
NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
COMPUTING SYSTEMS LABORATORY

# Non-linear memory layout transformations and data prefetching techniques to exploit locality of references for modern microprocessor architectures with multilayered memory hierarchies

PHD THESIS

**Evangelia G. Athanasaki**

Athens, Greece, July 2006

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΡΑΚΤΙΚΟ ΕΞΕΤΑΣΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

της

**Ευαγγελίας Γ. Αθανασάκη**

Διπλωματούχου Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών Ε.Μ.Π. (2002)

# Μη-Γραμμικοί Μετασχηματισμοί Αποθήκευσης και Τεχνικές Πρώιμης Ανάκλησης Δεδομένων για την Αξιοποίηση της Τοπικότητας Αναφοράς σε Σύγχρονες Αρχιτεκτονικές Μικροεπεξεργαστών με Πολυεπίπεδες Ιεραρχίες Μνημών

**Τριμελής Συμβουλευτική επιτροπή:** Παναγιώτης Τσανάκας, επιβλέπων
Γεώργιος Παπακωνσταντίνου
Νεκτάριος Κοζύρης

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την .....

| | | |
|---|---|---|
| ..... | ..... | ..... |
| Π. Τσανάκας | Γ. Παπακωνσταντίνου | Ν. Κοζύρης |
| Καθηγητής Ε.Μ.Π. | Καθηγητής Ε.Μ.Π. | Επίκ. Καθηγητής Ε.Μ.Π. |
| ..... | | ..... |
| Τ. Σελλής | | Α. Μπίλας |
| Καθηγητής Ε.Μ.Π. | | Αναπ. Καθηγητής, Πανεπ. Κρήτης |
| ..... | | ..... |
| Α. Σταφυλοπάτης | | Ν. Παπασπύρου |
| Καθηγητής Ε.Μ.Π. | | Λέκτορας Ε.Μ.Π. |

Αθήνα, Ιούλιος 2006

.....

Evangelia G. Athanasaki
School of Electrical and Computer Engineering, National Technical University of Athens, Greece

# Περίληψη

Ένα από τα βασικότερα ζητήματα που έχουν να αντιμετωπίσουν οι αρχιτέκτονες υπολογιστών και οι συγγραφείς μεταγλωττιστών είναι η συνεχώς αυξανόμενη διαφορά επίδοσης μεταξύ του επεξεργαστή και της κύριας μνήμης. Προκειμένου να ξεπεραστεί το πρόβλημα αυτό χρησιμοποιούνται διάφοροι μετασχηματισμοί κώδικα, με στόχο τη μείωση του αριθμού των αστοχιών μνήμης και επομένως τη μείωση του μέσου χρόνου που καθυστερούν οι εφαρμογές περιμένοντας τα δεδομένα να φτάσουν από την κύρια μνήμη. Ο μετασχηματισμός υπερκόμβων χρησιμοποιείται ευρέως στους κώδικες φωλιασμένων βρόχων, αναδιατάσσοντας τη σειρά εκτέλεσης των επαναλήψεων, για τη βελτίωση της τοπικότητας των αναφορών σε δεδομένα μνήμης.

Η διατριβή αυτή συμβάλλει στην περαιτέρω αξιοποίηση της τοπικότητας αναφοράς για ιεραρχίες μνημών, βελτιστοποιώντας τον τρόπο αναφοράς επαναληπτικών κωδίκων στη μνήμη. Επιχειρεί να ελαχιστοποιήσει τον αριθμό των αστοχιών μνήμης, αναδιατάσσοντας τη σειρά αποθήκευσης των δεδομένων των πολυδιάστατων πινάκων στη μνήμη, ώστε να ακολουθεί τη σειρά προσπέλασής τους από τον κώδικα φωλιασμένων βρόχων, αφότου έχει εφαρμοστεί σε αυτόν μετασχηματισμός υπερκόμβων. Στη μέθοδο μη-γραμμικής αποθήκευσης που αναπτύσεται, τα δεδομένα των πινάκων χωρίζονται σε ομάδες (blocks) οι οποίες ταυτίζονται με τα tiles του μετασχηματισμού υπερκόμβων, και αποθηκεύονται σε διαδοχικές θέσεις μνήμης. Με τον τρόπο αυτό, η ακολουθία των προσπελάσεων δεδομένων από τον κώδικα βρόχων ευθυγραμμίζεται με τη σειρά αποθήκευσής τους στη μνήμη. Η μετατροπή της πολυδιάστατης δεικτοδότησης των πινάκων στη δεικτοδότηση των μη-γραμμικών διατάξεων δεδομένων που προκύπτουν, γίνεται ταχύτατα με απλές δυαδικές πράξεις πάνω σε "αραιωμένους" ακεραίους. Τα πειραματικά αποτελέσματα και οι προσομοιώσεις αποδεικνύουν ότι η συνολική επίδοση βελτιώνεται σημαντικά, χάρη στη μείωση των αστοχιών μνήμης, όταν συνδυάζεται ο μετασχηματισμός υπερκόμβων με τις μη-γραμμικές διατάξεις δεδομένων σε ομάδες και την δυαδική δεικτοδότηση των δεδομένων.

Η συνέπεια της προτεινόμενης μεθόδου σε ότι αφορά τη συνολική επιτάχυνση του χρόνου εκτέλεσης εξαρτάται σε μεγάλο βαθμό από την επιλογή του κατάλληλου μεγέθους tile. Στην παρούσα διατριβή αναλύεται η διαδικασία παραγωγής των αστοχιών κρυφής μνήμης και TLB

στις μη-γραμμικές διατάξεις δεδομένων σε ομάδες. Σύμφωνα με τα αποτελέσματα της ανάλυσης αυτής, επιλέγεται το βέλτιστο μέγεθος tile, με στόχο τη μέγιστη αξιοποίηση του μεγέθους της L1 κρυφής μνήμης και ταυτόχρονα την αποτροπή των αστοχιών λόγω συγκρούσεων στην κρυφή μνήμη. Αποδεικνύεται ότι σε βελτιστοποιημένους κώδικες, που έχει εφαρμοστεί πρώιμη ανάκληση δεδομένων, ξεδίπλωμα βρόχων, ανάθεση μεταβλητών σε προσωρινούς καταχωρητές και ευθυγράμμιση των διαφορετικών πινάκων, τα tiles, με μέγεθος ίσο με την χωρητικότητα της L1 κρυφής μνήμης, αξιοποιούν την L1 κρυφή μνήμη κατά βέλτιστο τρόπο, ακόμα και στις περιπτώσεις που προσπελαύνονται περισσότεροι από ένας πίνακες δεδομένων. Το προτεινόμενο μέγεθος tile είναι συγκριτικά με άλλες μεθόδους μεγαλύτερο, δίνοντας επιπλέον το πλεονέκτημα του μειωμένου αριθμού αστοχιών λόγω των λανθασμένων προβλέψεων διακλάδωσης των φωλιασμένων βρόχων.

Τέλος, ένα άλλο θέμα που δεν είχε μέχρι στιγμής εξεταστεί στην εως τώρα βιβλιογραφία είναι η επίδοση μίας εφαρμογής, όταν αυτή εκτελείται μόνη της σε μηχανήματα που είναι εξοπλισμένα με την τεχνική της Ταυτόχρονης Πολυνημάτωσης. Η τεχνική αυτή είχε προταθεί για τη βελτίωση του ρυθμού εκτέλεσης εντολών μέσω της ταυτόχρονης εκτέλεσης εντολών από διαφορετικά νήματα (τα οποία προέρχονται είτε από διαφορετικές εφαρμογές ή από παραλληλοποιημένες εφαρμογές). Οι πρόσφατες μελέτες έχουν δείξει ότι η ανομοιογένεια των ταυτόχρονα εκτελούμενων νημάτων είναι από τους παράγοντες που επηρεάζουν θετικά την επίδοση των εφαρμογών. Ωστόσο, η επιτάχυνση των παράλληλων εφαρμογών εξαρτάται σε μεγάλο βαθμό από τους μηχανισμούς συγχρονισμού και επικονωνίας μεταξύ των διαφορετικών, αλλά πιθανότατα εξαρτώμενων μεταξύ τους, νημάτων. Επιπλέον, καθώς τα διαφορετικά νήματα των παράλληλων εφαρμογών έχουν στις περισσοτέρες περιπτώσεις όμοιου τύπου εντολές (π.χ. πράξεις κινητής υποδιαστολής, ακέραιες πράξεις, άλματα, κλπ), κατά την εκτέλεσή τους στους κοινούς πορους του συστήματος συνοστίζονται και σειριοποιούνται κατά τη διέλευσή τους μέσα από συγκεκριμένες λειτουργικές μονάδες, με αποτέλεσμα να μην μπορεί να επιτευχθεί σημαντική επιτάχυνση των εφαρμογών. Στη διατριβή αυτή αποτιμώνται και συγκρίνονται η πρώιμη ανάκληση δεδομένων και ο παραλληλισμός επιπέδου νήματος (thread-level parallelism - TLP). Τέλος εξετάζεται η επίδραση των μηχανισμών συγχρονισμού στις πολυνηματικές παράλληλες εφαρμογές που εκτελούνται από επεξεργαστές που διαθέτουν ταυτόχρονη πολυνημάτωση.

**Λέξεις-κλειδιά:** Ιεραρχία Μνήμης, Μετασχηματισμός Υπερκόμβων (Tiling), Μη-γραμμικοί μετασχηματισμοί δεδομένων, Δυαδικές μάσκες, Πρώιμη ανάκληση δεδομένων, Ξεδίπλωμα βρόχων, Επιλογή Μεγέθους Tile, Ταυτόχρονη Πολυνημάτωση, Φορτίο μίας μόνο Εφαρμογής, Απελευθέρωση Πόρων Συστήματος από τα άεργα Νήματα.

# Abstract

One of the key challenges computer architects and compiler writers are facing, is the increasing discrepancy between processor cycle times and main memory access times. To overcome this problem, program transformations that decrease cache misses are used, to reduce average latency for memory accesses. Tiling is a widely used loop iteration reordering technique for improving locality of references. Tiled codes modify the instruction stream to exploit cache locality for array accesses.

This thesis adds some intuition and some practical solutions to the well-studied memory hierarchy problem. We further reduce cache misses, by restructuring the memory layout of multi-dimensional arrays, that are accessed by tiled instruction code. In our method, array elements are stored in a blocked way, exactly as they are swept by the tiled instruction stream. We present a straightforward way to easily translate multi-dimensional indexing of arrays into their blocked memory layout using simple binary-mask operations. Indices for such array layouts are now easily calculated based on the algebra of dilated integers, similarly to morton-order indexing. Actual experimental and simulation results illustrate that execution time is greatly improved when combining tiled code with tiled array layouts and binary mask-based index translation functions.

The stability of the achieved performance improvements are heavily dependent on the appropriate selection of tile sizes, taking into account the actual layout of the arrays in memory. This thesis provides a theoretical analysis for the cache and TLB performance of blocked data layouts. According to this analysis, the optimal tile size that maximizes L1 cache utilization, should completely fit in the L1 cache, to avoid any interference misses. We prove that when applying optimization techniques, such as register assignment, array alignment, prefetching and loop unrolling, tile sizes equal to L1 capacity offer better cache utilization, even for loop bodies that access more than just one array. Increased self- or/and cross-interference misses are now tolerated through prefetching. Such larger tiles also reduce lost CPU cycles due to less mispredicted branches.

Another issue, that had not been thoroughly examined so far, is Simultaneous Multithreading (SMT) for single-program workloads. SMT has been proposed to improve system throughput by overlapping multiple (either multi-programmed or explicitly parallel) threads on a single wide-issue processor. Recent studies have demonstrated that heterogeneity of simultaneously executed applications can bring up significant performance gains due to SMT. However, the speedup of a single application that is parallelized into multiple threads, is often sensitive to the efficiency of synchronization and communication mechanisms between its separate, but possibly dependent, threads. Moreover, as these separate threads tend to put pressure on the same architectural resources, no significant speedup can be achieved. This thesis evaluates and contrasts software prefetching and thread-level parallelism (TLP) techniques. It also examines the effect of thread synchronization mechanisms on multithreaded parallel applications that are executed on a single SMT processor.

**Keywords:** Memory Hierarchy, Loop tiling, Blocked Array Layouts, Binary Masks, Prefetching, Loop Unrolling, Tile Size Selection, Simultaneous Multithreading, Single Program Workload, Resource Release by idle Threads.

# Contents

# List of Figures

# List of Tables

# Αντί Προλόγου

Η παρούσα διδακτορική διατριβή εκπονήθηκε στον Τομέα Τεχνολογίας Πληροφορικής και Υπολογιστών, της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, του Εθνικού Μετσόβιου Πολυτεχνείου. Περιλαμβάνει την έρευνα και τα συμπεράσματα που προέκυψαν κατά τη διάρκεια των μεταπτυχιακών σπουδών μου στο Εργαστήριο Υπολογιστικών Συστημάτων της σχολής αυτής. Η διατριβή αποτελείται από δύο μέρη: Το πρώτο είναι γραμμένο στα αγγλικά, προκειμένου να μπορεί να διαβαστεί από την ακαδημαϊκή κοινότητα εκτός Ελλάδας. Το δεύτερο μέρος αποτελεί περιληπτική μετάφραση του πρώτου στα ελληνικά.

Στις ακόλουθες γραμμές θα ήθελα να εκφράσω τις θερμές ευχαριστίες μου σε όλους εκείνους που με βοήθησαν είτε άμεσα με την καθοδήγηση που μου προσέφεραν, είτε έμμεσα με την ηθική στήριξή τους και συνέβαλαν ουσιαστικά στην ολοκλήρωση της παρούσας διατριβής. Πρώτον από όλους θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Παναγιώτη Τσανάκα, καθώς εκείνος με μύησε στον κόσμο της Αρχιτεκτονικής Υπολογιστών, ως προπτυχιακή φοιτήτρια, και μου μετέδωσε την αγάπη του για το αντικείμενο, το οποίο επέλεξα στη συνέχεια να διερευνήσω κατά τη διάρκεια των μεταπτυχιακών σπουδών μου. Πρόκειται για τον άνθρωπο του οποίου η ήρεμη δύναμη, η ευγένεια και η ηθική έδινε στις δύσκολες καταστάσεις την ισορροπία και τη διέξοδο. Τον ευχαριστώ ιδιαίτερα για την εμπιστοσύνη με την οποία με περιέβαλε και την αυτοπεποίθηση που μου μετέδιδε.

Στη συνέχεια θα ήθελα να ευχαριστήσω το δεύτερο μέλος της συμβουλευτικής επιτροπής μου, τον καθηγητή Γεώργιο Παπακωνσταντίνου, ο οποίος ως κεφαλή του εργαστηρίου μου έδωσε τη δυνατότητα να βρίσκομαι σε ένα καλά δομημένο και άρτια οργανωμένο χώρο. Είναι ο άνθρωπος που με την τεχνογνωσία και την περιρρέουσα γνώση που μου εξασφάλισε, αποτέλεσε σημείο αναφοράς στο επιστημονικό του πεδίο. Τον ευχαριστώ θερμά, γιατί με την ακεραιότητα των προθέσεων και των ενεργειών του, αποτελούσε πάντα πόλο έλξης αντίστοιχα ακέραιων προπτυχιακών και μεταπτυχιακών φοιτητών και πολύτιμων συνεργατών.

Επιπλέον, νιώθω το χρέος να εκφράσω τις ειλικρινείς ευχαριστίες μου στο τρίτο μέλος της συμβουλευτικής επιτροπής μου, τον Επίκουρο καθηγητή Νεκτάριο Κοζύρη. Αισθάνομαι ιδιαίτερα τυχερή που στα τελευταία έτη των προπτυχιακών σπουδών μου είχα την ευκαιρία να ακούσω τις διαλέξεις του και αργότερα, κατά την εκπόνηση της διδακτορικής μου διατριβής, να συνεργαστώ μαζί του. Με το προσωπικό παράδειγμα της εργατικότητας, της μεθοδικότητας και της αδιάλειπτης ανανέωσης των γνώσεών του, αποτελούσε πάντοτε, για όλα τα μέλη του εργαστηρίου, κινητήρια

δύναμη προς τη συνεχή αναζήτηση ερεθισμάτων, που οδήγησαν το εργαστήριο στην αιχμή της τεχνολογίας. Ήταν ο άνθρωπος που διεύρυνε τους ορίζοντές μου, επιστημονικούς και κοινωνικούς, με τις παροτρύνσεις του να επισκεφθώ χώρες του εξωτερικού με κορυφαία πανεπιστήμια, να έρθω σε επαφή με τους πλέον αναγνωρισμένους επιστήμονες στο πεδίο της επιστήμης μου και με άλλους υποψήφιους διδάκτορες, που διέθεταν ανέλπιστο ζήλο για την έρευνά τους. Τον ευχαριστώ πραγματικά για το χρόνο που διέθεσε για να παρακολουθεί την ερευνητική μου πορεία, αλλά και για τις ώρες που αφιέρωσε παρευρισκόμενος στις συλλογικές συζητήσεις του εργαστηρίου.

Θα ήταν παράλειψη να μην ευχαριστήσω τα υπόλοιπα μέλη του εργαστηρίου, με πρώτους τον Νίκο Αναστόπουλο και τον Κορνήλιο Κούρτη, που η συνεργασία μαζί τους αποδείχτηκε ιδιαίτερα δημιουργική. Τα αποτελέσματα της παρούσας διατριβής θα ήταν σίγουρα φτωχότερα χωρίς την ουσιαστική συμβολή τους. Αν και νεώτερα μέλη του εργαστηρίου, η βαθιά γνώση του αντικειμένου που καλλιέργησαν από πολύ νωρίς, η εφευρετικότητα και η διερευνητική τους διάθεση, σε συνδυασμό με την αθόρυβη εργατικότητά τους, τους κατέστησε για μένα αναντικατάστατους συνοδοιπόρους. Εύχομαι το μέλλον να ανταμείψει τους κόπους τους και να ικανοποιήσει τα όνειρά τους.

Ακόμη, θα ήθελα να ευχαριστήσω τον Γιώργο Γκούμα, τον Άρη Σωτηρόπουλο, το Νίκο Δροσινό και την αδελφή μου, Μαρία Αθανασάκη. Αισθάνομαι ότι χωρίς τις γνώσεις και την παρουσία τους το εργαστήριο θα ήταν πολύ άδειο από τεχνογνωσία και ωριμότητα, αλλά και από ζωή, νέες ιδέες και αντίλογο, ψυχισμό: όλα εκείνα τα στοιχεία που σε τραβούν να περάσεις αμέτρητες ώρες μέσα στους τέσσερις τοίχους του εργαστηριακού περιβάλλοντος, νιώθοντας ότι ζεις αληθινή ζωή. Δεν θα μπορούσα να μην κάνω ιδιαίτερη αναφορά στην αδελφή μου, γιατί με έμαθε να μοιράζομαι, να αγαπάω, μου έδινε και μου δίνει μαθήματα προνοητικότητας, αποτελεί για μένα τον οδηγό προς ανώτερους στόχους, συμπληρώνει τις αδυναμίες μου, με κάνει να νιώθω πιο δυνατή και μόνο που είναι δίπλα μου. Τέλος, θα ήθελα να αναφέρω ξεχωριστά τον Βαγγέλη Κούκη, τον Αντώνη Χαζάπη, τον Αντώνη Ζήσιμο, το Γιώργο Τσουκαλά, το Γιώργο Βερυγάκη και όλα τα υπόλοιπα νέα ή παλιότερα μέλη του εργαστηρίου, που με την ενεργητικότητα, τις γνώσεις και το χαρακτήρα τους χτίζουν την κουλτούρα του Εργαστηρίου Υπολογιστικών Συστημάτων.

Τέλος, ευχαριστώ θερμά το Κοινωφελές Ίδρυμα Αλέξανδρος Ωνάσης για την οικονομική στήριξη που μου παρείχε μέσω μίας υποτροφίας μεταπτυχιακών σπουδών.

Η εργασία αυτή αφιερώνεται στην οικογένειά μου, σε όλους εκείνους που κατέχουν μία ξεχωριστή θέση στη ζωή μου.

*Αθήνα, Ιούλιος 2006*

*Ευαγγελία Αθανασάκη*

CHAPTER 1

# Introduction

Due to the demand for memory space ever augmenting and the speed of microprocessors improving continuously at high rates, memory technology could not keep up with programmers' needs. Large storage memories cannot respond as fast as smaller RAM modules. To keep processor performance unhampered, engineers addressed the insertion of small but fast memories between the main memory and the processor. This is the cache memory and, theoretically, allows the microprocessor to execute at full speed. However, when the program execution leads to instructions or data not in the cache, they need to be fetched from the main memory and the microprocessor has to stall execution.

As a result, the problem persists and researchers indicate that unless memory bandwidth and latency improve extremely in future machines, we are going to face the memory wall [WM95]. This thesis describes a software approach of bridging the widening gap between processors and main memory, exploiting the hardware characteristics of memory structure. The rest of this chapter provides a detailed motivation behind this thesis and introduces the goals that have been achieved. Finally, we list the contributions of this work and present a road map for the remainder of the thesis.

## 1.1 Motivation

The performance gap between memory latency and processor speed is constantly dilating at the rate of almost 50% per year [PJ03]. Thus, many types of applications still waste much of their execution time, waiting for data to be fetched from main memory. To alleviate this problem, multi-level memory hierarchies are used and compiler transformations are introduced to increase locality of references for iterative instruction streams.

Memory is organized into several hierarchical levels. Main memory serves as a cache for the virtual memory swapfile stored on the hard disk. Level 2 (L2) cache lies between the actual processor die and main memory, and all processors have a fast L1 cache integrated into

the processor die itself. The idea of cache has grown along with the size and complexity of microprocessor chips. A current high-end microprocessor can have 2 MB of cache (L2) built into the chip. This is more than twice the entire memory address space of the original 8088 chip used in the first PC and clones. The cache hierarchy makes sense because each level, from registers down to the hard disk, is an order of magnitude less expensive than the previous, and thus each level can be an order of magnitude larger than the previous.

The cache design counts on the fundamental principles of locality: both Temporal and Spatial Locality. Considering that the principles behind cache are successfully applied, data storage systems in modern computers can theoretically approach the speed of the register file while having the size of a hard disk. The effectiveness of memory hierarchy depends not only on the hardware design, but also on the application code, the one generated by the compiler. Of course, the code of real applications can not have perfect locality of references. Most numerical applications have poor performance, as they operate on large data sets that do not fit in any cache level. However, data accesses form regular patterns that favor locality. If regularity of references is appropriately exploited, accesses to main memory can be eliminated. For this reason, an important amount of research has been devoted to code optimization, applying different kinds of transformation in order to make data transfers from main memory to cache be a close approximation of cold start (compulsory) misses.

Nested loops are usually the most time and memory consuming parts of a program, and are commonly found in scientific applications, such as image processing, computational fluid dynamics, geophysical data analysis, and computer vision. Optimizing execution of such critical nested loops, is equivalent in essence with whole program optimization. However, manual optimization is generally a bad practice, as the resultant code is difficult to debug or proof read and, quite often, too complicated to be handled by anybody else except for the prime programmer. Even worse, it is not flexible and easily adaptable for machines with different architectural characteristics.

We could conclude that code optimizations have to be the result of an automatic or semi-automatic process. On the other hand, machine-dependent hand-optimizations can reach the highest performance limits. So, even an automatic optimization tool needs to take feedback about architecture specifications and the code profile.

The goal of this thesis is to develop a code optimization technique that exploits locality of references and memory hierarchy to gain performance on complex numerical applications. We focus on restructuring data memory layouts to minimize stall cycles due to cache misses. In order to facilitate the automatic generation of tiled code that accesses nonlinear array layouts, we propose an address calculation method of the array indices. We adopted some kind of blocked layout and dilated integer indexing similar to Morton-order arrays. Thus, we combine data locality, due to blocked layouts, with efficient element access, as simple boolean operations are only used to find the location of the right array element. Since data are now stored block-wise,

we provide the instruction stream with a straightforward indexing to access the correct elements. Our method is very effective at reducing cache misses, since the deployment of the array data in memory follows the exact order of accesses by the tiled instruction code, achieved at no extra runtime cost.

### 1.1.1 Compiler Optimizations

In order to increase the percentage of memory accesses satisfied by caches, loop transformations are used to modify the instruction stream structure in favor of locality of references. Loop permutation, loop reversal and loop skewing attempt to modify the data access order to improve data locality. Loop unrolling and software pipelining are exploiting registers and pipelined datapath to improve temporal locality [Jim99]. Loop fusion and loop distribution can indirectly improve reuse by enabling control transformations that were previously not legal [MCT96]. Loop tiling and data shackling [KPCM99] although they handle the issue of locality from a different point of view, both restructure the control flow of the program, decreasing the working set size, to exploit temporal and spatial reuse of accessed arrays. Unimodular control transformations described in the most cited work of Wolf and Lam [WL91] and compound transformations of McKinley et al in [MCT96], attempt to find the best combination of such transformations which, when used with tiling, ensure the correct computation order, while increasing locality of accesses to cache memory.

Using control transformations, we change data access order but not the data storage order. In a data-centric approach [Kan01], Kandemir proposes array unification, a compiler technique that maps multiple arrays into a single array data space, in an interleaved way, grouping together arrays accessed in successive iterations. Each group is transformed to improve spatial locality and reduce the number of conflict misses. Rivera and Tseng in [RT98a] use padding to eliminate severe conflict misses. Inter-variable padding adjusts variable base addresses, while intra-variable padding modifies array dimension sizes. For different cache and problem sizes the conflict distances between different array columns for linear algebra codes are calculated through the Euclidean gcd algorithm.

Sometimes, increasing the locality of references for a group of arrays may affect the number of cache hits for the other referenced arrays. Combined loop and data transformations were proposed in [CL95] and [KRC99]. Cierniak and Li in [CL95] presented a cache locality optimization algorithm, which combines both loop (control) and linear array layout transformations; but to lessen its complexity extra constraints are required to be defined. Another unified, systematic algorithm, is the one presented by Kandemir et al in [KCS+99], [KRC97], [KRCB01], [KRC99], which aims at utilizing spatial reuse in order to obtain good locality.

All previous approaches assumed linear array layouts. Programming languages provide with multidimensional arrays which are finally stored in a linear memory layout, either column-wise or row-wise (canonical order). Nevertheless, the linear array memory layouts do not match

the instruction stream access pattern. Since tiled code focuses on a sub-block of an array, why not put these array elements in contiguous memory locations? In other words, since the instruction stream and consequently the respective data access patterns are blocked ones, it would be ideal to store arrays following exactly the same pattern, so that instruction and data streams are aligned. Conflict misses, especially for direct mapped or small associativity caches are considerably reduced, since now all array elements within the same block are mapped in contiguous cache locations and self interference is avoided.

### 1.1.2 Non-linear Memory Layouts

Chatterjee et al [CJL+99], [CLPT99] explored the merit of non-linear memory layouts and quantified their implementation cost. They proposed two families of blocked layout functions, both of which split array elements up to a tile size, which fits to the cache characteristics and elements inside each tile are linearly stored. Although they claim for increasing execution-time performance, using four-dimensional arrays, as proposed, any advantage obtained by data locality due to blocked layouts seems to be counterbalanced by the slowdown caused by referring to four-dimensional array elements (in comparison to response time of two- or one-dimensional arrays). Even if we convert these four-dimensional arrays to two-dimensional ones, as proposed by Lin et al in [LLC02], indexing the right array elements naively, requires expensive array subscripts. Furthermore, blocked layouts in combination with level-order, Ahnentafel and Morton indexing were used by Wise et al in [WAFG01] and [WF99]. Although their quad-tree based layouts seems to work well with recursive algorithms, due to efficient element indexing, no locality gain can be obtained at non-recursive codes. Especially, since they use recursion down to the level of a single array element, extra loss of performance is induced.

Non-linear layouts were proved to favor cache locality in all levels of memory hierarchy, including L1, L2 caches and TLBs. Experiments in [RT99b] that exploit locality in the L2 cache, rather than considering only one level caches, demonstrate reduced cache misses, but performance improvements are rarely significant. Targeting the L1 cache, nearly all locality benefits can be achieved. Most of the previous approaches target mainly the cache performance. As problem sizes become larger, TLB performance becomes more significant. If TLB thrashing occurs, [PHP02], the overall performance will be drastically degraded. Hence TLB and cache must be considered in concert while optimizing application performance. Park et al in [PHP03] derive a lower bound on TLB performance for standard matrix access patterns and show that block data layouts and Morton Layout (for recursive codes) achieve this bound. These layouts with block size equal to the page size, minimize the number of TLB misses. Considering both all levels of cache and TLB, a block size selection algorithm should be used to calculate a tight range of optimal block sizes.

However, the automatic application of non-linear layouts in real compilers is a really tedious task. It does not suffice to identify the optimal non-linear (blocked) layout for a specific array,

we also need to automatically generate the mapping from the multidimensional iteration indices to the correct location of the respective data element in linear memory. Blocked layouts are very promising subject to an efficient address computation method. In the following, when referring to our non-linear layouts, we will name them *Blocked Array Layouts*, as they are always combined with loop tiling (they split array elements to blocks) and apply efficient indexing to the derived tiles.

### 1.1.3 Tile Size/Shape Selection

Early efforts [MCT96], [WL91] have been dedicated to selecting the tile in such a way that its working set fits in the cache, so as to eliminate capacity misses. To minimize loop overhead, the tile size should be the maximum that meets the above requirement. Recent work takes conflict misses into account, as well. Conflict misses [TFJ94] may occur when too many data items map to the same set of cache locations, causing cache lines to be flushed from cache before they may be used, despite sufficient capacity in the overall cache. As a result, in addition to eliminating capacity misses [MCT96], [WL91] and maximizing cache utilization, the tile should be selected in such a way that there are no (or few) self conflict misses, while cross conflict misses are minimized [CM99], [CM95], [Ess93], [LRW91], [RT99a].

To find tile sizes that have few capacity misses, the surveyed algorithms restrict their candidate tile sizes to be the ones whose working set can entirely fit in the cache. To model self conflict misses due to low associativity cache, [WMC96] and [MHCF98] use the effective cache size $q \times C$ ($q < 1$), instead of the actual cache size $C$, while [CM99], [CM95], [LRW91] and [SL01] explicitly find the non-conflicting tile sizes. Taking into account cache line size as well, column dimensions (without loss of generality, assume a column major data array layout) should be a multiple of the cache line size [CM95]. If fixed blocks are chosen, Lam et al. in [LRW91] have found that the best square tile is not larger than $\sqrt{\frac{aC}{a+1}}$, where $a$ = associativity. In practice, the optimal choice may occupy only a small fraction of the cache, typically less than 10%. What's more, the fraction of the cache used for optimal block size decreases as the cache size increases.

The desired tile shape has been explicitly specified in algorithms such as [Ess93], [CM99], [CM95], [WL91], [WMC96], [LRW91]. Both [WL91] and [LRW91] search for square tiles. In contrast, [CM99], [CM95] and [WMC96] find rectangular tiles or [Ess93] even extremely tall tiles (the maximum number of complete columns that fit in the cache). Tile shape and cache utilization are two important performance factors considered by many algorithms, either implicitly through the cost model or explicitly through candidate tiles. However, extremely wide tiles may introduce TLB thrashing. On the other hand, extremely tall or square tiles may have low cache utilization. Apart from the static techniques, iteration compilation has been implemented in [KKO00]. Although it can achieve high speedups, the obvious drawback of iterative compilation is its long compilation time, required to generate and profile many versions of the source program.

Unfortunately, the performance of a tiled program resulting from existing tiling heuristics does not have robust performance [PNDN99], [RT99a]. Instability comes from the so-called pathological array sizes, when array dimensions are near powers of two, since cache interference is a particular risk at that point. Array padding [HK04], [PNDN99], [RT98b], [SL01] is a compiler optimization that increases the array sizes and changes initial locations to avoid pathological cases. It introduces space overhead but effectively stabilizes program performance. Cache utilization for padded benchmark codes is much higher overall, since padding is used to avoid small tiles [RT99a]. As a result, more recent research efforts have investigated the combination of both loop tiling and array padding in the hope that both magnitude and stability of performance improvements of tiled programs can be achieved at the same time [HK04], [RT98b], [PNDN99]. An alternative method for avoiding conflict misses is to copy tiles to a buffer and modify code to use data directly from the buffer [Ess93], [LRW91], [TGJ93]. Since data in the buffer is contiguous, self-interference is eliminated. If buffers are adjacent, then cross-interference misses are also avoided. Copying in [LRW91] can take full advantage of the cache as it enables the use of tiles of size $\sqrt{C} \times \sqrt{C}$ in each blocked loop nest. However, performance overhead due to runtime copying is low if tiles only need to be copied once.

TLB thrashing is a crucial performance factor since a TLB miss costs more than a L1 cache miss and can cause severe cache stalls. While one-level cost functions suffices to yield good performance at a single level of memory hierarchy, it may not be globally optimal. Multi-level tiling can be used to achieve locality in multiple levels (registers, caches and TLBs) simultaneously. Such optimizations have been considered in [CM95], [HK04] and a version of [MHCF98], guided by multi-level cost functions. Minimizing a multi-level cost function balances the relative costs of TLB and cache misses. Optimal tiling must satisfy the capacity requirements of both TLB and cache.

