναδιατάξεων jki, kij, και kji.

   Παρατηρώντας λίγο καλύτερα όμως, διαπιστώνουμε ότι στην περίπτωση jik συμβαίνει το εξής παράδοξο: ενώ τα misses και των δύο caches αυξάνονται, έχουμε μια μικρή αύξηση της απόδοσης, δηλαδή μείωση του αριθμού των κύκλων. Το φαινόμενο αυτό μπορεί να εξηγηθεί αν σκεφτούμε ότι η μεταγλώττιση του κώδικα έγινε με το flag –O, που σημαίνει ότι ο compiler είχε την ευκαιρία να εφαρμόσει optimizations στον κώδικά μας.

   Εφόσον λοιπόν κάθε φορά μεταγλωττίζαμε διαφορετικό κώδικα (για τις διαφορετικές αναδιατάξεις), ο compiler είναι πολύ πιθανόν να έβγαζε διαφορετικό (optimized) κώδικα προς εκτέλεση, γεγονός που σημαίνει ότι μπορεί στην περίπτωση του jik οι εντολές assembly που εκτέλεσε ο επεξεργαστής μας να ήταν τόσο λίγες, που ακόμη και συνυπολογίζοντας τα αυξημένα misses, ο συνολικός αριθμός κύκλων να μην ξεπέρασε τον αριθμό κύκλων που χρειάστηκε η εκτέλεση της αρχικής διάταξης (ijk).

   Επίσης, είναι δυνατό να δούμε περίπτωση όπου εμφανίζονται λιγότερα misses αλλά μεγαλύτερος αριθμός κύκλων (kij), γεγονός που εξηγείται με βάση τα παραπάνω.

   Σε κάθε περίπτωση, αποφασίζουμε η απόδοση των προγραμμάτων μας να κρίνεται με βάση το χρόνο εκτέλεσής τους, συνεπώς διαλέγουμε την αναδιάταξη jik, ως βέλτιστη, γιατί αυτή εμφανίζει το μεγαλύτερο speedup.

3. Θεωρητική μελέτη συμπεριφοράς cache (υποθέτουμε ότι οι πίνακες αποθηκεύονται στη μνήμη κατά γραμμές):

   - ikj: Η αναδιάταξη αυτή θεωρητικά δίνει λιγότερα misses από την αρχική καθώς η προσπέλαση του πίνακα B γίνεται κατά γραμμή και όχι κατά στήλη (όπως στην ijk).

- jik: Η αναδιάταξη αυτή θεωρητικά δίνει περισσότερα misses από την αρχική, καθώς η προσπέλαση του πίνακα C γίνεται κατά στήλη και όχι κατά γραμμή (όπως στην ijk).

- jki: Η αναδιάταξη αυτή θεωρητικά δίνει πολύ περισσότερα misses από την αρχική, καθώς η προσπέλαση των πινάκων A και C γίνεται κατά στήλη και όχι κατά γραμμή (όπως στην ijk).

- kij: Η αναδιάταξη αυτή θεωρητικά πλησιάζει τα miss rates της αρχικής, καθώς ο πίνακας A προσπελάζεται κατά στήλη, όμως ο πίνακας B κατά γραμμή. Συνεπώς η διαφορά παρουσιάζουν στα πειραματικά δεδομένα οφείλεται σε άλλους παράγοντες (σειρά προσβάσεων στη μνήμη, optimizations του compiler κλπ).

- kji: Η αναδιάταξη αυτή θεωρητικά δίνει περισσότερα misses από την αρχική καθώς η προσπέλαση των πινάκων A και C γίνεται κατά στήλη ενώ μόνο του B κατά γραμμή.

