ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Ενοποιημένη εφαρμογή μετασχηματισμών κώδικα και δεδομένων για την αντιμετώπιση καθυστερήσεων λόγω αστοχιών (misses) σε πολυεπίπεδες ιεραρχίες μνήμης πολυνηματικών αρχιτεκτονικών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χάρη Μ. Βώλου

Επιβλέπων: Νεκτάριος Κοζύρης
Επ. Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2005

Εθνικο Μετσοβιο Πολυτεχνειο
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Ενοποιημένη εφαρμογή μετασχηματισμών κώδικα και δεδομένων για την αντιμετώπιση καθυστερήσεων λόγω αστοχιών (misses) σε πολυεπίπεδες ιεραρχίες μνήμης πολυνηματικών αρχιτεκτονικών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χάρη Μ. Βώλου

Επιβλέπων: Νεκτάριος Κοζύρης
Επ. Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $11^{η}$ Ιουλίου 2005.

.....	.....	.....
Νεκτάριος Κοζύρης Επ. Καθήγητης ΕΜΠ	Τίμος Σελλής Καθηγητής ΕΜΠ	Παναγιώτης Τσανάκας Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2005.

.....

Χάρης Μ. Βώλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright ©Χάρης Μ. Βώλος, 2005
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προυπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Ένα από τα πιο σοβαρά προβλήματα των σημερινών υπολογιστικών συστημάτων είναι το χάσμα επίδοσης μεταξύ μονάδας επεξεργασίας και συστήματος μνήμης. Το πρόβλημα αυτό αντιμετωπίζεται με την ιεράρχηση του συστήματος μνήμης σε πολλαπλά επίπεδα (καταχωρητές, κρυφή μνήμη, εικονική μνήμη), όπου τα δεδομένα ενός επιπέδου αποτελούν υποσύολο των δεδομένων του αμέσως χαμηλώτερου επιπέδου. Αν και η εισαγωγή της ιεραρχίας λύνει σημαντικά το πρόβλημα, η επίδοση μπορεί να αυξηθεί ακόμη περισσότερο όταν τα ίδια τα προγράμματα σχεδιάζονται με στόχο την μέγιστη δυνατή αξιοποίηση της ιεραρχίας μνήμης.

Σκοπός της παρούσας διπλωματικής εργασίας είναι η αξιοποίηση των πολυεπίπεδων ιεραρχιών μνήμης των σύγχρονων υπολογιστικών συστημάτων μέσω της ενιαίας εφαρμογής μετασχηματισμών κώδικα και μετασχηματισμών διατάξεων δεδομένων, οι οποίοι αύξάνουν την τοπικότητα των αναφορών και μείωνουν επομένως τις αστοχίες στα διάφορα επίπεδα. Παρουσιάζουμε μεταξύ άλλων τον μετασχηματισμό πλακόστρωσης (tiling) ο οποίος αποτελεί τον πιο διαδεδομένο μετασχηματισμό κώδικα για την αναδιάταξη των επαναλήψεων σε φωλιασμένους βρόχους με στόχο την αύξηση της τοπικότητας, καθώς και τις διατάξεις ενοτήτων οι οποίες αναδιατάσσουν το χώρο δεδομένων, έτσι ώστε τα δεδομένα να αποθηκεύονται με τη σειρά που αυτά προσπελάνονται από τους tiled κώδικες. Για την δεικτοδότηση των ενοτήτων στηριζόμαστε στον αποδοτικό μηχανισμό δεικτοδότησης MBaLt.

Η μελέτη της αποτελεσματικότητας των μετασχηματισμών, καθώς και του μηχανισμού δεικτοδότηση ενοτήτων, γίνεται μέσω της υλοποίησης διαφόρων εκδοχών του μετροπρογράμματος 'Παραγοντοποίηση Cholesky'. Τα πραγματικά αποτελέσματα που λαμβάνουμε από την εκτέλεση των εκδοχών του μετροπρογράμματος σε δύο διαφορετικά υπολογιστικά συστήματα δείχουν ότι η επίδοση αυξάνεται σημαντικά με την ενιαία εφαρμογή της πλακόστρωσης και διατάξεων ενοτήτων με δεικτοδότηση MBaLt. Επιπρόσθετα, τα αναλυτικά αποτελέσματα που λαμβάνονται από την εξομείωση μέσω του εργαλείου SimpleScalar, επιβεβαιώνουν ότι η βελτίωση της επίδοσης οφείλεται στην ελάττωση των αστοχιών κρυφής μνήμης και τον περιορισμό της κατάχρησης TLB.

Τέλος, υλοποιούμε πολυνηματικές εκδοχές του μετροπρογράμματος με στόχο την μελέτη της επίδοσης της ενιαίας εφαρμογής των προαναφερομένων μετασχηματισμών σε ιεραρχίες μνήμης πολυνηματικών αρχιτεκτονικών τύπου SMT. Τα αποτελέσματα που προκύπτουν, αποκαλύπτουν την αδυναμία επίδοσης αυτών των αρχιτεκτονικών κατά την εκτέλεση παράλληλων εκδοχών βελτιστοποιημένων προγραμμάτων που παρουσιάζουν υψηλή παραλληλία σε επίπεδο εντολής.

Λέξεις Κλειδιά: Ιεραρχία Μνήμης, Κρυφή Μνήμη, Αστοχία, Τοπικότητα Αναφορών, Μετασχηματισμός Βρόχων και Δεδομένων, Μετασχηματισμός Πλακόστρωσης (tiling), Διατάξεις Ενοτήτων, Δεικτοδότηση Ενοτήτων MBaLt, Πολυνηματική Αρχιτεκτονική SMT, 'Παραγοντοποίηση Cholesky'

Abstract

One of the most important problems of modern computer systems is the increasing discrepancy between processor cycle times and main memory access times. One solution to this problem is the use of a multiple level hierarchical memory system (registers, cache memory, virtual memory), where the data of one level is a subset of the data of a lower level. The performance can be further increased, when programs are written in such a way to exploit the existence of a memory hierarchy.

The goal of this diploma thesis is the exploit of the memory hierarchy found in contemporary computer systems, through the unified application of code and data layout transformations. These transformations increase the locality of data references and hence decrease misses in different levels of the memory hierarchy. We present the tiling transformation, which is a widely used loop iteration reordering technique for improving locality of references, and also block data layouts which reorder the data space in the order that is swept by the tiled instruction stream. For the indexing of the blocks we use the MBaLt mechanism, which is an efficient and fast indexing method.

To study the effectiveness of such transformations and also the efficieny of MBaLt, we implement different versions of Cholesky factorization benchmark. Actual experimental results on two different computer systems, using these versions, illustrate that performance is greatly improved when combining tiled code with block data layouts and MBaLt indexing mechanism. Furthermore, analytical simulation results taken using the Simplescalar tool, verify that our enhanced performance is due to the considerable reduction of cache misses in all levels of memory hierarchy.

Finally, we implement some multithreaded versions of the benchmark to study the effectiveness of the above transformations in the memory hierarchy of simultaneous multithreading (SMT) architecture. The results show the weakness of this architecture to perform well when executing parallel versions of optimized programs with high level of instruction level parallelism.

Keyrords: Memory Hierarchy, Cache Memory, Miss, Locality of References, Loop And Data Layout Transformations, Tiling, Block Data Layouts, MBaLt Blocks Indexing, Simultaneous Multithreading (SMT), Cholesky Factorization

Ευχαριστίες

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου, υπό την επίβλεψη του Επίκουρου Καθηγητή Νεκτάριου Κοζύρη.

Θα ήθελα κατ' αρχήν να ευχαριστήσω τον καθηγητή μου Νεκτάριο Κοζύρη, τόσο για την εποπτεία του κατά την εκπόνηση της εργασίας μου, όσο και για τη συμβολή του στην διαμόρφωση μου ως μηχανικού μέσα από τις διδασκαλίες του και τη γενικότερη στάση του.

Τις ευχαριστίες μου θα ήθελα να εκφράσω επίσης σε όλα τα μέλη του εργαστηρίου και ιδιαίτερα στην Υποψήφια Διδάκτορα Ευαγγελία Αθανασάκη για τη συνεχή καθοδήγηση και ενθάρρυνση που μου προσέφερε προς ολοκλήρωση της διπλωματικής μου εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω όλους όσους βρίσκονται στο οικογενειακό και φιλικό μου περιβάλλον και κυρίως τους γονείς μου, η κάθε είδους υποστήριξη των οποίων συνετέλεσε τόσο στην περάτωση της διπλωματικής μου εργασίας όσο και στην επιτυχή ολοκλήρωση των προπτυχιακών μου σπουδών.

Περιεχόμενα

0.1 Γενικά 13
0.2 Σκοπός 15
0.3 Οργάνωση-Αντικείμενο της εργασίας 15

1 Αρχιτεκτονική Συστήματος Μνήμης **17**
1.1 Βασικές Έννοιες Κρυφής Μνήμης 17
1.1.1 Οργάνωση Κρυφής Μνήμης 18
1.1.2 Εύρεση Ενότητας μέσα στην Κρυφή Μνήμη 19
1.1.3 Αντικατάσταση Ενότητας στην περίπτωση αστοχίας Κρυφής Μνήμης . 20
1.1.4 Μηχανισμοί Εγγραφής Ενότητας 21
1.2 Βελτιστοποίηση Επίδοσης Κρυφής Μνήμης 22
1.2.1 Στρατηγικές Υλικού 22
1.2.2 Στρατηγικές Μεταγλωττιστή 23
1.3 Οργάνωση Εικονικής Μνήμης 25

2 Μετασχηματισμοί Βρόχων **27**
2.1 Βασικές Έννοιες 27
2.1.1 Φωλιασμένοι Βρόχοι 27
2.1.2 Χώρος Επαναλήψεων 28
2.1.3 Εξαρτήσεις Δεδομένων 29
2.2 Τοπικότητα και Επαναχρησιμοποίηση 32
2.2.1 Διάκριση Τοπικότητας και Επαναχρησιμοποίησης 32
2.2.2 Εύρεση Επαναχρησιμοποίησης 32
2.3 Unimodular Μετασχηματισμοί Βρόχων 33
2.4 Μη Γραμμικοί Μετασχηματισμοί Βρόχων 36
2.5 Μετασχηματισμός Tiling 38
2.5.1 Γενικά 38
2.5.2 Εφαρμόζοντας τον Μετασχηματισμό Tiling 40

3 Μετασχηματισμοί Δεδομένων **45**
3.1 Γενικά 45
3.2 Βασικές Έννοιες 46
3.3 Μετασχηματισμοί δεδομένων 47
3.3.1 Ορισμός 47
3.3.2 Εφαρμογή Μετασχηματισμού 47
3.3.3 Ορισμένες Βασικές Μορφές 50

3.3.4 Εγκυρότητα . . . 52
3.4 Ενοποίηση Μετασχηματισμών Βρόχων και Δεδομένων . . . 52
3.4.1 Γενικά . . . 52
3.4.2 Γραμμικές Διατάξεις . . . 52
3.4.3 Διάταξη Ενοτήτων . . . 53
3.5 Η Μέθοδος Διευθυνσιοδότησης MBaLt . . . 54
3.5.1 Διεσταλμένοι Ακέραιοι (Dilated Integers) και Δεικτοδότηση Morton . 55
3.5.2 Μια Πρώτη Προσέγγιση . . . 56
3.5.3 Θεωρία Μασκών . . . 57
3.5.4 Υλοποίηση-Σχεδιασμός Κώδικα . . . 58
3.5.5 Ο αλγόριθμος . . . 61

4 Παραγοντοποίηση Cholesky - Επίδοση MBaLt 65
4.1 Γενικά . . . 65
4.2 Σχεδίαση Tiled Μορφής με Γραμμικές Διατάξεις . . . 66
4.3 Σχεδίαση Tiled Μορφής με MBaLt . . . 70
4.4 Πειραματικά Αποτελέσματα . . . 72
4.4.1 Περιβάλλον Εκτέλεσης . . . 72
4.4.2 Χρονικές Μετρήσεις . . . 73
4.4.3 Αποτελέσματα Προσομοίωσης . . . 74
4.5 Συμπεράσματα . . . 75

5 Πολυνηματικές Αρχιτεκτονικές 85
5.1 Γενικά . . . 85
5.2 Πολυνημάτωση . . . 85
5.3 Ταυτόχρονη Πολυνημάτωση (SMT) . . . 86
5.4 Τεχνολογία Hyper-Threading (HT) . . . 88

6 Επίδοση MBaLt σε Ιερ. Μνήμης Πολυν. Αρχ. 91
6.1 Γενικά . . . 91
6.2 Πολυνηματική Παραλληλοποίηση . . . 91
6.3 Πολυνηματική Παραλληλοποίηση Παραγοντοποίησης Cholesky . . . 92
6.3.1 Λεπτοκομμένη Διαμέριση Εργασίας . . . 93
6.3.2 Χοντροκομμένη Διαμέριση Εργασίας . . . 94
6.3.3 Συνδυασμός Χοντροκομμένης και Λεπτοκομμένης Διαμέρισης . . . 95
6.4 Θέματα Υλοποίησης . . . 96
6.4.1 Νήματα POSIX . . . 96
6.4.2 CPU Affinity . . . 96
6.4.3 Μηχανισμοί Συγχρονισμού . . . 97
6.5 Πειραματικά Αποτελέσματα . . . 98
6.5.1 Περιβάλλον Εκτέλεσης . . . 98
6.5.2 Αποτελέσματα . . . 98
6.6 Συμπεράσματα . . . 101

Α΄ Συμπληρωματικά Θέματα **111**
Α΄.1 Εκτίμηση Κόστους Εκτέλεσης Φωλιασμένων Βρόχων 111
Α΄.1.1 Γενικά 111
Α΄.1.2 Ομάδες Αναφορών 111
Α΄.1.3 Κόστος Βρόχου 112
Α΄.1.4 Εύρεση της Βέλτιστης Δομής 114

Β΄ Πηγαίος Κώδικας **117**
Β΄.1 Πηγαίος Κώδικας Μετροπρογράμματος Cholesky 117
Β΄.2 Πηγαίος Κώδικας Πολυνηματικών Εκδοχών 125
Β΄.3 Βοηθητικές Συναρτήσεις 146
Β΄.3.1 Μεθόδου MBaLt 146
Β΄.3.2 CPU Affinity 148
Β΄.3.3 Spin-Locks 148

Βιβλιογραφία **151**

Ευρετήριο **153**

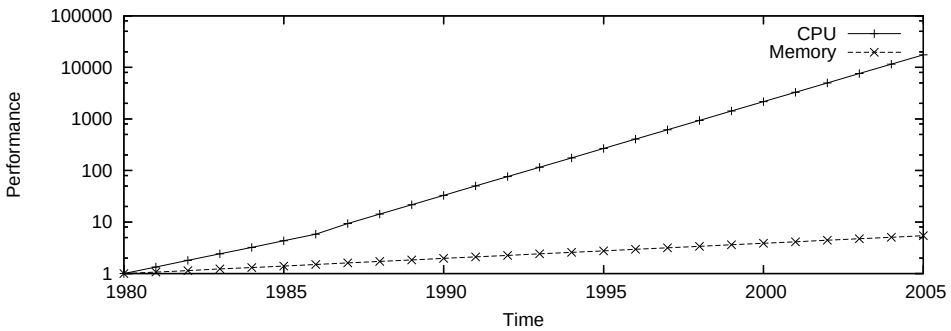
Εισαγωγή

0.1 Γενικά

Η τρομακτική τεχνολογική ανάπτυξη που σημειώθηκε τις τελευταίες δύο με τρεις δεκαετίες στον τομέα των VLSI είχε ως αποτέλεσμα την αύξηση της πυκνότητας ολοκλήρωσης σε μια ψηφίδα, που όπως σωστά πρόβλεψε ο J. Moore στα μέσα της δεκαετίας του 1960, η πυκνότητα αυτή διπλασιάζεται κάθε 18-24 μήνες. Η δυνατότητα αυτή για χρησιμοποίηση περισσότερων τρανζίστορ είχε ως αποτέλεσμα την επινόηση και υλοποίηση σωληνωτών και υπερβαθμωτών αρχιτεκτονικών οι οποίες αύξησαν την επεξεργαστική ισχύ της *κεντρικής μονάδας επεξεργασίας (ΚΜΕ)* (CPU).

Δυστυχώς όμως η ανάπτυξη αυτή της ΚΜΕ δεν συνοδευόταν και από την αντίστοιχη αύξηση επίδοσης της μονάδας μνήμης, οδηγώντας όπως φαίνεται και στο σχήμα 1 στην δημιουργία χάσματος μεταξύ επίδοσης του επεξεργαστή και της μνήμης ενός σύγχρονου υπολογιστικού συστήματος. Ενώ η επίδοση του επεξεργαστή παρουσιάζει ένα ρυθμό αύξησης 60% ανά έτος, ο ρυθμός αύξησης της επίδοσης της μονάδας μνήμης είναι ιδιαίτερα χαμηλός, μόνο 7%. Φτάσαμε λοιπόν σε ένα σημείο όπου η επίδοση της μονάδας μνήμης είναι 100 μέχρι και 1000 φορές μικρότερη από την επίδοση του επεξεργαστή.

Η μικρή διάρκεια του κύκλου μηχανής σε συνάρτηση με τη μη δραστική μείωση της καθυστέρησης μεταφοράς δεδομένων από την μονάδα μνήμης έχει σαν αποτέλεσμα να χρειάζονται ολοένα και περισσότεροι κύκλοι μηχανής για την μεταφορά αυτή. Η επίδοση της μονάδας μνήμης ενός σύγχρονου υπολογιστικού συστήματος είναι λοιπόν ένας σημαντικός παράγοντας για



Σχήμα 1: Το Χάσμα μεταξύ Επίδοσης της ΚΜΕ και της Μονάδας Μνήμης τα τελευταία 25 χρόνια

Σχήμα 2: Ιεραρχία Μνήμης

τον περιορισμό της συνολικής επίδοσης του.

Μια οικονομική, αλλά ταυτόχρονα και αποδοτική λύση στο πρόβλημα αυτό είναι η χρήση της *Ιεραρχίας Μνήμης* (memory hierarchy) . Η ιδέα πίσω από την ιεραρχία μνήμης όπως φαίνεται και στο σχήμα 2 είναι η τοποθέτηση μιας πολύ γρήγορης μικρής μνήμης, η οποία ονομάζεται *κρυφή μνήμη* (cache memory) και είναι τεχνολογίας SRAM, κοντά στον επεξεργαστή η οποία τροφοδοτείται από την μεγαλύτερη αλλά και πιο αργή *κεντρική μνήμη* (main memory) τεχνολογίας DRAM. Η ιεραρχία μνήμης παρέχει με αυτό τον τρόπο ένα σύστημα μνήμης με κόστο σχεδόν τόσο χαμηλό όσο το φθηνότερο επίπεδο μνήμης και ταχύτητα τόση όση του γρηγοροτέρου επιπέδου. Σήμερα, ιδιαίτερα διαδεδομένες είναι οι πολυεπίπεδες ιεραρχίες μνήμης οι οποίες δεν περιορίζονται μόνο στο ένα επίπεδο κρυφής μνήμης, αλλά χρησιμοποιούν μέχρι και τρία τέτοια επίπεδα.

Για την επίτευξη βέβαια υψηλής επίδοσης με τη χρήση της ιεραρχίας μνήμης απαιτείται η ελαχιστοποίηση της μεταφοράς δεδομένων από την κεντρική μνήμη προς την κρυφή μνήμη. Η δυνατότητα μιας τέτοιας ελαχιστοποίησης ευτυχώς δικαιολογείται από την *αρχή της τοπικότητας της αναφοράς* (principle of locality of reference) , η οποία δηλώνει ότι τα περισσότερα προγράμματα δεν προσπελάνουν ομοιόμορφα τον κώδικα και τα δεδομένα τους, αλλά έχουν την τάση κάποια μέρη να τα επαναπροσπελάνουν και μάλιστα χρονικά κοντα.

Αυξάνοντας λοιπόν την τοπικότητα των αναφορών σε ένα πρόγραμμα επιτυγχάνεται η όσο το δυνατόν καλύτερη εκμετάλλευση της ιεραρχίας μνήμης. Μια τέτοια αύξηση μπορεί να επιτευχθεί με τη χρήση μετασχηματισμών κώδικα[1] και δεδομένων οι οποίοι σκοπό έχουν την αναδιάταξη των αναφορών και τη σειρά αποθήκευσης των δεδομένων στη μνήμη. Τέτοιοι μετασχηματισμοί εφαρμόζονται συνήθως από τον *μεταγλωττιστή* (compiler) και είναι αρκετά αποτελεσματικοί στην περίπτωση εφαρμογών που διαθέτουν *μορφότυπα αναφορών* (reference patterns) που ακολουθούν μια κανονική δομή, όπως τα προγράμματα αριθμητικής ανάλυσης.

[1] Η παρούσα εργασία εστιάζεται στην κατηγορία των μετασχηματισμών κώδικα που εφαρμόζονται σε επαναληπτικούς βρόχους και οι οποίοι ονομάζονται *μετασχηματισμοί βρόχων*.

0.2 Σκοπός

Σκοπός της παρούσας διπλωματικής εργασίας είναι η ενιαία εφαρμογή μετασχηματισμών κώδικα και δεδομένων και η μελέτη της αποτελεσματικότητας τους για μείωση των καθυστερήσεων λόγω αστοχιών σε πολυεπίπεδες ιεραρχίες μνήμης σύγχρονων υπολογιστικών συστημάτων, συμπεριλαμβανομένων και πολυνηματικών αρχιτεκτονικών. Μεταξύ άλλων χρησιμοποιείται ο μετασχηματισμός tiling και οι διατάξεις ενοτήτων με αποδοτική δεικτόδότηση MBaLt για την υλοποίηση σειριακών και παράλληλων εκδοχών του μετροπρογράμματος 'Παραγοντοποίηση Cholesky', οι οποίες χρησιμοποιούνται για την μελέτη της επίδοσης των μετασχηματισμών.

0.3 Οργάνωση-Αντικείμενο της εργασίας

Στο πρώτο κεφάλαιο παρουσιάζεται η αρχιτεκτονική του συστήματος μνήμης. Χρησιμοποιούμε τον όρο αρχιτεκτονική, διότι εκτός από την παρουσίαση της δομής και της οργάνωσης της ιεραρχίας μνήμης, παρουσιάζουμε και ορισμένες βασικές βελτιστοποιήσεις που εφαρμόζονται τόσο από την πλευρά του υλικού, όσο και από την πλευρά του λογισμικού.

Στη συνέχεια, στο δεύτερο κεφάλαιο αναπτύσσεται το απαραίτητο θεωρητικό και μαθηματικό υπόβαθρο που αφορά τους μετασχηματισμούς βρόχων. Οι φωλιασμένοι βρόχοι αποτελούν τα πιο χρονοβόρα τμήματα σε ένα πρόγραμμα και η βελτιστοποίηση τους επηρεάζει καθοριστικά το συνολικό χρόνο εκτέλεσης. Εδώ, παρουσιάζεται και ο μετασχηματισμός tiling ο οποίος αποτελεί ένα από τους πιο σημαντικούς μετασχηματισμούς βρόχων για αύξηση της τοπικότητας των δεδομένων.

Στο τρίτο κεφάλαιο παρουσιάζεται το γενικό πεδίο γύρω από τους μετασχηματισμούς δεδομένων οι οποίοι προσανατολίζονται στην τροποποίηση των διατάξεων με τις οποίες αποθηκεύονται τα δεδομένα στη μνήμη. Το κεφάλαιο αυτό εστιάζεται στις διατάξεις ενοτήτων και παρουσιάζεται μια αποδοτική μέθοδος [2] για τη δεικτοδότηση των στοιχείων σε τέτοιες διατάξεις.

Στο τέταρτο κεφάλαιο υλοποιούνται διάφορες εκδοχές του μετροπρογράμματος 'Παραγοντοποίηση Cholesky' χρησιμοποιώντας μεταξύ άλλων το μετασχηματισμό tiling, γραμμικές διατάξεις και διατάξεις ενοτήτων. Παρουσιάζονται πραγματικές μετρήσεις που λήφθηκαν από την εκτέλεση του μετροπρογράμματος σε δύο διαφορετικά υπολογιστικά συστήματα, καθώς και αποτελέσματα προσομοιώσεων, τα οποία όλα μαζί υποστηρίζουν την αποδοτικότητα της μεθόδου δεικτοδότησης που παρουσιάζεται στο τρίτο κεφάλαιο.

Ακολούθως, στο πέμπτο κεφάλαιο γίνεται μια σύντομη αναφορά γύρω από τις πολυνηματικές αρχιτεκτονικές με έμφαση στην αρχιτεκτονική ταυτόχρονης πολυνημάτωσης SMT, για την οποία παρουσιάζεται μια εμπορική υλοποίηση της, η τεχνολογία Hyper-Threading της Intel.

Τέλος, στο έκτο κεφάλαιο υλοποιείται μια παράλληλη πολυνηματωμένη εκδοχή του μετροπρογράμματος 'Παραγοντοποίηση Cholesky' με χρήση tiling και διατάξεων ενοτήτων, για την μελέτη της επίδοσης των διατάξεων και σε αρχιτεκτονικές τύπου SMT. Τα αποτελέσματα τα οποία παρουσιάζονται, προκύπτουν από την εκτέλεση του μετροπρογράμματος σε υπολογιστικό σύστημα με τεχνολογία Hyper-Threading.

Κεφάλαιο 1

Αρχιτεκτονική Συστήματος Μνήμης

1.1 Βασικές Έννοιες Κρυφής Μνήμης

Η κρυφή μνήμη[1] αποτελεί το ένα από δύο βασικά επίπεδα της ιεραρχίας μνήμης. Το άλλο είναι η εικονική μνήμη η οποία θα συζητηθεί στην ενότητα 1.3

Η κρυφή μνήμη όπως αναφέρθηκε στην Εισαγωγή είναι το επίπεδο της ιεραρχίας μνήμης που βρίσκεται πιο κοντά στον επεξεργαστή και συνήθως στα σύγχρονα υπολογιστικά συστήματα αποτελείται από δύο ή και τρία επίπεδα κρυφής μνήμης τα οποία ονομάζονται L1, L2 και L3. Το πρώτο επίπεδο είναι πάντα ενσωματωμένο στην μονάδα επεξεργασίας, το οποίο συμβαίνει συνήθως και με το δεύτερο επίπεδο στις σύγχρονες ΚΜΕ.

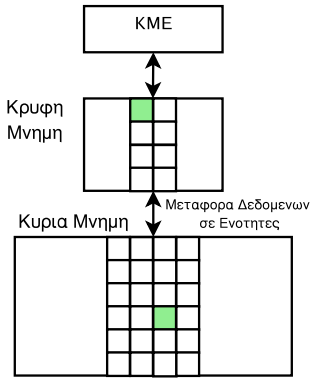
Όπως έχει ήδη αναφερθεί η ιεραρχία μνήμης και ειδικότερα η κρυφή μνήμη στηρίζει την ύπαρξη της στην *αρχή της τοπικότητας της αναφοράς* (principle of locality of reference) η οποία χαρακτηρίζει τον τρόπο που λειτουργούν τα προγράμματα. Η αρχή της τοπικότητας δηλώνει ότι σε ένα πρόγραμμα οι αναφορές στις διευθύνσεις μνήμης δεν είναι τελείως τυχαίες, αλλά αντίθετα τείνουν να είναι τοπικώς περιορισμένες για ορισμένο χρονικό διάστημα. Διακρίνονται δε, δύο είδη τοπικότητας:

- *Χρονική Τοπικότητα* (temporal locality) : Ένα δεδομένο που έχει χρησιμοποιηθεί τείνει να επαναχρησιμοποιηθεί σύντομα.

- *Χωρική Τοπικότητα* (spatial locality) : Δεδομένα που οι διευθύνσεις μνήμης τους είναι κοντά σε ένα δεδομένο που έχει χρησιμοποιηθεί, τείνουν να χρησιμοποιηθούν σύντομα. Για παράδειγμα τα γειτονικά δεδομένα ενός πίνακα.

Όταν ο επεξεργαστής κάνει μια αναφορά στην μνήμη, εξετάζεται αρχικά εάν η ζητούμενη *λέξη* (word) περιέχεται στην κρυφή μνήμη. Εάν πράγματι η λέξη βρίσκεται στην κρυφή μνήμη, *επιτυχία κρυφής μνήμης* (cache hit), τότε γίνεται αμέσως προσπέλαση σε αυτή. Εάν, όμως δεν βρεθεί η λέξη στην κρυφή μνήμη, *αστοχία κρυφής μνήμης* (cache miss), τότε διαβάζεται από την κύρια μνήμη και ταυτοχρόνως μεταφέρεται από την κύρια μνήμη στην κρυφή μνήμη μια *ενότητα* (block) λέξεων όπως φαίνεται στο σχήμα 1.1, που περιλαμβάνει και τη λέξη για

[1] Το παρών κεφάλαιο στηρίζεται στα [7][8]



Σχήμα 1.1: Μεταφορά δεδομένων σε ενότητες μεταξύ των επιπέδων της ιεραρχίας μνήμης

την οποία ξεκίνησε η παρούσα προσπέλαση. Η λέξη για την οποία ξεκίνησε η προσπέλαση μεταφέρεται στην κρυφή μνήμη για την ικανοποίηση τυχόν μελλοντικών αναφορών στην ίδια λέξη, λαμβάνοντας με αυτό τον τρόπο υπόψιν την ιδιότητα της χρονικής τοπικότητας της αναφοράς. Οι επιπλέον λέξεις της ενότητας μεταφέρονται στη γρήγορη μνήμη για να ικανοποιηθούν τυχόν μελλοντικές αναφορές μνήμης, λαμβάνοντας έτσι υπόψιν την ιδιότητα της χωρικής τοπικότητας της αναφοράς.

Ο χρόνος για την επίλυση μιας αστοχίας κρυφής μνήμης εξαρτάται τόσο από την καθυστέρηση απόκρισης (latency) της κύριας μνήμης, όσο και από το εύρος ζώνης (bandwidth) αυτής. Η καθυστέρηση απόκρισης καθορίζει τον χρόνο για την απόκτηση της πρώτης λέξης μιας ενότητας λέξεων. Το δε εύρος ζώνης καθορίζει τον χρόνο για την μεταφορά των υπόλοιπων λέξεων της ενότητας.

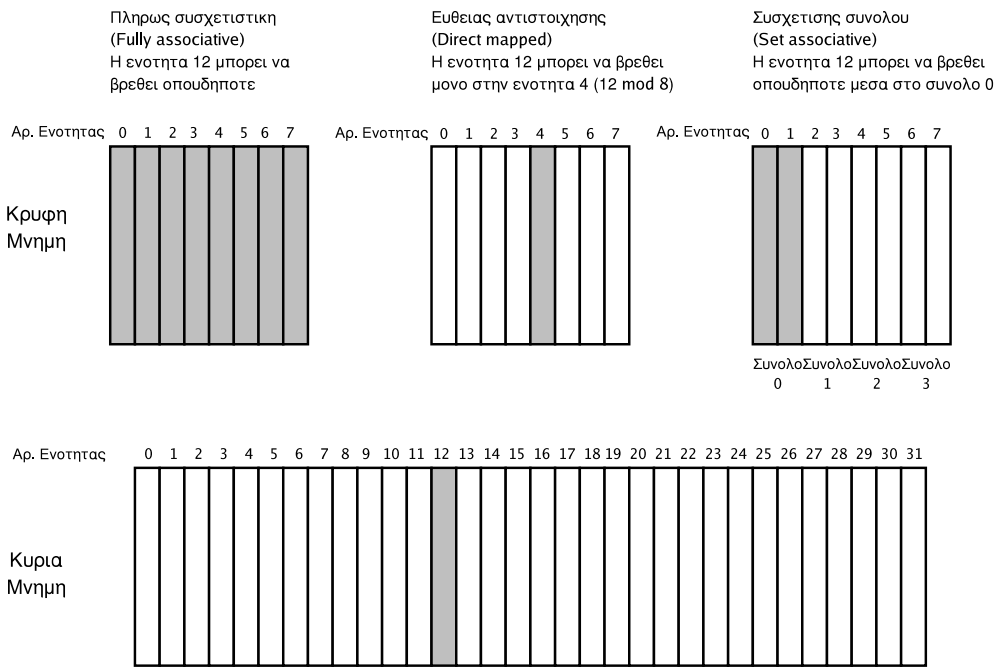
1.1.1 Οργάνωση Κρυφής Μνήμης

Όπως φαίνεται και στο σχήμα 1.2 οι περιορισμοί για το που θα τοποθετηθεί μια ενότητα δημιουργούν τρεις βασικές κατηγορίες οργάνωσης κρυφών μνημών:

- *Ευθείας αντιστοίχησης* (direct mapped) : Κάθε ενότητα μπορεί να τοποθετηθεί μόνο σε μια συγκεκριμένη θέση μέσα στην κρυφή μνήμη. Η αντιστοίχηση είναι συνήθως:
  (θέση στην κρυφή μνήμη) = (διεύθυνση ενότητας) MOD (πλήθος ενοτήτων στην κρυφή μνήμη)

- *Πλήρως συσχετιστική* (fully associative) : Κάθε ενότητα μπορεί να τοποθετηθεί σε οποιαδήποτε θέση μέσα στην κρυφή μνήμη.

- *Συσχέτισης συνόλου* (set associative) : Κάθε ενότητα μπορεί να τοποθετηθεί μόνο σε ένα σύνολο θέσεων της κρυφής μνήμης. Ένα σύνολο είναι μια ομάδα ενοτήτων. Μια ενότητα αρχικά αντιστοιχίζεται σε ένα σύνολο και ακολούθως μπορεί να τοποθετηθεί οπουδήποτε μέσα στο σύνολο. Η αντιστοίχηση είναι συνήθως:
  (αρ. συνόλου) = (διεύθυνση ενότητας) MOD (πλήθος συνόλων στην κρυφή μνήμη)



Σχήμα 1.2: Παράδειγμα οργάνωσης κρυφής μνήμης

Εάν υπάρχουν n ενότητες σε ένα σύνολο, τότε η κρυφή μνήμη ονομάζεται συσχέτισης συνόλου n δρόμων (n-way set associative).

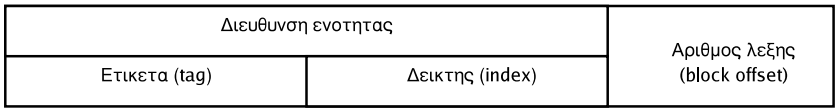
Την κρυφή μνήμη ευθείας αντιστοίχησης μπορεί να την δει κανείς ως μια κρυφή μνήμη συσχέτισης συνόλου ενός δρόμου, ενώ μια κρυφή μνήμη πλήρως συσχετιστική m ενοτήτων ως μια κρυφή μνήμη συσχέτισης συνόλου m δρόμων.

Τα παραπάνω εφαρμόζονται στο παράδειγμα του σχήματος 1.2 όπου παρουσιάζονται οι τρεις δυνατές οργανώσεις κρυφής μνήμης. Κάθε κρυφή μνήμη έχει χωρητικότητα 8 ενοτήτων, ενώ η κύρια μνήμη 32 ενοτήτων. Στην περίπτωση της πλήρως συσχετιστικής κρύφής μνήμης, η ενότητα 12 όπως φαίνεται και στο σχήμα μπορεί να τοποθετηθεί σε οποιαδήποτε θέση. Αντίθετα στις άλλες δύο οργανώσεις υπάρχει περιορισμός στη θέση. Συγκεκριμένα για την κρυφή μνήμη ευθείας αντιστοίχησης η ενότητα 12 μπορεί να τοποθετηθεί μόνο στη θέση 4 (12 mod 8), ενώ στην περίπτωση της κρυφής μνήμης συσχέτισης συνόλου δύο δρόμων υπάρχει μια λίγο μεγαλύτερη ελευθερία κι έτσι η ενότητα 12 μπορεί να τοποθετηθεί οπουδήποτε μέσα στο σύνολο 0 (12 mod 4).

1.1.2 Εύρεση Ενότητας μέσα στην Κρυφή Μνήμη

Στο σχήμα 1.3 φαίνεται πως γίνεται ο διαχωρισμός μιας διεύθυνσης μνήμης στα τρία πεδία: ετικέτα, δείκτης και αριθμός λέξης. Οι κρυφές μνήμες για να μπορούν να εντοπίζουν την ζητούμενη ενότητα αποθηκεύουν μια *ετικέτα* (tag) για κάθε ενότητα που περιέχουν μαζί με ένα *μπιτ εγκυρότητας* (valid bit) το οποίο δείχνει αν τα δεδομένα που υπάρχουν γι' αυτή τη θέση μνήμης ειναι έγκυρα.

Ο εντοπισμός της ζητούμενης ενότητας γίνεται ως εξής:



Σχήμα 1.3: Τα τρία πεδία μιας διεύθυνσης μνήμης

1. Αρχικά, το πεδίο *δείκτης* (index) της διεύθυνσης μνήμης χρησιμοποιείται για τον εντοπισμό του συνόλου στο οποίο ανήκει η ζητούμενη ενότητα.

2. Ακολούθως, για κάθε έγκυρη ενότητα που βρίσκεται μέσα στο σύνολο ελέγχεται αν η ετικέτα συμπίπτει με το αντίστοιχο τμήμα της διεύθυνσης μνήμης που δίνεται προς επίλυση από την ΚΜΕ. Εάν βρεθεί μια τέτοια ενότητα, τότε αυτό σημαίνει ότι η λέξη με τη δοθείσα διεύθυνση μνήμης υπάρχει μέσα στην κρυφή μνήμη. Ο έλεγχος αυτός κατά κανόνα πρέπει να γίνεται παράλληλα για λόγους ταχύτητας.

3. Τέλος, χρησιμοποιείται ο *αριθμός λέξης* (block offset) για να επιλεχθεί η επιθυμητή λέξη μέσα από την ενότητα.

1.1.3 Αντικατάσταση Ενότητας στην περίπτωση αστοχίας Κρυφής Μνήμης

Στην περίπτωση αστοχίας κρυφής μνήμης, ο ελεγκτής της κρυφής μνήμης πρέπει να αποφασίσει ποια ενότητα θα αντικατασταθεί. Ένα πλεονέκτημα της κρυφής μνήμης με ευθεία αντιστοίχηση είναι η απλότητα της λήψης μιας τέτοιας απόφασης. Από τη στιγμή που μόνο μια ενότητα ελέγχεται, αυτό σημαίνει ότι μόνο αυτή η ενότητα μπορεί να αντικατασταθεί. Αντίθετα, με τους άλλους δύο τύπους οργάνωσης υπάρχουν περισσότερες από μια ενότητες προς αντικατάσταση και αυτό περιπλέκει τη λήψη απόφασης. Σ' αυτές τις περιπτώσεις υπάρχουν τρεις δημοφιλείς στρατηγικές για την επιλογή της ενότητας που θα αντικατασταθεί:

- *Τυχαία επιλογή* (random) : Οι υποψήφιες ενότητες επιλέγονται τυχαία χρησιμοποιώντας κάποια γεννήτρια ψευδοτυχαίων αριθμών.

- *Η πιο πρόσφατα χρησιμοποιούμενη* (LRU-least recently used) : Επίλεγεται η ενότητα στην οποία δεν έγινε αναφορά για το μεγαλύτερο χρονικό διάστημα. Η στρατηγική αυτή στηρίζεται έμμεσα στην αρχή της τοπικότητας της αναφοράς: αφού οι ενότητες που χρησιμοποιήθηκαν πρόσφατα έχουν μεγάλη πιθανότητα να επαναχρησιμοποιηθούν, τότε ένας καλός υποψήφιος για αντικατάσταση είναι η ενότητα που έχει τον περισσότερο χρόνο να χρησιμοποιηθεί.

- *Με βάση το χρόνο παραμονής* (FIFO) : Επιλέγεται η ενότητα με τον περισσότερο χρόνο παραμονής στην κρυφή μνήμη.

Από τις παραπάνω στρατηγικές η πιο απλή στην υλοποίηση της είναι η στρατηγική της τυχαίας επιλογής η οποία και χρησιμοποιείται συχνά. Η στρατηγική της πιο πρόσφατα χρησιμοποιούμενης χρειάζεται την φύλαξη αρκετής πληροφορίας γι' αυτό και το κόστος υλοποίησης της είναι αρκετά υψηλό για κρυφές μνήμες με μεγάλο πλήθος ενοτήτων. Έτσι στην πράξη υλοποιείται συνήθως κάποια προσέγγιση της στρατηγικής αυτής.

1.1.4 Μηχανισμοί Εγγραφής Ενότητας

Υπάρχουν δύο βασικοί μηχανισμοί για την εγγραφή μιας ενότητας στην κρυφή μνήμη στην περίπτωση *επιτυχίας εγγραφής* (write hit):

- *Επανεγγραφή* (write-back) : Η πληροφορία εγγράφεται μόνο στην ενότητα της κρυφής μνήμης, ενώ η κύρια μνήμη ενημερώνεται μόνο κατά την απομάκρυνση της ενότητας από την κρυφή μνήμη.

- *Δι-εγγραφή* (write-through) : Η πληροφορία εγγράφεται τόσο στην ενότητα της κρυφής μνήμης, όσο και της κύριας μνήμης.

Για να ελαττωθεί περαιτέρω η συχνότητα εγγραφών στην κύρια μνήμη στην περίπτωση της επανεγγραφής, χρησιμοποιείται το *μπιτ τροποποίησης* (dirty bit). Αυτό το μπιτ κατάστασης δηλώνει κατά πόσο μια ενότητα είναι τροποποιημένη ή μη. Εάν είναι μη τροποποιημένη τότε δεν χρειάζεται η επανεγγραφή της ενότητας στην περίπτωση απομάκρυνση της, αφού τα δεδομένα που περιέχει η κύρια μνήμη και η κρυφή μνήμη για τη συγκεκριμένη ενότητα είναι ταυτόσημα.

Τόσο η μέθοδος της επανεγγραφής, όσο και η μέθοδος της δι-εγγραφής έχουν τα πλεονεκτήματα τους. Τα πλεονεκτήματα της επανεγγραφής είναι ότι οι εγγραφές πραγματοποιούνται με την ταχύτητα της κρυφής μνήμης. Επίσης πολλές εγγραφές στην ίδια ενότητα ανάγονται μόνο σε μια ενημέρωση της κύριας μνήμης, χρησιμοποιώντας με αυτό τον τρόπο συνολικά λιγότερο εύρος ζώνης κύριας μνήμης.

Η μέθοδος της δι-εγγραφής είναι ευκολότερη στην υλοποίηση της. Επίσης με την δι-εγγραφή εγγυάτε ότι το αμέσως κατώτερο επίπεδο της ιεραρχίας έχει πάντα το πιο πρόσφατο αντίγραφο, κάτι το οποίο απλοποιεί την συνάφεια δεδομένων. Βέβαια με την δι-εγγραφή υπάρχει αυξημένη μετακίνηση δεδομένων προς την κύρια μνήμη, συνεπάγοντας χρησιμοποίηση μεγαλύτερου εύρου ζώνης.

Στην περίπτωση *αστοχίας εγγραφής* (write miss) υπάρχουν δύο δυνατές επιλογές:

- *Εγγραφή με παραχώρηση* (write allocate) : Παραχωρείται χώρος από την κρυφή μνήμη, η ενότητα επομένως φορτώνεται από την κύρια μνήμη στην κρυφή και στη συνέχεια ακολουθούνται οι ενέργειες για επιτυχία εγγραφής που αναφέρονται παραπάνω.

- *Εγγραφή χωρίς παραχώρηση* (No-write allocate) : Δεν παραχωρείται κάποιος χώρος από την κρυφή μνήμη και επομένως οι μετατροπές δεδομένων γίνονται απευθείας στην κύρια μνήμη.

Συνήθως η επανεγγραφή συνδυάζεται με εγγραφή με παραχώρηση και η δι-εγγραφή με εγγραφή χωρίς παραχώρηση.

Οι Τρεις Κατηγορίες αστοχιών Κρυφής Μνήμης

Οι αστοχίες κρυφής μνήμης μπορούν να διαχωρισθούν σε τρεις κατηγορίες:

- *Αστοχίες Υποχρεωτικές* (Compulsory misses): Είναι οι αστοχίες κρυφής μνήμης που προκαλούνται από την πρώτη πρόσβαση σε μια ενότητα που δεν τοποθετήθηκε ποτέ στην κρυφή μνήμη.

- *Αστοχίες Χωρητικότητας* (Capacity misses): Είναι οι αστοχίες κρυφής μνήμης που προκαλούνται όταν η κρυφή μνήμη δεν μπορεί να περιέχει όλες τις ενότητες που χρειάζονται κατά την εκτέλεση ενός προγράμματος.

- *Αστοχίες Σύγκρουσης* (Conflict misses): Είναι οι αστοχίες κρυφής μνήμης που εμφανίζονται στις κρυφές μνήμες συσχέτισης συνόλου ή ευθείας αντιστοίχησης όταν πολλαπλές ενότητες ανταγωνίζονται για το ίδιο σύνολο. Εναλλακτικά, οι αστοχίες σύγκρουσης είναι οι αστοχίες μιας κρυφής μνήμης συσχέτισης συνόλου ή ευθείας αντιστοίχησης κρυφής μνήμης που εξαλείφονται με τη χρήση μιας πλήρους συσχετιστικής κρυφής μνήμης. Οι αστοχίες σύγκρουσης περαιτέρω χωρίζονται σε δύο κατηγορίες: στις συγκρούσεις μεταξύ διαφορετικών μεταβλητών. (cross-interference) και στις συγκρούσεις μεταξύ στοιχείων του ίδιου πίνακα (self-interference).

1.2 Βελτιστοποίηση Επίδοσης Κρυφής Μνήμης

Για την βελτιστοποίηση της επίδοσης της κρυφής μνήμης οι ερευνητές επικεντρώθηκαν στην μείωση των τριων συνιστωσών που επηρεάζουν τον μέσο χρόνο πρόσβασης των δεδομένων στη μνήμη: την χρονική επιβάρυνση από μια αστοχία κρύφης μνήμης λόγω μεταφοράς δεδομένων από το χαμηλότερο επίπεδο της ιεραρχίας μνήμης, το ρυθμό αστοχιών κρυφής μνήμης και το χρόνο προσπέλασης κρυφής μνήμης. Το πρόβλημα προσεγγίζεται τόσο από την πλευρά του υλικού, όσο και από την πλευρά του λογισμικού μέσω των μεταγλωττιστών.

1.2.1 Στρατηγικές Υλικού

Μια μέθοδος για την μείωση της χρονικής επιβάρυνσης από μια αστοχία μνήμης είναι μέσω της απομόνωσης των αναφορών προς τη μνήμη. Η χρήση *απομονωτή εγγραφής* (write buffer) εφαρμόζεται για την απομόνωση των αναφορών εγγραφής. Χρησιμοποιώντας απομονωτές εγγραφής, ο επεξεργαστής δεν χρειάζεται να περιμένει την συμπλήρωση της εγγραφής. Αντίθετα, απλώς εκτελεί την εγγραφή στέλνοντας την στον απομονωτή, το οποίο γίνεται μέσα σ΄ ένα κύκλο μηχανής. Το πλεονέκτημα που προκύπτει από τη χρήση ενός απομονωτή εγγραφής δεν είναι μόνο ότι ο επεξεργαστής δεν μπλοκάρεται κατά την εκτέλεση της εγγραφής, αλλά και το ότι μειώνεται η συμφόρηση στον διάδρομο του συστήματος καθυστερώντας την εγγραφή μέχρι να βρεθεί ένας κενός κύκλος διαδρόμου.

Στους σύγχρονους υπολογιστές τύπου αγωγού που υποστηρίζουν εκτέλεση εντολών *εκτός σειράς* (out-of-order) ή σε *πολυνηματικές* (multithreading) αρχιτεκτονικές όπου γίνεται *μεταγωγή* (switching) μεταξύ *νημάτων* , ο επεξεργαστής δεν χρειάζεται να μπλοκάρεται αναμένοντας την επίλυση μιας αστοχίας κρυφής μνήμης, αλλά μπορεί να συνεχίσει με την εκτέλεση άλλων ανεξάρτητων εντολών μέχρι την ολοκλήρωση της επίλυσης. Σε τέτοια μηχανήματα είναι απαραίτητη η χρήση *κρυφής μνήμης χωρίς - μπλοκάρισμα* (non-blocking cache) . Μια τέτοια κρυφή μνήμη μπορεί να εξυπηρετήσει μια νέα αναφορά μνήμης, ενώ ταυτόχρονα επεξεργάζεται την προηγούμενη αστοχία, κρύβοντας έτσι το χρόνο που χρειάζεται η επίλυση της αστοχίας. Οι σύγχρονες κρυφές μνήμες αυτού του τύπου επιτρέπουν δε, την εκκρεμότητα αρκετών αστοχιών.

Για την μείωση του ρυθμού αστοχιών εξαιρετικά ωφέλιμες είναι βέβαια οι κρυφές μνήμες με μεγάλο μέγεθος ή με ψηλό βαθμό συσχέτισης. Ένας πρακτικός κανόνας λέει ότι οι κρυφές

μνήμες ευθείας αντιστοίχησης μεγέθους N έχουν τον ίδιο ρυθμό αστοχιών με τις κρυφές μνήμες συσχέτισης συνόλου δύο δρόμων μεγέθους N/2.

Δύο τεχνικές με τις οποίες επιτυγχάνεται τόσο η μείωση της χρονικής επιβάρυνσης λόγω αστοχίας, όσο και η μείωση του ρυθμού αστοχιών είναι το hardware prefetching και η χρήση κρυφής μνήμης θυμάτων. Η τεχνική του hardware prefetching, στηρίζεται κι αυτή στη κρυφή μνήμη χωρίς - μπλοκάρισμα. Στην τεχνική αυτή γίνονται αναφορές σε δεδομένα πριν αυτα χρειαστούν από τον επεξεργαστή, έτσι ώστε μέχρι ο επεξεργαστής τα χρειαστεί, να έχουν ήδη επιλυθεί οι οποιεσδήποτε αστοχίες κρυφής μνήμης που προέκυψαν από τις αναφορές αυτές. Βέβαια για την καλή απόδοση αυτής της τεχνικής, πρέπει να γίνει νωρίς η σωστή πρόβλεψη των δεδομένων που θα χρειαστεί ο επεξεργαστής το οποίο είναι αρκετά πολύπλοκο να γίνει με μεγάλη ακρίβεια από το υλικό. Γι' αυτό και το prefetching των εντολών μπορεί να καθοδηγείται από τον μεταγλωττιστή (compiler prefetching) , ο οποίος έχει τη δυνατότητα της ανάλυσης του προγράμματος και ακολούθως να προσθέσει σε κατάλληλα σημεία εντολές για το prefetching δεδομένων τα οποία θα χρειαστούν πιθανές μελλοντικές αναφορές.

Στην στρατηγική χρήσης *κρυφής μνήμης θυμάτων* (victim cache) χρησιμοποιείται μια επιπλέον μικρή πλήρως συσχετιστική κρυφή μνήμη η οποία βρίσκεται μεταξύ της κρυφής μνήμης και του επόμενου κατώτερου επιπέδου της ιεραρχίας μνήμης. Αυτή η επιπλέον κρυφή μνήμη περιέχει μόνο τις ενότητες θύματα οι οποίες απομακρύνθηκαν από την κρυφή μνήμη λόγω αστοχίας. Η κρυφή μνήμη θυμάτων ελέγχεται στην περίπτωση αστοχίας εάν περιέχει τα απαιτούμενα δεδομένα, πριν αυτά αναζητηθούν στο επόμενο επίπεδο, κι αν όντος βρεθούν μέσα σε αυτή, τότε τα δεδομένα των δύο κρυφών μνημών ανταλλάσονται.

Η μείωση του χρόνου προσπέλασης κρυφής μνήμης επιτυγχάνεται με τη χρήση μικρών κι απλών κρυφών μνημών. Στις κρυφές μνήμες ξοδεύεται αρκετός χρόνος για την εύρεση της υποψήφιας ενότητας μέσω του δείκτη, κι ακολούθως για τη σύγκριση των ετικετών. Επομένως σε μια κρυφή μνήμη με μικρό μέγεθος, ο χρόνος εύρεσης της υποψήφιας ενότητας είναι και πιο μικρός. Επίσης η απλή σχεδίαση της κρυφής μνήμης πλεονεκτεί, αφού για την έρευση της ενότητας στις κρυφές μνήμες συσχέτισης συνόλου χρειάζεται πιο πολύπλοκο υλικό, που συνεπάγεται περισσότερο χρόνο. Επιπρόσθετα, οι κρυφές μνήμες ευθείας αντιστοίχησης πλεονεκτούν και λόγω του ότι είναι δυνατή η επικάληψη της σύγκρισης της ετικέτας με την ταυτόχρονη αποστολή των δεδομένων προς τον επεξεργαστή. Εάν η σύγκριση προκύψει λανθασμένη, τότε ο επεξεργαστής απλώς απορρίπτει τα δεδομένα αυτά. Τέλος η μικρή κι απλή κρυφή μνήμη τοποθετήται στην ίδια ψηφίδα με την ΚΜΕ, μικραίνοντας με αυτό τον τρόπο το χρόνο μεταφοράς δεδομένων μεταξύ επεξεργαστή και κρυφής μνήμης.

Ο χρόνος μετάφρασης των εικονικών διευθύνσεων στις φυσικές προκειμένου να κληθούν τα δεδομένα από τη μνήμη μπορεί να αποφευχθεί, αν η κρυφή μνήμη που χρησιμοποιείται είναι εικονική. Από τη στιγμή που οι επιτυχίες μνήμης είναι πιο συχνές από τις αστοχίες, είναι προτιμότερο η διευθυνσιοδότηση της κρυφής μνήμης να είναι εικονική, οπότε θα απαιτείται αντιστοίχηση με τις φυσικές διευθύνσεις μόνο στην περίπτωση που πρέπει να γίνει επίλυση της αστοχίας με ανάτρεξη στην κύρια μνήμη. Κάθε φορά όμως που ξεκινάει η εκτέλεση ενός προγράμματος απαιτείται η φόρτωση της κρυφής μνήμης, το οποίο βέβαια επιβαρύνει τον συνολικό χρόνο εκτέλεσης.

1.2.2 Στρατηγικές Μεταγλωττιστή

Η αποδοτικότητα όλων των προηγούμενων μηχανισμών εξαρτάται από την ικανότητα των μεταγλωττιστών να μετασχηματίζουν την δομή των προγραμμάτων (κώδικα και δεδομένα)

προς εκτέλεση, έτσι ώστε να αξιοποιούνται πλήρως οι δυνατότητες και τα πλεονεκτήματα που παρέχουν οι μηχανισμοί αυτοί, επιτυγχάνοντας με αυτό υψηλή επίδοση. Ο μεταγλωττιστής καλείται να βρει τη μέγιστη *παραλληλία στο επίπεδο εντολής* (ILP - instruction level parallelism) για να τροφοδοτεί τους πόρους της μηχανής, κι επίσης να μετασχηματίσει κατάλληλα το προγράμμα έτσι ώστε να επιτύχει το μέγιστο δυνατό βαθμό τοπικότητας αξιοποιώντας την ιεραρχία μνήμης.

Κατά την βελτιστοποίηση ενός προγράμματος, το μεγαλύτερο κέρδος προκύπτει από τη βελτιστοποίηση των τμημάτων του προγράμματος που χρειάζονται το μεγαλύτερο χρόνο για να διεκπεραιωθούν. Οι επαναληπτικοί βρόχοι, οι οποίοι περιέχουν μεταβλητές πολυδιάστατων πινάκων, είναι τέτοια τμήματα που προσφέρονται για βελτιστοποίηση και οι οποίοι γι΄ αυτό το λόγο τυγχάνουν μεγάλης προσοχής από την ερευνητική κοινότητα.

Οι μετασχηματισμοί που μπορεί να εφαρμόσει ο μεταγλωττιστής για την βελτίστοποίηση ενός προγράμματος χωρίζονται σε δύο κατηγορίες:

- *Μετασχηματισμοί Βρόχων* (loop transformations) : Είναι μια μορφή μετασχηματισμών κώδικα οι οποίοι σκοπό έχουν την ελάττωση του πλήθους εντολών προς εκτέλεση ή/και την αλλαγή της σειράς εκτέλεσης αυτών.

- *Μετασχηματισμοί Δεδομένων* (data transformations) : Σκοπό έχουν την αναδιοργάνωση της διάταξης των δεδομένων με την οποία αυτά αποθηκεύονται στην μνήμη. Είναι αποτελεσματικοί για την ελάττωση των αστοχιών κρυφής μνήμης λόγω σύγκρουσης, cross-interference και self-interference. Συνήθως συνδυάζονται με τους μετασχηματισμούς βρόχους για να δώσουν την βέλτιστη δυνατή επίδοση.

Οι μετασχηματισμοί βρόχων, οι οποίοι μπορούν και να συνδυαστούν[14] μεταξύ τους για περισσότερη βελτιστοποίηση, μπορούν να χωριστούν σε δύο κατηγορίες:

- Σε αυτούς που σκοπό έχουν την βελτίωση της παραλληλίας στο επίπεδο εντολής: *ξεδίπλωμα εσωτερικότερου βρόχου* (inner unrolling), *software pipelining*.

- Σε αυτούς που σκοπό έχουν την βελτίωση της τοπικότητας των δεδομένων: *μετάθεση βρόχων* (loop permutation), *διάσπαση βρόχου* (loop fission), *συγχώνευση βρόχων* (loop fussion), *scalar replacement* (αλλά στο επίπεδο των καταχωρητών), *πλακόστρωση* (tiling) .

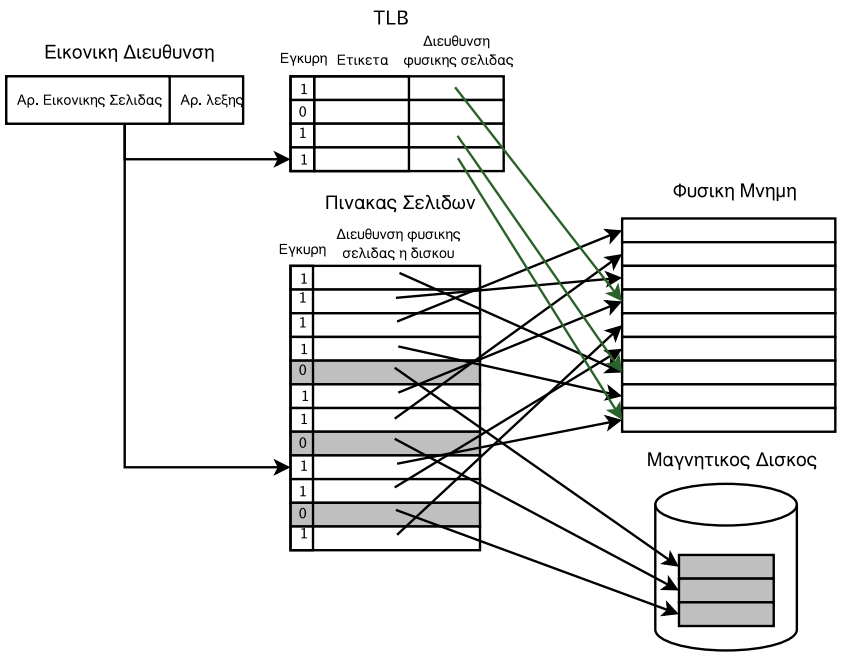
Ο πιο δημοφιλής μετασχηματισμός είναι ο μετασχηματισμός tiling, ο οποίος έχει κάποιες ενδιαφέρους ιδιότητες:

- Βελτιώνει την παραλληλία στο επίπεδο εντολής, όταν εφαρμοσθεί στο επίπεδο καταχωρητών.

- Εκμεταλλεύεται την επαναχρησιμοποίηση δεδομένων σε διάφορες διαστάσεις του *χώρου επαναλήψεων* (iteration space).

- Βελτιώνει ταυτόχρονα την τοπικότητα των δεδομένων σε διάφορα επίπεδα της ιεραρχίας μνήμης.

Οι παραπάνω μετασχηματισμοί βρόχων παρουσιάζονται με μεγαλύτερη λεπτομέρεια στο κεφάλαιο 2, όπου επίσης παρουσιάζεται η μαθηματική θεωρία γύρω από τους *φωλιασμένους βρόχους* (nested loops) και τους μετασχηματισμούς βρόχων. Η προσοχή μας όμως εστιάζεται στην ενιαία εφαρμογή και μελέτη των μετασχηματισμών δεδομένων με τους μετασχηματισμούς βρόχων και ξεκινάει από το κεφάλαιο 3.



Σχήμα 1.4: Παράδειγμα μιας τυπικής οργάνωσης Εικονικής Μνήμης

1.3 Οργάνωση Εικονικής Μνήμης

Όπως η κρυφή μνήμη λειτουργεί ως ένας ενδιάμεσος γρήγορος αποθηκευτικός χώρος μεταξύ επεξεργαστή και κύριας μνήμης, έτσι και η κύρια μνήμη μπορεί να λειτουργήσει ως ένας ενδιάμεσος χώρος μεταξύ κρυφής μνήμης και μαγνητικού δίσκου. Αυτή η τεχνική ονομάζεται *εικονική μνήμη* (virtual memory) και έχει δύο σημαντικά πλεονεκτήματα: επιτρέπει την κοινή χρήση της μνήμης από διάφορα προγράμματα με αποδοτικό και ασφαλή τρόπο, και απαλλάσει τον προγραμματιστή από το βάρος της μικρής και περιορισμένης κύριας μνήμης.