In general, most algorithms search for the largest tile sizes that generate the least amount of capacity misses, eliminate self-conflict misses and minimize cross-conflict misses. Sometimes, high cache utilization [SL99] and low cache misses may not be achieved simultaneously. As a result, each algorithm has a different approximation to cache utilization and number of cache misses, and weigh between these two quantities.

Significant work has been done to quantify the total number of conflict misses [CM99], [FST91], [GMM99], [HKN99], [Ver03]. Cache behaviour is extremely difficult to analyze, reflecting its unstable nature, in which small modifications can lead to disproportionate changes in cache miss ratio [TFJ94]. Traditionally, cache performance evaluation has mostly used simulation. Although the results are accurate, the time needed to obtain them is typically many times greater than the total execution time of the program being simulated. To try to overcome such problems, analytical models of cache behaviour combined with heuristics have also been developed, to guide optimizing compilers [GMM99], [RT98b] and [WL91], or study the cache performance of particular types of algorithm, especially blocked ones [CM99], [HKN99],

[LRW91], and [Ver03]. Code optimizations, such as tile size selection, selected with the help of predicted miss ratios require a really accurate assessment of program's code behaviour. Performance degradation, due to tiled code complexity and miss-predicted branches, should also be taken into account. Miss ratios of blocked kernels are generally a lot smaller than these of unblocked kernels, amplifying the significance of small errors in prediction. For this reason, a combination of cache miss analysis, simulation and experimentation is the best solution for optimal selection of critical transformations.

The previous approaches assumed linear array layouts. However, as aforementioned studies have shown, such linear array memory layouts produce unfavorable memory access patterns, that cause interference misses and increase memory system overhead. In order to quantify the benefits of adopting nonlinear layouts to reduce cache misses, there exist several different approaches. In [RT99b], Rivera et al. considers all levels of memory hierarchy to reduce L2 cache misses as well, rather than reducing only L1 ones. He presents even fewer overall misses, however performance improvements are rarely significant. In another approach, TLB and cache misses should be considered in concert. Park et al. in [PHP02] analyze the TLB and cache performance for standard matrix access patterns, when tiling is used together with block data layouts. Such layouts with block size equal to the page size, seem to minimize the number of TLB misses. Considering both all levels of cache (L1 and L2) and TLB, a block size selection algorithm calculates a range of optimal block sizes.

## 1.2 Contributions

A detailed model of cache behaviour can give accurate information to compilers or programmers to optimize codes. However, this is a really demanding task, especially in respect of giving feedback to guide code transformations. This thesis offers some advance in automation of code optimization, focusing on the application of non-linear layouts in numerical codes. The optimization algorithm takes into account cache parameters, in order to determine best processing sizes that match the memory hierarchy characteristics of each specific platform.

The primary contributions of this thesis are:

- The proposal of a fast indexing scheme that makes the performance of blocked data layouts efficient. We succeed in increasing the effectiveness of such layouts when applied to complex numerical codes, in combination with loop tiling transformation. The provided framework can be integrated in a static tool, like compiler optimizations.

- The proposal of a simple heuristic to make one-level tiling size decisions easy. It outlines the convergence point of factors that affect or determine the performance of the multiple hierarchical memory levels.

- A study of the effects of multiple code optimization methods in different machines. This study matches specific optimizations with hardware characteristics, tracking down weaknesses that can be healed when pipeline is being fed by an appropriate stream of instructions.

## 1.3 Thesis Overview

This work is organized as follows:

Chapter 2 gives some background about memory hierarchy organization and basic cache memory characteristics. We present the terminology of code optimization techniques used in this work. Finally, we provide an introduction about loop tiling and transformations.

Chapter 3 presents the underlying model on which this work is based, that is pre-existing non-linear array layouts. Then, the proposed blocked array layouts are analytically described, combined with a fast indexing scheme. A system of binary masks has been developed, which is used to pick up data elements of multidimensional arrays in the one dimensional storage order.

Chapter 4 analyzes the cache behavior when tiled codes with blocked array layouts are being executed. The best tile size that utilizes the cache capacity and deters conflict misses is being calculated as the convergence point of all factors (cache misses and latency of multiple cache levels and TLBs, and branch miss penalty) that affect cache performance.

Chapter 5 outlines the framework of Simultaneous Multithreading (SMT) micro-architecture and discusses implementation and synchronization issues. It also explores the performance limits of simultaneously executed instruction streams, using homogeneous instruction streams, executed in parallel.

Chapter 6 presents experimental and simulation results on various platforms, evaluating blocked array layouts versus other non-linear array layouts, the tile selection technique and the performance of Simultaneous Multithreading on optimized codes.

Finally, chapter 7 gives the conclusion remarks.

## 1.4 Publications

**INTERNATIONAL CONFERENCES**

- Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris. Exploring the Capacity of a Modern SMT Architecture to Deliver High Scientific Application Performance. In *Proc. of the 2006 International Conference on High Performance Computing and Communications (HPCC-06)*, pages , Munich, Germany, Sep 2006. Lecture Notes in Computer Science.

- Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris. Exploring the Performance Limits of Simultaneous Multithreading for Scientific Codes. In *Proc. of the 2006 International Conference on Parallel Processsing (ICPP-06)*, pages , Columbus, OH, Aug 2006. IEEE Computer Society Press.

- Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris. Tuning Blocked Array Layouts to Exploit Memory Hierarchy in SMT Architectures. In *Proc. of the 10th Panhellenic Conference in Informatics*, pages , Volos, Greece, Nov 2005. Lecture Notes in Computer Science.

- Evangelia Athanasaki, Nectarios Koziris and Panayiotis Tsanakas. A Tile Size Selection Analysis for Blocked Array Layouts. In *Proc. of the 9-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9), In conjuction with the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 70–80, San Francisco, CA, Feb 2005. IEEE Computer Society Press.

- Evangelia Athanasaki and Nectarios Koziris. Fast Indexing for Blocked Array Layouts to Improve Multi-Level Cache Locality. In *Proc. of the 8-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-8), In conjuction with the 10th International Symposium on High-Performance Computer Architecture (HPCA-10)*, pages 109–119, Madrid, Spain, Feb 2004. IEEE Computer Society Press.

- Evangelia Athanasaki and Nectarios Koziris. Improving Cache Locality with Blocked Array Layouts. In *Proceedings of the 12-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP'04)*, pages 308–317, A Coruna, Spain, Feb. 2004. IEEE Computer Society Press.

- Evangelia Athanasaki and Nectarios Koziris. Blocked Array Layouts for Multilevel Memory Hierarchies. In *Proceedings of the 9th Panhellenic Conference in Informatics*, pages 193–207, Thessaloniki, Greece, Nov. 2003.

**INTERNATIONAL JOURNALS**

- Evangelia Athanasaki and Nectarios Koziris. Fast Indexing for Blocked Array Layouts to Reduce Cache Misses. *International Journal of High Performance Computing and Networking (IJHPCN)*, 3(5/6): 417–433, 2005.

- Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris. Exploring the Performance Limits of Simultaneous Multithreading for Memory Intensive Applications. *The Journal of Supercomputing*, submitted.

CHAPTER 2

# Basic Concepts

This chapter provides a background about memory hierarchy issues: basic characteristics and organization. We, also, provide the definitions for data locality and reuse, the fundamental principles of cache design. Finally, we present an introduction to loop tiling and other loop and data transformations.

## 2.1 Memory Hierarchy

The memory system is organized hierarchically. One end is disk storage, with high storage density, low structural cost per bit, and a relative low access rate. On the other end of hierarchy, is the register file, which has low storage capacity, high structural cost and low latency. Two important levels of the memory hierarchy are the cache and virtual memory.

Figure 2.1 illustrates the levels in a typical memory hierarchy. Drawing farther away from the CPU, the memory levels become larger, slower and more expensive.

To weight up the effectiveness of the memory hierarchy, the following formula should be evaluated:

$$\text{Memory stall cycles} = \text{IC} \times \text{Mem\_Refs} \times \text{Miss\_Rate} \times \text{Miss\_Penalty}$$

| | | |
|---|---|---|
| where | IC | = Instruction count |
| | Mem_Refs | = Average Memory References per Instruction |
| | Miss_Rate | = the fraction of accesses that do not hit in the cache |
| | Miss_Penalty | = the time needed to service a miss |

*Cache* memory is a special high-speed storage mechanism made of static random access memory (SRAM) instead of the slower and cheaper dynamic RAM (DRAM) used for main memory. Cache memories are placed hierarhically between the CPU and the main memory. A current computing system contains more than just one level of cache, described in levels of

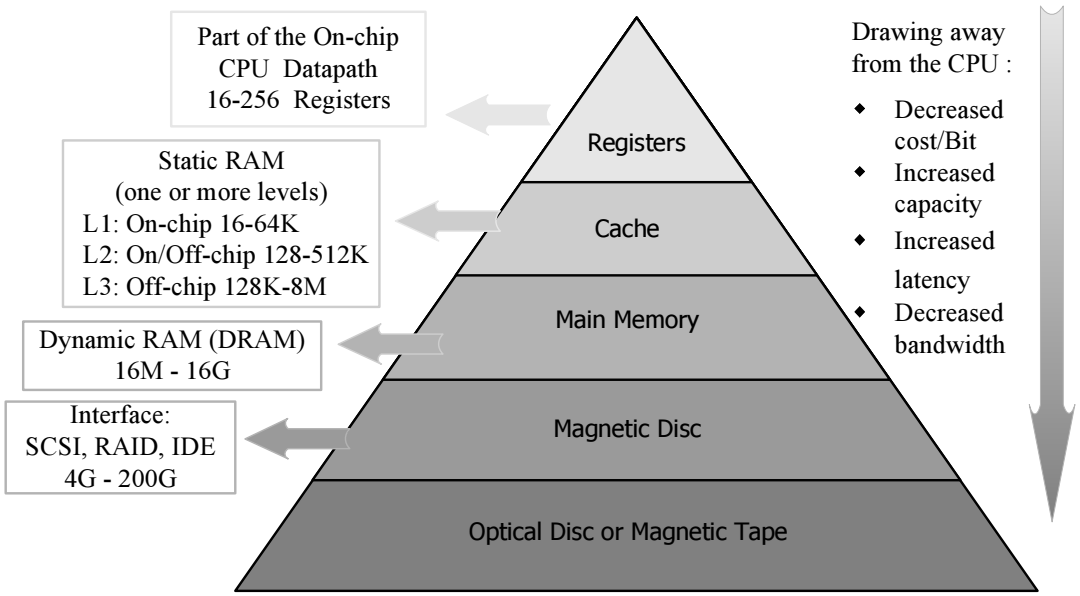



**Figure 2.1:** The Memory Hierarchy pyramid

proximity and accessibility to the microprocessor. An L1 (level 1) cache is usually built into the processor chip, while L2 (level 2) is usually a separate static RAM (SRAM) chip. As the microprocessor processes data, if the requested information are not found in the register file, it looks first in the cache memory. Caches contain only a small portion of the information stored in main memory. Given that cache latency is much lower than that of other memory levels, we can have a significant performance improvement if a previous reading of data has brought the requested data in the cache. This way, there is no need for the time-consuming access of remote memory levels. The above principles suggest that we should try to preserve recently accessed and frequently used instructions and data in the fastest memory level.

Programs tend to reuse data and instructions they have used recently: "A program spends 90% of its time in 10% of its code". Thus, we have to exploit data *locality*, in order to improve performance. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.

Two different types of locality have been observed:

- *Temporal locality*: states that recently accessed items are likely to be accessed in the near future. If accessed words are placed in the cache, waste of time (due to memory latency) will be averted when they are reused.

- *Spatial locality*: states that items whose addresses are near one another tend to be referenced close together in time. As a result, it is advantageous to organize memory words in blocks so that whenever there is access to a word, the whole block will be transferred in the cache, not just one word.

During a memory reference, a request for specific data can be satisfied from the cache without using the main memory, when the requested memory address is present in cache. This situation is known as a cache hit. The opposite situation, when the cache is consulted and found not to contain the desired data, is known as a cache miss. In the latter case, the block of data (where the requested element belongs), is usually inserted into the cache, ready for the next access. This block is the smaller amount of elements that can be transferred from one memory level to another. This block of data is alternatively called cache line.

Memory latency and bandwidth are the two key factors that determine the time needed to resolve a cache miss. Memory latency specifies the intermediate time between the data request to main memory and the arrival into the cache of the first data element in the requested block. Memory bandwidth specifies the arrival rate of the remaining elements in the requested block. A cache miss resolution is critical, because the processor has to stall until the requested data arrive from main memory (especially in an in-order execution).

## 2.2 Cache misses

Cache memories were designed to keep the most recently used piece of content (either a program instruction or data). However, it is not feasible to satisfy all data requests. In a case of a cache miss in instruction cache, the processor stall is resolved when the requested instruction is fetched from main memory. A cache read miss (data load instruction) can be less severe as there can be other instructions not dependant on the expected data element. Execution is continued until the operation which really needs the loaded data is ready to be executed. However, data is often used immediately after the load instruction. The last case is a cache write miss, it is the least serious miss because there are write buffers usually, to store data until they are transferred to main memory or a block is allocated in the cache. The processor can continue until the buffer is full.

In order to lower cache miss rate, a great deal of analysis has been done on cache behavior in an attempt to find the best combination of size, associativity, block size, and so on. Sequences of memory references performed by benchmark programs are saved as address traces. Subsequent analysis simulates many different possible cache designs on these long address traces. Making sense of how the many variables affect the cache hit rate can be quite confusing. One significant contribution to this analysis was made by Mark Hill, who separated misses into three categories (known as the Three Cs):

There are three different types of cache misses.

- *Compulsory misses*: are those misses caused by the very first reference to a block of data. They are alternatively called cold-start or first-reference misses. Cache capacity and associativity do not affect the number of compulsory misses that come up by an

application execution. Larger cache block sizes, and software optimization techniques, such as prefetching, can help at this point (as we will see later in this chapter).

- *Capacity misses*: are those misses which a cache of a given size will have, regardless of its associativity or block size. The curve of capacity miss rate versus cache size gives some measure of the temporal locality of a particular reference stream: if cache capacity is not enough to hold the total amount of data being reused by the instruction stream executed, then useful data should be withdrawn from the cache. The next reference to them, brings on capacity misses.

  Increasing the cache size could be beneficial for capacity misses. However, such an enlargement aggravates structural cost. Over-and-above, a large cache size would result to inflation of access time, that is, cache hit latency. To counterbalance the capacity constraints of just a single cache level, memory systems are equipped with two or three levels of caches. The first cache level is integrated in the processor chip, for low latency, and is chosen to have a small size, while other cache levels are placed off chip, with greater capacity.

- *Conflict misses*: are those misses that could have been avoided, had the cache not evicted an entry earlier. A conflict miss can occur even if there is available space in the cache. It is the result of a claim for the same cache line, by two different blocks of data. Conflict misses can be further broken down into mapping misses, which are unavoidable given a particular amount of associativity, and replacement misses, which are due to the particular victim choice of the replacement policy.

  Another classification concerns data references to array structures. Self conflict misses are conflict misses between elements of the same array. Cross-conflict misses take place when there is conflict between elements of different arrays.

## 2.3 Cache Organization

Cache memories can have different organizations, according to the location in the cache where a block can be placed:

- *Direct mapped* cache: A given block of memory elements can appear in one single place in cache. This place is determined by the block address:

  (cache line mapped) = (Block address) MOD (Number of blocks in cache)

  Direct mapped organization is simple and data retrieval is straightforward. This effectively reduces hit time and allows large cache sizes. On the other hand, direct mapping can not avert conflict misses, even if the cache capacity can support higher hit rates. Hence, fast

processor clocks favor this organization for fist-level caches, that should be kept small in size.

- *Fully Associative* cache: A given block of memory elements can reside anywhere in cache.

  This mapping reduces miss rate because it eliminates conflict misses. Of course, increased associativity comes at a cost. This is the increase of hit time and the high structural cost. However, even if fast processor clocks favor simple caches for on-chip caches, as we draw away from processor chip to higher-level caches, increasing of miss penalty rewards associativity.

- *N-way Set Associative* cache: A set associative organization defines that cache is divided into sets of N cache lines. A given block of memory can be placed anywhere within a single set. This set is determined by the block address:

  (set mapped) = (Block address) MOD (Number of sets in cache)

  A direct mapped cache can be considered as an 1-way set associative cache, while a fully associative cache with capacity of M cache lines can be considered as a M-way set associative cache.

  As far as cache performance is considered, an 8-way set associative cache has, in essence, the same miss rate as a fully associative cache.

### 2.3.1 Pseudo-associative caches

Another approach to improve miss rates without affecting the processor clock is *pseudo-associative caches*. This mechanism is so effective as 2-way associativity. Pseudo-associative caches then have one fast and one slow hit time -corresponding to a regular hit and a pseudo hit. On a hit, pseudo-associative caches work just like direct mapped caches. When a miss occurs in the direct mapped entry, an alternate entry (the index with the highest index bit flipped) is checked. A hit to the alternate entry (pseudo-hit) requires an extra cycle. This pseudo-hit results in the two entries being swapped, so that the next access to the same line would be a fast access. A miss in both entries (fast and alternative) causes an eviction of whichever of the two lines is LRU. The new data is always placed in the fast index, so if the alternate index was evicted, the line in the fast index will need to be moved to the alternate index. So a regular hit takes no extra cycles, a pseudo-hit takes 1 cycle, and access to the L2 and main memory takes 1 cycle longer in a system with a pseudo-associative cache than in one without.

### 2.3.2 Victim Caches

Cache conflicts can be addressed in the hardware through associativity of some form. While associativity has the advantage of reducing conflicts by allowing locations to map to multiple

cache entries, it has the disadvantage of slowing down cache access rates due to added complexity. Therefore minimizing the degree of associativity is an important concern. One option is to have a set-associative primary cache, where addresses are mapped to N-way associative sets. Another approach is to keep the first level cache direct-mapped but also add a *victim cache* off to the side. The goal is to make the victim cache transparent to system so the first level cache thinks it is requesting directly to the DRAM. Victim caching places a small fully-associative cache between the first level cache and its refill path. This small fully-associative cache is loaded with the victims of a miss in the L1 cache. In the case of a miss in L1 cache that hits in the victim cache, the contents of the L1 cache line and the matching victim cache line are swapped. Misses in the cache that hit in the victim cache have only a one cycle miss penalty, as opposed to a many cycle miss penalty without the victim cache. The victim cache approach is appealing because it is tailored to cases where data are reused shortly after being displaced-this is precisely what happens with prefetching conflicts.

According to measurement [Jou90], a 4-entry victim cache removes 20%-95% of conflict misses for a 4 KByte direct mapped data cache. Typically a small fully associated victim cache is used with a direct mapped main (first level) cache. The victim cache usually uses a FIFO (first in, first out) replacement policy. The reason for using FIFO instead of random is that random replacement is more effective for larger cache sizes. The small size of the victim cache makes it probable that certain cache blocks will be replaced more often. A FIFO replacement policy ensures each block has the same chance to be replaced. Theoretically a LRU is the best replacement policy, but is too difficult to implement with more than two sets.

### 2.3.3 Prefetching

Victim caches and pseudo-associativity both promise to improve miss rates without affecting the processor clock rate. *Prefetching* is another technique that predicts soon-to-be used instructions or data and loads them into the cache before they are accessed. Subsequently, when the prefetched instructions or data are accessed there is a cache hit, rather than a miss.

A commonly used method is sequential prefetching, according to which it is predicted that data or instructions immediately following those currently accessed, will be needed in the near future and should be prefetched. Sequential prefetching fetches a memory block that caused a cache miss, along with the next $n$ consecutive cache blocks. It can be more or less "aggressive" depending on how far ahead in the access stream it attempts to run - that is, how large $n$ is.

Both instructions and data can be prefetched, either directly into the caches or into an external buffer that can be more quickly accessed than main memory. An example of this is the Intel Xeon Pentium 4 processor, which can perform both software and hardware prefetching. The hardware prefetcher requires a couple of misses before it starts operating. It will attempt to prefetch two cache lines ahead of the prefetch stream into the unified second-level cache based on prior reference patterns. In a cache miss, the hardware mechanism fetches the adjacent

cache line within an 128-byte sector that contains the data needed. There is also a software controlled mechanism that fetches data into the caches using prefetch instructions. The hardware instruction fetcher reads instructions that are likely to be executed along the path predicted by the branch target buffer (BTB) into instruction streaming buffers.

Although usually beneficial, prefetching, especially aggressive prefetching may reduce performance. In some cases prefetched data or instructions may not actually be used, but will still consume system resources - memory bandwidth and cache space. It is possible that a prefetch will result in useless instructions or data replacing other instructions or data in the cache that will soon be needed. This effect is referred to as cache pollution. The effects of cache pollution most often increase as prefetching becomes more aggressive. Another prevalent effect of aggressive prefetching is bus contention. Bus contention occurs when multiple memory accesses have to compete to transfer data on the system bus. This effect can create a scenario where a demand-fetch is forced to wait for the completion of a useless prefetch, further increasing the number of cycles the processor is kept idle.

On average, hardware prefetching benefits performance and works with existing applications, without requiring extensive study of prefetch instructions. However it may not be optimal for any particular program especially for irregular access patterns and has a start-up penalty before the hardware prefetcher triggers and begins initiating fetches. This start-up delay has a larger effect for short arrays when hardware prefetching generates a request for data beyond the end of an array (not actually utilized). There is a software alternative to hardware prefetching, the compiler-controlled prefetching - the compiler requests the data before it is needed. This technique can improve memory access time significantly, even irregular access patterns, if the compiler is well implemented and can avoid references that are likely to be cache misses. However it requires new instructions to be added which results to issuing port bandwidth overhead.

## 2.4 Cache replacement policies

When a cache miss occurs, the desired data element should be retrieved from memory and placed in a cache location, according to the cache mapping organization. This may result in the eviction of another blocked data stored in the same location. In direct mapping, there exists only a single candidate. In set associative and fully associative mappings more than one cache lines are candidate for eviction. The cache line selection is critical, as it affects performance. Which block should be replaced is determined by a replacement policy:

The primary strategies used in practice are:

- Random: Candidate cache lines are randomly selected. It counts on the principle of uniform propagation of cache line allocation due to fortuity of selection. This strategy is simple to implement in hardware, but passes over the locality of references.

- Least-Recently Used (LRU): The candidate block is the one that has been left unused for the longest time. It implements the principle of locality, avoiding to discard blocks of data that have increased probability to be needed soon. This strategy keeps track of block accesses. Hence, when the number of blocks increases, LRU becomes more expensive (harder to implement, slower and often just approximated).

Other strategies that have been used are:

- First In First Out (FIFO)
- Most-Recently Used (MRU)
- Least-Frequently Used (LFU)
- Most-Frequently Used (MFU)

## 2.5 Write policies

Special care should be taken when an instruction writes to memory. In the common case, that is in a read cache access, the block can be read at the same time that the tag is read and compared. So the block read begins as soon as the block address is available. This is not the case for writes. Modifying a block cannot begin until the tag is checked to see if the address is a hit.

The write policies on write hit are:

- *Write Through*: The modified item is written to the corresponding blocks in all memory levels.

  This is an easy to implement strategy. Main memory is always the consistency point: it has the most current copy of the data. As a result, read miss never results in writes to main memory. On the other hand, writes are slow. Every write draws a main memory access and memory bandwidth is spent.

- *Write back*: The modified item is written only to the block in the cache, setting "dirty" bit for this block. The whole cache block is written to main memory only when the cache line is replaced.

  In this strategy the consistency point is the cache. As a result, read misses which replace other blocks in the cache, draw writes of dirty blocks to main memory. On the other hand, writes are completed efficiently, at cache hit time. Moreover, this will eliminate extra memory accesses, as multiple writes within a block require only one write to main memory, which results to less memory bandwidth.

In case of a write miss there are two common options:

- *Write Allocate*: The block is loaded into the cache and is updated with the modified data afterwards.
- *No Write Allocate*: Write instructions do not affect the cache. The modified data updates the block in main memory but this block is never brought into the cache.

Although either write-miss policy could be used with write through or write back, write-back caches generally use write allocate (hoping that subsequent writes to that block will be captured by the cache) and write-through caches often use no-write allocate (since subsequent writes to that block will still have to go to memory).

## 2.6 Virtual Memory

The whole amount of data that are accessed by an application program can not always fit in the main memory. This was the reason that *virtual memory* was invented, to allow some of the data remain in hard drives. The address space is divided to blocks, called *pages*. Each page of a program are stored either in main memory or in the hard drive. When the processor calls for data that can not be found either in the cache or in the main memory, then a *page fault* occurs. Then, the whole page has to be transferred from hard drive to main memory. The page fault resolution lasts for a significant large time space. So, it is handled by the operating system, while another process is being executed.

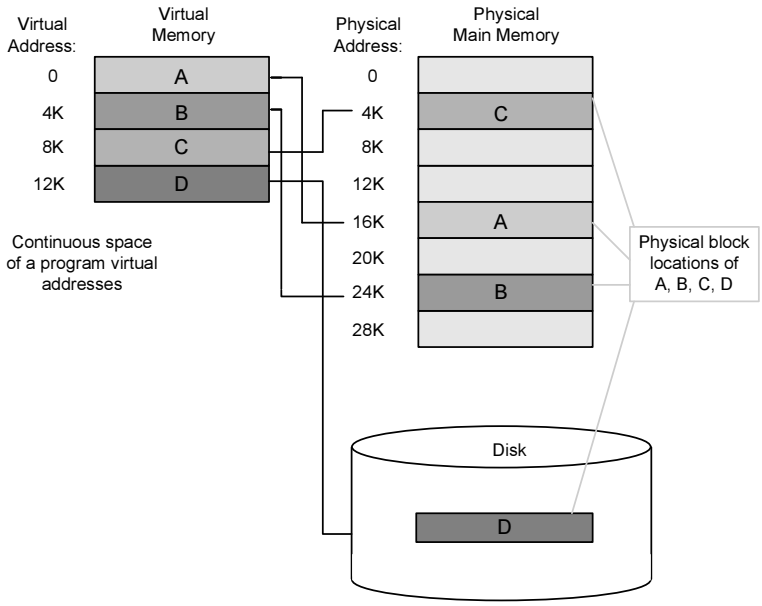



**Figure 2.2:** Virtual Memory

The processor generates *virtual addresses* while the memory is accessed using *physical addresses*. Every virtual page is mapped to a physical page through a mapping table called *page*

*table*. Of course, it is possible for a virtual page to be absent from main memory and not be mapped to a physical address as soon, residing instead on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data and code.

Page tables contain a large number of entries. Hence, they are stored in main memory. *Translation Lookaside Buffer (TLB)* holds entries only for the most recently accessed pages and only for valid pages. It is a special purpose cache, invented to deter main memory access for every address translation. Fully associative organization is usually applied to TLBs.

## 2.7 Iteration Space

Each iteration of a nested loop of depth n is represented by an n-dimensional vector:

$$\vec{j} = (j_1, \ldots, j_n)$$

Each vector coordinate is related with one of the loops. The leftmost coordinate ($j_1$) is related to the outer loop, while the rightmost coordinate ($j_n$) is related to the inner loop.

*Iteration space* of a nested loop of depth n is an n-dimensional polyhedron, which is taken in the boundaries of the n loops. Every point of this n-dimensional space is a distinct iteration of the loop body. The iteration space is represented as follows:

$$J_n = \{\vec{j}(j_1, \ldots, j_n) \mid j_i \in Z \wedge l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$$

## 2.8 Data dependencies

Data dependencies arise when two memory references access the same memory location. Such a dependency can be:

- *True Data Dependence*: The first reference writes in a memory location while the second wants to read the new value.
- *Anti-dependence*: The first reference reads from a memory location while the second wants to write in it.
- *Output Dependence*: Both references write in the same memory location.
- *Input Dependence*: Both references read from the same memory location.

On a code transformation, we should preserve relative iteration order which is determined by the first three kind of dependencies. Otherwise, code results will be wrong. Overlooking the input dependencies does not bring any wrong results. However, they should be taken into account when evaluating data reuse

*Dependence Vector* of a nested loop iteration is the difference between two vectors: the vector $\vec{j} = (j_1, \ldots, j_n)$, that represents the iteration in issue, and the vector $\vec{j'} = (j_1', \ldots, j_n')$, that represents the dependent iteration (the results of iteration $\vec{j'}$ are directly used in processing of iteration $\vec{j}$).

Every dependence vector is represented by an n-dimensional vector: $\vec{d_i} = (d_{i1}, \ldots, d_{in})$.

## 2.9 Data Reuse

Since transformations change iteration order, identifying reuse, guides transformations to exploit locality. In some cases we need to evaluate different kinds of reuse, in order to find the best combination of transformations [WL91].

There are four different types of reuse.

---

In order to better explain the different types of reuse, we will use the following nested loop, as an example:

```
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] + A[i][k+1] * B[k+1][j];
```

---

- Self-temporal reuse: A reference within a loop accesses the same data element in different iterations.

  Self-temporal reuse occurs in reference $C[i][j]$ along loop $k$. Similarly, references $A[i][k]$, $A[i][k+1]$ have self-temporal reuse in loop $j$ and references $B[k][j]$, $B[k+1][j]$ have self-temporal reuse in loop $i$.

- Self-spatial reuse: A reference accesses data of the same cache line in different iterations.

  In the above example, self-spatial reuse occurs in references $A[i][k]$, $A[i][k+1]$ along loop $k$, in references $B[k][j]$, $B[k+1][j]$ along loop $j$ and in reference $C[i][j]$ along loop $j$. However, if $N$ is quite large, spatial reuse can only be exploited along the innermost loop $k$, because critical cache lines will have been evicted from the cache, when array references will ask for them. When tiling (see section 2.10) is applied, choosing tiles sizes that fit in the cache, reuse can be exploited in all three innermost loops ($i$, $j$, $k$).

- Group-temporal reuse: Different references access the same data element.

In the above example, group-temporal reuse exists between references $A[i][k]$ and $A[i][k+1]$ of the previous $k$ iteration. Similarly, reference $B[k][j]$ uses the same data as $B[k+1][j]$ of the previous $k$ iteration.

- Group-spatial reuse: Different references access the same cache line.

  In the above example, group-spatial reuse exists between references $A[i][k]$ and $A[i][k+1]$ in each iteration, because they access neighboring data. On the other hand, there is no group-spatial reuse between references $B[k][j]$ and $B[k+1][j]$, as loop $k$ controls the column dimension of $B$ (we assume that the C storage order is followed, that is row-major).

## 2.10 Loop Transformations

Iterative instruction streams are usually the most time and memory consuming parts of a program. Thus, a lot of compiler analysis and optimization techniques have been developed to make the execution of such streams faster. Nested loops are usually the most common form of iterative instruction streams. Loop transformations play an important role in improving cache performance and effective use of parallel processing capabilities. In this thesis we focus on $n$-nested loops, which contain references to $n$-dimensional array data. Loop and data transformations aim at enhancing code performance by changing data access order and arrangement of the data in address space, to improve data locality.

Data layout optimizations [CL95], [RT98a], [KRC99] rearrange the allocation of data in memory, enhancing code performance by improving spatial locality and avoiding conflict misses. Increased spatial locality can reduce the working set of a program (the number of pages accessed) or can minimize reuse distance to avoid eviction of data from the cache once they have been loaded into it. Conflict misses is also a determinant factor of performance. A significant percentage of the whole program cache misses is evoked due to conflicts in the cache. Data layout optimization techniques include modification of the mapping distance between variables or rearrange of structure fields, change of the array dimension sizes, interchange of array dimensions, and linearizing multi-dimensional arrays. Data alignment and array padding are such transformations that put additional (null) elements among useful data in order to change mapping of these data to the cache. Transformations of array layouts is also an important category. A special case, non-linear array layouts, will be presented analytically in chapter 3.

*Code (loop) transformations* change the execution order of iterations, to better exploit data reuse and memory hierarchy. Most cases need a combination of such transformations, to achieve best performance. These techniques include:

- **Induction variable elimination**: Some loops contain two or more induction variables that can be combined into one induction variable. Induction variable elimination can reduce the number of operations in a loop and improve code space.

- **Copy propagation**: propagates the right content of an assignment statement to the places where the left argument appears (until one or the other of the variables involved in the statement is reassigned). This optimization tries to eliminate useless copy assignments in code. It is a useful "clean up" optimization frequently used after other optimizations have already been run.

- **Constant propagation**: It is an optimization usually applied during compilation time. If a particular constant value is known to be assigned to a variable at a particular point in the program, references to this variable can be substituted with that constant value. This is similar to copy propagation, except that it applies to eliminating useless assignment operations involving constants.

- **Dead code elimination**: Removes all unused code (unreachable or that does not affect the program).

- **Constant folding**: Expressions with constant (literal) values can be evaluated at compile time and simplified to a constant result value. Thus, run-time performance is improved by reducing code size and avoiding evaluation at compile-time. Particularly useful when constant propagation is performed.