4. Το μέγιστο speedup παρατηρείται στην αναδιάταξη jik και ισούται με 1.05

A.3.3)

1. Ο κώδικας που υλοποιεί το blocking και στους τρείς πίνακες φαίνεται παρακάτω:

```
for(jj=0; jj<N; jj=jj+BS)
        for(ii=0; ii<N; ii=ii+BS)
                for(kk=0; kk<N; kk=kk+BS)
                        for (j=jj; j<min(jj+BS,N);j=j+1)
                                for (i=ii; i<min(ii+BS,N); i=i+1)
                                        for (k=kk; k<min(kk+BS,N);k=k+1)
                                                C[i][j] += A[i][k]*B[k][j];
```

2. Τα αποτελέσματα των προσομοιώσεων για τα μεγέθη των blocks που ζητούνται φαίνονται παρακάτω

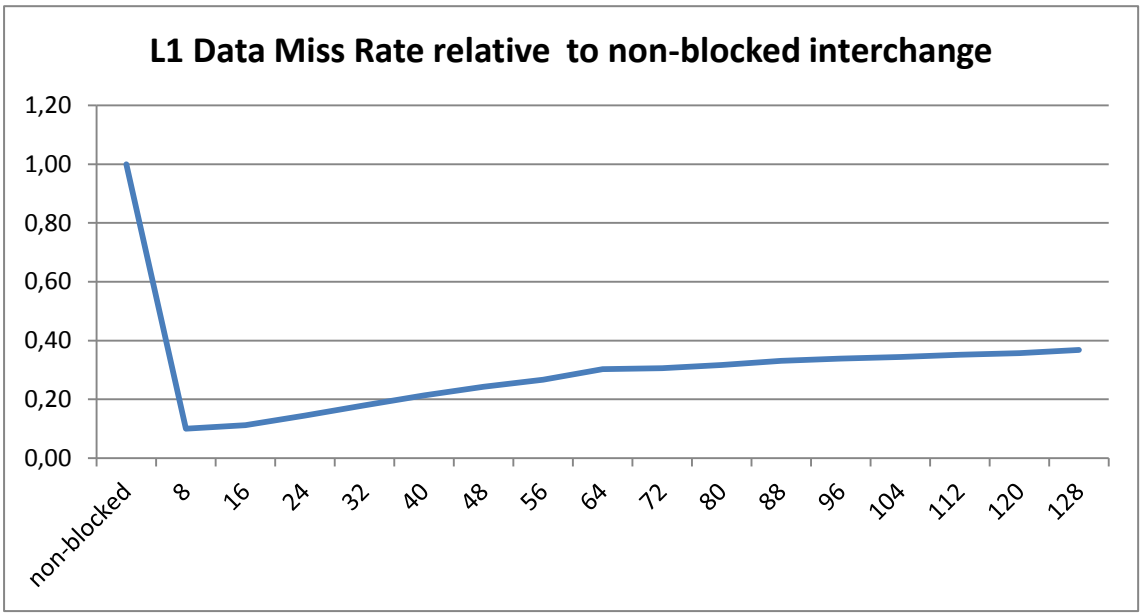
L1 Data Miss Rate (%)																
Block Size	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128
miss rate	0,23	0,26	0,33	0,41	0,49	0,56	0,62	0,70	0,70	0,73	0,76	0,78	0,79	0,81	0,82	0,85

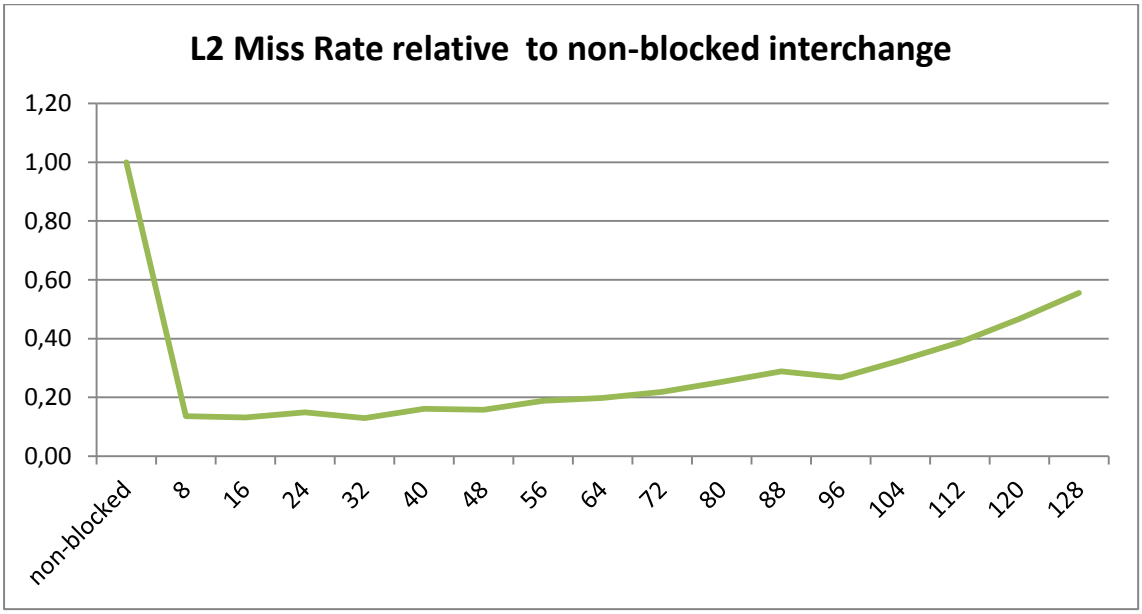
L2 Miss Rate (%)																
Block Size	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128
miss rate	0,41	0,39	0,45	0,39	0,48	0,47	0,56	0,59	0,66	0,75	0,86	0,80	0,97	1,16	1,40	1,66