Με την εικονική μνήμη κάθε πρόγραμμα έχει το δικό του *εικονικό πεδίο διευθύνσεων* (virtual address space), δηλαδή ένα ξεχωριστό χώρο εικονικών θέσεων μνήμης προσβάσιμο μόνο από αυτό το πρόγραμμα. Οι δε θέσεις αυτές αντιστοιχίζονται από τον μηχανισμό εικονικής μνήμης σε πραγματικές φυσικές θέσεις μνήμης. Έχοντας λοιπόν αυτόνομα πεδία διευθύνσεων, επιτυγχάνεται η απομόνωση του κάθε προγράμματος, αφού κάθε πρόγραμμα γνωρίζει κι έχει πρόσβαση μόνο στο δικό του χώρο διευθύνσεων. Επίσης ο εικονικός χώρος διευθύνσεων δεν περιορίζεται από το μικρό μέγεθος της φυσικής κύριας μνήμης, παρέχοντας με αυτό τον τρόπο σε κάθε πρόγραμμα ένα πολύ μεγάλο χώρο διευθύνσεων.

Ο χώρος των εικονικών διευθύνσεων μνήμης χωρίζεται σε ένα αριθμό από ίσα και συνεχόμενα μέρη, τα οποία αναφέρονται ως *σελίδες* (pages) . Στα συστήματα που διαθέτουν εικονική μνήμη η ΚΜΕ παράγει διευθύνσεις εικονικής μνήμης οι οποίες μεταφράζονται με βάση ένα πίνακα σελίδων στην αντίστοιχη φυσική διεύθυνση η οποία χρησιμοποιείται για την πρόσβαση στην κύρια μνήμη.

Όπως φαίνεται και στο σχήμα 1.4, η εικονική διεύθυνση χωρίζεται σε δύο τμήματα: τον *αριθμός εικονικής σελίδας* (virtual page number) και τον *αριθμό λέξης* (page offset). Ο αριθ-

μός εικονικής σελίδας μεταφράζεται με βάση των πίνακα σελίδων στον αριθμό της αντίστοιχης φυσικής σελίδας και αποτελεί το άνω μέρος της φυσικής διεύθυνσης. Ο δε *αριθμός λέξης* (page offset) αποτελεί το κάτω μέρος της φυσικής διεύθυνσης και εντοπίζει την ζητούμενη λέξη μέσα στη φυσική σελίδα.

Επειδή ο πίνακας σελίδων βρίσκεται αποθηκευμένος στην κύρια μνήμη, αυτό συνεπάγεται ότι καθε φορά που θα πρέπει να μεταφραστεί μια εικονική διεύθυνση μνήμης θα πρέπει να γίνει πρόσβαση της κύριας μνήμης δύο φορές: μία φορά για την εύρεση της αντιστοίχησης και μία για την εύρεση της λέξης. Για την αποφυγή της πρώτης πρόσβασης και τη μείωση επομένως του χρόνου μετάφρασης χρησιμοποιείται μια κρυφή μνήμη η οποία ονομάζεται *προσωρινός καταχωρητής μετάφρασης* (TLB - translation look-aside buffer), και περιέχει ορισμένες καταχωρήσεις του πίνακα σελίδων. Και πάλι βάση της αρχής της τοπικότητας της αναφοράς, αποθηκεύονται οι πιο προσφατες αντιστοιχήσεις σελίδων.

Έτσι, όταν ο επεξεργαστής ζητήσει μια διεύθυνση εικονικής μνήμης ελέγχεται πρώτα ο TLB κι αν βρεθεί μια έγκυρη καταχώρηση (*επιτυχία TLB*) της οποίας η ετικέτα συμπίπτει με τον αριθμό της εικονικής σελίδας, τότε χρησιμοποιείται ο αριθμός φυσικής σελίδας που δίνει ο TLB. Αντίθετα, αν δεν βρεθεί κάποια καταχώρηση (*αστοχία TLB*) τότε η μετάφραση γίνεται μέσω του πίνακα σελίδων.

Όταν ο ρυθμός αστοχιών ενός επιπέδου της ιεραρχίας μνήμης είναι αρκετά υψηλός τότε λέμε ότι υπάρχει *υπερχείληση* (thrash) του επιπέδου αυτού. Στην περίπτωση της εικονικής μνήμης το πρόβλημα της υπερχείλησης είναι οξύτερο απ' ότι σ' άλλα επίπεδα, κυρίως λόγω του υψηλού κόστους το οποίο χαρακτηρίζει τη μεταφορά μιας σελίδας από το δίσκο στην κύρια μνήμη. Επίσης, η υπερχείληση TLB (TLB thrashing) είναι κι αυτή αρκετά δαπανηρή, λόγω του ότι καταναλώνεται αρκετό εύρος ζώνης και εισάγωνται μεγάλες χρονικές καθυστερήσεις για την μετάφραση των εικονικών διευθύνσεων σε φυσικές.

Κεφάλαιο 2

Μετασχηματισμοί Βρόχων

2.1 Βασικές Έννοιες

2.1.1 Φωλιασμένοι Βρόχοι

Ένα *σύστημα τέλεια φωλιασμένων βρόχων* (perfectly nested loops) έχει την ακόλουθη μορφή:

```
for (i1=lb1; i1<=ub1; i1++)
    for (i1=lb1; i1<=ub1; i1++)
        ...    ...    ...
        for (in=lbn; in<=ubn; in++) {
            /* loop body */
        }
```

, δηλαδή όλες οι εντολές που εκτελούνται βρίσκονται στον εσωτερικότερο βρόχο. Τα όρια των βρόχων έχουν την ακόλουθη μορφή:

$$lb_r = max(l_{r,0}, l_{r,1}, l_{r,2}, \ldots) \qquad ub_r = max(u_{r,0}, u_{r,1}, u_{r,2}, \ldots)$$

και τα $l'_{r,j}$ και $u'_{r,j}$ είναι γραμμικές συναρτήσεις της ακόλουθης μορφής:

$$l_{r,j} = \sum_{k=1}^{r-1} a^k_{r,j} \cdot i_k + Y_{r,j} \qquad u_{r,j} = \sum_{k=1}^{r-1} b^k_{r,j} \cdot i_k + F_{r,j}$$

όπου $a^k_{r,j}, b^k_{r,j} \in \mathbf{Z}$ $(1 \leq k \leq r-1)$, $Y_{r,j}$ και $F_{r,j}$ είναι σταθερές και i_k $(1 \leq k \leq r-1)$ είναι οι μεταβλητές ελέγχου των βρόχων.

Πολλές φορές οι βρόχοι σε ένα πρόγραμμα ενδέχεται να μην είναι τελείως φωλιασμένοι, όπως συμβαίνει για παράδειγμα στην *αποσύνθεση LU* και την *παραγοντοποίηση Cholesky*. Τέτοιοι όμως βρόχοι μπορούν να αναχθούν σε τελείως φωλιασμένους βρόχους χρησιμοποιώντας τον μετασχηματισμό *βύθισης κώδικα*[8] (code sinking). Ο μετασχηματισμός αυτός μετατοπίζει όλες τις εντολές που βρίσκονται μεταξύ των βρόχων προς τον εσωτερικότερο βρόχο προσθέτοντας έναν ή περισσότερους ελέγχους συνθηκών. Πρέπει όμως να προσεχθεί ότι η εφαρμογή ενός τέτοιου μετασχηματισμού είναι έγκυρη αν και μόνο αν οι βρόχοι δεν εκτελούν κενές επαναλήψεις. Ο μετασχηματισμός αυτός παρουσιάζεται στο σχήμα 2.1.

```
for (i1=lb1; i1<=ub1; i1++) {
   /* statements 1 */
   for (i1=lb1; i1<=ub1; i1++) {
      /* statements 2 */
      ...   ...   ...
      for (in=lbn; in<=ubn; in++) {
         /* statements n */
      }
      /* statements 2' */
   }
   /* statements 1' */
}
```

αρχική μορφή

```
for (i1=lb1; i1<=ub1; i1++) {
   for (i1=lb1; i1<=ub1; i1++) {
      ...   ...   ...
      for (in=lbn; in<=ubn; in++) {
         if (in==lbn && ... && i2==lb2)
            /* statements 1 */
         if (in==lbn && ... )
            /* statements 2 */
         /* statements n */
         if (in==lbn && ... )
            /* statements 2' */
         if (in==lbn && ... && i2==lb2)
            /* statements 1' */
      }
   }
}
```

τελική μορφή

Σχήμα 2.1: Μετατροπή ενός μη τέλειου φωλιασμένου βρόχου σε τέλειο φωλιασμένο βρόχο

Η προσθήκη των συνθηκών ελέγχου αυξάνει βέβαια τον χρόνο εκτέλεσης, επειδή σε κάθε επανάληψη θα πρέπει οι συνθήκες αυτές να αποτιμούνται. Για την αποφυγή λοιπόν αυτής της επιβάρυνσης, θα πρέπει να γίνεται απλοποίηση και απαλοιφή των περιττών συνθηκών.

2.1.2 Χώρος Επαναλήψεων

Ορισμός 2.1 *Κάθε επανάληψη του συστήματος αυτού των φωλιασμένων βρόχων αντιπροσωπεύεται από το n-διάστατο διάνυσμα* $\vec{I} = (i_1, i_2, \ldots, i_n) \in \mathbf{Z}^n$*, που ονομάζεται* **διάνυσμα επανάληψης** *(iteration vector).*

Κάθε συνιστώσα του διανύσματος επανάληψης αντιστοιχεί σε ένα από τους φωλιασμένους βρόχους, με την αριστερότερη συνιστώσα να αντιστοιχεί στον εξωτερικότερο βρόχο και την δεξιότερη συνιστώσα στον εσωτερικότερο βρόχο.

Ορισμός 2.2 *Το σύνολο των επαναλήψεων που ορίζεται από τα όρια των φωλιασμένων βρόχων:*

$$I^n = \{(i_1, i_2, \ldots, i_n) \mid i_j \in \mathbf{Z} \wedge l_j \leq i_j \leq u_j, 1 \leq j \leq n\} \subseteq \mathbf{Z}^n$$

αποτελεί ένα κυρτό πολύεδρο του $\mathbf{Z}^n$ *και ονομάζεται* **χώρος επαναλήψεων** *(iteration space).*

Ο χώρος επαναλήψεων δύναται να αναπαραστηθεί σε μορφή πίνακα. Λαμβάνοντας υπόψιν την σημασιολογία ενός βρόχου *for* κάθε μια από τις γραμμικές συναρτήσεις $l_{r,j}$ και $u_{r,j}$ παριστάνει μια ανισότητα της μορφής:

$$\sum_{k=1}^{r-1} a_{r,j}^k \cdot i_k + Y_{r,j} \quad \leq i_r \leq \quad \sum_{k=1}^{r-1} b_{r,j}^k \cdot i_k + F_{r,j}$$

Παίρνοντας μαζί όλες τις ανισότητες, σχηματίζεται η ακόλουθη ανισότητα πινάκων:

$$A \cdot \vec{I} \leq \beta$$

Κάθε γραμμή του πίνακα A ορίζει ένα κατώτερο όριο $l'_{r,j}$ (ή ένα ανώτερο όριο $u'_{r,j}$) και αποτελείται από τους συντελεστές $a^k_{r,j}$ και -1 (ή $b^k_{r,j}$ και 1). Οι n συνιστώσες του διανύσματος $\vec{I}$ είναι οι μεταβλητές ελέγχου i_r, ενώ οι συνιστώσες του διανύσματος β είναι οι συντελεστές $-Y_{r,j}$ (ή $F_{r,j}$).

Παράδειγμα Έστω το ακόλουθο σύστημα φωλιασμένων βρόχων:

```
for (i1=0; i1<=4; i1++)
    for (i1=0; i1<=6; i1++) {
        A[i1][i2]+=A[i1-1][i2]+D[i1][i2+1];
        D[i1][i2]+=D[i1-1][i2+1]+A[i1][i2];
    }
```

Τα όρια μπορούν να αναπαρασταθούν με τον ακόλουθο πίνακα ανισώσεων:

$$\begin{matrix} 0 \le i_1 \le 4 \\ 0 \le i_2 \le 6 \end{matrix} \quad \Rightarrow \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \le \begin{pmatrix} 4 \\ 6 \\ 0 \\ 0 \end{pmatrix}$$

Ορισμός 2.3 *Έστω τα σύνολα $A_1, A_2, \ldots, A_n$, το καρτεσιανό γινόμενο $A^n = A_1 \times A_2 \times \ldots \times A_n$ και τα διανύσματα $\vec{a} = (a_1, a_2, \ldots, a_n), \vec{b} = (b_1, b_2, \ldots, b_n)$. Η* **λεξικογραφική διάταξη**[1] *(lexicographic order) ορίζει ότι $\vec{a} \prec \vec{b}$ αν και μόνο αν:*

1. $a_1 \prec_1 b_1$, *ή*
2. $a_1 = b_1, a_2 = b_2, \ldots, a_k = b_k, a_{k+1} \prec_{k+1} b_{k+1} \quad , 1 \le k \le n-1$

Οι επαναλήψεις ενός συστήματος φωλιασμένων βρόχων εκτελούνται επομένως σε λεξικογραφική σειρά. Δηλαδή, εάν η επανάληψη $\vec{I_2}$ είναι λεξικογραφικά μεγαλύτερη από την $\vec{I_1}$ ($\vec{I_2} \succ \vec{I_1}$) τότε η επανάληψη $\vec{I_2}$ εκτελείται μετά την $\vec{I_1}$.

2.1.3 Εξαρτήσεις Δεδομένων

Οι *εξαρτήσεις δεδομένων* (data dependencies) χρησιμοποιούνται από τον μεταγλωττιστή για την αναπαράσταση των αναγκαίων περιορισμών στην διάταξη μεταξύ των εντολών ή της επαναχρησιμοποίησης των τιμών ενός προγράμματος. Μεταξύ δύο αναφορών υπάρχει εξάρτηση εάν υπάρχει ένα μονοπάτι ροής από την πρώτη αναφορά προς τη δεύτερη και οι δύο αναφορές προσπελάνουν την ίδια θέση μνήμης. Διακρίνονται τέσσερα είδη εξαρτήσεων:

- *Ανάγνωση Μετά από Εγγραφή* (Read After Write-RAW) ή αλλιώς *πραγματική εξάρτηση* (true dependence): εάν η πρώτη αναφορά εγγράφει στην θέση μνήμης και η δεύτερη διαβάζει από αυτή.
- *Εγγραφή Μετά από Ανάγνωση* (Write After Read-WAR) ή αλλιώς *αντι-εξάρτηση* (anti-dependence): εάν η πρώτη αναφορά διαβάζει από την θέση μνήμης και η δεύτερη εγγράφει σ' αυτήν.

[1] Με βάση τον ορισμό αυτό η λεξικογραφική διάταξη ορίζει μια μερική διάταξη.

- *Εγγραφή Μετά από Εγγραφή* (Write After Write-WAW) ή αλλιώς *εξάρτηση εξόδου* (output dependence): εάν η πρώτη αναφορά εγγράφει στην θέση μνήμης και η δεύτερη επίσης εγγράφει σ' αυτήν.

- *Ανάγνωση Μετά από Ανάγνωση* (Read After Read-RAR) ή αλλιώς *εξάρτηση εισόδου* (input dependence): εάν η πρώτη αναφορά διαβάζει από την θέση μνήμης και η δεύτερη επίσης διαβάζει από αυτή.

Όταν ένας κώδικας μετασχηματίζεται με σκοπό την βελτιστοποίηση είναι αναγκαία η διατήρηση της σχετικής διάταξης μεταξύ των εγγραφών και των αναγνώσεων στην ίδια θέση μνήμης, αλλιώς ο μετασχηματισμένος κώδικας θα παράγει λανθασμένα αποτελέσματε. Συγκεκριμένα θα πρέπει να διατηρούνται οι εξαρτήσεις τύπου RAW, WAR, WAW. Ο τέταρτος τύπος εξάρτησης, η RAR, δεν χρειάζεται να διατηρείται.

Ορισμός 2.4 *Ως* **διάνυσμα εξάρτησης** *dependence vector* $\vec{d} = (d_1, d_2, \ldots, d_n)$ *μιας επανάληψης ενός προγράμματος με φωλιασμένους βρόχους ορίζεται η διαφορά του διανύσματος* $\vec{I} = (i_1, i_2, \ldots, i_n)$ *που εκφράζει την συγκεκριμένη επανάληψη μείον το διάνυσμα* $\vec{I'} = (i_1, i_2, \ldots, i_n)$*, το οποίο εκφράζει μια επανάληψη, της οποίας τα αποτελέσματα χρησιμοποιούνται άμεσα κατά τους υπολογισμούς που πραγματοποιούνται από την* $\vec{I}$*.*

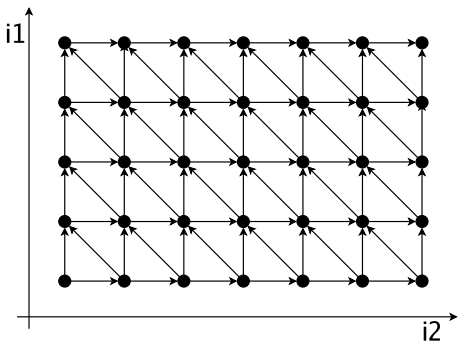
Ο βρόχος ο οποίος σχετίζεται με την αριστερότερη μη μηδενική απόσταση του διανύσματος εξάρτησης ονομάζεται *βρόχος μεταφοράς εξάρτησης* (carried loop), ενώ στην περίπτωση που όλες οι συνιστώσες του διανύσματος είναι μηδενικές τότε η εξάρτηση είναι *ανεξάρτητη βρόχων* (loop independent).

Συχνά για την αναπαράσταση του διανύσματος είναι αρκετό μόνο το πρόσημο των αποστάσεων. Με βάση τον συμβολισμό που προτείνει ο [18], χρησιμοποιείται το σύμβολο $+$ για εύρος απόστασης $[1, \infty]$, το σύμβολο $-$ για εύρος $[-\infty, -1]$, και το σύμβολο $\pm$ για άγνωστη απόσταση $[-\infty, \infty]$.

Για την αναπαράσταση των διανυσμάτων εξάρτησης χρησιμοποιειούνται συχνά δύο τρόποι: ο *πίνακας εξαρτήσεων* (dependence table) ή/και ο *γράφος εξάρτησης χώρου επαναλήψεων* (iteration space dependence graph). Ο πίνακας εξαρτήσεων των επαναλήψεων ενός συστήματος φωλιασμένων βρόχων είναι ένας πίνακας του οποίου οι στήλες αποτελούν το σύνολο των διανυσμάτων εξάρτησης των επαναλήψεων. Ο δε γράφος εξάρτησης χώρου επαναλήψεων παριστάνει τους περιορισμούς που δεν επιτρέπουν την αναδιάταξη των επαναλήψεων ενός συστήματος φωλιασμένων βρόχων. Εάν μια εντολή κάποιας επανάληψης I_2 εξαρτάται από μια οποιαδήποτε εντολή κάποιας άλλης επανάληψης I_1, δηλαδή υπάρχει εξάρτηση τύπου RAW, WAR, ή WAW, τότε η εξάρτηση αναπαριστάται με ένα βέλος από την επανάληψη I_1 προς την επανάληψη I_2.

Όπως αναφέρθηκε προηγουμένως οι επαναλήψεις ενός συστήματος φωλιασμένων βρόχων εκτελούνται σε λεξικογραφική διάταξη. Χρησιμοποιώντας την έννοια του διανύσματος εξάρτησης, η επανάληψη $\vec{I_2}$ εξαρτάται από την επανάληψη $\vec{I_1}$, και επομένως πρέπει να εκτελεστεί μετά από αυτήν, εάν για κάποιο διάνυσμα εξάρτησης $\vec{e}$ ισχύει $\vec{I_2} = \vec{I_1} + \vec{e}$. Βάση επομένως του ορισμού 2.3, αφού $\vec{I_2} \succ \vec{I_1}$, το $\vec{e}$ θα πρέπει να είναι λεξικογραφικά μεγαλύτερο από το $\vec{0}$, ή πιο απλά, *λεξικογραφικά θετικό*[2] (lexicographically positive).

[2]Δηλαδή ένα διάνυσμα είναι λεξικογραφικά θετικό αν η πρώτη μη μηδενική συνιστώσα του είναι θετική.



Σχήμα 2.2: Ο γράφος εξάρτησης χώρου επαναλήψεων του παραδείγματος

Πιο απλά, τα διανύσματα εξάρτησης που αφορούν ένα πρόγραμμα που βρίσκεται στην αρχική του μορφή είναι λεξικογραφικά θετικά, και ορίζουν μια μερική διάταξη μεταξύ των κόμβων του χώρου επαναλήψεων. Τέλος, οποιαδήποτε *τοπολογική διάταξη* (topological ordering)του γράφου εξάρτησης είναι έγκυρη, αρκεί να διατηρούνται όλες οι εξαρτήσεις, δηλαδή να παραμένουν λεξικογραφικά θετικές.

Παράδειγμα (συνέχεια) Για τον σύστημα φωλιασμένων βρόχων του παραδείγματος, τα διανύσματα εξάρτησης είναι τα ακόλουθα:

- $A[i_1][i_2] \rightarrow A[i_1-1][i_2]$: Εξάρτηση τύπου RAW με διάνυσμα εξάρτησης το $(i_1, i_2) - (i_1 - 1, i_2) = (1, 0) = (+, 0)$
- $D[i_1][i_2] \rightarrow D[i_1-1][i_2+1]$: Εξάρτηση τύπου RAW με διάνυσμα εξάρτησης το $(i_1, i_2) - (i_1 - 1, i_2 + 1) = (1, -1) = (+, -)$
- $A[i_1][i_2] \rightarrow A[i_1][i_2]$: Εξάρτηση τύπου RAW με διάνυσμα εξάρτησης το $(i_1, i_2) - (i_1, i_2) = (0, 0)$
- $D[i_1][i_2+1] \rightarrow D[i_1][i_2]$: Εξάρτηση τύπου WAR με διάνυσμα εξάρτησης το $(i_1, i_2 + 1) - (i_1, i_2) = (0, 1) = (0, +)$
- $A[i_1][i_2] \rightarrow A[i_1 - 1][i_2]$: Εξάρτηση τύπου RAR με διάνυσμα εξάρτησης το $(i_1, i_2) - (i_1 - 1, i_2) = (1, 0) = (+, 0)$

Παρατηρούμε ότι όλες οι εξαρτήσεις είναι λεξικογραφικά θετικές. Οι δε τέσσερεις πρώτες εξαρτήσεις αποτελούν εξαρτήσεις που πρέπει να διατηρηθούν, ενώ η διατήρηση της πέμπτης εξάρτησης τύπου RAR δεν είναι υποχρεωτική. Τα διανύσματα εξάρτησης παριστάνονται από τον γράφο εξάρτησης χώρου επαναλήψεων στο σχήμα 2.2. Τέλος, ο πίνακας εξαρτήσεων είναι ο:

$$D = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 \end{pmatrix}$$

2.2 Τοπικότητα και Επαναχρησιμοποίηση

2.2.1 Διάκριση Τοπικότητας και Επαναχρησιμοποίησης

Είναι πολυ σημαντικό να διακρίνουμε τη διαφορά μεταξύ *τοπικότητας* (locality) και *επαναχρησιμοποίησης* (reuse) [18]. Λέμε ότι ένα στοιχείο *επαναχρησιμοποιείται* όταν το ίδιο στοιχείο χρησιμοποιείται σε πολλαπλές επαναλήψεις ενός συστήματος φωλιασμένων βρόχων. Η επαναχρησιμοποίηση λοιπόν είναι έμφυτη σε μια υπολογιστική διαδικασία και δεν εξαρτάται από τον συγκεκριμένο τρόπο που οι βρόχοι είναι γραμμένοι. Επομένως, η ύπαρξη επαναχρησιμοποίησης των δεδομένων δεν σημαίνει και την ελαχιστοποίση των αναφορών στη μνήμη, όταν μεταξύ των επαναχρησιμοποιήσεων τα δεδομένα διώχνονται από την κρυφή μνήμη λόγω αντικαταστάσεων που προκαλούν νέα δεδομένα που εισέρχονται σε αυτή. Στην περίπτωση όμως, που το επαναχρησιμοποιήσιμο στοιχείο παραμένει στην κρυφή μνήμη και έχουμε επιτυχία κρυφής μνήμης στην νέα αναφορά που γίνεται πάνω στο στοιχείο αυτό, τότε λέμε ότι η αναφορά έχει τοπικότητα.

Για να γίνει περισσότερη κατανοητή η διαφορά, έστω το ακόλουθο τμήμα του κώδικα ενός προγράμματος:

```
for (i=0; i<N; i++) {
    for (j=0; jN<; j++)
        A[i]+=A[j]*A[i];
}
```

Η αναφορά $A[j]$ προσπελάνει διαφορετικά δεδομένα κατα τις επαναλήψεις του εσωτερικού βρόχου, αλλά επαναχρησιμοποιεί τα ίδια δεδομένα κατά τις επαναλήψεις του εξωτερικού βρόχου. Ακριβέστερα, το δεδομένο $A[j]$ χρησιμοποιείται στις επαναλήψεις $\{(i,j)|0 \leq i < N\}$ κι επομένως υπάρχει επαναχρησιμοποίηση. Μεταξύ όμως δύο επαναχρησιμοποιήσεων υπάρχουν $N-1$ αναφορές σε άλλα δεδομένα. Άρα, στην περίπτωση που το N είναι αρκετά μεγάλο, έτσι ώστε τα δεδομένα να διώχνονται από την κρυφή μνήμη πριν επαναχρησιμοποιηθούν, δεν υπάρχει τοπικότητα.

2.2.2 Εύρεση Επαναχρησιμοποίησης

Η εύρεση σε ποιες περιοχές του χώρου επαναλήψεων υπάρχει επαναχρησιμοποίηση είναι αρκετά βοηθητική για την εφαρμογή των κατάλληλων μετασχηματισμών βρόχων, έτσι ώστε η επαναχρησιμοποίηση να μετατραπεί σε όσο το δυνατόν περισσότερη τοπικότητα. Η επαναχρησιμοποίηση χωρίζεται σε τέσσερις κατηγορίες: αυτο-χρονική, αυτο-χωρική, ομαδο-χρονική και ομαδο-χωρική επαναχρησιμοποίηση.

Πριν προχωρήσουμε στις τέσσερις κατηγορίες επαναχρησιμοποίησης ακολουθεί ένας χρήσιμος ορισμός.

Ορισμός 2.5 *Σε ένα σύστημα φωλιασμένων βρόχων βάθους n, μια αναφορά στον m-διάστατο πίνακα A μπορεί να αναπαρασταθεί με τη διανυσματική συνάρτηση $\vec{f}_A(i_1, i_2, \ldots, i_n) = U \cdot \vec{I} + \vec{u}$, όπου ο πίνακας U διαστάσεων $m \times n$ ονομάζεται πίνακας αναφοράς και το m-διάστατο διάνυσμα $\vec{u}$ ονομάζεται διάνυσμα απόκλισης. Η διανυσματική συνάρτηση $\vec{f}_A$ θα ονομάζεται διάνυσμα αναφοράς.*

Ακολούθως, παρουσιάζονται περιληπτικά οι τέσσερις αυτές κατηγορίες και πως μπορεί να γίνει η εύρεση της κάθε επαναχρησιμοποίησης, όπως αναφέρεται στο [18].

Αυτο-χρονική Επαναχρησιμοποίηση Η *αυτο-χρονική* (self-temporal reuse) εμφανίζεται όταν μια αναφορά ενός συστήματος φωλιασμένων βρόχων προσπελάνει το ίδιο δεδομένο σε διαφορετικές επαναλήψεις. Χρησιμοποιώντας τον ορισμό 2.5, αυτο-χρονική επαναχρησιμοποίηση μεταξύ των επαναλήψεων $\vec{I}_1$ και $\vec{I}_2$ υπάρχει όταν ισχύει $U \cdot \vec{I}_1 + \vec{u} = \cdot\vec{I}_2 + \vec{u}$, δηλαδή όταν $U \cdot (\vec{I}_1 - \vec{I}_2) = \vec{0}$. Λέμε ότι υπάρχει επαναχρησιμοποίηση στην κατεύθυνση $\vec{r}$ (*διάνυσμα επαναχρησιμοποίησης*-τεξτλατινρευσε εστορ) όταν $U \cdot \vec{r} = \vec{0}$. Η λύση της εξίσωσης αποτελεί το διανυσματικό χώρο αυτο-χρονικής επαναχρησιμοποίσης: $R_{ST} = kerU \subset \mathbf{R}^n$.

Αυτο-χωρική Επαναχρησιμοποίηση Η *αυτο-χωρική* (self-spatial reuse) εμφανίζεται όταν μια αναφορά ενός συστήματος φωλιασμένων βρόχων προσπελάνει την ίδια ενότητα κρυφής μνήμης σε διαφορετικές επαναλήψεις. Δηλαδή, η αυτο-χωρική επαναχρησιμοποίηση αποτελεί γενίκευση της αυτο-χρονικής. Έστω, ότι τα δεδομένα του πίνακα U αποθηκεύονται παράλληλα με την k-διάσταση. Τότε, για να υπάρχει στην αναφορά $U[L^U \cdot \vec{I} + \vec{b}^U]$ αυτο-χωρική επαναχρησιμοποίηση, θα πρέπει οι συντεταγμένες όλων των διαστάσεων πλην της k-διάστασης να ταυτίζονται. Αντικαθιστώντας την k-γραμμή του πίνακα L^U με μηδενικά, παίρνουμε τον πίνακα L^U_S. Τότε, ο διανυσματικός χώρος αυτο-χωρικής επαναχρησιμοποίσης είναι ο: $R_{SS} = kerL^U_S \subset \mathbf{R}^n$.

Ομαδο-χρονική Επαναχρησιμοποίηση Η *ομαδο-χρονική* (group-temporal reuse) εμφανίζεται όταν οι αναφορές ενός συστήματος φωλιασμένων βρόχων προσπελάνουν το ίδιο δεδομένο σε διαφορετικές επαναλήψεις. Έστω, ότι οι αναφορές διαφέρουν το πολύ στον σταθερό όρο $\vec{u}$, δηλαδή $U \cdot \vec{I} + \vec{u}_1$ και $L^U \cdot \vec{I} + \vec{u}_2$. Τότε, υπάρχει ομαδο-χρονική επαναχρησιμοποίηση μεταξύ των δύο αυτών αναφορών αν και μόνο αν $\exists \vec{r} : L^U \cdot \vec{r} = \vec{u}_1 - \vec{u}_2$, όπου $|\vec{u}_1 - \vec{u}_2| <$(*μήκος ενότητας κρυφής μνήμης/μέγεθος στοιχείου*). Λύνοντας την εξίσωση αυτή προκύπτει το διάνυσμα $\vec{r}_p$, αν υπάρχει τέτοιο. Ο διανυσματικός χώρος αυτο-χρονικής επαναχρησιμοποίσης προκύπτει ως: $R_{GT} = kerU + [\vec{r}_p]$.

Ομαδο-χωρική Επαναχρησιμοποίηση Η *ομαδο-χωρική* (group-spatial reuse) εμφανίζεται όταν οι αναφορές ενός συστήματος φωλιασμένων βρόχων προσπελάνουν την ίδια ενότητα κρυφής μνήμης σε διαφορετικές επαναλήψεις. Δηλαδή, η ομαδο-χωρική επαναχρησιμοποίηση αποτελεί γενίκευση της ομαδο-χρονικής. Έστω, ότι οι αναφορές διαφέρουν το πολύ στον σταθερό όρο $\vec{u}$, δηλαδή $U \cdot \vec{I} + \vec{u}_1$ και $U \cdot \vec{I} + \vec{u}_2$. Αντικαθιστώντας την k-γραμμή του πίνακα U και των διανυσμάτων u_1, u_2 με μηδενικά, παίρνουμε τα U_S, $u_{S,1}$ και $u_{S,2}$. Τότε, υπάρχει ομαδο-χωρική επαναχρησιμοποίηση μεταξύ των δύο αυτών αναφορών αν και μόνο αν $\exists \vec{r} : U_S \cdot \vec{r} = \vec{u}_{S,1} - \vec{u}_{S,2}$, όπου $|\vec{u}_1 - \vec{u}_2| <$(*μήκος ενότητας κρυφής μνήμης/μέγεθος στοιχείου*). Ο διανυσματικός χώρος αυτο-χρονικής επαναχρησιμοποίσης προκύπτει ως: $R_{GS} = kerU_S + [\vec{r}_p]$.

2.3 Unimodular Μετασχηματισμοί Βρόχων

Ορισμός 2.6 *Ένας* **μετασχηματισμός βρόχων** T *αντιστοιχίζει κάθε επανάληψη* $\vec{I} = (i_1, i_2, \ldots, i_n)$ *του* n*-διάστατου χώρου επαναλήψεων στην επανάληψη* $\vec{J} = (j_1, j_2, \ldots, j_n)$ *του επίσης* n*-διάστατου μετασχηματισμένου χώρου επαναλήψεων:*

$$J^n = \{\vec{J} = T \cdot \vec{I} | \vec{I} \in \mathbf{Z}^n\}$$

Τα όρια του n-διάστατου μετασχηματισμένου χώρου επαναλήψεων βρίσκονται με την εφαρμογή του πίνακα μετασχηματισμού T επί της ανισότητας πινάκων που παριστάνει τα όρια του αρχικού n-διάστατου χώρου επαναλήψεων. Δηλαδή τα όρια του μετασχηματισμένου χώρου επαναλήψεων βρίσκονται ως εξής[3]:

$$\left.\begin{array}{l} A \cdot \vec{I} \leq \beta \\ \vec{J} = T \cdot \vec{I} \\ det(T) = \pm 1 \neq 0 \end{array}\right\} \Rightarrow A \cdot T^{-1} \cdot \vec{J} \leq \beta \tag{2.1}$$

Ορισμός 2.7 *Ένας* **unimodular μετασχηματισμός βρόχων**[4] *παριστάνεται με ένα πίνακα T, για τον οποίο $det(T) = \pm 1$. Ο συνδυασμός unimodular μετασχηματισμών δίνει ένα επίσης unimodular μετασχηματισμό και ο πίνακας του νέου μετασχηματισμού προκύπτει από το γινόμενο των επιμέρους μετασχηματισμών.*

Οι μετασχηματισμοί βρόχων είναι χρήσιμοι όταν διατηρούν τις εξαρτήσεις δεδομένων μετά την εφαρμογή τους σε ένα σύστημα φωλιασμένων βρόχων. Το ακόλουθο θεώρημα αφορεί τους unimodular μετασχηματισμούς βρόχων και είναι πολύ σημαντικό.

Θεώρημα 2.1 *Έστω D το σύνολο των διανυσμάτων εξάρτησης ενός συστήματος φωλιασμένων βρόχων. Ένας unimodular μετασχηματισμός βρόχων T είναι* έγκυρος *(legal) αν και μόνο αν $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.*

Ακολουθεί η περιγραφή τριών βασικών unimodular μετασχηματισμών βρόχων: η ανταλλαγή, η μετάθεση, η αναστροφή και η περιστροφή.

1. Η *ανταλλαγή* (interchange) δύο βρόχων μετατρέπει το διάνυσμα επανάληψης (i, j) στο διάνυσμα (j, i). Ο πίνακας του μετασχηματισμού είναι ο:

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Από το συνδυασμό διαδοχικών ανταλλαγών προκύπτει η πιο γενική μορφή του μετασχηματισμού, η *μετάθεση* (permutation), η οποία μεταθέτει τη διάταξη των βρόχων επανάληψης. Για παράδειγμα, ο πίνακας μετασχηματισμού για την μετάθεση του διανύσματος επανάληψης $\vec{I} = (i, j, k)$ στο διάνυσμα $\vec{J} = (k, j, i)$ δίνεται από τον:

$$T = T_3 T_2 T_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Τόσο η μετάθεση όσο και η ανταλλαγή χρησιμοποιούνται για να τροποποιήσουν την διάταξη των βρόχων έτσι ώστε να προσπελάνουν τα δεδομένα με τη διάταξη που αυτά είναι αποθηκευμένα, βελτιώνοντας με αυτό τον τρόπο την χωρική τοπικότητα. Από τη στιγμή που τροποποιούν τη διάταξη των βρόχων κι επομένως τη σειρά εκτέλεσης των επαναλήψεων, αυτό σημαίνει ότι η εφαρμογή τους δεν είναι πάντοτε έγκυρη. Ιδιαίτερα χρήσιμο είναι το επόμενο θεώρημα.

[3] Η λύση των συστημάτων ανισώσεων που προκύπτουν παρότι είναι πολύ σημαντική, είναι εκτός του αντικειμένου της παρούσας εργασίας και γι' αυτό δεν θα ασχοληθούμε περαιτέρω.

[4] Οι unimodular μετασχηματισμοί αποτελούν μια ειδική κατηγορία γραμμικών μετασχηματισμών.

Θεώρημα[18] 2.2 *Σε ένα σύστημα φωλιασμένων βρόχων, οι βρόχοι l_i μέχρι l_j είναι* πλήρως μεταθέσιμοι *(fully permutable) αν και μόνο αν όλα τα διανύσματα εξάρτησης είναι όλα λεξικογραφικά θετικά. Επιπλέον για κάθε διάνυσμα ισχύει ότι, είτε το $(d_1, \ldots, d_{i-1})$ είναι λεξικογραφικά θετικό, είτε οι συνιστώσες $d_i, \ldots, d_j$ είναι μη αρνητικές.*

2. Η *αναστροφή* (reversal) ενός βρόχου πραγματοποιείται με πολλαπλασιασμό με -1 του αντίστοιχου δείκτη. Για παράδειγμα σε ένα σύστημα φωλιασμένων βρόχων με βάθος 2, οι πίνακες μετασχηματισμού για κάθε μια μεταβλητή είναι οι:

$$T_{rev,i} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \qquad T_{rev_j} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

3. Η *περιστροφή* skewing ενός βρόχου παριστάνεται από τον πίνακα μετασχηματισμού T_{skew}, ο οποίος προσθέτει κάποιο ακέραιο πολλαπλάσιο ενός δείκτη βρόχου σε έναν άλλο δείκτη:

$$T_{skew} = \begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix}$$

Η περιστροφή είναι πάντοτε έγκυρη. Παρόλο που επηρεάζει τα διανύσματα εξάρτησης, η περιστροφή δεν αλλάζει καθόλου την σειρά εκτέλεσης των επαναλήψεων. Έτσι, αν και γενικά δεν βελτιώνει την επίδοση, μπορεί να επιτρέψει την εφαρμογή άλλων μετασχηματισμών των οποίων η εφαρμογή προηγουμένως δεν ήταν έγκυρη, λ.χ. η μετατροπή ενός μη-πλήρως μεταθέσιμου συστήματος φωλιασμένων βρόχων σε πλήρως μεταθέσιμο.

Παράδειγμα (συνέχεια) Συνεχίζοντας το παράδειγμα μας έστω ότι επιθυμούμε να εφαρμόσουμε ανταλλαγή (ή μετάθεση) στον κώδικα μας. Η εφαρμογή όμως απευθείας της ανταλλαγής δεν είναι έγκυρη βάση του θεωρήματος 2.1. Πράγματι εφαρμόζοντας τον μετασχηματισμό της ανταλλαγής σε ένα από τα διανύσματα εξάρτησης, το επαληθεύουμε:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \prec 0$$

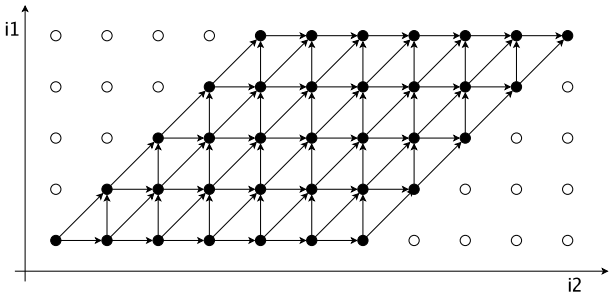
Για να μπορέσουμε λοιπόν να εφαρμόσουμε την ανταλλαγή, εφαρμόζουμε πρώτα την περιστροφή:

$$\vec{I}_{new} = \begin{pmatrix} i_{1,new} \\ i_{2,new} \end{pmatrix} = T_{skew}\vec{I} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} i_{1,new} \\ i_{2,new} \end{pmatrix} = \begin{pmatrix} i_{1,new} \\ i_{2,new} - i_{1,new} \end{pmatrix}$$

και παίρνουμε τον ακόλουθο κώδικα:

```
for (i1=0; i1<=4; i1++)
    for (i2=i1; i2<=6+i1; i2++) {
        A[i1][i2-i1]+=A[i1-1][i2-i1]+D[i1][i2-i1+1];
        D[i1][i2-i1]+=D[i1-1][i2-i1+1]+A[i1][i2-i1];
    }
```



Σχήμα 2.3: Ο γράφος εξάρτησης χώρου επαναλήψεων μετά την εφαρμογή του μετασχηματισμού της περιστροφής

του οποίου τα διανύσματα εξάρτησης παριστάνονται στο σχήμα 2.3. Ακολούθως, εφαρμόζουμε τον μετασχηματισμό της ανταλλαγής ο οποίος είναι έγκυρος με την νέα διάταξη που έδωσε η περιστροφή:

$$\vec{I}_{new} = \begin{pmatrix} i_{1,new} \\ i_{2,new} \end{pmatrix} = T_{int}\vec{I} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} i_{1,new} \\ i_{2,new} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}^{-1} \begin{pmatrix} i_{1,new} \\ i_{2,new} \end{pmatrix} = \begin{pmatrix} i_{2,new} \\ i_{1,new} \end{pmatrix}$$

και παίρνουμε τον ακόλουθο κώδικα:

```
for (i1n=0; i1n<=6; i1n++)
    for (i2n=i1n; i2n<=4+i1n; i2n++) {
        A[i2n][i1n-i2n]+=A[i2n-1][i1n-i2n]+D[i2n][i1n-i2n
            +1];
        D[i2n][i1n-i2n]+=D[i2n-1][i1n-i2n+1]+A[i2n][i1n-
            i2n];
    }
```

2.4 Μη Γραμμικοί Μετασχηματισμοί Βρόχων

Ορισμένοι σημαντικοί μετασχηματισμοί βρόχων οι οποίοι είναι μη γραμμικοί είναι η συγχώνευση, η διάσπαση, το scalar replacement, το ξεδίπλωμα βρόχου, ο strip mining και η πλακόστρωση:

1. Η *συγχώνευση βρόχων* (loop fussion) σκοπό έχει την βελτίωση της χρονικής τοπικότητας συνδυάζοντας βρόχους οι οποίοι σε διαφορετικά σημεία του προγράμματος προσπελάνουν τις ίδιες θέσεις μνήμης. Ακολουθεί ένα παράδειγμα μιας τέτοιας συγχώνευσης:

```
for (i=2; i<=N; i++) {
    for (j=1; j<=N; j++)
        X[i][j]-=X[i-1][j]*A[i][j]/B[i-1][j]
```

```
    for (j=1; j<=N; j++)
        B[i][j]-=A[i][j]*A[i][j]/B[i-1][j]
}
```

⇓ loop fussion ⇓

```
for (i=2; i<=N; i++) {
    for (j=1; j<=N; j++) {
        X[i][j]-=X[i-1][j]*A[i][j]/B[i-1][j]
        B[i][j]-=A[i][j]*A[i][j]/B[i-1][j]
    }
}
```

Η κοινή χρήση των πινάκων A και B μαρτυρά υπέρ της συγχώνευσης, έτσι ώστε τα δεδομένα των πινάκων A και B να μην χρειαστεί να αποθηκευθούν στην κρυφή μνήμη δύο φορές.

2. Η *διάσπαση βρόχων* (loop fission) μπορεί να βελτιώσει την τοπικότητα προγραμμάτων που περιέχουν μεγάλο πλήθος εντολών και μεταβλητών, διασπώντας ένα σύστημα φωλιασμένων βρόχων σε επιμέρους μικρότερα συστήματα, που το καθένα χρειάζεται μικρότερο και ξεχωριστό όγκο δεδομένων. Κάθε σύστημα επομένως μπορεί να δουλέψει με τον μικρότερο όγκο δεδομένων τα οποία χωρούν στην κρυφή μνήμη, χωρίς αυτά να απομακρυνθούν πριν τελειώσει ο μικρότερος σε έργο υπολογισμός.

3. Το *ξεδίπλωμα βρόχου* (inner unrolling) αντικαθιστά το σώμα του εσωτερικότερου βρόχου σε ένα σύστημα φωλιασμένων βρόχων με δύο ή περισσότερα αντίγραφα του σώματος και τροποποιώντας κατάλληλα τον έλεγχο ορίων του βρόχου. Με τον τρόπο αυτό αυξάνεται η παραλληλία σε επίπεδο εντολής, αφού υπάρχουν περισσότερες εντολές προς εκτέλεση ανά επανάληψη και επίσης μικραίνει η χρονική επιβάρυνση που οφείλεται στον έλεγχο ορίων, αφού γίνονται λιγότεροι ελέγχοι.

4. Η *ανάθεση μεταβλητών σε καταχωρητές scalar replacement* βελτιώνει την τοπικότητα των δεδομένων, αλλά στο επίπεδο των καταχωρητών. Σε ένα σύστημα φωλιασμένων βρόχων οι μεταβλητές εκείνες που χρησιμοποιούνται συχνότερα, αποθηκεύονται σε καταχωρητές.

5. Ο strip mining χρησιμοποιείται για την διαμέριση του χώρου επαναλήψεων σε λουρίδες. Για να το πετύχει αυτό, αποσυνθέτει ένα βρόχο σε δύο φωλιασμένους βρόχους ως εξής:

```
for (i=lb; i<=up; i++) {
    /* loop body */
}
```

⇓ strip mining ⇓

```
for (ii=floor((lb-oft_ii)/step)*step+oft_ii; i<=up;
        i+=step)
    for (i=MAX(ii,lb); i<=MIN(ii+step-1,up); i++) {
    /* loop body */
}
```

Το *step* είναι το μέγεθος της λουρίδας, και το $oft_i i$ ($0 \leq oft_{ii} < step$) καθορίζει την αρχή της πρώτης λουρίδας. Ο εξωτερικός βρόχος ii μετακινείται μεταξύ των λουρίδων, ενώ ο εσωτερικός βρόχος i διατρέχει τις επαναλήψεις που αντιστοιχούν στην τρέχουσα λουρίδα.

6. Η *πλακόστρωση* (tiling) είναι ένας πολύ δημοφιλής και σημαντικός μετασχηματισμός και γι' αυτό παρουσιάζεται ξεχωριστά στην ενότητα 2.5.

2.5 Μετασχηματισμός Tiling

2.5.1 Γενικά

Ο μετασχηματισμός *tiling* αποτελεί ένα από τους πιο δημοφιλείς μετασχηματισμούς, ο οποίος εκτός από τις εφαρμογές του στο χώρο της παράλληλης επεξεργασίας, χρησιμοποιείται και για την αύξηση της τοπικότητας στην ιεραρχία μνήμης των μικροεπεξεργαστών, ειδικά όταν οι αναφορές αφορούν πίνακες μεγάλων διαστάσεων οι οποίοι δεν χωρούν ολόκληροι στα χαμηλά επίπεδα της ιεραρχίας.

Σε ένα σύστημα φωλιασμένων βρόχων όπου εφαρμόσθηκε tiling, αντίς να προσπελάνονται ολόκληρες γραμμές ή στήλες ενός πίνακα, οι βρόχοι εργάζονται πάνω σε υποπίνακες, που ο κάθενας χωρεί ολόκληρος στο επίπεδο της ιεραρχίας μνήμης που ενδιαφέρει. Έτσι, αυξάνεται το πλήθος των προσπελάσεων που γίνονται στα δεδομένα πριν αυτά απομακρυνθούν από το επίπεδο αυτό.

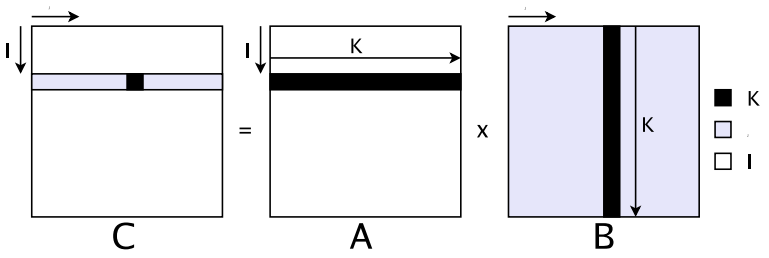
Η σημασία του μετασχηματισμού tiling επιδεικνύεται με το ακόλουθο παράδειγμα.

Παράδειγμα Ο ακόλουθος κώδικας αποτελεί την αρχική μορφή του πολλαπλασιασμού δυο πινάκων:

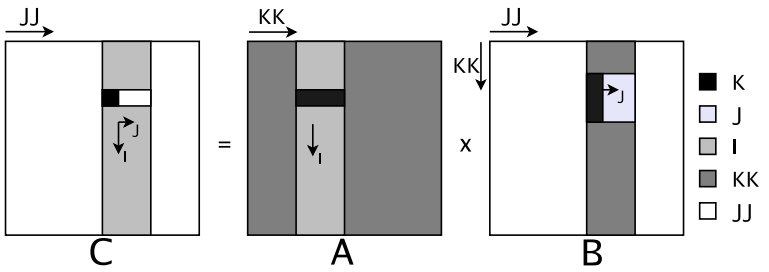
```
for (i=0; iN<; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j]+=A[i][k]*B[k][j];
```

Μετά την εφαρμογή του μετασχηματισμού tiling προκύπτει ο ακόλουθος κώδικας:

```
for (jj=0; jj<N; jj+=step)
    for (kk=0; kk<N; kk+=step)
        for (i=0; iN<; i++)
            for (j=jj; j<MIN(N,jj+step); j++)
                for (k=kk; k<MIN(N,kk+step); k++)
                    C[i][j]+=A[i][k]*B[k][j];
```



(α′) κανονική μορφή



(β′) tiled μορφή

Σχήμα 2.4: Προσπέλαση δεδομένων κατά την εκτέλεση του πολλαπλασιασμού πίνακα

Μελετώντας τον τρόπο που ο αρχικός κώδικας προσπελάνει τα δεδομένα, δες σχήμα 2.4(α′), παρατηρούμε ότι το στοιχείο $C[i][j]$ επαναχρησιμοποιείται από όλες τις επαναλήψεις του βρόχου k. Η δε γραμμή του A που προσπελάνεται από τον εσωτερικότερο βρόχο k, επαναχρησιμοποιείται από την επόμενη επανάληψη του j και παρόμοια η ίδια στήλη του B επαναχρησιμοποιείται από τον εξωτερικότερο βρόχο i. Στην χειρότερη περίπτωση που η κρυφή μνήμη δεν μπορεί να χωρέσει ούτε μια γραμμή του πίνακα A, τότε η γραμμή του A δεν επαναχρησιμοποιείται κι επομένως $2N^3 + N^2$ λέξεις δεδομένων πρέπει να διαβαστούν από την κύρια μνήμη σε N^3 επαναλήψεις. Η μετασχηματισμένη όμως μορφή προσπελάνει μικρά τετράγωνα τμήματα του πίνακα D κάθε φορά, δες σχήμα 2.4(β′), διαστάσεων $step \times step$. Εάν λοιπόν οι διαστάσεις του τμήματος επιλεγούν έτσι ώστε να χωρεί στην κρυφή μνήμη, τότε οι λέξεις που προσπελάνονται μειώνονται στις $(2N^3/step) + N^2$, δηλαδή έχουμε βελτίωση $size$ φορές.

2.5.2 Εφαρμόζοντας τον Μετασχηματισμό Tiling

Εγκυρότητα Μετασχηματισμού

Ο μετασχηματισμός *tiling* στη συνηθισμένη περίπτωση εφαρμογής του σε ένα μόνο επίπεδο, αυξάνει το βάθος n ενός συστήματος φωλιασμένων βρόχων σε βάθος που κυμαίνεται από $n + 1$ μέχρι $2n$, ανάλογα με το πόσοι βρόχοι πλακοστρώνονται. Η πλακόστρωση ενός ή περισσότερων βρόχων αλλάζει τη σειρά διάσχυσης του χώρου επαναλήψεων, έτσι ώστε να αποτελείται από μικρότερα πολύεδρα τα οποία εκτελούνται στη σειρά.

Παράδειγμα Για το ακόλουθο σύστημα φωλιασμένων βρόχων βάθους 2, η σειρά διάσχυσης τροποποιείται όπως φαίνεται στο σχήμα 2.5.

Αρχική μορφή

```
for (i=0; iN<; i++)
    for (j=0; j<N; j++)
        /* loop body */
```

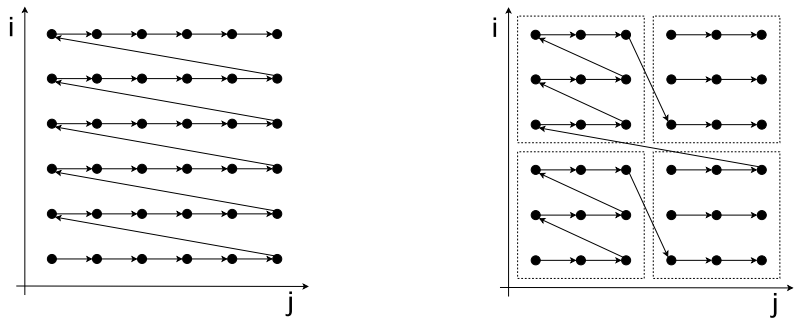
Tiled μορφή

```
for (ii=0; iiN<; ii++)
    for (jj=0; jj<N; jj++)
        for (i=ii; i<MINN(, jj+size); i++)
            for (j=jj; j<MIN(N, ii+size); j++)
                /* loop body */
```

Αφού ο μετασχηματισμός tiling τροποποιεί τη σειρά διάσχυση του χώρου επαναλήψεων, αυτό σημαίνει ότι η εφαρμογή του δεν είναι πάντοτε έγκυρη.

Θεώρημα[18] 2.3 *Σε ένα σύστημα φωλιασμένων βρόχων, οι βρόχοι l_i μέχρι l_j μπορούν να μετασχηματιστούν με tiling, αν είναι πλήρως μεταθέσιμοι (θεώρημα 2.2).*

Υπενθυμίζουμε ότι η μετατροπή ενός μη-πλήρως μεταθέσιμου συστήματος φωλιασμένων βρόχων σε πλήρως μεταθέσιμο, μπορεί να γίνει με χρήση της περιστροφής.



(α′) αρχική μορφή (β′) tiled μορφή

Σχήμα 2.5: Αλλαγή της σειράς εκτέλεσης των επαναλήψεων ενός συστήματος φωλιασμένων βρόχων βάθους n μετά την εφαρμογή του tiling

.

Επιλογή Παραμέτρων

Για την μεγιστοποίηση της τοπικότητας των δεδομένων, που αυτός άλλωστε είναι κι ο σκοπός του μετασχηματισμού, πρέπει να επιλυθούν τα δύο ερωτήματα που αφορούν τις παραμέτρους του μετασχηματισμού. Η λύση τους καθοδηγείται από την ανάλυση και εύρεση της επαναχρησιμοποίησης (ενότητα 2.2) η οποία πρέπει να προηγηθεί. Το πρώτο ερώτημα αφορά ποιοι είναι οι βρόχοι που πρέπει να πλακοστρωθούν και ποια είναι η σχετική διάταξη των βρόχων αυτών, έτσι ώστε να υπάρξει όσο το δυνατόν περισσότερη εκμετάλλευση της επαναχρησιμοποίησης και επομένως βελτίωση της τοπικότητας των δεδομένων.

Το δεύτερο ερώτημα και το πιο ευαίσθητο για την απόδοση του μετασχηματισμού είναι το μεγέθους του tile. Επιλέγοντας το βέλτιστο μέγεθος, κάθε τμήμα της υπολογιστικής διαδικασίας θα ολοκληρώνεται, χωρίς αστοχίες κρυφής μνήμης, εκτός από τις υποχρεωτικές. Βασικο ρόλο για την επιλογή του μεγέθους του tile παίζει η οργάνωση της κρυφής μνήμης. Για παράδειγμα, μια κρυφή μνήμη ευθείας αντιστοίχησης, παρόλο που μπορεί να είναι αρκετά μεγάλη για να χωρεί τα δεδομένα που χρησιμοποιούνται, δεν είναι ικανή να απαλείψει τις αστοχίες λόγω συγκρούσεων μεταξύ των στοιχείων του ίδιου πίνακα ή διαφορετικών πινάκων όπως αναφέρεται στο [10]. Σε μια τέτοια περίπτωση το μέγεθος του tile ίσως να πρέπει να επιλεγεί αρκετά μικρό για να αποφεχθούν τέτοιες συγκρούσεις. Στο [10] επίσης αναφέρεται ότι η λογική να επιλέγεται ένα σταθερό μέγεθος για το tile είναι λανθασμένη και καταστροφική. Το μέγεθος πρέπει να είναι συναρτήσει τόσο του μεγέθους της κρυφής μνήμης, όσο και του μεγέθους του προβλήματος, αλλιώς σε κρυφές μνήμες μικρής συσχέτισης, εκτός από τις αστοχίες λόγω χωρητικότητας παρατηρούνται και αστοχίες λόγω συκγρούσεων μεταξύ στοιχείων του ίδιου πίνακα.

Επίσης, για την βελτίωση της τοπικότητας στα διάφορα επίπεδα της ιεραρχίας μνήμης, θα πρέπει να εφαρμόζεται πολλαπλό tiling. Αυτό σημαίνει, αναδρομική εφαρμογή του tiling σε κάθε επίπεδο, διαιρώντας με αυτό τον τρόπο ένα tile σε μικρότερα tiles. Η βελτιστοποίηση βέβαια του πολλαπλού tiling δεν είναι τόσο απλή, λόγω των αλληλεπιδράσεων μεταξύ των διαφόρων επιπέδων. Μια πρακτική μέθοδος για την εφαρμογή πολλαπλού tiling, είναι η

διαδοχική σειριακή εφαρμογή του tiling, έτσι ώστε οι κατευθύνσεις των tiles[5] που προκύπτουν σε διπλανά επίπεδα της ιεραρχίας, να είναι διαφορετικές. Πλεονέκτημα αυτής της μεθόδου είναι ότι για την συνολική βελτιστοποίηση του tiling, αρκεί η βελτιστοποίηση του κάθε επιπέδου ξεχωριστά.

Παραγωγή Κώδικα

Αφού επιλυθούν τα δύο αυτά ζητήματα και επιλεγούν οι δύο βασικοί παράμετροι, διάταξη βρόχων και μέγεθος tile, αυτό που απομένει είναι η εφαρμογή του μετασχηματισμού για την παραγωγή του κώδικα. Συνήθως ο μετασχηματισμός του tiling διαμερίζει το χώρο επαναλήψεων σε tiles χρησιμοποιώντας δύο γνωστούς μετασχηματισμούς, strip-mining και μετάθεση, που και οι δυο περιγράφονται στην ενότητα 2.4.

Για την διαμέριση του χώρου επαναλήψεων σε tiles, οι δύο μετασχηματισμοί (strip mining, μετάθεση) εφαρμόζονται διαδοχικά, τόσες φορές όσο το πλήθος των διαστάσεων που πρόκειται να διαμεριστούν. Αρχικά εφαρμόζεται μετάθεση, έτσι ώστε ο βρόχος που πρόκειται να διαμεριστεί να γίνει ο εξωτερικότερος. Ακολούθως για κάθε βρόχο που πρόκειται να διαμεριστεί, χρησιμοποιείται ο strip mining για την διαμέριση του και στη συνέχεια η μετάθεση για την αναδιάταξη των εσωτερικών βρόχων που διατρέχουν το χώρο επαναλήψεων, έτσι ώστε ο επόμενος βρόχος που πρόκειται να διαμεριστεί να γίνει ο εξωτερικότερος αυτών. Αφού διαμεριστούν όλοι οι επιθυμητοί βρόχοι, εφαρμόζεται για ακόμη μια τελευταία φορά η μετάθεση για να επιτευχθεί η επιθυμητή διάταξη των εσωτερικών βρόχων, η οποία βελτιώνει την τοπικότητα των δεδομένων μέσα στο tile.

Ακολούθως, στην περίπτωση πολυεπίπεδου tiling που συμπεριλαμβάνει και το επίπεδο των καταχωρητών, ξεδιπλώνονται πλήρως οι εσωτερικοί βρόχοι που διατρέχουν επαναλήψεις που βρίσκονται εντός των μικρών tiles των καταχωρητών. Τέλος, μπορεί να χρησιμοποιηθεί scalar replacement για την ανάθεση ορισμένων δεδομένων σε καταχωρητές, αποφεύγοντας με αυτό τον τρόπο περιττά load και stores.

Παράδειγμα Παρουσιάζονται τα στάδια της εφαρμογής του μετασχηματισμού tiling στον κώδικα που εκτελεί τον πολλαπλασιασμό δυο πινάκων, για το επίπεδο κρυφής μνήμης. Η τιμή που χρησιμοποιείται για το oft_{jj} και oft_{kk} κατά την εφαρμογή του strip mining είναι το 0.