- **Forward Store**: Stores to global variables in loops can be moved out of the loop to reduce memory bandwidth requirements.

- **Function Inlining**: The overhead associated with calling and returning from a function can be eliminated by expanding the body of the function inline, and additional opportunities for optimization may be exposed as well. Function inlining usually increases code space, which is affected by the size of the inlined function, the number of call sites that are inlined, and the opportunities for additional optimizations after inlining. Some compilers can only inline functions that have already been defined in the file; some compilers parse the entire file first, so that functions defined after a call site can also be inlined; still other compilers can inline functions that are defined in separate files.

- **Code hoisting and sinking**: If the same code sequence appears in two or more alternative execution paths, the code may be hoisted to a common ancestor or sunk to a common successor. This reduces code size, but does not reduce instruction count. Another interpretation of hoisting for loop-invariant expressions is to hoist the expressions out of loops, thus improving run-time performance by executing the expression only once rather than at each iteration.

- **Instruction combining**: Sometimes, two statements can be combined into one statement. Loop unrolling can provide additional opportunities for instruction combining.

- **Forward expression substitution**: Substitutes the left variable of an assignment expression to the places where the left argument appears with the expression itself. This substitution needs control flow analysis, to guarantee that the definition is always executed before the statement into which it is substituted.

- **Register allocation**: There are two types of register allocation: Local - Within a basic block (a straight line sequence of code) tracks register contents, allocates values into registers, and reuses variables and constants from registers. This can also avoid unnecessary register-register copies. Global - Within a sub-program, frequently (using weighted counts) accessed variables and constants, used in more than one basic block, are allocated to registers. Usually there are many more register candidates than available registers.

- **Unswitching**: A loop containing a loop-invariant `if` statement can be transformed into an `if` statement containing two loops. After unswitching, the `if` expression is only executed once, thus improving run-time performance.

A special case of loop transformations is compound transformations, consisting of loop permutation, loop fusion, loop distribution, and loop reversal.

---

In the following, (where a special code is not presented) we will use the example code found below.

```
for (i = 0; i <= N; i++)
  for (j = i + 1; j <= N; j++)
    A[j][i] = ...;
```

---

- **Loop Normalization**

  The loop normalization consists of transforming all the loops of a given module into a normal form. In this normal form, the lower bound and the increment are equal to one (1).

---

If the initial loop is:

```
for (i = lower; i <= upper; i+ = increment)
  ...
```

the transformation gives the following code:

```
for (NLi = 1; NLi <= ceiling(upper − lower + 1)/increment; NLi + +)
{
  i = increment ∗ NLi + lower − increment;
  ...
}
```

The normalization is done only if the initial increment is a constant number. The normalization produces two assignment statements on the initial loop index. The first one (at the beginning of the loop body) assigns it to its value function of the new index and the second one (after the end of the loop) assigns it to its final value

- **Loop Reversal**

  Loop reversal reverses the order in which the iterations of a loop nest are executed and is legal if dependencies remain unchanged (they are still carried on outer loops). Reversal does not change the pattern of reuse, but makes other transformations operational, i.e., it may enable permutation to achieve better locality.

---

The unimodular matrix that declares loop reversal of the example code is:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

The resulting code is:

```
for (i = 0; i <= N; i + +)
  for (j = −N; j <= −i − 1; j + +)
    A[j][i] = ...;
```

---

- **Loop Interchange or Permutation**

  Changes the nesting order of some or all loops. This can minimize the stride of array element access during loop execution and reduce the number of memory accesses needed. To determine the loop permutation which accesses the fewest cache lines, the following observation should be taken into account. If loop $i$ promotes more reuse than loop $j$ (accesses to array data by successive iterations of loop $i$ follow the storage order for more references than by successive iterations of loop $j$) when both are considered for the innermost loop, $i$ will likely be nested inside loop $j$, if permissible.

---

The following unimodular matrix specifies the interchange of our example code:

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

The resulting code is:

```
for (j = 1; j <= N; j++)
  for (i = 0; i <= j - 1; i++)
    A[j][i] = ...;
```

---

- **Loop Skewing**

  This transformation changes the iteration space shape . After applying loop skewing, the iteration space is no longer orthogonal or square, but trapezoidal. It does not optimize the code performance on its own, but makes other kind of transformations permissible. On the other hand, in some cases the loop boundaries become complicated, so that they are too time-consuming to be calculated. For this reason, loop skewing has to be applied only if necessary, as it can worsen performance, instead of boosting it.

---

The unimodular matrix that describes loop skewing is:

$$T = \begin{bmatrix} 1 & f \\ 0 & 1 \end{bmatrix} \quad or \quad T = \begin{bmatrix} 1 & 0 \\ f & 1 \end{bmatrix}$$

If we skew the loops using the unimodular matrix:

$$T = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

The resulting example code is:

```
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++)
    A[j][i] = ...;
```

---

- **Loop Distribution**

  Moves independent statements in one loop into multiple, new loops. This transformation is used in cases where there is a large number of instructions and data in a single loop. This can reduce the amount of memory space (working set) used during each loop so that the remaining data may fit in the cache, reducing the probability of capacity and conflict misses. It can also create improved opportunities for loop blocking.

---

Regarding the initial code:

```
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++) {
    A[j][i] = ... ;
    D[i][j] = ... ; }
```

The resulting example code is:

```
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++)
    A[j][i] = ... ;
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++)
    D[i][j] = ... ;
```

---

- **Loop Fusion**

  Combines the bodies from two or more adjacent loops that use some of the same memory locations into a single loop. It is legal only if no data dependencies are reversed. This can avoid the need to load those memory locations into the cache multiple times and improves opportunities for instruction scheduling. Although loop fusion reduces loop overhead, it should be applied with care, as it does not always improve run-time performance, or may even reduce performance. For example, a memory architecture may provide better performance if two arrays are initialized in separate loops, rather than initializing both arrays simultaneously in one loop.

---

Regarding the initial code:

```
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++)
    A[i][j] = B[i][j] * C[j];
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++)
    D[i][j] = A[i][j] * C[j];
```

The resulting example code is:

```
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++) {
    A[i][j] = B[i][j] * C[j];
    D[i][j] = A[i][j] * C[j];
  }
```

---

- **Scalar Replacement**

  Replaces the use of an array element with a scalar variable under certain conditions. It gives the capability of assigning scalars to registers, to keep their content close to the processor for fast access.

---

Regarding the initial code:

```
for (i = 0; i <= N; i++)
  for (j = 1; j <= N - i; j++)
    A[i] += B[i][j] * C[j];
```

The resulting example code is:

```
for (i = 0; i <= N; i++) {
  reg_var = A[i];
  for (j = 1; j <= N - i; j++)
    reg_var += B[i][j] * C[j];
  A[i] = reg_var;
}
```

---

- **Outer Loop Unrolling**

  Unrolling the outer or inner loops replicates the body of the loop, to minimize the number of instructions and memory accesses needed. This also improves opportunities for instruction level parallelism and scalar replacement. Finally, it reduces the number of iterations, minimizing loop overhead due to latencies introduced by mispredicted branches.

In the example code, the body of the example loop is replicated once and the number of loop iterations is reduced from $N$ to $N/2$. Below is the code fragment after loop unrolling.

```
for (i = 0; i <= N; i++)
  for (j = i + 1; j <= N; j+= 2)
  {
    A[i][j] = ...;
    A[i][j + 1] = ...;
  }
```

- **Loop Tiling (blocking)**

  It splits the initial iteration space $J_n$ into n-dimensional iteration spaces, called tiles. This transformation reduces the working set of inner loops (number of instructions included in a tile). It can maximize data reuse of multidimensional array elements by completing as many operations as possible on array elements currently in the cache.

Below is the example code after loop tiling.

```
for (ii = 0; ii <= N; ii+= tile_stepi)
  for (jj = ii; jj <= N; jj+= tile_stepj)
    for (i = ii; i < ii + tile_stepi; i++)
      for (j = max(i + 1, jj); j <= jj + tile_stepj; j++)
        A[i][j] = ...;
```

CHAPTER 3

# Fast Indexing for Blocked Array Layouts

This chapter proposes a new method to perform blocked array layouts combined with a fast indexing scheme for numerical codes. We use static loop performance analysis to specify the optimal loop nesting order and legal transformations (including tiling) that give the best recomposition of iterations. Array elements are stored exactly as they are swept by the tiled instruction stream in a blocked layout. We finally apply our efficient indexing to the resulting optimized code, to easily translate multi-dimensional indexing of arrays into their blocked memory layout using quick and simple binary-mask operations.

The remainder of this chapter is organized as follows: Section 3.1 briefly discusses the problem of data locality using as example the typical matrix multiplication algorithm. Section 3.2 reviews definitions related to Morton ordering. Section 3.3 presents previously proposed non-linear array layouts, as well as our blocked array layouts along with our efficient array indexing. Finally, concluding remarks are presented in Section 3.4.

## 3.1 The problem: Improving cache locality for array computations

In this section, we elaborate on the necessity for both control (loop) and data transformations, to fully exploit data locality. We present, stepwise, all optimization phases to improve locality of references with the aid of the typical matrix multiplication kernel.

---

1. unoptimized version

```
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];
```

2. tiled code

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

3. loop and data transformation

```
for (kk=0; kk < N; kk += step)
  for (jj = 0; jj < N; jj += step)
    for (i = 0; i < N; i++)
      for (k = kk; (k < N && k < kk + step); k++)
        for (j = jj; (j < N && j < jj + step); j++)
          Cr[i,j] += Ar[i,k] * Br[k,j];
```

4. blocked array layouts

```
for (ii = 0; ii < N; ii += step)
  for (kk = 0; kk < N; kk += step)
    for (jj = 0; jj < N; jj += step)
      for (i = ii; (i < N && i < ii + step); i++)
        for (k = kk; (k < N && k < kk + step); k++)
          for (j = jj; (j < N && j < jj + step); j++)
            Czz[i + j] += Azz[i + k] * Bzz[k + j];
```

---

**Loop Transformations:** Code #1 presented above, shows the typical, unoptimized version of the matrix multiplication code. Tiling (code #2) restructures the execution order, such that the number of intermediate iterations and, thus, data fetched between reuses, are reduced. So, useful data are not evicted from the register file or the cache, before being reused. The tile size (*step*) should be selected accordingly to allow reuse for a specific level of memory hierarchy [LRW91].

**Loop and Data Transformations:** Since, loop transformation alone can not result in the best possible data locality, a unified approach that utilizes both control and data transformations becomes necessary. In code #2, loop $k$ scans different rows of $B$. Given a row-order array layout,

spatial reuse can not be exploited for $B$ along the innermost loop $k$. Focusing on self-spatial reuse [KRC99] (since self-temporal reuse can be considered as a subcase of self-spatial, while group spatial are rare), the transformed code takes the form of code #3. Firstly we fixed the layout of the LHS (Left Hand Side) array, namely array $C$, because in every iteration the elements of this array are both read and written, while the elements of arrays $A$ and $B$ are only read. Choosing $j$ to be the innermost loop, the fastest changing dimension of array $C[i, j]$ should be controlled by this index, namely $C$ should be stored by rows (Cr). Similarly, array $B[k, j]$ should also be stored by rows (Br). Finally, placing loops in $ikj$ order is preferable, because we exploit self-temporal reuse in the second innermost loop for array $C$. Thus, $A[i, k]$ should also be stored by rows (Ar).

**Loop and non-linear Data Transformations:** Selecting the best loop order within a nested loop, when optimizing code, is a very intriguing task. In order to evaluate the merit of non-linear data transformations, let us consider the code shown in code #4. We assume that the elements of all three arrays are stored exactly as swept by the program (we call this layout ZZ-order, as extensively presented in section 3.3). The loop ordering remains the same as in code #3, except that, tiling is also applied in loop $i$, so as to have homomorphic shaped tiles in all three arrays and simplify the computations needed to find the array element location.

## 3.2 Morton Order matrices

In order to capture the notion of Morton order [Wis01], a recursive storage order of arrays, the basic elements of the dilated integer algebra are presented. Morton defined the indexing of a two-dimensional array and pointed out the conversion to and from cartesian indexing available through bit interleaving (figure 3.1). The following definitions are only for two-dimensional arrays (A $d$-dimensional array is represented as a $2^d$-ary tree).

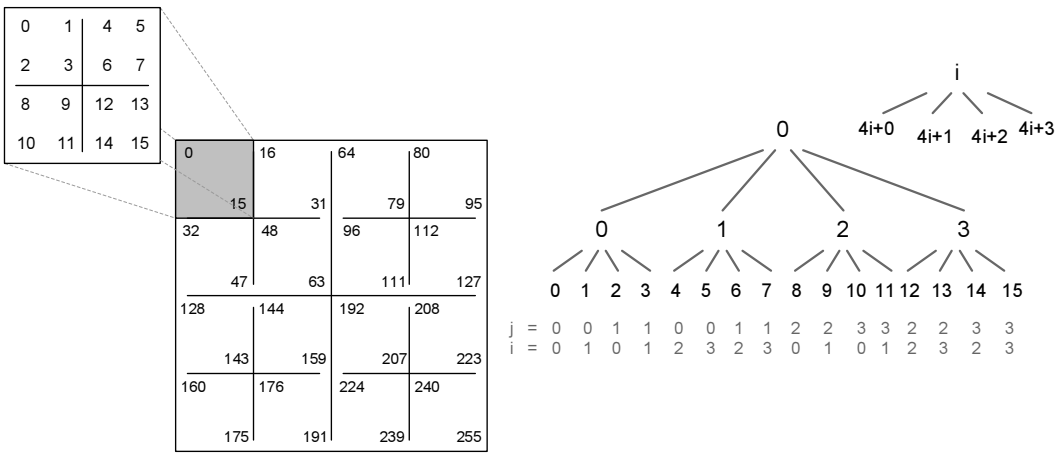



**Figure 3.1:** Row-major indexing of a 4×4 matrix, analogous Morton indexing and Morton indexing of the order-4 quadtree

**Definition 3.1** *The integer* $\overrightarrow{b} = \sum_{k=0}^{w-1} 4^k$ *is the constant 0x55555555 and is called evenBits. Similarly,* $\overleftarrow{b} = 2\vec{b}$ *is the constant 0xaaaaaaaa and is called oddBits.*

**Definition 3.2** *The even-dilated representation of* $j = \sum_{k=0}^{w-1} j_k 2^k$ *is* $\sum_{k=0}^{w-1} j_k 4^k$, *denoted* $\overrightarrow{j}$. *The odd-dilated representation of* $i = \sum_{k=0}^{w-1} i_k 2^k$ *is* $2\overrightarrow{i}$ *and is denoted* $\overleftarrow{i}$.

**Theorem 3.1** *The Morton index for the* $< i, j >^{th}$ *element of a matrix is* $\overleftarrow{i} \vee \overrightarrow{j}$, *or* $\overleftarrow{i} + \overrightarrow{j}$.

Thus, let us consider a cartesian row index $i$ with its significant bits "dilated" so that they occupy the digits set in *oddBits* and a cartesian index $j$ been dilated so its significant bits occupy those set in *evenBits*. If array $A$ is stored in Morton order, then element $[i, j]$ can be accessed as $A[i + j]$ regardless of the size of array.

---

The implementation of the loop

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

will be modified to

`for(`$i_m$`=0; `$i_m < N_m$`; `$i_m$`=(`$i_m - evenBits$`)&`$evenBits$`)` ...

where $i_m = \overrightarrow{i}$ and $N_m = \overrightarrow{N}$

---

Using a dilated integer representation for the indexing of an array referenced by a loop expression, if an index is only used as a column index, then its dilation is odd. Otherwise it is even-dilated. If used in both roles, then doubling gives the odd-dilation as needed.

---

So, the code of matrix multiplication will be as follows:

```
#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];
```

where *rowsEven*, *colsOdd* and *rowsAEven* are the bounds of arrays when transformed in dilated integers. Notice that $k$ is used both as column and as row index. Therefore, it is translated to $\overrightarrow{k}$ and for the column indexing of array $B$, $2*k$ is used,

which represents $\overleftarrow{k}$.

---

**Example 3.1:**

Let's assume that we are looking for the array element:

$$C[i,j] = C[3,4]$$

In order to find the storage location of this element in linear memory, in the Morton layout, the following procedure should be kept.

Row index $j$ should be dilated by $evenBits = 1010101_{<2>}$ constant. The linear value of $j$ is:

$$j = 4_{<10>} = 100_{<2>}$$

Even dilation of this value means that among the binary digits of $j$, zero digits should be inserted. That is, where an 1 is found in the $evenBits$ constant, the binary digits of $j$ should be placed:

$$\overrightarrow{j} = x_3 0 x_2 0 x_1 0 x_0$$

where $x_n$, the value of the $n$-th binary digit of $j$.

$$\overrightarrow{j} = 0010000_{<2>} = 8_{<10>}$$

Similarly, column index $i$ should be dilated by $oddBits = 0101010_{<2>}$ constant. The linear value of $i$ is:

$$i = 3_{<10>} = 011_{<2>}.$$
$$\overleftarrow{i} = 0 x_2 0 x_1 0 x_0 0$$

where $x_n$, the value of the $n$-th binary digit of $i$.

$$\overleftarrow{i} = 0001010_{<2>} = 10_{<10>}$$

As a result, $C[i,j]$ is stored in:

$$\overleftarrow{i} + \overrightarrow{j} = 0001010 + 0010000 = 0011010_{<2>}$$
$$\overleftarrow{i} + \overrightarrow{j} = 10_{<10>} + 8_{<10>} = 18_{<10>}.$$

This is element $C[18]$. ◇

## 3.3 Blocked array layouts

In this section we present the non-linear data layout transformation. Since the performance of so far presented methods [CL95], [KRC99], [WL91] is better when tiling is applied, we would achieve even better locality, if array data are stored neither column-wise nor row-wise, but in a blocked layout. Such layouts were used by Chatterjee et al in [CJL+99], [CLPT99], in order to obtain better interaction with cache memories. According to [CJL+99], a 2-dimensional $m \times n$ array can be viewed as a $\lceil \frac{m}{t_R} \rceil \times \lceil \frac{n}{t_C} \rceil$ array of $t_R \times t_C$ tiles. Equivalently, the original 2-dimensional array space $(i, j)$ is mapped into a 4-dimensional space $(t_i, t_j, f_i, f_j)$, as seen in figures 3.2 and 3.4. The proposed layout for the space $(f_i, f_j)$ of the tile offsets is a canonical one, namely column-major or row-major, according to the access order of array elements by the program code. The transformation function for the space $(t_i, t_j)$ of the tile co-ordinates can be either canonical or follow the Morton ordering.

### 3.3.1 The 4D layout

In $L_{4D}$, both $L_T$ and $L_F$ are canonical layouts. In our example, both of them should be row-major, in order to follow the access order in matrix multiplication algorithm. In the following, consider an array of $27 \times 27$ elements, divided to $4 \times 4$ tiles.

The final 4-dimensional array is drawn in figure 3.2. The 4-dimensional space is presented as an 2-dimensional space of tile co-ordinates which contain pointers on 2-dimensional arrays of tile offsets.

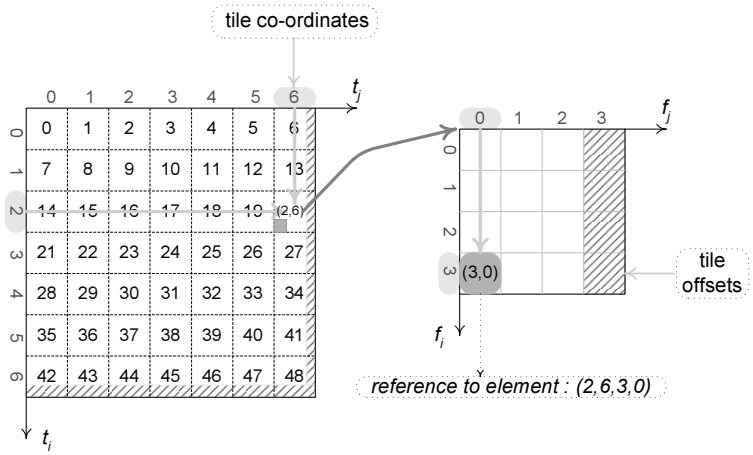



**Figure 3.2:** The 4-dimensional array in 4D layout

---

The value of each dimension is calculated as follows:

$$\begin{aligned} \text{tile co-ordinates}: \quad & t_i = i \; div \; step \\ & t_j = j \; div \; step \\ \text{tile offsets}: \quad & f_i = i \; mod \; step \\ & f_j = j \; mod \; step \end{aligned}$$

If the original code of matrix multiplication is tiled in all three dimensions the derived code is a 6-depth loop:

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

Using the 4D layout, the implementation of the matrix multiplication code should be:

```
for (ii = 0; ii < NN; ii++) {
  stepi = (N > step * (ii + 1)?step : (N − ii * step));
  for (kk = 0; kk < NN; kk++) {
    stepk = (N > step * (kk + 1)?step : (N − kk * step));
    for (jj = 0; jj < NN; jj + +) {
      stepj = (N > step * (jj + 1)?step : (N − jj * step));
      for (i=0; i < stepi; i++)
        for (k=0; k < stepk; k++)
          for (j=0; j < stepj; j++)
            C[ii][jj][i][j]+ = A[ii][kk][i][k] * B[kk][jj][k][j]; }
  }
}
```

where $N$, $step$ are the array and tile bound respectively, and $NN = \lceil \frac{N}{step} \rceil$.

---

**Example 3.2:**

Let's assume that we are looking for the array element:

$$C[i, j] = C[11, 24]$$

To calculate the storage position of this element, the procedure that is described below and illustrated in figure **??** should be followed. We assume that array $C$ is of size $27 \times 27$, follows the $L_{4D}$ layout, and is split to tiles of size $4 \times 4$.

Tile coordinates:

$$t_i = i \; div \; step = 11 \; div \; 4 = 2_{<10>}$$
$$t_j = j \; div \; step = 24 \; div \; 4 = 6_{<10>}$$

The requested element is stored in the tile, with coordinates $[2, 6]$.

Tile offsets:

$$f_i = i \; mod \; step = 11 \; mod \; 4 = 3_{<10>}$$
$$f_j = j \; mod \; step = 24 \; mod \; 4 = 0_{<10>}$$

The requested element is stored inside a tile, in a position with coordinates $[3, 0]$

In $L_{4D}$, element $C[11, 24]$ of the original array, is stored in a 4-dimensional array in position $[2][6][3][0]$. ◇

### 3.3.2 The Morton layout

In $L_{MO}$, $L_T$ follows Morton ordering, while $L_F$ has a canonical layout (in our example row-major layout in order to keep step with access order). The original array is the same as the previous case (array of $27 \times 27$ elements, divided to $4 \times 4$ tiles), apart from the number of padding-elements which is greater than in the previous case, so that the array dimension becomes a power of two.

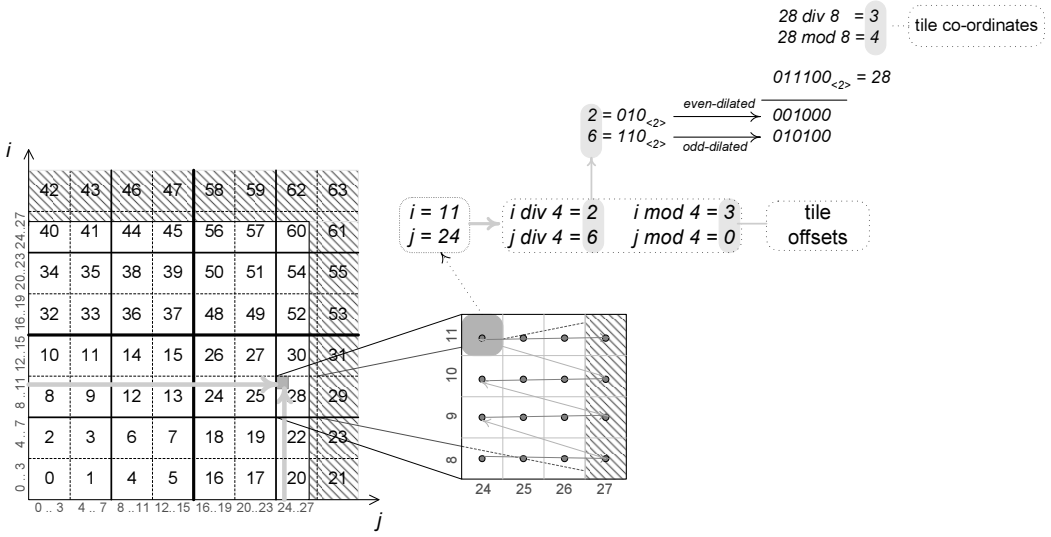



**Figure 3.3:** The tiled array and indexing according to Morton layout

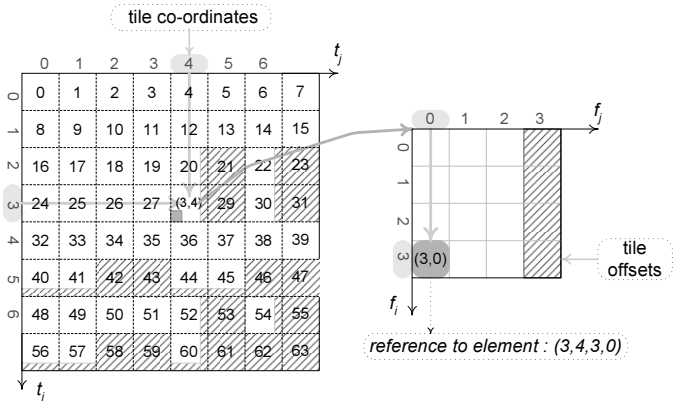



**Figure 3.4:** The 4-dimensional array in Morton layout in their actual storage order: padding elements are not canonically placed on the borders of the array, but mixed with useful elements

The final 4-dimensional array is drawn in figure 3.4. Notice that the padded elements are not in the border of the array but they are mixed with the original elements.

---

The value of each dimension is calculated as follows:

$$\text{tile co-ordinates :} \quad tile_{num} = \overleftarrow{(i \ div \ step)} | \overrightarrow{(j \ div \ step)}$$
$$t_i = tile_{num} \ div \ n_t$$
$$t_j = tile_{num} \ mod \ n_t$$
$$\text{tile offsets :} \quad f_i = i \ mod \ step$$
$$f_j = j \ mod \ step$$

where $\overleftarrow{x}$ and $\overrightarrow{x}$ is the odd and even dilated representation of $x$, respectively, and $n_t$ the number of tiles that fit in each row or column of the padded array (we consider square padded arrays, as this is a demand of the Morton layout).

When applying the Morton layout, the resulting code for matrix multiplication is :

```
for (ii = 0; ii < NNodd; ii = (ii − oddBits)&oddBits) {
  if (((ii − oddBits)&oddBits) == NNodd)
    stepi = (step_b == 0?step : step_b);
  else stepi = step;
  for (kk = 0; kk < NNeven; kk = (kk − evenBits)&evenBits) {
    if (((kk − evenBits)&evenBits) == NNeven)
      stepk = (step_b == 0?step : step_b);
    else stepk = step;
    x = ii|kk;
    tiA = x/NN;
    tkA = x%NN;
    for (jj = 0; jj < NN; jj++) {
      if (((jj − evenBits)&evenBits) == NNeven)
        stepj = (step_b == 0?step : step_b);
      else stepj = step;
      y = ii|jj;
      tiC = y/NN;
      tjC = y%NN;
      z = (kk << 1)|kk;
      tkB = z/NN;
      tjB = z%NN;
      for (i=0; i < stepi; i++)
        for (k=0; k < stepk; k++)
          for (j=0; j < stepj; j++)
            C[tiC][tjC][i][j]+ = A[tiA][tkA][i][k] ∗ B[tkB][tjB][k][j];
    }
  }
}
```

where $step\_b = N\%step$

$NNodd = \overleftarrow{NN}$

$NNeven = \overrightarrow{NN}$

The above code does not contain any time-consuming calculations to locate the right array elements. Even for the $L_{MO}$ layout, boolean operations are considered, which add the minimum possible overhead. However, referring to 4-dimensional arrays produces long assembly codes, thus, repetitive load and add instructions, which, as seen in experimental results, are too time consuming and, thus, degrade the total performance.

**Example 3.3:**

that we are looking for the array element:

$$C[i,j] = C[11,24]$$

To calculate the storage position of this element, the procedure that is described below and illustrated in figure **??** should be followed. We assume that array $C$ is of size $27 \times 27$, follows the $L_{4D}$ layout, and is split to tiles of size $4 \times 4$.

Tile coordinates:

$$tile_{num} = \overleftarrow{(i\ div\ step)}|\overrightarrow{(j\ div\ step)} = \overleftarrow{(11 div 4)}|\overrightarrow{(24 div 4)} = \overleftarrow{2}\,|\,\overrightarrow{6}$$

The above indices should be dilated:

$$even - dilation\text{: } \overleftarrow{2_{<10>}} = \overleftarrow{010_{<2>}} = 001000_{<2>} = 8_{<10>}$$
$$odd - dilation\text{: } \overrightarrow{6_{<10>}} = \overrightarrow{110_{<2>}} = 010100_{<2>} = 20_{<10>}$$
$$tile_{num} = \overleftarrow{2}\,|\,\overrightarrow{6} = 8 + 20 = 28$$

$$t_i = tile_{num}\ div\ n_t = 28\ div\ 8 = 3_{<10>}$$
$$t_j = tile_{num}\ mod\ n_t = 28\ mod\ 8 = 4_{<10>}$$

The requested element in found in tile #28 $[3,4]$ coordinates.

Tile offsets:

$$f_i = i\ mod\ step = 11\ mod\ 4 = 3_{<10>}$$
$$f_j = j\ mod\ step = 24\ mod\ 4 = 0_{<10>}$$

As a result, inside the tile, the requested element is found in position of $[3,0]$ coordinates.

In $L_{MO}$, element $C[11,24]$ of the original array, is stored in a 4-dimensional array in position [3][4][3][0] (figure **??**). ◇

### 3.3.3 Our approach

Tiled codes do not access array elements according to their storage order, when a linear layout has been applied. $L_{4D}$ stores array elements efficiently, as far as data locality is concerned. The storage order follows the access order by the tiled code. However, 4-dimensional arrays, that are included in the $L_{4D}$ implementation, insert large delays, because they involve four different indices and four memory references in each array reference. On the other hand, Morton order proposes a fast indexing scheme, based on dilated integer algebra. However, Morton order calls for fully recursive arrays, which fits with fully recursive applications. What's more, non-recursive codes are difficult or/and inefficient to transformed to recursive ones.

We adopt non-linear layouts and expand indexing scheme, using the dilated integer algebra, to be applied to non-recursive codes. Thus we combine data storage in the same order as they are accessed when operations are executed and a fast indexing scheme. This storage layout is presented in figure 3.5 for an $8 \times 8$ array which is split into tiles of size $4 \times 4$. The grey line shows the order by which the program sweeps the array data, while numbering illustrates the order, data are stored. We split arrays in tiles of the same size as those used by the program that scans the elements of the array. We denote the transformation of this example as "ZZ", because the inner of the tiles is scanned row-wise (in a Z-like order) and the shift from tile to tile is Z-like as well. The first letter (Z) of transformation denotes the shift from tile to tile, while the second indicates the sweeping within a tile. Sweeping data of an array $A$ in ZZ-order can be done using the code shown below:

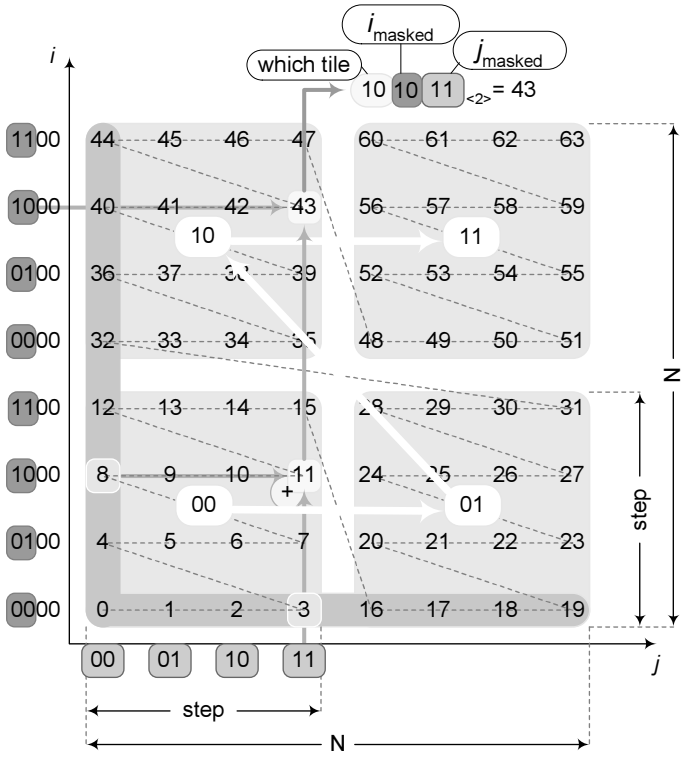



**Figure 3.5:** ZZ-transformation

```
"ZZ"  for (ii=0; ii < N; ii+=step)
        for (jj=0; jj < N; jj+=step)
          for (i=ii; (i < ii + step && i < N); i++)
            for (j = jj; (j < jj+step && j < N); j++)
              A[i, j]=...;
```

In an analogous way, the other three types of transformation are shown in figures 3.6, 3.7

and 3.8. For example, according to the "NN" transformation, both the sweeping of the array data within a tile and the shift from tile to tile are done columnwise (N-like order). The respective codes are found below.