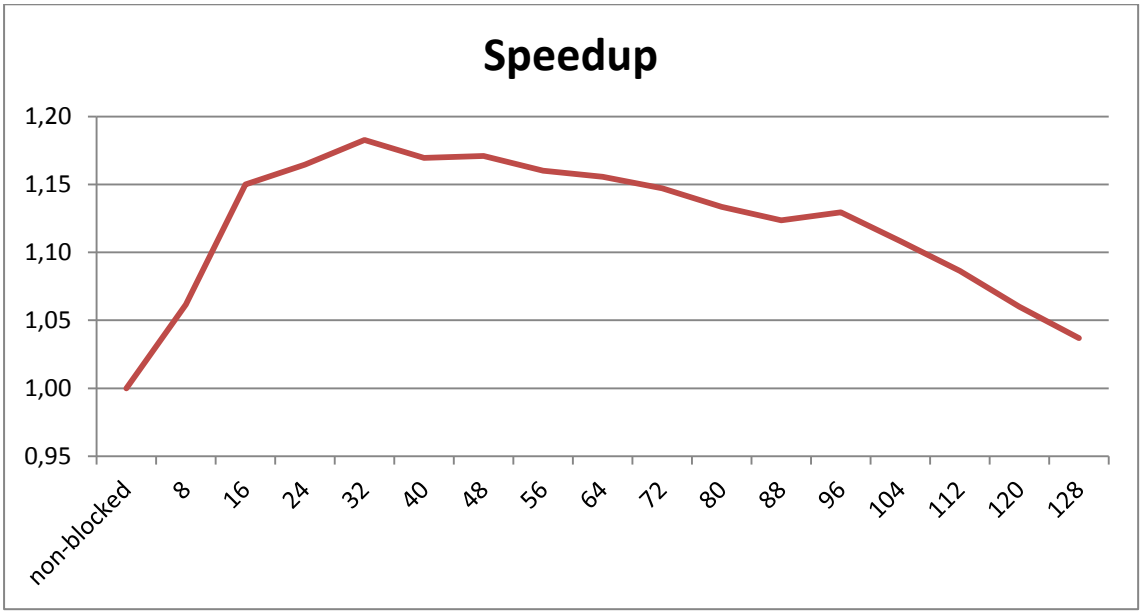
Number of Cycles																
Block Size	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128
# of cycles	42448 5772	39191 7862	38699 5645	38105 0635	38534 9042	38488 1475	38848 3275	39000 6234	39286 3030	39764 1674	40114 9156	39906 0026	40667 6767	41484 5527	42517 7527	43464 5938

Number of Instructions																
Block Size	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128
# of instructions	39406 1662	36655 6362	35870 1775	35412 5555	35263 1422	35116 0115	34968 1535	34822 5264	34822 3820	34822 5264	34678 0466	34678 0466	34678 1227	34678 1227	34678 1227	34534 8788