Αρχική μορφή

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

$$\Downarrow \qquad T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad \Downarrow$$

[5] Η κατεύθυνση ενός tile καθορίζεται από τον βρόχο που δεν πλακοστρώθηκε γι΄ αυτό το επίπεδο.

```
for (j=0; j<N; j++)
    for (k=0; k<N; k++)
        for (i=0; iN<; i++)
            /* loop body */
```

$\Downarrow$ strip mining στον βρόχο j $\Downarrow$

```
for (jj=0; j<N; j+=step)
    for (j=MAX(0,jj); j<MIN(N,jj+step); j++)
        for (k=0; k<N; k++)
            for (i=0; iN<; i++)
                /* loop body */
```

$\Downarrow$ $T = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ $\Downarrow$

```
for (jj=0; j<N; j+=step)
    for (k=0; k<N; k++)
        for (j=jj; j<MIN(N,jj+step); j++)
            for (i=0; iN<; i++)
                /* loop body */
```

$\Downarrow$ strip mining στον βρόχο k $\Downarrow$

```
for (jj=0; j<N; j+=step)
    for (kk=0; kk<N; kk+=step)
        for (k=kk; k<MIN(N,kk+step); k++)
            for (j=jj; j<MIN(N,jj+step); j++)
                for (i=0; iN<; i++)
                    /* loop body */
```

Tiled μορφή

$\Downarrow$ $T = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ $\Downarrow$

```
for (jj=0; jj<N; jj+=step)
    for (kk=0; kk<N; kk+=step)
        for (i=0; iN<; i++)
            for (j=jj; j<MIN(N,jj+step); j++)
                for (k=kk; k<MIN(N,kk+step); k++)
                    C[i][j]+=A[i][k]*B[k][j];
```

Κεφάλαιο 3

Μετασχηματισμοί Δεδομένων

3.1 Γενικά

Οι μετασχηματισμοί βρόχων δεν είναι οι μόνοι μετασχηματισμοί που μπορούν να χρησιμοποιηθούν για την βελτιστοποίηση της τοπικότητας των δεδομέμων σε ένα προγράμμα και κατά επέκταση του χρόνου εκτέλεσης του. Μια άλλη κατηγορία μετασχηματισμών είναι οι μετασχηματισμοί δεδομένων, οι οποίοι ασχολούνται με την διάταξη, αποθήκευση και προσπέλαση των δεδομένων, παρά με την τροποποίηση και αναδιάταξη του ελέγχου ροής ενός προγράμματος.

Σημαντικό πλεονέκτημα των μετασχηματισμών δεδομένων είναι ότι μπορούν να αγνοήσουν τις εξαρτήσεις δεδομένων και να εφαρμοσθούν στις περιπτώσεις που οι εξαρτήσεις δεν επιτρέπουν την εφαρμογή μετασχηματισμών βρόχων. Επίσης, είναι δυνατή η εφαρμογή τους και σε μη τέλεια φωλιασμένους βρόχους, των οποιων η πολυπλοκότητα δεν επιτρέπει την εφαρμογή τεχνικών από το πεδίο των μετασχηματισμών βρόχων. Από την άλλη μεριά, ένα βασικό μειονέκτημα των μετασχηματισμών δεδομένων είναι η αδυναμία τους να εκμεταλλευτούν την χρονική τοπικότητα των δεδομένων όπου αυτή υπάρχει, το οποίο δεν συμβαίνει με τους μετασχηματισμούς βρόχων. Επίσης, όπως αναφέρεται στο [6] οι μετασχηματισμοί δεδομένων δεν μπορούν να εφαρμοσθούν σε προγράμματα που χρησιμοποιούν αριθμητική δεικτών (pointer arithmetic) ή όταν δημιουργούν ψευδώνυμα (aliases) μεταξύ διαφορετικών τύπων.

Το συμπέρασμα που προκύπτει είναι ότι κανένα από τα δύο είδη μετασχηματισμού δεν υπερτερεί του άλλου αρκετά. Γι΄ αυτό το λόγο συνήθως οι δύο μετασχηματισμοί εφαρμόζονται ενοποιημένα για να πετύχουν το καλύτερο δυνατό αποτέλεσμα. Το ακόλουθο παράδειγμα[6] ενθαρρύνει το επιχείρημα αυτό.

Παράδειγμα Ο ακόλουθος κώδικας υλοποιεί τον πολλαπλασιασμό τριών πινάκων:

```
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];
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
```

```
F[i][j]+=E[i][k]*C[k][j];
```

Υπάρχουν τρεις εναλλακτικοί τρόποι για να προσεγγίσουμε το πρόβλημα της βελτιστοποίησης της τοπικότητας:

- *Με μετασχηματισμούς δεδομένων.* Οι πίνακες A, E, F πρέπει να αποθηκευτούν κατά γραμμές, διότι ο δείκτης των στηλών ελέγχεται από βρόχο εσωτερικότερο από το βρόχο που ελέγχει το δείκτη των γραμμών. Το αντίθετο συμβαίνει για τον B κι άρα πρέπει να αποθηκευτεί κατά στήλες. Για τον πίνακα C καμιά από τις δύο δυνατές διατάξεις δεν προτιμάτε έναντι της άλλης διότι η μια είναι κατάλληλη για το πρώτο σύστημα φωλιασμένων βρόχων, ενώ η άλλη για το δεύτερο, κι επομένως υπάρχει σύγκρουση.

- *Με μετασχηματισμούς βρόχων.* Υποθέτουμε αποθήκευση των πινάκων κατά γραμμές. Ο βρόχος j θα πρέπει να γίνει ο εσωτερικότερος βρόχος, κι έτσι να προκύψει η ακόλουθη διάταξη βρόχων (i, k, j). Έτσι, εμφανίζεται χωρική τοπικότητα για τους πίνακες B, C, F και χρονική για τους A, E.

- *Με μετασχηματισμούς βρόχων και δεδομένων.* Επιτυγχάνεται η βέλτιστη τοπικότητα μετασχηματίζοντας τους βρόχους, έτσι ώστε η διάταξη τους να είναι η (j, i, k), και αποθηκεύοντας τους πίνακες A, E κατά γραμμές και τους B, C, F κατά στήλες.

3.2 Βασικές Έννοιες

Ορισμός 3.1 [1] *Κάθε επανάληψη ενός συστήματος m φωλιασμένων βρόχων αντιπροσωπεύεται από το m-διάστατο διάνυσμα $\vec{J} = (j_1, j_2, \ldots, j_m) \in \mathbf{Z}^m$, που ονομάζεται* **διάνυσμα επανάληψης** *(iteration vector).*

Ορισμός 3.2 *Το σύνολο των επαναλήψεων που ορίζεται από τα όρια των φωλιασμένων βρόχων:*

$$J^m = \{(j_1, j_2, \ldots, j_m) \mid j_k \in \mathbf{Z} \wedge l_k \leq i_k \leq u_k, 1 \leq k \leq m\} \subseteq \mathbf{Z}^m$$

αποτελεί ένα κυρτό πολύεδρο του $\mathbf{Z}^m$ και ονομάζεται **χώρος επαναλήψεων** *(iteration space). Παίρνοντας μαζί όλες τις ανισότητες, σχηματίζεται η ακόλουθη ανισότητα πινάκων:*

$$A \cdot \vec{J} \leq \alpha$$

Ορισμός 3.3 *Κάθε στοιχείο ενός n-διάστατου πίνακα αντιπροσωπεύεται από το n-διάστατο διάνυσμα $\vec{I} = (i_1, i_2, \ldots, i_n) \in \mathbf{Z}^n$, που ονομάζεται* **διάνυσμα δεικτών** *(index vector).*

Ορισμός 3.4 *Το σύνολο των δεικτών:*

$$I^n = \{(i_1, i_2, \ldots, i_n) \mid i_j \in \mathbf{Z} \wedge \lambda_j \leq i_j \leq \mu_j, 1 \leq j \leq n\} \subseteq \mathbf{Z}^n$$

αποτελεί ένα ορθογώνιο πολύεδρο[2] *του $\mathbf{Z}^n$ και ονομάζεται* **χώρος δεικτών** *(index space). Παίρνοντας μαζί όλες τις ανισότητες, σχηματίζεται η ακόλουθη ανισότητα πινάκων*[3]*:*

$$B \cdot \vec{J} \leq \beta$$

[1] Επαναλαμβάνονται ορισμένοι ορισμοί από το κεφάλαιο 2 για σκοπούς πληρότητας.

[2] Θεωρούμε σταθερά όρια $\lambda_j, \leq \mu_j$, αλλιώς το πολύεδρο θα είναι κυρτό.

[3] Ο πίνακας B της ανισότητας σχηματίζεται με τον ανάλογο τρόπο που σχηματίζεται ο πίνακας A της παραπάνω ανισότητας.

Ορισμός 3.5 *Σε ένα σύστημα φωλιασμένων βρόχων βάθους m, μια αναφορά στον n-διάστατο πίνακα A μπορεί να αναπαρασταθεί με τη διανυσματική συνάρτηση $\vec{f}_A(j_1, j_2, \ldots, j_n) = U \cdot \vec{J} + \vec{u}$, όπου ο πίνακας U διαστάσεων $n \times m$ ονομάζεται πίνακας αναφοράς και το n-διάστατο διάνυσμα $\vec{u}$ ονομάζεται διάνυσμα απόκλισης. Η διανυσματική συνάρτηση $\vec{f}_A$ θα ονομάζεται διάνυσμα αναφοράς.*

Ορισμός 3.6 Αντιστοίχηση δεδομένων *(data mapping) είναι η συνάρτηση η οποία απεικονίζει το διάνυσμα αναφοράς στην απόσταση από το πρώτο στοιχείο του πίνακα.*

Η συνάρτηση αντιστοίχησης δεδομένων είναι σημαντική, διότι είναι αυτή που προβάλλει ένα πολυδιάστατο πίνακα στην μονοδιάστατη μνήμη.

Για ένα δισδιάστατο πίνακα τα διάνυσματα αντιστοίχησης που εκφράζουν την αποθήκευση των στοιχείων του πίνακα κατά γραμμές και κατά στήλες, δεδομένου ότι το μήκος κάθε γραμμής ή στήλης είναι n, είναι αντίστοιχα τα:

$$m = \begin{pmatrix} n \\ 1 \end{pmatrix} \qquad m = \begin{pmatrix} 1 \\ n \end{pmatrix}$$

Το εσωτερικό γινόμενο των διανυσμάτων αναφοράς αντιστοίχησης δίνει τη μετατόπιση της αντιστοίχησης. Για παράδειγμα μια αναφορά $A[i][j]$ έχει διάνυσμα αναφοράς το (i, j) και επομένως η μετατόπιση της αντιστοίχησης σε πίνακα αποθηκευμένο κατα γραμμές είναι:

$$\psi = (i, j) \cdot \begin{pmatrix} n \\ 1 \end{pmatrix} = in + j$$

3.3 Μετασχηματισμοί δεδομένων

3.3.1 Ορισμός

Βασιζόμενοι στο [15] θα παρουσιάσουμε σε γενικές γραμμές το βασικό θεωρητικό πλαίσιο των μετασχηματισμών δεδομένων των οποίων ο πίνακας μετασχηματισμού είναι αντιστρέψιμος.

Ορισμός 3.7 *Ένας* **μετασχηματισμός δεδομένων** *(data transformation) $(\mathcal{A}, a)$ για ένα n-διάστατο πίνακα A, αποτελείται από ένα $n \times n$ αντιστρέψιμο πίνακα $\mathcal{A}$ και ένα n-διάστατο διάνυσμα a.*

Στην περίπτωση που το διάνυσμα a είναι ίσο με $\mathbf{0}$, το $(\mathcal{A}, a)$ γράφεται απλά $\mathcal{A}$. Εφαρμόζοντας τον μετασχηματισμό $\mathcal{A}$ στο διάνυσμα αναφοράς $\vec{f}_A$ ενός πίνακα A, παίρνουμε το νέο διάνυσμα αναφοράς:

$$\vec{f}'_A = U' \cdot J + \vec{u}' = \mathcal{A} \cdot U \cdot J + (\mathcal{A}\vec{u} + \vec{a})$$

3.3.2 Εφαρμογή Μετασχηματισμού

Ο μετασχηματισμός δεδομένων $(\mathcal{A}, a)$ που αφορά ένα πίνακα δεδομένων A, εφαρμόζεται σε κάθε προσπέλαση που γίνεται στον πίνακα, ενημερώνοντας έτσι τους δείκτες των προσπελάσεων αυτών, και επίσης αλλάζει την διάταξη με την οποία αποθηκεύεται στην μνήμη ο πίνακας A. Γι΄ αυτό και οι μετασχηματισμοί δεδομένων είναι οικουμενικοί, σε αντίθεση με

τους μετασχηματισμούς βρόχων που είναι τοπικοί. Επίσης, ο $\mathcal{A}$ είναι ένας αριστερά εφαρμοζόμενος μετασχηματισμός, σε αντίθεση με τους μετασχηματισμούς βρόχων οι οποίοι είναι δεξιά εφαρμοζόμενοι.

Για την αλλαγή της διάταξης με την οποία αποθηκεύεται ο πίνακας A στη μνήμη, μετασχηματίζουμε το χώρο δεικτών χρησιμοποιώντας τον μετασχηματισμό $(\mathcal{A}, a)$.
Οι νέοι δείκτες J' σχετίζονται με τους παλιούς J ως:

$$J' = \mathcal{A} \cdot \vec{J} + \vec{a}$$

Επομένως:

$$J = \mathcal{A}^{-1} \cdot \vec{J'} - \mathcal{A}^{-1}\vec{a}$$

Και αντικαθιστώντας την τελευταία στην

$$B \cdot \vec{J} \leq \beta$$

προκύπτει:

$$B \cdot (\mathcal{A}^{-1} \cdot \vec{J'} - \mathcal{A}^{-1}\vec{a}) \leq \beta \Rightarrow B \cdot \mathcal{A}^{-1} \cdot \vec{J'} \leq \beta + \mathcal{A}^{-1}\vec{a})$$

Συνοψίζοντας, με την εφαρμογή του μετασχηματισμού $(\mathcal{A}, a)$ τα όρια του νέου χώρου δεικτών βρίσκονται από την λύση της ανισότητας:

$$B' \cdot \vec{J'} \leq \beta'$$

όπου: $B' = B \cdot \mathcal{A}^{-1}$ και $\beta' = \beta + B \cdot \mathcal{A}^{-1}\vec{a}$

Η παραπάνω ανίσωση στη γενική περίπτωση δεν θα δώσει σταθερά όρια. Εάν υπάρχει ανάγκη για σταθερά όρια τότε μπορεί να χρησιμοποιηθεί το μικρότερο ορθογώνιο πολύεδρο που περικλείει το χώρο δεικτών.

Παράδειγμα Έστω το ακόλουθο τμήμα του κώδικα ενός προγράμματος:

```
int A[8][4], B[4][8];
...
for (i=0; i<3; i++)
    for (j=1; j<4; j++)
        A[j][i]+=A[j-1][i+1]+B[j][i+j];
```

Εφαρμόζοντας τους ακόλουθους μετασχηματισμούς δεδομένων:

$$\mathcal{A}_A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad a_A = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\mathcal{A}_B = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \qquad a_B = \begin{pmatrix} 0 \\ 4 \end{pmatrix}$$

παίρνουμε:

$$U'_A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$u'_A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Επομένως, η εικόνα της αναφοράς $A[j][i]$ είναι η $A[i][j]$. Η εικόνα της $A[j-1][i+1]$ βρίσκεται παρόμοια.

$$\begin{aligned} U'_B &= \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ u'_B &= \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 4 \\ 0 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \end{pmatrix} \end{aligned}$$

Επομένως, η εικόνα της αναφοράς $B[j][i+j]$ είναι η $B[i+4][j]$.
Απομένει να βρούμε τα όρια του χώρου δεικτών. Ο αρχικός χώρος δεικτών για τους δύο πίνακες απεικονίζεται από τις ανισώσεις:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 7 \\ 3 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \leq \begin{pmatrix} 3 \\ 7 \\ 0 \\ 0 \end{pmatrix}$$

Οι αντίστροφοι των πινάκων μετασχηματισμών είναι:

$$\mathcal{A}_A^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathcal{A}_B^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$$

Και άρα οι ανισότητες που περιγράφουν τους νέους χώρους δεικτών που προκύπτουν από τους μετασχηματισμούς είναι οι ακόλουθες:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & -1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i'_1 \\ i'_2 \end{pmatrix} \leq \begin{pmatrix} 7 \\ 3 \\ 0 \\ 0 \end{pmatrix}$$

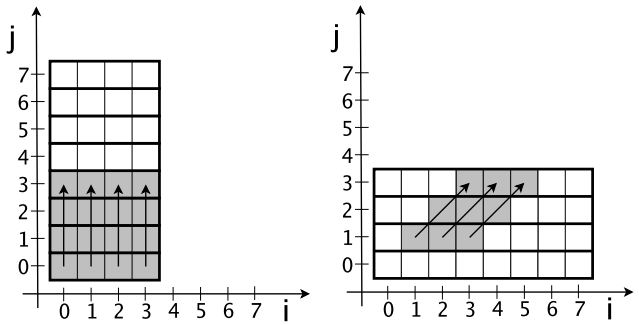
$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 0 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} i'_1 \\ i'_2 \end{pmatrix} \leq \begin{pmatrix} 3 \\ 11 \\ 0 \\ -4 \end{pmatrix}$$

Για τον πίνακα Α τα όρια είναι προφανή. Για τον πίνακα Β όμως θα πρέπει να λυθούν οι ανισότητες:

$$\begin{aligned} 4 - i'_2 \leq \quad & i'_1 \quad \leq 11 - i'_2 \\ 0 \leq \quad & i'_2 \quad \leq 3 \end{aligned}$$

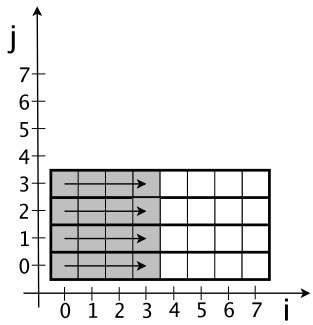
Αυτό όμως που μας ενδιαφέρει είναι να βρούμε το μικρότερο ορθογώνιο πολύεδρο που περικλείει το χώρο δεικτών, το οποίο είναι:

$$\begin{aligned} 1 \leq \quad & i'_1 \quad \leq 11 \\ 0 \leq \quad & i'_2 \quad \leq 3 \end{aligned}$$

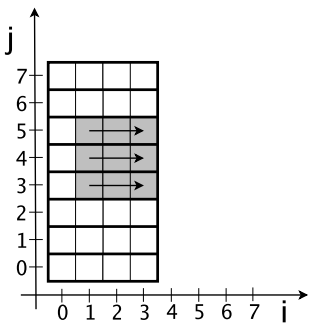


(α′) αρχική μορφή πίνακα A

(β′) αρχική μορφή πίνακα B



(γ′) τελική μορφή πίνακα A



(δ′) τελική μορφή πίνακα B

Σχήμα 3.1: Μοτίβο προσπέλασης των δεδομένων των πινάκων A και B πριν και μετά την εφαρμογή του μετασχηματισμού δεδομένων

```
int A[4][8], B[12][4];
...
for (i=0; i<3; i++)
    for (j=1; j<4; j++)
        A[i][j]+=A[i+1][j-1]+B[i+4][j];
```

Στο σχήμα 3.1 παρουσιάζονται τα μοτίβα προσπέλασης δεδομένων πριν και μετά την εφαρμογή του μετασχηματισμού δεδομένων.

3.3.3 Ορισμένες Βασικές Μορφές

Θα παρουσιάσουμε ορισμένους βασικούς μετασχηματισμούς δεδομένων των οποίων ο πίνακας μετασχηματισμού είναι αντιστρέψιμος.

Unimodular Μετασχηματισμοι [4] Τέτοιοι είναι η μετάθεση, η αναστροφή και η περιστροφή του πίνακα δεδομένων.

$$\begin{array}{rl} \text{μετάθεση} & \left(\begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array}\right), \quad \left(\begin{array}{c} 0 \\ 0 \end{array}\right) \\ \text{αναστροφή} & \left(\begin{array}{cc} 1 & 0 \\ 0 & -1 \end{array}\right), \quad \left(\begin{array}{c} 0 \\ 0 \end{array}\right) \\ \text{περιστροφή} & \left(\begin{array}{cc} 1 & 0 \\ 1 & 1 \end{array}\right), \quad \left(\begin{array}{c} 0 \\ 0 \end{array}\right) \end{array}$$

Σημειώνεται ότι, αν και ο μετασχηματισμένος χώρος δεικτών έχει το ίδιο πλήθος σημείων με τον αρχικό χώρο, το πολύεδρο είναι κυρτό εν γένει, όπως για παράδειγμα με τον μετασχηματισμό της περιστροφής. Σε μια τέτοια περίπτωση, εάν ο πίνακας πρέπει να αποθηκευτεί με τη χρήση ενός ορθογώνιου πίνακα, θα υπάρχουν σημεία που θα μένουν κενά. Επίσης, η συνηθισμένη μεταγωγή μεταξύ της κατά γραμμές ή της κατά στήλες αποθήκευσης του πίνακα γίνεται με τον μετασχηματισμό της μετάθεσης.

Διεύρυνση(Scaling) Χρησιμοποιείται για την διεύρυνση ενός μικρού πίνακα. Για παράδειγμα, εάν θέλουμε να κλιμακώσουμε το πρώτο δείκτη κατα 2, ο πίνακας μετασχηματισμού είναι ο ακόλουθος:

$$\left(\begin{array}{cc} 1 & 0 \\ 0 & 2 \end{array}\right), \left(\begin{array}{c} 0 \\ 0 \end{array}\right)$$

Συμπίεση(Compression) Χρησιμοποιείται για την συμπίεση ενός πίνακα, όταν το μέγεθος του χώρου είναι κρίσιμο. Για παράδειγμα, εάν θέλουμε να συμπιέσουμε το πρώτο δείκτη κατα 2, ο πίνακας μετασχηματισμού είναι ο ακόλουθος:

$$\left(\begin{array}{cc} 1/2 & 0 \\ 0 & 1 \end{array}\right), \left(\begin{array}{c} 0 \\ 0 \end{array}\right)$$

Ολίσθιση(Shift) Χρησιμοποιείται για την ολίσθηση ενός πίνακα κατά μια σταθερά. Κάτι τέτοιο είναι χρήσιμο στην περίπτωση που επιθυμούμε ο πίνακας να ευθυγραμμιστεί (data alignment) και να αποθηκευτεί σε διαφορετική θέση μέσα στην κρυφή μνήμη, με σκοπό την ελάττωση των συγκρούσεων με άλλους πίνακες που υπάρχουν στην κρυφή μνήμη. Για παράδειγμα, εάν θέλουμε να ολισθήσουμε τον πίνακα κατά -1, ο πίνακας μετασχηματισμού είναι ο ακόλουθος:

$$\left(\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right), \left(\begin{array}{c} 1 \\ 0 \end{array}\right)$$

[4] Σε αναλογία με τους μετασχηματισμούς βρόχων.

3.3.4 Εγκυρότητα

Αρχικά, για να είναι έγκυρος ένας μετασχηματισμός δεδομένων, θα πρέπει να ενημερώνονται όλες οι αναφορές και να μην προκύπτουν συγκρούσεις μεταξύ δύο αναφορών. Για παράδειγμα εάν τα στοιχεία $A[1][2]$ και $A[3][4]$ προσπελάνονται και τα δύο, τότε δεν θα πρέπει να μετασχηματιστούν και να αποθηκευτούν στο ίδιο σημείο, έστω $A'[1][2]$. Θα πρέπει επίσης να βεβαιωθούμε ότι δεν χάνονται χρήσιμα δεδομένα[5] με τη χρήση του νέου σχήματος. Επιπρόσθετα, θα πρέπει να βεβαιωθούμε ότι όλες οι εξαρτήσεις δεδομένων διατηρούνται. Το επόμενο σημαντικό θεώρημα αποδικνύεται στο [15].

Θεώρημα 3.1 *Η εφαρμογή μετασχηματισμών δεδομένων που ο πίνακας τους είναι αντιστρέψιμος, $det(\mathcal{A}) \neq 0$, είναι πάντοτε έγκυρη.*

3.4 Ενοποίηση Μετασχηματισμών Βρόχων και Δεδομένων

3.4.1 Γενικά

Φορμαλιστικά, η ενοποιημένη εφαρμογή ενός μετασχηματισμού βρόχων, έστω $\mathcal{L}$, και ενός μετασχηματισμού δεδομένων, έστω $\mathcal{A}$, μετασχηματίζει ένα πίνακα αναφορών U στον νέο πίνακα αναφορών U':

$$U' = \mathcal{A}U\mathcal{L}^{-1}$$

Η ερευνητική κοινότητα αναγνώρισε την ανάγκη για ενοποιηση των μετασχηματισμών βρόχων και δεδομένων με σκοπό την εκμετάλλευση της τοπικότητας που δεν είναι δυνατό να εκμεταλλευτεί η ξεχωριστή εφαρμογή των δύο τύπων μετασχηματισμών. Σκοπός είναι με το μετασχηματισμό δεδομένων να επιτυγχάνεται η αποθήκευση των δεδομένων με τη σειρά που αυτά προσπελάνονται από τον βελτιστοποιημένο με τους μετασχηματισμους βρόχων κώδικα.

3.4.2 Γραμμικές Διατάξεις

Στις γραμμικές διατάξεις τα δεδομένα ενός πίνακα αποθηκεύονται στη μνήμη ως μια φωλιασμένη διάτρεξη των διαστάσεων. Η εσωτερικότερη διάσταση αποτελεί τη διάσταση κατά την οποία αποθηκεύεται ο πίνακας στην μνήμη και αποτελεί τη γρηγορότερα μεταβαλλόμενη διάσταση του πίνακα. Για ένα n-διάστατο πίνακα υπάρχουν $n!$ πιθανές γραμμικές διατάξεις. Για παράδειγμα, ένας δισδιάστατος πίνακας έχει δύο πιθανές τέτοιες διατάξεις: κατά γραμμές ή κατά στήλες. Στην κατά γραμμές (βλέπε σχήμα 3.2(α′)) αποθηκεύεται αρχικά η πρώτη γραμμή του πίνακα, ακολουθεί η δεύτερη γραμμή κ.ο.κ και η γρηγορότερα μεταβαλλόμενη διάσταση είναι η διάσταση i_1 ή αλλιώς διάσταση γραμμών. Στην κατά στήλες αποθηκεύεται αρχικά η πρώτη στήλη του πίνακα, ακολουθεί η δεύτερη στήλη κ.ο.κ και η γρηγορότερα μεταβαλλόμενη διάσταση είναι η διάσταση i_2 ή αλλιώς διάσταση στηλών.

Στο [6] παρουσιάζεται μια ενιαία προσέγγιση η οποία περιορίζεται σε γραμμικές διατάξεις πινάκων. Εισάγεται η έννοια του *διανύσματος διασκελισμού*(stride vector) ως εναλλακτικό

[5] Με τον όρο χρήσιμα δεδομένα, αναφερόμαστε στα δεδομένα που γίνεται τουλάχιστο μια αναφορά κι άρα χρησιμοποιούνται. Για παράδειγμα, σε ένα αραιό πίνακα αναφορές γίνονται μόνο στα μη μηδενικά στοιχεία.

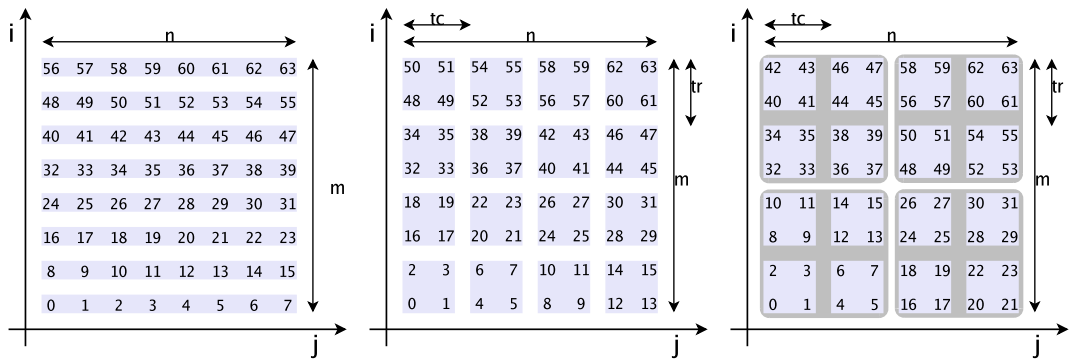
του διανύσματος επαναχρησιμοποίησης (*βλέπε* εότητα 2.2) και αναπτύσσεται μια στρατηγική βελτιστοποίησης με σκοπό την εύρεση του επιθυμητού *διανύσματος αντιστοίχησης*(mapping vector), που αντιπροσωπεύει διατάξεις δεδομένων, και τον πίνακα μετασχηματισμού. Στο τέλος προκύπτει η εξίσωση: $L\vec{v} = U\vec{m}$, όπου το μόνο γνωστό είναι ο πίνακας αναφορών U. Ο αλγόριθμος προσπαθεί να βρει τον μετασχηματισμό του χώρου επαναλήψεων L, το διάνυσμα αντιστοίχησης $\vec{m}$, το οποίο έχει $n!$ πιθανές μορφές για ένα n-διάστατο πίνακα, και το διάνυσμα διασκέλισης $\vec{v}$. Επειδή το πρόβλημα είναι αρκετά δύσκολο, χρησιμοποιείται η ακόλουθη ευριστική μέθοδος. Πρώτο, υποθέτει ότι ο πίνακας L περιέχει μόνο 0 και 1. Δεύτερο, η τιμή του διανύσματος $\vec{v}$ θεωρείτε γνωστή. Ο αλγόριθμος κατασκευάζει σταδιακά γραμμή-γραμμή τον πίνακα T χρησιμοποιώντας ένα περιορισμένο αριθμό από μετασχηματισμούς. Μειονεκτήματα της μεθόδου αυτής είναι η μικρή ακρίβεια, λόγω του περιορισμένου εύρους μετασχηματισμών που χρησιμοποιούνται και επίσης το γεγονός ότι είναι αναγκαία η γνώση του $\vec{v}$.

Στο [9] παρουσιάζεται μια άλλη μέθοδος για τον συνδυασμό μετασχηματισμών δεδομένων και βρόχων, η οποία όντως στηρίζόμενη στην παραπάνω μέθοδο περιορίζεται κι αυτή σε γραμμικές διατάξεις πινάκων. Ο αλγόριθμος διακρίνει τους πίνακες δεδομένων ανάλογα με τη θέση που εμφανίζονται μέσα σε μια έκφραση: πίνακας LHS, όταν εμφανίζεται στο αριστερό μέλος και πίνακες RHS, όταν εμφανίζονται στο δεξιό μέλος. Επειδή συνήθως ένας πίνακας LHS, εμφανίζεται και ως RHS, ο αλγόριθμος προσπαθεί αρχικά να βρει τη βέλτιστη διάταξη του LHS πίνακα. Ο πίνακας μετασχηματισμού L θα πρέπει να είναι τέτοιος έτσι ώστε ο δείκτης j_m του διανύσματος επανάληψης $\vec{J} = j_1, \ldots, j_m$ του μετασχηματισμένου συστήματος φωλιασμένων βρόχων να δεικτοδοτεί μονάχος μόνο μια διάσταση r του πίνακα LHS. Τότε ο πίνακας LHS αποθηκεύεται με γρηγορότερα μεταβαλλόμενη διάσταση την διάσταση r. Ακολούθως, βελτιστοποιείται η διάταξη του πίνακα RHS. Για την βελτιστοποίηση του πίνακα RHS ο αλγόριθμος ελέγχει αν εμφανίζεται μονάχος ο δείκτης j_m μόνο σε κάποια διάσταση s και αν πράγματι συμβαίνει αυτό, τότε ο πίνακας αποθηκεύεται με γρηγορότερα μεταβαλλόμενη διάσταση, την διάσταση s. Αν αυτό δεν είναι δυνατό, τότε ελέγχεται ο δείκτης j_{m-1} αν εμφανίζεται μόνο με τον j_m σε κάποια διάσταση s', κ.ο.κ. και τότε αποθηκεύεται ο πίνακας RHS με γρηγορότερα μεταβαλλόμενη διάσταση την διάσταση s'. Η όλη διαδικασία επαναλαμβάνεται για όλες τις εναλλακτικές διατάξεις αποθήκευσης του πίνακα LHS και στο τέλος επιλέγεται η λύση η οποία εκμεταλλεύεται χωρική τοπικότητα στον εσωτερικότερο βρόχο για τις περισσότερες αναφορές πινάκων. Με τον αλγόριθμο αυτό κατασκευάζεται ταυτόχρονα ο μετασχηματισμός βρόχων και η διάταξη αποθήκευσης των πινάκων.

3.4.3 Διάταξη Ενοτήτων

Οι προηγούμενες μεθόδοι προσπαθούν να βρουν την βέλτιστη γραμμική διάταξη για την αποθήκευση των πινάκων. Τέτοιες όμως γραμμικές διατάξεις ευνοούν μόνο μια διάσταση του πίνακα, δημιουργώντας ανεπιθύμητα μοτίβα προσπέλασης δεδομένων. Οι αστοχίες κρυφής μνήμης λόγω συγκρούσεων που προκαλούνται από τις γραμμικές διατάξεις έχουν σαν αποτέλεσμα την δραστική μείωση της επίδοσης του συστήματος μνήμης. Μια λύση στο πρόβλημα είναι η χρήση των *διατάξεων ενοτήτων (block data layouts-BDL)* . Η χρήση τέτοιων διατάξεων στηρίζεται στην υψηλή επίδοση του μετασχηματισμού tiling, η οποία θα μπορούσε να αυξηθεί ακόμη περισσότερο αν τα δεδομένα αποθηκεύονταν με τη διάταξη που αυτά προσπελάνονται, δηλαδή σε ενότητες (blocks).

Σύμφωνα με το [5] ένας δισδιάστατος πίνακας διαστάσεων $m \times n$ μπορεί να θεωρηθεί ως ένας πίνακας διαστάσεων $\lceil \frac{m}{t_R} \rceil \times \lceil \frac{n}{t_C} \rceil$ αποτελούμενος από ενότητες διαστάσεων $t_R \times$



(α′) διάταξη κατά γραμμές (β′) διάταξη ενοτήτων (BDL) (γ′) διάταξη Morton

Σχήμα 3.2: Τρεις διαφορετικές διατάξεις αποθήκευσης δεδομένων

t_C. Κατ΄ επέκταση, ο δισδιάστατος χώρος (i, j) του αρχικού πίνακα αντιστοιχίζεται σε ένα τετραδιάστατο χώρο (t_i, t_j, f_i, f_j). Για την διάταξη των δεδομένων μέσα στο χώρο (f_i, f_j) της κάθε ενότητας προτείνεται η χρήση γραμμικής διάταξης, δηλαδή διάταξη κατά στήλες ή κατά γραμμές, ανάλογα με τη σειρά που τα δεδομένα προσπελάνονται από τον κώδικα του προγράμματος (*βλέπε* σήμα 3.2(β′)). Παρόλα αυτά, οι επιπλέον εντολές που εκτελούνται για την μετατροπή των δισδιάστατων αναφορών προσπέλασης σε τετραδιάστατες, επιβαρύνουν το συνολικό χρόνο εκτέλεσης.

Μια άλλη παρόμοια διάταξη είναι η αναδρομική διάταξη Morton (*βλέπε* σήμα 3.2(γ′)). Στην διάταξη αυτή ο αρχικός πίνακας χωρίζεται στα τέσσερα τεταρτημόρια, και κάθε υποπίνακας που προκύπτει προβάλλεται συνεχόμενα στην μνήμη. Στη συνέχεια, κάθε τέτοιος υποπίνακας υποδιαιρείται αναδρομικά και προβάλλεται με τον ίδιο τρόπο, μέχρι οι υποπίνακες φτάσουν το επιθυμητό μέγεθος της ενότητας.

Στην επόμενη ενότητα παρουσιάζεται η μέθοδος MBaLt η οποία είναι ιδιαίτερα αποτελεσματική για την γρήγορη δεικτοδότηση των ενοτήτων που προκύπτουν από την εφαρμογή διατάξεων τύπου BDL.

3.5 Η Μέθοδος Διευθυνσιοδότησης MBaLt

Η Αθανασάκη στο [2][16] προτείνει μια αποδοτική μέθοδο υπολογισμού διευθύνσεων για διατάξεις ενοτήτων, την οποία ονομάζει *MBaLt*(Masked Blocked array Layouts). Υιοθετεί διατάξεις ενοτήτων και δεικτοδότηση διεσταλμένων ακεραίων, παρόμοια με την δεικτοδότηση που χρησιμοποιείται στην διάταξη Morton. Επομένως, συνδυάζει τοπικότητα δεδομένων, λόγω των διατάξεων BDL, μαζί με αποδοτική προσπέλαση των στοιχείων, αφού οι πράξεις που γίνονται για την δεικτοδότηση ανάγονται σε απλές πράξεις δυαδικών αριθμών. Η ενσωμάτωση της μεθόδου σε κώδικες που εφαρμόζουν tiling[6] έχει σαν αποτέλεσμα την αποθήκευση των

[6]Η ενσωμάτωση της MBaLt σε κώδικες tiling είναι πιο εύκολη, όταν το σύστημα φωλιασμένων βρόχων είναι πλακωστρωμένο σε όλες τις διαστάσεις, δηλαδή έχει βάθος βάθος $2N$, όπου N το βάθος

δεδομένων με τη σειρά που αυτά προσπελάνονται, χωρίς οι επιπλέον πράξεις που γίνονται για τη δεικτοδότηση να επιβαρύνουν σε μεγάλο βαθμό το συνολικό χρόνο εκτέλεσης.

3.5.1 Διεσταλμένοι Ακέραιοι (Dilated Integers) και Δεικτοδότηση Morton

Οι ακόλουθοι ορισμοί ισχύουν μόνο για δισδιάστατους πίνακες.

Ορισμός 3.8 *Ο ακέραιος* $\overrightarrow{b} = \sum_{k=0}^{w-1} 4^k$ *είναι η σταθερά* $0x55555555$ *και καλείται evenBits. Παρόμοια, ο ακέραιος* $\overleftarrow{b} = 2\overrightarrow{b}$ *είναι η σταθερά* $0xaaaaaaaa$ *και καλείται oddBits.*

Ορισμός 3.9 *Η άρτια-διεσταλμένη (even-dilated) αναπαράσταση του* $j = \sum_{k=0}^{w-1} j_k 2^k$ *είναι* $\sum_{k=0}^{w-1} j_k 4^k$ *και συμβολίζεται με* $\overrightarrow{j}$*. Η περιττή-διεσταλμένη (odd-dilated) αναπαράσταση του* $i = \sum_{k=0}^{w-1} i_k 2^k$ *είναι* $2\overrightarrow{i}$ *και συμβολίζεται με* $\overleftarrow{i}$*.*

Θεώρημα 3.2 *Η δεικτοδότηση Morton για το* $< i, j >^o$ *στοιχείο ενός πίνακα είναι* $\overleftarrow{i} \vee \overrightarrow{j}$*, ή* $\overleftarrow{i} + \overrightarrow{j}$*.*

Έστω λοιπόν ο δείκτης γραμμής i με τα δυαδικά ψηφία του διεσταλμένα, έτσι ώστε να καταλαμβάνουν τις θέσεις των ψηφίων 1 του oddBits, και ο δείκτης στήλης j με τα δυαδικά ψηφία του διεσταλμένα, έτσι ώστε να καταλαμβάνουν τις θέσεις των ψηφίων 1 του evenBits. Εάν ο πίνακας A είναι αποθηκευμένος σε διάταξη *Morton*, τότε το στοιχείο (i, j) προσπελάνεται ως $A[i + j]$ ασχέτως του μεγέθους του πίνακα.

Ένας βρόχος της μορφής:

```
for (i=0; i<N; i++)
    ...
```

τροποποιείται στον:

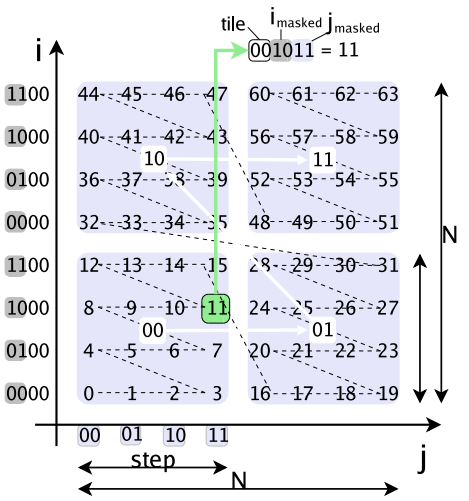
```
for (im=0; im<Nm; im=(im-evenBits)&evenBits)
    ...
```

όπου $im = \overrightarrow{i}$ και $Nm = \overrightarrow{N}$.

Εάν χρησιμοποιούμε αναπαράσταση διεσταλμένων ακέραιων για την δεικτοδότηση ενός πίνακα, τότε εάν ένας δείκτης:

- Χρησιμοποιείται μόνο ως δείκτης στήλης, η αναπαράσταση του είναι άρτια.
- Χρησιμοποιείται μόνο ως δείκτης γραμμής, η αναπαράσταση του είναι περιττή.
- Χρησιμοποιείται ως δείκτης στήλης και γραμμής, η αναπαράσταση του είναι άρτια και όταν θέλουμε την περιττή αναπαράσταση του απλώς διπλασιάζουμε.

του συστήματος χωρίς tiling.



Σχήμα 3.3: Μετασχηματισμός ZZ

Παράδειγμα Ο πολλαπλασιασμός πινάκων με πίνακες αποθηκευμένους σε διάταξη Morton είναι ο ακόλουθος:

```
#define evenIncrement(i)(i=((i-evenBits)&evenBits))
#define oddIncrement(i)(i=((i-oddBits)&oddBits))
for (i=0; i<colsOdd; oddIncrement(i))
    for (j=0; j<rowsEven; evenIncrement(j))
        for (k=0; k<rowsAEven; evenIncrement(k))
            C[i+j]+=A[i+k]*B[2*k+j];
    ...
```

Όπου, rowsEven, colsOdd και rowsAEven είναι τα όρια σε αναπαράσταση διεσταλμένων ακέραιων. Παρατηρείστε ότι ο k χρησιμοποιείται ως δείκτης στήλης και ως δείκτης γραμμής. Επομένως, μετατρέπεται στον $\overrightarrow{k}$ και όταν χρησιμοποιηθεί ως δείκτης γραμμής διπλασιάζεται για να μας δώσει το $\overleftarrow{k}$.

3.5.2 Μια Πρώτη Προσέγγιση

Τα BDL επεκτείνονται, έτσι ώστε τα δεδομένα να αποθηκεύονται με τη σειρά που αυτά προσπελάνονται όταν εκτελείται το πρόγραμμα. Αυτή η διάταξη αποθήκευσης παρουσιάζεται στο σχήμα 3.3 για ένα πίνακα μεγέθους 8×8, ο οποίος υποδιαιρείται σε tiles μεγέθους 4×4. Η διακεκομμένη γραμμή δείχνει τη σειρά με την οποία το πρόγραμμα προσπελάνει τα δεδομένα, ενώ η αρίθμηση δηλώνει τη σειρά με την οποία τα δεδομένα είναι αποθηκευμένα. Οι πίνακες υποδιαιρούνται σε tiles ίδιου μεγέθους με αυτά που χρησιμοποιεί το πρόγραμμα το οποίο προσπελάνει τα στοιχεία του πίνακα.

Ο μετασχηματισμός του παραδείγματος ονομάζεται ZZ, διότι το εσωτερικό των tiles σαρώνεται κατά γραμμές, όμοια με Z και η μετακίνηση από tile σε tile είναι επίσης όμοια με Z. Γενικά, το πρώτο γράμμα του μετασχηματισμού δηλώνει τη μετακίνηση από tile σε tile, ενώ το δεύτερο γράμμα δηλώνει τη σειρά με την οποία τα δεδομένα σαρώνονται στο εσωτερικό του tile.

Η σάρωση των στοιχείων ενός πίνακα σε διάταξη ZZ μπορεί να γίνει με τον ακόλουθο κώδικα:

```
/* ZZ-order */
for (ii=0; ii<N; ii+=step)
    for (jj=0; jj<N; jj+=step)
        for (i=ii; i<MIN(ii+step,N); i++)
            for (j=jj; j<MIN(jj+step,N); j++)
                A[i][j]=...;
```

Παρόμοια προκύπτουν και οι άλλες τρεις μορφές του μετασχηματισμού: NZ, NN και ZN, οι οποίες παρουσιάζονται στο σχήμα 3.4. Οι αντίστοιχοι κώδικες σάρωσης είναι οι παρακάτω:

```
/* NZ-order */
for (jj=0; jj<N; jj+=step)
    for (ii=0; ii<N; ii+=step)
        for (i=ii; i<MIN(ii+step,N); i++)
            for (j=jj; j<MIN(jj+step,N); j++)
                A[i][j]=...;
```

```
/* NZ-order */
for (ii=0; ii<N; ii+=step)
    for (jj=0; jj<N; jj+=step)
        for (i=ii; i<MIN(ii+step,N); i++)
            for (j=jj; j<MIN(jj+step,N); j++)
                A[j][i]=...;
```

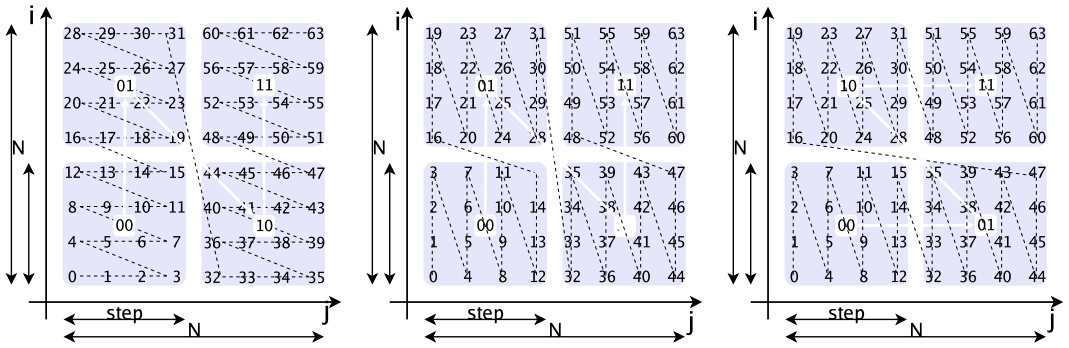
```
/* ZN-order */
for (jj=0; jj<N; jj+=step)
    for (ii=0; ii<N; ii+=step)
        for (i=ii; i<MIN(ii+step,N); i++)
            for (j=jj; j<MIN(jj+step,N); j++)
                A[j][i]=...;
```

Επειδη οι μεταγλωττιστές υποστηρίζουν μόνο γραμμικές διατάξεις και όχι διατάξεις ενοτήτων, η σάρωση του πίνακα όταν είναι αποθηκευμένος σε μία από τις παραπάνω μορφές μπορεί να γίνει χρησιμοποιώντας μονοδιάστατο πίνακα του οποίου η δεικτοδότηση γίνεται μέσω διεσταλμένων ακέραιων. Η δεικτοδότηση Morton δεν μπορεί να εφαρμοσθεί, διότι προυποθέτει την ύπαρξη αναδρομικού σχήματος πλακόστρωσης, σε αντίθεση με την παρούσα περίπτωση που η πλακόστρωση εφαρμόζεται μόνο σε ένα επίπεδο. Αντίς λοιπόν των oddBits και evenBits, χρησιμοποιούνται δυαδικές μάσκες.

3.5.3 Θεωρία Μασκών

Ακολουθεί η παρουσίαση της μορφής των μασκών για ένα πίνακα $A[i][j]$ μεγέθους $N_i \times N_j$ και μέγεθος tile $step_i \times step_j$.

Ο αριθμός των συνεχόμενων 0 και 1 που αποτελούν το κάθε τμήμα της μάσκας ορίζεται από τις συναρτήσεις $m_x = log(step_x)$ και $t_x = log\frac{N_x}{step_x}$, όπου $step$ είναι δύναμη του 2. Εάν



(α′) Μετασχηματισμός NZ (β′) Μετασχηματισμός NN (γ′) Μετασχηματισμός ZN

Σχήμα 3.4: Οι άλλες τρεις μορφές του μετασχηματισμού

το N δεν είναι δύναμη του 2, τότε δεσμεύουμε τον αμέσος μεγαλύτερο πίνακα με $N = 2^n$ και γεμίζουμε τα κενά στοιχεία με τυχαίες τιμές. Αυτά τα επιπρόσθετα στοιχεία δεν επιβαρύνουν το συνολικό χρόνο εκτέλεσης, διότι δεν προσπελάνονται ποτέ.

Όπως και στην περίπτωση της δεικτοδότησης των 4-δεντρων, έτσι κι εδώ κάθε στοιχείο $A[i][j]$ του δισδιάστατου πίνακα βρίσκεται στην θέση $[i_m + j_m] = [i_m|j_m]$ του μονοδιάστατου πίνακα. Τα i_m, j_m παράγονται από τα i, j όταν εφαρμοσθεί στα τελευταία η κατάλληλη μάσκα.

3.5.4 Υλοποίηση-Σχεδιασμός Κώδικα

Στο σχήμα 3.5 παρουσιάζεται πως γίνεται η αποθήκευση των στοιχείων $A[i][j]$ του δισδιάστατου πίνακα σε ένα μονοδιάστατο πίνακα, χρησιμοποιώντας τη διάταξη ZZ. Γενικά, για την υλοποίηση των BDL, οι δείκτες που ελέγχουν τη θέση των στοιχείων στο δισδιάστατο πίνακα δεν θα πρέπει να παίρνουν διαδοχικές τιμές. Στο παράδειγμα μας, οι ορθές τιμές των δεικτών i και j που ελέγχουν τις γραμμές και τις στήλες αντίστοιχα παρουσιάζονται στον παρακάτω πίνακα:

Γραμμή/Στήλη:	0	1	2	3	4	5	6	7
i:	0	4	8	12	32	36	40	44
j:	0	1	2	3	16	17	18	19

Αυτές οι τιμές αντιστοιχούν στα στοιχεία της πρώτης γραμμής και στήλης (βλέπε σχήμα 3.3). Για παράδειγμα, όταν γίνεται μια αναφορά στο στοιχείο της 2ης γραμμής και 3ης στήλης, δηλαδή $A[2][3]$, οι αντίστοιχοι δείκτες για τον μονοδιάστατο πίνακα θα είναι $i = 8, j = 3$. Το επιθυμητό στοιχείο προσπελάνεται προσθέτωντας τους δύο δείκτες: $A[i + j] = A[8 + 3] = A[11]$.

Οι κατάλληλες μάσκες για τον 8×8 πίνακα του παραδείγματος μας, με $step = 4$, είναι 101100 για το δείκτη γραμμής i και 010011 για το δείκτη στήλης j. Εάν οι τιμές $0 - 7$ των γραμμικών δεικτών φιλτραριστούν χρησιμοποιώντας αυτές τις μάσκες (βλέπε σχήμα 3.6), τότε προκύπτουν οι επιθυμητές διεσταλμένες τιμές.

Μετασχηματισμός ZZ

Δείκτης Γραμμής	$11\ldots1$	$00\ldots0$	$11\ldots1$	$00\ldots0$
Δείκτης Στήλης	$00\ldots0$	$11\ldots1$	$00\ldots0$	$11\ldots1$
	$\leftarrow t_i \rightarrow$	$\leftarrow t_j \rightarrow$	$\leftarrow m_i \rightarrow$	$\leftarrow m_j \rightarrow$

Μετασχηματισμός NZ

Δείκτης Γραμμής	$00\ldots0$	$11\ldots1$	$11\ldots1$	$00\ldots0$
Δείκτης Στήλης	$11\ldots1$	$00\ldots0$	$00\ldots0$	$11\ldots1$
	$\leftarrow t_i \rightarrow$	$\leftarrow t_j \rightarrow$	$\leftarrow m_i \rightarrow$	$\leftarrow m_j \rightarrow$

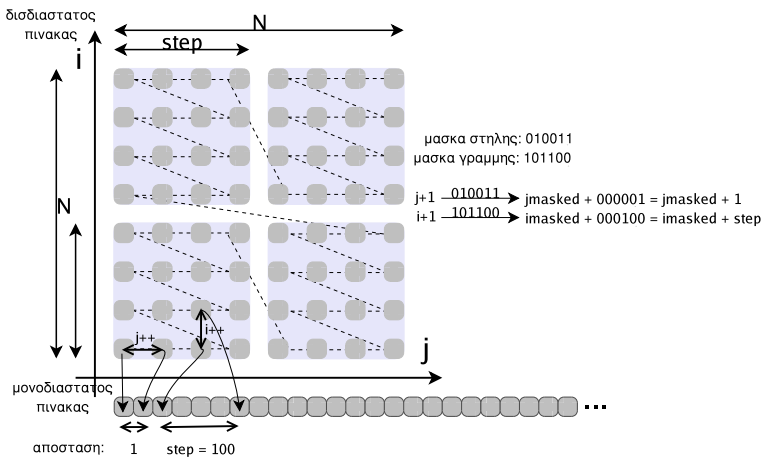
Μετασχηματισμός NN

Δείκτης Γραμμής	$00\ldots0$	$11\ldots1$	$00\ldots0$	$11\ldots1$
Δείκτης Στήλης	$11\ldots1$	$00\ldots0$	$11\ldots1$	$00\ldots0$
	$\leftarrow t_i \rightarrow$	$\leftarrow t_j \rightarrow$	$\leftarrow m_i \rightarrow$	$\leftarrow m_j \rightarrow$

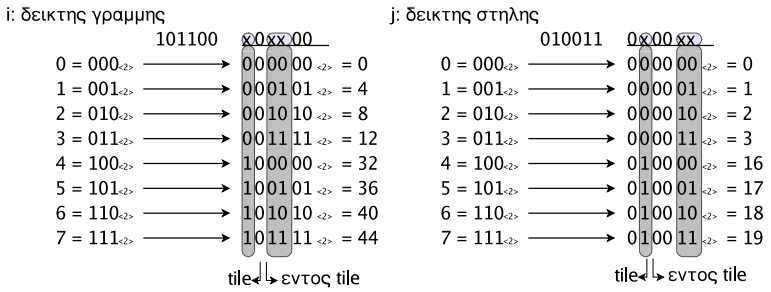
Μετασχηματισμός ZN

Δείκτης Γραμμής	$11\ldots1$	$00\ldots0$	$00\ldots0$	$11\ldots1$
Δείκτης Στήλης	$00\ldots0$	$11\ldots1$	$11\ldots1$	$00\ldots0$
	$\leftarrow t_i \rightarrow$	$\leftarrow t_j \rightarrow$	$\leftarrow m_i \rightarrow$	$\leftarrow m_j \rightarrow$

Πίνακας 3.1: Μάσκες Μετασχηματισμών



Σχήμα 3.5: Μετατροπή ενός δισδιάστατου πίνακα σ΄ ένα μονοδιάστατο με τη χρήση του μετασχηματισμού ZZ



Σχήμα 3.6: Μετατροπή των γραμμικών τιμών των δεικτών γραμμής και στήλης στις αντίστοιχες διεσταλμένες τιμές με τη χρήση μασκών ZZ

Παράδειγμα (Πολλαπλασιασμός Πινάκων) Στο κεφάλαιο 2 εφαρμόσαμε τον μετασχηματισμό tiling στον κωδικα του πολλαπλασιασμού πινάκων για την βελτίωση της τοπικότητας. Εδώ θα προχωρήσουμε περαιτέρω την βελτιστοποίηση χρησιμοποιώντας τις διατάξεις ενοτήτων (BDL) και τη μέθοδο MBaLt για την γρήγορη δεικτοδότηση των ενοτήτων.

Επειδή γενικά η εφαρμογή της μεθόδου MBaLt είναι πιο απλή σε κώδικες που ο μετασχηματισμός tiling εφαρμόσθηκε σε όλες τις διαστάσεις, θα χρησιμοποιήσουμε τον κώδικα πολλαπλασιασμού πίνακων με tiling και στις τρεις διαστάσεις. Επίσης όπως προκύπτει με την εφαρμογή του μοντέλου εκτίμησης κόστους φωλιασμένων βρόχων που παρουσιάζεται στο παράστημα Α′.1, η βέλτιστη διάταξη των βρόχων είναι η (i, k, j). Ο μετασχηματισμένος κώδικας είναι ο ακόλουθος:

```
for (ii=0; ii<8; ii+=4)
  for (kk=0; kk<8; kk+=4)
    for (jj=0; jj<8; jj+=4)
      for (i=ii; i<MIN(8,ii+4); i++)
        for (k=kk; k<MIN(8,kk+4); k++)
          for (j=jj; j<MIN(8,jj+step); j++)
            C[i][j]+=A[i][k]*B[k][j];
```

Και οι τρεις πίνακες A, B, C σαρώνονται με βάση τη διάταξη του μετασχηματισμού ZZ. Οι τρεις εσωτερικότεροι βρόχοι (i, k, j) ελέγχουν τη σάρωση εντός του tile. Όπως φαίνεται και στο σχήμα 3.6, τα 4 λιγότερο σημαντικά ψηφία των μασκών είναι αρκετά για την σάρωση όλων των επαναλήψεων εντός του tile. Οι τρεις εξωτερικότεροι βρόχοι ελέγχουν τη μετακίνηση από tile σε tile. Τα δύο περισσότερο σημαντικά ψηφία των μασκών είναι ικανά για την μετακίνηση μεταξύ των tiles.

Στον ακόλουθο πίνακα παρουσιάζονται οι ρόλοι των δεικτών και οι κατάλληλες μάσκες:

Δείκτης	Ελέγχει	Πίνακα	Κατάλληλη Μάσκα
i, ii	γραμμές	A, C	μάσκα γραμμής (101100)
j, jj	στήλες	B, C	μάσκα στήλης (010011)
k, kk	γραμμές	B	μάσκα γραμμής (για kB, kkB)
k, kk	στήλες	A	μάσκα στήλης (για kA, kkA)

Ο δείκτης i παίρνει τιμές στο $\{xx00|x \in 0, 1\}$ και το ii παίρνει τιμές στο $\{x00000|x \in 0, 1\}$, κι άρα το $i|ii = x0xx00$ δίνει τις επιθυμητές τιμές για τις γραμμές των πινάκων

A, C. Παρόμοια, ο δείκτης j παίρνει τιμές στο $\{00xx|x \in 0,1\}$ και το jj παίρνει τιμές στο $\{0x0000|x \in 0,1\}$, κι άρα το $j|jj = 0x00xx$ δίνει τις επιθυμητές τιμές για τις στήλες των πινάκων B, C. Ο δείκτης k δεν ελέγχει τον ίδιο τύπο διάστασης και στους δύο πίνακες που χρησιμοποιείται. Έτσι, για τον πίνακα A η κατάλληλη μάσκα είναι η μάσκα στήλης, $kA \in \{00xx|x \in 0,1\}$ και $kkA \in \{0x0000|x \in 0,1\}$. Για τον πίνακα B η κατάλληλη μάσκα είναι η μάσκα γραμμής, $kB \in \{xx00|x \in 0,1\}$ και $kkB \in \{x00000|x \in 0,1\}$. Μια χρήσιμη παρατήρηση είναι ότι $kB = kA << log(step)$ και $kkB = kkA << log(\frac{N}{step})$.

Έχουμε λοιπόν:

$$\begin{aligned} ibound &= column_mask = 101100_{<2>} = 44 \\ iiincrement &= 100000_{<2>} = 32 = 4 \times 4 << \frac{N}{step} \\ iincrement &= 100_{<2>} = 4 = step \\ ireturn &= min\{column_mask, ii|1100_{<2>}\} \end{aligned}$$

```
for (ii=0; ii<ibound; ii+=iiincrement) {
    itilebound=(ii|imask)+1;
    ireturn=MIN(ibound,itilebound);
    for (kk=0; kk<kjbound; kk+=kkjjincrement) {
        ktilebound=(kk|kjmask)+1;
        kreturn=MIN(kjbound,ktilebound);
        kkB=kk<<logNxy;
        for (jj=0; jj<kjbound; jj+=kkjjincrement) {
            jtilebound=(jj|kjmask)+1;
            jreturn=MIN(kjbound,jtilebound);
            for (i=ii; i<ireturn; i+=iincrement)
                for (k=kk; k<kreturn; k+=kjincrement) {
                    kB=(k & (step_1))<<logstep;
                    ktB=kkB|kB;
                    xA=i|k;
                    for (j=jj; j<jreturn; j+=kjincrement)
                        C[i|j]+=A[xA]*B[ktB|j];
                }
        }
    }
}
```

3.5.5 Ο αλγόριθμος

Επίσης, η Αθανασάκη στο [2] προτείνει ένα ευριστικό αλγόριθμο[7] για την εύρεση του μετασχηματισμού που πετυχαίνει την μέγιστη δυνατή βελτιστοποίηση. Ο αλγόριθμος είναι οικουμενικός, δηλαδή για την εύρεση της καλυτερης διάταξης του κάθε πίνακα, λαμβάνει υπόψιν όλες τις εμφανίσεις των πινάκων μέσα στο πρόγραμμα. Ακολουθούν τα στάδια του αλγορίθμου με παράλληλη αναφορά στο παράδειγμα του πολλαπλασιασμού:

[7] Ο αλγόριθμος επεκτείνει τον αλγόριθμο που παρουσιάζεται στο [9] και στον οποίο γίνεται μια σύντομη αναφορά στην ενότητα 3.4.2.