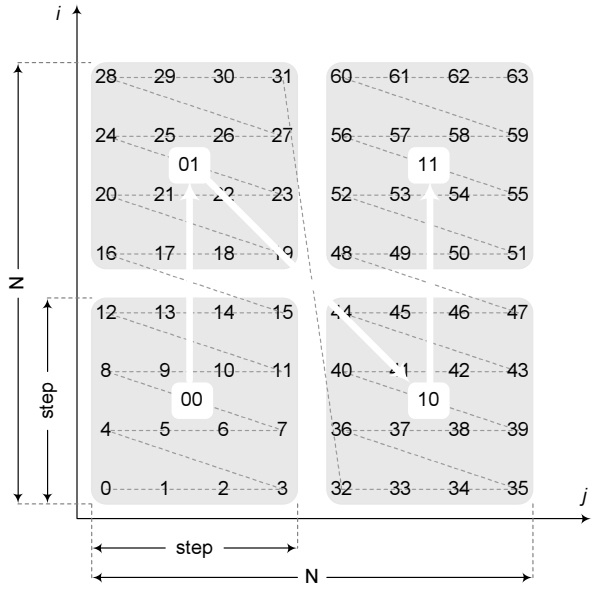



**Figure 3.6:** NZ-transformation

---

**"NZ"**
```
for (jj=0; jj < N; jj+=step)
  for (ii=0; ii < N; ii+=step)
    for (i = ii; (i < ii + step && i < N); i++)
      for (j = jj; (j < jj + step && j < N); j++)
        A[i,j]=...;
```

**"NN"**
```
for (ii=0; ii < N; ii+=step)
  for (jj=0; jj < N; jj+=step)
    for (i = ii; (i < ii + step && i < N); i++)
      for (j = jj; (j < jj + step && j < N); j++)
        A[i,j]=...;
```

**"ZN"**
```
for (jj=0; jj < N; jj+=step)
  for (ii=0; ii < N; ii+=step)
    for (i = ii; (i < ii + step && i < N); i++)
      for (j = jj; (j < jj+step && j < N); j++)
        A[i,j]=...;
```

---

Since compilers support only linear layouts (column-order or row-order) and not blocked array layouts, sweeping the array data when stored in one of the four aforementioned ways,

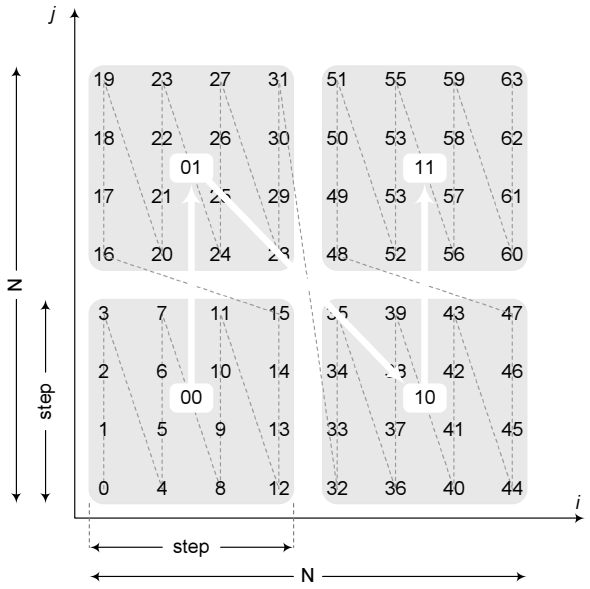



**Figure 3.7:** NN-transformation

can be done using one-dimensional arrays and indexing them through dilated integers. Morton indexing [WF99] cannot be applied, since it implies a recursive tiling scheme, whereas, in our case, tiling is applied only once (1-level tiling). Thus, instead of *oddBits* and *evenBits*, binary masks are used.

### 3.3.4 Mask Theory

Instead of bit interleaving of Morton indexing in recursive layouts, we use binary masks to convert to and from cartesian indexing in our Blocked Array Layouts. The form of the masks, for an array $A[i, j]$ of size $N_i \times N_j$ and tile size $step_i \times step_j$, is illustrated below.

The number of subsequent 0 and 1 that consist every part of the masks is defined by the functions $m_x = log(step_x)$ and $t_x = log\left(\frac{N_x}{step_x}\right)$, where *step* is a power of 2. If $N$ is not a power of 2, we round up by allocating the just larger array with $N = 2^n$ and padding the empty elements with arbitrary values, while the padding does not aggravate the execution time, since padded elements are not scanned.

Likewise in Morton indexing, every array element $A[i, j]$ can be found in the 1-dimensional array in position $[i_m + j_m] = [i_m|j_m]$, where $i_m, j_m$ are generated by $i, j$ when appropriately masked.

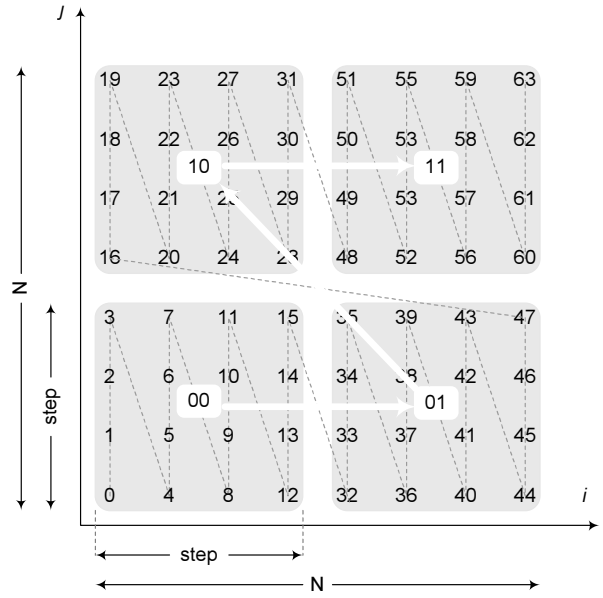



**Figure 3.8:** ZN-transformation

ZZ transformation

| | | | | |
|---|---|---|---|---|
| row index | 00..0 | 11..1 | 00..0 | 11..1 |
| column index | 11..1 | 00..0 | 11..1 | 00..0 |
| | $\leftarrow t_i \rightarrow$ | $\leftarrow t_j \rightarrow$ | $\leftarrow m_i \rightarrow$ | $\leftarrow m_j \rightarrow$ |

NZ transformation

| | | | | |
|---|---|---|---|---|
| | 11..1 | 00..0 | 00..0 | 11..1 |
| | 00..0 | 11..1 | 11..1 | 00..0 |
| | $\leftarrow t_j \rightarrow$ | $\leftarrow t_i \rightarrow$ | $\leftarrow m_i \rightarrow$ | $\leftarrow m_j \rightarrow$ |

NN transformation

| | | | | |
|---|---|---|---|---|
| row index | 11..1 | 00..0 | 11..1 | 00..0 |
| column index | 00..0 | 11..1 | 00..0 | 11..1 |
| | $\leftarrow t_j \rightarrow$ | $\leftarrow t_i \rightarrow$ | $\leftarrow m_j \rightarrow$ | $\leftarrow m_i \rightarrow$ |
| | depends on $\hookleftarrow$ tile to tile shift | | $\hookrightarrow$ depends on the inner of tiles | |

ZN transformation

| | | | | |
|---|---|---|---|---|
| | 00..0 | 11..1 | 11..1 | 00..0 |
| | 11..1 | 00..0 | 00..0 | 11..1 |
| | $\leftarrow t_i \rightarrow$ | $\leftarrow t_j \rightarrow$ | $\leftarrow m_j \rightarrow$ | $\leftarrow m_i \rightarrow$ |
| | depends on $\hookleftarrow$ tile to tile shift | | $\hookrightarrow$ depends on the inner of tiles | |

### 3.3.5 Implementation

Let us store elements in the same order, as they are swept by the instruction stream, for an array $A[i, j]$ using a code similar to ZZ-transformation of section 3.3.3. Figure 3.9 illustrates such storage order. To implement blocked layouts, indices that control the location of elements in the two-dimensional array should take non-successive values. In our example, the right values for indices $i$ and $j$ that control columns and rows respectively (array size $= 8 \times 8$, $step = 4$), are shown in the following table.

| linear enumeration of columns /rows : | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| masked values of | column $i$ : | 0 | 4 | 8 | 12 | 32 | 36 | 40 | 44 |
| array/loop indices | row $j$ : | 0 | 1 | 2 | 3 | 16 | 17 | 18 | 19 |

**Example 3.4:**

Let's assume that we are looking for the array element:

$$A[i, j] = A[2, 3]$$

To calculate the storage position of this element, the procedure that is described below and illustrated in figures **??** and **??** should be followed. We assume that array $C$ is of size $N_i \times N_j = 8 \times 8$, follows the ZZ layout, and is split to tiles of size $step_i \times step_j = 4 \times 4$.

The appropriate binary masks (section **??**) for the two-dimensional array $A$ of our example is:

for the row index $j$, $m_j = 010011$

for the column index $i$, $m_i = 101100$

If values 0-7 of linear indices (which are the values given to indices i and j according to the way compilers handle arrays so far), are filtered, using these masks (as analytically described in figure 3.10), the desired values will arise.

$$j_m = masked_{010011}(3_{<10>} = masked_{010011}(011_{<2>} = 000011_{<2>} = 3_{<10>}$$

$$i_m = masked_{101100}(2_{<10>} = masked_{101100}(010_{<2>} = 001000_{<2>} = 8_{<10>}$$

Finally, the desired element is accessed by adding these values of the two indices:

$$A[i_m + j_m] = A[8 + 3] = A[11]$$

Where $i_m$ and $j_m$ are the masked values of $i$ and $j$ indices, respectively. ◇

Practically, the masked values are the values of the first column and the first row elements (figure 3.5). So, requesting the element of the 2nd row and 3rd column, namely $A[2, 3]$, the index values in the one-dimensional array would be $i$=8, $j$=3.

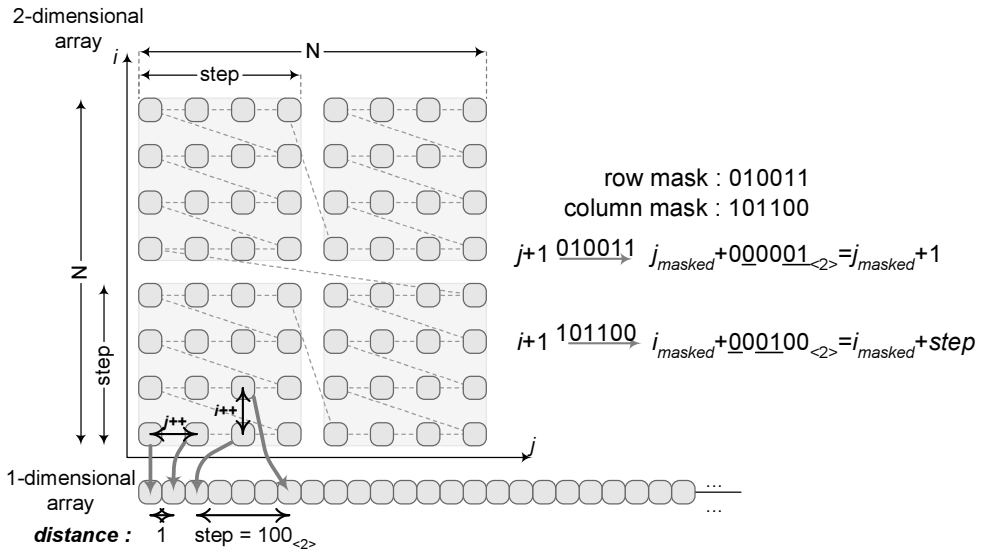



**Figure 3.9:** A 2-dimensional array converted to 1-dimensional array in ZZ-transformation

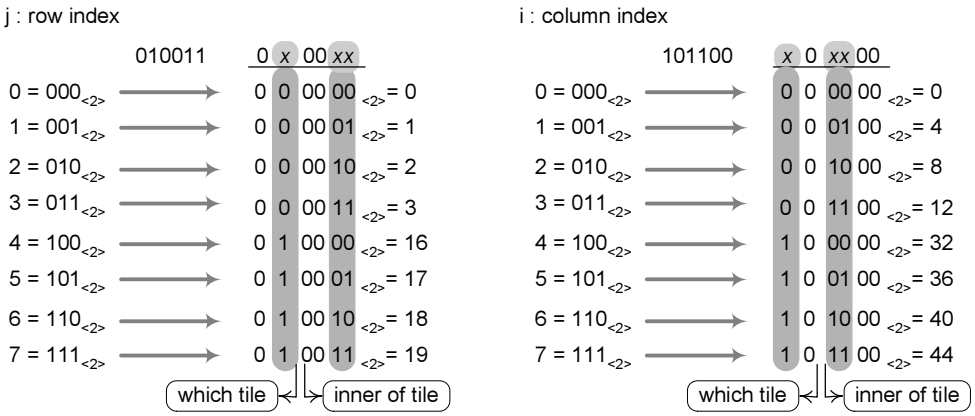



**Figure 3.10:** Conversion of the linear values of row and column indices to dilated ones through the use of masks. This is an $8 \times 8$ array with $4 \times 4$ element tiles

### 3.3.6 Example : Matrix Multiplication

According to Kandemir et al in [KRC99], the best data locality for the Matrix Multiplication code ($C[i,j]$+=$A[i,k]$*$B[k,j]$) is achieved when loops are ordered from outermost to innermost in ($i$, $k$, $j$) order and tiling is being applied to both loops $k$ and $j$. This nested loop order exploits, in the innermost loop, spatial reuse in arrays $C$ and $A$, and temporal reuse in array $A$. The column index $i$ of two of the arrays, is also kept in the outermost position.

We use a modified version of [KRC99] (as explained in section 3.1, to preserve homomorphy of tile shapes), with a nested loop of depth 6, instead of depth 5, since

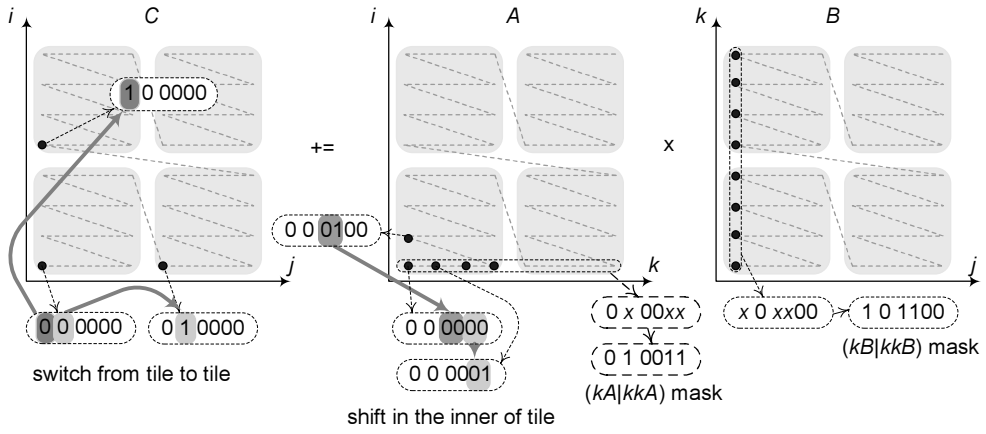



**Figure 3.11:** Matrix multiplication: $C[i,j] += A[i,k] * B[k,j]$: Oval boxes show the form of row and column binary masks. Shifting from element to element inside a tile takes place by changing the four least significant digits of the binary representation of the element position. Switching from tile to tile takes place by changing the 2 most significant digits of the binary representation of the element position.

the implementation of the mask theory is simpler in this case.

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

All three arrays $A, B, C$ are scanned according to ZZ-storage order, which matches with ZZ-transformation. Figure 3.11 depicts the access order of array elements (which is identical with their storage order), as well as indexing heuristics.

In the nested code of our example, the three inner loops $(i, j, k)$ control the sweeping within the tiles. The 4 least significant digits of the proposed binary masks are adequate to sweep all iterations within these loops, as shown in figure 3.10 and 3.11. The three outer loops $(ii, kk, jj)$ control the shifting from tile to tile, and the 2 most significant digits of the masks can define the moving from a tile to its neighboring one.

Thus, $i$ takes values in the range of $xx00$ where $x = (0 \text{ or } 1)$ and $ii$ takes values in the range of $xx00000$, so $i|ii = x00xx00$ gives the desired values for the columns

| indices : | control | of array(s) | appropriate mask |
|---|---|---|---|
| $i, ii$ : | columns | $A$ & $C$ | column mask (101100) |
| $j, jj$ : | rows | $B$ & $C$ | row mask (010011) |
| $k, kk$ : | columns | $B$ | column mask (for $kB, kkB$) |
| $k, kk$ : | rows | $A$ | row mask (for $kA, kkjA$) |

**Table 3.1:** Indexing of array dimensions, in the matrix multiplication code, when loops are nested in ($ii$, $jj$, $kk$, $i$, $j$, $k$) order : $C[i,j] += A[i,k] * B[k,j]$

of matrix $A$ and $C$. Similarly, $j$ takes values in the range of $00xx$ and $jj$ in the range of $0x0000$, so $(j|jj)$ is the desired value for rows of $B$ and $C$. Index $k$, does not control the same kind of dimension in the arrays in which it is involved, as shown in table 3.1. So, for array $A$, the proper mask is a row one, namely $kA \in 00xx$ and $kkA \in 0x0000$. On the other hand, for array $B$ the mask is a column one. Thus, $kB \in xx00$ and $kkB \in x00000$. A useful notice is that $kB = kA << logstep$ and $kkB = kkA << log\left(\frac{N}{step}\right)$, which results to the fast and easy calculation of these masks, since the second value comes out through a binary shift of the value of the first one.

```
for (ii=0; ii < ibound; ii+=iiincrement){
  itilebound=(ii|imask)+1;
  ireturn=(ibound < itilebound?ibound : itilebound);
  for (kk=0; kk < kjbound; kk+=kkjjincrement){
    ktilebound=(kk|kjmask)+1;
    kreturn=(kjbound < ktilebound?kjbound : ktilebound);
    kkB=kk << logNxy;
    for (jj=0; jj < kjbound; jj+=kkjjincrement) {
      jtilebound=(jj|kjmask)+1;
      jreturn=(kjbound < jtilebound?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];
        }
    }
  }
}
```

Where for our example,

$ibound$=column_mask=$101100_{<2>} = 44$,

$iiincrement = 100000_{<2>} = 32 = 4 \times 4 << \frac{N}{step}$,

$iincrement = 100_{<2>} = 4 = step$

and $ireturn = min\{\text{column\_mask}, ii|1100_{<2>}\}$.

---

### 3.3.7 The Algorithm to select the Optimal Layout per Array

In order to succeed in finding the best possible transformation which maximizes performance, we present a strategy, based on the algorithm presented in [KRC99], adjusted to allow for blocked array layouts. Notice that our algorithm finds the best data layout of each array, taking into account all its instances in the whole program (instances of the same array in different loop nests are also included). This way, no replicates of an array are needed.

Figure 3.12 gives a graphic representation of the following algorithm. As easily derived from the three loops of the flow chart, the worst case complexity is $O(A \cdot r \cdot D)$, where $A$ is the number of different array references, $r$ is the number of dimensions each array reference contains and $D$ is the loop depth (before tiling is being applied).

- Create a matrix $R$ of size $r \times l$, where $r = number\ of\ different\ array\ references$ and $l = nested\ loop\ depth$ (before tiling is applied). If there are two identical references to an array, there is no need for an extra line. We can just add an indication to the row representing this array, putting double priority to its optimization. For the example of matrix multiplication $R$ is:

$$R = \begin{array}{c} \begin{matrix} i & k & j \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \end{array} \begin{array}{c} \text{C**} \\ \text{A} \\ \text{B} \end{array}$$

  Notice that the Left Hand Side (LHS) array of an expression (for our example this is array C) is more important, because in every iteration the referred element is both read and written.

  Each column of $R$ represents one of the loop indices. So, elements of $R$ are set when the corresponding index controls one of the array's dimensions. Otherwise array elements are reset.

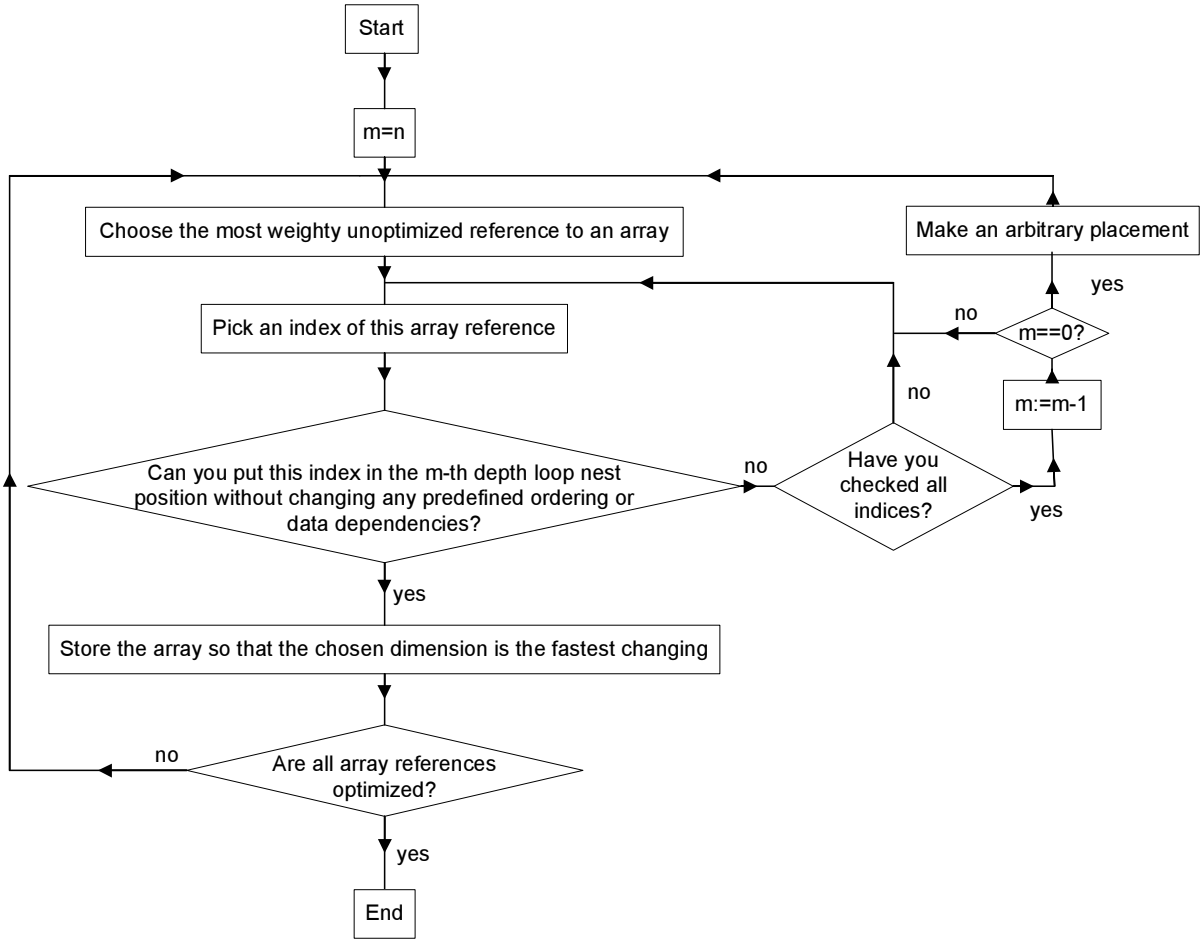



**Figure 3.12:** Flow Chart of the proposed optimization algorithm: it guides optimal nested loop ordering, which is being matched with respective storage transformation

- Optimize first the layout of the array with the greatest number of references in the loop body. (In our example this is array $C$). The applied loop transformations should be such that one of the indices that control an array dimension of it, is brought in the innermost position of the nested loop. Thus, the $C$ row of $R$ should come to the form $(x, \ldots, x, 1)$, where $x = 0$ or 1. To achieve this, swapping of columns can be involved. The chosen index is preferable be the only element of the stated dimension and should not appear in any other dimension of $C$.

  After that, in order to exploit spatial locality for this reference, array $C$ should be stored in memory such that the chosen dimension (let's say $x$-th) is the fastest changing dimension. Notice that all possible values for $x$ should be checked.

- Then, fix the remaining references to arrays by priority order. The target is to bring as many of the $R$-rows in the form $(x, \ldots, x, 1)$. So, if an array index, lets consider this is the one that controls the $y$-th dimension of array $A$, is identical to the $x$-th of $C$ then store $A$ such that its fastest changing dimension is $y$-th. If there is no such dimension

for $A$, then we should try to transform the reference so that it is brought to the form $A[*, \ldots, *, f(i_{in-1}), *, \ldots, *]$, where $f(i_{in-1})$ is a function of the second innermost loop $i_{in-1}$ and other indices except the innermost $i_{in}$, and $*$ indicates a term independent of both $i_{in-1}$ and $i_{in}$. Thus, the $R$-row for $A$ would be $(x, \ldots, x, 1, 0)$ and the spatial locality along $i_{in-1}$ is exploited. If no such transformation is possible, the transformed loop index $i_{in-2}$ is tried and so on. If all loop indices are tried unsuccessfully, then the order of loop indices is set arbitrarily, taking into account the data dependencies.

- After a complete loop transformation and a respective memory layout are found, they are stored, and the next alternative solution is tried. Among all feasible solutions, the one which exploits spatial locality in the innermost loop for the maximum number of array references, is selected.

- When the nested loop has been completely reordered, we apply tiling. In this stage, the complete layout type is defined. For each one of the array references, the dimension which was defined to be the fastest changing should remain the same for the tiled version of the program code as far as the storage of the elements within each tile is concerned. This means, that for two-dimensional arrays, if the form of the reference is $C[*, i_{in}]$, so the storing order inside the tile should be row-major, for the blocked array layout we should use the $x$Z-order (namely ZZ- or NZ-order). If the form of the reference is $C[i_{in}, *]$, so the storing order inside the tile should be column major, for the blocked array layout we should use the $x$N-order (namely ZN- or NN-order). The shifting from tile to tile, and therefore the first letter of the applied transformation, is defined by the kind of tiling we will choose. For the two dimensional example, if no tiling is applied to the dimension marked as * then we will have NZ or ZN, respectively, transformation. Otherwise, if for a nested loop of depth $n$: $(i_1, i_2, \ldots, i_n)$ the tiled form is: $(ii_1, ii_2, \ldots, ii_n, i_1, i_2, \ldots, i_n)$ (where $ii_x$ is the index that controls the shifting from one tile to the other of dimension $i_x$), then the layout should be ZZ or NN respectively. In most cases, applying tiling in all dimensions brings uniformity in the tile shapes and sizes that arrays are split and as a result, fewer computations for finding the position of desired elements are needed. The size of tiles depends on the capacity of the cache level we want to exploit.

- Alternatively to the above procedure of selecting the best loop ordering, there is another approach proposed by McKinley et al in [MCT96]. They quantify the loop order cost by the formula:

$$\frac{iterations}{cline/stride}$$

where $iterations$ = total number of loop iterations

*cline* = cache line size in data elements (number of array elements that fit in one cache line)
*stride* = loop step multiplied by the coefficient of the loop index in the array reference

If the loop index is not present in an array reference, the loop cost for this reference is 1. If non-consecutive array elements are being accessed by an array reference, and $stride > cline$, then the loop cost for this reference is equal to *iterations*. Each loop can be considered as candidate in the innermost position. The above function calculates the cost of each loop, when it is placed in the innermost position, while the remaining loops of the nest are ordered arbitrarily. The total nested loop cost comes up by multiplying the innermost loop cost with the total number of iterations of the remaining loops.

In the following example, we apply the above formula in the matrix multiplication benchmark.

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

| References | i | j | k |
|---|---|---|---|
| $A[i,k]$ : | $N^3$ | $N^2$ | $\frac{N}{8}N^2$ |
| $B[k,j]$ : | $N^2$ | $\frac{N}{8}N^2$ | $N^3$ |
| $C[i,j]$ : | $N^3$ | $\frac{N}{8}N^2$ | $N^2$ |
| total : | $2N^3 + N^2$ | $\frac{1}{4}N^3 + N^2$ | $\frac{9}{8}N^3 + N^2$ |

According to the calculated loop costs, the loop order should be chosen to be (from innermost to outermost): $j$, $k$ , $i$. This is the same result, as the initial method.

## 3.4 Summary

Low locality of references, thus poor performance in algorithms which contain multidimensional arrays, is due to incompatibility of canonical array layouts with the pattern of memory accesses from tiled codes. In this chapter, we described the effectiveness of blocked array layouts and provided with an addressing scheme that uses simple binary masks, which are based on the

algebra of dilated integers, accomplished at low cost. Both experimental and simulation results of chapter 6 illustrate the efficiency of the proposed address computation methods.

CHAPTER 4

# A Tile Size Selection Analysis

This chapter provides a theoretical analysis for the cache and TLB performance of blocked data layouts. According to this analysis, the optimal tile size that maximizes L1 cache utilization, should completely fit in the L1 cache, to avoid any interference misses. We prove that when applying optimization techniques, such as register assignment, array alignment, prefetching and loop unrolling, tile sizes equal to L1 capacity, offer better cache utilization, even for loop bodies that access more than just one array. Increased self- or/and cross-interference misses are now tolerated through prefetching. Such larger tiles also reduce lost CPU cycles due to less mispredicted branches.

So far, significant work has been done to predict cache behavior when blocked benchmarks are executed and, hence, select the optimal tile size that compromizes the minimization of capacity and interference misses. All related work selects tiles smaller than half of the cache capacity (they usually refer to L1 cache) or cache and TLB concurrently. However, blocked array layouts almost eliminate self-interference misses, while cross-interference can be easily obviated. Therefore, other factors, before negligible, now dominate cache and TLB behavior, that is, code complexity, number of mispredicted branches and cache utilization. We have managed to reduce code complexity of accesses on data stored in a blocked-wise manner by the use of efficient indexing, described in detail in chapter 3. Experimentation has proved that maximum performance is achieved when L1 cache is fully utilized. At this point, tile sizes fill the whole L1 cache. Proper array alignment obviates cross-conflict misses, while the whole cache is exploited, as all cache lines contain useful data. Such large tiles reduce the number of mispredicted branches, as well.

It is proven that blocked array layouts have really easily predictable cache behavior, which gives straightforward heuristics for choosing the best tile size. Example codes are considered to be optimized using well-known transformation techniques proposed in the literature. Apart from tiling, the most critical transformations are loop permutation, reversal and skewing, loop fusion and loop distribution, loop unrolling [Jim99], and software controlled prefetching [BAYT01],

[MG91]. Note that there is no need to apply either copying or padding in blocked array layouts.

The remainder of the chapter is organized as follows: Section 4.1 demonstrates the need of optimizing L1 cache behavior, as it is the dominant factor on performance, presenting a theoretical analysis of cache performance. A tight lower bound for cache and TLB misses is calculated, which meets the access pattern of the Matrix Multiplication kernel. Finally, concluding remarks are presented in Section 4.2.

## 4.1 Theoretical analysis

In this section we study the cache and TLB behavior, while executing the widely used benchmark of matrix multiplication. The greatest part of the analysis is devoted to the Sun UltraSPARC II architecture. This machine bears direct mapped caches, which brings a large number of conflict misses and complicates cache miss equations. Arrays are considered to be stored in memory according to the proposed blocked layouts, that is, elements accessed in consecutive iterations are found in nearby memory locations. Blocked layouts eliminate all self-conflict misses. We examine only square tiles. Such tile shapes are required for symmetry reasons, to enable the simplification of the benchmark code. As a result, while optimizing nested loop codes and selecting tile sizes, we should focus on diminishing the remaining factors that affect performance. The following analysis is an effort to identify such factors.

### 4.1.1 Machine and benchmark specifications

---

In this chapter we will continue to use the example matrix multiplication code, optimized as proposed by [KRC99], [RT98b], [AK04a]: $C+ = A * B$

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

---

Figure 4.1 illustrates groups of data elements reused in the three arrays. In-tile reuse is the amount of data reused along the execution of ($i$, $k$, $j$) loops. For example, reference $C[i, j]$ reuses data along loop $k$, which contains a whole tile line of $C$. Maximum distance reuse is the amount of data reused along the three outer loops ($ii$, $kk$, $jj$). For example, reference $A[i, k]$ reuses data along loop $jj$, which contains a whole tile of $A$.