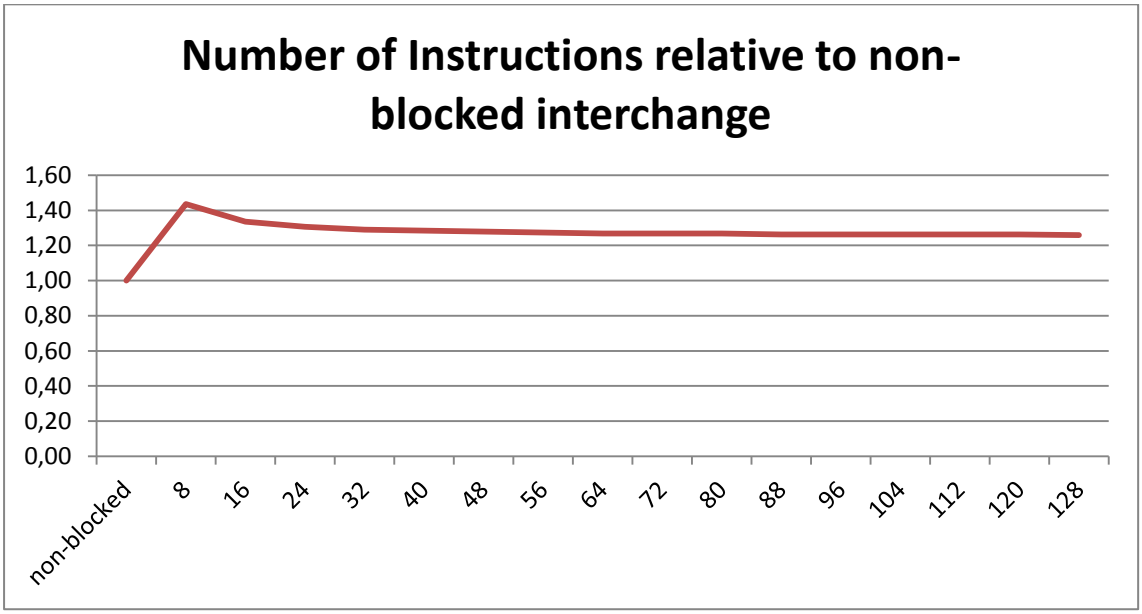
Για πιο καλή εποπτεία και πιο εύκολη εξαγωγή συμπερασμάτων, χρησιμοποιούμε τα παρακάτω διαγράμματα:







3. Η εφαρμογή του blocking στοχεύει στο να μειώσει τα misses, εκμεταλλευόμενη την προσπέλαση των πινάκων. Είναι όμως φανερό ότι προκύπτει ένα overhead, καθώς αυξάνεται σημαντικά ο αριθμός των εντολών του προγράμματος λόγω της ύπαρξης του διπλάσιου αριθμού επαναληπτικών βρόχων και παραπάνω μεταβλητών. Στο σχήμα που ακολουθεί φαίνεται η μεταβολή του συνολικού αριθμού εντολών των blocked εκδόσεων, σε σχέση με την non-blocked έκδοση της αναδιάταξης jik:



Από το παραπάνω σχήμα γίνεται φανερό ότι το blocking ¨κοστίζει¨σε αριθμό εντολών.

4. Εφόσον το μέγεθος του κάθε στοιχείου των πινάκων που πολλαπλασιάζουμε είναι 32bit (float), και η L2 Cache έχει line size ίσο με 128, ενώ η L1 Data Cache έχει line size ίσο με 64, καταλήγουμε στο συμπέρασμα ότι ο βέλτιστος blocking factor θα ήταν το 4 ή το 2 αντίστοιχα. Είναι λογικό λοιπόν τα misses αυτών των 2 caches να αυξάνονται, όσο αυξάνεται και το μέγεθος του block με το οποίο ασχολούμαστε κάθε φορά. Αυτό αποτυπώνεται και στο διάγραμμα του Speedup, όπου φαίνεται ότι η απόδοση μειώνεται όταν το μέγεθος του block περάσει μια συγκεκριμένη τιμή.