1. Δημιούργησε ένα πίνακα R μεγέθους $r \times l$, όπου $r =$ πλήθος διαφορετικών αναφορών πίνακα και $l =$ βάθος συστήματος φωλιασμένων βρόχων (πριν την εφαρμογή tiling). Εάν υπάρχουν δύο ταυτόσημες αναφορές σ΄ ένα πίνακα, δεν υπάρχει λόγος να σημειωθούν και οι δύο. Απλά, σημείωσε την αναφορά αυτή, έτσι ώστε να έχει μεγαλύτερη προτεραιότητα κατά την βελτιστοποίηση. Για το παράδειγμα του πολλαπλασιασμού πινάκων ο πίνακας R είναι ο ακόλουθος:

$$R = \begin{array}{c} \begin{array}{ccc} i & k & j \end{array} \\ \left(\begin{array}{ccc} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{array} \right) \end{array} \begin{array}{c} C** \\ A \\ B \end{array}$$

   Πρέπει να σημειωθεί ότι ο LHS πίνακας μιας έκφρασης (στο παράδειγμα μας ο C) είναι πιο σημαντικός, διότι συνήθως σε κάθε επανάληψη το στοιχείο διαβάζεται και γράφεται.

   Κάθε στήλη του πίνακα C αντιπροσωπεύει ένα δείκτη επανάληψης. Έτσι, κάθε στοιχείο του R είναι ίσο με 1 όταν ο αντίστοιχος δείκτης ελέγχει μια από τις διαστάσεις του πίνακα, αλλιώς είναι ίσο με 0.

2. Βελτιστοποίησε πρώτα τη διάταξη του πίνακα με τις περισσότερες αναφορές μέσα στο σύστημα φωλιασμένων βρόχων (στο παράδειγμα μας ο C). Ο εφαρμοζόμενος μετασχηματισμός βρόχων θα πρέπει να είναι τέτοιος ώστε ένας από τους δείκτες επανάληψης του συστήματος φωλιασμένων βρόχων που ελέγχουν κάποια διάσταση του πίνακα αυτού, να μετακινηθεί στην εσωτερικότερη θέση. Άρα, η γραμμή C του R θα πρέπει να έρθει στη μορφή $(x_1, \ldots x_{l-1}, 1)$, όπου $x_k \in \{0, 1\}$. Για να επιτευχθεί αυτό, μπορεί να γίνει εναλλαγή των στηλών. Ο επιλεγόμενος δείκτης επανάληψης είναι προτιμότερο να είναι ο μόνος δείκτης που δεικτοδοτεί την διάσταση αυτή και να μην εμφανίζεται σε άλλη διάσταση του πίνακα C.

   Για την εκμετάλλευση της χωρικής επαναχρησιμοποίησης της αναφοράς αυτής, ο πίνακας C θα πρέπει να αποθηκευτεί στη μνήμη με την επιλεγμένη διάσταση, έστω fcd_C να αποτελεί την γρηγορότερα μεταβαλλόμενη διάσταση. Σημειώνεται ότι θα πρέπει να ελεγχούν όλες οι πιθανές διαστάσεις.

3. Ακολούθως, θα πρέπει να διορθωθούν οι υπόλοιπες αναφορές κατά σειρά προτεραιότητας. Ο στόχος είναι να έρθουν όσο το δυνατό περισσότερες γραμμές του R στη μορφή $(x_1, \ldots x_{l-1}, 1)$, όπου $x_k \in \{0, 1\}$. Εάν ένας δείκτης πίνακα, έστω ότι είναι αυτός που ελέγχει τη διάσταση y του πίνακα A, είναι ταυτόσημος με τον fcd_C του πίνακα C, τότε αποθήκευσε τον A έτσι ώστε η γρηγορότερα μεταβαλλόμενη διάσταση να είναι η y. Εάν δεν υπάρχει τέτοια διάσταση για τον A, τότε θα πρέπει να ελεγχθεί εάν η αναφορά μπορεί να μετασχηματιστεί στη μορφή $A[*, \ldots, *, f(j_{in-1}, *, \ldots, *]$, όπου $f(j_{in-1})$ είναι συνάρτηση του δείκτη του δεύτερου εσωτερικότερου βρόχου και άλλων δεικτών εκτός του j_{in}, και το $*$ δηλώνει όρο ανεξάρτητο των j_{in-1} και j_{in}. Άρα, η γραμμή A του πίνακα R θα πρέπει να είναι της μορφής $(x_1, \ldots x_{l-2}, 1, 0)$, όπου $x_k \in \{0, 1\}$, και να εκμεταλλεύεται η χωρική επαναχρησιμοποίηση κατά μήκος του j_{in-1}. Εάν δεν υπάρχει τέτοιος μετασχηματισμός, τότε ελέγχεται ο j_{in-2}, κ.ο.κ. Εάν ελεγχθούν όλοι οι δείκτες επανάληψης, τότε οι βρόχοι θέτονται σε μια τυχαία διάταξη, λαμβάνοντας υπόψιν τις εξαρτήσεις δεδομένων.

4. Μετά την εύρεση ενός πλήρους μετασχηματισμού βρόχων και μιας διάταξης μνήμης ελέγχονται οι υπόλοιπες εναλλακτικές λύσεις. Στο τέλος επιλέγεται η λύση η οποία εμφανίζει χωρική τοπικότητα στον εσωτερικότερο βρόχο στις περισσότερες αναφορές.

5. Αφού το σύστημα φωλιασμένων βρόχων αναδιατακτεί πλήρως, γίνεται στη συνέχεια εφαρμογή του μετασχηματισμού tiling. Σε αυτό το στάδιο καθορίζεται η πλήρης διάταξη αποθήκευσης των πινάκων.

   Στην περίπτωση της tiled μορφής, για κάθε μια από τις αναφορές πίνακα, η διάσταση η οποία ορίστηκε ως η γρηγορότερα μεταβαλλόμενη θα πρέπει να παραμείνει η ίδια όσον άφορα τη διάταξη αποθήκευσης στο εσωτερικό των tiles. Αυτό σημαίνει ότι στην περίπτωση των δισδιάστατων πινάκων, έαν η αναφορά έχει τη μορφή $C[*, j_n]$, έτσι ώστε η επιθυμητή διάταξη αποθήκευσης στο εσωτερικό του tile να είναι διάταξη γραμμών, θα πρέπει να χρησιμοποιήσουμε τη διάταξη xZ, δηλαδή τη ZZ ή τη NZ.Εάν η αναφορά έχει τη μορφή $C[j_n, *]$, έτσι ώστε η επιθυμητή διάταξη αποθήκευσης στο εσωτερικό του tile να είναι διάταξη στηλών, θα πρέπει να χρησιμοποιήσουμε τη διάταξη xN, δηλαδή τη NN ή τη ZN. Η μετακίνηση από tile σε tile, κι άρα το πρώτο γράμμα του μετασχηματισμού, καθορίζεται από το είδος του tiling που θα εφαρμοστεί. Για το παράδειγμα μας, εάν δεν εφαρμοσθεί tiling στη διάσταση $*$ τότε ο μετασχηματισμός θα είναι ο NZ ή ο ZN αντίστοιχα. Αλλιώς, εάν η tiled μορφή του συστήματος φωλιασμένων βρόχων βάθους n: $(j_1, j_2, \ldots, j_n$ είναι: $(jj_1, jj_2, \ldots, jj_n, j_1, j_2, \ldots j_n)$ (όπου jj_k είναι ο δείκτης επανάληψης που ελέγχει την μετακίνηση από ένα tile σε ένα άλλο της διάστασης j_k), τότε χρησιμοποιείται η διάταξη ZZ ή NN αντίστοιχα.

Κεφάλαιο 4

Το Μετροπρόγραμμα 'Παραγοντοποίηση Cholesky' και η Επίδοση Διατάξεων Ενοτήτων με Δεικτοδότηση MBaLt

4.1 Γενικά

Δεδομένου ενός πίνακα A για τον οποίο ισχύει $\forall x \in \mathbf{R}(x^T Ax > 0)$, η παραγοντοποίηση Cholesky βρίσκει ένα πίνακα $C : A = C^T C$. Τέτοιες παραγοντοποιήσεις είναι χρήσιμες για την επίλυση γραμμικών συστημάτων εξισώσεων. Ο κώδικας που εκτελεί την παραγοντοποίηση είναι ο ακόλουθος:

```
for (k=0; k<N;k++) {
    A[k][k]=sqrt(A[k][k]);
    for (i=k+1;i<N;i++)
        A[i][k]=A[i][k]/A[k][k];

    for (j=k+1;j<N;j++)
        for (i=j;i<N;i++)
            A[i][j]=A[i][j]-A[i][k]*A[j][k];
}
```

Η *παραγοντοποίηση Cholesky* χρησιμοποιείται ευρέως ως μετροπρόγραμμα για την μέτρηση της επίδοσης υπολογιστικών συστημάτων. Στο παρών κεφάλαιο θα υλοποιήσουμε την παραγοντοποίηση Cholesky εφαρμόζοντας διάφορους μετασχηματισμούς βρόχων και δεδομένων, συμπεριλαμβανομένου και των διατάξεων ενοτήτων με διευθυνσιόδοτηση MBaLt, για να μελετήσουμε και να συγκρίνουμε την αποτελεσματικότητα της τελευταίας.

4.2 Σχεδίαση Tiled Μορφής με Γραμμικές Διατάξεις

Εξαρτήσεις δεδομένων Αρχικά, θα πρέπει να βρούμε τις εξαρτήσεις δεδομένων. Έστω δύο επαναλήψεις $\mathcal{I} = (k_1, j_1, i_1)$ και $\mathcal{I}' = (k_1, j_1, i_1)$ του χώρου επαναλήψεων (k, j, i) τέτοιες ώστε $\mathcal{I}' \succ \mathcal{I}$. Ο χώρος επαναλήψεων καθορίζεται από τις ανισώσεις:

$$\begin{aligned} 0 \leq \quad & k_{1,2} \quad \leq N-1 \\ k_{1,2}+1 \leq \quad & j_{1,2} \quad \leq N-1 \\ j_{1,2} \leq \quad & i_{1,2} \quad \leq N-1 \end{aligned}$$

οι οποίες σε μορφή πίνακα γράφονται ως:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} k \\ j \\ i \end{pmatrix} \leq \begin{pmatrix} N-1 \\ N-1 \\ N-1 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

Τα διανύσματα εξαρτήσεων βρίσκονται ως ακολούθως.

- Εξάρτηση τύπου RAW $\mathbf{A[i_1][j_1]} \rightarrow \mathbf{A[i_2][k_2]}$ Αφού αναφέρονται στο ίδιο στοιχείο, ισχύει: $i_1 = i_2, j_1 = k_2 \Rightarrow j_2 \geq k_1+1$ και $j_2 \geq j_1+1 \Rightarrow k = k_2-k_1 \geq 1, j = j_2-j_1 \geq 1$ και $i = i_2 - i_1$. Επομένως το διάνυσμα εξάρτησης είναι:

$$d_1 = (+, +, 0) \succ 0$$

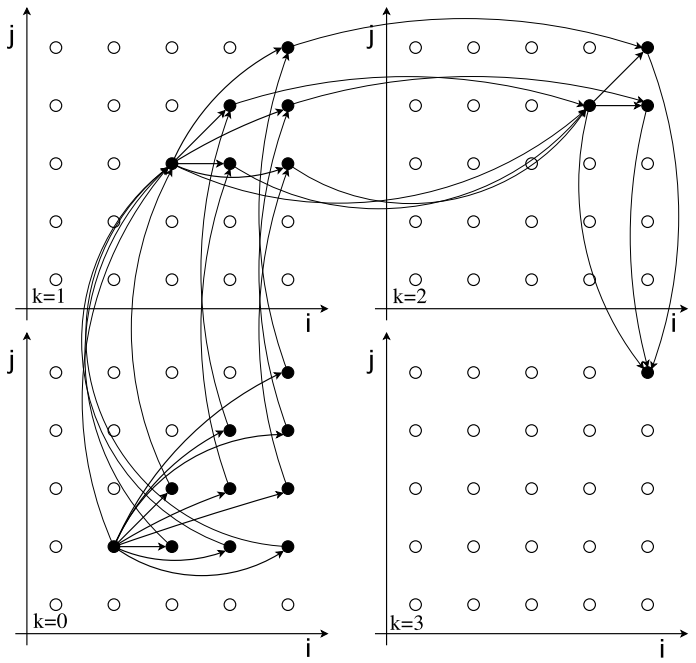
- Εξάρτηση τύπου RAW $\mathbf{A[i_1][j_1]} \rightarrow \mathbf{A[j_2][k_2]}$ Αφού αναφέρονται στο ίδιο στοιχείο, ισχύει: $i_1 = j_2, j_1 = k_2 \Rightarrow k_2 \geq k_1 + 1, j_2 \geq j_1$ και $i_2 \geq i_1 \Rightarrow k = k_2 - k_1 \geq 1, j = j_2 - j_1 \geq 0, i = i_2 - i_1 \geq 0$. Επομένως το διάνυσμα εξάρτησης είναι:

$$d_2 = (+, +, +) \succ 0$$

- Εξάρτηση τύπου RAW, WAW $\mathbf{A[i_1][k_1]} \rightarrow \mathbf{A[k_2][k_2]}$ Αφού αναφέρονται στο ίδιο στοιχείο, ισχύει: $i_1 = k_2, k_1 = k_2 \Rightarrow j_2 - 1 \geq j_1$ και $i_2 \geq j_2 \geq i_1 + 1 \Rightarrow k = k_2 - k_1 \geq 0, j = j_2 - j_1 \geq 1, i = i_2 - i_1 \geq 1$. Επομένως το διάνυσμα εξάρτησης είναι:

$$d_2 = (0, +, +) \succ 0$$

Τα διανύσματα εξάρτησης είναι όλα λεξικογραφικά θετικά, άρα όλες οι διατάξεις των βρόχων είναι επιτρεπτές κι επομένως η εφαρμογή του μετασχηματισμού tiling.



Σχήμα 4.1: Οι εξαρτήσεις δεδομένων στην παραγοντοποίηση Cholesky

Βέλτιστη διάταξη Για να βρούμε την βέλτιστη διάταξη εφαρμόζουμε τον αλγόριθμο Α΄.1, σελ.112 με τον οποίο βρίσκουμε τα κόστη των βρόχων για τις δύο δυνατές γραμμικές διατάξεις αποθήκευσης του πίνακα.

Πίνακας Αποθηκευμένος Κατά Γραμμές

Ομάδες	βρόχος k	βρόχος j	βρόχος i
$A[k][k]$	$n \times n$	-	$1 \times n$
$A[i][k]$	$\frac{n}{cls} \times n^2$	$1 \times n^2$	$n \times n^2$
$A[i][j]$	$1 \times n^2$	$\frac{n}{cls} \times n^2$	$n \times n^2$
$A[j][k]$	$\frac{n}{cls} \times n^2$	$n \times n^2$	$1 \times n^2$
Συνολικό	$\frac{2}{cls}n^3 + 2n^2$	$(\frac{1}{cls} + 1)n^3 + n^2$	$2n^3 + n^2 + n$

Πίνακας Αποθηκευμένος Κατά Στήλες

Ομάδες	βρόχος k	βρόχος j	βρόχος i
$A[k][k]$	$n \times n$	-	$1 \times n$
$A[i][k]$	$n \times n^2$	$1 \times n^2$	$\frac{n}{cls} \times n^2$
$A[i][j]$	$1 \times n^2$	$n \times n^2$	$\frac{n}{cls} \times n^2$
$A[j][k]$	$n \times n^2$	$\frac{n}{cls} \times n^2$	$1 \times n^2$
Συνολικό	$2n^3 + 2n^2$	$(\frac{1}{cls} + 1)n^3 + n^2$	$\frac{2}{cls}n^3 + n^2 + n$

Από τα παραπάνω εκτιμόμενα κόστη και βάση του λήμματος Α΄.1, σελ.114 συμπεραίνουμε ότι η βέλτιστη διάταξη των βρόχων είναι:

- (i, j, k) για αποθήκευση κατά γραμμές

- (k, j, i) για αποθήκευση κατά στήλες

Επομένως, ο κώδικας του μετροπρογράμματος για αποθήκευση κατά στήλες δεν χρειάζεται κάποια μετατροπή, ενώ για αποθήκευση κατά γραμμές θα πρέπει να εφαρμοσθεί μετάθεση στη διάταξη βρόχων. Ο πίνακας μετασχηματισμού για τη μετάθεση (i, j, k) είναι ο ακόλουθος:

$$T_{ijk} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}, T_{ijk}^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Στηρίζομενοι στην εξίσωση 2.1, σελ. 34 ο μετασχηματισμένος χώρος επαναλήψεων βρίσκεται από το ακόλουθο σύστημα ανισώσεων:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} \leq \begin{pmatrix} N-1 \\ N-1 \\ N-1 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

Λύνοντας το σύστημα αυτό προκύπτουν τα νέα όρια των βρόχων για τη διάταξη (i, j, k):

$$\begin{aligned} 1 \leq \quad & i \quad \leq N-1 \\ 1 \leq \quad & j \quad \leq i \\ 0 \leq \quad & k \quad \leq j-1 \end{aligned}$$

Επειδή το σύστημα φωλιασμένων βρόχων δεν είναι τέλεια φωλιασμένο, μετατρέπεται σε τέλεια φωλιασμένο χρησιμοποιώντας το μετασχηματισμό βύθισης κώδικα (βλέπε 2.1.1, σελ.27) για να μπορεί να εφαρμοστεί ο μετασχηματισμός της μετάθεσης. Το νέο σώμα του συστήματος φωλιασμένων βρόχων είναι:

```
if (k+1==i && j==k+1) A[k][k]=sqrt(A[k][k]);
if (j==k+1) A[i][k]=A[i][k]/A[k][k];
A[i][j]=A[i][j]-A[i][k]*A[j][k];
```

Επειδή οι συνθήκες ελέγχου εισάγουν καθυστέρηση, προσπαθούμε να τις απαλείψουμε ή να τις απλοποιήσουμε. Πράγματι, οι δύο πρώτες πράξεις μπορούν να βγουν προς εξωτερικότερες θέσεις όπως φαίνεται και στον τελικό κώδικα στο παράρτημα B′.1, το οποίο σημαίνει ότι οι ελέγχοι θα γίνονται λιγότερες φορές κι επομένως η χρονική καθυστέρηση που εισάγουν μειώνεται. Στο B′.1.cholesky_IJK (γραμμές 51-65) και B′.1.cholesky_column_major (γραμμές 70-84) βρίσκονται οι τελικοί κώδικες για τις διάταξεις (i, j, k) και (k, j, i) αντίστοιχα.

Εφαρμογή tiling Ακολούθως, προχωράμε με την εφαρμογή του μετασχηματισμού tiling για να πάρουμε την tiled μορφή του μετροπρογράμματος. Αν και η εφαρμογή του tiling μπορεί να γίνει και στις δύο διατάξεις, οι ελέγχοι συνθηκών που υπάρχουν στην τελική tiled μορφή και οι οποίοι εισάγονται κατά τον μετασχηματισμό των μη τέλειων φωλιασμένων βρόχων σε τέλεια φωλιασμένους, απλοποιούνται περισσότερο στην περίπτωση της διάταξης (k, j, i) με

αποθήκευση κατά στήλες, παρά στην περίπτωση της (i, j, k) με αποθήκευση κατά γραμμές. Γι' αυτό και η πρώτη διάταξη προτιμάται έναντι της δεύτερης.[1]

Ακολουθεί η παρουσίαση της εφαρμογής του μετασχηματισμού tiling στη διάταξη (k, j, i). Υλοποιούμε δύο εκδόσεις της tiled μορφής: tiling μόνο στις διαστάσεις k, j και tiling σε όλες τις διαστάσεις. Η δεύτερη έκδοση απότελεί το μεταβατικό στάδιο προς τις διατάξεις ενοτήτων.

- *Tiling στις διαστάσεις k, j:* (kk, jj, k, j, i)[2]. Αρχικά το σύστημα φωλιασμένων βρόχων γίνεται τέλειο με βύθιση κώδικα. Ακολούθως εφαρμόζοται διαδοχικά strip mining και μετάθεση στους βρόχους k, j.

$\Downarrow$ Βύθιση Κώδικα $\Downarrow$

```
for ( k=0; kN<;k++)
    for ( j=k+1;j<N; j++)
        for ( i=j ; i<N; i++)
            ...
```

$\Downarrow$ strip mining στον βρόχο k $\Downarrow$

```
for ( kk=0; kk<N; kk+=Bkk)
    for ( k=kk ; k<MIN(kk+Bkk,N) ; k++)
        for ( j=k+1;j<N; j++)
            for ( i=j ; i<N; i++)
                ...
```

$\Downarrow$ $T = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ $\Downarrow$

```
for ( kk=0; kk<N; kk+=Bkk)
    for ( j=kk+1;j<N; j++)
        for ( k=kk ; k<MIN(kk+Bkk, j ) ; k++)
            for ( i=j ; i<N; i++)
                ...
```

[1] Αν και στο πλαίσιο της παρούσας εργασίας έγινε και η μελέτη της (i, j, k), επειδή η επίδοση της είναι σημαντικά χειρότερη από αυτή της (k, j, i), δεν θα ασχοληθούμε αναλυτικά με την παρουσίαση της πρώτης. Γίνεται μόνο παρουσίαση πειραματικών αποτελεσμάτων της διάταξης (k, j, i) με συνδυασμό MBaLt τα οποία δικαιολογούν την απόφαση μας αυτή.

[2] Η επιλογή για tiling στις δύο αυτές διαστάσεις, στην περίπτωση που το tiling εφαρμόζεται στις δύο από τις τρεις διαστάσεις, θεωρητικά υποστηρίζεται από το γεγονός ότι γίνονται περισσότερες αναφορές που έχουν το δείκτη k σε γραμμή, παρά το i, κι άρα υπάρχει μεγαλύτερη εκμετάλλευση της επαναχρησιμοποίησης. Επίσης, επαληθεύτηκε με πειραματικές μετρήσεις που δεν παρουσιάζονται λόγω χώρου, ότι αποτελεί τη βέλτιστη επιλογή.

$$\Downarrow \qquad \text{strip mining στον βρόχο } j \qquad \Downarrow$$

```
for (kk=0; kk<N; kk+=Bkk)
    jjlow = (int) floor((double)kk/Bjj)*Bjj;
    for (jj = jjlow; jj<N; jj+=Bjj)
        for (j=MAX(jj,kk+1); j<MIN(jj+Bjj,N); j++)
            for (k=kk; k<MIN(kk+Bkk,j); k++)
                for (i=j; i<N; i++)
                    ...
```

$$\Downarrow \qquad T = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \Downarrow$$

```
for (kk=0; kk<N; kk+=Bkk)
    jjlow = (int) floor((double)kk/Bjj)*Bjj;
    for (jj = jjlow; jj<N; jj+=Bjj)
        for (k=kk; k<MIN(kk+Bkk,N); k++)
            for (j=MAX(jj,k+1); j<MIN(jj+Bjj,N); j++)
                for (i=j; i<N; i++)
                    ...
```

- *Tiling στις διαστάσεις k, j, i:* (kk, jj, ii, k, j, i). Στον τελευταίο κώδικα γίνεται ακόμη μια φορά εφαρμογή strip mining, συγκεκριμένα στον βρόχο i και ακολούθως μετάθεση στους εσωτερικότερους βρόχους για να έρθουν στη διάταξη (k, j, i).

Ακολούθως, εφαρμόζοντας τις κατάλληλες απλοποιήσεις στους ελέγχους συνθηκών, που προστέθηκαν από τη βύθιση κώδικα, παίρνουμε τις τελικές μορφές οι οποίες παρατίθενεται στο παράρτημα B′.1.cholesky _tiled_KJ (γραμμές 90-123) και B′.1.cholesky _tiled_full (γραμμές 185-222).

Οι τελευταίοι δύο κώδικες μπορούν να βελτιστοποιηθούν περαιτέρω εφαρμόζοντας 'με το χέρι' ξεδίπλωμα βρόχων και scalar replacement. Συγκεκριμένα, ο εσωτερικότερος βρόχος i ξεδιπλώνεται ανά τετράδες και η αναφορά $A[k][j]$ αποθηκεύεται σε καταχωρητή. Οι νέοι κώδικες παρατίθενεται στο παράρτημα B′.1.cholesky _tiled_KJ_hopt (γραμμές 130-179) και B′.1.cholesky _tiled_full_hopt (γραμμές 229-281).

4.3 Σχεδίαση Tiled Μορφής με Διατάξεις Ενοτήτων και Αποδοτική Δεικτοδότηση MBaLt

Στην τελική tiled μορφή (kk, jj, ii, k, j, i), εφαρμόζονται Διατάξεις Ενοτήτων με Αποδοτική Δεικτοδότηση MBaLt για την αποθήκευση των δεδομένων. Ο λόγος για την επιλογής αυτής

της μορφής, αντίς της μορφής (kk, jj, k, j, i), είναι πρώτο το γεγονός ότι η εφαρμογή της Διεθυνσιοδότησης MBaLt είναι απλούστερη σε συστήματα φωλιασμένων βρόχων με tiling σε όλες τις διαστάεις και δεύτερο ότι οι επιδόσεις των δύο tiled μορφων είναι παραπλήσιες.

Αφού η tiled μορφή (kk, jj, ii, k, j, i) χρησιμοποιεί την κατά στήλες γραμμική διάταξη για την αποθήκευση των δεδομένων, αυτό συνεπάγεται ότι μετά την εφαρμογή της διάταξης ενοτήτων τα δεδομένα μέσα στις ενότητες θα είναι κι αυτά αποθηκευμένα κατά στήλες. Λαμβάνοντας ακόμη υπόψιν ότι το tiling εφαρμόστηκε σε όλες τις διαστάσεις, καταλήγουμε στο συμπέρασμα ότι ο μετασχηματισμός που θα πρέπει να χρησιμοποιηθεί είναι ο NN.

Στον ακόλουθο πίνακα παρουσιάζονται οι ρόλοι των δεικτών και οι κατάλληλες μάσκες:

Δείκτης	Ελέγχει	Αναφορά	Κατάλληλη Μάσκα
i, ii	γραμμές	$A[i][j], A[i][k]$	μάσκα γραμμής
j, jj	γραμμές	$A[j][k]$	μάσκα γραμμής
j, jj	στήλες	$A[i][j]$	μάσκα στήλης
k, kk	γραμμές	$A[k][k]$	μάσκα γραμμής
k, kk	στήλες	$A[i][k], A[j][k]$	μάσκα στήλης

Οι δείκτες i και ii ελέγχουν μόνο γραμμές του πίνακα A κι άρα η μάσκα που θα χρησιμοποιηθεί είναι η μάσκα γραμμής. Οι δείκτες όμως j, k, jj και kk ελέγχουν τόσο γραμμές όσο και στήλες του πίνακα A. Γι΄ αυτό και στους δείκτες αυτούς εφαρμόζεται αρχικά η μάσκα στήλης. Όταν σε κάποιο σημείο του προγράμματος είναι αναγκαία η χρήση της έκδοσης των δεικτών με μάσκα γραμμής, τότε απλώς ολισθαίνουμε τους δείκτες. Για παράδειγμα αν τα k, kk χρησιμοποιούνται ως δείκτες στήλης, τότε τα $kkC = kk >> log(\frac{N}{step}), kC = k >> log(step)$ και $kfC = kkC|kC$ μπορούν να χρησιμοποιηθούν για δεικτοδότηση γραμμών, αντικαθιστώντας στις αναφορές το kk με το kkC και το k με το kfC.

Για την παραγωγή των μασκών χρησιμοποιούμε τις βοηθητικές συναρτήσεις του παραρτήματος Β΄.3.1. Συγκεκριμένα χρησιμοποιούμε τις Β΄.3.1.create_mask (γραμμές 93-117) και Β΄.3.1.create _innermask (γραμμές 28-44) για την παραγωγή των μασκών. Επίσης η create_mask χρησιμοποιείται για τη δημιουργία των άνω ορίων. Οι συναρτήσεις Β΄.3.1.create_increment (γραμμές 49-65) και Β΄.3.1.create _tile_increment (γραμμές 70-88) χρησιμοποιούνται για τη δημιούργια των βημάτων για μετακίνηση από στοιχείο σε στοιχείο εσωτερικά του tile και μετακίνηση από tile σε tile αντίστοιχα.

Ένα σημείο που θα πρέπει να υπενθυμίσουμε αφορά το μέγεθος N, που χρησιμοποιείται για τη δημιουργία των μασκών. Όπως αναφέρεται στην ενότητα 3.5.3 το N θα πρέπει να είναι δύναμη του 2. Επειδή το πρόγραμμα μας θέλουμε να χειρίζεται και πίνακες που οι διαστάσεις τους δεν είναι απαραίτητα δυνάμεις του 2, αυτό που κάνουμε είναι να βρίσκουμε τον αμέσως μεγαλύτερο πίνακα που η διαστάση του N_m είναι δύναμη του 2. Ακολούθως, δημιουργούμε τις μάσκες χρησιμοποιώντας το νέο N_m, ενώ τα όρια βρίσκονται με εφαρμογή της μάσκας πάνω στο αρχικό N. Σημειώνεται ότι, οι συναρτήσεις create_mask και create _tile_increment λαμβάνουν υπόψιν τους την περίπτωση αυτή.

Τέλος, όπως και στην περίπτωση του tiling με γραμμικές διατάξεις, έτσι κι εδώ ο κώδικας μπορεί να βελτιστοποιηθεί περαιτέρω εφαρμόζοντας 'με το χέρι' ξεδίπλωμα βρόχων και scalar replacement. Συγκεκριμένα, ο εσωτερικότερος βρόχος i ξεδιπλώνεται ανά τετράδες και η αναφορά $A[jfC|k]$ αποθηκεύεται σε καταχωρητή.

Ο κώδικας που προκύπτει με την εφαρμογή διάταξης ενοτήτων και δεικτοδότησης MBaLt, καθώς και η βελτιστοποιημένη 'με το χέρι' έκδοση του, παρατίθονται στο παράρτημα Β΄.1.cholesky

Χαρακτηριστικό	UltraSparc II 400	Pentium III 500 (Katmai)
Συχνότητα Λειτουργίας	400MHz	500MHz
Κρυφή Μνήμη I-L1, D-L1	Ευθείας Αντιστοίχησης	Συσχέτισης Συνόλου 4-Δρόμων
Μέγεθος I-L1, D-L1	16KB	16KB
Τοποθεσία L1	στη ψηφίδα	στη ψηφίδα
Μέγεθος Γραμμής L1	32B	32B
Καθυστέρηση αστοχίας L1	8 κύκλοι	20 κύκλοι
Κρυφή Μνήμη L2	Συσχέτισης Συνόλου 4-Δρόμων	Συσχέτισης Συνόλου 4-Δρόμων
Μέγεθος L2	4MB	512KB
Τοποθεσία L2	εξωτερική	εξωτερική (στη συσκευασία)
Μέγεθος Γραμμής L2	64B	32B
Καθυστέρηση αστοχίας L2	84 κύκλοι	51 κύκλοι
Αριθμός Εγγραφών D-TLB	32	64
Συσχέτιση	Πλήρης Συσχέτισης	Ευθείας Αντιστοίχησης
Μέγεθος Σελίδας	8KB	4KB
Καθυστέρηση αστοχίας D-TLB	51 κύκλοι	5 κύκλοι

Πίνακας 4.1: Τεχνικά χαρακτηριστικά ιεραρχίας μνήμης επεξεργαστών

_tiled_full _bdl_NN (γραμμές 286-378) και B′.1.cholesky _tiled_full _bdl_NN_hopt (γραμμές 384-495).

4.4 Πειραματικά Αποτελέσματα

4.4.1 Περιβάλλον Εκτέλεσης

Για την επίδειξη της αποτελεσματικότητας της μεθόδου παρουσιάζονται πραγματικοί χρόνοι εκτέλεσης των διαφόρων εκδόσεων του μετροπρογράμματος, καθώς και αποτελέσματα που προκύπτουν από την προσομοίωση της εκτέλεσης τους στο εργαλείο SimpleScalar [4]. Οι πραγματικοί χρόνοι λήφθησαν από την εκτέλεση του μετροπρογράμματος σε δύο διαφορετικά υπολογιστικά συστήματα: Sun HPC 450 Ultra το οποίο αποτελείται από 4 επεξεργαστές UltraSparc II 400, και Linux Cluster το οποίο αποτελείται από 16 επεξεργαστές Pentium III 500 (Katmai).[3]. Οι εκτελέσεις έγιναν σε ένα κόμβο/επεξεργαστή. Στον πίνακα 4.1 παρουσιάζονται τα τεχνικά χαρακτηριστικά της ιεραρχίες μνήμης που ενδιαφέρουν.

Για την μεταγλώττιση χρησιμοποιήθηκε ο *cc* (Sun Compiler) για τον UltraSparc II και ο *gcc* (GNU Compiler) για τον Pentium III. Επειδή θέλουμε να μελετήσουμε την καθαρή επίδοση των διαφόρων μεθόδων, αρχικά μεταγλωττίζουμε τους κώδικες χωρίς επιπλέον βελτιστοποίηση (cc -xO0, gcc -O0) και ακολούθως χρησιμοποιώντας την μέγιστη δυνατή βελτιστοποίηση που παρέχει ο μεταγλωττιστής (cc -fast -xtarget=native, gcc -O2).

[3]Αποτελούν μέρος του εργαστηρίου CSLAB/ECE/NTUA Περισσότερες λεπτομέρειες για τα συστήματα στο: http://pdsg.cslab.ece.ntua.gr/cluster.html#topology

Μετασχηματισμός Βρόχων	Μετασχηματισμός Δεδομένων	Κωδική Ονομασία
Καμιά	Καμιά	raw
Βέλτιστη διάταξη βρόχων	Γραμμική διάταξη κατά γραμμές	linear_row
Βέλτιστη διάταξη βρόχων	Γραμμική διάταξη κατά στήλες	linear_col
Tiling στις διαστάσεις k, j	Γραμμική διάταξη κατά στήλες	tiled_KJ_col, tiled_KJ_col_hopt
Tiling σ′ ολες τις διαστάσεις	Γραμμική διάταξη κατά στήλες	tiled_full_col, tiled_full_col_hopt
Tiling σ′ ολες τις διαστάσεις	Διάταξη Ενοτήτων NN - MBaLT	tiled_full_bdl_NN, tiled_full_bdl_NN_hopt
Tiling σ′ ολες τις διαστάσεις	Διάταξη Ενοτήτων ZZ - MBaLT	tiled_full_bdl_ZZ

Πίνακας 4.2: Βελτιστοποιήσεις για τις οποίες παρουσιάζονται πειραματικά αποτελέσματα

Στον πίνακα 4.2 δίνονται οι διάφορες βελτιστοποιημένες εκδοχές για τις οποίες λήφθησαν και παρουσιάζονται πειραματικά αποτελέσματα. Ορισμένες βελτιστοποιήσεις συνοδεύονται από μια επιπλέον έκδοση στην οποία εφαρμόσθηκε περαιτέρω βελτιστοποίηση 'με το χέρι' και περιλαμβάνει ξεδίπλωμα βρόχων και scalar replacement όπως αναφέρεται στις ενότητες 4.2 και 4.3

4.4.2 Χρονικές Μετρήσεις

Οι πειραματικές μετρήσεις λήφθησαν για διάφορα μεγέθη προβλήματος, συγκεκριμένα για τιμές (N) στο διάστημα 16 ως 2048, με το μέγεθος του tile να κυμαίνεται από 16 μέχρι N, αποσκοπώντας τόσο στη μελέτη της επίδοσης στην περίπτωση που το μέγεθος προβλήματος χωράει στην κρυφή μνήμη, όσο και στην περίπτωση που δεν χωράει.

Στα σχήματα 4.2-4.5, σελ.77-80 παρουσιάζονται οι χρόνοι εκτέλεσης των διαφόρων εκδοχών του μετροπρογράμματος, για διαφορετικό υπολογιστικό σύστημα και επίπεδο βελτιστοποίησης μεταγλωττιστή. Οι χρόνοι εκτέλεσης για τις εκδοχές που εφαρμόζουν tiling αντιστοιχούν στο καλύτερο tile. Ο χρόνος εκτέλεσης των tiled μορφών με γραμμικές διατάξεις είναι μικρότερος κατά ένα ποσοστό $17-25\%$ από το χρόνο εκείνων που δεν εφαρμόζουν tiling. Με τη χρήση διατάξεων ενοτήτων NN και δεικτοδότησης MBaLt ο χρόνος εκτέλεσης πέφτει επιπλέον κατά ένα ποσοστό 10% και το ποσοστό αυτό είναι ακόμη μεγαλύτερο στην περίπτωση που οι κώδικες μεταγλωττίζονται με cc -fast xtarget=native (σχήμα 4.5). Εξαίρεση αποτελεί η περίπτωση του gcc -O2. Επίσης, όπως αναμενόταν η επίδοση της διάταξης ενοτήτων ZZ είναι χειρότερη κι από την επίδοση των γραμμικών διατάξεων, κυρίως λόγω των φωλιασμένων στον εσωτερικότερο βρόχο ελεγχών συνθήκων.

Όπως φαίνεται και στα σχήματα, ειδικά στο σχήμα 4.5, η επίδοση των tiled εκδοχών με γραμμικές διατάξεις είναι ασταθής με τη μεταβολή του μεγέθους της διάστασης του πίνακα κι αυτό οφείλεται κυρίως λόγω των συγκρούσεων στην κρυφή μνήμη. Τέτοιες αστάθειες μπορούν να διορθωθούν με εφαρμογή αντιγραφής δεδομένων σε προσωρινούς καταχωρητές (buffers) σε συνεχόμενες θέσεις μνήμης, όπως προτείνεται στο [10]. Τέτοια όμως φαινόμενα δεν παρατηρούνται με τις διατάξεις ενοτήτων, οι οποίες εξομαλύνουν τις αστάθειες παρουσιάζοντας με αυτό τον τρόπο μια αρκετά ομαλή επίδοση. Το γεγονός αυτό είναι αρκετά θετικό,

αφού καθιστά την εφαρμογή των διατάξεων ενοτήτων άμεση, χωρίς την χρήση περαιτέρω βοηθητικών βελτιστοποιήσεων.

Ένα άλλο σημείο, είναι η εφαρμογή βελτιστοποίησης από τον μεταγλωττιστή η οποία φαίνεται ότι δεν ευνοεί αρκετά τους κώδικες που χρησιμοποιούν διατάξεις ενοτήτων, λόγω της περιπλοκότητάς τους. Ενώ με την εφαρμογή βελτιστοποίησης από τον gcc η επίδοση του tiled κωδίκα με γραμμικές διατάξεις βελτιώνεται κατά 45%, ο κώδικας με γραμμικές διατάξεις βελτιώνεται λιγότερο, γύρω στο 30%. Η βελτιστοποίηση 'με το χέρι' είναι επομένως αναγκαία στην περίπτωση που ο κώδικας είναι ιδιαίτερα περίπλοκος και η βελτιστοποίηση του μεταγλωττιστή δεν είναι τόσο ικανοποιητική. Όπως φαίνεται και στα σχήματα, η εφαρμογή βελτιστοποίησης 'με το χέρι', συγκεκριμένα ξεδίπλωμα βρόχων και ανάθεση μεταβλητών σε καταχωρητές, επιδρά θετικά τόσο στον κώδικα με γραμμικές διατάξεις, όσο και στον κώδικα με διατάξεις ενοτήτων.

Παρόλο, που ο κώδικας που χρησιμοποιεί διατάξεις ενοτήτων σε σχέση με τον κώδικα που εφαρμόζει γραμμικές διατάξεις είναι πολυπλοκότερος και μεγαλύτερος, εν τούτις αποδίδει καλύτερα, κυρίως χάριν του αποδοτικού μηχανισμού δεικτοδότησης MBaLt βάση του οποίου οι υπολογισμοί των δεικτών γίνονται αρκετά γρήγορα χρησιμοποιώντας δυαδικούς τελεστές.

Στα σχήματα 4.6-4.7, σελ.81-82 παρουσιάζονται οι χρόνοι εκτέλεσης των tiled μορφών του μετροπρογράμματος, με γραμμικές διατάξεις και με διατάξεις ενοτήτων, συναρτήσει της διάστασης του tile, για διάφορα μεγέθη προβλήματος. Παρατηρώντας τα σχήματα, διαπιστώνουμε ότι στην περίπτωση του tiling με γραμμικές διατάξεις δεν υπάρχει ένα σταθερό μέγεθος tile που να δίνει την καλύτερη επίδοση για όλα τα μεγέθη προβλημάτων, κάτι το οποίο είναι σύμφωνο με τις παρατηρήσεις του [10]. Αντίθετα, στην περίπτωση των διατάξεων ενοτήτων το μέγεθος του tile που δίνει βέλτιστη επίδοση παραμένει σχεδόν ανεξάρτητο του μεγέθους του προβλήματος. Στην περίπτωση μας το μέγεθος αυτό είναι ίσο με 32×32 και για τα δύο υπολογιστικά συστήματα. Πράγματι, το μέγεθος 32×32 είναι το μεγαλύτερο μέγεθος tile που χωράει στην κρυφή μνήμη $L1$, αφού $32 \times 32 \times sizeof(double) = 32 \times 32 \times 8bytes = 8KB = \frac{1}{2}C_{L1} < C_{L1}$, ενώ $64 \times 64 \times sizeof(double) = 64 \times 64 \times 8bytes = 32KB = 2C_{L1} > C_{L1}$. Για την περίπτωσητ του UltraSparc II η ανάλυση αυτή είναι και σε συμφωνία με την παρατήρηση του [10] ότι σε κρυφές μνήμες ευθείας αντιστοίχησης το μέγεθος tile δεν πρέπει να ξεπερνά το $\sqrt{C_{L1}}$. Υπενθυμίζεται ότι χρησιμοποιούμε τετραγωνικά tiles, γιατί κατά αυτόν τον τρόπο, χάρη στην ομοιομορφία της δεικτοδότησης MBaLt κερδίζουμε σε υπολογιστικό χρόνο. Το γεγονός ότι υπάρχει ένα σταθερό μέγεθος tile ικανό για όλα τα μεγέθη προβλήματος είναι αρκετά σημαντικό, αφού αμβλύνει το πρόβλημα επιλογής του ορθού μεγέθους tile, από το οποίο υποφέρουν οι κώδικες tiling με γραμμικές διατάξεις.

4.4.3 Αποτελέσματα Προσομοίωσης

Για να πάρουμε ποσοτικά αποτελέσματα σχετικά με τη συμπεριφορά της ιεραρχίας μνήμης, προσομοιώσαμε την εκτέλεση των δύο εκδοχών του μετροπρογράμματος, tiled_full_col και tiled_full_bdl_NN, με τη βοήθεια του εργαλείου SimpleScalar [4], για τα χαρακτηριστικά της ιεραρχίας μνήμης του UltraSparc II. Συγκεκριμένα, μετρήθηκαν οι αστοχίες κρυφής μνήμης δεδομένων L1 (dl1.misses), οι αστοχίες ενιαίας κρυφής μνήμης L2 (ul2.misses) και οι αστοχίες του TLB δεδομένων (dtlb.misses), για μεγέθη προβλήματος N στο διάστημα 32 ως 1024, με τη διάσταση του tile να παίρνει τις τιμές 16, 32, 64 (δεν χρησιμοποιήσαμε μεγαλύτερες διαστάσεις tile, διότι από τις χρονικές μετρήσεις παρατηρούμε ότι η βέλτιστη συμπεριφορά παρατηρείται για μικρές διαστάσεις tile). Η μεταγλώττιση έγινε χωρίς τη χρήση βελτιστοποίησης

από την πλευρά του μεταγλωττιστή.

Στο σχήμα 4.8, σελ.83 παρουσιάζεται το συνολικό κόστος σε κύκλους μηχανής των αστοχιών της ιεραρχίας μνήμης για τις δύο βασικές εκδοχές του μετροπρογράμματος, tiling με γραμμικές διατάξεις, και tiling με διατάξεις ενοτήτων και δεικτοδότηση MBaLt, για τρεις διαστάσεις tile (16,32,64). Παρατηρούμε ότι οι αστοχίες κρυφής μνήμης δεδομένων L1 κυριαρχούν σε σχέση με τους άλλους τύπους αστοχίων. Οι αστοχίες L1 για την περίπτωση tiling με διατάξεις ενοτήτων MBaLt ελαχιστοποιούνται για μέγεθος tile ίσο με 32×32, το οποίο είναι σταθερό για όλα τα μεγέθη προβλήματος, κρατάει τα δεδομένα εντός της κρυφής μνήμης L1 και προκύπτει αναλυτικά όπως είδαμε στην προηγούμενη υποενότητα. Επίσης, οι διατάξεις ενοτήτων αποθηκεύοντας τα δεδομένα ενός tile σε συνεχόμενες θέσεις, μειώνουν τις αστοχίες κρυφής μνήμης λόγω συγκρούσεων οι οποίες είναι σημαντικός παράγοντας, ειδικά σε κρυφές μνήμες ευθείας αντιστοίχησης. Όσον αφορά τις αστοχίες L2, αυτές είναι κατά πολύ λιγότερες από τις L1, ειδικά λόγω του μεγάλου μεγέθους της L2, $4MB$.

Συνεχίζοντας, στην περίπτωση tiling με διατάξεις ενοτήτων MBaLt δεν παρατηρείται υπερχείληση TLB, σε αντίθεση με το tiling με γραμμικές διατάξεις που για μεγάλους πίνακες η κατάχρηση TLB αποτελεί ένα μικρό, αλλά σημαντικό ποσοστό του συνολικού κόστους. Το μέγεθος tile 32×32 το οποίο ισοδυναμεί με την ελαχιστοποίηση των αστοχιών L1, ισοδυναμεί και με το μέγεθος σελίδας $8KB$ και επομένως βρίσκεται σε συμφωνία με την ελαχιστοποίηση των αστοχιών TLB. Ο λόγος είναι ότι στην περίπτωση των διατάξεων ενοτήτων τα δεδομένα αποθηκεύονται σε συνεχόμενες θέσεις, έτσι κάθε tile χρειάζεται μια μόνο σελίδα[4], κι επομένως μόνο μια εγγραφή TLB, σ' αντίθεση με τις γραμμικές διατάξεις όπου ενδεχομένως το tile εκτείνεται σε διαφορετικές σελίδες κι άρα οι αναφορές σε διαφορετικές σελίδες είναι περισσότερες.

Στο σχήμα 4.9, σελ.84 παρουσιάζονται συγκριτικά οι αστοχίες στα διάφορα επίπεδα της ιεραρχίας μνήμης για τις δύο εκδοχές του μετροπρογράμματος. Οι αστοχίες αντιστοιχούν στο μέγεθος tile που δίνει την καλύτερη συνολική επίδοση σε κάθε περίπτωση. Με τη χρήση διατάξεων ενοτήτων οι αστοχίες L1, καθώς και οι αστοχίες TLB μειώνονται σημαντικά, μέχρι και μία τάξη μεγέθους για μεγάλους πίνακες. Όσον αφορά τις αστοχίες L2 παρατηρούνται αυξομειώσεις μεταξύ των δύο μεθόδων, για τις οποίες δεν μπορούμε να πούμε ότι επηρεάζουν την επίδοση με κάποιο σταθερό παράγοντα.

4.5 Συμπεράσματα

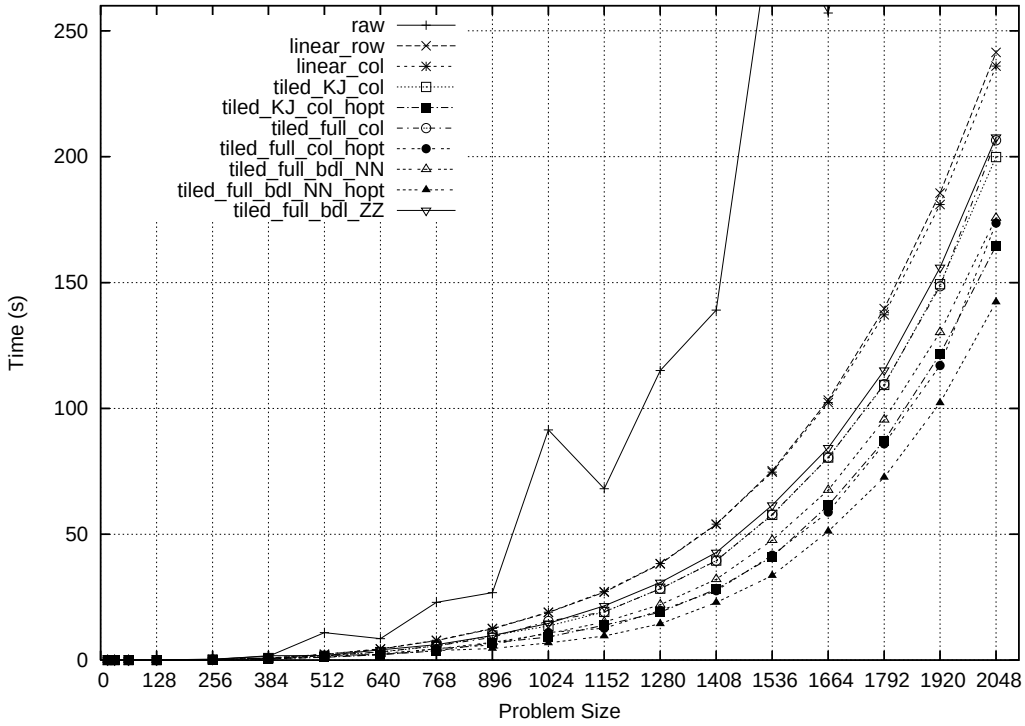
Η χρήση tiling είναι μια αρκετά διαδομένη μέθοδος για την αύξηση της τοπικότητας των δεδομένων και κατ' επέκταση για την αύξηση της επίδοσης της ιεραρχίας μνήμης των σύγχρονων υπολογιστικών συστημάτων. Η επίδοση όμως της ιεραρχίας μνήμης στους κώδικες με tiling μπορεί να αυξηθεί ακόμη περισσότερο με την ταυτόχρονη εφαρμογή διατάξεων ενοτήτων, οι οποίες αποθηκεύουν τα δεδομένα με τη σειρά που αυτά ζητούνται από τον μετασχηματισμένο με tiling κώδικα. Το πρόβλημα με τις διατάξεις ενοτήτων είναι η αποδοτική δεικτοδότηση των δεδομένων, το οποίο όμως λύνεται με την αποδοτική δεικτοδότηση MBaLt [2].

Στο παρών κεφάλαιο μελετήσαμε την επίδοση της μεθόδου, εφαρμόζοντάς την στο μετροπρόγραμμα 'Παραγοντοποίηση Cholesky' και τα αποτελέσματα ήταν αρκετά ενθαρρυντικά. Η επιπλέον βελτίωση στην επίδοση του μετροπρογράμματος από τη χρήση διατάξεων ενοτήτων

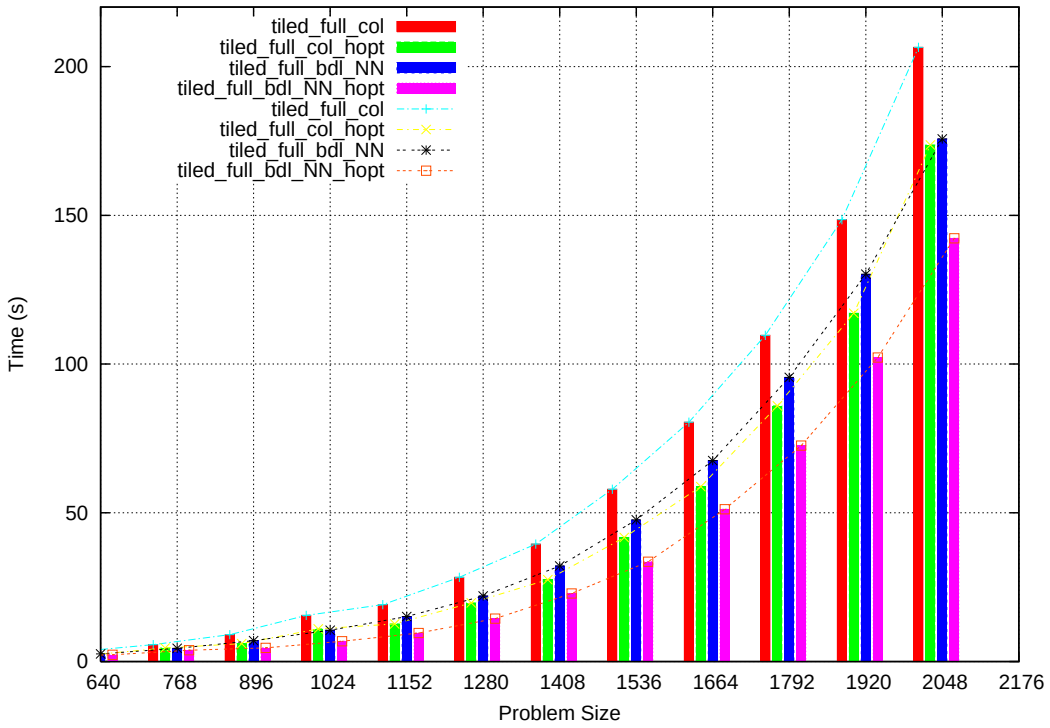
[4]Γίνεται η απλοποίηση ότι τα δεδομένα έχουν ευθυγραμμιστεί (aligned) κατά μήκος των σελίδων.

με δεικτοδότηση MBaLt κυμαίνεται γύρω από το 10%, ανάλογα με τον τύπο του υπολογιστικού συστήματος, το οποίο αποτελεί ένα σημαντικό ποσοστό βελτίωσης. Επιπλέον οι διατάξεις ενοτήτων παρουσιάζουν μια ομαλή εξέλιξη στην επίδοσή τους κατά την αύξηση του μεγέθους του προβλήματος, κάτι το οποίο δεν συμβαίνει με τις γραμμικές διατάξεις. Επίσης, το βέλτιστο μέγεθος tile στην περίπτωση διατάξεων ενοτήτων παραμένει σταθερό σε σχέση με το μέγεθος του προβλήματος, αλλά εξαρτάται απο την αρχιτεκτονική του συστήματος (κρυφή μνήμη L1), το οποίο βέβαια είναι θετικό, αφού μας γλιτώνει από το βάρος της εύρεσης του βέλτιστου μεγέθους tile.

Επίσης με τις διατάξεις ενοτήτων παρατηρείται βελτίωση στην υπερχείληση του TLB. Ένα tile πλέον δεν εκτείνεται σε αρκετές σελίδες, κι επομένως κατά την επεξεργασία του γίνονται λιγότερες αναφορές σε διαφορετικές σελίδες. Οι λιγότερες διαφορετικές αναφορές συνεπάγονται μικρότερο ενεργό σύνολο σελίδων κι άρα λιγότερες αστοχίες TLB.