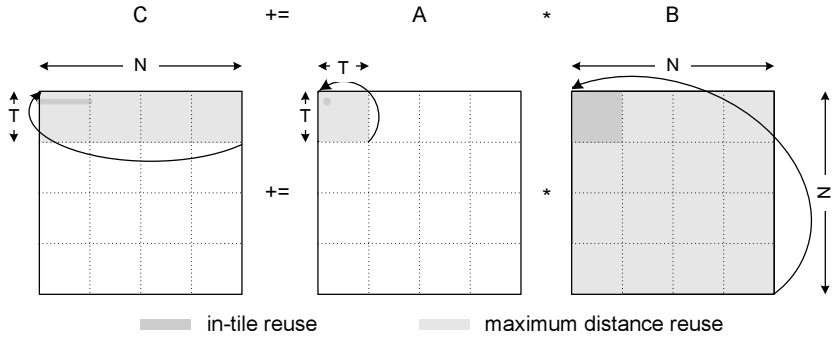



**Figure 4.1:** Reuse of array elements in the matrix multiplication code

We examine a Sun Enterprise UltraSPARC II machine with CPUs at 400MHz, each with a 16 KB direct-mapped on-chip instruction L1 cache, a 16 KB direct-mapped on-chip data L1 cache with 8 clock cycles miss penalty, a direct-mapped L2 external cache of 4 MB with 84 clock cycles miss penalty, and a 64-entry data TLB with 8 KB page size and 51 clock cycles miss penalty. Appendix A contains the symbols used in this section to represent the machine characteristics.

In the following, we offer a detailed analysis only of the most interesting area of array and tile sizes, as far as peak performance is concerned.

### 4.1.2 Data L1 misses

We study the case of a direct-mapped L1 cache. This is the most complicated case, because multiple conflict misses can easily arise and they are difficult to be counted. Array alignment should be carefully chosen, so that no more than two arrays are mapped in the same cache location, concurrently. In case of a naive data placement, there can be an extremely large number of conflict misses. Choosing the best alignment for L1 cache should not affect the number of conflict misses in L2 cache. L2 cache is larger enough than L1 and usually of higher associativity, so that by adding some multiples of $C_{L1}$ in the relevant position of arrays (that was chosen to comply with L1 cache capacity), we can find a quite good array alignment for L2 cache, without worsening mapping alignment in L1 cache. Anyway, L2 caches are usually set associative, so that they can be easily handled.

The following analysis keeps cache capacity unchanged, so that we can study the L1 cache behavior, for various array and tile sizes.

#### Direct mapped L1 caches

**a. More than just one array fit in the cache:** $N^2 < C_{L1}$

In this case, array sizes are small enough, so that all reused data of arrays $A$, $B$, $C$ fit in the cache, that is, all three arrays can exploit reuse, both in-tile and intra-tile. As a result, misses are

minimized, apart from compulsory misses (cold start misses). When $3N^2 \leq C_{L1}$, choosing the relevant storage placement of array element in main memory, which gives a specific mapping in the cache, is quite easy. However, when $2N^2 = C_{L1}$ the relevant positioning of fist data element of each array, should be really carefully chosen. As far as arrays $A$ and $C$ are considered, the minimum mapping distance of elements $A[0]$ and $C[0]$, which ensures zero conflict misses in L1 cache, is $NT - \left(T^2 - T\right)$ elements ($< C_{L1}$). This distance secures that during the reuse of $NT$ elements (one row of tiles) of array $C$, along $kk$ loop , there will not be any conflict with the $NT$ elements of array $A$. There are no conflict misses even during the last reuse of $NT$ elements of $C$, when $A$ reuses along loop $jj$ the $T^2$ elements of the last tile in the tile row (the one that is mapped almost completely inside the $A \bigcap C$ area). The $T$ elements distance in this case is enough to deter conflict misses. Figure 4.2 presents an example, where $T = \frac{N}{2}$.

For arrays $C$ and $B$ , this minimum mapping distance has to be $N^2 - \left(NT - T^2\right)$ elements, as shown in figure 4.2. In order to avoid conflict misses between $A$ and $B$ arrays, the overlapping mapping area of these two arrays in L1 cache has to be: $A \bigcap B \leq NT - T^2 + T - L_1$. In the proposed figure, this constraint is valid, since $A \bigcap B = T$ elements (this is the dark colored area of array $B$ that seems to overflow from the cache. This area is mapped to the top of the cache, where array's $A$ elements are stored - white area).

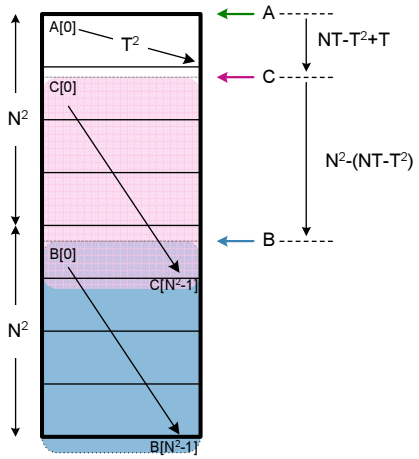



**Figure 4.2:** Alignment of arrays $A$, $B$, $C$, when $N^2 \leq C_{L1}$

In all cases, the total number of L1 misses are equal to the compulsory misses:
$M_A = M_B = M_C = \frac{N^2}{L_1}$

### b. Just one array fits in L1 cache: $C_{L1} = N^2$

In this case, L1 cache has enough capacity just for one array ($N^2 = C_{L1}$). As a result, we need to choose relative mapping distances of the three arrays in the cache carefully, in order to avoid a large number of conflict misses. As far as arrays $A$, $C$ are regarded, the minimum mapping distance of $A[0]$ and $C[0]$ has to comply with restrictions of section 4.1.2.a. The mapping distance

of arrays $C$ and $B$ (that is the mapping distance between first array elements $C[0]$ and $B[0]$) is chosen to be $T^2 - L_1$ elements. This mapping distance can deter references $B[k, j]$ and $C[i, j]$ from accessing the same cache line in the same loop iteration. According to this array alignment, the number of cache misses for each one of the three arrays is:

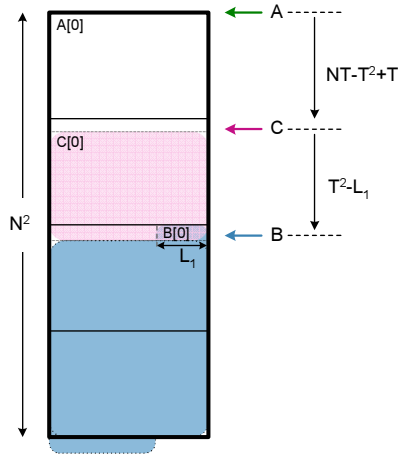



**Figure 4.3:** Alignment of arrays $A$, $B$, $C$, when $C_{L1} = N^2$

$M_A = \frac{N^2}{L_1}$ + conflict misses with array $B$
$M_B = \frac{N^2}{L_1}$ + conflict misses with arrays $A$, $C$
$M_C = \frac{N^2}{L_1}$ + conflict misses with array $B$

During the sequential iterations of loop $kk$, array references $A$ and $C$ access one row of tiles ($x$ tiles $= NT$ elements). On the contrary, array reference $B$ accesses a total of $N^2$ elements, that is the whole L1 cache. As a result, there is conflict between references to $B$ elements and references to $A$ and $C$ elements. Every conflict between $B$ and $A$ elements, removes $T^2$ elements of $A$, which are being reused along loop $jj$. Conflicts between $B$ and $C$ elements remove $T \cdot N$ elements of $C$, which are being reused along loop $kk$. The next reference to these removed elements will bring a cache miss. As a result, the missing elements will be brought again in L1 cache, to be restored. This restoring will take place place $x - 1$ times for array $A$ (for all $x$ iterations of loop $ii$, except for 1: when the conflict takes place during the last reuse of the $T^2$ elements of $A$. This means that there is no need for restoring these elements, since there will be no further reference to them - in the proposed alignment this is the first iteration of loop $ii$). Similarly, restoring of removed elements of $C$ will take place $x - 1$ times (the first iteration of loop $ii$ is being excluded, like array $A$).

Furthermore, whenever two tiles of $A$ and $B$ arrays are mapped in the same cache area (once every $ii$ loop iteration), the same cache line is being accessed at the same time for $L_1$ sequential iterations (out of $T^2$ in loops $k$, $j$). This conflict brings $L_1$ misses for both $A$ and $B$ arrays. During each $i$ iteration a whole tile line ($T$ elements) of array $B$ is being removed due to references to elements of array $A$ and vice versa. This tile line should be reloaded in the cache, which brings $\frac{T}{L_1}$ misses to both $A$ and $B$ arrays. The described conflict sequence $((L_1 + \frac{T}{L_1})$

misses) is being repeated $T$ times during execution of loop $i$, once in loops $jj$ and $kk$, and $x$ times in loop $ii$.

On the other hand, two tiles of arrays $C$ and $B$ are never mapped in the same L1 cache area. However, during execution of $kk$ loop iterations, while references to array $C$ access one tile row elements (this is $NT$ elements), they remove array $B$ elements. As a result, $2\frac{NT}{L_1}$ misses take place ($NT$ elements are being removed from the cache due to the previous iteration of loop $ii$ and $NT$ elements due to the current iteration). This iterative reloading takes place $x-1$ times (during all $x$ iterations of loop $ii$ except for the first one).

$$M_A = \frac{N^2}{L_1} + \frac{T^2}{L_1} \cdot (x-1) + \left(L_1 + \frac{T}{L_1}\right) \cdot T \cdot x = \frac{N^2}{L_1} + N \cdot \left(L_1 + \frac{T}{L_1}\right) - \frac{T^2}{L_1}$$

$$M_B = \frac{N^2}{L_1} + \left(L_1 + \frac{T}{L_1}\right) \cdot T \cdot x + \frac{2NT}{L_1} \cdot (x-1) = \frac{N^2}{L_1} + N \cdot \left(L_1 + \frac{T}{L_1}\right) + \frac{2N^2}{L_1} - \frac{2NT}{L_1}$$

$$M_C = \frac{N^2}{L_1} + \frac{NT}{L_1} \cdot (x-1) = \frac{N^2}{L_1} + \frac{NT}{L_1} \cdot \left(\frac{N}{T} - 1\right)$$

If $T = N$, the mapping distance of $C[0]$ and $A[0]$ elements, is chosen to be $T = N$ elements, to be equal to the number of reused elements during the execution of loop $k$. Moreover, the mapping distance between $B[0]$ and $C[0]$ is chosen to be $T - L_1$ elements.

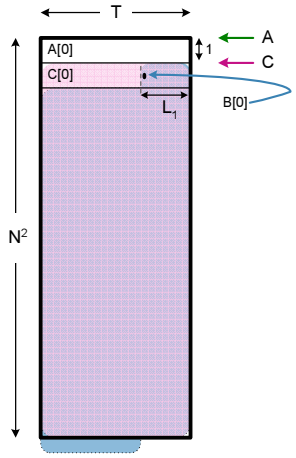



**Figure 4.4:** Alignment of arrays $A$, $B$, $C$, when $C_{L1} = N^2$, with $T = N$

Similarly:

$$M_A = \frac{N^2}{L_1} + L_1 \cdot (N-1)$$

$$M_B = \frac{N^2}{L_1} + L_1 \cdot (N-1) + \frac{N}{L_1} \cdot (N-1) + \frac{2N}{L_1} \cdot (N-1) = \frac{N^2}{L_1} + L_1 \cdot (N-1) + \frac{3N}{L_1} \cdot (N-1)$$

$$M_C = \frac{N^2}{L_1} + \frac{N}{L_1} \cdot (N-1)$$

### c. One row of tiles fits in the cache: $N^2 > C_{L1}$, $T \cdot N < C_{L1}$

In this case, at least two rows of tiles fit in L1 cache.

Data reuse can be exploited in arrays $A$ and $C$, both inner-tile reuse and whole-tile reuse, because L1 cache can accommodate $T^2$ and $T \cdot N$ elements respectively. On the other hand, for array $B$, only inner-tile reuse can be exploited. To exploit whole-tile reuse, $N^2$ ($\geq C_{L1}$) elements

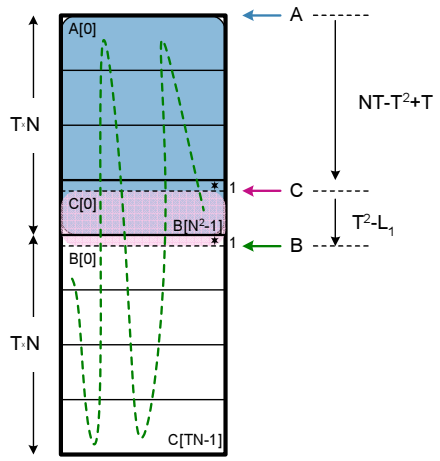



**Figure 4.5:** Alignment of arrays $A$, $B$, $C$, when $N^2 > C_{L1}$ and $T \cdot N < C_{L1}$

should be accommodated in L1 cache. The mapping distances of elements $A[0]$, $C[0]$ and $B[0]$ should be chosen as in section 4.1.2.b, when $T \neq N$. Figure 4.5 illustrates the chosen mapping.

The total number of misses is:

$M_A = \frac{N^2}{L_1}$ + conflict misses with array $B$

$M_B = \frac{N^3}{T \cdot L_1}$ + conflict misses with array $A$

$M_C = \frac{N^2}{L_1}$ + conflict misses with array $B$

Along loop $kk$, references to arrays $A$ and $C$ access elements of a whole tile row, each ($x$ tiles $= NT$ elements, by each array reference). At the same time, reference to array $B$ accesses all L1 cache lines $\frac{N^2}{C_{L1}}$ times ($N^2$ elements of array $B$ are being accessed). As a result, there is conflict between arrays $B$ and $A$ or $C$ in the cache. Every conflict removes $T^2$ elements from the cache for both arrays $A$ and $C$, which will have to be reloaded in the cache $\frac{N^2}{C_{L1}} \cdot x$ times. The exact number is $\frac{N^2}{C_{L1}} \cdot \left(x - \frac{N^2}{C_{L1}}\right) + \left(\frac{N^2}{C_{L1}} - 1\right) \cdot \frac{N^2}{C_{L1}} = \frac{N^2}{C_{L1}} \cdot (x-1)$: in $\frac{N^2}{C_{L1}}$ iterations (out of $x$ iterations of loop $kk$) one of the conflicts between $A$ and $C$ takes place during the very first use of one of the tiles of $A$. As a result, this number of misses have already been included in the number of compulsory misses This means that reloading need to be done only $\left(\frac{N^2}{C_{L1}} - 1\right)$ times in these cases.

Additionally, whenever two tiles of arrays $A$ and $B$ are being mapped in the same cache lines ($\frac{N^2}{C_{L1}}$ times along loop $kk$), $\left(L_1 + \frac{T}{L}\right)$ misses arise to both arrays $A$ and $B$, as described in section 4.1.2.b. The above sequence of misses iterates $T$ times along loop $i$, once along loop $jj$, $\frac{N^2}{C_{L1}}$ times along loop $kk$, and $x$ times along loop $ii$. There is no similar phenomenon in conflicts between arrays $A$ and $C$, due to the chosen mapping distance of arrays:

$$M_A = \frac{N^2}{L_1} + \frac{T^2}{L_1} \cdot \frac{N^2}{C_{L1}} \cdot (x-1) + \left(L_1 + \frac{T}{L}\right) \cdot T \cdot \frac{N^2}{C_{L1}} x = \frac{N^2}{L_1} + \frac{N^3}{C_{L1}} \cdot \left(L_1 + \frac{2T}{L_1}\right) - \frac{N^2 T^2}{L_1 C_{L1}}$$

$$M_B = \frac{N^3}{TL_1} + \left(L_1 + \frac{T}{L}\right) \cdot T \cdot \frac{N^2}{C_{L1}} x = \frac{N^3}{TL_1} + \frac{N^3}{C_{L1}} \cdot \left(L_1 + \frac{T}{L_1}\right)$$

$$M_C = \frac{N^2}{L_1} + \frac{T^2}{L_1} \cdot \frac{N^2}{C_{L1}} \cdot (x-1) = \frac{N^2}{L_1} + \frac{N^2 T^2}{L_1 C_{L1}} \cdot \left(\frac{N}{T} - 1\right)$$

**d. Just one tile from each one of the three arrays can fit in the cache:**

$3T^2 < C_{L1} \leq T \cdot N$ $(N^2 > C_{L1})$

In this case, three whole tiles can be accommodated in the cache, one from each one of the three arrays. The mapping distance between $A[0]$ and $C[0]$ is chosen to be $T^2 - T$ elements, so that inner-tile reuse of array $C$ will not result in any conflict with elements of array $A$ (figure 4.6). The mapping distance between elements $C[0]$, $B[0]$ has to be at least $T^2 - L_1$ elements, similarly to in section 4.1.2.b.

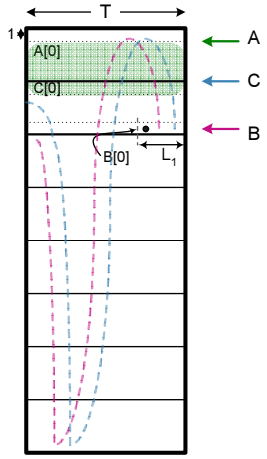



**Figure 4.6:** Alignment of arrays A, B, C, when $3T^2 < C_{L1} \leq T \cdot N$

Data reuse in arrays $B$ and $C$ along $ii$ and $kk$ loops respectively, can not be exploited, since there is not enough L1 cache capacity to accommodate $N^2$ or $T \cdot N$ elements respectively. Even if $C_{L1} = T \cdot N$, when reused elements of array $C$ along $kk$ could fit in L1 cache, references to array $B$ elements remove elements of array $C$ and avert exploitation of data reuse. Conflicts between elements of arrays $B$ and $C$, do not bring any further misses apart from averting data reuse in the cache. The total number of misses is:

$M_B = \frac{N^3}{TL_1}$ + conflict misses with array $A$

$M_C = \frac{N^3}{TL_1}$ + conflict misses with array $A$

On the other hand, data reuse of array $A$ can be exploited along $jj$ loop (this reuse includes just $T^2$ elements):

$M_A = \frac{N^2}{L_1}$ + conflict misses with array $B$, $C$

Along $jj$, while references to array $A$ access array elements which belong to the same tile, references to arrays $B$ and $C$ access all L1 cache lines ($x$ tiles are being accessed for each array $B$,$C = NT$ elements). As a result, both arrays $B$, $C$ conflict with array $A$ elements $\frac{NT}{C_{L1}}$ times. Every conflict removes $T^2$ elements of array $A$ from the cache, which should be reloaded $2 \cdot \frac{NT}{C_{L1}} \cdot x^2$ times ($\frac{NT}{C_{L1}}$ times due to conflicts with array $B$ elements and $\frac{NT}{C_{L1}}$ times due to conflicts with array $C$ elements, multiplied with $x^2$, the number of iterations along $ii$ and $kk$ loops). The precise number of iterative data reloading of array $A$ in the cache is

$2\frac{NT}{C_{L1}} \cdot \left(x - \frac{NT}{C_{L1}}\right) \cdot x + \left(2\frac{NT}{C_{L1}} - 1\right) \cdot \frac{NT}{C_{L1}} \cdot x = \frac{NT}{C_{L1}} \cdot (2x-1) \cdot x$: in $\frac{NT}{C_{L1}}$ iterations (out of $x$ iterations) of loop $kk$ one of the conflicts between arrays $A$ and $C$ takes place along the last reuse of an $A$ tile, which means that there is no need of reloading it in the cache.

Additionally, whenever two tiles of arrays $A$, $B$ are mapped in the same cache area ($\frac{NT}{C_{L1}}$ times along loop $jj$), $\left(L_1 + \frac{T}{L}\right)$ misses take place on both arrays $A$ and $B$, as described in section 4.1.2.b. These extra misses take place $T$ times along loop $i$, $\frac{NT}{C_{L1}}$ times along loop $jj$ and $x^2$ times along loops $kk$ and $ii$. We do not have similar phenomenon when conflicts between arrays $A$ and $C$ take place. This is due to chosen alignment, which is more than $T$ elements between any two concurrently accessed cache lines by references to these two arrays. Finally:

$M_A = \frac{N^2}{L_1} + \frac{T^2}{L_1}\frac{NT}{C_{L1}} \cdot (2x-1) \cdot x + \left(L_1 + \frac{T}{L}\right) \cdot T \cdot \frac{NT}{C_{L1}} \cdot x^2 = \frac{N^2}{L_1} \cdot \left(1 - \frac{T^2}{C_{L1}}\right) + \frac{N^3}{C_{L1}} \cdot \left(L_1 + \frac{3T}{L_1}\right)$

$M_B = \frac{N^3}{TL_1} + \left(L_1 + \frac{T}{L}\right) \cdot T \cdot \frac{NT}{C_{L1}} \cdot x^2 = \frac{N^3}{TL_1} + \frac{N^3}{C_{L1}}\left(L_1 + \frac{T}{L_1}\right)$

$M_C = \frac{N^3}{TL_1}$

**e. Less than three whole tiles fit in the cache:** $N^2 > C_{L1}$, $T^2 \leq C_{L1} < 3T^2$

We chose the mapping distance of $A[0]$ and $C[0]$ to be $T$ elements, that is, a whole tile row. In order to have the minimum number of conflict misses in arrays $B$ and $C$, the mapping distance of $B[0]$ and $C[0]$ should be at least $T^2 - L_1$ elements (figure 4.7).

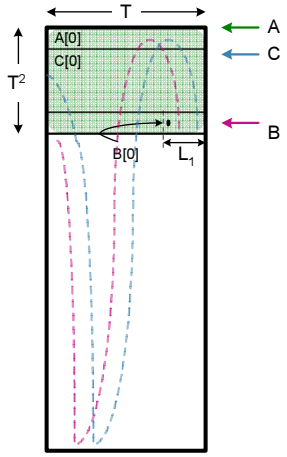



**Figure 4.7:** Alignment of arrays $A$, $B$, $C$, when $N^2 > C_{L1}$, $T^2 \leq C_{L1} < 3T^2$

Only inner-tile reuse can be exploited in three arrays along the three inner tiles. As a result:

$M_A = \frac{N^3}{TL_1}$ + conflict misses with array $B$

$M_B = \frac{N^3}{TL_1}$ + conflict misses with arrays $A$, $C$

$M_C = \frac{N^3}{TL_1}$ + conflict misses with array $B$

Conflicts between references to arrays $A$ and $B$ take place similarly to the previous case. The difference is in the frequency of conflicts: tiles of arrays $A$ and $B$ are mapped in the same area in the cache $\frac{T^2}{C_{L1}} \cdot x$ times along loop $jj$ ($\frac{T^2}{C_{L1}} \leq 1$).

Regarding conflicts between arrays $B$ and $C$, along loop $i$, a whole tile row ($T$ elements) of array $B$ are being removed from the cache due to overwriting of array $C$ elements, and vice versa. This tile row should be reloaded in the cache, which means $\frac{T}{L_1}$ additional misses for both arrays $B$ and $C$. The above sequence of conflicts reiterates $T$ times along loop $i$, $\frac{T^2}{C_{L1}} \cdot x$ times along loop $jj$ and $x^2$ times along loops $kk$ and $ii$.

$M_A = \frac{N^3}{TL_1} + L_1 \cdot T \cdot \frac{T^2 x}{C_{L1}} \cdot x^2 = \frac{N^3}{TL_1} + \frac{N^3 L_1}{C_{L1}}$

$M_B = \frac{N^3}{TL_1} + \left(L_1 + \frac{T}{L_1}\right) \cdot T \cdot \frac{T^2 x}{C_{L1}} \cdot x^2 + \frac{T}{L_1} \cdot T \cdot \left(\frac{T^2}{C_{L1}} x\right) \cdot x^2 = \frac{N^3}{TL_1} + \frac{N^3}{C_{L1}} \left(L_1 + \frac{2T}{L_1}\right)$

$M_C = \frac{N^3}{TL_1} + \frac{T}{L_1} \cdot T \cdot \frac{T^2 x}{C_{L1}} \cdot x^2 = \frac{N^3}{TL_1} + \frac{N^3 T}{C_{L1} L_1}$

### f. A whole tile fits in the cache: $N^2 > C_{L1}$, $T^2 > C_{L1} > T$

In order to minimize the number of conflicts between elements of different arrays, we chose the mapping distances to be $C[0] - A[0] = T$ elements and $B[0] - C[0] = \frac{C_{L1}}{4} - L_1$ elements (figure 4.8).

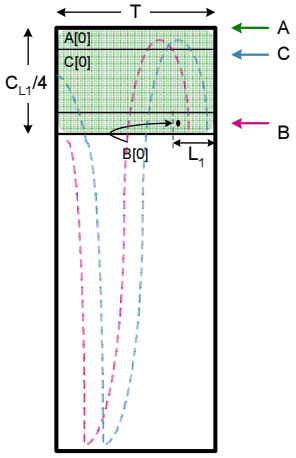



**Figure 4.8:** Alignment of arrays $A$, $B$, $C$, when $N^2 > C_{L1}$, $T^2 > C_{L1} > T$

Similarly to the case of section 4.1.2.e, whole-tile reuse can not be exploited. Additionally, inner tile reuse can not be exploited in array $B$ (along loop $i$), either. As as result, the total number of misses for each array is:

$M_A = \frac{N^3}{TL_1}$ + conflict misses with array $B$

$M_B = \frac{N^3}{L_1}$ + conflict misses with array $A$

$M_C = \frac{N^3}{TL_1}$ + conflict misses with array $B$

As a result, during conflicts between arrays $B$ and $A$ or $C$, we should not calculate the number of removed from cache elements of $B$, due to such conflict. However, a whole tile row ($T$ elements) of arrays $A$ or $C$ are being also removed, and have to be reloaded in the cache.

$M_A = \frac{N^3}{TL_1} + L_1 \cdot T \cdot \frac{T^2 x}{C_{L1}} \cdot x^2 = \frac{N^3}{TL_1} + \frac{N^3 L_1}{C_{L1}}$

$M_B = \frac{N^3}{L_1} + L_1 \cdot T \cdot \frac{T^2 x}{C_{L1}} \cdot x^2 = \frac{N^3}{L_1} + \frac{N^3 L_1}{C_{L1}}$

$M_C = \frac{N^3}{TL_1} + \frac{T}{L_1} \cdot T \cdot \frac{T^2 x}{C_{L1}} \cdot x^2 = \frac{N^3}{TL_1} + \frac{N^3 T}{C_{L1} L_1}$

**g. A tile row exceeds the capacity of the cache:** $N^2 > C_{L1}$, $T \geq C_{L1}$

Additionally to the calculated misses of section 4.1.2.f, in array $C$ inner-tile reuse (along loop $k$) can not be exploited, As a result, the total number of misses for each array is:

$M_A = \frac{N^3}{TL_1}$ + conflict misses with arrays $B$, $C$

$M_B = \frac{N^3}{L_1}$ + conflict misses with array $A$

$M_C = \frac{N^3}{L_1}$ + conflict misses with array $A$

Choosing the mapping distance of arrays $B$ and $C$ to be $B[0] - C[0] \geq L_1$ elements, there is no conflict miss between these two arrays. While reference $A[i,k]$ accesses iteratively the same element along loop $j$, references to arrays $B$ and $C$ traverse the whole cache, accessing $T$ sequential elements. Each of them conflicts with an array $A$ element $\frac{T}{C_{L1}}$ times, and $L_1$ misses arise. This process goes over $T^2 \cdot x^3$ times along loops $k$, $i$, $jj$, $kk$ and $ii$. As a result, the number of conflict misses (for each one of the involved arrays) is $\frac{T}{C_{L1}} \cdot L_1 \cdot T^2 x^3$:

$M_A = \frac{N^3}{TL_1} + 2\frac{T}{C_{L1}} \cdot L_1 \cdot T^2 x^3 = \frac{N^3}{TL_1} + \frac{2N^3 L_1}{C_{L1}}$

$M_B = \frac{N^3}{L_1} + \frac{T}{C_{L1}} \cdot L_1 \cdot T^2 x^3 = \frac{N^3}{L_1} + \frac{N^3 L_1}{C_{L1}}$

$M_C = \frac{N^3}{L_1} + \frac{T}{C_{L1}} \cdot L_1 \cdot T^2 x^3 = \frac{N^3}{L_1} + \frac{N^3 L_1}{C_{L1}}$

### Summary of L1 cache misses in direct mapped caches

Table 4.1 summarizes the total number of cache misses $M_1$ ($= M_A + M_B + M_C$) in a L1 direct mapped cache, for all possible array and tile sizes.

| **requirements** | $\mathbf{M_1}$ |
|---|---|
| $N^2 < C_{L1}$ | $\frac{3N^2}{L_1}$ |
| $N^2 = C_{L1}$, $T \neq N$ | $\frac{6N^2}{L_1} + 2NL_1 - \frac{T^2+NT}{L_1}$ |
| $N^2 = C_{L1}$, $T = N$ | $\frac{7N^2}{L_1} + 2NL_1 - \left(2L_1 + \frac{4N}{L_1}\right)$ |
| $N^2 > C_{L1}$, $T \cdot N < C_{L1}$ | $\frac{2N^2}{L_1}\left(1 - \frac{T^2}{C_{L1}}\right) + \frac{N^3}{TL_1} +$ $+ \frac{N^3}{C_{L1}}\left(2L_1 + \frac{4T}{L_1}\right)$ |
| $3T^2 < C_{L1} \leq T \cdot N$ | $\frac{N^2}{L_1}\left(1 - \frac{T^2}{C_{L1}}\right) + \frac{2N^3}{TL_1} +$ $+ \frac{N^3}{C_{L1}}\left(2L_1 + \frac{4T}{L_1}\right)$ |
| $T^2 \leq C_{L1} < 3T^2$ | $\frac{3N^3}{TL_1} + \frac{N^3}{C_{L1}}\left(2L_1 + \frac{3T}{L_1}\right)$ |
| $T^2 > C_{L1} > T$ | $\frac{N^3}{L_1} + \frac{N^3}{TL_1}\left(2 + \frac{T^2}{C_{L1}}\right) + \frac{2N^3 L_1}{C_{L1}}$ |
| $T \geq C_{L1}$ | $\frac{N^3}{TL_1} + \frac{2N^3}{L_1} + \frac{4N^3 L_1}{C_{L1}}$ |

**Table 4.1:** Calculation formulas for direct-mapped L1 cache misses

The above formulas are being graphically depicted in figure 4.9, for different problem sizes and for the cache characteristics of the UltraSPARC II machine.

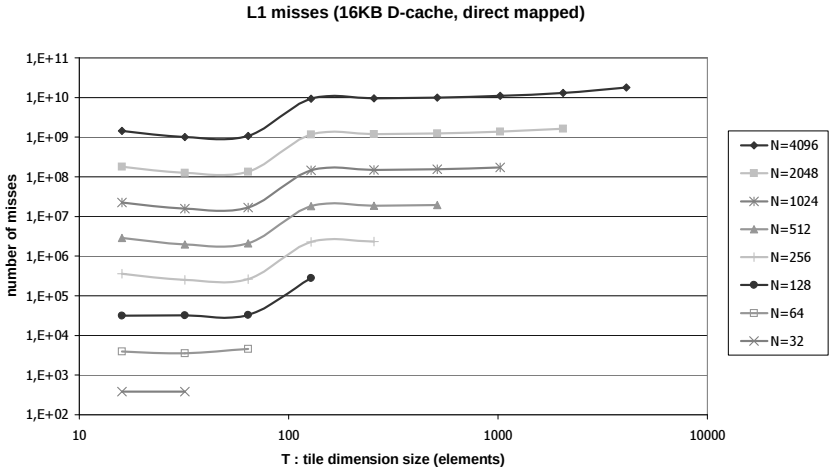



**Figure 4.9:** Number of L1 cache misses for various array and tile sizes in direct mapped caches, when the described in section 4.1.2 alignment has been applied (UltraSPARC II architecture)

The number of L1 cache misses drop to a minimum, for all array sizes, when $T = 32$, where three whole tile fit in the cache (one for each of the three arrays). When $T = 64$ ($T = \sqrt{C_{L1}}$), just one tile fit in the cache, however the whole cache capacity is being exploited, At this point, the number of cache misses slightly increases, but it is still quite close to its minimum value.

### Set Associative L1 caches

In case of set associative caches, apart from capacity misses, neither self- nor cross-interference misses arise. Even 2-way set associativity is enough for kernel codes such as matrix multiplication, where data elements from three different arrays are retrieved. Array alignment should be carefully chosen, so that no more than two arrays are mapped in the same cache location, concurrently. For this purpose, the starting mapping distances of arrays (that is, elements $A[0]$, $B[0]$, $C[0]$), should be chosen to be from $L_1$ to $T^2$ elements.