   Σημαντικό ρόλο στα παραπάνω αποτελέσματα παίζουν κι άλλοι παράγοντες, όπως το πώς είναι αποθηκευμένοι οι πίνακες στη μνήμη (και αν εκμεταλλεύονται το γεγονός ότι οι cache έχουν associativity μεγαλύτερο του 1) και το τι optimizations εφαρμόζει ο compiler σε κάθε περίπτωση.

5. Το μέγιστο speedup σε σχέση με την non-blocked έκδοση παρατηρείται για block size 32 και ισούται με με 1.18.

Α.3.4)

Παρατηρήσαμε ότι η τεχνική που εφαρμόσαμε στην αρχή (αναδιάταξη βρόχων) είχε τόσο θετικά όσο και αρνητικά αποτελέσματα, όσον αφορά την απόδοση, όμως καταφέραμε να βρούμε αναδιατάξεις που έδιναν καλύτερους χρόνους εκτέλεσης από τον αρχικό μας κώδικα. Σε μια περίπτωση καταφέραμε να μειώσουμε τα misses της L1 Data Cache σχεδόν στο 1/5 των misses του αρχικού μας κώδικα, όμως συνολικά, κρίνοντας από το χρόνο εκτέλεσης, η βελτίωση της απόδοσης του κώδικά μας δεν ξεπέρασε το 5%. Εδώ πρέπει ξανά να σημειώσουμε ότι σημαντικό ρόλο στη διαμόρφωση του χρόνου εκτέλεσης έπαιξαν και τα optimizations του compiler.

Στη συνέχεια εφαρμόσαμε την τεχνική του blocking, η οποία είχε ως αποτέλεσμα την επιτάχυνση του κώδικά μας ως και 18%, ενώ τα miss rates ήταν κι αυτά σημαντικά χαμηλότερα από αυτά που παρουσίασε ο non-blocked κώδικας.

Γενικά, καταλήγουμε στο συμπέρασμα ότι η τεχνική του blocking είναι πιο δραστική στην βελτίωση της απόδοσης του προγράμματός μας, καθώς είναι πιο παραμετροποιήσιμη (μέσω του blocking factor) και μπορούμε να τη φέρουμε στα μέτρα μας, ανάλογα με τα χαρακτηριστικά των μνημών μας, ενώ τεχνική της αναδιάταξης βρόχων είναι πιο εύκολα υλοποιήσιμη, με μηδενικό overhead, όμως όχι τόσο αποτελεσματική.

B.1)

Γνωρίζουμε ότι κάθε cache line, δηλαδή κάθε block της μνήμης, έχει μέγεθος 16bytes, που σημαίνει ότι το block offset της διεύθυνσής μας θα είναι ίσο με 4bits. Επίσης γνωρίζουμε ότι η caches μας έχουν 256 sets, άρα το index που θα χρειαστούμε για να μπορούμε να προσπελάσουμε όλα τα sets, είναι 8bits.

Εφόσον η διεύθυνσή μας έχει συνολικά 16bits, τα εναπομείναντα 4MSbits, θα αποτελούν το Tag του εκάστοτε block/cache line. Με βάση αυτά μπορούμε να προχωρήσουμε στην παρουσίαση των ζητούμενων καταστάσεων.

Σημείωση: με κόκκινο χρώμα θα φαίνονται οι αλλαγές στην κύρια μνήμη,
όταν αυτές καταχωρούνται για πρώτη φορά. Επίσης, τα flushes αναφέρονται πάντα, είτε αφορούν δεδομένα που έχουν ζητηθεί από άλλη cache, ή που είναι modified, είτε όχι.

1

Memory				
Location	0	4	8	C
0B5x	0000	0000	0000	0000
1B5x	0000	0000	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache						
Tag	Set	MESI State	Data			
0	B5	E	0000	0000	0000	0000

Processor 1 Cache						
Tag	Set	MESI State	Data			

2

Memory				
Location	0	4	8	C
0B5x	0000	0000	0000	0000
1B5x	0000	0000	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache						
Tag	Set	MESI State	Data			
0	B5	S	0000	0000	0000	0000

Processor 1 Cache						
Tag	Set	MESI State	Data			
0	B5	S	0000	0000	0000	0000

flush

3

Memory				
Location	0	4	8	C
0B5x	0000	0000	0000	0000
1B5x	0000	0000	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache