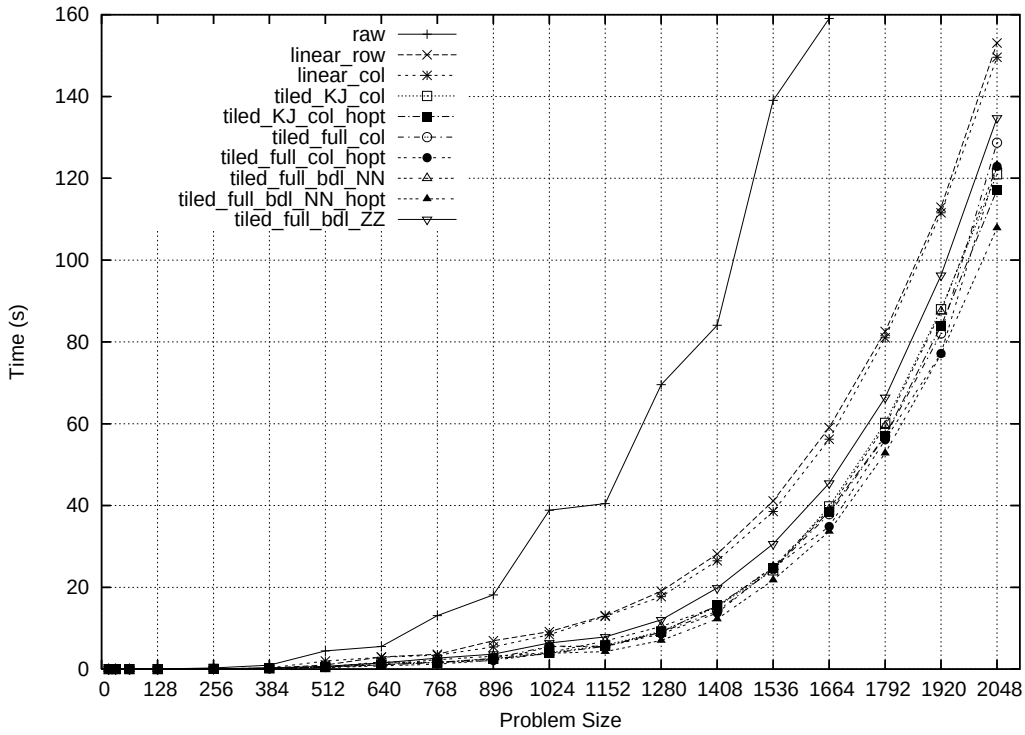


(α′) Όλες οι εκδοχές

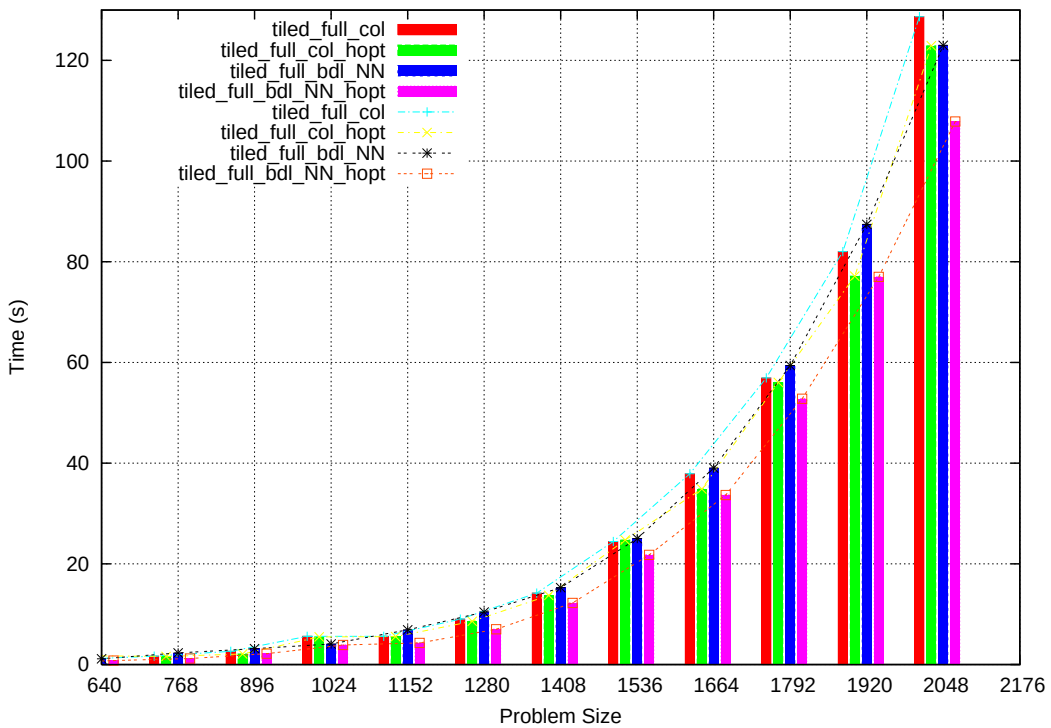


(β′) Μόνο οι tiled εκδοχές (γραμμικές διατάξεις και διατάξεις ενοτήτων)

Σχήμα 4.2: Χρόνος Εκτέλεσης στον Pentium III, -O0

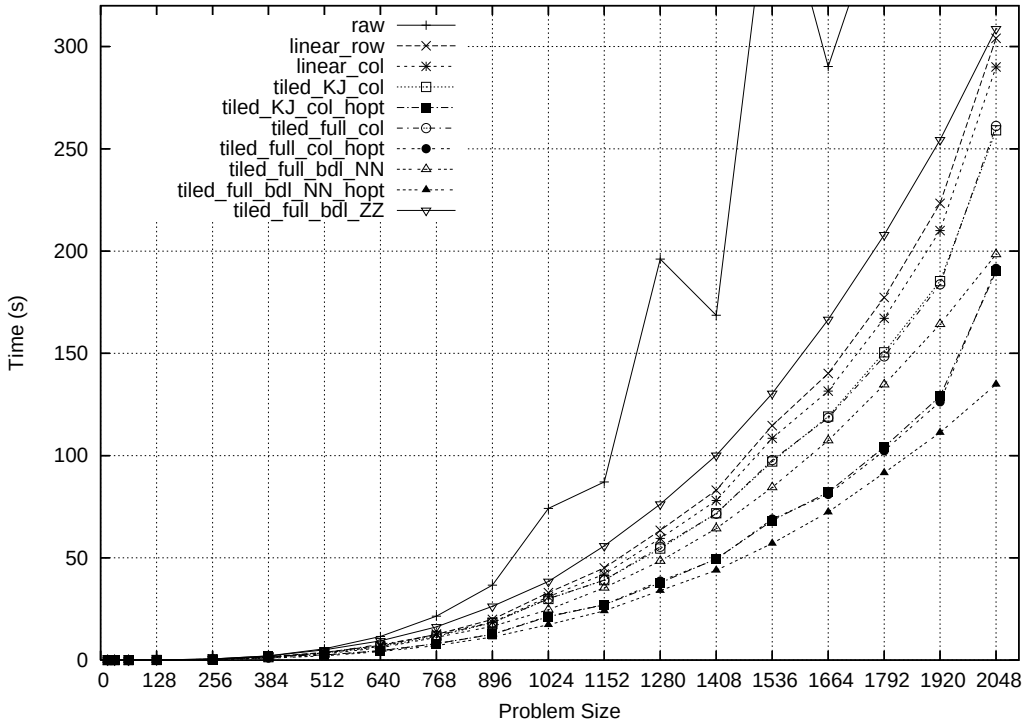


(α′) Όλες οι εκδοχές

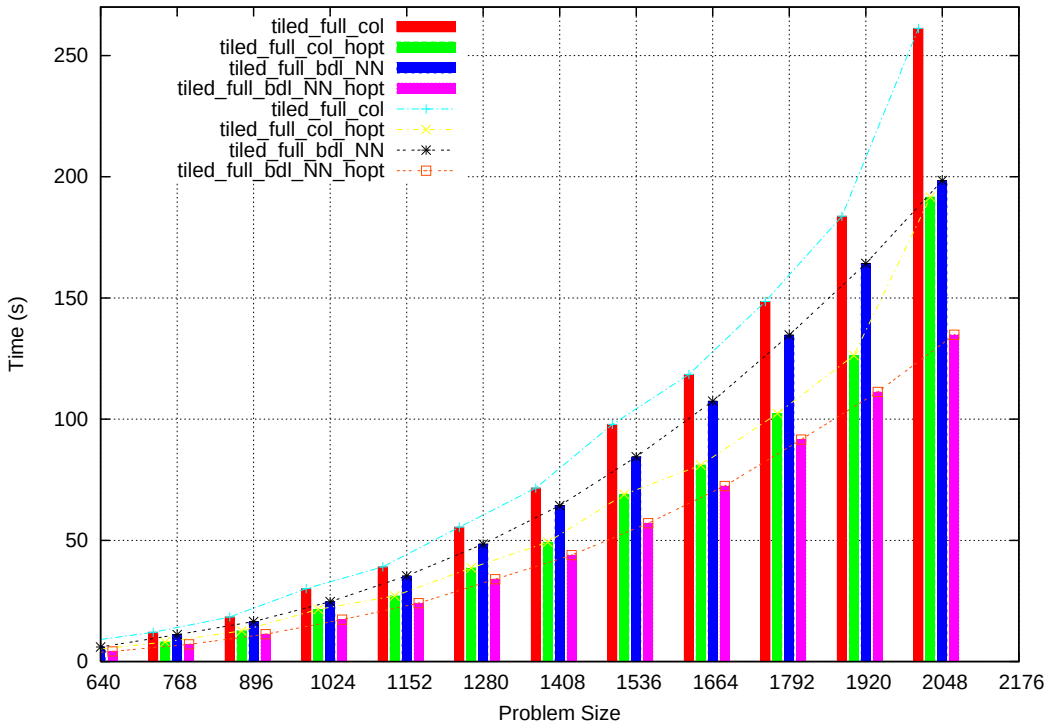


(β′) Μόνο οι tiled εκδοχές (γραμμικές διατάξεις και διατάξεις ενοτήτων)

Σχήμα 4.3: Χρόνος Εκτέλεσης στον Pentium III, -O2

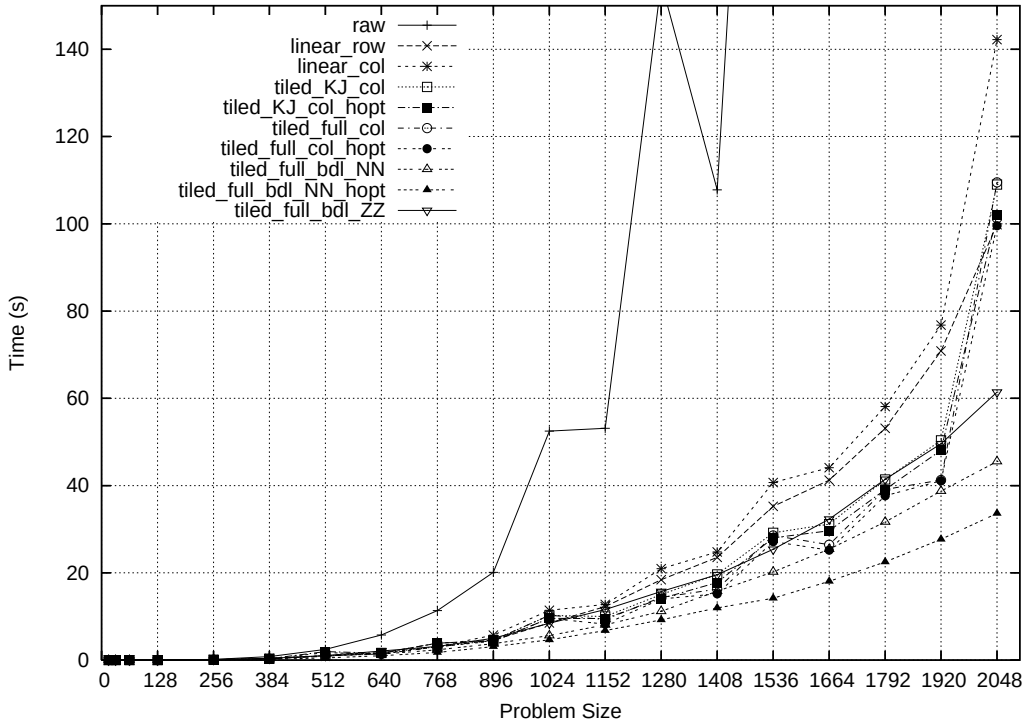


(α′) Όλες οι εκδοχές

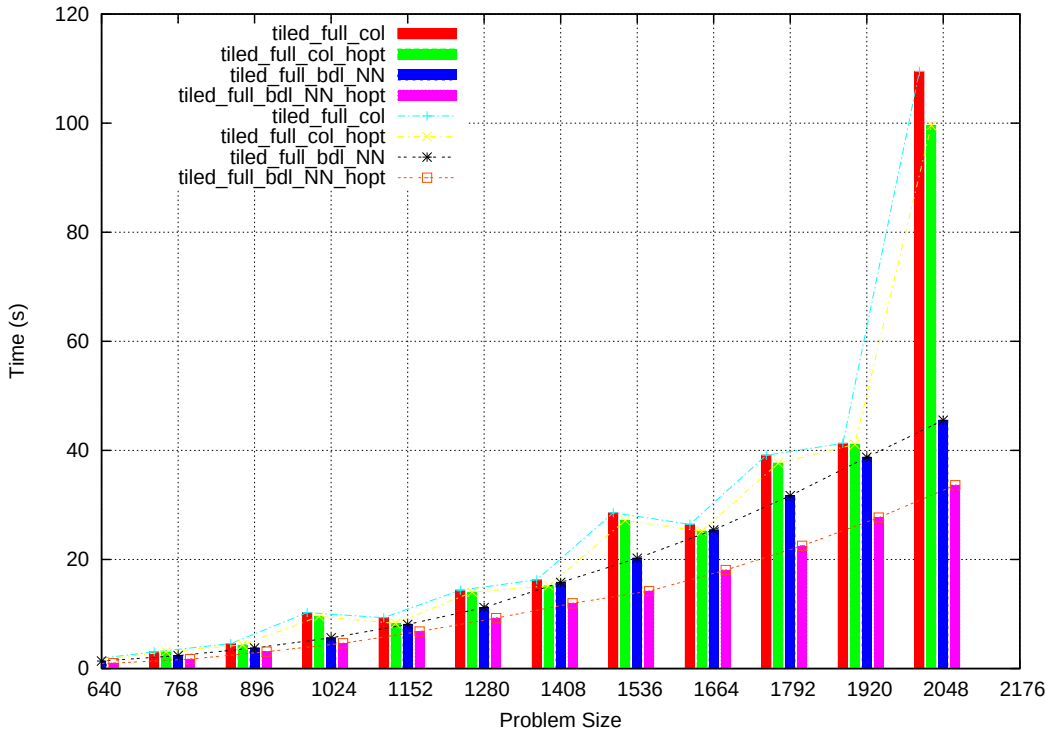


(β′) Μόνο οι tiled εκδοχές (γραμμικές διατάξεις και διατάξεις ενοτήτων)

Σχήμα 4.4: Χρόνος Εκτέλεσης στον UltraSparc II, -xO0

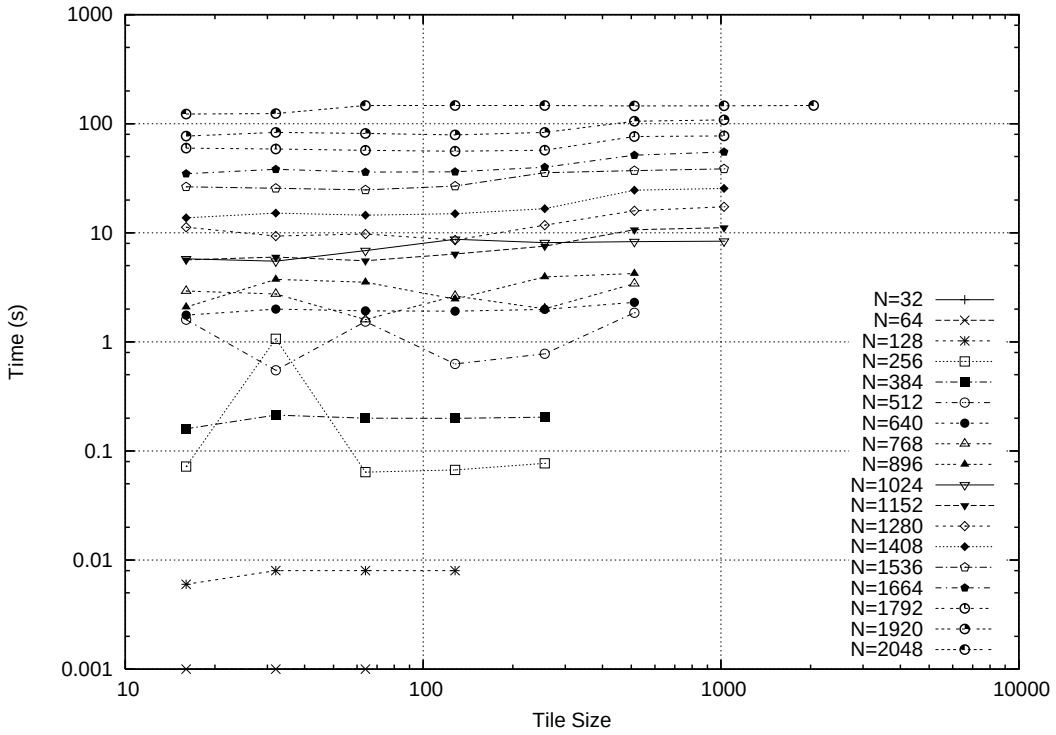


(α′) Όλες οι εκδοχές

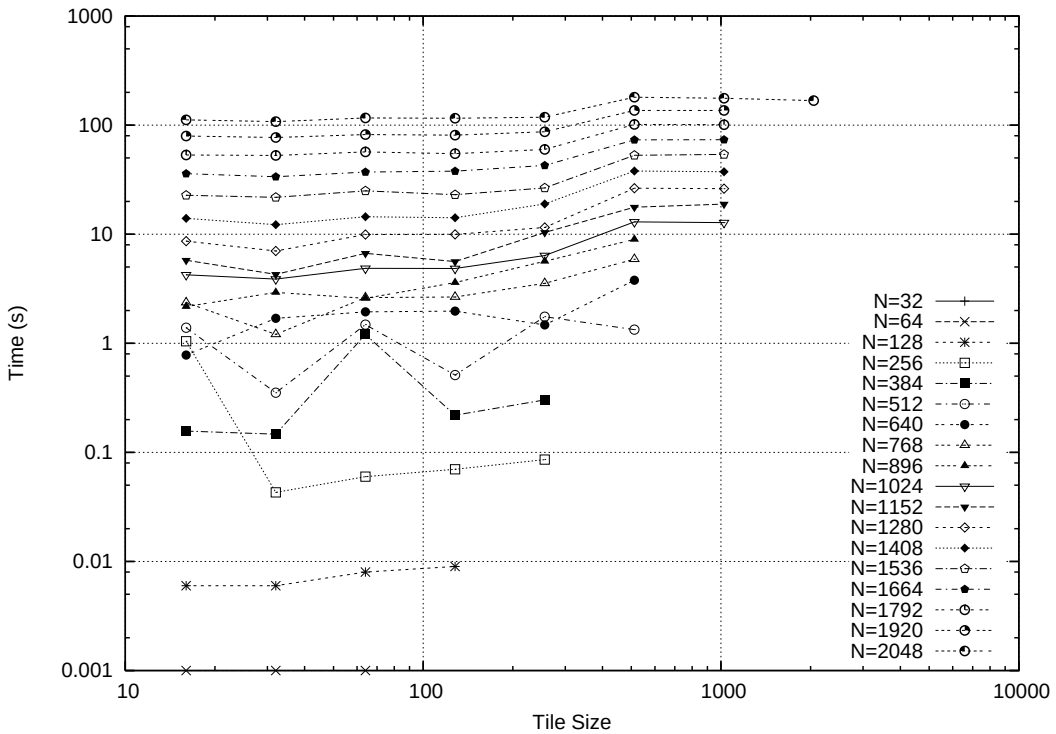


(β′) Μόνο οι tiled εκδοχές (γραμμικές διατάξεις και διατάξεις MBaLt ενοτήτων)

Σχήμα 4.5: Χρόνος Εκτέλεσης στον UltraSparc II, -fast

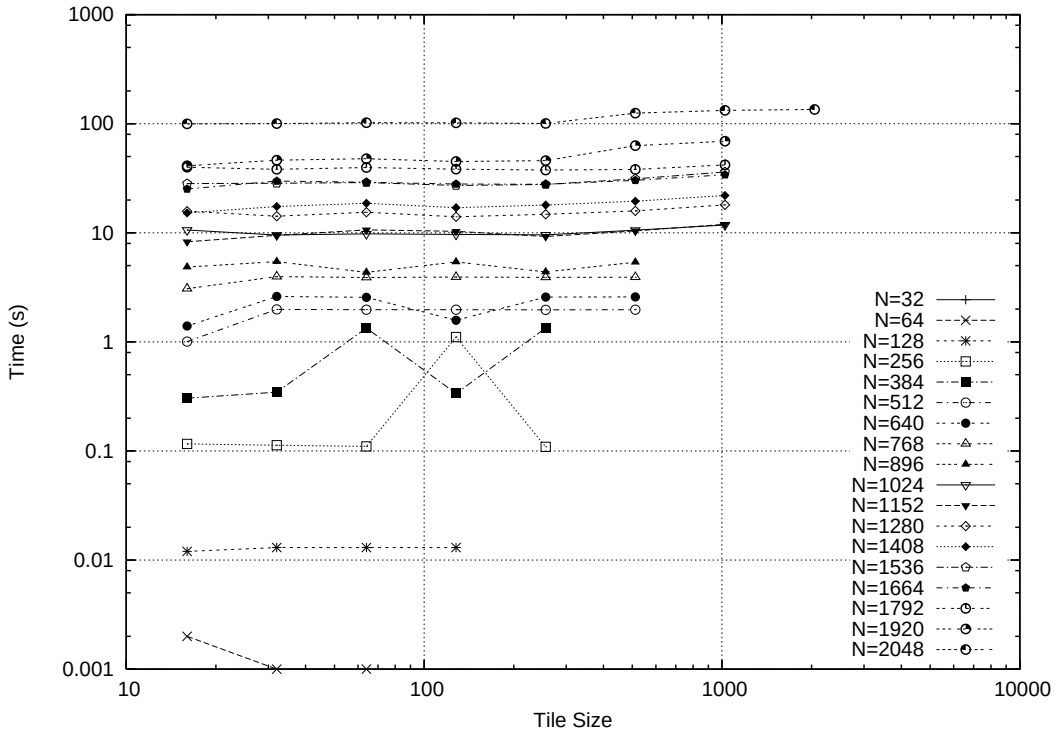


(α′) Tiling σε όλες τις διαστάσεις με γραμμικές διατάξεις

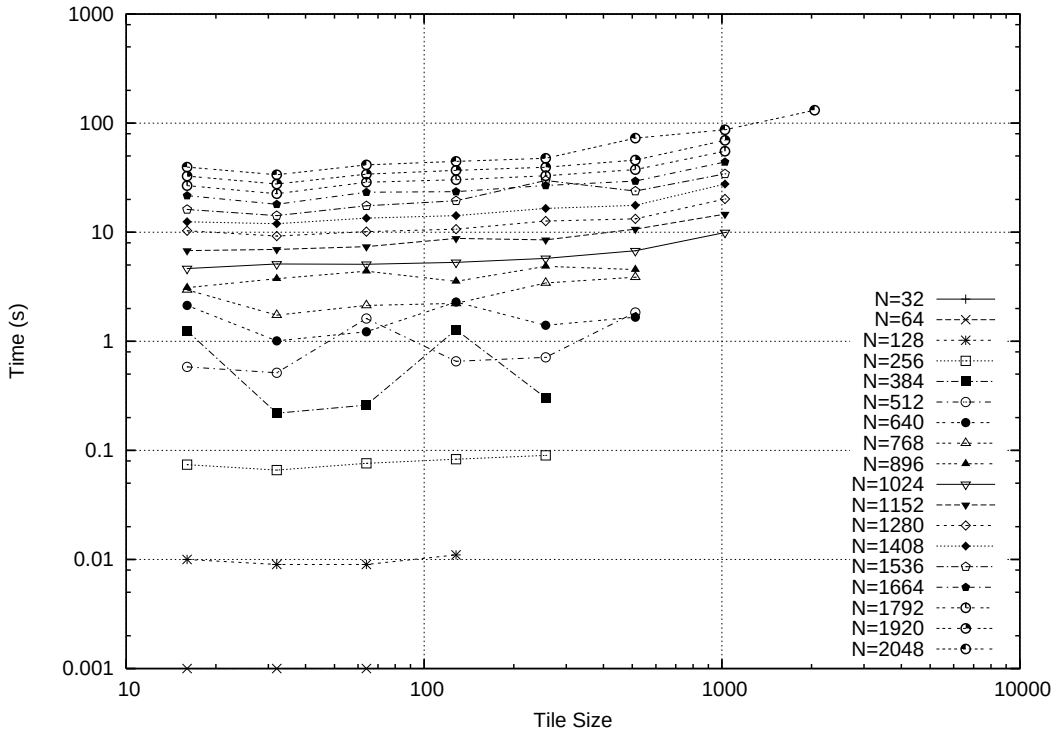


(β′) Tiling σε όλες τις διαστάσεις με διατάξεις ενοτήτων MBaLt

Σχήμα 4.6: Χρόνος Εκτέλεσης συναρτήσει Μεγέθους Tile στον Pentium III, -O2

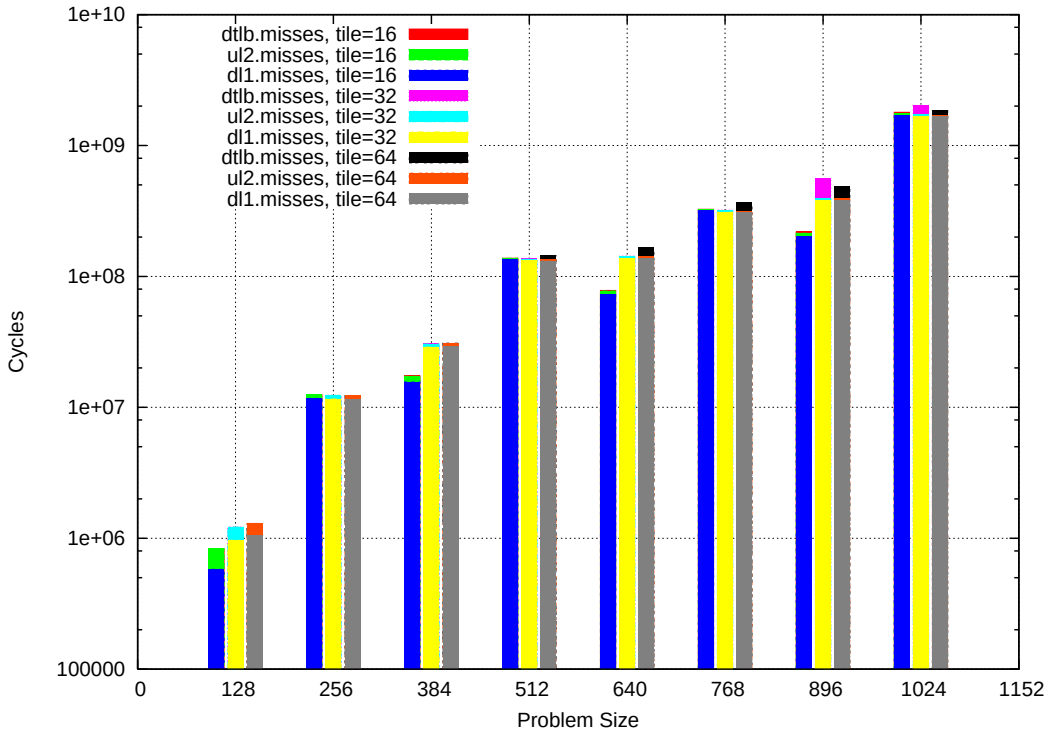


(α′) Tiling σε όλες τις διαστάσεις με γραμμικές διατάξεις

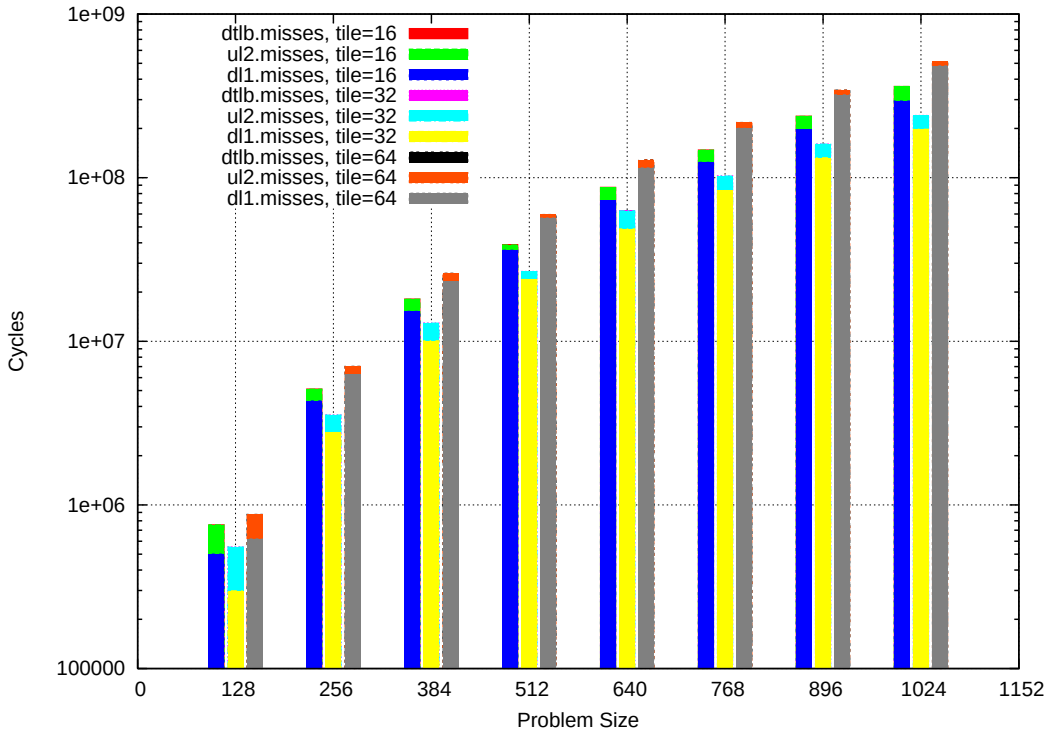


(β′) Tiling σε όλες τις διαστάσεις με διατάξεις ενοτήτων MBaLt

Σχήμα 4.7: Χρόνος Εκτέλεσης συναρτήσει Μεγέθους Tile στον UltraSparc II, -fast

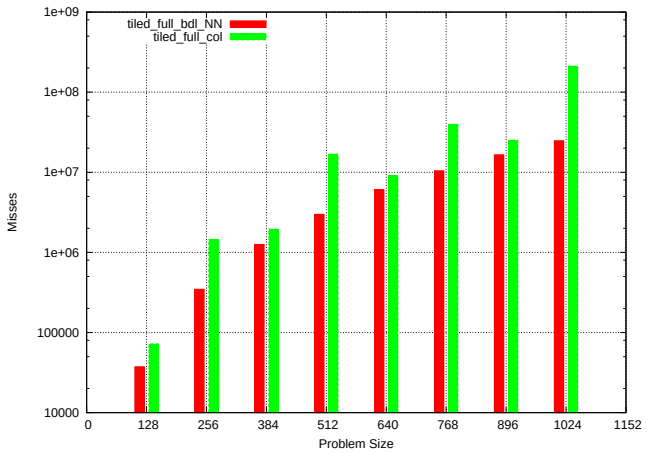


(α′) Tiling σε όλες τις διαστάσεις με γραμμικές διατάξεις

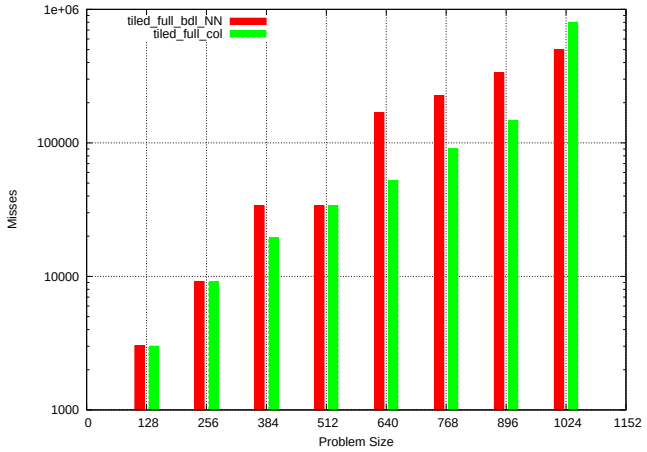


(β′) Tiling σε όλες τις διαστάσεις με διατάξεις ενοτήτων MBaLt

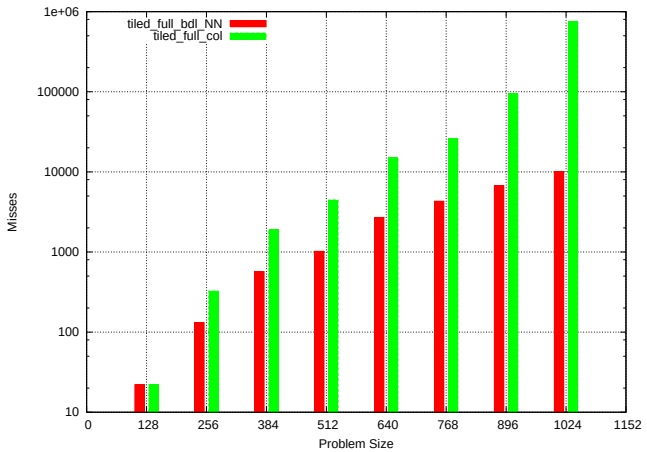
Σχήμα 4.8: Αποτελέσματα προσομοίωσης για το συνολικό κοστος αστοχιών ιεραρχίας μνήμης UltraSparc II



(α′) αστοχίες κρυφής μνήμης δεδομένων L1



(β′) αστοχίες ενιαίας κρυφής μνήμης L2



(γ′) αστοχίες TLB δεδομένων

Σχήμα 4.9: Αποτελέσματα προσομοίωσης για τις αστοχίες ιεραρχίας μνήμης UltraSparc II

Κεφάλαιο 5

Πολυνηματικές Αρχιτεκτονικές

5.1 Γενικά

Τα σύγχρονα υπολογιστικά συστήματα για να πετύχουν υψηλή επίδοση στηρίζονται σε δύο μορφές παραλληλίας: την παραλληλία σε επίπεδο εντολής (instruction-level parallelism (ILP)) και την παραλληλία σε επίπεδο νήματος (thread-level parallelism (TLP)). Οι *υπερβαθμωτοί επεξεργαστές* (superscalar processors) εκμεταλλεύονται την ILP εκτελώντας πολλαπλές εντολές από ένα πρόγραμα σε ένα κύκλο μηχανής. Από την άλλη μεριά οι πολυεπεξεργαστές εκμεταλλεύονται την TLP εκτελώντας παράλληλα διάφορετικά νήματα σε διαφορετικούς επεξεργαστές. Δυστυχώς, και οι δύο μεθόδοι παραλληλοποίησης διαμοιράζουν στατικά τους υπολογιστικούς πόρους και δεν προσαρμόζονται δυναμικά στις αυξομειώσεις της ILP και της TLP ενός προγράμματος. Χωρίς την ύπαρξη αρκετής TLP ορισμένοι επεξεργαστές θα μένουν ανενεργοί, και χωρίς την ύπαρξη αρκετής ILP η δυνατότητα των υπερβαθμωτών επεξεργαστών για εκτέλεση πολλαπλών εντολών σπαταλιέται.

Τα τελευταία χρόνια έχουν γίνει αρκετές προσπάθειες για εκμετάλλευση της TLP από υπολογιστικά συστήματα ενός επεξεργαστή, με πιο σημαντική την *ταυτόχρονη πολυνημάτωση* (simultaneous multithreading (SMT)) η οποία επιτρέπει την ύπαρξη πολλαπλών νημάτων τα οποία ανταγωνίζονται για τους πόρους του επεξεργαστή, εκμεταλλευόμενοι με αυτό τον τρόπο τόσο την ILP, όσο και την TLP. Το παρών κεφάλαιο παρουσιάζει τα βασικά στοιχεία της αρχιτεκτονικής SMT, καθώς και την τεχνολογία Hyper-Threading η οποία αποτελεί μια εμπορική υλοποίηση από την Intel.

5.2 Πολυνημάτωση

Η *πολυνημάτωση* (multithreading) είναι μια τεχνική η οποία προτάθηκε για την εκμετάλλευση της TLP από ένα επεξεργαστή και η οποία στηρίζεται στην διαμοίραση των μονάδων εκτέλεσης του επεξεργαστή ανάμεσα σε *πολλαπλά νήματα* (multiple threads). Για να είναι δυνατός ο διαμοιρασμός αυτός, θα πρέπει ο επεξεργαστής να διαθέτει επιπλέον μονάδες (καταχωρητές, μετρητής προγράμματος, πίνακας σελίδων) για την διατήρηση της κατάστασης του κάθε νήματος. Ακόμη, το υλικό θα πρέπει να υποστηρίζει την γρήγορη μεταγωγή ελέγχου ανάμεσα στα νήματα.

Υπάρχουν βασικά δύο προσεγγίσεις της πολυνημάτωσης. Στην *λεπτοκομμένη πολυνή-*

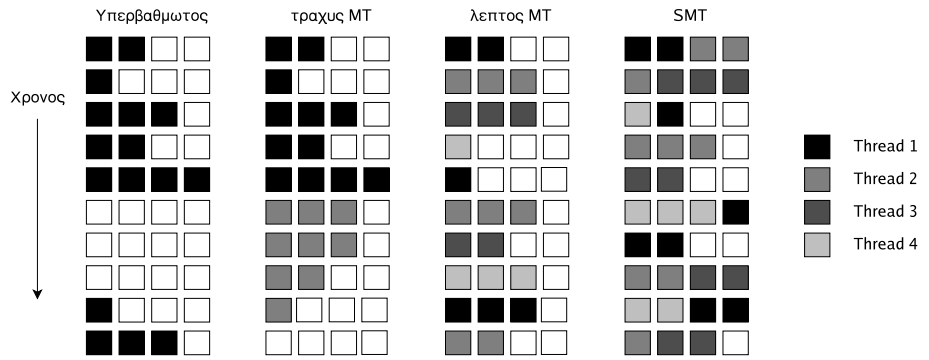
ματωση (fine-grained multithreading), η μεταγωγή ελέγχου ανάμεσα στα νήματα γίνεται εναλλάξ σε κάθε εντολή, προκαλώντας μια διαφυλλωμένη εκτέλεση των νημάτων. Πλεονέκτημα αυτής της προσέγγισης είναι η ικανότητα της να κρύβει την καθυστέρηση (latency) που προκαλούν τα κολλήματα (stalls), αφού εντολές από άλλα νήματα μπορούν να εκτελεστούν όταν είναι κολλημένο ένα νήμα. Το κυριότερο μειονέκτημά της είναι η επιβράδυνση του κάθε νήματος, λόγω της συνεχούς μεταγωγής ελέγχου. Ένα νήμα ενώ είναι έτοιμο να εκτελέσει μια σειρά από εντολές χωρίς κολλήματα, θα επιβραδυνθεί αναπόφευκτα από εντολές άλλων νημάτων.

Η δεύτερη προσέγγιση, η *χοντροκομμένη πολυνημάτωση* (coarse-grain multithreading) προσπαθεί να λύσει το μειονέκτημα της προσέγγισης της λεπτοκομμένης πολυνημάτωσης. Η μεταγωγή ελέγχου δεν γίνεται σε κάθε εντολή, αλλά μόνο στις περιπτώσεις που εμφανίζονται καθυστερήσεις μεγάλης διάρκειας, όπως είναι οι αστοχίες κρυφής μνήμης. Έτσι, η μεταγωγή δεν προκαλέι μεγάλη επιβράδυνση στην εκτέλεση του κάθε νήματος. Δεν παύει όμως να εμφανίζει κι αυτή η μέθοδος μειονεκτήματα, με πιο βασικό μειονέκτημα ότι υποφέρει από κολλήματα μικρής διάρκειας. Ο επεξεργαστής στην χοντροκομμένη πολυνημάτωση φέρνει κι εκτελεί εντολές από ένα νήμα. Όταν εμφανιστεί ένα κόλλημα, θα πρέπει ο επεξεργαστής να αδειάσει τον αγωγό του, πριν φέρει εντολές από ένα άλλο νήμα. Μέχρι την συμπλήρωση της εκτέλεσης των νέων εντολών, θα έχει περάσει ένα μικρό χρονικό διάστημα το οποίο επιβαρύνει την συνολική επίδοση του επεξεργαστή και το οποίο μπορεί να καταστεί καταστροφικό στην περίπτωση που τα κολλήματα μικρής διάρκειας εμφανίζονται συχνά.

Στο σχήμα 5.1 βλέπουμε πως χρησιμοποιούνται οι issue slots ενός υπερβαθμωτού επεξεργαστή στις περιπτώσεις των δύο προσεγγίσεων πολυνηματισμού που μόλις αναφερθήκαμε. Όπως φαίνεται και στο σχήμα η χρησιμοποίηση των issue slots στην περίπτωση του υπερβαθμωτού επεξεργαστή, περιορίζεται από την ILP. Επίπρόσθετα, μια αστοχία κρυφής μνήμης είναι ικανή να καταστήσει τον επεξεργαστή ανενεργό για σημαντικό χρόνο. Στην περίπτωση της χοντροκομμένης πολυνημάτωσης, τα διάφορα κολλήματα που εμφανίζονται κρύβονται με την εκτέλεση εντολών από άλλο νήμα. Και πάλι η ILP περιοριζει τον αριθμό εντολών που εκτελούνται ανά κύκλο μηχανής. Διακρίνεται, επίσης και η καθυστέρηση που εμφανίζεται στην εκκίνηση των νεών εντολών που εισήγχθηκαν στον αγωγό. Στην περίπτωση του λεπτοκομμένου πολυνηματισμού, η εναλλάξ εκτέλεση των νημάτων εξαφανίζει τις τυχόν άδειες issue slots κατά μήκος του κατακόρυφου άξονα, κι έτσι σε κάθε κύκλο μηχανής εκτελούνται εντολές. Αλλά και πάλι η ILP περιορίζει το πλήθος των εντολών που εκτελούνται ανα κύκλο μηχανής.

5.3 Ταυτόχρονη Πολυνημάτωση (SMT)

Η *ταυτόχρονη πολυνημάτωση* (simultaneous multithreading (SMT)) επιτρέπει την εκτέλεση πολλαπλών νημάτων σε ένα μόνο επεξεργαστή, χωρίς τη χρήση μεταγωγής. Το κίνητρο πίσω από την ταυτόχρονη πολυνημάτωση είναι το γεγονός, ότι ενώ οι σύγχρονοι επεξεργαστές έχουν την ικανότητα να φέρνουν και να εκτελούν σε κάθε κύκλο μέχρι και 8 εντολές, στην πραγματικότητα υπολειτουργούν, αφού στις περισσότερες περιπτώσεις ένα απλό νήμα που εκτελείται δεν παρουσιάζει ILP πέρα των 2-3 εντολών. Με τη χρήση όμως της ταυτόχρονης πολυνημάτωσης, ο επεξεργαστής κατά τη διάρκεια ενός κύκλου μηχανής ψάχνει και βρίσκει εντολές ταυτόχρονα από πολλά νήματα, τις οποίες δυναμικά δρομολογεί προς τις μονάδες εκτέλεσης, επιτυγχάνοντας με αυτό τον τρόπο την μέγιστη δυνατή χρησιμοποίηση των issue



Σχήμα 5.1: Χρησιμοποίηση των issue slots ενός υπερβαθμωτού επεξεργαστή

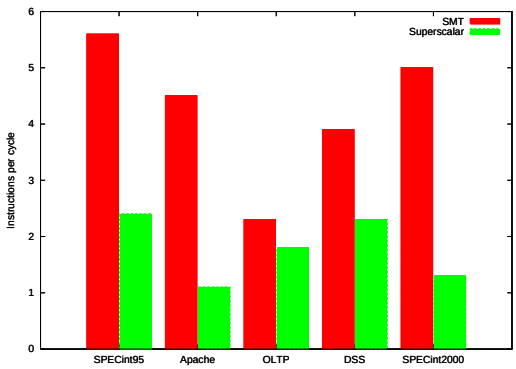
slots σε κάθε κύκλο, όπως φαίνεται και στο σχήμα 5.1.

Με την ύπαρξη λοιπόν πολλαπλών νημάτων τα οποία ανταγωνίζονται για τους πόρους του επεξεργαστή, η SMT συνδυάζει τις δυνατότητες των υπερβαθμωτών επεξεργαστών για εκτέλεση πολλαπλών εντολών (παραλληλία ILP) και τις δυνατότητες των πολυνηματικών για απόκρυψη των χρονικών καθυστερήσεων (παραλληλία TLP).

Η αρχιτεκτονική SMT όπως προτείνεται στο [11] μπορεί να εκτελέσει εντολές μέχρι κι από 8 νήματα και αποτελεί μια επέκταση του υπερβαθμωτού επεξεργαστή ο οποίος διαθέτει ήδη πολλούς από τους αναγκαίους μηχανισμούς για την εκμετάλλευση της TLP μέσω πολυνημάτωσης. Για την υποστήριξη της ταυτόχρονης πολυνημάτωσης ο υπερβαθμωτός επεξεργαστής χρειάζεται βασικά μόνο δύο αλλαγές: στο μηχανισμό εύρεσης εντολών (instruction fetch mechanism) και στο σύνολο καταχωρητών.

Ένα συνηθισμένο σύστημα πρόβλεψης διακλαδώσεων οδηγεί την εύρεση εντολών, μόνο που για να μπορεί ο επεξεργαστής να βρίσκει εντολές από 8 νήματα, διαθέτει 8 μετρητές προγράμματος και 8 δείκτες στοίβας, ένα για κάθε νήμα. Σε κάθε κύκλο, ο μηχανισμός εύρεσης εντολών επιλέγει δύο νήματα (από τα νήματα για τα οποία δεν εκκρεμεί αστοχία κρυφής μνήμης εντολών) και φέρνει μέχρι και 4 εντολές από κάθε νήμα. Ο μηχανισμός αυτός είναι λοιπόν ισοδύναμος με αυτό ενός υπερβαθμωτού επεξεργαστή πλάτους 8 εντολών, και χρειάζεται μόνο 2 θύρες στην κρυφή μνήμη εντολών. Για την πιο αποδοτική λειτουργία του μηχανισμού εύρεσης εντολών, ανατίθενται προτεραιότητες στα νήματα. Η τεχνική που εφαρμόζεται, αναθέτει την υψηλότερη προτεραιότητα στα νήματα τα οποία έχουν το μικρότερο πλήθος εντολών στα στάδια: αποκωδικοποίηση εντολών, μετονομασία καταχωρητών, ουρά εντολών προς εκτέλεση. Με αυτό τον τρόπο αποφεύγεται η λιμοκτονία ενός νήματος ή η αποκλειστική χρήση του επεξεργαστή από ένα νήμα, επιτυγχάνοντας ομοιόμορφη κατανομή των εντολών ανάμεσα στα νήματα.

Μετά την εύρεση και αποκωδικοποίηση εντολών ακολουθεί η *μετονομασία καταχωρητών*(register renaming). Κάθε νήμα διαθέτει 32 αρχιτεκτονικούς καταχωρητές οι οποίοι αντιστοιχίζονται μέσω της διαδικασίας μετονομασίας καταχωρητών στους φυσικούς καταχωρητές, οι οποίοι σε πλήθος είναι όσοι οι συνολικοί αρχιτεκτονικοί καταχωρητές ($8 \times 32 = 256$ συν κάποιους επιπλέον καταχωρητές για την μετονομασία καταχωρητών. Επειδή, το μεγαλύτερο σύνολο καταχωρητών χρειάζεται περισσότερο χρόνο προσπέλασης, τα στάδια ανάγνωσης και εγγραφής καταχωρητών επιμηκύνονται στους δύο κύκλους μηχανής για την αποφυγή αύξησης της διάρκειας του κύκλου μηχανής.



Σχήμα 5.2: Διαφορά επίδοσης ανάμεσα σε ένα SMT και ένα υπερβαθμωτό επεξεργαστή

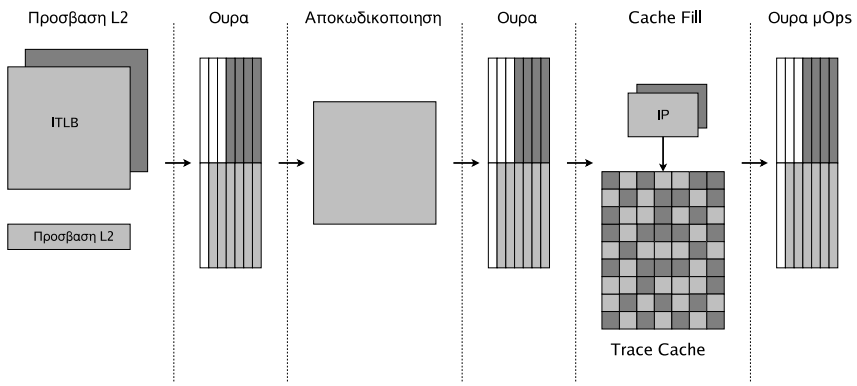
Επιπρόσθετα του νέου μηχανισμού εύρεσης εντολών, του μεγαλύτερου σύνολου καταχωρητών και του μακρύτερου αγωγού, χρειάζεται και ο πολλαπλασιασμός τριών πόρων του επεξεργαστή: του μηχανισμού αποδέσμευσης εντολων (instruction retire) ανά νήμα, του μηχανισμού διακοπών, και ένα επιπρόσθετο πεδίο στις εγγραφές του καταχωρητή πρόβλεψης διακλαδώσεων (branch prediction buffer) το οποίο δηλώνει την ταυτότητα του νήματος. Δεν χρειάζονται επιπλέον κυκλώματα για την πολυνηματική χρονοδρομολόγηση. Αυτό συμβαίνει, διότι η διαδικασία μετονομασίας καταχωρητών εξαλείφει τις οποιεσδήποτε εξαρτήσεις καταχωρητών μεταξύ νημάτων, έτσι ώστε οι συνηθισμένες ουρές εντολών να μπορούν να χρησιμοποιηθούν για την δυναμική χρονοδρομολόγηση εντολών από διαφορετικά νήματα. Οι εντολές των νημάτων περιμένουν μέσα στις ουρές, και μια εντολή μπορεί να εκτελεστεί μόλις τα ορίσματα της είναι διαθέσιμα.

Στην σχεδίαση που προτείνεται, οι περισσότεροι πόροι του επεξεργαστή μοιράζονται δυναμικά ανάμεσα στα νήματα, ενώ ελάχιστοι μοιράζονται στατικά. Επομένως στην περίπτωση που εκτελείται ένα μόνο νήμα, οι πόροι είναι σχεδόν όλοι διαθέσιμοι. Η αρχιτεκτονική μας επιτρέπει λοιπόν να πετυχαίνουμε την επίδοση της ταυτόχρονης πολυνημάτωσης, χωρίς να επηρεάζεται η επίδοση και η σχεδίαση του σύγχρονου υπερβαθμωτού επεξεργαστή.

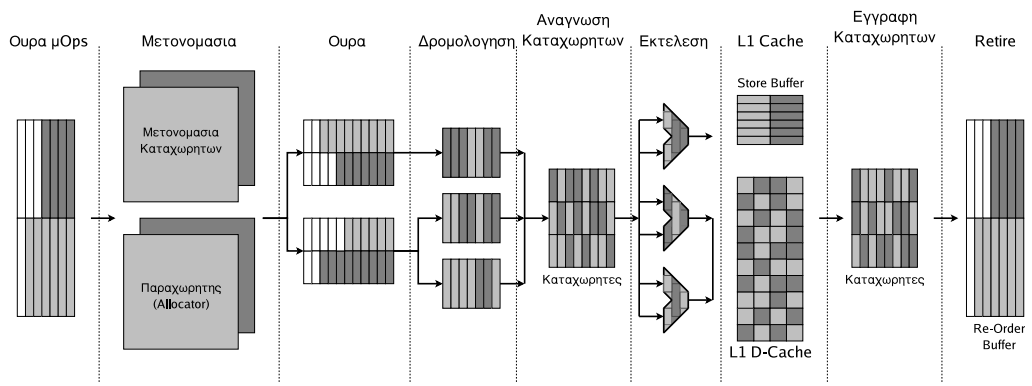
5.4 Τεχνολογία Hyper-Threading (HT)

Η τεχνολογία Hyper-Threading[13] αποτελεί την εμπορική υλοποίηση της αρχιτεκτονική SMT από την Intel.

Εφαρμόζοντας η Intel μια παραλλαγή της SMT με δύο νήματα, ένας απλός φυσικός επεξεργαστής με τεχνολογία Hyper-Threading εμφανίζεται στο λειτουργικό σύστημα ως δύο λογικοί επεξεργαστές. Κάθε λογικός επεξεργαστής έχει τη δική του αρχιτεκτονική κατάσταση. Η αρχιτεκτονική κατάσταση αποτελείται από τους καταχωρητές γενικού σκοπου, τους καταχωρητές ελέγχου, τους προγραμματίσιμους καταχωρητές διακοπών (APIC), και ορισμένους καταχωρητές κατάστασης. Από λογισμικής πλευράς, από τη στιγμή που υπάρχουν δύο ξεχωριστές αρχιτεκτονικές καταστάσεις, ένα λειτουργικό σύστημα το οποίο είναι ικανό για πολυεπεξεργαστικά περιβάλλοντα αντιμετωπίζει τον επεξεργαστή ως δύο ανεξάρτητους επεξεργαστές.



(α′) Μονοπάτι Μπροστινού Μέρους



(β′) Μονοπάτι Out-of-order

Σχήμα 5.3: Διαμοιρασμός των πόρων σε ένα επεξεργαστή τεχνολογίας Hyper-Threading

Η βασική διαφορά ανάμεσα στην Hyper-Threading υλοποίηση και την προτεινόμενη αρχιτεκτονική SMT εντοπίζεται στον τρόπο που οι πόροι του επεξεργαστή μοιράζονται ανάμεσα στα δύο νήματα. Ενώ η αρχιτεκτονική SMT προτείνει σαν πιο αποδοτικό τον δυναμικό διαμοιρασμό των πόρων αντίς του στατικού διαχωρισμού, στην υλοποίηση της Intel ορισμένες δομές διαχωρίζονται στατικά. Βασικά δύο είναι οι λόγοι γι′ αυτή την επιλογή. Πρώτο, ο στατικός διαχωρισμός απομονώνει καλύτερα ένα νήμα από ένα άλλο που δεν συμπεριφέρεται αποδοτικά. Δεύτερο, αν και η τεχνική του στατικού διαχωρισμού δεν επεκτείνεται τόσο καλά για πολλά νήματα, όπως τα 8 νήματα της SMT, στην περίπτωση των δύο νημάτων προσεγγίζει πολύ καλά την επίδοση του δυναμικού διαμοιρασμού. Θυσιάζοντας λοιπόν ένα μικρό ποσοστό της επίδοσης, επιτυγχάνεται μια πιο απλή υλοποίηση.

Στο σχήμα 5.3 φαίνεται πως γίνεται ο διαμοιρασμός των πόρων στο μονοπάτι δεδομένων (datapath) ενός επεξεργαστή τεχνολογίας Hyper-Threading. Πόροι που διαμοιράζονται δυναμικά όπως οι μονάδες εκτέλεσης, οι κρυφές μνήμες και οι καταχωρητές διακλαδώσεων χρησιμοποιούνται από κοινού από τα δύο νήματα. Όσον αφορά τους πόρους που διαχωρίζον-

ται στατικά όπως η ουρά μικροεντολών, οι καταχωρητές φόρτωσης και αποθήκευσης, ο ROB (reorder buffer), αυτοί δεν είναι πλήρως διαθέσιμοι στους λογικούς επεξεργαστές. Κάθε λογικός επεξεργαστής μπορεί να χρησιμοποιήσει τις μισές εγγραφές από κάθε τέτοια δομή. Για παράδειγμα αν οι συνολικές εγγραφές του ROB είναι 126, τότε κάθε λογικός επεξεργαστής μπορεί να χρησιμοποιήσει το μέγιστο 63 εγγραφές.

Στην περίπτωση που μόνο ένα νήμα εκτελείται στον επεξεργαστή, ή τουλάχιστο το δεύτερο νήμα είναι σε κατάσταση ύπνωσης, θα θέλαμε όλοι οι στατικά διαχωρισμένοι πόροι να είναι διαθέσιμοι στο ένα νήμα. Η τεχνολογία Hyper-Threading δίνει αυτή τη δυνατότητα μέσω των καταστάσεων λειτουργίας του επεξεργαστή. Συγκεκριμένα υπάρχουν δύο τέτοιες καταστάσεις: κατάσταση απλής διεργασίας (single-task mode (ST)) και κατάσταση πολλαπλών διεργασιών (multi-task mode (MT)). Στην κατάσταση MT και οι δύο λογικοί επεξεργαστές είναι ενεργοί και ο διαμοιρασμός των πόρων γίνεται όπως περιγράφτηκε νωρίτερα. Στην κατάσταση ST μόνο ο ένα λογικός επεξεργαστής είναι ενεργός, και οι πόροι που ήταν διαμοιρασμένοι στην κατάσταση MT συνενώνονται έτσι ώστε ο ένας λογικός επεξεργαστής να έχει στη διάθεση του όλους τους πόρους του επεξεργαστή. Η μετάβαση από την μια κατάσταση λειτουργίας στην άλλη, γίνεται μέσω της εντολής πυρήνα HALT. Η δε επαναφορά του λογικού επεξεργαστή, που εκτέλεσε την εντολή, γίνεται μέσω διακοπής που προκαλεί ο άλλος λογικός επεξεργαστής και η οποία αφυπνίζει τον πρώτο και θέτει επομένως ολόκληρο τον επεξεργαστή στην κατάσταση MT. Η εντολή HALT, προσδιορίζεται κυρίως για χρήση από τον χρονοδρομολογητή του λειτουργικού συστήματος για την βελτίωση της επίδοσης των χρονοδρομολογούμενων διεργασιών.

Ένα άλλο σημείο άξιο σχολιασμού είναι η κοινή χρήση της κρυφής μνήμης από τους δύο λογικούς επεξεργαστές. Η κοινή χρήση μπορεί να οδηγήσει σε ανεπιθύμητες συγκρούσεις μειώνοντας την επίδοση του επεξεργαστή. Παρόλα αυτά, μπορούν να αναπτυχθούν μεθόδοι που εκμεταλλεύονται την κοινή χρήση της κρυφής μνήμης, όπως είναι για παράδειγμα το μοντέλο παραγωγού-καταναλωτή στο οποίο ο ένας λογικός επεξεργαστής αναλαμβάνει να φέρνει τα δεδομένα που θα χρειαστεί ο άλλος λογικός επεξεργαστής πριν αυτός τα χρειαστεί, έτσι ώστε να κρύβονται οι καθυστερήσεις μνήμης. Επίσης ενθαρρύνει την ανάπτυξη γρήγορων μηχανισμών συγχρονισμού για την περίπτωση παράλληλων εργασιών.

Γενικά, όπως θα δούμε με περισσότερο βάθος στο επόμενο κεφάλαιο, αυτό που διαπιστώνεται σχετικά με την επίδοση της τεχνολογίας Hyper-Threading είναι ότι συμπεριφέρεται καλύτερα σε περιβάλλοντα πολλαπλών ανομοιογενών εφαρμογών, παρά σε περιβάλλοντα ομοιογενών εφαρμογών παράλληλων υπολογισμών, λόγω του ότι οι πολλαπλές εφαρμογές τείνουν να χρησιμοποιήσουν διαφορετικούς πόρους του επεξεργαστή, σε αντίθεση με τις παράλληλες εφαρμογές των οποίων τα νήματα χρησιμοποιούν τους ίδιους πόρους.

Κεφάλαιο 6

Επίδοση Διατάξεων Ενοτήτων με Δεικτοδότηση MBaLt σε Ιεραρχία Μνήμης Πολυνηματικής Αρχιτεκτονικής

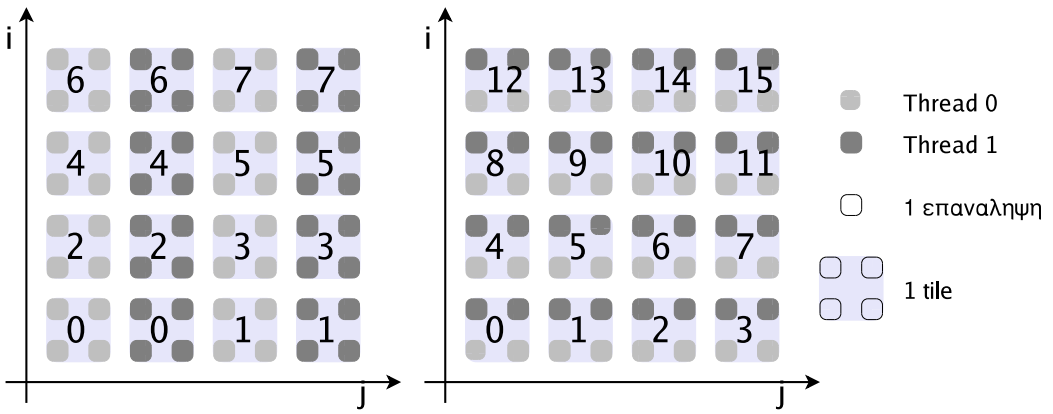
6.1 Γενικά

Αντίθετα με τους υπερβαθμωτούς επεξεργαστές, η αρχιτεκτονική SMT εκμεταλλεύεται την παραλληλία σε επίπεδο εντολής μεταξύ νημάτων, κατά την απόκρυψη καθυστερήσεων. Επομένως, είναι πιθανό οι βελτιστοποιήσεις οι οποίες είναι κατάλληλες για τέτοια μηχανήματα, να μην είναι κατάλληλες για την SMT. Στηριζόμενοι στα θετικά αποτελέσματα που μας έδωσε η εφαρμογή του tiling με διατάξεις ενοτήτων και δεικτοδότηση MBaLt, στο παρών κεφάλαιο προχωράμε στη μελέτη της επίδοσης της βελτιστοποίησης αυτής σε μια εμπορική υλοποίηση της SMT, την τεχνολογία Hyper-Threading της Intel. Οι διάφορες εκδοχές του μετροπρογράμματος 'Παραγοντοποίηση Cholesky' που υλοποιήσαμε στο κεφάλαιο 4 παραλληλοποιούνται με χρήση νημάτων Pthreads (POSIX Threads) και εκτελούνται σε ένα υπολογιστικό σύστημα αποτελούμενο από επεξεργαστές Intel Xeon οι οποίοι διαθέτουν τεχνολογία Hyper-Threading, για τη λήψη πραγματικών αποτελεσμάτων και την εξαγωγή συμπερασμάτων.

6.2 Πολυνηματική Παραλληλοποίηση

Για την διάσπαση του χώρου εργασίας ενός προγράμματος και της ανάθεσης των τμημάτων που προκύπτουν σε διαφορετικά νήματα τα οποία τρέχουν παράλληλα, χρησιμοποιούνται δύο βασικές μεθόδοι: λεπτοκομμένη διαμέριση εργασίας (fine-grain work partitioning), όπου μικρά τμήματα των δεδομένων ανατίθενται κυκλικά σε διαφορετικά νήματα και χοντροκομμένη διαμέριση εργασίας (coarse-grain work partitioning), όπου μεγάλα κομμάτια δεδομένων ανατίθενται σε κάθε νήμα.

Στην περίπτωση του tiling (και κατ' επέκταση στις διατάξεις ενοτήτων με συνδυασμό tiling), οι δύο διαμοιράσεις εφαρμόζονται ως εξής:



(α′) Χοντροκομμένη (β′) Λεπτοκομμένη

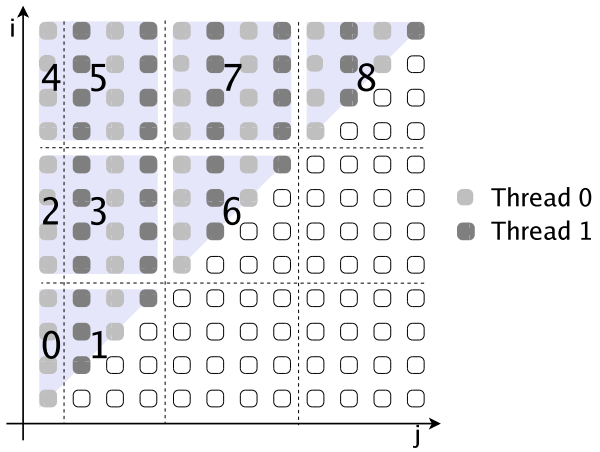
Σχήμα 6.1: Οι δύο βασικές μεθόδοι διαμέρισης χώρου εργασίας στην περίπτωση tiling

- *Χοντροκομμένη Διαμέριση Εργασίας*. Συνεχόμενα tiles ανατίθενται κυκλικά σε διαφορετικά νήματα, όπως φαίνεται στο σχήμα 6.1(α′): Η χρήση της διαμέρισης αυτής ενδείκνυται στην περίπτωση πολυεπεξεργαστών, διότι αναθέτοντας σε κάθε επεξεργαστή/νημα ένα tile αυξάνεται η επαναχρησιμοποίηση και μειώνεται η απαραίτητη επικοινωνία και συγχρονισμός μεταξύ των επεξεργαστών/νημάτων. Σ′ ένα επεξεργαστή όμως SMT, μια τέτοια διαμέριση όπως διαπιστώνεται στο [12] μπορεί να είναι επιβλαβής. Αυτό συμβαίνει διότι τα νήματα εργάζονται σε ξεχωριστά σύνολα δεδομένων, αυξάνοντας με αυτό τον τρόπο το χώρο εργασίας του προγράμματος και οδηγώντας σε υπερχείληση του TLB (TLB thrashing).

- *Λεπτοκομμένη Διαμέριση Εργασίας*: Συνεχόμενα στοιχεία ενός tile ανατίθενται κυκλικά σε διαφορετικά νήματα, όπως φαίνεται στο σχήμα 6.1(β′). Όπως σημειώνεται στο [12], η χρήση της διαμέρισης αυτής ενδείκνυται στην περίπτωση των επεξεργαστών αρχιτεκτονικής SMT, όπου τα νήματα μοιράζονται την κοινή κρυφή μνήμη κι επομένως δεν υποφέρουν από καθυστερήσεις λόγω συνάφειας κρυφής μνήμης. Επίσης είναι δυνατή η υλοποίηση αρκετά γρήγορων μηχανισμών συγχρονισμού, η οποία στηρίζεται στην ύπαρξη της κοινής κρυφής μνήμης. Επιπλέον ο χώρος εργασίας είναι μικρότερος, κι επομένως μειώνεται η υπερχείληση του TLB.