#### a. More than just one array fit in the cache: $N^2 < C_{L1}$:

In this case, all three arrays can exploit reuse, both in-tile and intra-tile. For example, array $C$ reuses one tile row along loop $j$ (in-tile reuse) and a whole row of tile along loop $jj$ (intra-tile reuse). In both cases, the working set fit in L1 cache. As a relult, for array $C$:

$M_C = x^2 \cdot \frac{T^2}{L_1} = \left(\frac{N}{T}\right)^2 \frac{T^2}{L_1} = \frac{N^2}{L_1}$

Similarly, $M_A = M_B = \frac{N^2}{L_1}$

**b. More than just one row of tiles fit in the cache:** $N^2 \geq C_{L1}$, $T \cdot N < C_{L1}$

For arrays $A$, $C$ reuse along loops $kk$ and $jj$ respectively can be exploited, as there is enough L1 cache capacity to hold $T^2$ and $T \cdot N$ elements respectively. As in section 4.1.2:

$M_A = M_C = \frac{N^2}{L_1}$

On the other hand, for array $B$ only in-tile reuse can be exploited, as loop $ii$ reuses $N^2$ elements, and the cache capacity is not adequate to hold them. As a result, each $ii$ iteration will have to reload the whole array in the cache:

$M_B = x^3 \cdot \frac{T^2}{L_1} = \left(\frac{N}{T}\right)^3 \frac{T^2}{L_1} = \frac{N^3}{TL_1}$

In case that $N^2 = C_{L1}$, $T = N$, there is in fact no tiling, so reuse takes place in loops $k$ and $j$ for arrays $A$ and $C$, containing just 1 element, one row of the array ($N$ elements) respectively. As a result, reuse is exploited as above. However, the reference to array $B$ reuses $N^2$ elements (the whole array) along loop $i$. In each iteration of $i$, two rows of $B$ elements ($2N$ elements) have been discarded from the cache, due to references to arrays $A$ and $C$. That is:

$M_B = \frac{N^2}{L_1} + (N-1) \cdot \frac{2N}{L_1}$

**c. Just one tile from each one of the three arrays can fit in the cache:** $N^2 > C_{L1}$, $3T^2 < C_{L1} \leq T \cdot N$**:**

Three whole tiles fit in the cache, one for each of the three arrays. For arrays $B$, $C$ reuse along loops $ii$ and $jj$ respectively can not be exploited, as there is not enough L1 cache capacity to hold $N^2$ and $T \cdot N$ elements respectively. The number of misses are:

$M_B = M_C = \frac{N^3}{TL_1}$

On the other hand, reuse along loop $kk$ for array $A$ can be exploited (only $T^2$ elements are included):

$M_A = \frac{N^2}{L_1}$

**d. Less than three whole tiles fit in the cache:** $N^2 > C_{L1}$, $T^2 \leq C_{L1} < 3T^2$**:**

There is enough space for at most two whole tiles. Only in-tile reuse can be exploited in the arrays along the three inner loops. Thus:

$M_A = M_B = M_C = \frac{N^3}{TL_1}$

**e. A whole tile fits in the cache:** $N^2 > C_{L1}$, $T^2 > C_{L1} > T$**:**

As in the previous case, no whole-tile reuse can be exploited. Additionally, in array $B$, in-tile reuse (along loop $i$) can not be exploited, either. Therefore, the total number of misses for each array is:

$M_A = M_C = \frac{N^3}{TL_1}$

$M_B = \frac{N^3}{L_1}$

**Summary of the Data L1 misses in set associative caches:**

Table 4.2 summarizes the total number of Data L1 cache misses $M_1$ for a set associative cache and different problem sizes, and figure 4.10 illustrates the graphic representation of these formulas, for the cache characteristics of the Xeon DP architecture (appendix B).

| requirements | $M_1$ |
|---|---|
| $N^2 < C_{L1}$ | $3\frac{N^2}{L_1}$ |
| $N^2 \geq C_{L1},\ T \cdot N < C_{L1}$ | $2\frac{N^2}{L_1} + \frac{N^3}{TL_1}$ |
| $N^2 = C_{L1},\ T = N$ | $5\frac{N^2}{L_1} - \frac{N)}{L_1}$ |
| $N^2 > C_{L1},\ 3T^2 < C_{L1} < T \cdot N$ | $\frac{N^2}{L_1} + 2\frac{N^3}{TL_1}$ |
| $T^2 \leq C_{L1} < 3T^2$ | $3\frac{N^3}{TL_1}$ |
| $T^2 > C_{L1} > T$ | $2\frac{N^3}{TL_1} + \frac{N^3}{L_1}$ |
| $T \geq C_{L1}$ | $\frac{N^3}{TL_1} + 2\frac{N^3}{L_1}$ |

**Table 4.2:** Formulas for set associative Data L1 misses

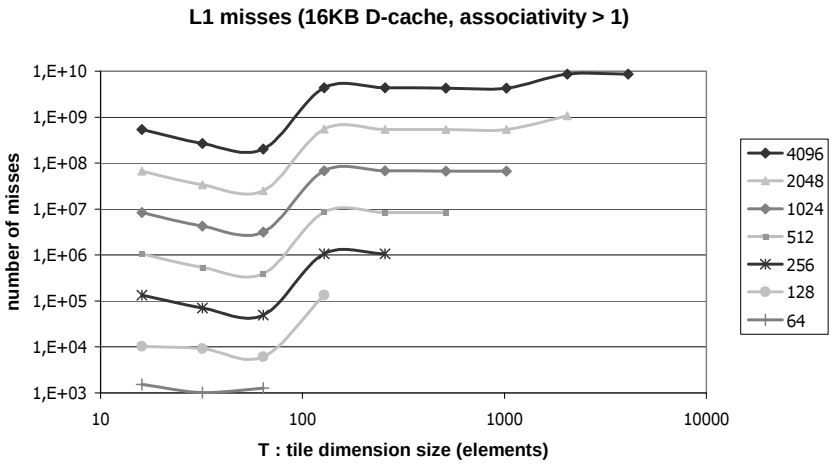


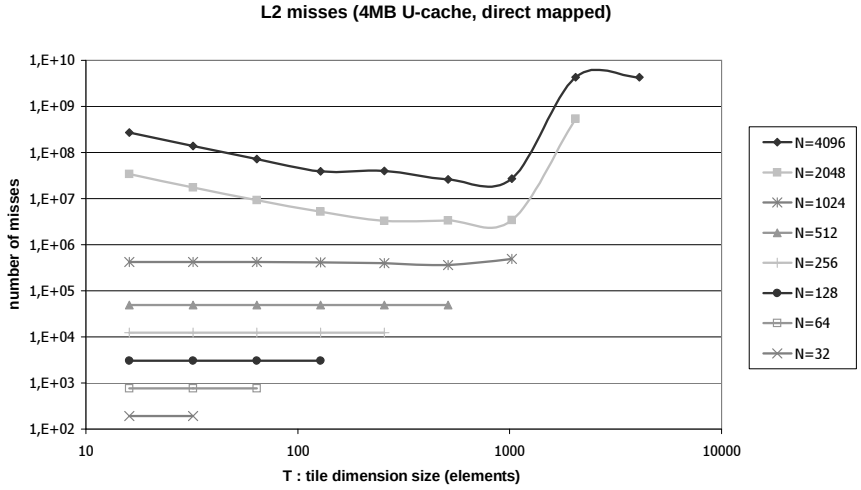

**Figure 4.10:** Number of L1 cache misses for various array and tile sizes in set associative caches (Xeon DP architecture)

Figure 4.10 illustrates the graphic representation of the set-associative case formulas. In direct mapped caches, many more conflict misses arose, and they were highly depended on array alignment. In both cases (set associative and direct mapped caches), the final conclusion is the same: L1 cache misses increase sharply when the working set, reused along the three innermost loops, overwhelms the L1 cache. That is, the tile size overexceeds the L1 cache capacity ($C_{L1}$), and no reuse can be exploited for at least one array. For direct mapped caches, when the tile size is equal to the L1 cache capacity ($T^2 = C_{L1}$), the number of cache misses increases compared to

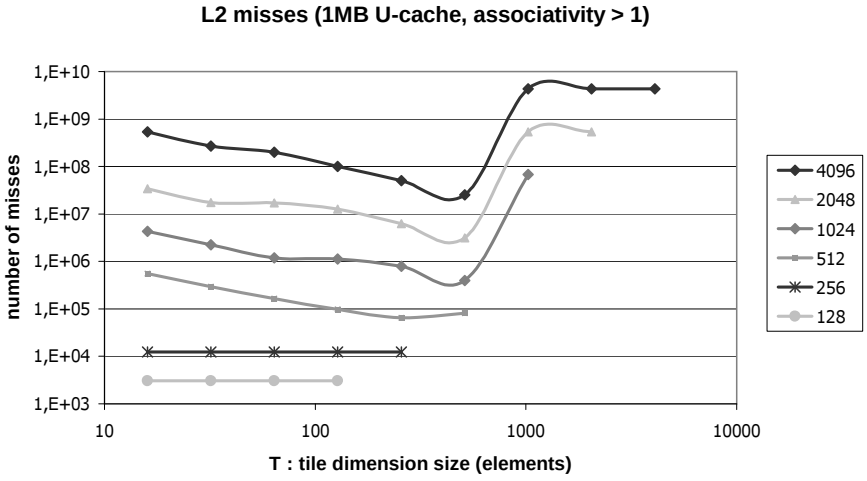
the just smaller tile size, however this increase is not significant and, as proved in the following sections, it is counterbalanced by the decrease of misses in other levels of memory hierarchy.

### 4.1.3 L2 misses

This cache level has similar behaviour as the L1 cache. As a result, we skip the detailed analysis and provide only with the corresponding graphs. Figure 4.11(a) presents the number of L2 cache misses in case of a direct mapped cache, with size equal the L2 cache of the Sun UltraSPARC platform. Figure 4.11(b) presents the number of L2 cache misses in case of a set associative cache, with size equal the L2 cache of the Intel Xeon platform (table 5.1). In the direct mapped case, array alignment chosen for the respective L1 direct mapped cache can easily meet the requirements of L2 cache, too, by interjecting some multiples of $C_{L1}$ among the mapping positions of different arrays (chosen in section 4.1.2).




(a) direct mapped caches




(b) set associative caches

**Figure 4.11:** Number of L2 direct mapped cache misses for various array and tile sizes

We note that L2 cache is unified (for data and instructions). However, the number of misses hardly increases (less than 1%) compared to an equal-sized data cache, when caches are large, like the one of the Xeon or the UltraSPARC platforms.

The number of L2 misses, for all array sizes, are minimized for $T^2 = C_{L2}$, when the whole cache is been used and a whole tile fits in the cache so that tile-level reuse can be exploited. However, L1 misses are 1 order of magnitude more than L2 misses. As a result, the L1 misses dominate in the total memory behaviour, as illustrated in figure 4.13(a).

### 4.1.4 Data TLB misses

This cache level is usually fully associative. So, there is no need to take care of array alignment.

**a. The addresses of three whole tile rows fit in the TLB:** $N^2 \geq E \cdot P$, $3T \cdot N < E \cdot P$

When $N^2 \geq E \cdot P$ but $3T \cdot N < E \cdot P$, the addresses of three rows of tiles (one of each array) fit in the TLB. For arrays $A$ and $C$, reuse along loops $kk$ and $jj$, respectively, can be exploited, as there is enough space in the TLB to hold $\frac{T^2}{P}$ and $\frac{T \cdot N}{P}$ addresses respectively. So, the total number of TLB misses is equal to the compulsory (cold start) misses, that is $\frac{N^2}{P}$ for each one of the two arrays.

On the other hand, reuse for array $B$ can not be fully exploited, because there is not adequate cache space for all $\frac{N^2}{P}$ addresses. Along loop $ii$, the reused addresses are not stored in the TLB any more and they should be reloaded. However, there is no conflict with arrays $A$, $C$, because the LRU (least recently used) pages are those of $B$.

$M_A = \frac{N^2}{P}$, $M_B = x \times \frac{N^2}{P} = \frac{N^3}{T \cdot P}$, $M_C = \frac{N^2}{P}$

**b. The addresses of more than just one row of tiles fit in the TLB:** $N^2 > E \cdot P$, $T \cdot N < E \cdot P < 3T \cdot N$

In this case, reuse along loop $jj$ can not be exploited for array $C$, as there is not enough space for $\frac{T \cdot N}{P}$ addresses.

$M_A = \frac{N^2}{P}$, $M_B = \frac{N^3}{T \cdot P}$, $M_C = x \times \frac{N^2}{P} = \frac{N^3}{T \cdot P}$

**c. The addresses of more than just one tile fit in the TLB:** $N^2 > E \cdot P$, $T^2 < E \cdot P < 3T^2$

In this case, although there is enough space for the $\frac{T^2}{P}$ addresses of array $A$ to be hold in the TLB in order to be reused along $kk$, when the cache is full, the critical addresses of $A$ are not the most frequently used, so they are erased from the cache.

$M_A = \frac{N^3}{T \cdot P}$, $M_B = \frac{N^3}{T \cdot P}$, $M_C = \frac{N^3}{T \cdot P}$

**d. The addresses of less than a whole tile fit in the TLB:** $N^2 > E \cdot P$, $T^2 > E \cdot P > T$

In this case we lose reuse of array $B$ along loop $i$, because the cache capacity is not adequate to store all $\frac{T^2}{P}$ addresses:

$M_A = \frac{N^3}{T \cdot P}$, $M_B = \frac{N^3}{P}$, $M_C = \frac{N^3}{T \cdot P}$

**e. The addresses of less than a whole tile row fit in the TLB:** $N^2 > E \cdot P$, $T \geq E \cdot P$

In this case reuse of array $C$ along loop $j$ can not be exploited, because the cache capacity is not adequate to store $\frac{T}{P}$ addresses:

$M_A = \frac{N^3}{T \cdot P}$, $M_B = \frac{N^3}{P}$, $M_C = \frac{N^3}{P}$

**Summary of the TLB misses**

Table 4.3 summarizes the total number of Data TLB misses $M_{TLB}$ for all problem sizes.

| **requirements** | $\mathbf{M}_{TLB}$ |
|---|---|
| $N^2 < E \cdot P$ | $3\frac{N^2}{P}$ |
| $3T \cdot N \leq E \cdot P$ | $2\frac{N^2}{P} + \frac{N^3}{T \cdot P}$ |
| $T \cdot N < E \cdot P < 3T \cdot N$ | $\frac{N^2}{P} + 2\frac{N^3}{T \cdot P}$ |
| $T^2 < E \cdot P < 3T^2$ | $3\frac{N^3}{T \cdot P}$ |
| $T^2 > E \cdot P > T$ | $2\frac{N^3}{T \cdot P} + \frac{N^3}{P}$ |
| $T > E \cdot P$ | $\frac{N^3}{T \cdot P} + 2\frac{N^3}{P}$ |

**Table 4.3:** Formulas for Data TLB misses

According to the above analysis, the number of Data TLB misses has the form of figure 4.12.

The number of TLB misses for all array sizes, for an example of 64 entries (as the size of both Xeon and UltraSPARC's TLB is), are minimized when $T = 256$, as the addresses of pages for a whole tile fit in the TLB entries, so that tile-level reuse can be exploited.

### 4.1.5 Mispredicted branches

The extra loop levels bring an increased number of mispredicted branches. In order to avoid a large performance degradation due to branch prediction faults, we incorporate the number of possible miss-predicted branches into our model.

We use a modified version of Vera's function [Ver03]. We consider $n$ nested loops, with executed iterations $I = \{I_1, \ldots, I_n\}$ respectively. Since current branch predictors may miss-speculate when loops finish their execution, the number of expected mispredicted branches is:

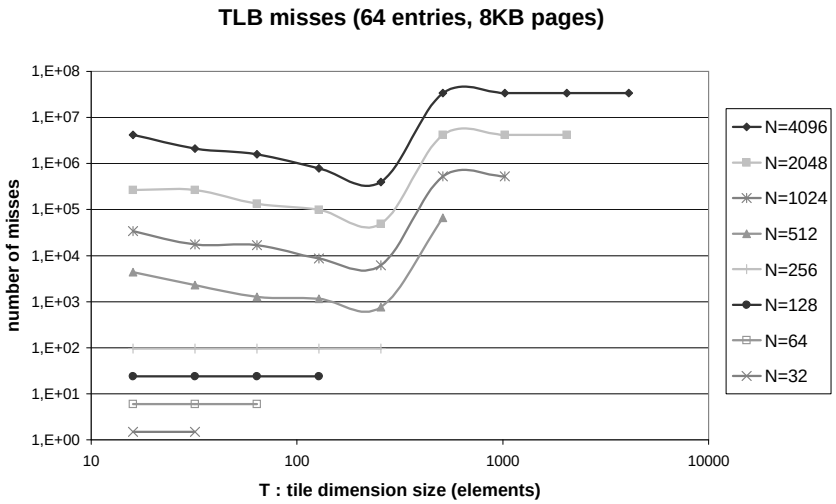



**Figure 4.12:** Number of TLB misses for various array and tile sizes

$M_{br} = \sum_{j=1}^{j \leq n} \prod_{i=1}^{i<j} I_i$

In the examined matrix multiplication example (using the proposed efficient indexing), we have a 6-nested loop with $I = \left\{\frac{N}{T}, \frac{N}{T}, \frac{N}{T}, T, T, T\right\}$. As a result the expected number of mispredicted branches is:

$M_{br_6} = 1 + \frac{N}{T} + \left(\frac{N}{T}\right)^2 + \left(\frac{N}{T}\right)^3 + \left(\frac{N}{T}\right)^3 \times T + \left(\frac{N}{T}\right)^3 \times T^2$

Notice that in comparison with a 5-nested loop, where tiling is not applied in loop $i$, there is not much difference in the number of mispredicted branches. Therefore, the increased depth of the nested loop, as enforced by the efficient indexing, does not bring any worthwhile performance degradation:

$\Delta M = M_{br_6} - M_{br_5} = \left(\frac{N}{T}\right)^3$

For $N = 1024$ and $T = 32$, the increase percentage $\left(\frac{\Delta M}{M_{br_5}}\right)$ is less than 0.01%.

### 4.1.6 Total miss cost

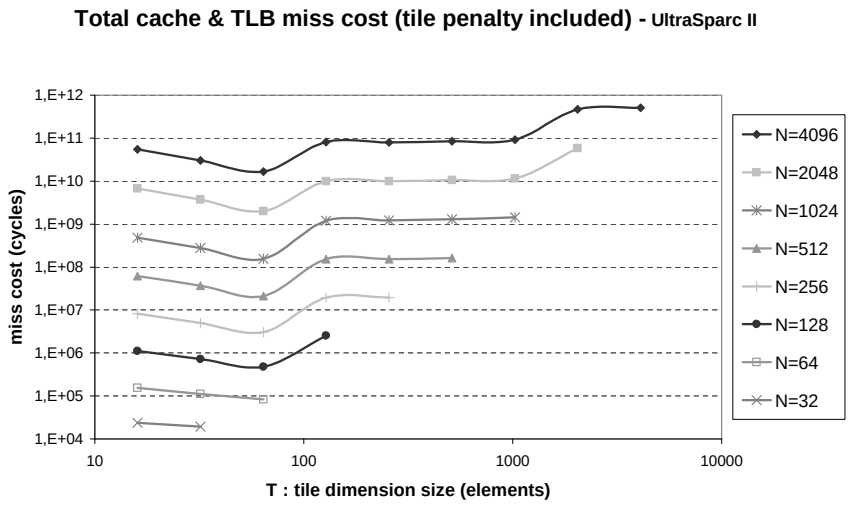
Taking into account the miss penalty of each memory level, as well as the penalty of mispredicted branches, we derive the total miss cost of figure 4.13(a). Notice that we count only the data L2 misses, although the L2 cache is unified for both instructions and data. Increase in the number of L2 misses due to instruction misses is really negligible.

Figure 4.13(a) makes clear that L1 misses dominate cache and, as a result, total performance in the UltraSPARC II architecture. Maximum performance is achieved when $T = 64$, which is the optimal tile size for L1 cache (the maximum tile that fits in L1 cache). L1 cache misses are more than one order of magnitude more than L2 misses and three orders of magnitude more than TLB misses. Notice that the UltraSPARC II architecture bears quite a large L2 cache (4Mbytes), which reduces the number of L2 misses significantly, and leaves L1 cache to dominate total performance. Thus, even though L1 misses cost fewer clock cycles, they are still

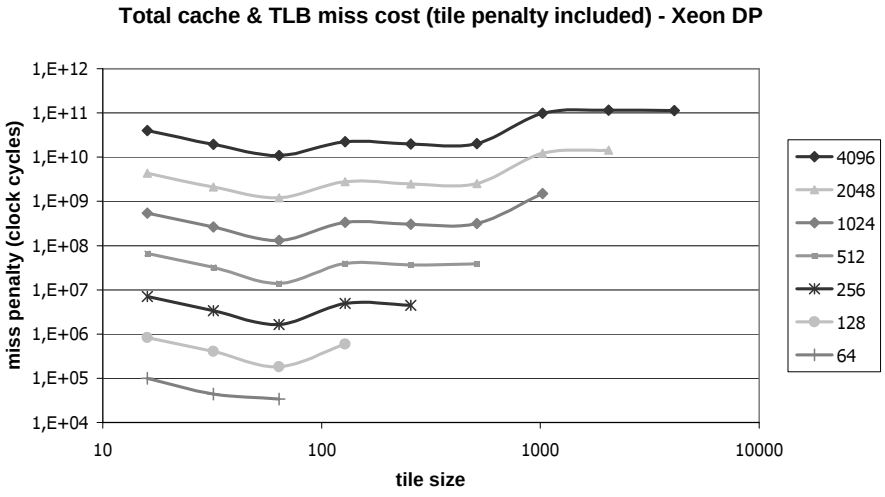
the most weighty factor.

The studied architecture has direct mapped caches, which brings a large number of conflict misses, as proved in section 4.1.2 and 4.1.3. When caches are multi-way set associative (even 2-way set associative is enough for our case), conflict misses are eliminated, and reuse can be exploited as far as cache capacity makes it permissible. Cache miss equations follow the TLB analysis of section 4.1.4 (which is a fully associative cache, too).

Figure 4.13(b) illustrates the cache and TLB performance of a Xeon DP architecture, which has multi-way set associative caches. More details about its architecture characteristics can be found in appendix B.




(a) direct mapped caches (UltraSPARC II architecture)




(b) set associative caches (Xeon DP architecture)

**Figure 4.13:** The total miss cost for various array and tile sizes

## 4.2 Summary

A large amount of related work has been devoted to the selection of optimal tile sizes and shapes, for numerical nested loop codes where tiling has been applied. In this chapter, we have found theoretically that, when blocked layouts are used, in direct mapped caches, L1 cache misses dominate overall performance. Prefetching in combination with other code optimization techniques, set optimal tiling dimension to be $T = \sqrt{C_{L1}}$, where decrease of L2 and TLB misses as well as mispredicted branches counterbalance a slight increase of L1 cache misses (compared to their minimum value).

CHAPTER 5

# Simultaneous Multithreading

Simultaneous multithreading (SMT) has been proposed to improve system throughput by overlapping multiple (either multi-programmed or explicitly parallel) threads on a single wide-issue processor. Recent studies have demonstrated that heterogeneity of simultaneously executed applications can bring up significant performance gains due to SMT. However, the speedup of a single application that is parallelized into multiple threads, is often sensitive to the efficiency of synchronization and communication mechanisms between its separate, but possibly dependent, threads. Moreover, as these separate threads tend to put pressure on the same architectural resources, no significant speedup can be achieved.

This chapter describes the characteristics of the SMT architecture and discusses so far proposed methods to exploit idle processor cycles through multithreading features. It also presents synchronization and implementation issues. Finally, it explores the CPI limits of the Intel Xeon processor enabled with hyperthreading technology.

The remainder of this chapter is organized as follows: Section 5.1 describes the spacial case of SMT platforms. Section 5.2 reviews related prior work. Section 5.3 deals with implementation aspects of software techniques to exploit hardware multithreading. Section 5.4 explores the performance limits and TLP-ILP tradeoffs, by considering a representative set of instruction streams. Finally, we conclude with section 5.5.

## 5.1 Introduction

One approach to maintain high throughput of processors despite the large relative memory latency has been Simultaneous Multithreading (SMT). SMT is a hardware technique that allows a processor to issue instructions from multiple independent threads, to the functional units of a superscalar processor, in the same cycle. This enables the processor to use all available parallelism to fully utilize the execution resources of the machine. Through this increased competition, SMT decreases wasted issue slots and increases flexibility [TEE+96].

Along with multithreading, prefetching is one of the most popular techniques for tolerating the ever-increasing memory wall problem. In contrast to multithreading, in which instructions from different threads are executed, when the running thread encounters a cache miss, prefetching tolerates latency by anticipating what data is needed and moving it to the cache ahead of time. As long as prefetching begins early enough and the data is not evicted prior to its use, memory access latency can be completely hidden.

Since the concept of SMT processors was first introduced in research publications during the past years, there have been proposed two main techniques to utilize the multiple hardware contexts of the processors for improving performance of a single program: thread-level parallelism (TLP) and speculative precomputation (SPR). With TLP, sequential codes are parallelized so that the total amount of work is decomposed into independent parts which are assigned to a number of software threads for execution. In SPR, the execution of programs is facilitated with the introduction of additional threads, which speculatively prefetch data that is going to be used by the sibling computation threads in the near future, thus hiding memory latencies and reducing cache misses [WWW+02], [KLW+04], [TWN04]. However, we have to be really careful when implementing parallel threads on SMT architectures. In the multithreading mode, processor units (reservation station, renaming registers, fetch/decode units) are either shared or partitioned between logical processors and resource contention can increase. Even an idle thread may lead to starvation of the dispatch unit, due to the static partitioning of the fetch and decode units [SU96].

Most previous work demonstrated simulation results on SMT model processors. Experiments on real machines equipped with HT-enabled processors [BP04], [KLW+04], [TT03], did not report any significant speedups, when parallelizing a single application. The benefit of multithreading on SMT architectures depends on the application and its level of tuning [MCFT99]. For example, optimized blocked matrix multiply has good register and cache reuse. With a high level of tuning to expose register and cache locality, additional threads on the same physical processor will harm performance. A naive matrix multiply has poor locality of references and hence benefits from TLP.

## 5.2 Related Work

Simultaneous multithreading [HKN+92], [TEE+96], [TEL95] is said to outperform previous execution models because it combines the multiple-instruction-issue features of modern superscalar architectures with the latency-hiding ability of multithreaded ones: all hardware contexts are simultaneously active, competing in each cycle for the available shared resources. This dynamic sharing of the functional units allows simultaneous multithreading to substantially increase throughput, attacking the two major impediments to processor utilization - long latencies and limited per-thread parallelism. However, the flexibility of SMT comes at a cost. When multiple

threads are active, the static partitioning of resources (instruction queue, reorder buffer, store queue) affects benchmarks with relative high instruction throughput. Static partitioning, in the case of identical thread-level instruction streams, limits performance, but mitigates significant slowdowns when non-similar streams of microinstructions are executed [TT03]. Moreover, multiple-issue processors are prone to instruction cache thrashing.

Cache prefetching [LM96], [LM99], [MG91] is a technique that reduces the observed latency of memory accesses by bringing data into the cache before it is accessed by the CPU. However, as processor throughput improves due to memory latency tolerance, prefetching suffers from certain disadvantages. Firstly, use of memory bandwidth is increased since prefetching increases memory traffic. Memory requirements are also increased as much more cache accesses are generated and thus, more data are being brought in the caches for future use. Finally, the number of executed instructions is increased, filling the pipeline cycles with time consuming operations [BAYT04].

Hardware prefetching schemes [Che95], [JG97], [Jou90] are advantageous in that they do not require complicated compiler support, they do not require additional instructions to be added to a program, and they are capable of exploiting information that is available only at run time. Most forms of hardware prefetching rely on the assumption that future cache miss patterns can be predicted from past memory access behavior.

The software approach [MG91] has limited negative impact on bus traffic and introduces less conflicts with the working set than the hardware approach. However, the overhead due to the extra prefetch instructions and associated computations is substantial in the software directed approach and can offset the performance gain of prefetching. The relative effectiveness of prefetching is slightly degraded by the increase of memory latencies, with the software prefetching suffering the less. The performance of hardware and software prefetching was found roughly equivalent in [CB94], but hardware prefetching offers performance improvements at a considerably higher price, both in terms of additional bandwidth requirements and total system cost.

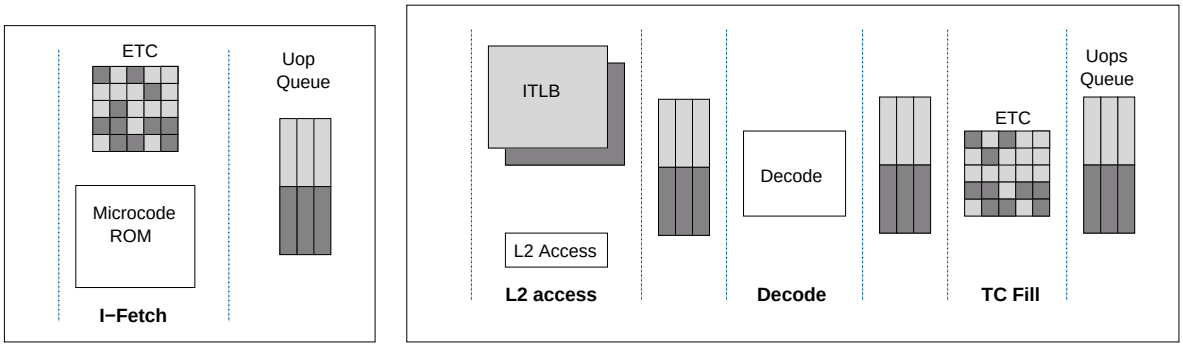
Numerous thread-based prefetching schemes, either static or dynamic, have recently been proposed. Zilles and Sohi's Speculative Slices [ZS01] execute as helper threads having their own registers (copy of main thread values), without performing any stores. In Roth and Sohi's Data Driven Multithreading [RS01], critical instructions are predicted in order to fork a speculative data-driven thread (DDT). Speculative results integrated into the main thread when a data-flow comparison determines that speculatively executed instructions exactly match instructions renamed by the main thread. This data-flow comparison requires one-to-one correspondence in the computations, precluding optimization of slices. Luk proposed in [Luk01] Software Controlled Pre-Execution, which can accurately generate data addresses with irregular patterns by directly executing the code that generates them. Annavaram et al.'s Data Graph Precomputation (DGP) [APD01] is an approach for dynamically identifying and precomputing the instructions that determine the addresses accessed by those load/store instructions marked

as being responsible for most data cache misses. Moshovos et al's Slice-Processors [MPB01] dynamically predict critical computation slices of applications and run them speculatively in advance to prefetch delinquent loads. Collins et al. propose Dynamic Speculative Precomputation [CWT+01] (hardware-constructed thread-based prefetching), which performs all necessary instruction analysis, extraction, and optimization through the use of back-end instruction analysis hardware, located off the processorős critical path. Kim et al [KLW+04] construct helper threads automatically, using an optimization module in the Intel pre-production compiler. Sundaramoorthy et al's [SPR00] slipstream architecture propose the production of a reduced version of the original program, removing ineffectual computations, with the potential to execute instructions early.

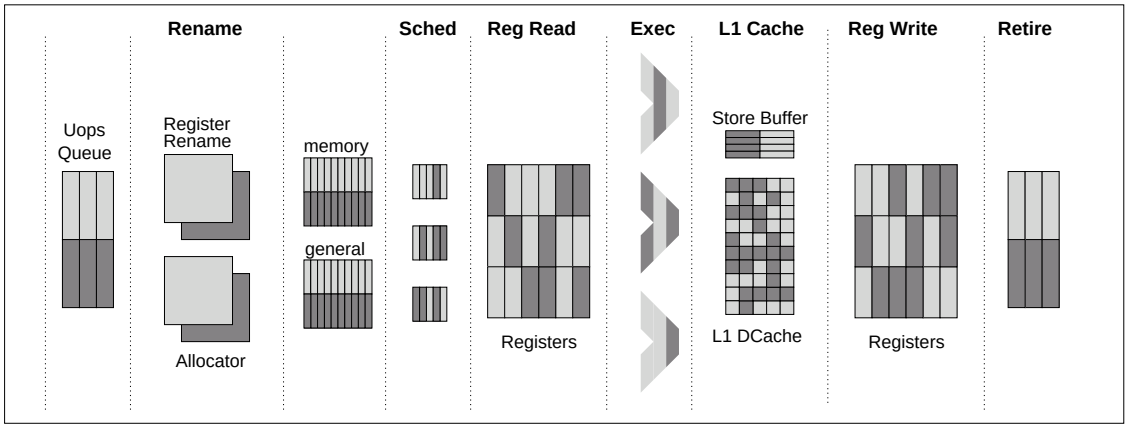
The key idea is to utilize otherwise idle hardware thread contexts to execute speculative threads on behalf of the main thread. These speculative threads attempt to trigger future cache-miss events far enough in advance of access by the non-speculative (main) thread, so that the memory miss latency can be masked. A common implementation pattern was used in these studies. A compiler identifies either statically or with the assistance of a profile the memory loads that are likely to cause cache misses with long latencies. Such load instructions, known as delinquent loads, may also be identified dynamically in hardware [CTWS01], [WWW+02], triggering speculative-helper threads. Speculative precomputation targets load instructions that exhibit unpredictable irregular, data-dependent or pointer chasing access patterns. Traditionally, these loads have been difficult to handle via either hardware or software prefetchers.

| | |
|---|---|
| Shared | Execution units, Trace cache, L1 D-cache, L2 cache, DTLB, Global history array, Allocator, Microcode ROM, $\mu$op retirement logic, IA-32 instruction decode, Instruction scheduler, Instruction fetch logic |
| Duplicated | Processor architecture state, Instruction pointers, Rename Logic, ITLB, Streaming buffers, Return stack buffer, Branch history buffer |
| Partitioned | $\mu$op queue, Memory instruction queue, Reorder buffer, General instruction queue |

**Table 5.1:** Hardware management in Intel hyper-threaded processors




(a) Trace cache hit datapath (b) Trace cache miss datapath




(c) Out of Order datapath

**Figure 5.1:** Resource partitioning in Intel hyperthreading architecture

## 5.3 Implementation

### Implementing speculative precomputation

There are two main issues that must be taken into account in order to effectively perform software prefetching using the multiple execution contexts of a hyper-threaded processor. First of all, the distance at which the precomputation thread runs ahead of the main computation thread, has to be sufficiently regulated. This requirement can be satisfied by imposing a specific upper bound on the amount of data to be prefetched. In our codes it ranges from $\frac{1}{A}$ ([TWN04]) to $\frac{1}{2}$ of the L2 cache size, where $A$ is the associativity of the cache (8 in our case). Whenever this upper bound is reached but the computation thread has not yet started using the prefetched data, the precomputation thread must stop its forward progress in order to prevent potential evictions of useful data from cache. It can only continue when it is signaled that the computation thread starts consuming the prefetched data. In our program codes, this scenario is implemented using synchronization barriers which enclose program regions (precomputation spans) whose memory footprint is equal to the upper bound we have imposed. In the general case, and considering their relatively lightweight workload, precomputation threads reach always first the barriers.