Tag	Set	MESI State	Data			
0	B5	I	-	-	-	-

flush

Processor 1 Cache

Tag	Set	MESI State	Data			
0	B5	M	0000	1111	0000	0000

4

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	0000	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache

Tag	Set	MESI State	Data			
0	B5	S	0000	1111	0000	0000

Processor 1 Cache

Tag	Set	MESI State	Data			
0	B5	S	0000	1111	0000	0000

flush

5

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	0000	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache

Tag	Set	MESI State	Data			
0	B5	M	2222	1111	0000	0000

Processor 1 Cache

Tag	Set	MESI State	Data			
0	B5	I	-	-	-	-

flush

6

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	0000	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache

Tag	Set	MESI State	Data			
0	B5	M	2222	1111	0000	0000

Processor 1 Cache

Tag	Set	MESI State	Data			
0	B5	I	-	-	-	-
1	B5	M	0000	3333	0000	0000

7

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	3333	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache						
Tag	Set	MESI State	Data			
0	B5	M	2222	1111	0000	0000
1	B5	M	0000	3333	0000	4444

Processor 1 Cache						
Tag	Set	MESI State	Data			
0	B5	I	-	-	-	-
1	B5	I	-	-	-	-

flush

8

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	3333	0000	0000
2B5x	0000	0000	0000	0000

Processor 0 Cache						
Tag	Set	MESI State	Data			
0	B5	M	2222	1111	0000	0000
1	B5	M	0000	3333	0000	4444

Processor 1 Cache						
Tag	Set	MESI State	Data			
0	B5	I	-	-	-	-
1	B5	I	-	-	-	-

9

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	3333	0000	0000
2B5x	0000	0000	0000	0000

Replace LRU block

Processor 0 Cache						
Tag	Set	MESI State	Data			
0	B5	M	2222	1111	0000	0000
1	B5	M	0000	3333	0000	4444

Processor 1 Cache						
Tag	Set	MESI State	Data			
2	B5	M	5555	0000	0000	0000
1	B5	I	-	-	-	-

10

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	3333	0000	4444
2B5x	5555	0000	0000	0000

Processor 0 Cache						
Tag	Set	MESI State	Data			
0	B5	M	2222	1111	0000	0000
2	B5	S	5555	0000	0000	0000

Processor 1 Cache						
Tag	Set	MESI State	Data			
2	B5	S	5555	0000	0000	0000
1	B5	I	-	-	-	-

flush

Replace LRU block

flush due to LRU replacement

11

Memory				
Location	0	4	8	C
0B5x	0000	1111	0000	0000
1B5x	0000	3333	0000	4444
2B5x	5555	0000	0000	0000

Processor 0 Cache						
Tag	Set	MESI State	Data			
0	B5	M	2222	1111	0000	0000
2	B5	M	5555	0000	0000	6666

Processor 1 Cache						
Tag	Set	MESI State	Data			
2	B5	I	-	-	-	-
1	B5	I	-	-	-	-

flush

B.2)

1.

Οι δυνατοί συνδυασμοί τελικών τιμών για τους καταχωρητές r1,r2,r3,r4 στο Relaxed Memory Model φαίνονται στον παρακάτω πίνακα:

r1	r2	r3	r4
1	0	0	0
1	0	0	2
1	0	1	0
1	0	1	2
1	3	0	0
1	3	0	2
1	3	1	0
1	3	1	2

2.

Μόνο ο παρακάτω συνδυασμός είναι δυνατό να εμφανιστεί αν υποθέσουμε ότι ισχύει το μοντέλο Sequential Consistency:

r1	r2	r3	r4
1	3	1	2

3.

Εισάγοντας τις εντολές memory barrier που φαίνονται παρακάτω, επιτυγχάνεται ταύτιση των δυνατών αποτελεσμάτων των RM και SC Models:

Processor 0	Processor 1
X=1	while (flag==0)
Y=2	**fence**
fence	r3=X
flag=1	r4=Y
while (flag==1)	Z=3
fence	**fence**
r1=X	flag=0
r2=Z	