6.3 Πολυνηματική Παραλληλοποίηση Παραγοντοποίησης Cholesky

Για τη μελέτη της επίδοσης των διατάξεων ενοτήτων με δεικτοδότηση MBaLt, σε ιεραρχία μνήμης πολυνηματικής αρχιτεκτονικής τύπου SMT και συγκεκριμένα του επεξεργαστή Intel Xeon DP με τεχνολογία Hyper-Threading, το μετροπρόγραμμα Cholesky παραλληλοποιείται σε 2 νήματα χρησιμοποιώντας τις δύο βασικές μεθόδους διαμέρισης: λεπτοκομμένη και χοντροκομμένη. Επίσης εφαρμόζεται και ο συνδυασμός των δύο μεθόδων για την διαμέριση του



Σχήμα 6.2: Οι προσπελάσεις όπως γίνονται από τα νήματα στην περίπτωση λεπτοκομμένης διαμέρισης εργασίας

χώρου εργασίας στην περίπτωση 4 νημάτων.

6.3.1 Λεπτοκομμένη Διαμέριση Εργασίας

Η περίπτωση λεπτοκομμένης διαμέρισης εργασίας είναι σχετικά απλή. Και τα δύο νήματα εργάζονται στο ίδιο tile (ή ενότητα) το οποίο το μοιράζονται κυκλικά όπως φαίνεται στο σχήμα 6.2. Σε κάθε tile το πρώτο νήμα εκτελεί αν χρειάζεται τον υπολογισμό της τετραγωνικής ρίζας των στοιχείων επί της διαγωνίου (1), εκτελεί τις διαιρέσεις (2) κι ακολούθως συνεχίζει με τους πολλαπλασιαμούς στο μισό tile. Το δεύτερο νήμα εκτελεί τους πολλαπλασιασμούς στο υπόλοιπο μισό tile παράλληλα με το πρώτο νήμα. Για μικρό μέγεθος προβλήματος η διαμέριση αυτή παρουσιάζει ανισορροπίες, οι οποίες όμως σε προβλήματα μεγάλου μεγέθους είναι αμελητέες.

Ο ακόλουθος κώδικας αποτελεί τμήμα της παράλληλης υλοποίησης του μετροπρογράμματος χρησιμοποιώντας λεπτοκομμένη διαμέριση εργασίας. Ο κώδικας αυτός προσπελάνει τα στοιχεία ενός tile και είναι παραμετρικός του αριθμού νήματος *thread* και του πλήθους νημάτων *threads*. Ο πλήρης κώδικας ο οποίος περιλαμβάνει διατάξεις ενοτήτων με δεικτοδότηση MBaLt μπορεί να βρεθεί στο παράρτημα B′.2.thread _tiled_full_bdl _NN_fine_grain (γραμμές 80-217).

```
for (k=kk; k<MIN(kk+Bkk,N);k++) {
    if (thread_ID==0) {
        if (jj<=k && k<jj+Bjj  && ii<=k && k<ii+Bii)
            A[k][k]=sqrt(fabs(A[k][k]));                    (1)
        if (jj==(int) floor((double)kk/Bjj)*Bjj)
            for (j = MAX(ii,k+1); j<MIN(ii+Bjj,N); j++)
                A[k][j]/=A[k][k];                           (2)
    }

    /* Barrier Synchronization */
```

```
    for (j=MAX(jj,k+1)+thread_ID;j<MIN(jj+Bjj,N);j+=
        threads)
        for (i=MAX(ii,j);i<MIN(ii+Bii,N);i++)
            A[j][i]-=A[k][i]*A[k][j];
}
```

6.3.2 Χοντροκομμένη Διαμέριση Εργασίας

Στην περίπτωση χοντροκομμένης διαμέρισης εργασίας, τα tiles ανατίθενται κυκλικά στα 2 νήματα. Κάθε νήμα λοιπόν εργάζεται στο δικό του tile παράλληλα με το άλλο νήμα. Η ύπαρξη όμως εξαρτήσεων μεταξύ των tiles όπως φαίνεται στο σχήμα 6.3, καθιστά αναγκαίο το συγχρονισμό μεταξύ των νημάτων για την μη παραβίαση των εξαρτήσεων. Οι εξαρτήσεις υποχρεώνουν το πρώτο, τα τρία τελευταία και ορισμένες φορές το τέταρτο tile από το τέλος, να υπολογισθούν ακολουθιακά κι έτσι ανατίθενται πάντοτε στο πρώτο thread. Τα υπόλοιπα tiles ομαδοποιούνται στατικά σε ομάδες των δύο tiles και ανατίθενται στα νήματα με το μοτίβο που φαίνεται στο σχήμα 6.3. Με αυτό τον τρόπο εξασφαλίζεται η διατήρηση των εξαρτήσεων και επιπρόσθετα επιτυγχάνεται υψηλή επίδοση, αφού στα ενδιάμεσα tiles τα νήματα προχωρούν παράλληλα σχεδόν σε πλήρη ταχύτητα.

Οι λόγοι που δεν επιτυγχάνεται πλήρης ταχύτητα είναι βασικά δύο:

- Η ύπαρξη ακολουθιακής εκτέλεσης, η οποία όμως είναι αμελητέα στην περίπτωση που υπάρχουν αρκετά tiles. Στην περίπτωση μας το πλήθος των tiles δίνεται από την σχέση:

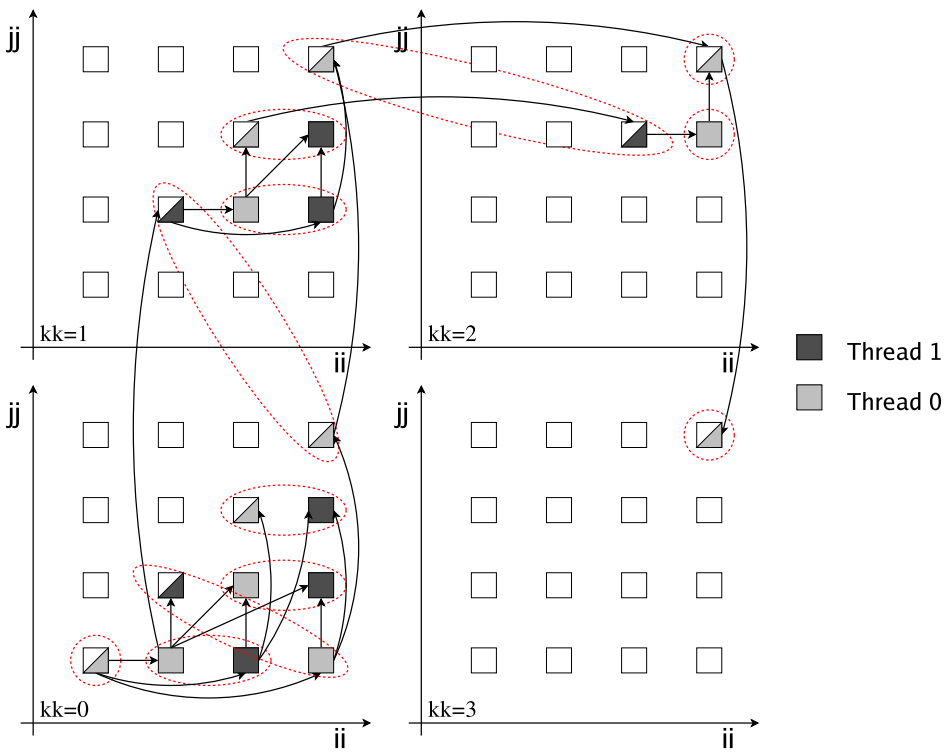
$$tiles = \sum_{k=1}^{M} \sum_{l=1}^{k} l = \frac{M(M+1)(M+2)}{6} \quad , M = \lfloor \frac{N}{T} \rfloor$$

  όπου N το μέγεθος του πίνακα και T το μέγεθος του tile. Υποθέτωντας πλήρης ισορροπία στο φορτίο των tiles η επιτάχυνση *speedup* δίνεται από την σχέση:

$$speedup = \frac{tiles}{\frac{tiles-5}{2} + 5}$$

  Στην περίπτωση που το N είναι αρκετά μεγαλύτερο του T, που αυτό είναι και το σύνηθες η επιτάχυνση $speedup \rightarrow 2$. Για παράδειγμα για μια συνηθισμένη περίπτωση $N = 1024$ και $T = 32$ $speedup = 1.998$.

- Η ανισορροπία που υπάρχει στο φορτίο των tiles. Τα tiles που βρίσκονται στα σύνορα του χώρου επαναλήψεων, χρειάζονται περισσότερο χρόνο για να ολοκληρωθούν, παρά τα εσωτερικά tiles. Έτσι, ένα νήμα αφού ολοκληρώσει το δικό του tile ενδέχεται να παραμείνει ανενεργό περιμένοντας την ολοκλήρωση του άλλου νήματος, για να συνεχίσουν ακολούθως μαζί στα επόμενα tiles. Θα μπορούσαμε να αντισταθμίσουμε τον παράγοντα της ανισσοροπίας, εάν επιτρέπαμε σε ορισμένες περιπτώσεις στα νήματα να ελέγχουν δυναμικά τις εξαρτήσεις και να προχωρούν πιο ελεύθερα. Για τους σκοπούς όμως της παρούσας εργασίας η επιτάχυνση που επιτυγχάνεται με το παραπάνω στατικό μοτίβο ανάθεσης των tiles στα νήματα είναι ικανοποιητική, και μια πιο πολύπλοκη υλοποίηση δεν χρειάζεται.



Σχήμα 6.3: Οι εξαρτήσεις μεταξύ των tiles στο χώρο επαναλήψεων της παραγοντοποίησης Cholesky

Ο κώδικας της παράλληλης υλοποίησης του μετροπρογράμματος που χρησιμοποιεί χοντροκομμένη διαμέριση εργασίας, καθώς και διατάξεις ενοτήτων με δεικτοδότηση MBaLt μπορεί να βρεθεί στο παράρτημα Β΄.2.thread _tiled_full_bdl _NN_coarse_grain (γραμμές 226-669).

6.3.3 Συνδυασμός Χοντροκομμένης και Λεπτοκομμένης Διαμέρισης

Στην περίπτωση συνδυασμού λεπτοκομμένης και χοντροκομμένης διαμέρισης εργασίας, αρχικά τα tiles ανατίθενται κυκλικά στους δύο φυσικούς επεξεργαστές χρησιμοποιώντας τη χοντροκομμένη διαμέριση. Ακολούθως, χρησιμοποιώντας λεπτοκομμένη διαμέριση εργασίας καθένα από τα δύο tile που ανατέθηκαν στους δύο φυσικούς επεξεργαστές, διαμερίζεται ανάμεσα σε 2 νήματα, τα οποία τρέχουν στους δυο λογικούς επεξεργαστές που διαθέτει ο κάθε φυσικός επεξεργαστής.

Ο κώδικας της παράλληλης υλοποίησης του μετροπρογράμματος που χρησιμοποιεί συνδυασμό λεπτοκομμένης και χοντροκομμένης διαμέρισης εργασίας, καθώς και διατάξεις ενοτήτων με δεικτοδότηση MBaLt μπορεί να βρεθεί στο παράρτημα Β΄.2.thread _tiled_full_bdl _NN_coarse_fine_grain (γραμμές 638-1041).

6.4 Θέματα Υλοποίησης

6.4.1 Νήματα POSIX

Ένα *νήμα* (thread) είναι μια ανεξάρτητη ροή ελέγχου μέσα σε μια διεργασία. Όλα τα νήματα μιας διεργασίας μοιράζονται:

- τον εικονικό χώρο διευθύνσεων της διεργασίας
- ανοικτά αρχεία (file descriptors)
- χειριστές διακοπών (signal handlers)

, ενώ κάθε νήμα έχει:

- το δικό του περιεχόμενο (καταχωρητές, μετρητής προγράμματος)
- τη δική του στοίβα για τοπικές μεταβλητές

Η βιβλιοθήκη NPTL (Native POSIX Threads Library) παρέχει τη δυνατότητα πολυνηματικού προγραμματισμού για το λειτουργικό σύστημα LInux. Αποτελεί την πιο ολοκληρωμένη και συμβατή με το πρότυπο POSIX λύση, παρέχοντας καλύτερες επιδόσεις από παλαιότερες βιβλιοθήκες πολυνηματικού προγραμματισμού, όπως η LinuxThreads.

6.4.2 CPU Affinity

Η ικανότητα του χρονοδρομολογητή ενός λειτουργικού συστήματος να *δένει* μια διεργασία σε ένα συγκεκριμένο επεξεργαστή ονομάζεται *CPU affinity*. Σε μια τέτοια περίπτωση ο χρήστης έχει τη δυνατότητα να περιορίζει το σύνολο των επεξεργαστών στους οποίους ο χρονοδρομολογητής δρομολογεί τη διεργασία προς εκτέλεση. Το σύνολο αυτό μπορεί να είναι μόνο ένας επεξεργαστής, οπότε η διεργασία θα εκτελείται μόνο στο συγκεκριμένο επεξεργαστή ή μπορεί να είναι ένα υποσύνολο των επεξεργαστών, οπότε η διεργασία θα εκτελείται σε έναν από αυτούς τους επεξεργαστές. Είναι επίσης επιθυμητό για λόγους επίδοσης, στη γενική περίπτωση που ο χρονοδρομολογητής αποφασίζει ελεύθερα την δρομολόγηση, να είναι σε θέση να δρομολογεί μια διεργασία πάντοτε στον ίδιο επεξεργαστή.

Στην περίπτωση του λειτουργικού συστήματος Linux, παρέχεται στο χρήστη η δυνατότητα αυτή μέσω της κλήσης συστήματος *sched_setaffinity*. Η κλήση αυτή δέχεται ως ορίσματα την ταυτότητα *id* της διεργασίας και μια μάσκα *affinity mask* η οποία ορίζει το σύνολο των επεξεργαστών στους οποίους η διεργασία μπορεί να εκτελεστεί. Η μάσκα αυτή είναι ένας ακέραιος αριθμός, όπου το ψηφίο k στην δυαδική του αναπαράσταση αντιστοιχεί στον επεξεργαστή k του συστήματος. Όταν το ψηφίο είναι 1 τότε η διεργασία επιτρέπεται να εκτελεστεί στον επεξεργαστή και το αντίθετο όταν είναι 0. Για παράδειγμα η μάσκα 1101 επιτρέπει την εκτέλεση μιας διεργασίας σε όλους τους επεξεργαστές πλην του επεξεργαστή 1.

Στην περίπτωση μας, το CPU affinity είναι πολύ σημαντικό, διότι μέσω αυτού ορίζουμε σε ποιους επεξεργαστές θα εκτελεστούν τα νήματα της παράλληλης υλοποίησης του μετροπρογράμματος. Όπως σε κάθε διεργασία, έτσι και σε κάθε νήμα αντιστοιχεί μια ταυτότητα νήματος την οποία μπορούμε να περάσουμε στην *sched_setaffinity* για να ορίσουμε σε ποιους επεξεργαστές επιτρέπεται να εκτελεστεί το νήμα. Στο παράρτημα Β΄.3.2 παρατίθενται οι βοηθητικές συναρτήσεις που χρησιμοποιούνται για την δήλωση του cpu affinity κάθε νήματος.

6.4.3 Μηχανισμοί Συγχρονισμού

Οι πολυνηματικές εκδοχές του μετροπρογράμματος χρησιμοποιούν μηχανισμούς συγχρονισμού για να εξασφαλίσουν την ατομικότητα των τροποποιήσεων στα κοινά δεδομένα, καθώς και για να πετύχουν τον μεταξύ τους συντονισμό στις εργασίες που έχουν να εκτελέσουν. Οι μηχανισμοί αυτοί θα πρέπει να είναι όσο το δυνατόν πιο ελαφροί από άποψη υπολογιστικών πόρων, έτσι ώστε η πολύ συχνή χρήση τους να μην επιβαρύνει σε μεγάλο βαθμό το σύστημα.

Ένας διαδομένος μηχανισμός συγχρονισμού στο χώρο της παράλληλης επεξεργασίας είναι τα *spin locks*. Σ' ένα τέτοιο κλείδωμα ο επεξεργαστής προσπαθεί συνεχώς να αποκτήσει το κλείδωμα, στριφογυρίζοντας γύρω από ένα βρόχο μέχρι να τα καταφέρει. Η χρήση τέτοιων κλειδωμάτων ενδείκνυται σε περιπτώσεις που ο προγραμματιστής περιμένει ότι θα δεσμεύσει το κλείδωμα για ένα μικρό χρονικό διάστημα κι όταν επιθυμεί η διαδικασία κλειδώματος να έχει μικρή καθυστέρηση όταν ένα τέτοιο κλείδωμα είναι διαθέσιμο. Εδώ χρησιμοποιούμε την υλοποίηση των μηχανισμών από τους [1]. Οι μηχανισμοί αυτοί είναι υλοποιημένοι σε συμβολική γλώσσα x86 και λειτουργούν εξ' ολοκλήρου στο χώρο χρήστη user space χρησιμοποιώντας κοινές μεταβλητές συγχρονισμού. Όταν οι spin locks εκτελεστούν σε επεξεργαστές που υποστηρίζουν τεχνολογία HT, προκαλούν επιβαρύνσεις στην επίδοση, κυρίως λόγω των διαδοχικών εκκενώσεων του αγωγού που προκαλούνται κατά την έξοδο από των βρόχων. Επιπρόσθετα, καταναλώνουν σημαντικούς πόρους, αφού στριφογυρίζουν γρηγορότερα σε σχέση με το χρόνο που χρειάζεται ο διάδρομος της μνήμης για να ενημερώσει την μεταβλητή συγχρονισμού. Αυτοί οι πόροι ουσιαστικά σπαταλιώνται και θα μπορούσαν να χρησιμοποιηθούν από τον άλλο λογικό επεξεργαστή. Γι' αυτό και στην υλοποίηση τους οι [1] ενσωμάτωσαν την εντολή *PAUSE*, όπως προτείνεται και από την Intel. Η εντολή αυτή εισάγει μια μικρή καθυστέρηση στο βρόχο και παγώνει την εκτέλεση του μέσα στον αγωγό, αποτρέποντας τον βρόχο να υπερκαταναλώνει πολύτιμους πόρους.

Με την εισαγωγή της εντολής *PAUSE* ουσιαστικά ο λογικός επεξεργαστής αποδεσμεύει τους πόρους που διαμοιράζονται δυναμικά. Δεν αποδεσμεύει όμως τους στατικά διαχωρισμένους πόρους. Γι' αυτό και στην υλοποίηση εισάγεται και η εντολή *HALT* η οποία όπως αναφέρεται στην ενότητα 5.4 όταν εκτελεστεί από ένα λογικό επεξεργαστή, αποδεσμεύει όλους τους πόρους που ήταν κατειλημμένοι από αυτόν, διαθέτοντας τους στον άλλο λογικό επεξεργαστή. Επειδή όμως η εντολή αυτή αποτελεί εντολή πυρήνα, ο πυρήνας του λειτουργικού συστήματος *Linux* επεκτάθηκε κατάλληλα, για να είναι δυνατή η κλήση της εντολής από το χώρο χρήστη. Με την εντολή *HALT* οι *spin locks* μπορούν να έχουν περισσότερη διάρκεια, χωρίς να σπαταλιώνται οι πόροι του επεξεργαστή. Βέβαια, η μεταγωγή από και προς την κατάσταση ύπνωσης εισάγει χρονική επιβάρυνση, η οποία όμως αντισταθμίζεται από τους πόρους που κερδίζουμε με τη χρήση της *HALT*.

Τέλος, χρησιμοποιώντας τους *spin locks* υλοποιείται μια μορφή *barriers*, η οποία στηρίζεται στο [7]. Οι barriers είναι ιδιαίτερα χρήσιμοι για τον συντονισμό των νημάτων και τον έλεγχο ροής του προγράμματος και είναι ο μηχανισμός συγχρονισμού που χρησιμοποιείται στους πολυνηματικούς κώδικες της παρούσας εργασίας.

6.5 Πειραματικά Αποτελέσματα

6.5.1 Περιβάλλον Εκτέλεσης

Οι πειραματικές μετρήσεις έγιναν στο υπολογιστικό σύστημα Xenon[1] το οποίο αποτελείτα από επεξεργαστές Intel Xeon DP οι οποίοι διαθέτουν τεχνολογία Hyper-Threading. Η οικογένεια των Intel Xeon βασίζεται στην μικροαρχιτεκτονική Netburst και είναι η πρώτη εμπορική οικογένεια επεξεργαστών που υλοποιεί την αρχιτεκτονική SMT. Στον πίνακα 6.1 παρουσιάζονται ορισμένα βασικά τεχνικά χαρακτηριστικά του επεξεργαστή, ενώ στον πίνακα 6.2 παρουσιάζονται οι αριθμητικές ταυτότητες των λογικών επεξεργαστών του υπολογιστικού συστήματος.

Για την μεταγλώττιση χρησιμοποιήθηκε ο *gcc* (GNU Compiler). Επειδή θέλουμε να μελετήσουμε την καθαρή επίδοση των διαφόρων εκδοχών, αρχικά μεταγλωττίζουμε τους κώδικες χωρίς επιπλέον βελτιστοποίηση (gcc -O0) και ακολούθως χρησιμοποιώντας την μέγιστη δυνατή βελτιστοποίηση που παρέχει ο μεταγλωττιστής (gcc -O2). Επίσης, σημειώνεται ότι για τη σύνδεση (linking) χρησιμοποιήθηκε στατική σύνδεση (static linking) η οποία σε σχέση με τη δυναμική σύνδεση (dynamic linking) δίνει γρηγορότερο, αλλά μεγαλύτερο κώδικα. Η δυναμική σύνδεση έδωσε γρηγορότερο κώδικα μόνο στην περίπτωση της σειριακής εκδοχής με διατάξεις ενοτήτων.

Για τη λήψη αναλυτικών αποτελεσμάτων που αφορούν τις αστοχίες κρυφής μνήμης χρησιμοποιούνται οι Model Specific Registers (MSRs) του επεξεργαστή, μέσω των οποίων γενικά μπορούμε να πάρουμε πληροφορίες που αφορούν γεγονότα επίδοσης. Για την ανάγνωση των καταχωρητών χρησιμοποιείται η βιβλιοθήκη και ο module που ανέπτυξαν οι [1]. Ο λόγος που χρησιμοποιείται ο module είναι διότι η εντολή rdsmr η οποία διαβάζει το περιεχόμενο ενός MSR μπορεί να εκτελεστεί μόνο από το χώρο πυρήνα.

Στον πίνακα 6.3 δίνονται οι διάφορες εκδοχές πολυνηματικής παραλληλοποίησης για τις οποίες λήφθησαν και παρουσιάζονται πειραματικά αποτελέσματα.

6.5.2 Αποτελέσματα

Στα σχήματα 6.4-6.6, σελ.103-105 παρουσιάζονται οι χρόνοι εκτέλεσης των σειριακών και πολυνηματικών εκδοχών του μετροπρογράμματος. Κι εδώ, οι χρόνοι εκτέλεσης για τις εκδοχές που εφαρμόζουν tiling αντιστοιχούν στο καλύτερο tile. Επίσης, στο σχήμα 6.8, σελ. 107 παρουσιάζονται αναλυτικά το πλήθος των αστοχιών κρυφής μνήμης L1 και L2, που αντιστοιχούν στο καλύτερο tile. Τέλος, στο σχήμα 6.7, σελ.106 παρουσιάζονται οι χρόνοι εκτέλεσης της σειριακής και των πολυνηματικών με διατάξεις ενοτήτων εκδοχών του μετροπρογράμματος και στο σχήμα 6.10, σελ.109 παρουσιάζονται οι αστοχίες κρυφής μνήμης L1, συναρτήσει της διάστασης του tile, για διάφορα μεγέθη προβλήματος.

Στο σχήμα 6.4 παρατηρούμε ότι η εφαρμογή διατάξεων ενοτήτων επιφέρει μόνο μια μικρή βελτίωση στο χρόνο εκτέλεσης, περίπου $3-4\%$. Αυτό οφείλεται κυρίως στον πολύ καλό μηχανισμό hardware prefetching που είναι εφοδιασμένος ο επεξεργαστής Xeon, ο οποίος φέρνει στην κρυφή μνήμη $L2$ δεδομένα πριν αυτά ζητηθούν. Ο μηχανισμός αυτός φαίνεται ότι βοηθά αρκετά την εκδοχή χωρίς tiling, αφού επιτυγχάνει μια επίδοση πολύ κοντά στην βελτιστοποιημένη εκδοχή με tiling και διατάξεις ενοτήτων. Επίσης, στην περίπτωση του tiling

[1] Αποτελεί μέρος του εργαστηρίου CSLAB/ECE/NTUA

Χαρακτηριστικό	Intel Xeon
Συχνότητα Λειτουργίας	2.8GHz
Βάθος Αγωγού	20 στάδια
Δυνατότητα Εκτέλεσης	6 εντολές
Δυνατότητα Παράδοσης (commit)	3 εντολές
Καταχωρητές Μετονομασίας	128
Εντολές σε Εκκρεμότητα	126
Κρυφή Μνήμη I-L1	Διαθέτει Trace Cache Συσχέτισης Συνόλου 8-Δρόμων
Μέγεθος Trace Cache	12Kupos
Κρυφή Μνήμη D-L1	Συσχέτισης Συνόλου 8-Δρόμων
Μέγεθος D-L1	16KB
Μέγεθος Γραμμής D-L1	64B
Καθυστέρηση αστοχίας L1	2 κύκλοι
Κρυφή Μνήμη L2	Συσχέτισης Συνόλου 8-Δρόμων
Μέγεθος L2	1MB
Μέγεθος Γραμμής L2	64B
Καθυστέρηση αστοχίας L2	7 κύκλοι
TLB	128 εγγραφών Συσχέτισης Συνόλου 4-Δρόμων

Πίνακας 6.1: Τεχνικά χαρακτηριστικά επεξεργαστή Intel Xeon

	Λογικός Επεξεργαστής 0	Λογικός Επεξεργαστής 1
Φυσικός Επεξεργαστής 0	0	2
Φυσικός Επεξεργαστής 1	1	3

Πίνακας 6.2: Ταυτότητες επεξεργαστών στη συστοιχεία Xenon

Διαμέριση	Πλ. Νημάτων	Κατανομή Νημάτων	Κωδική Ονομασία
Καμιά	1	1/φυσικό επεξ.	column_major, tiled_full_hopt, tiled_full_bdl_NN_hopt
Χοντροκομμένη	2	1/φυσικό επεξ.	tiled_full_smp_coarse, tiled_full_bdl_NN_smp_coarse,
Χοντροκομμένη	2	1/λογικό επεξ.	tiled_full_smt_coarse, tiled_full_bdl_NN_smt_coarse,
Λεπτοκομμένη	2	1/λογικό επεξ.	column_major_smt_fine, tiled_full_smt_fine, tiled_full_bdl_NN_smt_fine
Χοντροκομμένη και Λεπτοκομμένη	4	2/φυσικό επεξ. 1/λογικό επεξ.	tiled_full_smt_fine_smp_coarse, tiled_full_bdl_NN_smt_fine _smp_coarse

Πίνακας 6.3: Οι διάφορες εκδοχές πολυνηματικής παραλληλοποίησης για τις οποίες παρουσιάζονται πειραματικά αποτελέσματα

οι μακρύτεροι φωλιασμένοι βρόχοι οδηγούν σε περισσότερες λανθασμένες διακλαδώσεις [3], οι οποίες στην περίπτωση του Xeon είναι αρκετά κρίσιμες, αφού η ποινή των 20 κύκλων[2] είναι μεγαλύτερη κι από την αστοχία κρυφής μνήμης.

Συγκρίνοντας την επίδοση των πολυνηματικών εκδοχών του μετροπρογράμματος, παρατηρούμε ότι αντίθετα από ότι θα περίμενε κανείς, η χρήση περισσότερων νημάτων δεν επιφέρει την ανάλογη βελτίωση. Στο σχήμα 6.5 παρατηρείται ότι η επίδοση των πολυνηματικών εκδοχών σε δύο λογικούς επεξεργαστές του ίδιου φυσικού επεξεργαστή (ταυτόχρονη πολυνημάτωση - SMT), οι οποίες χρησιμοποιούν είτε χοντροκομμένη διαμέριση εργασίας, είτε λεπτοκομμένη διαμέριση εργασίας είναι ελαφρώς καλύτερη από την επίδοση της σειριακής εκδοχής κατά ένα ποσοστό 4%, η οποία δεν ικανοποιεί τις προσδοκίες μας. Επίσης, η χρησιμοποίηση λεπτοκομμένης διαμέρισης είναι ελαφρώς καλύτερη από την χοντροκομμένη διαμέριση, το οποίο είναι σε συμφωνία με την διαπίστωση του [12]. Αντίθετα, όταν η εκδοχή με χοντροκομμένη διαμέριση εκτελεστεί σε δυο λογικούς επεξεργαστές διαφορετικών φυσικών επεξεργαστών (πολυεπεξεργαστές - SMP), η επίδοση αυξάνεται σημαντικά κατά ένα παράγοντα που φτάνει το 1.85. Επεκτείνωντας παραπέρα σε 4 νήματα, τα οποία ανατίθενται στους 4 λογικούς επεξεργαστές 2 φυσικών επεξεργαστών, η επίδοση δυστυχώς δεν βελτιώνεται. Αυτο συμβαίνει κυρίως λόγω της σύγκρουσης στο μέγεθος tile που ευνοεί την χοντροκομμένη και την λεπτοκομμένη βελτίωση. Με την εισαγωγή βελτιστοποίησης από την πλευρά του μεταγλωττιστή (-O2) επιτυγχάνεται όπως φαίνεται και στο σχήμα 6.6 μια μικρή αύξηση στην επίδοση όλων των εκδοχών, με την διαφορά επίδοσης μεταξύ σειριακής εκδοχής και πολυνηματικών εκδοχών να μειώνεται.

Η ανάθεση των 2 νήματων στους λογικούς επεξεργαστές ενός φυσικού επεξεργαστή, προκαλεί όπως φαίνεται στο σχήμα 6.8(γ′) αύξηση των αστοχιών κρυφής μνήμης L1. Τα 2 νήματα μοιράζονται και ανταγωνίζονται για την κρυφή μνήμη L1 του φυσικού επεξεργα-

[2] Σε σύγχρονες αρχιτεκτονικές, η ποινή λανθασμένης πρόβλεψης διακλάδωσης είναι ίση με το βάθος του αγωγού.

στή, προκαλώντας αύξηση των αστοχιών λόγω συγκρούσεων. Παρόλα αυτά, η αύξηση των αστοχιών L1 δεν είναι αρκετά μεγάλη για να δικαιολογήσει την χαμηλή βελτίωση επίδοσης. Όπως επιβεβαιώνεται από το σχήμα 6.10 οι αστοχίες L1 μειώνονται για μέγεθος tile 32×32, το οποίο αποτελεί το μεγαλύτερο μέγεθος tile το οποίο χωρεί στην κρυφή μνήμη χωρίς να προκαλούνται συγκρούσεις μεταξύ στοιχείων του ίδιου tile στην περίπτωση της σειριακής και της λεπτοκομμένης εκδοχής, και συγκρούσεις μεταξύ των δύο tiles στην περίπτωση της χοντροκομμένης εκδοχής.[3] Όσον αφορά τις αστοχίες κρυφής μνήμης L2, όπως φαίνεται στο σχήμα 6.8(δ′) στην περίπτωση της πολυνηματικής εκδοχής με χοντροκομμένη διαμέριση οι αστοχίες παρουσιάζουν μια ελαφριά αύξηση σε σχέση με την σειριακή εκδοχή, ενώ στην περίπτωση πολυνηματικής εκδοχής με λεπτοκομμένη διαμέριση οι αστοχίες παρουσιάζουν μια ελαφριά πτώση. Με την εισαγωγή βελτιστοποίησης από την πλευρά του μεταγλωττιστή (-O2), όπως φαίνεται στο σχήμα 6.9 δεν υπάρχει ουσιαστική βελτιστοποίηση τόσο στο πλήθος των αστοχιών L1, όσο και στο πλήθος αστοχιών L2.

Οι αστοχίες κρυφής μνήμης πιθανόν κρύβονται από την ύπαρξη των 2 νημάτων, και επομένως η μικρή αύξηση τους δεν τις καθιστά υπεύθυνες για την μειωμένη απόδοση των πολυνηματικών εκδοχών του μετροπρογράμματος. Κυρίως, η μειωμένη απόδοση αποδίδεται στο κόστος συγχρονισμού σε συνδυασμό με την διαμέριση των υπολογιστικών πόρων του επεξεργαστή και τη σύγκρουση των 2 νημάτων μέσα στους υπολογιστικούς πόρους.

Φαίνεται λοιπόν ότι η υιοθέτηση τεχνικών από το χώρο της παράλληλης επεξεργασίας σε συστήματα πολυπεξεργαστών για τη διαμέριση του φορτίου και του υπολογισμού σε κώδικες που περιέχουν μεγάλο όγκο υπολογισμών του ίδιου τύπου (compute intensive applications) στην περίπτωση αρχιτεκτονικών SMT, δεν αποτελεί και την καταλληλότερη λύση. Η παραλληλοποίηση/πολυνημάτωση όπως έγινε χώρισε την εργασία σε δύο όμοια νήματα τα οποία έχουν τις ίδιες ανάγκες σε ποσότητα και τύπο υπολογιστικών πόρων. Η ταυτόχρονη πολυνημάτωση στηρίζεται στο ότι κάποιοι υπολογιστικοί πόροι που μένουν ανεκμετάλλευτοι λόγω έλλειψης ILP μπορούν να ικανοποιηθούν με την εύρεση εντολών από άλλα νήματα. Στην περίπτωσή μας όμως οι ανεκμετάλλευτοι πόροι που μένουν από το ένα νήμα δεν είναι αυτοί που χρειάζεται το άλλο νήμα, αφού και τα δύο ζητούν τους ίδιους τύπους πόρων.

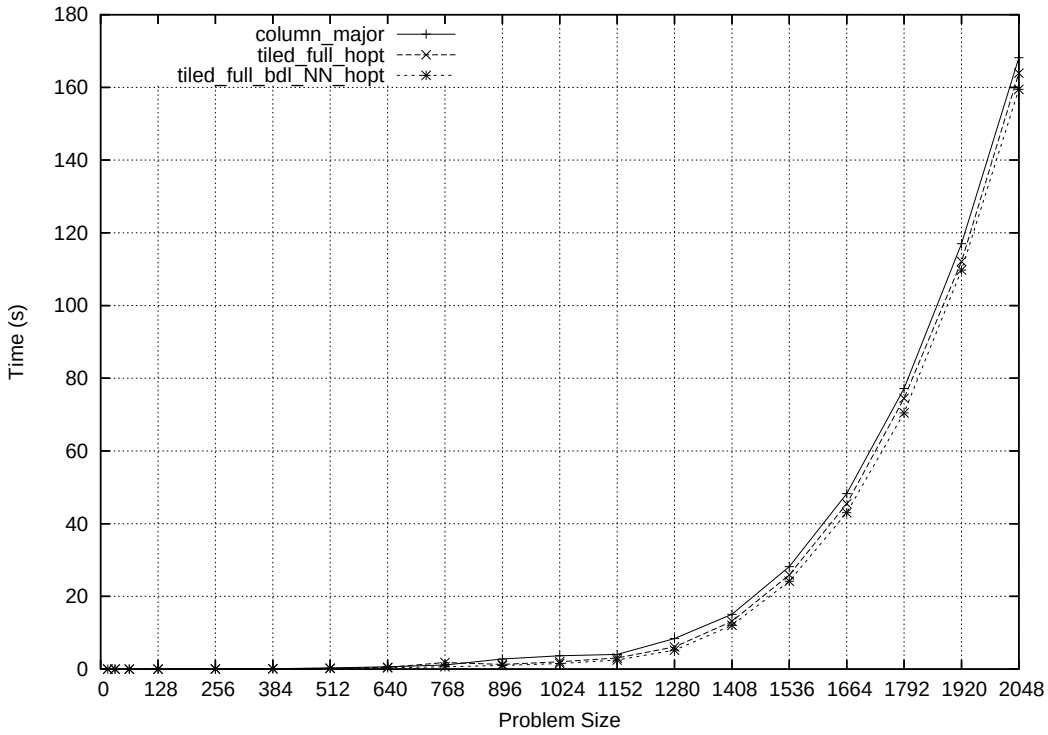
6.6 Συμπεράσματα

Συμπεραίνουμε ότι υπάρχει μια αδυναμία της τεχνολογίας Hyper-Threading και γενικότερα της αρχιτεκτονικής SMT να πετύχει υψηλή επίδοση σε πολυνηματικά παράλληλα προγράμματα με μεγάλα υπολογιστικά φορτία. Τα νήματα ενός παράλληλου προγράμματος έχοντας όμοια χαρακτηριστικά, τείνουν να ανταγωνίζονται τους ίδιους υπολογιστικούς πόρους της αρχιτεκτονικής SMT, καθιστώντας μ΄ αυτό τον τρόπο κάποιο πόρο σημείο συμφόρησης. Αυτό το φαινόμενο είναι ακόμη οξύτερο στην περίπτωση της τεχνολογίας Hyper-Threading η οποία εφαρμόζει στατική διαμέριση ορισμένων πόρων. Η αρχιτεκτονική SMT στην πράξη παρουσιάζει καλά αποτελέσματα σε ετερογενή φορτία [17], όπου η ανομοιογένεια των εντολών δείνει τη δυνατότητα στην SMT να μετατρέψει την παραλληλία επιπέδου νήματος σε παραλληλία επιπέδου εντολών.

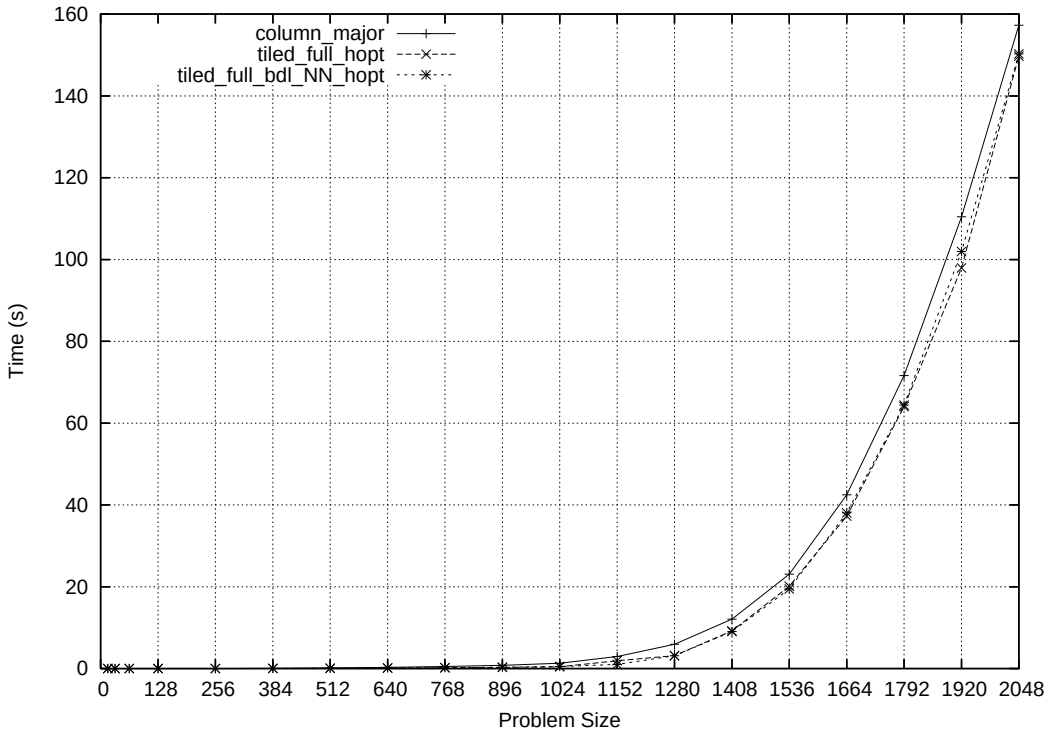
[3] Παρόλα αυτά κάποιες συγκρούσεις στην περίπτωση της πολυνηματικής εκδοχής με χοντροκομμένη διαμέριση υπάρχουν, διότι όπως έγινε η στατική διαμέριση του φορτίου μπορεί να υπάρξουν περιπτώσεις που τα δύο tiles αποθηκεύονται στις ίδιες θέσεις μέσα στην κρυφή μνήμη.

Τα βελτιστοποιημένα προγράμματα παρουσιάζουν υψηλή παραλληλία σε επίπεδο εντολής (ILP) η οποία ικανοποιεί σε μεγάλο βαθμό τις ανάγκες ενός υπερβαθμωτού επεξεργαστή, επιτυγχάνοντας κατ΄ αυτόν τον τρόπο υψηλή επίδοση. Η παραλληλοποίηση τέτοιων προγραμμάτων σε πολλαπλά νήματα και η εκτέλεση τους σε περιβάλλον αρχιτεκτονικής SMT δεν επιφέρει σημαντική βελτίωση, αφού τα νήματα που προκύπτουν παρουσιάζουν υψηλή ILP η οποία καταναλώνει τους ίδιους υπολογιστικούς πόρους, αφήνοντας ελάχιστα περιθώρια στην SMT να εκμεταλλευτεί την ύπαρξη πολλαπλών νημάτων. Έτσι, η παραλληλία που παρουσιάζει ένας SMT επεξεργαστής σ΄ αυτή την περίπτωση δεν είναι μεγαλύτερη από αυτήν που παρουσιάζει ένας υπερβαθμωτός επεξεργαστής. Επίσης, η αυξημένη τοπικότητα των δεδομένων η οποία χαρακτηρίζει τα βελτιστοποιημένα προγράμματα (μικρός αριθμός αστοχιών L1) δεν ωφελεί σε μεγάλο βαθμό τις πολυνηματικές εκδοχές, αφού η SMT μπορεί να κρύψει ένα μεγάλο αριθμό από αστοχίες L1.

Επομένως, η άμεση εφαρμογή των τεχνικών που χρησιμοποιούνται για αύξηση της ILP και της τοπικότητας των δεδομένων στους τυπικούς υπερβαθμωτούς, επεξεργαστές, όπως είναι το tiling με διατάξεις ενοτήτων και δεικτοδότηση MBaLt, δεν επιφέρει θεαματικά αποτελέσματα σε επεξεργαστές αρχιτεκτονικής τύπου SMT, οι οποίοι οφελούνται από προγράμματα με χαμηλό ILP και απλούς κώδικες εκτέλεσης. Η εφαρμογή μεθόδων παράλληλης πολυνημάτωσης, οι οποίες θα στηρίζονται σε νήματα με μη όμοια χαρακτηριστικά και ανάγκες σε διαφορετικούς τύπους υπολογιστικών πόρων, θα δίνει περισσότερες ευκαιρίες στους επεξεργαστές SMT για μετατροπή της TLP σε ILP.

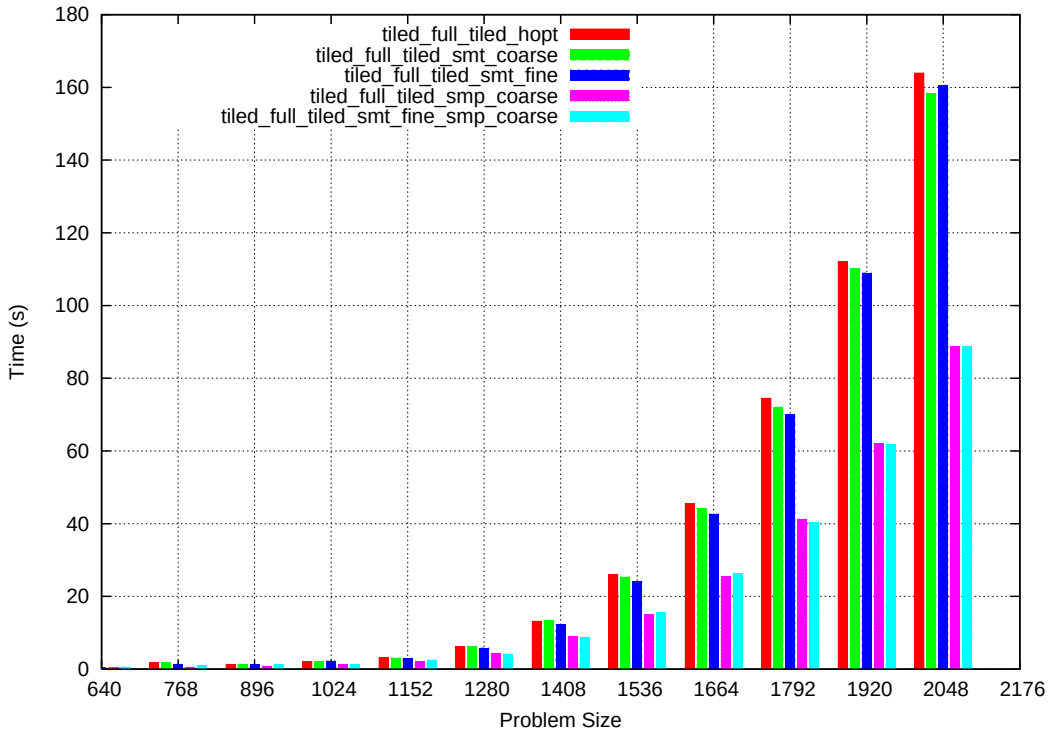


(α′) gcc -O0

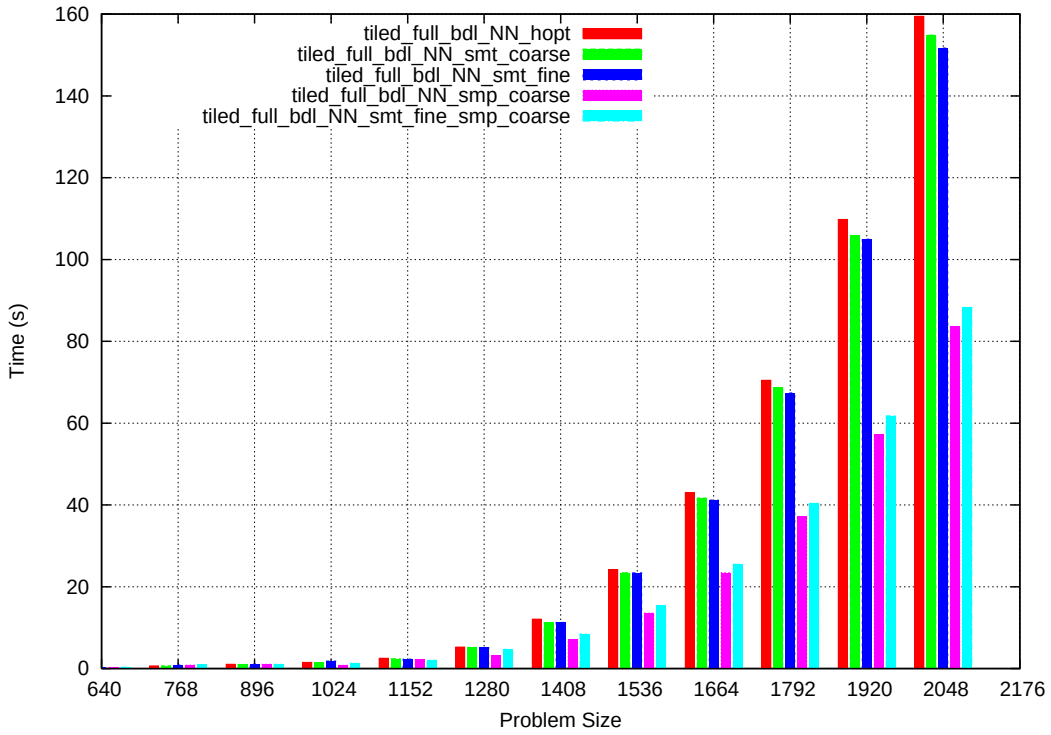


(β′) gcc -O2

Σχήμα 6.4: Χρόνος Εκτέλεσης στον Xeon (Σειριακές Εκδοχές)

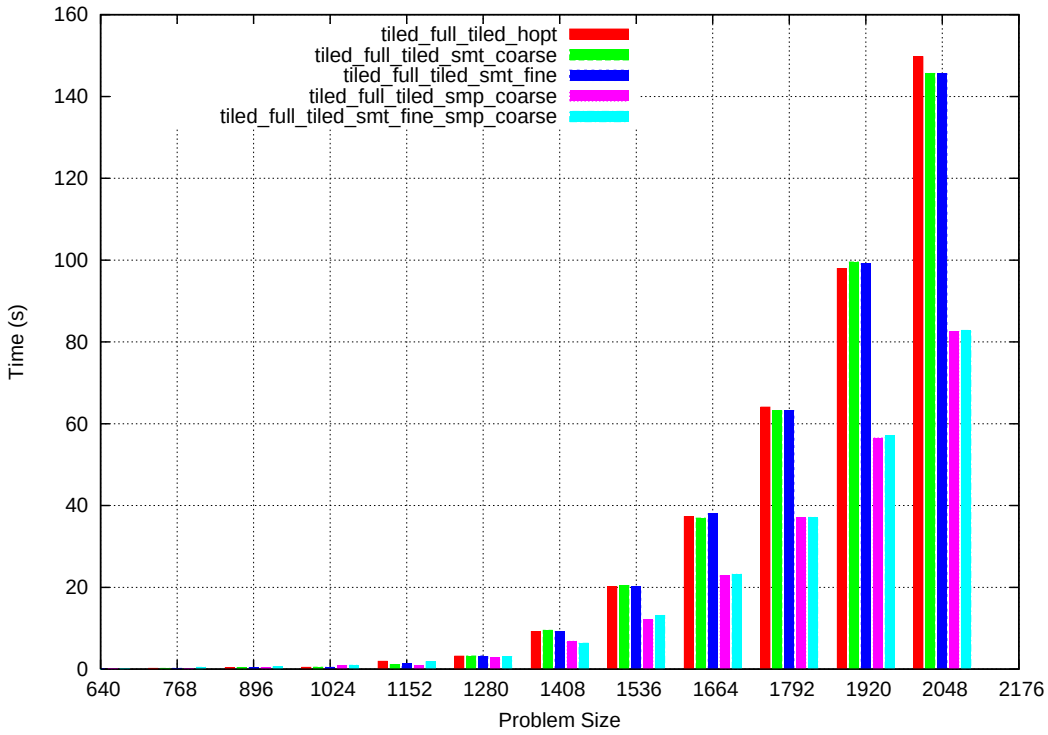


(α′) Tiling σε όλες τις διαστάσεις με γραμμικές διατάξεις

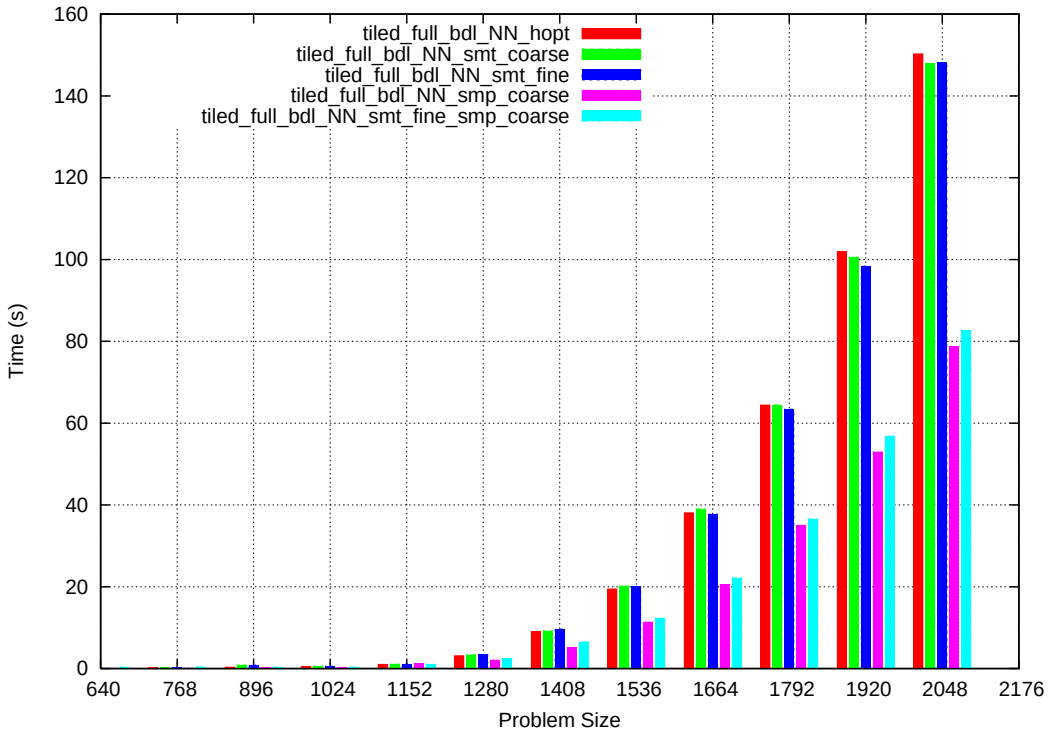


(β′) Tiling σε όλες τις διαστάσεις με διατάξεις ενοτήτων MBaLt

Σχήμα 6.5: Χρόνος Εκτέλεσης στον Xeon, -OO (Πολυνηματικές Εκδοχές)

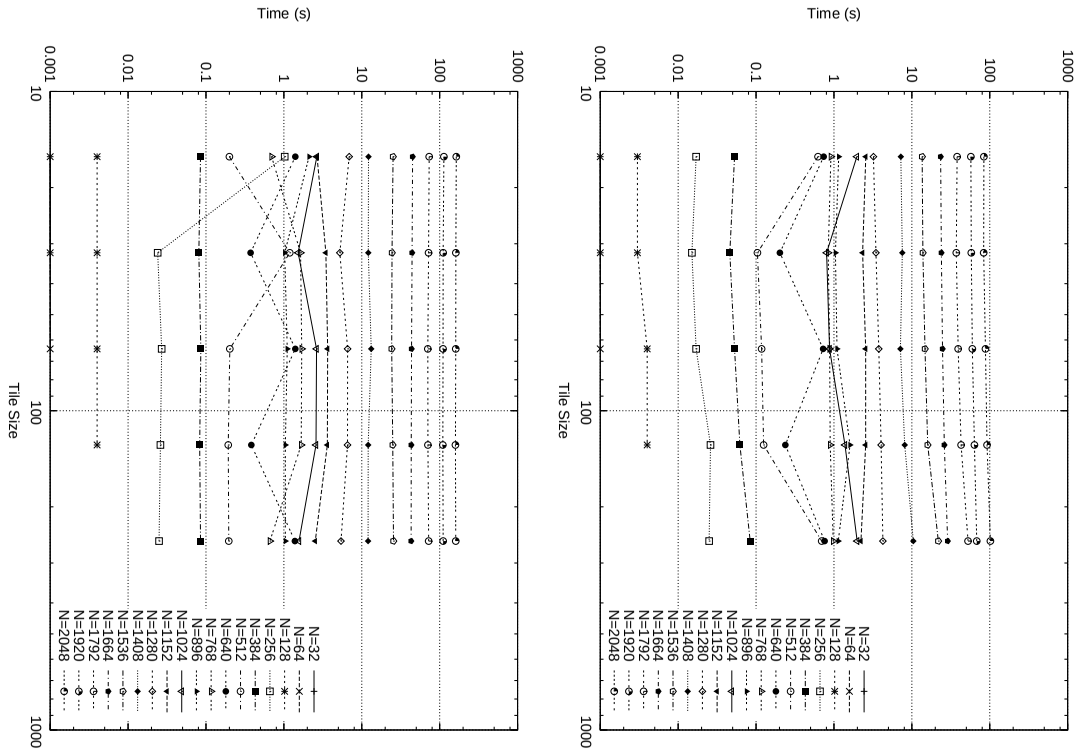


(α′) Tiling σε όλες τις διαστάσεις με γραμμικές διατάξεις



(β′) Tiling σε όλες τις διαστάσεις με διατάξεις ενοτήτων MBaLt

Σχήμα 6.6: Χρόνος Εκτέλεσης στον Xeon, -O2 (Πολυνηματικές Εκδοχές)



(α′) tiled_full_bdl_NN_hopt

(β′) tiled_full_bdl_NN_smp_coarse

(γ′) tiled_full_bdl_NN_smt_coarse

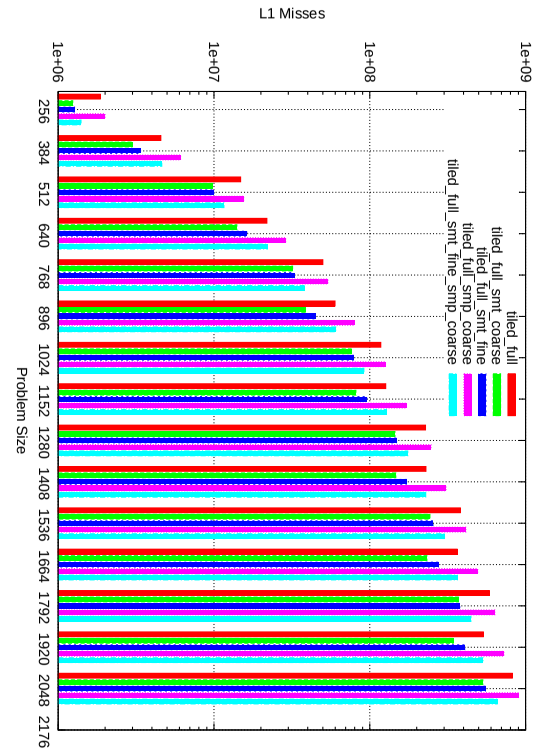
(δ′) tiled_full_bdl_NN_smt_fine

Σχήμα 6.7: Χρόνος Εκτέλεσης συναρτήσει Μεγέθους Tile στον Xeon, -O0

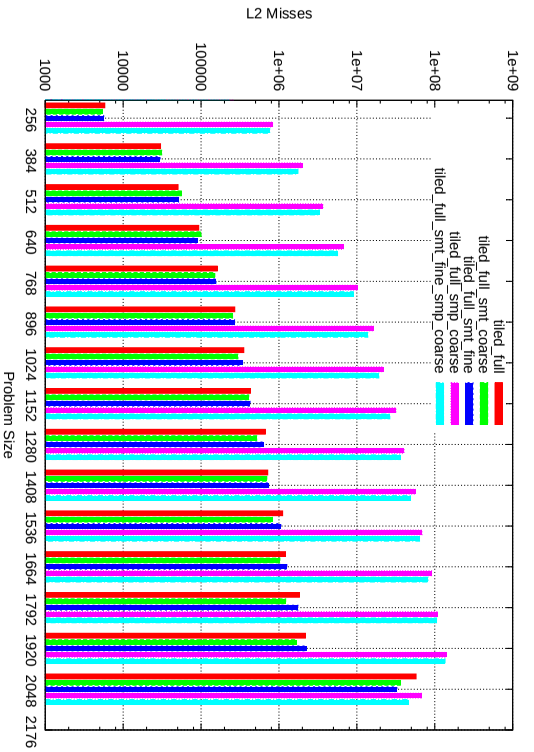
(γ΄) Κρυφή μνήμη L1 (δ) Κρυφή μνήμη L2

(α΄) Κρυφή μνήμη L1 (β΄) Κρυφή μνήμη L2

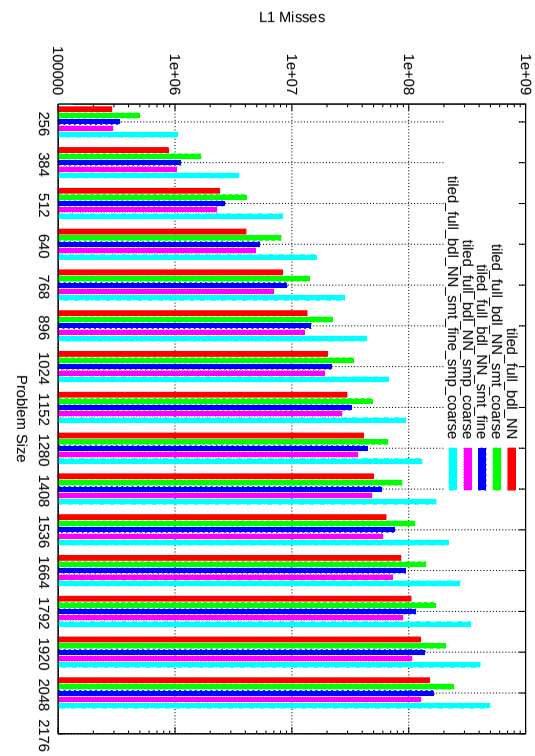
Σχήμα 6.8: Αστοσ Xeon, gcc -O0



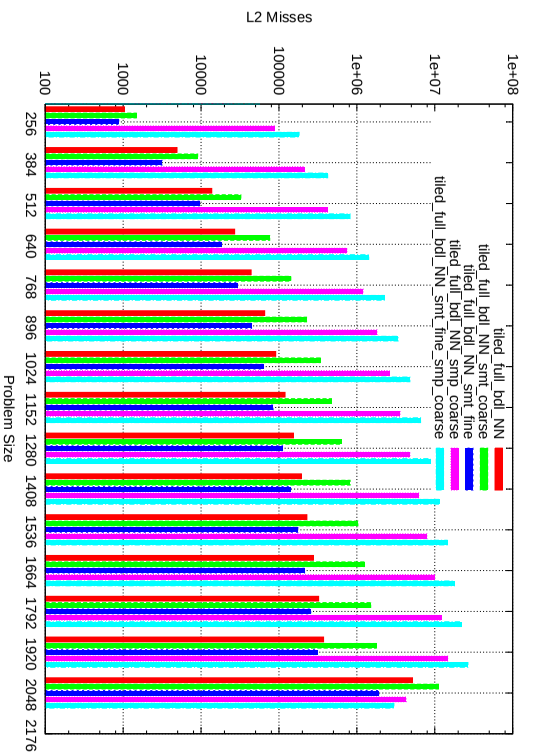
(α′) Κρυφή μνήμη L1



(β′) Κρυφή μνήμη L2

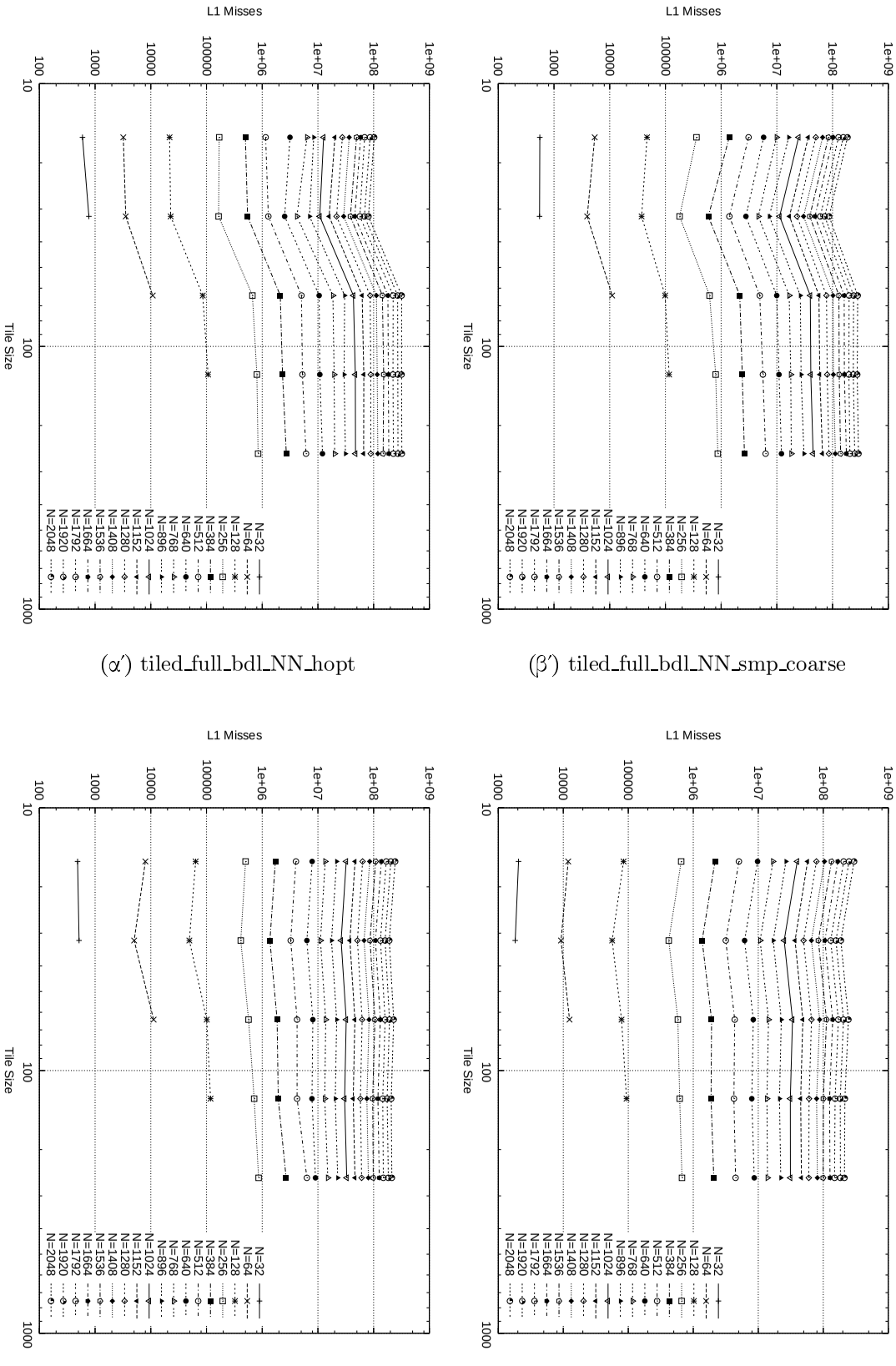


(γ′) Κρυφή μνήμη L1



(δ′) Κρυφή μνήμη L2

Σχήμα 6.9: Αστοχίες κρυφής μνήμης Xeon, gcc -O2



(α′) tiled_full_bdl_NN_hopt

(β′) tiled_full_bdl_NN_smp_coarse

(γ′) tiled_full_bdl_NN_smt_coarse

(δ′) tiled_full_bdl_NN_smt_fine

Σχήμα 6.10: Αστοχίες χρυφής μνήμης L1 συναρτήσει μεγέθους Tile στον Xeon, -O0

Παράρτημα Α΄

Συμπληρωματικά Θέματα

Α΄.1 Ένα Απλό Μοντέλο Εκτίμησης Κόστους Εκτέλεσης Φωλιασμένων Βρόχων

Α΄.1.1 Γενικά

Στο [14] χρησιμοποιείται ένα αποδοτικό, αλλά κι απλό, μοντέλο για την εκτίμηση του κόστους εκτέλεσης των βρόχων και με την εφαρμογή διαφόρων μετασχηματισμών, βρίσκεται η δομή των βρόχων που χρειάζεται το λιγότερο χρόνο εκτέλεσης.