For codes whose access patterns were difficult to determine a-priori, we had to conduct memory profiling using the Valgrind simulator[NS03]. From the profiling results we were able to determine and isolate the instructions that caused the majority(92% to 96%) of L2 misses. In all cases, precomputation threads were constructed manually from the original code of the main computation threads, preserving only the memory loads that triggered the majority of L2 misses; all other instructions were eliminated.

Secondly, we must guarantee that the co-execution of the precomputation thread does not result in excessive consumption of shared resources that could be critical for the sibling computation thread. Despite the lightweight nature of the precomputation threads, as mentioned in the following section, significant processor resources can be consumed even when they are simply spinning on synchronization barriers.

### Synchronization Issues

The synchronization mechanisms have to be as lightweight as possible and for this purpose we have implemented lightweight spin-wait loops as the core of our synchronization primitives, with embedded the `pause` instruction in the spin loop, as recommended by Intel [Int]. This instruction introduces a slight delay in the loop and de-pipelines its execution, preventing it from aggressively consuming valuable, dynamically shared, processor resources (e.g. execution units, branch predictors and caches).

However, some other units (such as micro-ops queues, load/store queues and re-order buffers), are statically partitioned and are not released when a thread executes a `pause`. By using the privileged `halt` instruction, a logical processor can relinquish all of its statically partitioned resources, make them fully available to the other logical processor, and stop its execution going into a sleeping state. The `halt` instruction is primarily intended for use by the operating system scheduler. Multithreaded applications with threads intended to remain idle for a long period, could take advantage of this instruction to boost their execution. We implemented kernel extensions that allow from user space the execution of `halt` on a particular logical processor, and the wake-up of this processor by sending IPIs to it. By integrating these extensions in the spin-wait loops, we are able to construct long duration wait loops that do not consume significant processor resources. Excessive use of these primitives, however, in conjunction with the resultant multiple transitions into and out of the halt state of the processor, incur extra overhead in terms of processor cycles. This is a performance tradeoff that we took into consideration throughout our experiments.

## 5.4 Quantitative analysis on the TLP and ILP limits of the processor

This section explores the ability and the limits of hyper-threading technology on interleaving and executing efficiently instructions from two independent threads. We constructed a series of homogeneous instruction streams, which include basic arithmetic operations (add,sub,mul,div), as well as memory operations (load, store), on integer and floating-point 32-bit scalars. For each of them, we tested different levels of instruction level parallelism.

In our experiments, we artificially increase(decrease) the ILP of the stream by keeping the source and target registers always disjoint, and at the same time expanding(shrinking) the target operands ($T$). We have considered three degrees of ILP for each instruction stream: minimum ($|T|$=1), medium ($|T|$=3), maximum ($|T|$=6).

### 5.4.1 Co-executing streams of the same type

| | *CPI* | | | | | |
|---|---|---|---|---|---|---|
| | ***min ILP*** | | ***med ILP*** | | ***max ILP*** | |
| instr. | ***1thr*** | ***2thr*** | ***1thr*** | ***2thr*** | ***1thr*** | ***2thr*** |
| **fadd** | 6.01 | 6.03 | 2.01 | 3.28 | **1.00** | 2.02 |
| **fmul** | 8.01 | 8.04 | 2.67 | 4.19 | 2.01 | **3.99** |
| **faddmul** | 7.01 | 7.03 | 2.34 | 3.83 | 1.15 | **2.23** |
| **fdiv** | **45.06** | 99.90 | 45.09 | 107.05 | 45.10 | 107.43 |
| **fload** | 1049.05 | 2012.62 | 1049.06 | 2012.43 | 1049.05 | **2011.86** |
| **fstore** | 1050.67 | 1982.99 | 1050.68 | 1983.07 | 1050.67 | **1982.93** |
| **iadd** | 1.01 | **1.99** | 1.01 | 2.02 | 1.00 | 2.02 |
| **imul** | 11.02 | 11.05 | 11.03 | **11.05** | 11.03 | 11.05 |
| **idiv** | 76.18 | 78.76 | 76.19 | **78.71** | 76.18 | 78.73 |
| **iload** | 2.46 | 4.00 | 2.46 | 3.99 | 2.46 | **3.99** |
| **istore** | 1.93 | 4.07 | 1.93 | 4.08 | **1.93** | 4.07 |

**Table 5.2:** Average CPI for different TLP and ILP execution modes of some common instruction streams

As a first step, we execute each instruction stream alone on a single logical processor, for all degrees of ILP (1thread columns of table 5.2). In this way, all the execution resources of the physical package are fully available to the thread executing that stream. As a second step, we co-execute within the same physical processor two independent instruction streams of the same ILP, each of which gets bound to a specific logical processor (threads columns of

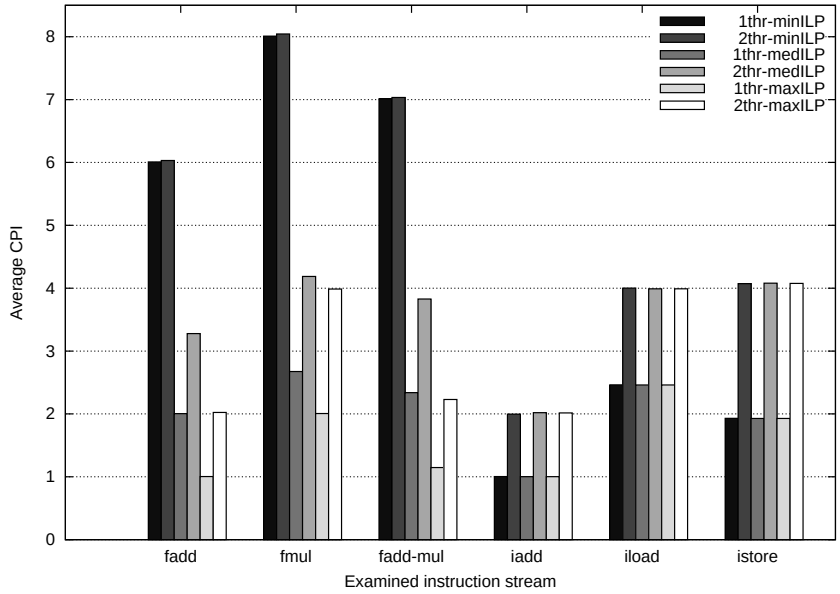



**Figure 5.2:** Average CPI for different TLP and ILP execution modes of some common instruction streams

table 5.2). This gives us an indication on how various kinds of simultaneously executing streams of a specific ILP level, contend with each other for shared resources, and an estimation whether the transition from single-threaded mode of a specific ILP level to dual-threaded mode of a lower ILP level, can hinder or boost performance. For example, let's consider a scenario where, in single-threaded and maximum ILP mode, instruction $A$ gives an average CPI of $C_{1thr-maxILP}$, while in dual-threaded and medium ILP mode the same instruction gives an average CPI of $C_{2thr-medILP} > 2 \times C_{1thr-maxILP}$. Because the second case involves half of the ILP of the first case, the above scenario prompts that we must probably not anticipate any speedup by parallelizing into multiple threads a program that uses extensively this instruction in the context of high ILP (e.g. unrolling). Bold figures of table 5.2 indicate best case performance. Figure 5.2 depicts the slowdown factors of table 5.2.

### 5.4.2 Co-executing streams of different types

Table 5.3 presents the results from the co-execution of different pairs of streams (for the sake of completeness, results from the co-execution of a given stream with itself, are also presented). We examine pairs whose streams have the same ILP level. The slowdown factor represents the ratio of the CPI when two threads are running concurrently, to the CPI when the benchmark indicated in the fist column is being executed in single-threaded mode. Note that the throughput of integer streams is not affected by variations of ILP and for this reason we present only exact figures of medium ILP. All figures that vary less than 0.05 compared to medium ILP representative ones are also omitted. Bold figures indicate the most significant slowdown factors. Figures 5.3 and 5.4 depict the slowdown factors of table 5.3.

| | | *Co-executed Instruction Streams* | | | | |
|---|---|---|---|---|---|---|
| | ILP | ***fadd*** | ***fmul*** | ***fdiv*** | ***fload*** | ***fstore*** |
| **fadd** | min: | 1.004 | 1.004 | | | |
| | med: | 1.635 | 1.787 | 1.010 | 1.398 | 1.409 |
| | max: | **2.016** | **2.801** | **2.023** | 1.474 | 1.462 |
| **fmul** | min: | 1.002 | 1.004 | 1.006 | | |
| | med: | 1.433 | 1.566 | 1.062 | 1.391 | 1.393 |
| | max: | 1.384 | **1.988** | | | |
| **fdiv** | min: | | | **2.217** | | |
| | med: | 1.017 | 1.027 | **2.374** | 1.413 | 1.422 |
| **fload** | min: | 1.144 | 1.169 | | | |
| | med: | 1.286 | 1.255 | 1.153 | **1.919** | **1.907** |
| | max: | 1.684 | 1.358 | | | |
| **fstore** | min: | 1.134 | 1.133 | | | |
| | med: | 1.229 | 1.229 | 1.150 | **1.897** | **1.887** |
| | max: | 1.625 | 1.316 | | | |
| | ILP | ***iadd*** | ***imul*** | ***idiv*** | ***iload*** | ***istore*** |
| **iadd** | med: | **2.014** | 1.316 | 1.117 | 1.515 | 1.405 |
| **imul** | med: | 1.116 | 1.002 | 1.008 | 1.003 | 1.004 |
| **idiv** | med: | 1.042 | 1.019 | 1.033 | 1.003 | 1.003 |
| **iload** | med: | **2.145** | 0.941 | 0.934 | 1.621 | 1.331 |
| **istore** | min: | 4.072 | | | | |
| | med: | **4.299** | 1.979 | 1.970 | 1.986 | 2.115 |
| | max: | 2.160 | 0.941 | 0.934 | 1.622 | 1.331 |

**Table 5.3:** Slowdown factors from the co-execution of various instruction streams

## 5.5 Summary

This chapter presents the architectural characteristics of a simultaneous multithreaded platform, the hyper-threaded Intel microarchitecture. We examined homogeneous instruction streams executed in parallel in the hyper-threaded processor, in order to find average CPIs for integer/floating point calculations and memory operations.

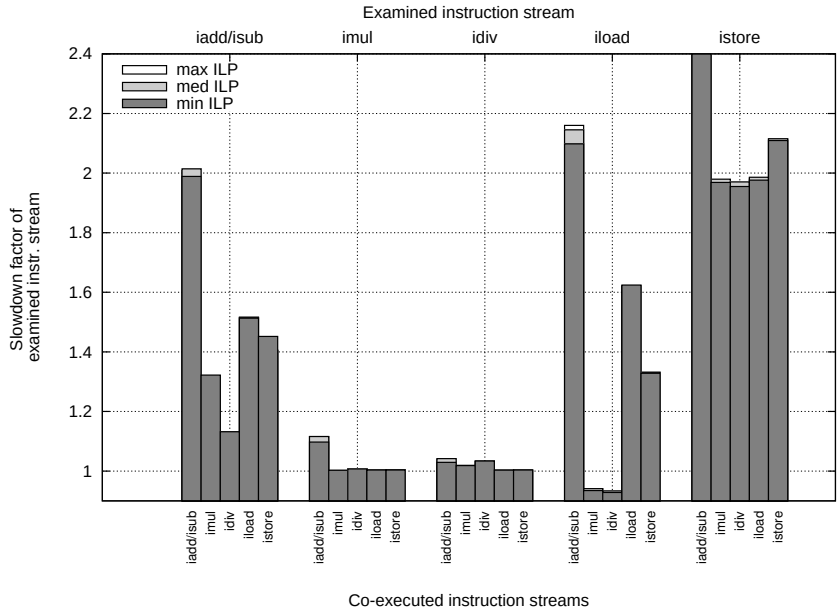



**Figure 5.3:** Slowdown factors for the co-execution of various integer instruction streams

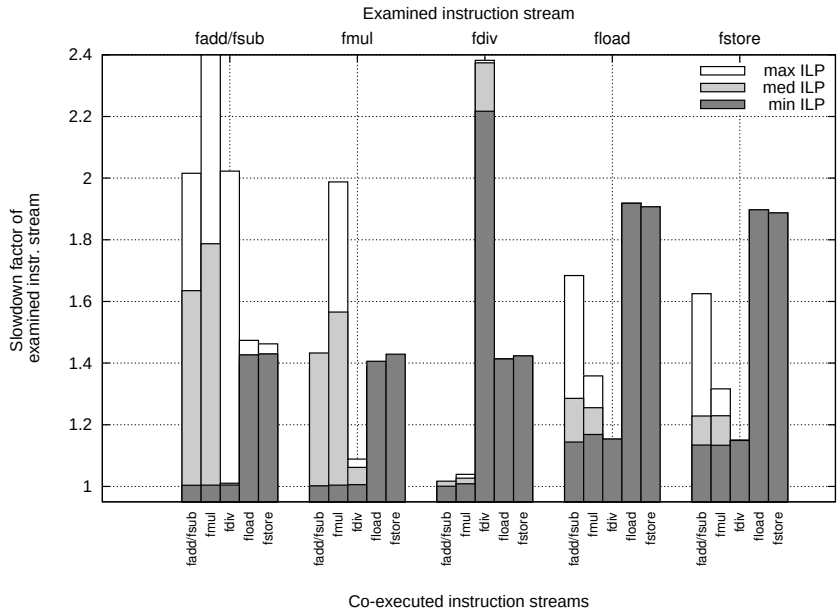



**Figure 5.4:** Slowdown factors for the co-execution of various floating-point instruction streams

CHAPTER 6

# Experimental Results

## 6.1 Experimental results for Fast Indexing

### 6.1.1 Execution Environment

In this section we present experimental results using Matrix Multiplication, LU-decomposition, SSYR2K, SSYMM and STRMM as benchmarks. There are two types of experiments: actual execution times of optimized codes using non-linear layouts and simulations using the SimpleScalar toolkit [LW94]. Firstly, the experiments were performed on three different platforms: an UltraSPARC II 450 machine, an SGI/Cray Origin2000 multiprocessor, and an Athlon XP 2600+ PC. The hardware characteristics are described in tables B.1 and B.2 of appendix B.

For the UltraSPARC and the SGI Origin platforms we used the cc compiler, first without any optimization flags (cc -xO0), in order to study the clear effect of different data layouts, avoiding any confusing results due to compiler optimizations. Then, we used the highest optimization level (-fast -xtarget=native), which includes memory alignment, loop unrolling, software pipeline and other floating point optimizations. The experiments were executed for various array dimensions ($N$) ranging from 16 to 2048 elements, and tile sizes (*step*) ranging from 16 to N elements, to reveal the potential of our optimization algorithm both on data sets that fit and do not fit in cache. In the Athlon XP platform the gcc compiler has been used, firstly without any optimization flags (gcc -O0) and then, at the highest optimization level (-O3).

We implemented 5 different codes: **B**locked **a**rray **L**ayou**t**s using our **M**ask theory for address computation (MBaLt), $L_{MO}$ (Lmo), $L_{4D}$ (L4d) of Chatterjee's approach and the Kandemir's method [KRC99] using both 2-dimensional arrays (tiled2D) and 1-dimensional arrays (tiled1D).Measured times do not concern runtime layout transformation from linear to blocked ones, because a single layout for all instances of each specific array is selected, which optimizes all possible references for the arrays.

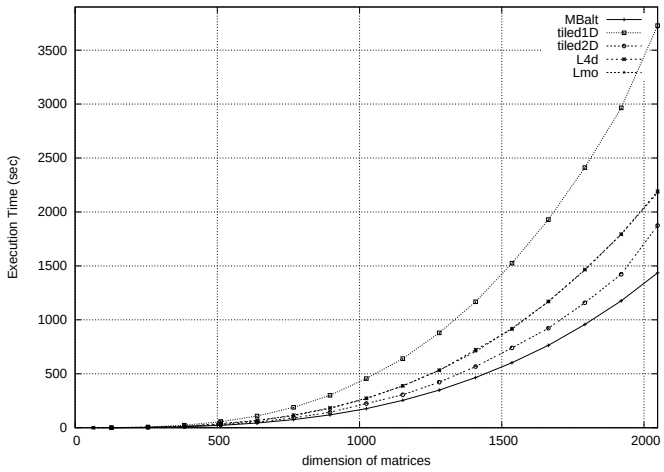



**Figure 6.1:** Total execution results in matrix multiplication (-xO0, UltraSPARC)

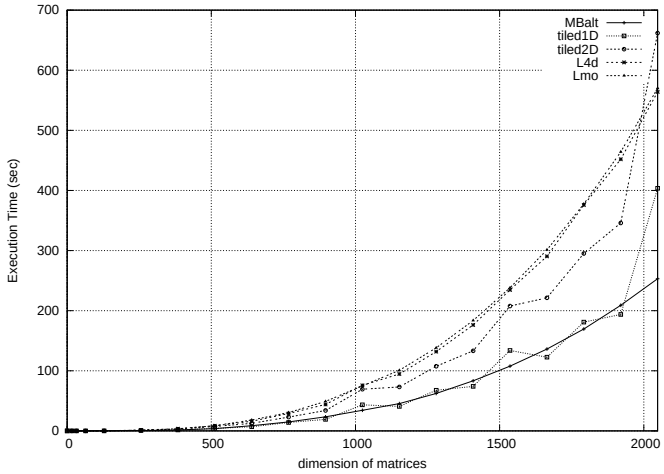



**Figure 6.2:** Total execution results in matrix multiplication (-fast, UltraSPARC)

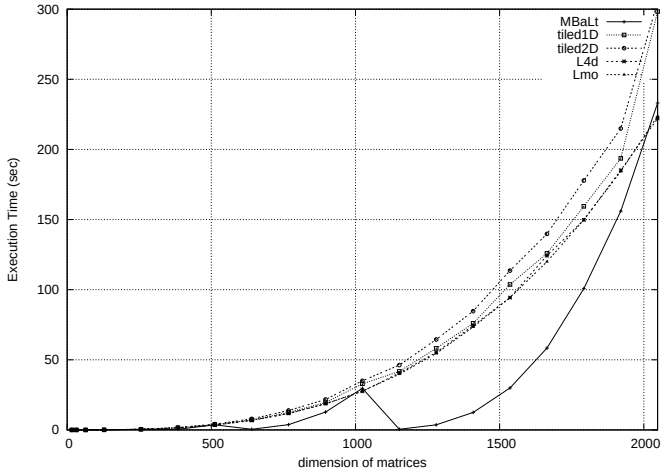



**Figure 6.3:** Total execution results in matrix multiplication (-fast, SGI Origin)

### 6.1.2 Time Measurements

For the Matrix Multiplication benchmark, we compared our method (MBaLt) for the best performance tile size, with the ones proposed in the literature [KRC99]. In figure 6.1 the

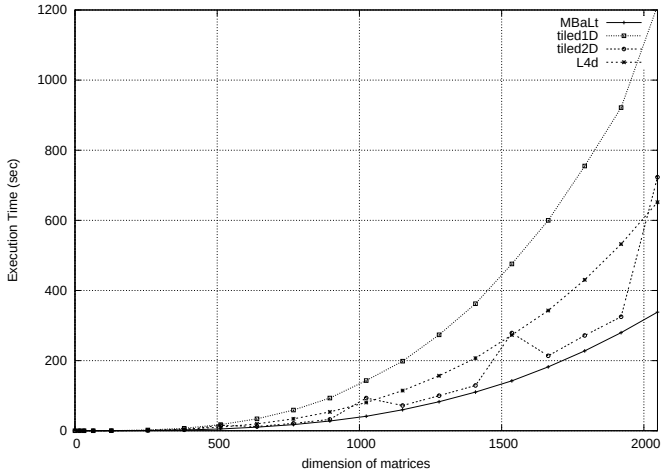



**Figure 6.4:** Total execution results in LU-decomposition (-xO0, UltraSPARC)

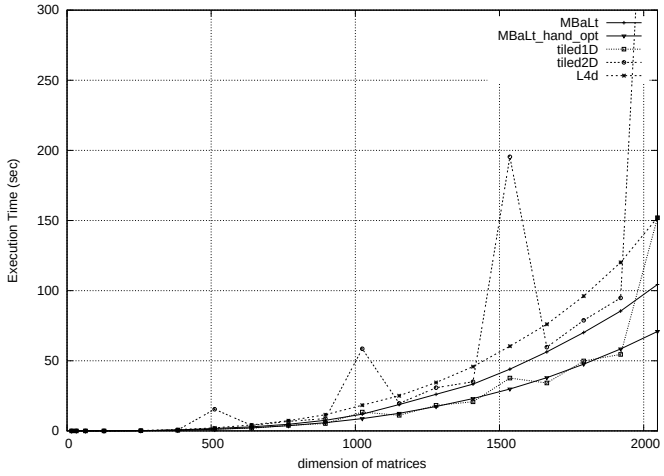



**Figure 6.5:** Total execution results in LU-decomposition (-fast, UltraSPARC)

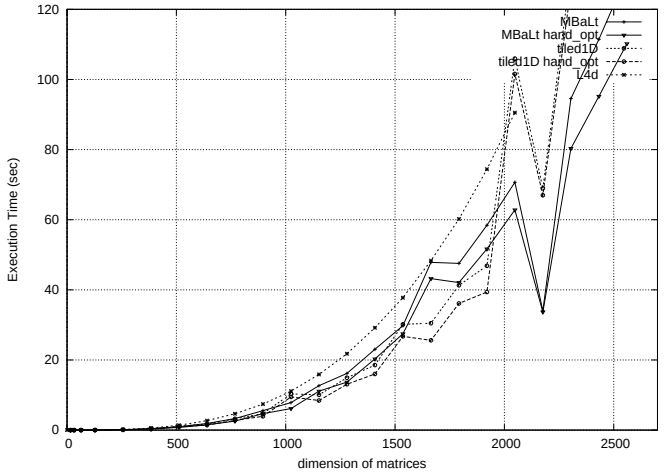



**Figure 6.6:** Total execution results in LU-decomposition for larger arrays and hand optimized codes (-fast, SGI Origin)

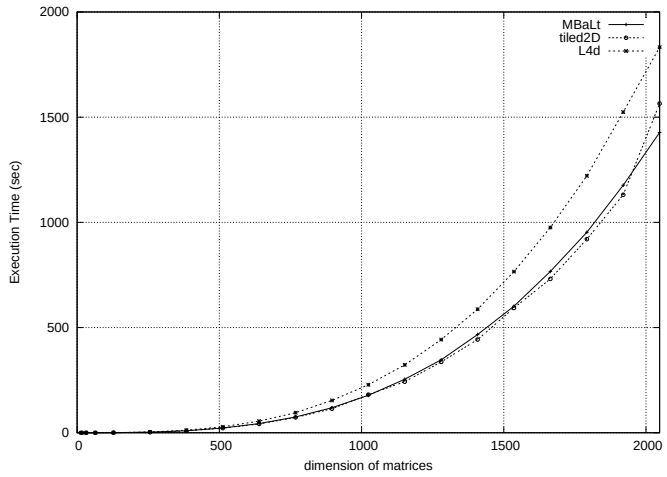



**Figure 6.7:** Total execution results in SSYR2K (-xO0, UltraSPARC)

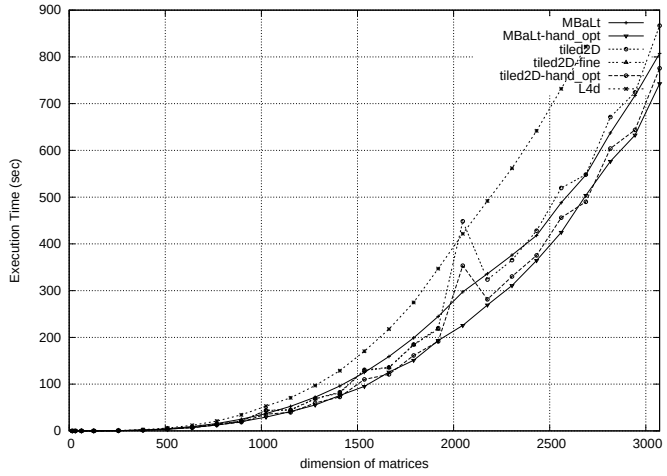



**Figure 6.8:** Total execution results in SSYR2K (-fast, UltraSPARC)

execution time of our method is almost 25% less than the one achieved using the optimal code from Kandemir et al for 2-dimensional arrays (tiled2D). In fact, since current compilers do not support non-linear (blocked) array layouts and, as a result, in the implementation of our methods we use one-dimensional arrays, the comparison should be done with one-dimensional arrays (tiled1D). In this case, MBaLt over-exceeds by 60%. Chatterjee's implementation performs worse than tiled2D, even though the $L_{4D}$ array layout is in practice the same as MBaLt, except for data are indexed through four-dimensional arrays, instead of the one-dimensional ones used in our approach. The performance degradation is due to the much more complicated memory location scheme used by four-dimensional arrays.

Using the -fast optimization level (figure 6.2), MBaLt gives stable performance as its graph increases smoothly when array sizes increase, due to independence on conflict misses. The tiled version of the benchmark, (especially in LU-decomposition and SSYR2K), is prone to such kind of misses, which can not be easily predicted, since sharp fluctuations occur in performance for specific matrix sizes. Furthermore, comparing figures 6.1 and 6.2, we conclude that the simpler

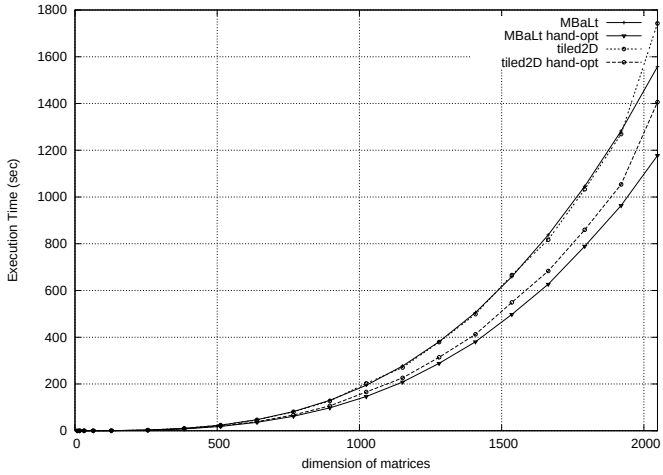



**Figure 6.9:** Total execution results in SSYMM (-xO0, UltraSPARC)

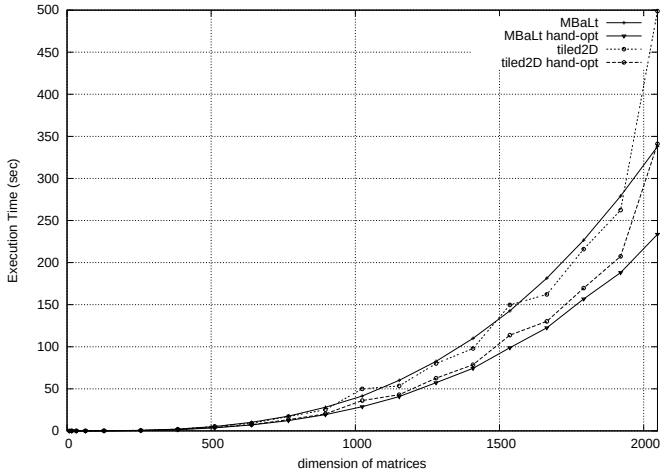



**Figure 6.10:** Total execution results in SSYMM (-fast, UltraSPARC)

the initial code is, the better optimization the -fast level can bring. Thus, standard compiler options are not enough, when optimizing complex codes. Applying hand optimization (e.g. loop unrolling and data prefetching) in conjuction with -fast optimization level, proves to be the best technique for achieving efficient performance (see hand-optimized performance of MBaLt: MBaLt-hand_opt in figure 6.6).

Although the MBalt code is larger in terms of instruction lines, it takes less to execute, since boolean operations are used for address computation. Furthermore, array padding does not affect the performance, since execution time increases regularly analogously to the actual array sizes. The reader can verify that no sharp peaks occur in the execution time plots. Delving into the assembly code, one-dimensional arrays are more efficiently implemented, than greater dimension ones, since they require for fewer memory references to find the array elements. On the other hand, finding the storage location of an array element in non-linear layouts needs a lot of computation. Consequently, what we achieved by the proposed implementation is handling one-dimensional arrays without posing any additional burden due to address computation, namely

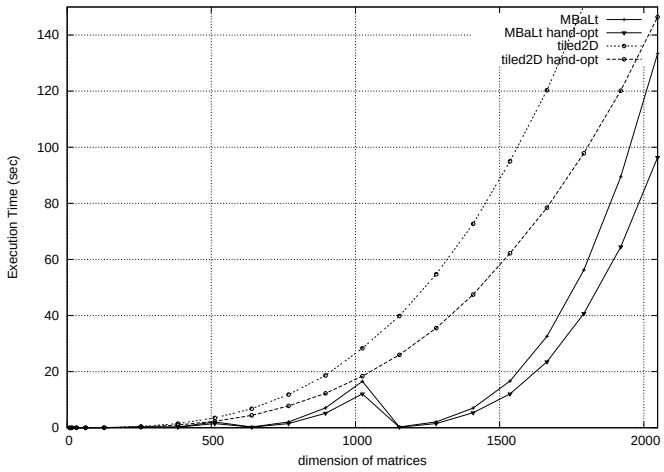



**Figure 6.11:** Total execution results in SSYMM (-O0, Athlon XP)

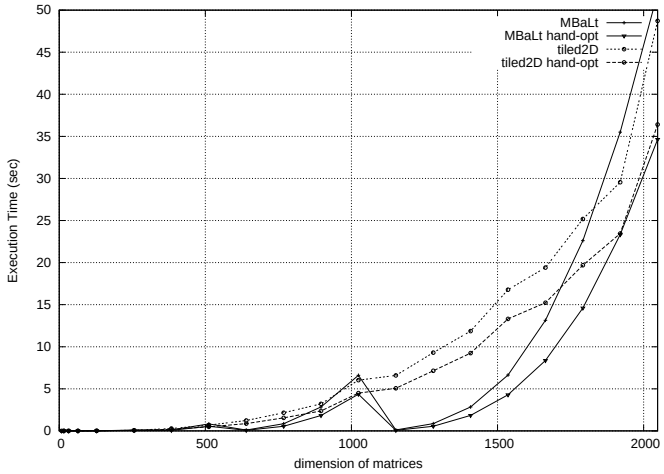



**Figure 6.12:** Total execution results in SSYMM (-O3, Athlon XP)

one-dimensional arrays with low address computation cost.

The above results are also verified by the LU-decomposition, SSYR2K, SSYMM and STRMM benchmarks (figures 6.4 - 6.14). Additionally, we notice that in LU-decomposition, the use of blocked array layouts in combination with our efficient indexing not only provides an average of 15% reduction in time measurements (when no optimization is applied), but also smooths the sharp peaks that come up due to conflict misses in the simple tiled code. The reduction is even better (can reach even 30%) with -fast flag. Observing that tiled1D (in the matrix multiplication benchmark) and LU-decomposition) outperform MBaLt when -fast optimization is applied, we conducted experiments with larger arrays than $2048 \times 2048$. Figure 6.6 illustrates that MBaLt is advantageous.