Το μέτρο εκτίμησης της τοπικότητας είναι το πλήθος των ενοτήτων της κρυφής μνήμης που προσπελάνει ένα σύστημα φωλιασμένων βρόχων. Ελαχιστοποιώντας τον αριθμό των φορών που μια ενότητα πρέπει να μεταφερθεί από την κύρια μνήμη προς την κρυφή μνήμη, ελαχιστοποιούνται οι προσπελάσεις στην κύρια μνήμη κι άρα αυξάνεται η τοπικότητα.

Το μοντέλο κάνει τις ακόλουθες υποθέσεις:

- Επικεντρώνεται στην επαναχρησιμοποίηση που εμφανίζεται μεταξύ μικρού αριθμού επαναλήψεων του εσωτερικότερου βρόχου.
- Δεν υπάρχουν αποτυχίες κρυφής μνήμης λόγω χωρητικότητας σε μια επανάληψη του εσωτερικότερου βρόχου [10].
- Χωρίς βλάβη της γενικότητας θεωρείται ότι οι πίνακες αποθηκεύονται κατά γραμμές, όπως συμβαίνει με τη γλώσσα C/C++[1].

Α΄.1.2 Ομάδες Αναφορών

Αρχικά εντοπίζονται ποιες αναφορές ανήκουν στην ίδια *ομάδα αναφορών* (reference group). Δύο αναφορές ανήκουν στην ίδια ομάδα, αν παρουσιάζουν ομαδο-χρονική ή ομαδο-χωρική επαναχρησιμοποίηση.

[1] Το μοντέλο όπως προτείνεται από το [14] ακολουθεί τη σύμβαση της Fortran η οποία αποθηκεύει τους πίνακες κατά στήλες. Εδώ παρουσιάζεται ελαφρώς τροποποιημένο για να ακολουθεί το μοντέλο της C/C++.

Ορισμός Α΄.1 *Δύο αναφορές Ref_1 και Ref_2 ανήκουν στην ίδια ομάδα αναφορών ως προς τον βρόχο l εάν:*

1. *$\exists Ref_1 \vec{\delta} Ref_2$, και*
   - *(α΄) $\vec{\delta}$ είναι εξάρτηση ανεξάρτητη βρόχου, ή*
   - *(β΄) δ_l είναι μια μικρή σταθερά d ($|d| \leq 2$) και όλες οι άλλες συνιστώσες του $\vec{\delta}$ είναι μηδέν.*
2. *ή, Ref_1 και Ref_2 αναφέρονται στον ίδιο πίνακα και διαφέρουν το πολύ d' στην τελευταία συνιστώσα, όπου το d' είναι μικρότερο ή ίσο με την ενότητα κρυφής μνήμης, μετρώντας στοιχεία πίνακα. Κι όλες οι άλλες συνιστώσες ταυτίζονται.*

Η πρώτη συνθήκη αναφέρεται στην ομαδο-χρονική επαναχρησιμοποίηση, και η δεύτερη στην ομαδο-χωρική επαναχρησιμοποίηση. Μια αναφορά ανήκει μόνο σε μία ομάδα αναφορών, αφού ο παραπάνω ορισμός τοποθετεί μια αναφορά σε μια ομάδα, αν ικανοποιεί την πρώτη ή την δεύτερη συνθήκη με κάποια άλλη αναφορά μιας ομάδας.

Παράδειγμα Έστω ο ακόλουθος κώδικας:

```
for (k=2; k<=N-1; k++)
    for (j=2; j<=N-1; j++)
        for (i=2; i<=N-1; i++)
            A[i][j][k]=A[i][j+1][k+1]+B[i][j][k]
                +B[i][j+1][k]+B[i][j][k+1];
```

Εφαρμόζοντας τον ορισμό βρίσκουμε τις ακόλουθες ομάδες αναφορών:

βρόχος j	βρόχος i & k
$\{A[i][j][k]\}$	$\{A[i][j][k]\}$
$\{A[i][j+1][k+1]\}$	$\{A[i][j+1][k+1]\}$
$\{B[i][j][k], B[i][j+1][k], B[i][j][k+1]\}$	$\{B[i][j][k], B[i][j][k+1]\}$
	$\{B[i][j+1][k]\}$

Για παράδειγμα, οι αναφορές $B[i][j][k]$ και $B[i][j+1][k]$ διαφέρουν κατά 1 στη δεύτερη συνιστώσα, κι άρα βάση της συνθήκης 1(α) ανήκουν στην ίδια ομάδα ως προς το βρόχο j. Επίσης, η αναφορά $B[i][j][k+1]$ διαφέρει από την $B[i][j][k]$ κατά 1 στην τελευταία συνιστώσα και επομένως βάσει της συνθήκης 2 ανήκει στην ίδια ομάδα αναφορών. Οι υπόλοιπες ομάδες αναφορών παράγονται με παρόμοιο τρόπο.

Α΄.1.3 Κόστος Βρόχου

Αλγόριθμος Α΄.1 *Αλγόριθμος για την εύρεση κόστους εκτέλεσης ενός βρόχου, σε σύστημα φωλιασμένων βρόχων*
Είσοδος:

$$L = l_1, \ldots l_n \text{ φωλιασμένοι βρόχοι με όρια } lb_l, ub_l \text{ και βήμα } step_l$$
$$R = Ref_1, \ldots, Ref_m \text{ αναφορές που αντιπροσωπεύουν κάθε ομάδα}$$

$$
\begin{aligned}
& & & \textit{αναφορών} \\
& trip_l & = & (ubl - lbl + step_l)/step_l \\
& cls & = & \textit{το μέγεθος της ενότητας μνήμης σε στοιχεία} \\
& coef(f, i_l) & = & \textit{ο συντελεστής της μεταβλητής ελέγχου } i_l \textit{ στη συνιστώσα } f \\
& stride(f_j, i_l, l) & = & |step_l \times coef(f_j, i_l)|
\end{aligned}
$$

Έξοδος:

$$
\begin{aligned}
LoopCost(l) &= \textit{το πλήθος των ενοτήτων κρυφής μνήμης που προσπελάνονται} \\
&\quad \textit{, όταν ο } l \textit{ αποτελεί τον εσωτερικότερο βρόχο}
\end{aligned}
$$

$$
\mathbf{LoopCost}(l) = \sum_{k=1}^{m} (\mathbf{RefCost}(Ref_k(f_1(i_1, \ldots, i_n), \ldots, f_j(i_1, \ldots, i_n)), l)) \prod_{h \neq} trip_h
$$

$$
\mathbf{Refcost}(Ref_k, l) = \begin{cases} 1 & if((coef(f_l, i_l) = 0) \wedge \ldots \wedge \\ & (coef(f_j, i_l) = 0)) \\ \frac{trip_l}{\left(\frac{cls}{stride(f_1, i_l, l}\right)} & if((stride(f_j, i_l, l) < cls) \wedge (coef(f_2, i_l) = 0 \\ & \wedge \ldots \wedge \\ & (coef(f_j, i_l) = 0)) \\ trip_l & \textit{αλλιώς} \end{cases}
$$

Ο αλγόριθμος Α΄.1 εφαρμόζεται αφού βρεθούν οι διάφορες ομάδες αναφορών. Για την εύρεση του κόστους σε ενότητες κρυφής μνήμης μιας ομάδας αναφορών, επιλέγουμε τυχαία μια αναφορά με το μεγαλύτερο βάθος από κάθε ομάδα αναφορών. Κάθε βρόχος l με $trip$ επαναλήψεις, θεωρείται ως υποψήφιος για ως ο εσωτερικότερος βρόχος στο σύστημα φωλιασμένων βρόχων. Το cls είναι το μέγεθος της ενότητας κρυφής μνήμης και το $stride$ είναι το βήμα του βρόχου l πολλαπλασιασμένο με το συντελεστή της μεταβλητής ελέγχου.

Η $RefCost$ υπολογίζει την τοπικότητα για τον βρόχο l, δηλαδή το πλήθος ενοτήτων κρυφής μνήμης που ο l χρησιμοποιεί: 1 για αναφορές ανεξάρτητες βρόχου, $trip/(cls/stride)$ για συνεχόμενες αναφορές και $trip$ για μη συνεχόμενες αναφορές. Ακολούθως η $LoopCost$ υπολογίζει το συνολικό πλήθος ενοτήτων που προσπελάνονται από όλες τις αναφορές όταν ο βρόχος l αποτελεί τον εσωτερικότερο βρόχο.

Παράδειγμα Έστω ο ακόλουθος κώδικας ο οποίος εκτελεί τον πολλαπλασιασμό δυο πινάκων:

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

Εφαρμόζοντας τον αλγόριθμο Α΄.1 βρίσκουμε τα ακόλουθα κόστη:

Ομάδες	βρόχος i	βρόχος j	k
$C[i][j]$	$n \times n^2$	$\frac{n}{cls} \times n^2$	$1 \times n^2$
$A[i][k]$	$n \times n^2$	$1 \times n^2$	$\frac{n}{cls} \times n^2$
$B[k][j]$	$1 \times n^2$	$\frac{n}{cls} \times n^2$	$n \times n^2$
Συνολικό	$2n^3 + n^2$	$\frac{2}{cls}n^3 + n^2$	$\frac{1+cls}{cls}n^3 + n^2$

Για παράδειγμα, η RefCost βρίσκει ως προς τον βρόχο j την αυτο-χωρική επαναχρησιμοποίηση των αναφορών $C[i][j]$ και $B[k][j]$ και αναθέτει σε κάθε αναφορά κόστος ίσο με $fracncls$ ενότητες κρυφής μνήμης. Η $A[i][k]$ είναι ανεξάρτητη από τον βρόχο j και επομένως έχει κόστος 1. Η *LoopCost* λοιπόν για τον βρόχο i όταν αυτός είναι ο εσωτερικότερος, έχει συνολικό κόστος $\frac{2}{cls}n^3 + n^2$ για n^2 επαναλήψεις των βρόχων i και k.

Α΄.1.4 Εύρεση της Βέλτιστης Δομής

Στο [10] παρουσιάζεται αναλυτικά ένας ευριστικός αλγόριθμος, ο οποίος στηριζόμενος στην παραπάνω εκτίμηση κόστους εφαρμόζει τους κατάλληλους μετασχηματισμούς βρόχων που οδηγούν στην βέλτιστη δομή βρόχων. Εδώ, θα παρουσιάσουμε περιληπτικά τη λειτουργία του αλγορίθμου αυτού.

Μετάθεση Βρόχων

Λήμμα Α΄.1 *Εάν ο βρόχος l εκμεταλλεύεται περισσότερη επαναχρησιμοποίηση από τον βρόχο l' όταν και οι δυο λαμβάνονται υπόψιν ως εσωτερικότεροι βρόχοι, τότε είναι πολύ πιθανών ο l να εκμεταλλεύεται περισσότερη επαναχρησιμοποίηση από τον l' και σε οποιαδήποτε άλλη εξωτερικότερη θέση.*

Ορισμός Α΄.2 *Η διάταξη βρόχων $(i_1, i_2, \ldots, i_n)$ για την οποία ισχύει $LoopCost(i_{k-1}) \geq LoopCost(i_k)$ ονομάζεται* **διάταξη φθίνουσας μνήμης** *(memory order) .*

Για την εύρεση της βέλτιστης διάταξης βρόχων, ο αλγόριθμος που προτείνεται στο [14], για ένα σύνολο βρόχων $\{l_1, l_2, \ldots, l_n\}$ με διάταξη φθίνουσας μνήμης $L = (l_{s_1}, l_{s_2}, \ldots, l_{s_n})$, τοποθετεί ως εξωτερικότερο βρόχο τον ακριβότερο βρόχο l_{s_k} που βρίσκει ότι είναι έγκυρο (δηλαδή οι εξαρτήσεις παραμένουν λεξικογραφικά θετικές) να τοποθετηθεί εκεί. Ακολούθως βρίσκει τον επόμενο ακριβότερο βρόχο που επιτρέπεται να τοποθετηθεί σε βάθος 2, κ.ο.κ.

Θεώρημα Α΄.1 *Εάν υπάρχει έγκυρη μετάθεση όπου ο βρόχος i_{s_n} αποτελεί τον εσωτερικότερο βρόχο, τότε ο αλγόριθμος θα βρει αυτή την μετάθεση.*

Αναστροφή Βρόχου Η αναστροφή βρόχου εφαρμόζεται από την μετάθεση βρόχων, όταν κάποιος βρόχος λόγω εγκυρότητας δεν μπορεί να τοποθετηθεί σε μια επιθυμητή θέση.

Συγχώνευση Βρόχων

Ορισμός Α΄.3 *Δύο βρόχοι είναι* **συμβατοί** *εάν οι βρόχοι έκτελούν τον ίδιο αριθμό επαναλήψεων. Δύο συστήματα φωλιασμένων βρόχων είναι συμβατά σε βάθος d αν οι βρόχοι σε βάθος 1 ως d είναι συμβατοί, και είναι τελείως φωλιασμένοι μέχρι βάθος d.*

Δύο συστήματα φωλιασμένων βρόχων μπορούν να συγχωνευθούν αν είναι συμβατά. Για την εκτίμηση του κατά πόσο η συγχώνευση είναι επικερδής αρχικά υπολογίζονται τα $RefCost$ και *LoopCost* κάθε συστήματος ξεχωριστά, και ακολούθως υπολογίζονται για τις συγχωνευμένες εκδόσεις των συστημάτων. Εάν το *LoopCost* της φωλιασμένης έκδοσης είναι μικρότερο από το *LoopCost* των αρχικών εκδόσεων, τότε η εφαρμογή της συγχώνευσης αυξάνει την τοπικότητα κι άρα είναι επιθυμητή. Πρέπει βέβαια η συγχώνευση να ελέγχεται για την νομιμότητα της.

Σύνθεση Μετασχηματισμών Τελικός σκοπός είναι να επιτευχθεί διάταξη φθίνουσας μνήμης για όσο το δυνατό περισσότερες εντολές. Έτσι, ο ευριστικός αλγόριθμος αρχικά βελτιστοποιεί κάθε σύστημα φωλιασμένων βρόχων ξεχωριστά κι ακολούθως εφαρμόζει συγχώνευση αν αυτό είναι επικερδές. Για την βελτιστοποίηση κάθε επιμέρους συστήματος, ο αλγόριθμος εφαρμόζει μετάθεση για να πετύχει διάταξη φθίνουσας μνήμης. Αν δεν επιτυγχάνεται η βέλτιστη διάταξη λόγω παραβίασης της νομιμότητας, τότε εφαρμόζεται συγχώνευση η οποία μπορεί να νομιμοποιήσει την βέλτιστη διάταξη. Αν η συγχώνευση δεν επιτυγχάνει την νομιμοποίηση, τότε μπορεί να εφαρμοστεί η διάσπαση βρόχων.

Παράρτημα Β΄

Πηγαίος Κώδικας

Β΄.1 Πηγαίος Κώδικας Μετροπρογράμματος Cholesky

```
1  /* CHOLESKY BENCHMARK SUIT
2   * Final Version
3   * Copyright (R) by Haris M. Volos in 2005
4   *
5   * Different methods used to implement and optimize the benchmark
6   * See main for the different optimization levels
7   *
8   * THIS IS USED TO OBTAIN MEASUREMENTS
9   */
10
11 #include <stdio.h>
12 #include <math.h>
13 #include <sys/time.h>
14 #include "bdl.c"
15
16 #define MAX(a, b) ((a>b)?a:b)
17 #define MIN(a, b) ((a<b)?a:b)
18
19 #define eps 0.001
20
21 /* ==========================================================
22  *
23  * The implementations-optimizations of the Cholesky benchmark follow
24  *
25  * ==========================================================
26  */
27
28
29 /* Function:    cholesky_raw
30  * The unoptimized version of the benchmark
31  */
32 void cholesky_raw(double **A, int N)
33 {
34    int i, j, k;
35
36    for (k=0; k<N;k++)
37    {
38       A[k][k]=sqrt(fabs(A[k][k]));
39       for (i=k+1;i<N;i++)
40          A[i][k]=A[i][k]/A[k][k];
41
42       for (j=k+1;j<N;j++)
43          for (i=j;i<N;i++)
44             A[i][j]=A[i][j]-A[i][k]*A[j][k];
45    }
46 }
47
48 /* Function:    cholesky_IJK
49  * Loops are in the order (i, j, k)
50  */
```

```
51  void cholesky_IJK(double **A, int N)
52  {
53     int i, j, k;
54     for (i=1;i<N;i++)
55     {
56        A[i-1][i-1] = sqrt(fabs(A[i-1][i-1]));
57        for (j=1;j<=i;j++)
58        {
59           A[i][j-1]=A[i][j-1]/A[j-1][j-1];
60           for (k=0; k<j;k++)
61              A[i][j]=A[i][j]-A[i][k]*A[j][k];
62        }
63     }
64     A[N-1][N-1]=sqrt(fabs(A[N-1][N-1]));
65  }
66
67  /* Function:    cholesky_linear_layout_column_major
68   * Column major linear layout is used for the storage of the array
69   */
70  void cholesky_column_major(double **A, int N)
71  {
72     int i, j, k;
73
74     for (k=0; k<N;k++)
75     {
76        A[k][k]=sqrt(fabs(A[k][k]));
77        for (j=k+1;j<N;j++)
78           A[k][j]/=A[k][k];
79
80        for (j=k+1;j<N;j++)
81           for (i=j;i<N;i++)
82              A[j][i]-=A[k][i]*A[k][j];
83     }
84  }
85
86  /* Function:    cholesky_tiled_KJ
87   * Cholesky is tiled along loops K, J
88   * Column major linear layout is used for the storage of the array
89   */
90  void cholesky_tiled_KJ(double **A, int N, int tile_size)
91  {
92     int i, j, k;
93     int ii, jj, kk;
94     int Bii=tile_size, Bjj=tile_size, Bkk=tile_size;
95     int jjlow, kkup, jlow, jup;
96
97
98     for (kk=0; kk<N;kk+=Bkk)
99     {
100       jjlow = (int) floor((double)kk/Bjj)*Bjj;
101       for (jj = jjlow;jj<N;jj+=Bjj)
102       {
103          kkup = MIN(kk+Bkk,N);
104          for (k=kk; k<kkup;k++)
105          {
106             if (jj<=k && k<jj+Bjj)
107                A[k][k]=sqrt(fabs(A[k][k]));
108
109             if (jj==jjlow)
110             {
111                for (j=k+1;j<N;j++)
112                   A[k][j]/=A[k][k];
113             }
114
115             jlow = MAX(jj,k+1);
116             jup = MIN(jj+Bjj,N);
117             for (j=jlow;j<jup;j++)
118                for (i=j;i<N;i++)
119                   A[j][i]-=A[k][i]*A[k][j];
120          }
121       }
122    }
123 }
124
125 /* Function:    cholesky_tiled_KJ_hopt
126  * Cholesky is tiled along loops K, J
127  * Column major linear layout is used for the storage of the array
128  * Hand optimization includes loop unrolling and scalar replacement
129  */
130 void cholesky_tiled_KJ_hopt(double **A, int N, int tile_size)
131 {
132    int i, j, k;
133    int ii, jj, kk;
134    int Bii=tile_size, Bjj=tile_size, Bkk=tile_size;
135    int jjlow, kkup, jlow, jup;
```

```
136     int iup_unrolled;
137     register double Akj;
138
139     for (kk=0; kk<N;kk+=Bkk)
140     {
141        jjlow = (int) floor((double)kk/Bjj)*Bjj;
142        for (jj = jjlow;jj<N;jj+=Bjj)
143        {
144           kkup = MIN(kk+Bkk,N);
145           for (k=kk; k<kkup;k++)
146           {
147              if (jj<=k && k<jj+Bjj)
148                 A[k][k]=sqrt(fabs(A[k][k]));
149
150              if (jj==jjlow)
151              {
152                 for (j=k+1;j<N;j++)
153                    A[k][j]/=A[k][k];
154              }
155
156              jlow = MAX(jj,k+1);
157              jup = MIN(jj+Bjj,N);
158              for (j=jlow;j<jup;j++)
159              {
160                    Akj = A[k][j];
161
162                    iup_unrolled = j + (N-j)/4*4;
163
164                    for (i=j;i<iup_unrolled;i+=4)
165                    {
166                       A[j][i]-=A[k][i]*Akj;
167                       A[j][i+1]-=A[k][i+1]*Akj;
168                       A[j][i+2]-=A[k][i+2]*Akj;
169                       A[j][i+3]-=A[k][i+3]*Akj;
170                    }
171
172                    for (;i<N;i++)
173                       A[j][i]-=A[k][i]*Akj;
174
175              }
176           }
177        }
178     }
179  }
180
181  /* Function:    cholesky_tiled_full
182   * Cholesky is tiled along all loops
183   * Column major linear layout is used for the storage of the array
184   */
185  void cholesky_tiled_full(double **A, int N, int tile_size)
186  {
187     int i, j, k;
188     int ii, jj, kk;
189     int Bii=tile_size, Bjj=tile_size, Bkk=tile_size;
190     int kkup, jlow, jup, ilow, iup;
191     int j2low, j2up;
192
193     for (kk=0; kk<N;kk+=Bkk)
194     {
195        kkup = MIN(kk+Bkk,N);
196        for (jj=(int) floor((double)kk/Bjj)*Bjj;jj<N;jj+=Bjj)
197           for (ii=(int) floor((double)jj/Bii)*Bii;ii<N;ii+=Bii)
198              for (k=kk; k<kkup;k++)
199              {
200                 if (jj<=k && k<jj+Bjj  && ii<=k && k<ii+Bii)
201                    A[k][k]=sqrt(fabs(A[k][k]));
202
203                 if (jj==(int) floor((double)kk/Bjj)*Bjj)
204                 {
205                    j2low = MAX(ii, k + 1);
206                    j2up = MIN(ii + Bjj, N);
207                    for (j=j2low;j<j2up;j++)
208                       A[k][j]/=A[k][k];
209                 }
210
211                 jlow = MAX(jj, k + 1);
212                 jup = MIN(jj + Bjj, N);
213                 iup = MIN(ii + Bii, N);
214                 for (j=jlow;j<jup;j++)
215                 {
216                    ilow = MAX(ii, j);
217                    for (i=ilow;i<iup;i++)
218                       A[j][i]-=A[k][i]*A[k][j];
219                 }
220              }
221     }
222  }
```

```
223
224 /* Function:    cholesky_tiled_full_hopt
225  * Cholesky is tiled along all loops
226  * Column major linear layout is used for the storage of the array
227  * Hand optimization includes loop unrolling and scalar replacement
228  */
229 void cholesky_tiled_full_hopt(double **A, int N, int tile_size)
230 {
231    int i, j, k;
232    int ii, jj, kk;
233    int Bii=tile_size, Bjj=tile_size, Bkk=tile_size;
234    int kkup, jlow, jup, ilow, iup;
235    int j2low, j2up;
236    int iup_unrolled;
237    register double Akj;
238
239    for (kk=0; kk<N;kk+=Bkk)
240    {
241       kkup = MIN(kk+Bkk,N);
242       for (jj=(int) floor((double)kk/Bjj)*Bjj;jj<N;jj+=Bjj)
243          for (ii=(int) floor((double)jj/Bii)*Bii;ii<N;ii+=Bii)
244             for (k=kk; k<kkup;k++)
245             {
246                if (jj<=k && k<jj+Bjj  && ii<=k && k<ii+Bii)
247                   A[k][k]=sqrt(fabs(A[k][k]));
248
249                if (jj==(int) floor((double)kk/Bjj)*Bjj)
250                {
251                   j2low = MAX(ii, k + 1);
252                   j2up = MIN(ii + Bjj, N);
253                   for (j=j2low;j<j2up;j++)
254                      A[k][j]/=A[k][k];
255                }
256
257                jlow = MAX(jj, k + 1);
258                jup = MIN(jj + Bjj, N);
259                iup = MIN(ii + Bii, N);
260
261                for (j=jlow;j<jup;j++)
262                {
263                   Akj = A[k][j];
264
265                   ilow = MAX(ii, j);
266                   iup_unrolled = ilow + (iup-ilow)/4*4;
267
268                   for (i=ilow;i<iup_unrolled;i+=4)
269                   {
270                      A[j][i]-=A[k][i]*Akj;
271                      A[j][i+1]-=A[k][i+1]*Akj;
272                      A[j][i+2]-=A[k][i+2]*Akj;
273                      A[j][i+3]-=A[k][i+3]*Akj;
274                   }
275
276                   for (;i<iup;i++)
277                      A[j][i]-=A[k][i]*Akj;
278                }
279             }
280    }
281 }
282
283 /* Function:    cholesky_tiled_full_bdl_NN
284  * Cholesky is tiled along all loops, and Block Data Layout NN is used
285  */
286 void cholesky_tiled_full_bdl_NN(double *A, int N, int Nmask, int step)
287 {
288    int i, j, k;
289    int iR, jC, kC;
290    int ii, jj, kk, jjC, kkC;
291    int jfC, kfC;
292    int kk_plus_jjkkincrement;
293
294    int transformation_type = NN, transformation_type2 = Nn;
295
296    int logNstep, logstep;
297
298    int imask, jkmask;
299    int ibound, jkbound, itilebound, jtilebound, ktilebound;
300    int iiincrement, jjkkincrement;
301    int iincrement, jkincrement;
302    int iup, ilow, jup, jlow, kup, klow, ilowDiv;
303    int jjlow, iilow;
304    int step_1, kplus1, kplus1C, kplus1fC;
305
306    logNstep = lg(Nmask/step);
307    logstep = lg(step);
308
```

```
309     ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
310     jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
311
312     imask = create_innermask(step, transformation_type2, 0) + 1;
313     jkmask = create_innermask(step, transformation_type2, 1) + 1;
314
315     iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
316     jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
317
318
319     iincrement = create_increment(step, transformation_type2, COL);
320     jkincrement = create_increment(step, transformation_type2, ROW);
321
322     step_1 = step - 1;
323
324     for (kk = 0; kk < jkbound; kk += jjkkincrement)
325     {
326        ktilebound = (kk | jkmask);
327        kup = (jkbound < ktilebound ? jkbound : ktilebound);
328        kk_plus_jjkkincrement = kk+jjkkincrement;
329
330        kkC = kk >> logNstep;
331
332        jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
333        for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
334        {
335           jtilebound = (jj | jkmask);
336           jup = (jkbound < jtilebound ? jkbound : jtilebound);
337
338           jjC = jj >> logNstep;
339
340           iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep
                 ;
341           for (ii=iilow;ii<ibound;ii+=iiincrement)
342           {
343              itilebound = (ii | imask);
344              iup = (ibound < itilebound ? ibound : itilebound);
345
346              for (k=kk; k<kup;k+=jkincrement)
347              {
348                 kC = ((k >> logstep) & step_1);
349                 kfC = kkC | kC;
350
351                 kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                       jkincrement);
352                 kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                       iincrement);
353                 ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
354
355
356                 if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
357                    A[kfC|k]=sqrt(fabs(A[kfC|k]));
358
359                 if (jj==jjlow)
360                 {
361                    for (i=ilowDiv;i<iup;i+=iincrement)
362                       A[i|k]/=A[kfC|k];
363                 }
364
365                 jlow = (jj > kplus1 ? jj : kplus1);
366                 for (j=jlow;j<jup;j+=jkincrement)
367                 {
368                    jC = ((j >> logstep) & step_1);
369                    jfC = jjC | jC;
370                    ilow = (ii > jfC ? ii : jfC);
371                    for (i=ilow;i<iup;i+=iincrement)
372                       A[i|j]-=A[i|k]*A[jfC|k];
373                 }
374              }
375           }
376        }
377     }
378  }
379
380  /* Function:    cholesky_tiled_full_bdl_NN_hopt
381   * Cholesky is tiled along all loops, Block Data Layout NN is used
382   * Hand optimization includes loop unrolling and scalar replacement
383   */
384  void cholesky_tiled_full_bdl_NN_hopt(double *A, int N, int Nmask, int step)
385  {
386     int i, j, k;
387     int iR, jC, kC;
388     int ii, jj, kk, jjC, kkC;
389     int jfC, kfC;
390     int kk_plus_jjkkincrement;
391
392     int transformation_type = NN, transformation_type2 = Nn;
```

```
393
394     int logNstep, logstep;
395
396     int imask, jkmask;
397     register double AjfC_k;
398     register int ibound, jkbound, itilebound, jtilebound, ktilebound;
399     register int iiincrement, jjkkincrement;
400     register int iincrement, jkincrement;
401     int iup, ilow, jup, jlow, kup, klow, ilowDiv;
402     int jjlow, iilow;
403     register int step_1, kplus1, kplus1C, kplus1fC;
404
405     int iup_unrolled, iincrement_unroll;
406
407     logNstep = lg(Nmask/step);
408     logstep = lg(step);
409
410     ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
411     jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
412
413     imask = create_innermask(step, transformation_type2, 0) + 1;
414     jkmask = create_innermask(step, transformation_type2, 1) + 1;
415
416     iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
417     jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
418
419
420     iincrement = create_increment(step, transformation_type2, COL);
421     jkincrement = create_increment(step, transformation_type2, ROW);
422
423     /*
424      * iincrement_unroll in our case is 4 that is why we replace it with the constant
            4
425      * iincrement_unroll = iincrement * 4;
426      */
427
428     step_1 = step - 1;
429
430     for (kk = 0; kk < jkbound; kk += jjkkincrement)
431     {
432        ktilebound = (kk | jkmask);
433        kup = (jkbound < ktilebound ? jkbound : ktilebound);
434        kk_plus_jjkkincrement = kk+jjkkincrement;
435
436        kkC = kk >> logNstep;
437
438        jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
439        for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
440        {
441           jtilebound = (jj | jkmask);
442           jup = (jkbound < jtilebound ? jkbound : jtilebound);
443
444           jjC = jj >> logNstep;
445
446           iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep
                 ;
447           for (ii=iilow;ii<ibound;ii+=iiincrement)
448           {
449              itilebound = (ii | imask);
450              iup = (ibound < itilebound ? ibound : itilebound);
451
452              for (k=kk; k<kup;k+=jkincrement)
453              {
454                 kC = ((k >> logstep) & step_1);
455                 kfC = kkC | kC;
456
457                 kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                       jkincrement);
458                 kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                       iincrement);
459                 ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
460
461
462                 if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
463                    A[kfC|k]=sqrt(fabs(A[kfC|k]));
464
465                 if (jj==jjlow)
466                 {
467                    for (i=ilowDiv;i<iup;i+=iincrement)
468                       A[i|k]/=A[kfC|k];
469                 }
470
471                 jlow = (jj > kplus1 ? jj : kplus1);
472                 for (j=jlow;j<jup;j+=jkincrement)
473                 {
474                    jC = ((j >> logstep) & step_1);
475                    jfC = jjC | jC;
```

```
476                     ilow = (ii > jfC ? ii : jfC);
477                     AjfC_k = A[jfC|k];
478
479                     iup_unrolled = ilow + (iup-ilow)/4*4;
480                     for (i=ilow;i<iup_unrolled;i+=4)
481                     {
482                        A[i|j]-=A[i|k]*AjfC_k;
483                        A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
484                        A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
485                        A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
486                     }
487
488                     for (;i<iup;i++)
489                        A[i|j]-=A[i|k]*AjfC_k;
490                  }
491               }
492            }
493         }
494      }
495   }
496
497   /* Function:    cholesky_full_tiled_bdl_ZZ
498    * Cholesky is tiled along all loops, Block Data Layout ZZ is used
499    */
500   void cholesky_tiled_full_bdl_ZZ(double *A, int N, int Nmask, int step)
501   {
502      struct timeval start,finish;
503
504      int i, j, k;
505      int iR, jC, kC;
506      int ii, jj, kk, jjC, kkC;
507      int jfC, kfC;
508      register int temp1;
509      int temp2, temp3;
510
511      int transformation_type = ZZ, transformation_type2 = Zz;
512
513      int logNstep, logstep;
514
515      int imask, jkmask;
516      int ibound, jkbound, itilebound, jtilebound, ktilebound;
517      int iiincrement, jjkkincrement;
518      int iincrement, jkincrement;
519      int iup, ilow, jup, jlow, kup, kkup;
520      int jjup;
521      int step_1, kplus1;
522
523      logNstep = lg(Nmask/step);
524      logstep = lg(step);
525
526      ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
527      jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
528
529      imask = create_innermask(step, transformation_type2, 0) + 1;
530      jkmask = create_innermask(step, transformation_type2, 1) + 1;
531
532      iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
533      jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
534
535
536      iincrement = create_increment(step, transformation_type2, COL);
537      jkincrement = create_increment(step, transformation_type2, ROW);
538
539      step_1 = step - 1;
540
541      gettimeofday(&start,(struct timezone*) NULL);
542
543      for (ii = 0; ii < ibound; ii += iiincrement)
544      {
545         itilebound = (ii | imask);
546         iup = (ibound < itilebound ? ibound : itilebound);
547         ilow = (iincrement > ii ? iincrement : ii);
548
549         jjup = (ii >> logNstep);
550         for (jj = 0; jj <= jjup; jj += jjkkincrement )
551         {
552            jtilebound = (jj | jkmask);
553            jlow = (jkincrement > jj ? jkincrement : jj);
554
555            jjC = jj << logNstep;
556
557            for (kk = 0; kk <= jj; kk += jjkkincrement)
558            {
559               ktilebound = (kk | jkmask);
560
561               kkC = kk << logNstep;
562
```

```
563                 temp1 = kk+jjkkincrement;
564                 for (i = ilow; i < iup; i += iincrement)
565                 {
566                    iR = (((i & (imask-1)) >> logstep) | (ii >> logNstep));
567                    jup = (iR  + 1 < jtilebound ? iR + 1 : jtilebound);
568
569                    for (j = jlow; j < jup; j += jkincrement)
570                    {
571                       jC = ((j & step_1) << logstep);
572                       jfC = jjC | jC;
573
574                       kup = (j < ktilebound ? j : ktilebound);
575
576                       for (k = kk; k < kup; k++)
577                       {
578
579                          kplus1 = ((k+1==ktilebound)? temp1:k+1);
580
581                          if (j==kplus1)
582                          {
583                             kC = ((k & step_1) << logstep);
584                             kfC = kkC | kC;
585                             if (kplus1 ==iR)
586                                A[kfC|k]=sqrt(fabs(A[kfC|k]));
587
588                             A[i|k]=A[i|k]/A[kfC|k];
589                          }
590
591                          A[i|j]=A[i|j]-A[i|k]*A[jfC|k];
592
593                       }
594                    }
595                 }
596              }
597           }
598        }
599        A[N*N-1]=sqrt(fabs(A[N*N-1]));
600     }
601
602     main(int argc, char **argv)
603     {
604        int i, j, k;
605        int optimization_level=0, tile_size;
606        int N, Nincr, Nsize;
607        struct timeval start,finish;
608
609        double **B_2D;
610        double *B;
611        double temp;
612
613        for (i=0;i<argc;i++)
614           if (strcmp(argv[i],"-O")==0)
615              optimization_level=atoi(argv[++i]);
616
617        printf("Running Cholesky with optimization level %d...\n", optimization_level);
618        printf("N\tNsize\tTile Size\tDuration\n");
619
620        srand(time(NULL));
621
622        Nincr=16;
623
624        for (N=16; N<=2048; N+=Nincr) {
625          if (N<128) Nincr=N;
626           else Nincr=128;
627           for (tile_size=16; tile_size<=N; tile_size*=2) {
628              Nsize=2;
629              while (Nsize<N) Nsize*=2;
630
631
632              if (optimization_level >= 7) {
633                 if (B=(double *) malloc(Nsize*Nsize*sizeof(double)))
634                    for (i=0;i<Nsize*Nsize;i++)
635                          B[i]=rand() % 100 +1;
636                 else {
637                    printf("Memory allocation failed!\n");
638                    exit(1);
639                 }
640              }
641              else {
642                 if (B_2D=(double **) malloc(N*sizeof(double *) + N*N*sizeof(double))) {
643                   B_2D[0]=(double *) (&B_2D[N]);
644
645                   for (i=1;i<N;i++)
646                    B_2D[i]=&B_2D[0][N*i];
647
648                   for (i=0;i<N;i++)
649                      for (j=0;j<N;j++)
650                         B_2D[i][j] = rand() % 100 + 1;
```

```
651                 }
652                 else {
653                     printf("Memory allocation failed!\n");
654                     exit(1);
655                 }
656             }
657 
658             gettimeofday(&start,(struct timezone*) NULL);
659 
660             switch (optimization_level) {
661                 case 0:
662                     cholesky_raw(B_2D, N);
663                     break;
664                 case 1:
665                     cholesky_IJK(B_2D, N);
666                     break;
667                 case 2:
668                     cholesky_column_major(B_2D, N);
669                     break;
670                 case 3:
671                     cholesky_tiled_KJ(B_2D, N, tile_size);
672                     break;
673                 case 4:
674                     cholesky_tiled_KJ_hopt(B_2D, N, tile_size);
675                     break;
676                 case 5:
677                     cholesky_tiled_full(B_2D, N, tile_size);
678                     break;
679                 case 6:
680                     cholesky_tiled_full_hopt(B_2D, N, tile_size);
681                     break;
682                 case 7:
683                     cholesky_tiled_full_bdl_NN(B, N, Nsize, tile_size);
684                     break;
685                 case 8:
686                     cholesky_tiled_full_bdl_NN_hopt(B, N, Nsize, tile_size);
687                     break;
688                 case 9:
689                     cholesky_tiled_full_bdl_ZZ(B, N, Nsize, tile_size);
690                     break;
691                 default:
692                     cholesky_raw(B_2D, N);
693             }
694             gettimeofday(&finish,(struct timezone*) NULL);
695 
696             /* Show report */
697             printf("%d\t%d\t%d\t",N , Nsize, tile_size);
698             printf("%8.3lf\n", finish.tv_sec-start.tv_sec
699                     + (double)((finish.tv_usec-start.tv_usec>=0)?(finish.tv_usec-start.
                            tv_usec):
700                     (1000000+finish.tv_usec-start.tv_usec))/1000000);
701 
702             if (optimization_level >= 7)
703                 free(B);
704             else
705                 free(B_2D);
706         }
707     }
708 }
```

Β΄.2 Πηγαίος Κώδικας Μετροπρογράμματος Cholesky - Πολυνηματικές Εκδοχές

Λόγου χώρου παρουσιάζονται μόνο οι πολυνηματωμένες εκδοχές με μετασχηματισμό tiling και διατάξεις δεδομένων με δεικτοδότηση MBaLt.

```
1  /* CHOLESKY BENCHMARK SUIT FOR HYPER-THREADING ENABLED PROCESSORS (Intel Xeon,
       Pentium 4)
2   * Final Version
3   * Copyright (R) by Haris M. Volos in 2005
4   *
5   * Different methods used to parallelize (fine-grain, coarse-grain)
6   * and optimize (tiling, block data layouts, unrolling) the benchmark
7   * See main for the different optimization levels
8   *
9   * THIS IS USED TO OBTAIN MEASUREMENTS
10  */
11 
12 #include <pthread.h>
13 #include <sched.h>
```

```
14 #include <stdio.h>
15 #include <math.h>
16 #include <sys/time.h>
17 #include "bdl.c"
18 #include "aff.h"
19 #include "spin.h"
20 #include "prfcnt.h"
21
22 #define MAX(a, b) ((a>b)?a:b)
23 #define MIN(a, b) ((a<b)?a:b)
24
25 #define eps 0.001
26
27 typedef struct mm_thread_args {
28    double **A;
29    double *A_1D;
30    int N, Nmask;
31    int tile_size;
32    int tid;
33    int tgid;
34    int n_threads;
35    int ng_threads;
36    int n_groups;
37    int pk;
38    cpu_t sib;
39    spin_barrier_t *spin_barrier_fine;
40    spin_barrier_t *spin_barrier_coarse;
41 } mm_thread_args_t;
42
43 static spin_barrier_t spin_barrier_A2;
44 static spin_barrier_t spin_barrier_B2;
45 static spin_barrier_t spin_barrier_A4;
46
47 struct cpuinfo cpus;
48 int prf_flag;
49 cpu_t prf_cpu;
50
51 void myprfcnt_start(cpu_t cpu_id, int sib_id)
52 {
53    if (prf_flag == 1 && cpu_id == prf_cpu)
54    {
55       if (sib_id == 0)
56          prfcnt_init(prf_cpu, PRFCNT_FL_T0);
57       else if (sib_id == 1)
58          prfcnt_init(prf_cpu, PRFCNT_FL_T1);
59       prfcnt_start();
60    }
61 }
62
63 void myprfcnt_stop(int cpu_id)
64 {
65    if (prf_flag == 1 && cpu_id == prf_cpu)
66    {
67       prfcnt_pause();
68       prfcnt_report();
69       prfcnt_shut();
70    }
71 }
72
73 /* Function: thread_tiled_full_bdl_NN_fine_grain
74  * Block Data Layout NN is used
75  * Tiling is applied along all loops
76  * Hand optimization includes loop unrolling and scalar replacement
77  * All threads work on the same tile simultaneously and the data of the tile are
       distributed
78  * between them in a fine grain fashion
79  */
80 void thread_tiled_full_bdl_NN_fine_grain(mm_thread_args_t *args)
81 {
82    mm_thread_args_t * thread_args = (mm_thread_args_t *) args;
83
84    int i, j, k;
85    int iR, jC, kC;
86    int ii, jj, kk, jjC, kkC;
87    int jfC, kfC;
88    int kk_plus_jjkkincrement;
89
90    int transformation_type = NN, transformation_type2 = Nn;
91
92    int logNstep, logstep;
93
94    int imask, jkmask;
95    int ibound, jkbound, itilebound, jtilebound, ktilebound;
96    int iiincrement, jjkkincrement;
97    int iincrement, jkincrement;
98    int jkincrement_2;
```

```
 99      int iup, ilow, jup, jlow, kup, klow, ilowDiv, iup_div;
100      int jjlow, iilow;
101      int step, step_1, kplus1, kplus1C, kplus1fC;
102
103      int j_tonus;
104
105      int N, Nmask;
106      double *A;
107      register double AjfC_k;
108      int lsense;
109
110      spin_barrier_init_lsense(&lsense);
111
112      // Bind the thread to the given physical package and logical processor unit
113      if(thread_args->sib == -1) {
114         run_on_cpus( ( 1 << cpus.pk[thread_args->pk].sib[0] ) | ( 1 << cpus.pk[
             thread_args->pk].sib[1]) );
115      } else
116         run_on_cpu( cpus.pk[thread_args->pk].sib[thread_args->sib] );
117
118      // Some assignments (aliases) and initializations
119      A = thread_args->A_1D;
120      N = thread_args->N;
121      Nmask = thread_args->Nmask;
122      step = thread_args->tile_size;
123
124      logNstep = lg(Nmask/step);
125      logstep = lg(step);
126
127      ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
128      jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
129
130      imask = create_innermask(step, transformation_type2, 0) + 1;
131      jkmask = create_innermask(step, transformation_type2, 1) + 1;
132
133      iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
134      jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
135
136
137      iincrement = create_increment(step, transformation_type2, COL);
138      jkincrement = create_increment(step, transformation_type2, ROW);
139
140      jkincrement_2 = jkincrement * 2;
141      step_1 = step - 1;
142
143      /* start the performance counter */
144      myprfcnt_start(cpus.pk[thread_args->pk].sib[thread_args->sib], thread_args->sib);
145
146      for (kk = 0; kk < jkbound; kk += jjkkincrement)
147      {
148         ktilebound = (kk | jkmask);
149         kup = (jkbound < ktilebound ? jkbound : ktilebound);
150         kk_plus_jjkkincrement = kk+jjkkincrement;
151
152         kkC = kk >> logNstep;
153
154         jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
155         for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
156         {
157            jtilebound = (jj | jkmask);
158            jup = (jkbound < jtilebound ? jkbound : jtilebound);
159
160            jjC = jj >> logNstep;
161
162            iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep
                 ;
163            for (ii=iilow;ii<ibound;ii+=iiincrement)
164            {
165               itilebound = (ii | imask);
166               iup = (ibound < itilebound ? ibound : itilebound);
167
168               for (k=kk; k<kup;k+=jkincrement)
169               {
170                  kC = ((k >> logstep) & step_1);
171                  kfC = kkC | kC;
172
173                  kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                       jkincrement);
174                  kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                       iincrement);
175                  ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
176
177                  if (thread_args->tid == 0)
178                  {
179                     if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
180                        if (thread_args->tid == 0)
181                           A[kfC|k]=sqrt(fabs(A[kfC|k]));
```

```
182
183                     if (jj==jjlow)
184                         for (i=ilowDiv;i<iup;i++)
185                             A[i|k]/=A[kfC|k];
186                 }
187                 spin_barrier_lsense(thread_args->spin_barrier_fine, &lsense);
188
189                 jlow = (jj > kplus1 ? jj : kplus1);
190
191                 for (j=jlow + (thread_args->tid)*jkincrement;j<jup;j+=jkincrement_2)
192                 {
193                     jC = ((j >> logstep) & step_1);
194                     jfC = jjC | jC;
195                     ilow = (ii > jfC ? ii : jfC);
196                     AjfC_k = A[jfC|k];
197
198                     iup_div = ilow + (iup-ilow)/4*4;
199                     for (i=ilow;i<iup_div;i+=4)
200                     {
201                         A[i|j]-=A[i|k]*AjfC_k;
202                         A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
203                         A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
204                         A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
205                     }
206
207                     for (;i<iup;i++)
208                         A[i|j]-=A[i|k]*AjfC_k;
209                 }
210             }
211         }
212     }
213 }
214
215 /* stop the performance counter */
216 myprfcnt_stop(cpus.pk[thread_args->pk].sib[thread_args->sib]);
217 }
218
219
220 /* Function: thread_tiled_full_bdl_NN_coarse_grain
221  * Block Data Layout NN is used
222  * Tiling is applied along all loops
223  * Hand optimization includes loop unrolling and scalar replacement
224  * Coarse Grain work partitioning, i.e Threads work on two separate adjacent tiles
225  */
226 void thread_tiled_full_bdl_NN_coarse_grain(void * args)
227 {
228     mm_thread_args_t * thread_args = (mm_thread_args_t *) args;
229
230     int i, j, k;
231     int iR, jC, kC;
232     int ii, jj, kk, jjC, kkC;
233     int jfC, kfC;
234     int kk_plus_jjkkincrement;
235
236     int N, Nmask;
237     int t_initial_ii;
238     int tile_size, step;
239     int last_iteration_ii, last_iteration_jj, last_iteration_kk;
240     double *A;
241
242     int transformation_type = NN, transformation_type2 = Nn;
243
244     int logNstep, logstep;
245
246     int imask, jkmask;
247     int ibound, jkbound, itilebound, jtilebound, ktilebound;
248     int iiincrement, jjkkincrement;
249     int iincrement, jkincrement;
250     int iup, ilow, jup, jlow, kup, klow, ilowDiv, iup_div;
251     int jjlow, iilow;
252     int step_1, kplus1, kplus1C, kplus1fC;
253
254     register double AjfC_k;
255
256     int lsense;
257
258     spin_barrier_init_lsense(&lsense);
259
260     N = thread_args->N;
261     Nmask = thread_args->Nmask;
262     A = thread_args->A_1D;
263     step = tile_size = thread_args->tile_size;
264
265     logNstep = lg(Nmask/step);
266     logstep = lg(step);
267
```

```
268     ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
269     jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
270
271     imask = create_innermask(step, transformation_type2, 0) + 1;
272     jkmask = create_innermask(step, transformation_type2, 1) + 1;
273
274     iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
275     jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
276
277
278     iincrement = create_increment(step, transformation_type2, COL);
279     jkincrement = create_increment(step, transformation_type2, ROW);
280
281     step_1 = step - 1;
282
283     // Bind the thread to the given physical package and logical processor unit
284     if(thread_args->sib == -1) {
285        run_on_cpus( ( 1 << cpus.pk[thread_args->pk].sib[0] ) | ( 1 << cpus.pk[
               thread_args->pk].sib[1] ) );
286     } else
287        run_on_cpu( cpus.pk[thread_args->pk].sib[thread_args->sib] );
288
289
290     t_initial_ii = tile_size * (thread_args->tid + 1);
291
292     last_iteration_kk = ceil((double)jkbound/jjkkincrement)*jjkkincrement-
           jjkkincrement*3;
293     last_iteration_jj = ceil((double)jkbound/jjkkincrement)*jjkkincrement-
           jjkkincrement;
294     last_iteration_ii = ceil((double)ibound/iiincrement)*iiincrement-iiincrement;
295
296     myprfcnt_start(cpus.pk[thread_args->pk].sib[thread_args->sib], thread_args->sib);
297
298     // Iterations 0:0:0 to and not including ~ N-2*Bkk:N:N are partitioned among
           threads and executed in
299     // parallel in groups
300     for (kk=0; kk<jkbound-2*jjkkincrement;kk+=jjkkincrement)
301     {
302        ktilebound = (kk | jkmask);
303        kup = (jkbound < ktilebound ? jkbound : ktilebound);
304        kk_plus_jjkkincrement = kk+jjkkincrement;
305
306        kkC = kk >> logNstep;
307
308        if ( (thread_args->tid == 0 && kk == 0) ||
309                (thread_args->tid == 0 && last_iteration_kk >=0 && (jj-jjkkincrement
                      != last_iteration_jj || ii-2*iiincrement != last_iteration_ii))
                      )
310        {
311           jj = kk;
312
313           jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
314
315           ii = jj >> logNstep;
316
317           t_initial_ii = iiincrement * (thread_args->tid + 1);
318
319
320           jtilebound = (jj | jkmask);
321           jup = (jkbound < jtilebound ? jkbound : jtilebound);
322
323           jjC = jj >> logNstep;
324
325           itilebound = (ii | imask);
326           iup = (ibound < itilebound ? ibound : itilebound);
327
328
329           // ============================================
330           for (k=kk; k<kup;k+=jkincrement)
331           {
332              kC = ((k >> logstep) & step_1);
333              kfC = kkC | kC;
334
335              kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                    jkincrement);
336              kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+iincrement
                    );
337              ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
338
339
340              if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
341                 A[kfC|k]=sqrt(fabs(A[kfC|k]));
342
343              if (jj==jjlow)
344              {
345                 for (i=ilowDiv;i<iup;i+=iincrement)
346                    A[i|k]/=A[kfC|k];
```

```
347                 }
348
349                 jlow = (jj > kplus1 ? jj : kplus1);
350                 for (j=jlow;j<jup;j+=jkincrement)
351                 {
352                    jC = ((j >> logstep) & step_1);
353                    jfC = jjC | jC;
354                    ilow = (ii > jfC ? ii : jfC);
355
356 //                 for (i=ilow;i<iup;i+=iincrement)
357 //                    A[i|j]-=A[i|k]*A[jfC|k];
358
359                    AjfC_k = A[jfC|k];
360
361                    iup_div = ilow + (iup-ilow)/4*4;
362                    for (i=ilow;i<iup_div;i+=4)
363                    {
364                       A[i|j]-=A[i|k]*AjfC_k;
365                       A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
366                       A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
367                       A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
368                    }
369
370                    for (;i<iup;i++)
371                       A[i|j]-=A[i|k]*AjfC_k;
372
373
374                 }
375              }
376              // ============================================
377
378
379              spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
380           }
381
382           if ( (thread_args->tid == 1 && kk == 0) ||
383                 (thread_args->tid == 1 && last_iteration_kk >=0 && (jj-jjkkincrement
                       == last_iteration_jj && ii-2*iiincrement == last_iteration_ii)
                       )
384                 )
385           {
386              t_initial_ii = iiincrement * (thread_args->tid + 1);
387
388              //spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
389              //if (thread_args->tgid == 0) printf("Next group\n");
390              spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
391           }
392
393
394           jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
395           for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
396           {
397              jtilebound = (jj | jkmask);
398              jup = (jkbound < jtilebound ? jkbound : jtilebound);
399
400              jjC = jj >> logNstep;
401
402              iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep
                    ;
403              for (ii=t_initial_ii + iilow;ii<ibound;ii+=2*iiincrement)
404              {
405                 itilebound = (ii | imask);
406                 iup = (ibound < itilebound ? ibound : itilebound);
407
408
409                 // ============================================
410                 for (k=kk; k<kup;k+=jkincrement)
411                 {
412                    kC = ((k >> logstep) & step_1);
413                    kfC = kkC | kC;
414
415                    kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                          jkincrement);
416                    kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                          iincrement);
417                    ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
418
419
420                    if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
421                       A[kfC|k]=sqrt(fabs(A[kfC|k]));
422
423                    if (jj==jjlow)
424                    {
425                       for (i=ilowDiv;i<iup;i+=iincrement)
426                          A[i|k]/=A[kfC|k];
427                    }
```

```
428
429                 jlow = (jj > kplus1 ? jj : kplus1);
430                 for (j=jlow;j<jup;j+=jkincrement)
431                 {
432                    jC = ((j >> logstep) & step_1);
433                    jfC = jjC | jC;
434                    ilow = (ii > jfC ? ii : jfC);
435 //                 for (i=ilow;i<iup;i+=iincrement)
436 //                    A[i|j]-=A[i|k]*A[jfC|k];
437
438                    AjfC_k = A[jfC|k];
439
440                    iup_div = ilow + (iup-ilow)/4*4;
441                    for (i=ilow;i<iup_div;i+=4)
442                    {
443                       A[i|j]-=A[i|k]*AjfC_k;
444                       A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
445                       A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
446                       A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
447                    }
448
449                    for (;i<iup;i++)
450                       A[i|j]-=A[i|k]*AjfC_k;
451                 }
452              }
453              //=============================================
454
455
456              //spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
457              //if (thread_args->tgid == 0) printf("Next group\n");
458              spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
459           }
460
461           t_initial_ii = ((ii == last_iteration_ii + iiincrement)? 0: iiincrement);
462        }
463     }
464
465     // The execution of the last 4 iterations follows
466     //=============================================
467     // In the case that the last iteration of the previous loops was executed by
         thread 0, then the first iteration of the last 4
468     // iterations should be executed by thread 1 so as to exploit the parallelism
         available in this case
469     if (thread_args->tid == 1 && last_iteration_kk >=0 &&
470              (kk-jjkkincrement != last_iteration_kk || jj-jjkkincrement !=
                   last_iteration_jj || ii-2*iiincrement != last_iteration_ii) )
471     {
472                 ktilebound = (kk | jkmask);
473                 kup = (jkbound < ktilebound ? jkbound : ktilebound);
474                 kk_plus_jjkkincrement = kk+jjkkincrement;
475
476                 kkC = kk >> logNstep;
477
478              jj = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
479              jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
480                 jtilebound = (jj | jkmask);
481                 jup = (jkbound < jtilebound ? jkbound : jtilebound);
482
483
484              jjC = jj >> logNstep;
485
486              ii = ((int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep;
487                 itilebound = (ii | imask);
488                 iup = (ibound < itilebound ? ibound : itilebound);
489
490
491              //=============================================
492              for (k=kk; k<kup;k+=jkincrement)
493              {
494                 kC = ((k >> logstep) & step_1);
495                 kfC = kkC | kC;
496
497                 kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                       jkincrement);
498                 kplus1fC = ((kfC+iincrement==itilebound)? kkC+iincrement:kfC+
                       iincrement);
499                 ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
500
501
502                 if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
503                    A[kfC|k]=sqrt(fabs(A[kfC|k]));
504
505                 if (jj==jjlow)
506                 {
507                    for (i=ilowDiv;i<iup;i+=iincrement)
```

```
508                     A[i|k]/=A[kfC|k];
509                 }
510
511                 jlow = (jj > kplus1 ? jj : kplus1);
512                 for (j=jlow;j<jup;j+=jkincrement)
513                 {
514                    jC = ((j >> logstep) & step_1);
515                    jfC = jjC | jC;
516                    ilow = (ii > jfC ? ii : jfC);
517
518 //                 for (i=ilow;i<iup;i+=iincrement)
519 //                    A[i|j]-=A[i|k]*A[jfC|k];
520
521                    AjfC_k = A[jfC|k];
522
523                    iup_div = ilow + (iup-ilow)/4*4;
524                    for (i=ilow;i<iup_div;i+=4)
525                    {
526                       A[i|j]-=A[i|k]*AjfC_k;
527                       A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
528                       A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
529                       A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
530                    }
531
532                    for (;i<iup;i++)
533                       A[i|j]-=A[i|k]*AjfC_k;
534
535
536
537                 }
538              }
539              // ============================================
540
541
542              //spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
543              //if (thread_args->tgid == 0) printf("Next group\n");
544              spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
545     }
546     // The upper iterations ~ N-2*Bkk:X:X and ~ N-Bkk:X:X are executed by thread 0.
           In the worst case there will be 4 such
547     // iterations
548     else if (thread_args->tid == 0)
549     {
550        t_initial_ii = ((kk-jjkkincrement == last_iteration_kk && jj-jjkkincrement ==
               last_iteration_jj && ii-2*iiincrement == last_iteration_ii)? iiincrement
              : 0);
551
552        for (; kk<jkbound;kk+=jjkkincrement)
553        {
554           ktilebound = (kk | jkmask);
555           kup = (jkbound < ktilebound ? jkbound : ktilebound);
556           kk_plus_jjkkincrement = kk+jjkkincrement;
557
558           kkC = kk >> logNstep;
559
560           jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
561           for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
562           {
563              jtilebound = (jj | jkmask);
564              jup = (jkbound < jtilebound ? jkbound : jtilebound);
565
566              jjC = jj >> logNstep;
567
568              iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >>
                    logNstep;
569              for (ii=t_initial_ii + iilow;ii<ibound;ii+=iiincrement)
570              {
571                 itilebound = (ii | imask);
572                 iup = (ibound < itilebound ? ibound : itilebound);
573
574
575                 // ============================================
576                 for (k=kk; k<kup;k+=jkincrement)
577                 {
578                    kC = ((k >> logstep) & step_1);
579                    kfC = kkC | kC;
580
581                    kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                          jkincrement);
582                    kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                          iincrement);
583                    ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
584
585
586                    if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
587                       A[kfC|k]=sqrt(fabs(A[kfC|k]));
```

```
588
589                     if (jj==jjlow)
590                     {
591                         for (i=ilowDiv;i<iup;i+=iincrement)
592                             A[i|k]/=A[kfC|k];
593                     }
594
595                     jlow = (jj > kplus1 ? jj : kplus1);
596                     for (j=jlow;j<jup;j+=jkincrement)
597                     {
598                         jC = ((j >> logstep) & step_1);
599                         jfC = jjC | jC;
600                         ilow = (ii > jfC ? ii : jfC);
601 //                      for (i=ilow;i<iup;i+=iincrement)
602 //                          A[i|j]-=A[i|k]*A[jfC|k];
603
604                         AjfC_k = A[jfC|k];
605
606                         iup_div = ilow + (iup-ilow)/4*4;
607                         for (i=ilow;i<iup_div;i+=4)
608                         {
609                             A[i|j]-=A[i|k]*AjfC_k;
610                             A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
611                             A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
612                             A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
613                         }
614
615                         for (;i<iup;i++)
616                             A[i|j]-=A[i|k]*AjfC_k;
617
618
619                     }
620                 }
621                 //=============================================
622
623             }
624             t_initial_ii = 0;
625         }
626     }
627 }
628 myprfcnt_stop(cpus.pk[thread_args->pk].sib[thread_args->sib]);
629 }
630
631 /* Function: thread_tiled_full_bdl_NN_coarse_fine_grain
632  * Block Data Layout NN is used
633  * Tiling is applied along all loops
634  * Hand optimization includes loop unrolling and scalar replacement
635  * Coarse Grain work partitioning, i.e Two groups of two threads work on two
       separate adjacent tiles
636  * The two threads of each group work on the same tile and divide their work in a
       fine grain fashion way
637  */
638 void thread_tiled_full_bdl_NN_coarse_fine_grain(void * args)
639 {
640     mm_thread_args_t * thread_args = (mm_thread_args_t *) args;
641
642     int i, j, k;
643     int iR, jC, kC;
644     int ii, jj, kk, jjC, kkC;
645     int jfC, kfC;
646     int kk_plus_jjkkincrement;
647
648     int N, Nmask;
649     int t_initial_ii;
650     int tile_size, step;
651     int last_iteration_ii, last_iteration_jj, last_iteration_kk;
652     double *A;
653
654     int transformation_type = NN, transformation_type2 = Nn;
655
656     int logNstep, logstep;
657
658     int imask, jkmask;
659     int ibound, jkbound, itilebound, jtilebound, ktilebound;
660     int iiincrement, jjkkincrement;
661     int iincrement, jkincrement;
662     int jkincrement_2;
663     int iup, ilow, jup, jlow, kup, klow, ilowDiv, iup_div;
664     int jjlow, iilow;
665     int step_1, kplus1, kplus1C, kplus1fC;
666
667     register double AjfC_k;
668
669     int lsense, lsense_fine;
670
```

```
671     spin_barrier_init_lsense(&lsense);
672     spin_barrier_init_lsense(&lsense_fine);
673
674     N = thread_args->N;
675     Nmask = thread_args->Nmask;
676     A = thread_args->A_1D;
677     step = tile_size = thread_args->tile_size;
678
679     logNstep = lg(Nmask/step);
680     logstep = lg(step);
681
682     ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
683     jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
684
685     imask = create_innermask(step, transformation_type2, 0) + 1;
686     jkmask = create_innermask(step, transformation_type2, 1) + 1;
687
688     iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
689     jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
690
691
692     iincrement = create_increment(step, transformation_type2, COL);
693     jkincrement = create_increment(step, transformation_type2, ROW);
694
695     jkincrement_2 = jkincrement * 2;
696     step_1 = step - 1;
697
698     // Bind the thread to the given physical package and logical processor unit
699     if(thread_args->sib == -1) {
700        run_on_cpus( ( 1 << cpus.pk[thread_args->pk].sib[0] ) | ( 1 << cpus.pk[
              thread_args->pk].sib[1] ) );
701     } else
702        run_on_cpu( cpus.pk[thread_args->pk].sib[thread_args->sib] );
703
704
705     t_initial_ii = tile_size * (thread_args->tgid + 1);
706
707     last_iteration_kk = ceil((double)jkbound/jjkkincrement)*jjkkincrement-
           jjkkincrement*3;
708     last_iteration_jj = ceil((double)jkbound/jjkkincrement)*jjkkincrement-
           jjkkincrement;
709     last_iteration_ii = ceil((double)ibound/iiincrement)*iiincrement-iiincrement;
710
711     myprfcnt_start(cpus.pk[thread_args->pk].sib[thread_args->sib], thread_args->sib);
712
713     // Iterations 0:0:0 to and not including ~ N-2*Bkk:N:N are partitioned among
           threads and executed in
714     // parallel in groups
715     for (kk=0; kk<jkbound-2*jjkkincrement;kk+=jjkkincrement)
716     {
717        ktilebound = (kk | jkmask);
718        kup = (jkbound < ktilebound ? jkbound : ktilebound);
719        kk_plus_jjkkincrement = kk+jjkkincrement;
720
721        kkC = kk >> logNstep;
722
723        if ( (thread_args->tgid == 0 && kk == 0) ||
724                (thread_args->tgid == 0 && last_iteration_kk >=0 && (jj-jjkkincrement
                      != last_iteration_jj || ii-2*iiincrement != last_iteration_ii))
                      )
725        {
726           jj = kk;
727
728           jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
729
730           ii = jj >> logNstep;
731
732           t_initial_ii = iiincrement * (thread_args->tgid + 1);
733
734
735           jtilebound = (jj | jkmask);
736           jup = (jkbound < jtilebound ? jkbound : jtilebound);
737
738           jjC = jj >> logNstep;
739
740           itilebound = (ii | imask);
741           iup = (ibound < itilebound ? ibound : itilebound);
742
743
744           // ============================================
745
746           for (k=kk; k<kup;k+=jkincrement)
747           {
748              kC = ((k >> logstep) & step_1);
749              kfC = kkC | kC;
750
```

```
751             kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                    jkincrement);
752             kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+iincrement
                    );
753             ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
754
755             if (thread_args->tid == 0)
756             {
757                 if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
758                     if (thread_args->tid == 0)
759                         A[kfC|k]=sqrt(fabs(A[kfC|k]));
760
761                 if (jj==jjlow)
762                     for (i=ilowDiv;i<iup;i++)
763                         A[i|k]/=A[kfC|k];
764             }
765
766             spin_barrier_lsense(thread_args->spin_barrier_fine, &lsense_fine);
767
768             jlow = (jj > kplus1 ? jj : kplus1);
769
770             for (j=jlow + (thread_args->tid)*jkincrement;j<jup;j+=jkincrement_2)
771             {
772                 jC = ((j >> logstep) & step_1);
773                 jfC = jjC | jC;
774                 ilow = (ii > jfC ? ii : jfC);
775                 AjfC_k = A[jfC|k];
776
777                 iup_div = ilow + (iup-ilow)/4*4;
778                 for (i=ilow;i<iup_div;i+=4)
779                 {
780                     A[i|j]-=A[i|k]*AjfC_k;
781                     A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
782                     A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
783                     A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
784                 }
785
786                 for (;i<iup;i++)
787                     A[i|j]-=A[i|k]*AjfC_k;
788             }
789
790         }
791
792         // ============================================
793
794
795         spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
796     }
797
798     if ( (thread_args->tgid == 1 && kk == 0) ||
799             (thread_args->tgid == 1 && last_iteration_kk >=0 && (jj-jjkkincrement
                    == last_iteration_jj && ii-2*iiincrement == last_iteration_ii)
                    )
800             )
801     {
802         t_initial_ii = iiincrement * (thread_args->tgid + 1);
803
804         spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
805     }
806
807
808     jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
809     for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
810     {
811         jtilebound = (jj | jkmask);
812         jup = (jkbound < jtilebound ? jkbound : jtilebound);
813
814         jjC = jj >> logNstep;
815
816         iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep
                ;
817         for (ii=t_initial_ii + iilow;ii<ibound;ii+=2*iiincrement)
818         {
819             itilebound = (ii | imask);
820             iup = (ibound < itilebound ? ibound : itilebound);
821
822
823             // ============================================
824
825             for (k=kk; k<kup;k+=jkincrement)
826             {
827                 kC = ((k >> logstep) & step_1);
828                 kfC = kkC | kC;
829
830                 kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                        jkincrement);
831                 kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                        iincrement);
```

```
832                 ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
833
834                 if (thread_args->tid == 0)
835                 {
836                     if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
837                         if (thread_args->tid == 0)
838                             A[kfC|k]=sqrt(fabs(A[kfC|k]));
839
840                     if (jj==jjlow)
841                         for (i=ilowDiv;i<iup;i++)
842                             A[i|k]/=A[kfC|k];
843                 }
844
845                 spin_barrier_lsense(thread_args->spin_barrier_fine, &lsense_fine);
846
847                 jlow = (jj > kplus1 ? jj : kplus1);
848
849                 for (j=jlow + (thread_args->tid)*jkincrement;j<jup;j+=jkincrement_2)
850                 {
851                     jC = ((j >> logstep) & step_1);
852                     jfC = jjC | jC;
853                     ilow = (ii > jfC ? ii : jfC);
854                     AjfC_k = A[jfC|k];
855
856                     iup_div = ilow + (iup-ilow)/4*4;
857                     for (i=ilow;i<iup_div;i+=4)
858                     {
859                         A[i|j]-=A[i|k]*AjfC_k;
860                         A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
861                         A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
862                         A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
863                     }
864
865                     for (;i<iup;i++)
866                         A[i|j]-=A[i|k]*AjfC_k;
867                 }
868
869             }
870             //=============================================
871
872             spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
873         }
874
875         t_initial_ii = ((ii == last_iteration_ii + iiincrement)? 0: iiincrement);
876     }
877 }
878
879 // The execution of the last 4 iterations follows
880 //=============================================
881 // In the case that the last iteration of the previous loops was executed by
        thread 0, then the first iteration of the last 4
882 // iterations should be executed by thread 1 so as to exploit the parallelism
        available in this case
883 if (thread_args->tgid == 1 && last_iteration_kk >=0 &&
884         (kk-jjkkincrement != last_iteration_kk || jj-jjkkincrement !=
                last_iteration_jj || ii-2*iiincrement != last_iteration_ii) )
885 {
886             ktilebound = (kk | jkmask);
887             kup = (jkbound < ktilebound ? jkbound : ktilebound);
888             kk_plus_jjkkincrement = kk+jjkkincrement;
889
890             kkC = kk >> logNstep;
891
892         jj = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
893         jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
894             jtilebound = (jj | jkmask);
895             jup = (jkbound < jtilebound ? jkbound : jtilebound);
896
897
898         jjC = jj >> logNstep;
899
900         ii = ((int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep;
901             itilebound = (ii | imask);
902             iup = (ibound < itilebound ? ibound : itilebound);
903
904
905         //=============================================
906
907         for (k=kk; k<kup;k+=jkincrement)
908         {
909             kC = ((k >> logstep) & step_1);
910             kfC = kkC | kC;
911
912             kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                    jkincrement);
```

```
913                 kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                        iincrement);
914                 ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
915
916                 if (thread_args->tid == 0)
917                 {
918                     if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
919                         if (thread_args->tid == 0)
920                             A[kfC|k]=sqrt(fabs(A[kfC|k]));
921
922                     if (jj==jjlow)
923                         for (i=ilowDiv;i<iup;i++)
924                             A[i|k]/=A[kfC|k];
925                 }
926
927                 spin_barrier_lsense(thread_args->spin_barrier_fine, &lsense_fine);
928
929                 jlow = (jj > kplus1 ? jj : kplus1);
930
931                 for (j=jlow + (thread_args->tid)*jkincrement;j<jup;j+=jkincrement_2)
932                 {
933                     jC = ((j >> logstep) & step_1);
934                     jfC = jjC | jC;
935                     ilow = (ii > jfC ? ii : jfC);
936                     AjfC_k = A[jfC|k];
937
938                     iup_div = ilow + (iup-ilow)/4*4;
939                     for (i=ilow;i<iup_div;i+=4)
940                     {
941                         A[i|j]-=A[i|k]*AjfC_k;
942                         A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
943                         A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
944                         A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
945                     }
946
947                     for (;i<iup;i++)
948                         A[i|j]-=A[i|k]*AjfC_k;
949                 }
950
951             }
952             //=============================================
953
954             spin_barrier_lsense(thread_args->spin_barrier_coarse, &lsense);
955     }
956     // The upper iterations ~ N-2*Bkk:X:X and ~ N-Bkk:X:X are executed by thread 0.
          In the worst case there will be 4 such
957     // iterations
958     else if (thread_args->tgid == 0)
959     {
960         t_initial_ii = ((kk-jjkkincrement == last_iteration_kk && jj-jjkkincrement ==
              last_iteration_jj && ii-2*iiincrement == last_iteration_ii)? iiincrement
              : 0);
961
962         for (; kk<jkbound;kk+=jjkkincrement)
963         {
964             ktilebound = (kk | jkmask);
965             kup = (jkbound < ktilebound ? jkbound : ktilebound);
966             kk_plus_jjkkincrement = kk+jjkkincrement;
967
968             kkC = kk >> logNstep;
969
970             jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
971             for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
972             {
973                 jtilebound = (jj | jkmask);
974                 jup = (jkbound < jtilebound ? jkbound : jtilebound);
975
976                 jjC = jj >> logNstep;
977
978                 iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >>
                      logNstep;
979                 for (ii=t_initial_ii + iilow;ii<ibound;ii+=iiincrement)
980                 {
981                     itilebound = (ii | imask);
982                     iup = (ibound < itilebound ? ibound : itilebound);
983
984
985                     //=============================================
986
987                     for (k=kk; k<kup;k+=jkincrement)
988                     {
989                         kC = ((k >> logstep) & step_1);
990                         kfC = kkC | kC;
991
992                         kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                              jkincrement);
```

```
993                     kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                            iincrement);
994                     ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
995
996                     if (thread_args->tid == 0)
997                     {
998                         if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
999                             if (thread_args->tid == 0)
1000                                A[kfC|k]=sqrt(fabs(A[kfC|k]));
1001
1002                        if (jj==jjlow)
1003                            for (i=ilowDiv;i<iup;i++)
1004                                A[i|k]/=A[kfC|k];
1005                    }
1006
1007                    spin_barrier_lsense(thread_args->spin_barrier_fine, &lsense_fine);
1008
1009                    jlow = (jj > kplus1 ? jj : kplus1);
1010
1011                    for (j=jlow + (thread_args->tid)*jkincrement;j<jup;j+=
                            jkincrement_2)
1012                    {
1013                        jC = ((j >> logstep) & step_1);
1014                        jfC = jjC | jC;
1015                        ilow = (ii > jfC ? ii : jfC);
1016                        AjfC_k = A[jfC|k];
1017
1018                        iup_div = ilow + (iup-ilow)/4*4;
1019                        for (i=ilow;i<iup_div;i+=4)
1020                        {
1021                            A[i|j]-=A[i|k]*AjfC_k;
1022                            A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
1023                            A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
1024                            A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
1025                        }
1026
1027                        for (;i<iup;i++)
1028                            A[i|j]-=A[i|k]*AjfC_k;
1029                    }
1030
1031                }
1032
1033                // ============================================
1034
1035            }
1036            t_initial_ii = 0;
1037        }
1038    }
1039    }
1040    myprfcnt_stop(cpus.pk[thread_args->pk].sib[thread_args->sib]);
1041
1042 }
1043
1044 /* Function: thread_tiled_full_bdl_NN
1045  * Block Data Layout NN is used
1046  * Tiling is applied along all loops
1047  */
1048 void thread_tiled_full_bdl_NN(void *args)
1049 {
1050    mm_thread_args_t * thread_args = (mm_thread_args_t *) args;
1051
1052    int i, j, k;
1053    int iR, jC, kC;
1054    int ii, jj, kk, jjC, kkC;
1055    int jfC, kfC;
1056    int kk_plus_jjkkincrement;
1057
1058    int transformation_type = NN, transformation_type2 = Nn;
1059
1060    int logNstep, logstep;
1061
1062    int imask, jkmask;
1063    int ibound, jkbound, itilebound, jtilebound, ktilebound;
1064    int iiincrement, jjkkincrement;
1065    int iincrement, jkincrement;
1066    int iup, ilow, jup, jlow, kup, klow, ilowDiv;
1067    int jjlow, iilow;
1068    int step, step_1, kplus1, kplus1C, kplus1fC;
1069
1070    int N, Nmask;
1071    double *A;
1072    register double AjfC_k;
1073
1074    // Bind the thread to the given physical package and logical processor unit
1075    if(thread_args->sib == -1) {
1076       run_on_cpus( ( 1 << cpus.pk[thread_args->pk].sib[0]) | ( 1 << cpus.pk[
               thread_args->pk].sib[1]) );
```

```
1077      } else
1078         run_on_cpu( cpus.pk[thread_args->pk].sib[thread_args->sib] );
1079
1080      // Some assignments (aliases) and initializations
1081      N = thread_args->N;
1082      Nmask = thread_args->Nmask;
1083      A = thread_args->A_1D;
1084      step = thread_args->tile_size;
1085
1086      logNstep = lg(Nmask/step);
1087      logstep = lg(step);
1088
1089      ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
1090      jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
1091
1092      imask = create_innermask(step, transformation_type2, 0) + 1;
1093      jkmask = create_innermask(step, transformation_type2, 1) + 1;
1094
1095      iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
1096      jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
1097
1098
1099      iincrement = create_increment(step, transformation_type2, COL);
1100      jkincrement = create_increment(step, transformation_type2, ROW);
1101
1102      step_1 = step - 1;
1103
1104      myprfcnt_start(cpus.pk[thread_args->pk].sib[thread_args->sib], thread_args->sib);
1105
1106      for (kk = 0; kk < jkbound; kk += jjkkincrement)
1107      {
1108         ktilebound = (kk | jkmask);
1109         kup = (jkbound < ktilebound ? jkbound : ktilebound);
1110         kk_plus_jjkkincrement = kk+jjkkincrement;
1111
1112         kkC = kk >> logNstep;
1113
1114         jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
1115         for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
1116         {
1117            jtilebound = (jj | jkmask);
1118            jup = (jkbound < jtilebound ? jkbound : jtilebound);
1119
1120            jjC = jj >> logNstep;
1121
1122            iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep
                  ;
1123            for (ii=iilow;ii<ibound;ii+=iiincrement)
1124            {
1125               itilebound = (ii | imask);
1126               iup = (ibound < itilebound ? ibound : itilebound);
1127
1128               for (k=kk; k<kup;k+=jkincrement)
1129               {
1130                  kC = ((k >> logstep) & step_1);
1131                  kfC = kkC | kC;
1132
1133                  kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                        jkincrement);
1134                  kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                        iincrement);
1135                  ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
1136
1137
1138                  if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
1139                     A[kfC|k]=sqrt(fabs(A[kfC|k]));
1140
1141                  if (jj==jjlow)
1142                  {
1143                     for (i=ilowDiv;i<iup;i+=iincrement)
1144                        A[i|k]/=A[kfC|k];
1145                  }
1146
1147                  jlow = (jj > kplus1 ? jj : kplus1);
1148                  for (j=jlow;j<jup;j+=jkincrement)
1149                  {
1150                     jC = ((j >> logstep) & step_1);
1151                     jfC = jjC | jC;
1152                     ilow = (ii > jfC ? ii : jfC);
1153                     AjfC_k = A[jfC|k];
1154                     for (i=ilow;i<iup;i++)
1155                        A[i|j]-=A[i|k]*AjfC_k;
1156                  }
1157               }
1158            }
1159         }
1160      }
1161
```

```
1162     myprfcnt_stop(cpus.pk[thread_args->pk].sib[thread_args->sib]);
1163  }
1164
1165  /* Function: thread_tiled_full_bdl_NN
1166   * Block Data Layout NN is used
1167   * Tiling is applied along all loops
1168   * Hand optimization includes loop unrolling and scalar replacement
1169   */
1170  void thread_tiled_full_bdl_NN_hopt(void *args)
1171  {
1172     mm_thread_args_t * thread_args = (mm_thread_args_t *) args;
1173
1174     int i, j, k;
1175     int iR, jC, kC;
1176     int ii, jj, kk, jjC, kkC;
1177     int jfC, kfC;
1178     int kk_plus_jjkkincrement;
1179
1180     int transformation_type = NN, transformation_type2 = Nn;
1181
1182     int logNstep, logstep;
1183
1184     int imask, jkmask;
1185     int ibound, jkbound, itilebound, jtilebound, ktilebound;
1186     int iiincrement, jjkkincrement;
1187     int iincrement, jkincrement;
1188     int iup, ilow, jup, jlow, kup, klow, ilowDiv, ilow_div, iup_div;
1189     int jjlow, iilow;
1190     double AjfC_k;
1191     int step, step_1, kplus1, kplus1C, kplus1fC;
1192     int N, Nmask;
1193     double prefetch1, prefetch2;
1194     double *A;
1195
1196     // Bind the thread to the given physical package and logical processor unit
1197     if(thread_args->sib == -1) {
1198        run_on_cpus( ( 1 << cpus.pk[thread_args->pk].sib[0] ) | ( 1 << cpus.pk[
               thread_args->pk].sib[1]) );
1199     } else
1200        run_on_cpu( cpus.pk[thread_args->pk].sib[thread_args->sib] );
1201
1202     // Some assignments (aliases) and initializations
1203     N = thread_args->N;
1204     Nmask = thread_args->Nmask;
1205     A = thread_args->A_1D;
1206     step = thread_args->tile_size;
1207
1208     logNstep = lg(Nmask/step);
1209     logstep = lg(step);
1210
1211     ibound = create_mask(N, Nmask, step, transformation_type, COL) + 1;
1212     jkbound = create_mask(N, Nmask, step, transformation_type, ROW) + 1;
1213
1214     imask = create_innermask(step, transformation_type2, 0) + 1;
1215     jkmask = create_innermask(step, transformation_type2, 1) + 1;
1216
1217     iiincrement = create_tileincrement(Nmask, step, transformation_type2, COL);
1218     jjkkincrement = create_tileincrement(Nmask, step, transformation_type2, ROW);
1219
1220     iincrement = create_increment(step, transformation_type2, COL);
1221     jkincrement = create_increment(step, transformation_type2, ROW);
1222
1223     step_1 = step - 1;
1224
1225     myprfcnt_start(cpus.pk[thread_args->pk].sib[thread_args->sib], thread_args->sib);
1226
1227     for (kk = 0; kk < jkbound; kk += jjkkincrement)
1228     {
1229        ktilebound = (kk | jkmask);
1230        kup = (jkbound < ktilebound ? jkbound : ktilebound);
1231        kk_plus_jjkkincrement = kk+jjkkincrement;
1232
1233        kkC = kk >> logNstep;
1234
1235        jjlow = (int) floor((double)kk/jjkkincrement)*jjkkincrement;
1236        for (jj = jjlow; jj < jkbound; jj += jjkkincrement )
1237        {
1238           jtilebound = (jj | jkmask);
1239           jup = (jkbound < jtilebound ? jkbound : jtilebound);
1240
1241           jjC = jj >> logNstep;
1242
1243           iilow = ( (int) floor((double)jj/jjkkincrement)*jjkkincrement ) >> logNstep
                  ;
1244           for (ii=iilow;ii<ibound;ii+=iiincrement)
1245           {
```

```
1246                itilebound = (ii | imask);
1247                iup = (ibound < itilebound ? ibound : itilebound);
1248                for (k=kk; k<kup;k+=jkincrement)
1249                {
1250                   kC = ((k >> logstep) & step_1);
1251                   kfC = kkC | kC;
1252
1253                   kplus1 = ((k+jkincrement==ktilebound)? kk_plus_jjkkincrement:k+
                           jkincrement);
1254                   kplus1fC = ((kfC+iincrement==itilebound)? kkC+iiincrement:kfC+
                           iincrement);
1255                   ilowDiv = (ii > kplus1fC ? ii : kplus1fC);
1256
1257
1258                   if (jj<=k && k<jtilebound  && ii<=kfC && kfC<itilebound)
1259                      A[kfC|k]=sqrt(fabs(A[kfC|k]));
1260
1261                   if (jj==jjlow)
1262                   {
1263                      for (i=ilowDiv;i<iup;i+=iincrement)
1264                         A[i|k]/=A[kfC|k];
1265                   }
1266
1267                   jlow = (jj > kplus1 ? jj : kplus1);
1268                   for (j=jlow;j<jup;j+=jkincrement)
1269                   {
1270                      jC = ((j >> logstep) & step_1);
1271                      jfC = jjC | jC;
1272                      ilow = (ii > jfC ? ii : jfC);
1273
1274                      AjfC_k = A[jfC|k];
1275
1276                      iup_div = ilow + (iup-ilow)/4*4;
1277                      for (i=ilow;i<iup_div;i+=4)
1278                      {
1279                         A[i|j]-=A[i|k]*AjfC_k;
1280                         A[(i + 1)|j]-=A[(i + 1)|k]*AjfC_k;
1281                         A[(i + 2)|j]-=A[(i + 2)|k]*AjfC_k;
1282                         A[(i + 3)|j]-=A[(i + 3)|k]*AjfC_k;
1283                      }
1284
1285                      for (;i<iup;i++)
1286                         A[i|j]-=A[i|k]*AjfC_k;
1287                   }
1288                }
1289             }
1290          }
1291       }
1292       myprfcnt_stop(cpus.pk[thread_args->pk].sib[thread_args->sib]);
1293    }
1294
1295    /* Function: cholesky_tiled_full_bdl_NN_smt_fine
1296     * SMT Architecture
1297     * cholesky_tiled_full_bdl_NN_smt_fine creates 2 threads using
           thread_tiled_full_bdl_NN_smt_fine
1298     * and assigns these 2 threads to two logical processors of the same physical
           package
1299     */
1300    void cholesky_tiled_full_bdl_NN_smt_fine(double *A, int N, int Nmask, int tile_size)
1301    {
1302       pthread_t tids[2];
1303       mm_thread_args_t t0, t1;
1304
1305       t0.pk = 0;
1306       t1.pk = 0;
1307       t0.sib = 0;
1308       t1.sib = 1;
1309
1310       t0.tid = 0;
1311       t1.tid = 1;
1312
1313       t0.N = t1.N = N;
1314       t0.Nmask = t1.Nmask = Nmask;
1315       t0.tile_size = t1.tile_size = tile_size;
1316       t0.A_1D = t1.A_1D = A;
1317       t0.n_threads = t1.n_threads = 2;
1318
1319       t0.spin_barrier_fine = t1.spin_barrier_fine = &spin_barrier_A2;
1320       spin_barrier_init(t0.spin_barrier_fine, 2);
1321
1322       pthread_create(&tids[0], NULL, thread_tiled_full_bdl_NN_fine_grain, (void *) &t0)
              ;
1323       pthread_create(&tids[1], NULL, thread_tiled_full_bdl_NN_fine_grain, (void *) &t1)
              ;
1324
1325       pthread_join(tids[0], NULL);
```

```
1326     pthread_join(tids[1], NULL);
1327  }
1328
1329  /* Function: cholesky_tiled_full_bdl_NN_smt_coarse
1330   * SMT Architecture
1331   * cholesky_tiled_full_bdl_NN_smt_coarse creates 2 threads using
         thread_tiled_full_bdl_NN_smt_coarse
1332   * and assigns these 2 threads to two logical processors of the same physical
         package
1333   */
1334  void cholesky_tiled_full_bdl_NN_smt_coarse(double *A, int N, int Nmask, int
       tile_size)
1335  {
1336     pthread_t tids[2];
1337     mm_thread_args_t t0, t1;
1338
1339     t0.pk = 0;
1340     t1.pk = 0;
1341     t0.sib = 0;
1342     t1.sib = 1;
1343
1344     t0.tid = 0;
1345     t1.tid = 1;
1346     t0.N = t1.N = N;
1347     t0.Nmask = t1.Nmask = Nmask;
1348     t0.tile_size = t1.tile_size = tile_size;
1349     t0.A_1D = t1.A_1D = A;
1350     t0.n_threads = t1.n_threads = 2;
1351
1352     t0.spin_barrier_coarse = t1.spin_barrier_coarse = &spin_barrier_A2;
1353     spin_barrier_init(t0.spin_barrier_coarse, 2);
1354
1355     pthread_create(&tids[0], NULL, thread_tiled_full_bdl_NN_coarse_grain, (void *) &
           t0);
1356     pthread_create(&tids[1], NULL, thread_tiled_full_bdl_NN_coarse_grain, (void *) &
           t1);
1357
1358     pthread_join(tids[0], NULL);
1359     pthread_join(tids[1], NULL);
1360  }
1361
1362  /* Function: cholesky_tiled_full_bdl_NN_smp_coarse
1363   * SMP Architecture
1364   * cholesky_tiled_full_bdl_NN_smp_coarse creates 2 threads using
         thread_tiled_full_bdl_NN_smp_coarse
1365   * and assigns these 2 threads to two logical processors of a different physical
         package
1366   */
1367  void cholesky_tiled_full_bdl_NN_smp_coarse(double *A, int N, int Nmask, int
       tile_size)
1368  {
1369     pthread_t tids[2];
1370     mm_thread_args_t t0, t1;
1371
1372     t0.pk = 0;
1373     t1.pk = 1;
1374     t0.sib = 0;
1375     t1.sib = 0;
1376
1377     t0.tid = 0;
1378     t1.tid = 1;
1379     t0.N = t1.N = N;
1380     t0.Nmask = t1.Nmask = Nmask;
1381     t0.tile_size = t1.tile_size = tile_size;
1382     t0.A_1D = t1.A_1D = A;
1383     t0.n_threads = t1.n_threads = 2;
1384
1385     t0.spin_barrier_coarse = t1.spin_barrier_coarse = &spin_barrier_A2;
1386     spin_barrier_init(t0.spin_barrier_coarse, 2);
1387
1388     pthread_create(&tids[0], NULL, thread_tiled_full_bdl_NN_coarse_grain, (void *) &
           t0);
1389     pthread_create(&tids[1], NULL, thread_tiled_full_bdl_NN_coarse_grain, (void *) &
           t1);
1390
1391     pthread_join(tids[0], NULL);
1392     pthread_join(tids[1], NULL);
1393  }
1394
1395  /* Function: cholesky_tiled_full_bdl_NN_smt_fine_smp_coarse
1396   * SMT Architecture
1397   * cholesky_tiled_full_bdl_NN_smt_fine_smp_coarse creates 2 groups of 2 threads
         using
```

```
1398  * thread_tiled_full_bdl_NN_smt_fine_smp_coarse
1399  * Each group is assigned to a different physical package and each thread of the
          group
1400  * to a different logical processor of the same physical package
1401  */
1402 void cholesky_tiled_full_bdl_NN_smt_fine_smp_coarse(double *A, int N, int Nmask, int
          tile_size)
1403 {
1404    pthread_t tids[4];
1405    mm_thread_args_t t0, t1, t2, t3;
1406
1407    t0.pk = 0;  t0.sib = 0;
1408    t1.pk = 0;  t1.sib = 1;
1409    t2.pk = 1;  t2.sib = 0;
1410    t3.pk = 1;  t3.sib = 1;
1411
1412    t0.tgid = 0;   t0.tid = 0;
1413    t1.tgid = 0;   t1.tid = 1;
1414    t2.tgid = 1;   t2.tid = 0;
1415    t3.tgid = 1;   t3.tid = 1;
1416
1417    t0.N = t1.N = t2.N = t3.N = N;
1418    t0.Nmask = t1.Nmask = t2.Nmask = t3.Nmask = Nmask;
1419    t0.tile_size = t1.tile_size = t2.tile_size = t3.tile_size = tile_size;
1420    t0.A_1D = t1.A_1D = t2.A_1D = t3.A_1D = A;
1421    t0.n_threads = t1.n_threads = t2.n_threads = t3.n_threads = 4;
1422
1423    t0.spin_barrier_coarse = t1.spin_barrier_coarse = \
1424       t2.spin_barrier_coarse = t3.spin_barrier_coarse = &spin_barrier_A4;
1425    t0.spin_barrier_fine = t1.spin_barrier_fine = &spin_barrier_A2;
1426    t2.spin_barrier_fine = t3.spin_barrier_fine = &spin_barrier_B2;
1427
1428    spin_barrier_init(&spin_barrier_A4, 4);
1429    spin_barrier_init(&spin_barrier_A2, 2);
1430    spin_barrier_init(&spin_barrier_B2, 2);
1431
1432    pthread_create(&tids[0], NULL, thread_tiled_full_bdl_NN_coarse_fine_grain, (void
          *) &t0);
1433    pthread_create(&tids[1], NULL, thread_tiled_full_bdl_NN_coarse_fine_grain, (void
          *) &t1);
1434    pthread_create(&tids[2], NULL, thread_tiled_full_bdl_NN_coarse_fine_grain, (void
          *) &t2);
1435    pthread_create(&tids[3], NULL, thread_tiled_full_bdl_NN_coarse_fine_grain, (void
          *) &t3);
1436
1437    pthread_join(tids[0], NULL);
1438    pthread_join(tids[1], NULL);
1439    pthread_join(tids[2], NULL);
1440    pthread_join(tids[3], NULL);
1441 }
1442
1443 /* Function: cholesky_tiled_full_bdl_NN
1444  * cholesky_tiled_full_bdl_NN creates 1 thread using thread_tiled_full_bdl_NN to
          implement the benchmark
1445  */
1446 void cholesky_tiled_full_bdl_NN(double *A, int N, int Nmask, int tile_size)
1447 {
1448    pthread_t tids[1];
1449    mm_thread_args_t t0;
1450
1451    t0.pk = 0;  t0.sib = 0;
1452
1453    t0.N = N;
1454    t0.Nmask = Nmask;
1455    t0.A_1D = A;
1456    t0.tile_size = tile_size;
1457
1458    pthread_create(&tids[0], NULL, thread_tiled_full_bdl_NN, (void *) &t0);
1459
1460    pthread_join(tids[0], NULL);
1461 }
1462
1463 /* Function: cholesky_tiled_full_bdl_NN_hopt
1464  * cholesky_tiled_full_bdl_NN_hopt creates 1 thread using
          thread_tiled_full_bdl_NN_hopt to implement the benchmark
1465  */
1466 void cholesky_tiled_full_bdl_NN_hopt(double *A, int N, int Nmask, int tile_size)
1467 {
1468    pthread_t tids[1];
1469    mm_thread_args_t t0;
1470
1471    t0.pk = 0;  t0.sib = 0;
1472
1473    t0.N = N;
1474    t0.Nmask = Nmask;
```

```
1475      t0.A_1D = A;
1476      t0.tile_size = tile_size;
1477
1478      pthread_create(&tids[0], NULL, thread_tiled_full_bdl_NN_hopt, (void *) &t0);
1479
1480      pthread_join(tids[0], NULL);
1481  }
1482
1483  main(int argc, char **argv)
1484  {
1485      int i, j, k;
1486      int optimization_level=0, tile_size;
1487      int N, Nincr, Nsize;
1488      int platform_id = 0, prf_flag = 1;
1489      struct timeval start,finish;
1490
1491      double **B_2D;
1492      double *B;
1493
1494      for (i=0;i<argc;i++)
1495          if (strcmp(argv[i],"-N")==0)
1496              N=atoi(argv[++i]);
1497          else if (strcmp(argv[i],"-O")==0)
1498              optimization_level=atoi(argv[++i]);
1499          else if (strcmp(argv[i],"-platform")==0)
1500              platform_id=atoi(argv[++i]);
1501          else if (strcmp(argv[i],"-prfcnt")==0)
1502          {
1503              prf_flag = 1;
1504              prf_cpu=atoi(argv[++i]);
1505          }
1506
1507      /* processor IDs */
1508      if (platform_id == 0)
1509      {
1510          /* processor IDs for the CSLab Xeon Cluster */
1511          printf("Platform: CSLab Xeon Cluster\n");
1512          cpus.pk[0].sib[0] = 0;
1513          cpus.pk[0].sib[1] = 2;
1514          cpus.pk[1].sib[0] = 1;
1515          cpus.pk[1].sib[1] = 3;
1516      }
1517      else if (platform_id == 1)
1518      {
1519          /* processor IDs for the Intel Xeon Cluster */
1520          printf("Platform: Intel Xeon Cluster\n");
1521          cpus.pk[0].sib[0] = 0;
1522          cpus.pk[0].sib[1] = 4;
1523          cpus.pk[1].sib[0] = 1;
1524          cpus.pk[1].sib[1] = 5;
1525      }
1526      else if (platform_id == 2)
1527      {
1528          /* processor IDs for the Intel Xeon Cluster */
1529          printf("Platform: Intel Xeon Cluster\n");
1530          cpus.pk[0].sib[0] = 0;
1531          cpus.pk[0].sib[1] = 1;
1532          cpus.pk[1].sib[0] = 2;
1533          cpus.pk[1].sib[1] = 3;
1534      }
1535
1536      printf("Running Cholesky with optimization level %d...\n", optimization_level);
1537      printf("N\tNsize\tTile Size\tDuration\n");
1538
1539      srand(time(NULL));
1540
1541      Nincr=16;
1542
1543      for (N=16; N<=2048; N+=Nincr) {
1544        if (N<128) Nincr=N;
1545          else Nincr=128;
1546          for (tile_size=16; tile_size<=MIN(256,N); tile_size*=2) {
1547              Nsize=2;
1548              while (Nsize<N) Nsize*=2;
1549
1550              if (optimization_level >= 10) {
1551                  if (B=(double *) malloc(Nsize*Nsize*sizeof(double)))
1552                      for (i=0;i<Nsize*Nsize;i++)
1553                              B[i]=rand() % 100 +1;
1554                  else {
1555                      printf("Memory allocation failed!\n");
1556                      exit(1);
1557                  }
1558              }
1559              else {
1560                  if (B_2D=(double **) malloc(N*sizeof(double *) + N*N*sizeof(double))) {
1561                    B_2D[0]=(double *) (&B_2D[N]);
```

```
1562
1563                  for (i=1;i<N;i++)
1564                   B_2D[i]=&B_2D[0][N*i];
1565
1566                   for (i=0;i<N;i++)
1567                      for (j=0;j<N;j++)
1568                         B_2D[i][j] = rand() % 100 + 1;
1569               }
1570               else {
1571                   printf("Memory allocation failed!\n");
1572                   exit(1);
1573               }
1574           }
1575
1576           gettimeofday(&start,(struct timezone*) NULL);
1577
1578           switch (optimization_level)
1579           {
1580               case 0:
1581                   cholesky_column_major(B_2D, N);
1582                   break;
1583               case 1:
1584                   cholesky_column_major_smt_fine(B_2D, N);
1585                   break;
1586               case 2:
1587                   cholesky_tiled_full(B_2D, N, tile_size);
1588                   break;
1589               case 3:
1590                   cholesky_tiled_full_hopt(B_2D, N, tile_size);
1591                   break;
1592               case 4:
1593                   cholesky_tiled_full_smp_coarse(B_2D, N, tile_size);
1594                   break;
1595               case 5:
1596                   cholesky_tiled_full_smt_coarse(B_2D, N, tile_size);
1597                   break;
1598               case 6:
1599                   cholesky_tiled_full_smt_fine(B_2D, N, tile_size);
1600                   break;
1601               case 7:
1602                   cholesky_tiled_full_smt_fine_smp_coarse(B_2D, N, tile_size);
1603                   break;
1604               case 12:
1605                   cholesky_tiled_full_bdl_NN(B, N, Nsize, tile_size);
1606                   break;
1607               case 13:
1608                   cholesky_tiled_full_bdl_NN_hopt(B, N, Nsize, tile_size);
1609                   break;
1610               case 14:
1611                   cholesky_tiled_full_bdl_NN_smp_coarse(B, N, Nsize, tile_size);
1612                   break;
1613               case 15:
1614                   cholesky_tiled_full_bdl_NN_smt_coarse(B, N, Nsize, tile_size);
1615                   break;
1616               case 16:
1617                   cholesky_tiled_full_bdl_NN_smt_fine(B, N, Nsize, tile_size);
1618                   break;
1619               case 17:
1620                   cholesky_tiled_full_bdl_NN_smt_fine_smp_coarse(B, N, Nsize, tile_size
                        );
1621                   break;
1622               default:
1623                   cholesky_column_major(B_2D, N);
1624           }
1625           gettimeofday(&finish,(struct timezone*) NULL);
1626
1627
1628           /* Show report */
1629           printf("%d\t%d\t%d\t",N, Nsize, tile_size);
1630           printf("%8.3lf\n", finish.tv_sec-start.tv_sec
1631                  + (double)((finish.tv_usec-start.tv_usec>=0)?(finish.tv_usec-start.
                       tv_usec):
1632                  (1000000+finish.tv_usec-start.tv_usec))/1000000);
1633
1634           if (optimization_level >= 10)
1635               free(B);
1636           else
1637               free(B_2D);
1638       }
1639   }
1640 }
```

Β΄.3 Βοηθητικές Συναρτήσεις

Β΄.3.1 Μεθόδου MBaLt

Οι ακόλουθες συναρτήσεις βασίζονται στις συναρτήσεις της Ευαγγελίας Αθανασάκη.

```
1  /* MASK ADDRESSING for BLOCK DATA LAYOUT
2   * Based on Evaggelia Athanasaki's methods
3   *
4   * Different methods used to implement masked addressing for block data layouts
5   *
6   */
7
8  #include <math.h>
9
10 typedef enum {Zz=0, Nn, ZZ, NN, ZN, NZ} transform_type_def;
11 typedef enum {COL=0, ROW} colrow_type_def;
12
13 int lg(double x)
14 {
15    int l = 0;
16
17    while (x>1)
18    {
19       x/=2;
20       l++;
21    }
22    return l;
23 }
24
25 /* Function:    create_innermask
26  * Creates the mask for an inner loop, which can be used as the bound of the tile
27  */
28 int create_innermask(int step, transform_type_def type, colrow_type_def col_row)
29 {
30    int logstep;
31
32    logstep = lg(step);
33
34    switch (type) {
35       case Zz:
36          if (col_row==ROW) return (step-1);
37          else return ((step-1)<<logstep);
38       case Nn:
39          if (col_row==ROW) return ((step-1)<<logstep);
40          else return (step-1);
41       default:
42          return 0;
43    }
44 }
45
46 /* Function:    create_increment
47  * Creates the increment used to step between elements of the same tile
48  */
49 int create_increment(int step, transform_type_def type, colrow_type_def col_row)
50 {
51    int logstep;
52
53    logstep = lg(step);
54
55    switch (type) {
56       case Zz:
57          if (col_row==ROW) return 1;
58          else return (1<<logstep);
59       case Nn:
60          if (col_row==ROW) return (1<<logstep);
61          else return 1;
62       default:
63          return 0;
64    }
65 }
66
67 /* Function:    create_tileincrement
68  * Creates the increment used to step between tiles
69  */
70 int create_tileincrement(int N, int step, transform_type_def type, colrow_type_def
       col_row)
71 {
72    int logstep, logNstep;
73
74    logstep = lg(step);
75    logNstep = lg(N/step);
76
```

```
77      switch (type) {
78          case Zz:
79              if (col_row==ROW) return (1<<(2*logstep));
80              else return (1<<(2*logstep+logNstep));
81            break;
82          case Nn:
83              if (col_row==ROW) return (1<<(2*logstep+logNstep));
84              else return (1<<(2*logstep));
85          default:
86              return 0;
87      }
88  }
89
90  /* Function:    create_mask
91   * Creates the mask for an outer loop, which can be used as the bound of the loop
         space
92   */
93  int create_mask(int N, int Nmask, int step, transform_type_def type, colrow_type_def
        col_row)
94  {
95      int logstep, logNstep, Nstep;
96
97      Nstep = Nmask / step;
98      logstep = lg(step);
99      logNstep = lg(Nstep);
100     N=N-1;
101     switch (type) {
102         case ZZ:
103             if (col_row==COL) return (((N&(step-1))<<logstep)|((N&((Nstep-1)<<logstep))
                    <<(logstep+logNstep)));
104                 else return ((N&(step-1))|((N&((Nstep-1)<<logstep))<<logstep));
105         case NN:
106             if (col_row==COL) return ((N&(step-1))|((N&((Nstep-1)<<logstep))<<logstep))
                    ;
107                 else return (((N&(step-1))<<logstep)|((N&((Nstep-1)<<logstep))<<(logstep
                        +logNstep)));
108         case ZN:
109             if (col_row==COL) return (((N&(step-1))<<logstep)|((N&((Nstep-1)<<logstep))
                    <<logstep));
110                 else return ((N&(step-1))|((N&((Nstep-1)<<logstep))<<(logstep+logNstep))
                        );
111         case NZ:
112             if (col_row==COL) return ((N&(step-1))|((N&((Nstep-1)<<logstep))<<(logstep+
                    logNstep)));
113                 else return (((N&(step-1))<<logstep)|((N&((Nstep-1)<<logstep))<<logstep)
                        );
114         default:
115             return 0;
116     }
117 }
118
119 /* Function:    apply_mask
120  * Dilates a number using a mask
121  * this function is deprecated and is no more used
122  */
123 int apply_mask(unsigned long int number, unsigned long int mask)
124 {
125     unsigned long int temp_mask, apply_bit, result;
126     int mask_length, i;
127
128     result = 0;
129     temp_mask=mask;
130     mask_length=1;
131     while ((temp_mask >>= 1) > 0)
132         mask_length++;
133
134     mask <<= mask_length ;
135     apply_bit = 1 << mask_length;
136
137     for (i = 1; i <= mask_length; i++) {
138         if ((apply_bit & mask) == apply_bit) {
139             if ((number & 1) == 1)
140                 result = result | apply_bit;
141             number >>= 1;
142         }
143         result >>= 1;
144         mask >>= 1;
145     }
146
147     return result;
148 }
149
150 /* Function:    displayBits
151  * Prints the binary representation of a number
152  */
```

```
153 void displayBits(unsigned long int value, int length)
154 {
155     unsigned c, displayMask = 1 << length-1;
156
157     printf("%7u = ", value);
158
159     for (c = 1; c <= length; c++) {
160         putchar(value & displayMask ? '1' : '0');
161         value <<= 1;
162     }
163
164     putchar('\n');
165 }
```

B΄.3.2 CPU Affinity

```
1  #ifndef __AFF_H_
2  #define __AFF_H_
3
4      /* processor id type
5       * each logical processor has its own id, which is defined by the OS
6       */
7      typedef int cpu_t;
8
9      /* number of physical packages—processors in the cluster */
10     #define NPACKAGES     2
11
12     /* siblings: the logical processors found in one physical package */
13     #define NSIBLINGS     2
14
15     /* physical package type */
16     struct pkginfo {
17         cpu_t sib[NSIBLINGS];
18     };
19
20     /* Cluster processor info */
21     struct cpuinfo {
22         struct pkginfo pk[NPACKAGES];
23     };
24
25     extern void run_on_cpu(cpu_t);
26     extern void run_on_cpus(int);
27 #endif
```

```
1  #include <sys/types.h>
2  #include <linux/unistd.h>
3  #include "aff.h"
4  _syscall0(pid_t,gettid);
5  _syscall3(int,sched_setaffinity,pid_t,pid,unsigned int,len,unsigned long *,maskptr)
6
7  void run_on_cpu(cpu_t cpu)
8  {
9      unsigned long mask;
10     unsigned int len = sizeof(mask);
11     mask = 1<<cpu;
12     if (sched_setaffinity(gettid(), len, &mask))
13         printf("ERROR: Process %d not bound succesfully on processor %d\n", gettid(),
               cpu);
14 }
15
16 void run_on_cpus(cpu_t msk)
17 {
18     unsigned long mask=msk;
19     unsigned int len = sizeof(mask);
20     sched_setaffinity(gettid(), len, &mask);
21 }
```

B΄.3.3 Spin-Locks

Ο ακόλουθος κώδικας είναι προσφορά του Νικόλαου Αναστόπουλου.

```
1  #ifndef __SPIN_H_
2  #define __SPIN_H_
3
4  #include <asm/unistd.h>
5
6  /************************** spin—locks ****************************************/
7
8  #define LOCK_PREFIX "lock;"
```

```
 9  typedef volatile long spin_t;
10
11  #define spin_init spin_unlock
12
13  #if defined NOP
14  #  define SPIN_INSTR    "rep; nop"
15  #elif defined PAUSE
16  #  define SPIN_INSTR "pause"
17  #endif
18
19  /******************************** barriers **********************************/
20  struct spin_barrier_s {
21     spin_t lock;
22     spin_t release_flag;
23     int current_count;
24     int total;
25  };
26  typedef struct spin_barrier_s spin_barrier_t;
27
28
29
30  extern void spin_lock(spin_t *spin_var);
31  extern void spin_unlock(spin_t *spin_var);
32  extern void spin_on_condition(spin_t *spin_var, const int condition_val);
33
34  extern void spin_barrier_init(spin_barrier_t *sb, int total);
35  extern void spin_barrier_init_lsense(int *lsense);
36  extern void spin_barrier_lsense(spin_barrier_t *sb, int *lsense);
37
38  #endif
```

```
 1  #include "spin.h"
 2
 3  inline void spin_lock(spin_t *spin_var)
 4  {
 5     __asm__ __volatile__ ("\n"
 6
 7
 8           "1:\tmovl $1, %%eax\n\t"
 9           "xchgl %0,%%eax\n\t"
10           "cmpl $0,%%eax\n\t"
11           "jne 2f\n\t"
12           "jmp 3f\n"
13           "2:\t" SPIN_INSTR "\n\t"
14           "cmpl $0,%0\n\t"
15           "jne 2b\n\t"
16           "jmp 1b\n"
17           "3:\t\n"
18           : "=m" (*spin_var)
19           : "m" (*spin_var)
20           : "%eax","memory"
21           );
22  }
23
24  inline void spin_unlock(spin_t *spin_var)
25  {
26     __asm__ __volatile__("movl $0, %0" \
27                     :"=m"(*spin_var));
28  }
29
30
31  /******************************** barriers **********************************/
32  inline void spin_on_condition(spin_t *spin_var, const int condition_val)
33  {
34     __asm__ __volatile__ ("\n"
35           "cmpl %1, %0\n"
36           "je 2f\n"
37           "1:\tpause\n\t"
38           "cmpl %1, %0\n\t"
39           "jne 1b\n"
40           "2:\t\n"
41           :
42           :"m"(*spin_var), "ir"(condition_val)
43           );
44
45  }
46
47  inline void spin_barrier_init(spin_barrier_t *sb, int total)
48  {
49     spin_init(&(sb->lock));
50     sb->release_flag = 0;
51     sb->current_count = 0;
52     sb->total = total;
53  }
54
55  inline void spin_barrier_init_lsense(int *lsense)
56  {
```

```
57      *lsense = 0;
58  }
59
60
61  /*
62   * BARRIER (with local sense)
63   *
64   */
65  inline void spin_barrier_lsense(spin_barrier_t *sb, int *lsense)
66  {
67      *lsense = ~(*lsense);
68
69      spin_lock(&(sb->lock));
70      (sb->current_count)++;
71      if(sb->current_count == sb->total) {
72          sb->current_count = 0;
73          sb->release_flag = *lsense;
74      }
75      spin_unlock(&(sb->lock));
76      spin_on_condition(&(sb->release_flag), *lsense);
77  }
```

Βιβλιογραφία

[1] E. Athanasaki, N. Anastopoulos, K. Kourtis και N. Koziris. On prefetching and tlp performance limits for optimized memory intensive applications on smt processors. 2005.

[2] E. Athanasaki και N. Koziris. Fast indexing for blocked array layouts to improve multi-level cache locality. Στο *Proceedings of the 8-th Workshop on Interaction between Compilers and Computer Architectures, in conjuction with HPCA-10*, Madrid, Spain, Φεβρουάριος 2004.

[3] E. Athanasaki, N. Koziris και P. Tsanakas. A tile size selection analysis for blocked array layouts. Στο *Proceedings of the 9-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9), in conjunction with the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, σελίδες 70–80, San Francisco, CA, Φεβρουάριος 2005.

[4] D. Burger και T. Austin. The simplescalar tool set, version 2.0, 1997.

[5] S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra και M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. Στο *Proceedings of the 13th ACM International Conference in Supercomputing*, Ιούνιος 1999.

[6] M. Cierniak και W. Li. Unifying data and control transformations for distributed shared-memory machines. Στο *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, USA, Ιούνιος 1995.

[7] J. L. Hennessy και D. A. Patterson. *Computer Architecture. A Quantitative Approach, 3rd edition.* Morgan Kaufmann Publications, San Francisco, 2002.

[8] M. Jiminez. *Multilevel tiling for non-rectangular iteration spaces.* Διδακτορική Διατριβή , Universitat Politecnica de Catalunya, Μάιος 1999.

[9] M. Kandemir, J. Ramanujam και A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):159–167, Φεβρουαρίος 1999.

[10] M. S. Lam, E. E. Rotheberg και M. E. Wolf. The cache performance and optimizations of blocked algorithms. Στο *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, Μάιος 1991.

[11] J. L. Lo, S. J. Eggers, J. S. Emmer, H. M. Levy, R. L Stamm και D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(03):322–354, Αύγουστος 1997.

[12] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh και D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. Στο *IEEE Proceedings of Micro-30*, Research Triangle Park, North Carolina, Δεκέμβριος 1997.

[13] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller και M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 06(01):4–15, Φεβρουάριος 2002.

[14] K. S. McKinley, S. Carr και C. W Tseng. Improving data locality with loop tranformations. *ACM Transactions on Programming Languages and Systems*, 18(04):424–453, Ιούλιος 1996.

[15] M. F.P. O'Boyle και P. M.W. Knijneburg. Nonsingular data transformations: Definition, validity, and applications. *International Journal of Parallel Programming*, 27(3):131–159, 1999.

[16] Ε. Αθανασάκη. Μεθόδοι Βελτιστοποίησης για την Εκτέλεση Αλγορίθμων σε Αρχιτεκτονικές Υπολογιστών με Αξιοποίηση της Ιεραρχίας Μνήμης. Διπλωματική εργασία Master, Εθνικό Μετσόβιο Πολυτεχνείο, 2002.

[17] N. Tuck και D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. Στο *Proceedings of 12th Intel Conference on Parallel Architectures and Compilation Techniques*, Σεπτέμβριος 2003.

[18] M. E. Wolf και M. S. Lam. A data locality optimizing algorithm. Στο *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, σελίδες 30–44, Toronto, Ontario, Canada, Ιούνιος 1991.

Ευρετήριο

CPU affinity, 96
MBaLt, 54
block data layout, 53
compiler prefetching, 23
hardware prefetching, 23
thash, 26
thread, 22

αντικατάσταση ενότητας
  η πιο πρόσφατα χρησιμοποιούμενη, 20
  με βάση το χρόνο παραμονής, 20
  τυχαία επιλογή, 20
αντιστοιχηση δεδομένων, 47
αρχή της τοπικότητας της αναφοράς, 14, 17
  χρονική τοπικότητα, 17
  χωρική τοπικότητα, 17

δεικτοδότηση Morton, 55
διάνυσμα δεικτών, 46
διάνυσμα επανάληψης, 28
διάνυσμα εξάρτησης, 30
διάταξη ενοτήτων, 53
διεσταλμένοι ακέραιοι, 55

εύρος ζώνης, 18
εικονική μνήμη, 25
  αριθμός εικονικής σελίδας, 25
  προσωρινός καταχωρητής μετάφρασης (TLB), 26
    αστοχία TLB, 26
    επιτυχία TLB, 26
  σελίδα, 25
επαναχρησιμοποίηση, 32
  αυτο-χρονική, 33
  αυτο-χωρική, 33
  διάνυσμα επαναχρησιμοποίησης, 33
  ομαδο-χρονική, 33
  ομαδο-χωρική, 33
εξάρτηση δεδομένων
  γράφος εξάρτησης χώρου επαναλήψεων, 30
  λεξικογραφικά θετική, 30
εξαρτήσεις δεδομένων, 29
  ανάγνωση μετά από ανάγνωση, 30
  ανάγνωση μετά από εγγραφή, 29
  ανεξάρτητη βρόχων, 30
  αντι-εξάρτηση, 29
  βρόχος μεταφοράς εξάρτησης, 30
  εγγραφή μετά από ανάγνωση, 29
  εγγραφή μετά από εγγραφή, 30
  εξάρτηση εισόδου, 30
  εξάρτηση εξόδου, 30
  πραγματική εξάρτηση, 29

φωλιασμένοι βρόχοι, *βλέπε* σύστημα τέλειων φωλιασμένων βρόχων

ιεραρχία μνήμης, 14

καθυστέρηση απόκρισης, 18
κρυφή μνήμη, 14, 17
  compiler prefetching, 23
  hardware prefetching, 23
  αριθμός λέξης, 20
  αστοχία, 17
    χωρητικότητας, 22
    σύγκρουσης, 22
    υποχρεωτική, 21
  βελτιστοποίηση επίδοσης
    απομονωτής εγγραφής, 22
    χωρίς - μπλοκάρισμα, 22
  δείκτης, 20
  εγγραφή ενότητας
    δι-εγγραφή, 21
    εγγραφή με παραχώρηση, 21

εγγραφή χωρίς παραχώρηση, 21
επανεγγραφή, 21
επιτυχία, 17
ετικέτα, 19
ευθείας αντιστοίχησης, 18
θυμάτων, 23
μπιτ εγκυρότητας, 19
μπιτ τροποποίησης, 21
πλήρως συσχετιστική, 18
χωρίς - μπλοκάρισμα, 22
συσχέτισης συνόλου, 18

λεπτοκομμένη διαμέριση εργασίας, 91
λεξικογραφική διάταξη, 29

μετασχηματισμός δεδομένων, 47
μετασχηματισμοί βρόχων, 24
έγκυρος, 34
inner unrolling, 37
scalar replacement, 37
strip mining, 37
tiling, 24, 38
ανάθεση μεταβλητών σε καταχωρητές, 37
αναστροφή, 35
ανταλλαγή, 34
βύθισης κώδικα, 27
διάσπαση, 37
διάταξη φθίνουσας μνήμης, 114
μετάθεση, 34
περιστροφή, 35
πλήρως μεταθέσιμοι, 35
πλακόστρωση, 24, 38
συγχώνευση, 36
μετασχηματισμοί δεδομένων, 24
βλέπε διάταξη ενοτήτων 53
unimodular, 51
διάταξη Morton, 54
διεύρυνση, 51
εγκυρότητα, 52
ολίσθηση, 51
συμπίεση, 51
μηχανισμοί συγχρονισμού, 97
barriers, 97
spin locks, 97

νήμα, 96

πολυνημάτωση, 85
λεπτοκομμένη, 86
χοντροκομμένη, 86
ταυτόχρονη πολυνημάτωση *βλέπε* ταυτόχρονη πολυνημάτωση (SMT) 86

χώρος δεικτών, 46
χώρος επαναλήψεων, 28
χοντροκομμένη διαμέριση εργασίας, 91

σύστημα τέλειων φωλιασμένων βρόχων, 27

τέλεια φωλιασμένοι βρόχοι, *βλέπε* σύστημα τέλειων φωλιασμένων βρόχων
ταυτόχρονη πολυνημάτωση (SMT), 86
τοπικότητα, 32

υπερχείληση TLB, 26