In the SSYR2K benchmark (figures 6.7 and 6.8), we searched for the best tile size among a wide range of values (not only for power of 2 values), when linear layouts are used (tiled2D-fine). We notice that in all previous tiled codes we conducted experiments for tile sizes equal to some power of 2, in order to agree with MBaLt implementation. As graphs for tiled and tiled-fine are

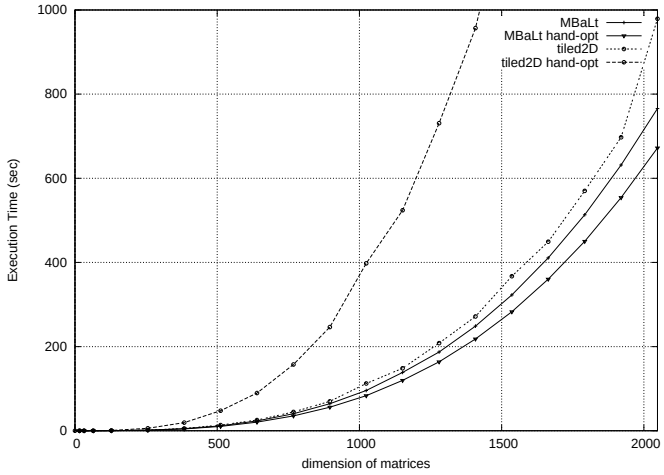



**Figure 6.13:** Total execution results in STRMM (-xO0, UltraSPARC)

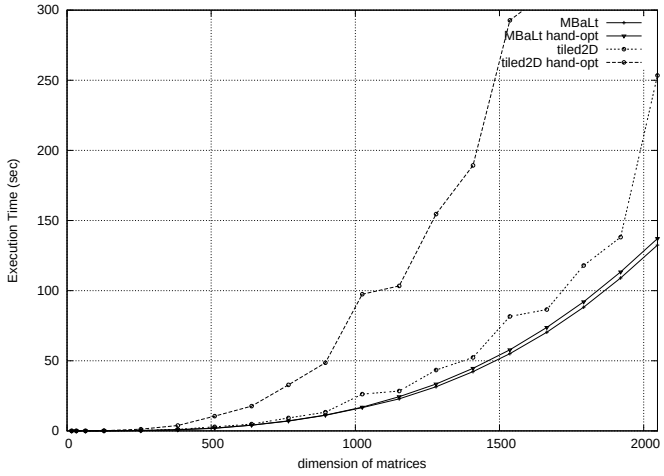



**Figure 6.14:** Total execution results in STRMM (-fast, UltraSPARC)

practically identical, execution times prove that no burden is imposed when only a power of 2 for tile sizes is used.

Finally, as far as tile sizes are considered, MBaLt versions perform better for small tile sizes, thus, keep data mainly in L1 cache. Best tile sizes remain fixed for almost for all array sizes. On the other hand, tiled versions (tiled1D and tiled2D) give minimum execution time for different tile sizes as array sizes change.

### 6.1.3 Simulation results

In order to verify the results of time measurements, we applied the program codes of matrix-multiplication (MBaLt, tiled2D, tiled1D) and the ones of LU-decomposition (non-tiled, tiled, MBaLt) to the SimpleScalar 2.0 toolkit, for both the cache characteristics of the UltraSPARC and the SGI Origin machines. We measured the data L1 (dl1), unified L2 (ul2) cache and data TLB (dtlb) misses, for various values of $N$ ranging from 16 to 1024 and *step* from 8 to $N$, at

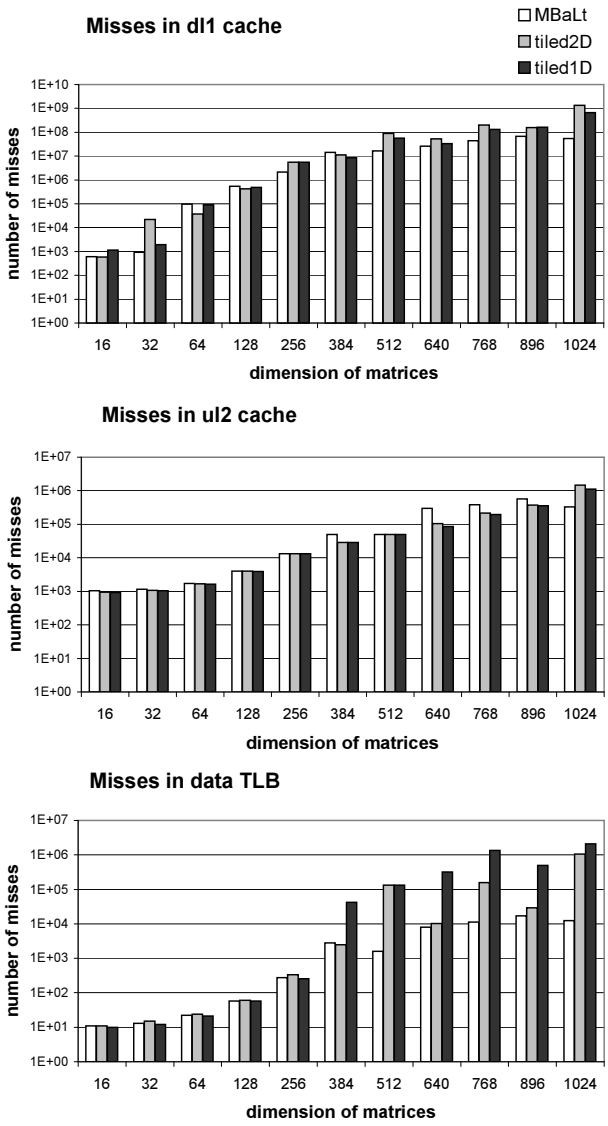



**Figure 6.15:** Misses in Data L1, Unified L2 cache and data TLB for matrix multiplication (UltraSPARC)

the standard optimization level. The scale in vertical axis, in all plots, is a logarithmic one.

The results show that dl1 misses are reduced in matrix multiplication for tile size $32 \times 32$ because data fit in the L1 cache. For the same reason, for $step = 1024$ misses in ul2 cache increase sharply, compared to ul2 misses for $step = 512$. The minimum value fluctuates with the problem size, because it depends on factors such as conflict misses which can not be easily foreseen. Figures 6.15 to 6.17 show the number of misses for different array sizes. In all cases, MBaLt gives the least misses. The number of accesses offer the same conclusions, so respective plots are not presented here, due to lack of space.

Finally, we observed that among all the tile sizes used in the matrix multiplication benchmark, in MBaLt the minimum TLB misses occur for a value of *step* near to 256. This is in accordance with the minimization of L1 and L2 cache misses. Thus, the reduction of the total

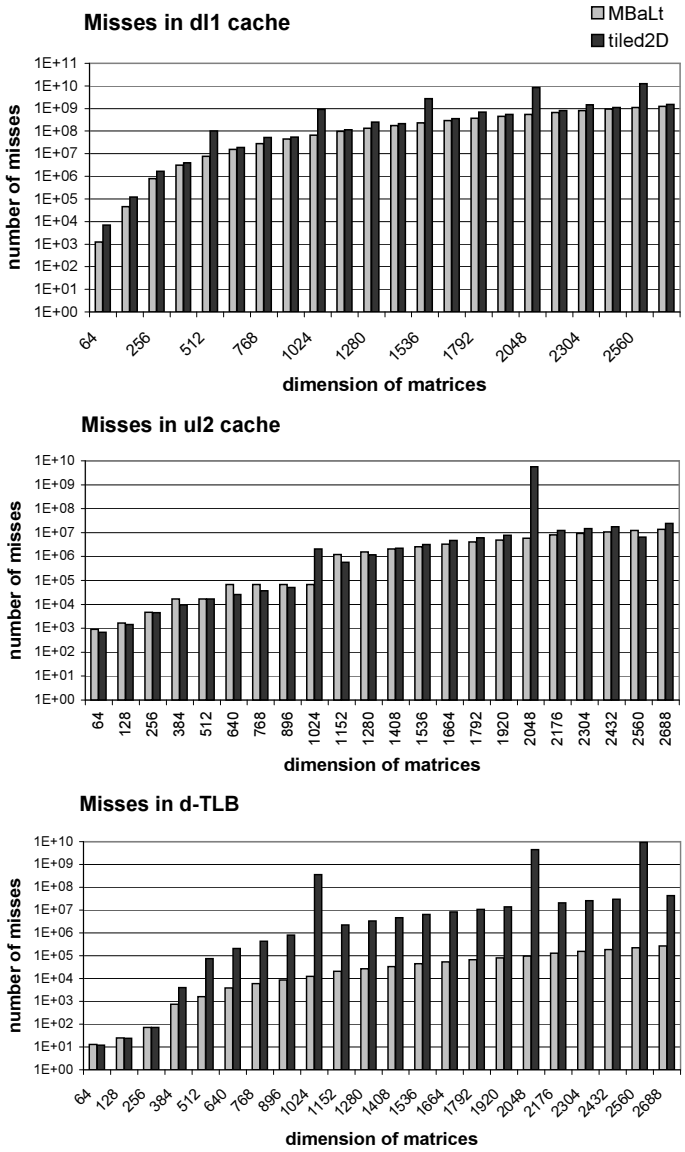



**Figure 6.16:** Misses in Data L1, Unified L2 cache and data TLB for LU-decomposition (UltraSPARC)

time performance is reinforced. On the contrary, in tiled1D and tiled2D the best *step* size is 128 and 32 respectively while for other values of *step*, misses increase rapidly.

## 6.2 Experimental Results for Tile Sizes

### 6.2.1 Execution Environment

In this section we present experimental results using Matrix Multiplication, LU-decomposition, SSYR2K, SSYMM and STRMM as benchmarks. There are two types of experiments: actual execution times of optimized codes using non-linear layouts and simulations using the SimpleScalar's sim-outorder toolkit [LW94]. Firstly, the experiments were performed on four different platforms:

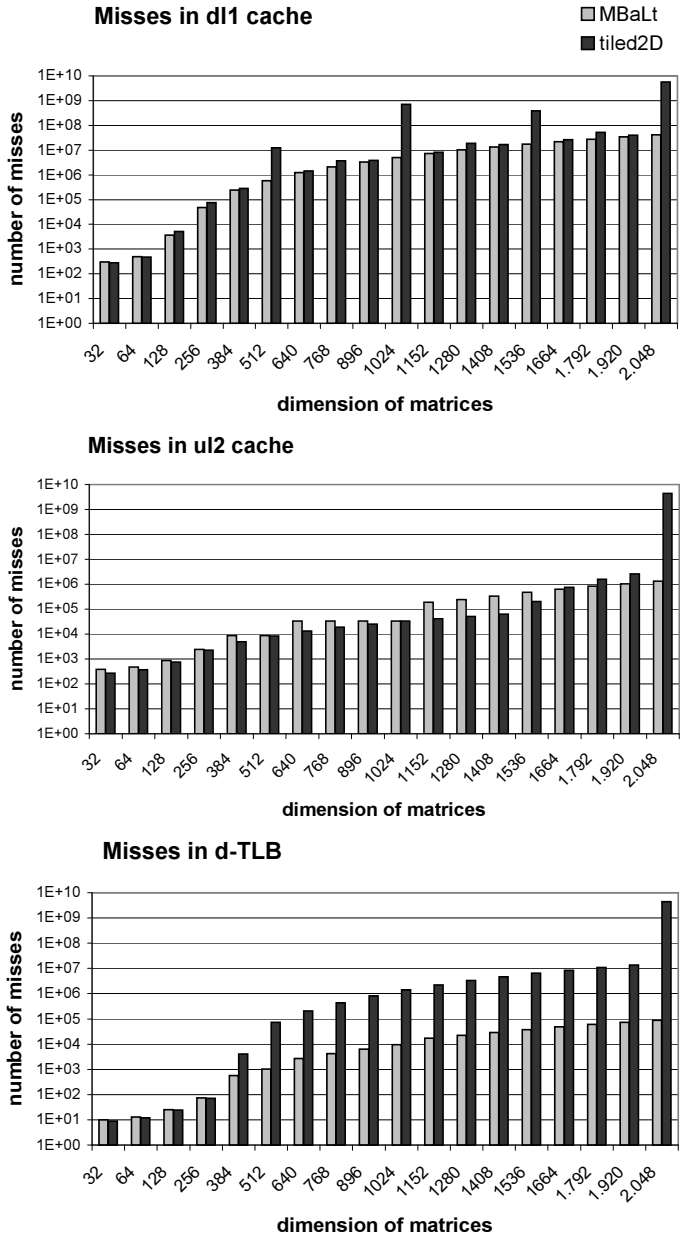



**Figure 6.17:** Misses in Data L1, Unified L2 cache and data TLB for LU-decomposition (SGI Origin)

an UltraSPARC II 450 machine, a Pentium III Symmetric Multiprocessor (SMP),an Athlon XP 2600+, and a Dual SMT Xeon (Simultaneous Multi-Threading supported).

The hardware characteristics are described in tables B.1 and B.2 of appendix B. For the UltraSPARC platform we used the SUN cc compiler, at the highest optimization level (-fast -xtarget=native), which includes memory alignment, loop unrolling, software pipeline and other floating point optimizations. In the Pentium III, Athlon and Xeon the gcc compiler has been used.

The experiments were executed for various array dimensions ($N$) ranging from 64 to 4096 elements, and tile sizes (*step*) ranging from 16 to N elements, to reveal the potential of our

optimization tile size selection algorithm both on data sets that fit and do not fit in cache.

### 6.2.2 Experimental verification

Firstly, we executed the Matrix Multiplication kernel, applying fast indexing to our Blocked Array Layouts. The execution results verify the cache and TLB behaviour of section **??**: L1 cache misses seem to dominate total performance (we achieve maximum performance when $T = sqrtC_{L1}$). Only a limited range of values is depicted in figure 6.18, to make the image readable.

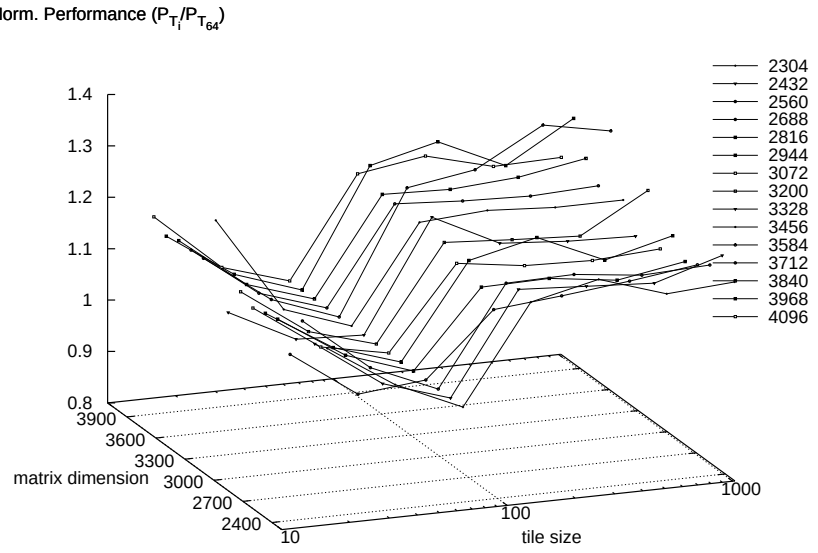



**Figure 6.18:** Execution time of the Matrix Multiplication kernel for various array and tile sizes (UltraSPARC, -fast)

### 6.2.3 MBaLt performance: Tile size selection

Execution and simulation results of this optimized version are illustrated in figure 6.19. Blocked array layouts, when appropriately combined with loop optimization techniques, have optimal tile size $T = \sqrt{C_{L1}}$. In the MBaLt code, only square tiles can be used, with dimension sizes to be a power of 2. That is, for the UltraSPARC II architecture, the tile size is $T = 64$.

### 6.2.4 MBaLt vs linear layouts

In this section we compare the MBaLt (Blocked array layouts with use of binary masks) performance, with a fully optimized version of linear layouts (it concerns the best performance algorithm proposed in the literature [KRC99]). We investigate the exact reasons that boost performance when blocked layouts are implemented.

As it can be clearly discriminated in figure 6.20, linear layouts have unstable performance, while array sizes vary. Notice that we choose either row-major or column-major storage of array

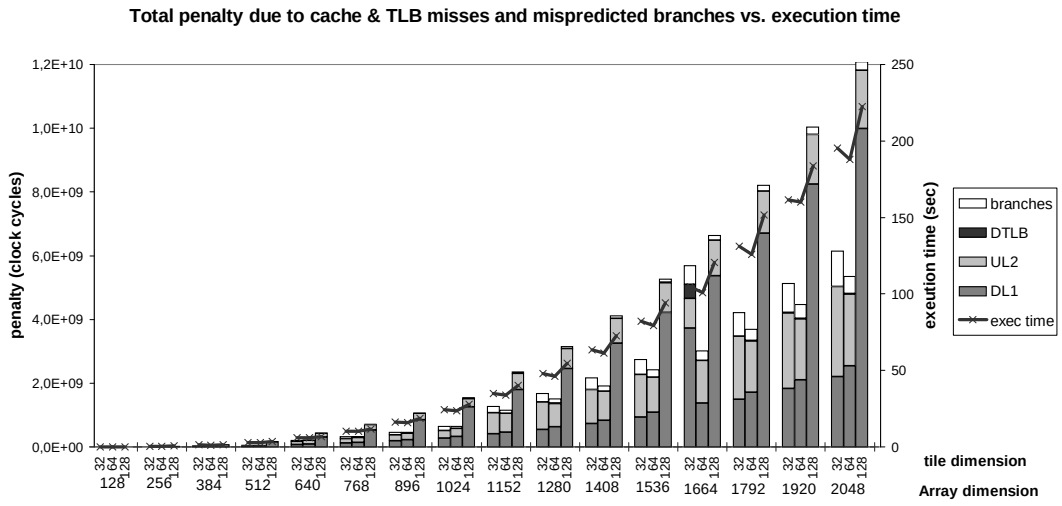



**Figure 6.19:** Total performance penalty due to data L1 cache misses, L2 cache misses and data TLB misses for the Matrix Multiplication kernel with use of Blocked array Layouts and efficient indexing. The real execution time of this benchmark is also illustrated (UltraSPARC)

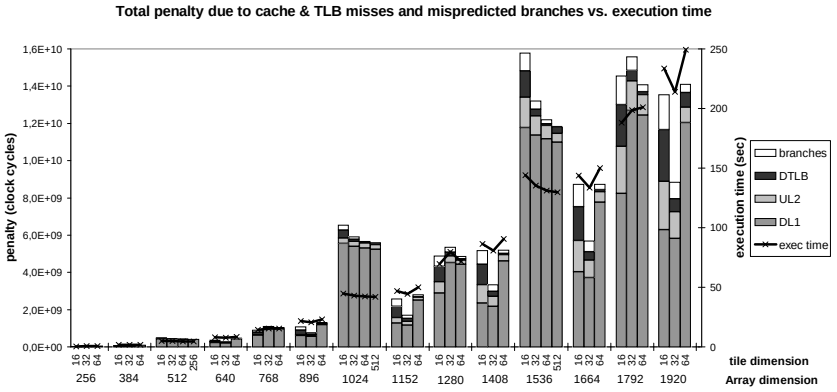



**Figure 6.20:** Total performance penalty and real execution time for the Matrix Multiplication kernel (linear array layouts - UltraSPARC)

elements in memory, to match as well as possible the access order of the benchmark code. This instability is due to the non-easily predicted conflict misses. Techniques, such as padding or copying that should be used to avoid pathological array sizes in linear layouts, there is no need to be applied in Blocked array layouts. Consequently, optimizing a nested loop code proves to be a much more standard process in MBaLt case.

Analytical search of degradation causes in linear layouts are represented graphically in figure 6.21. A limited range of values are illustrated here, to make image as clear as possible. In both cases (MBaLt & kand), the best performance tile sizes are chosen. Data L1 misses in linear array layouts can be even one order of magnitude more than in MBaLt, due to conflict misses that is too complicated to be avoided. Data TLB misses are significantly reduced in MBaLt (they actually reach the minimum possible, given the TLB capacity) The access code pattern matches exactly the storage order and array elements that belong to the same tile, are located in $\frac{T^2}{P}$ consecutive pages. In the worst case they may be found on the border of $\frac{T^2}{P} + 1$

pages. L2 misses are slightly increased in MBaLt. However, performance improvement due to the reduced number of L1 cache and TLB misses is so determinative, that this L2 increase is not even worthwhile. Finally, the number of mispredicted branches are also reduced in most cases of MBaLt, because maximum performance is usually achieved by smaller tiles in linear layouts, in order to avoid interference misses.

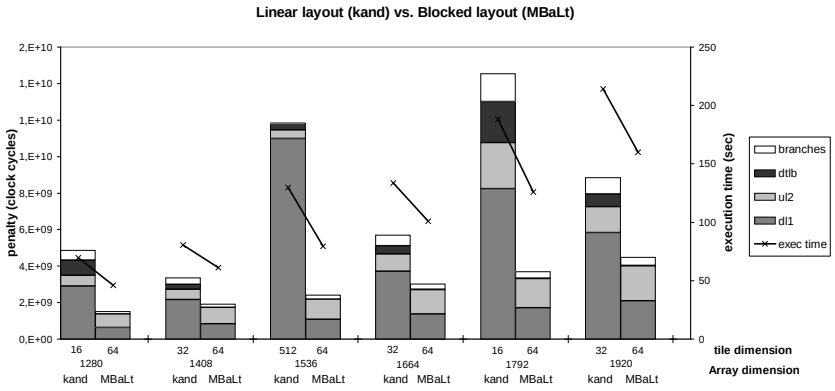



**Figure 6.21:** The relative performance of the two different data layouts (UltraSPARC)

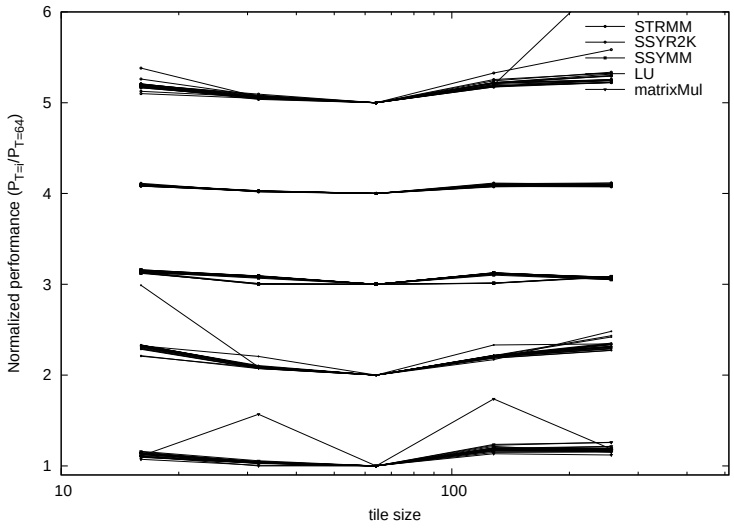



**Figure 6.22:** Normalized performance of 5 benchmarks for various array and tile sizes (UltraSPARC)

### 6.2.5 More Experiments

To validate the above results, we implemented five different benchmarks executed in four different platforms. Experiments are conducted with Matrix Multiplication, LU-decomposition, SSYR2K, SSYMM and STRMM benchmarks from BLAS3 routines.

In UltraSPARC II architecture, as verified in figure 6.22 all five cases of tested benchmarks, that the execution time is minimized when $T = 64$. This is the tile size that just fits in the direct mapped L1 cache (L1 cache thrashing is restrained until this point). Figure 6.22 illustrates the normalized performance of the tested benchmarks, shifted along the vertical axis,

so that different codes do not overlap That is, $\frac{performance\ of\ T_N=(any\ tested\ value)}{performance\ of\ T_N=64} + x$, where $x = \{0, 1, 2, 3, 4\}$, $T_N \in [8, N]$ and $N \in [64, 4096]$. In figure 6.22, depicted values of $T_N \in [16, 256]$, zooming in the area of maximum performance.

The Athlon XP and the Pentium III machine have different architecture characteristics. Associativity of caches eliminates all conflict misses. As a result, L1 cache misses are significantly reduced. L2 cache is multy-way set associative, too. On the other hand, it is quite smaller in capacity than the one of UltraSPARC, which brings many more L2 cache misses. However, the reduced number of L1+L2 misses achieved for tile sizes that fit in L1 cache can not be achieved elsewhere This situation makes L1 misses to dominate overall performance again (see figure 6.23). The optimal tile size is the one that meets the requirement: $T^2 = C_{L1}$. The performance graphs are illustrated in figures 6.24 and 6.25. Notice that in the Pentium III experiments (figure **??**) $x = \{0, 4, 8, 12, 16\}$ while in the Athlon XP (figure 6.25) $x = \{0, 2, 4, 6, 8\}$. These are the needed values to avoid overlapping among the different benchmark plots. In both figures the depicted values of $T_N \in [32, 512]$, zooming in the area of maximum performance.

The dual Xeon platform needed special attention, in order to efficiently exploit the hyper-threading technology. We conducted three different experiments. Firstly, the serial blocked algorithm of Matrix Multiplication (MBaLt code - with use of fast indexing) was executed. Secondly, we enabled hyperthreading running 2 threads in the same physical cpu. For large tile sizes, execution times obtained with the 2threads-MBaLt version are quite larger than those of the serial version. Smaller tile sizes lead to more mispredicted branches and loop boundary calculations, thus increasing the overhead of tiling implementation. In the case of 2threads-MBaLt, this tiling overhead gets almost doubled, since the two threads are executing the same code in an interleaved fashion. In other words, the total overhead introduced overlaps the extra benefits we have with the simultaneous execution capabilities of the hyperthreaded processor. This is not the case for larger tile sizes, where the tiling overhead is not large enough to overlap the advantages of extra parallelism. Figure 6.26 illustrates only best performance measurements for each different array dimension (tile sizes are equal to the one minimize execution time). The serial-MaLt version seems to have better performance compared to the 2threads-MBaLt version, as execution time is minimized for small tile sizes. Finally, we executed a parallel version of matrix multiplication MBaLt code (4threads-MBaLt), where 2 threads run on each of the 2 physical cpus, that belong to the same SMP. Execution time is reduced, and performance speed up reaches 44% compared to the serial-MBaLt version.

As far as the optimal tile size is concerned, serial MBaLt obey to the general rule, this is $T_{optimal} = \sqrt{C_{L1}} = 64$. However, when hyperthreading had been enabled, $T_{optimal}$ seams to be shifted to the just smaller size, in order to make room in the L1 cache for the increased number of concurrently used array elements. This behaviour is observed, both when two threads run on the same physical cpu (2threads-MBaLt), as well as in the parallel version (4threads-MBaLt) where $T_{optimal} = 32$ (figures 6.27, 6.28 and 6.29). Note that for the 2threads-MBaLt version

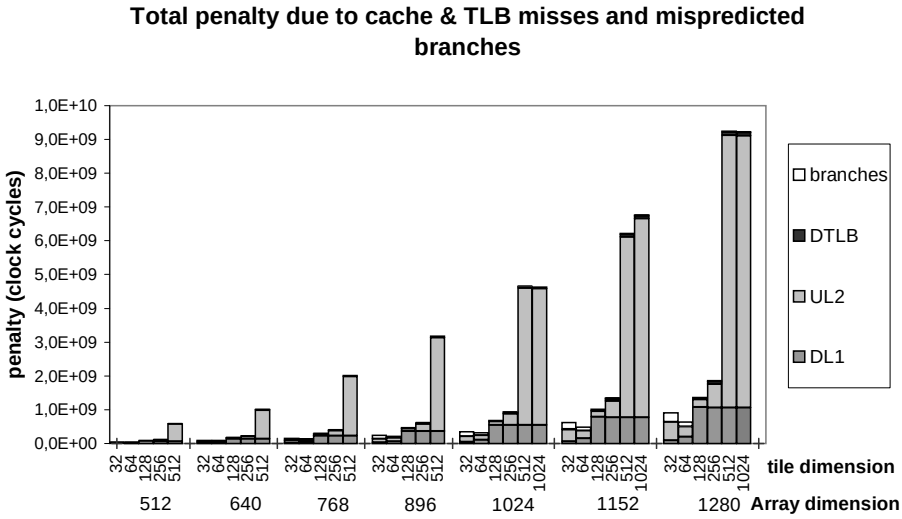



**Figure 6.23:** Total performance penalty for the Matrix Multiplication kernel (Pentium III)

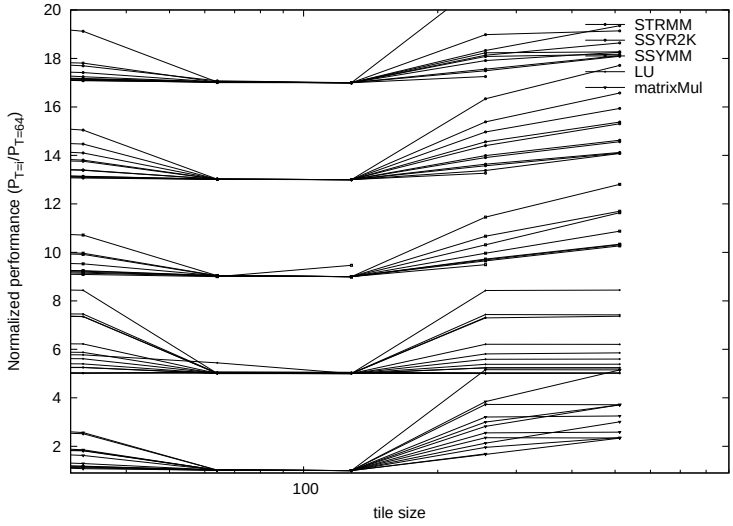



**Figure 6.24:** Pentium III - Normalized performance of five benchmarks for various array and tile sizes

$T_{optimal} = 32$ when $N < 2048$. For larger arrays, the 2threads-MBaLt version behaves similarly to the serial one, filling the whole L1 cache with useful array elements.

## 6.3 Experimental Framework and Results on SMTs

This section evaluates and contrasts software prefetching and thread-level parallelism techniques. It also examines the effect of thread synchronization mechanisms on multithreaded parallel applications that are executed on a single SMT processor.Finally, we examine the effect of maintaining a single thread per parallel application and embodying precomputation in it, to avoid any synchronization overheads and static resource partitioning.

The experimental results demonstrate that significant performance improvements are really

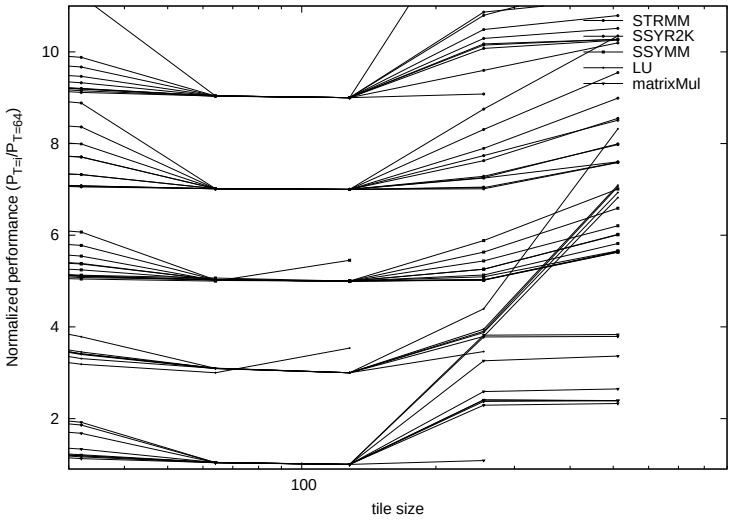



**Figure 6.25:** Athlon XP - Normalized performance of five benchmarks for various array and tile sizes

hard to be achieved for optimized parallel applications running on SMT processors. We tested two different configurations. Firstly, we balanced the computational workload of a parallel benchmark on two threads, statically partitioning the iteration space to minimize dynamic scheduling overhead. Secondly, we ran a main computation thread in parallel with a helper-prefetching thread. The latter was spawned to speculatively precompute L2 cache misses. Synchronization of the two threads is essential, in order to avoid the helper thread from running too far ahead, evicting useful data from the cache. The above two configurations were tested on memory intensive benchmarks, both with regular and irregular or random access patterns.

We experimented on Intel Xeon processor enabled with HT technology. The hardware characteristics can be found in table B.2 of appendix B. The performance monitoring capabilities of the processor were extended and a simple custom library was developed, so that the performance counters could be programmed to select events that are qualified by logical processors.

For each of the multithreaded execution modes, measurements consider, apart from the total execution time needed for real work to be completed, L2 misses, resource stall cycles (waiting to enter into the store buffer) and $\mu$ops retired. Note that in the pure software prefetch method, only the L2 misses of the working thread are presented. In all other cases and events, the sum of both threads (working and/or prefetching) is calculated.

We evaluate performance results using two computational kernels, Matrix Multiplication and LU-decomposition, and two NAS benchmarks, CG and BT. In Matrix Multiplication (MM) and LU-decomposition (LU), we present experimental results for $4096 \times 4096$ matrices, while in CG and BT we consider Class A problem sizes, (which means 14000 elements for CG and $64^3$ elements for BT). In MM and LU, we applied tiling choosing tiles that completely fit in L1 cache, as discussed in chapter 4. Furthermore, in MM we used blocked array layouts (non-linear layouts) with binary masks [AK04a] and applied loop unrolling. The implementations of

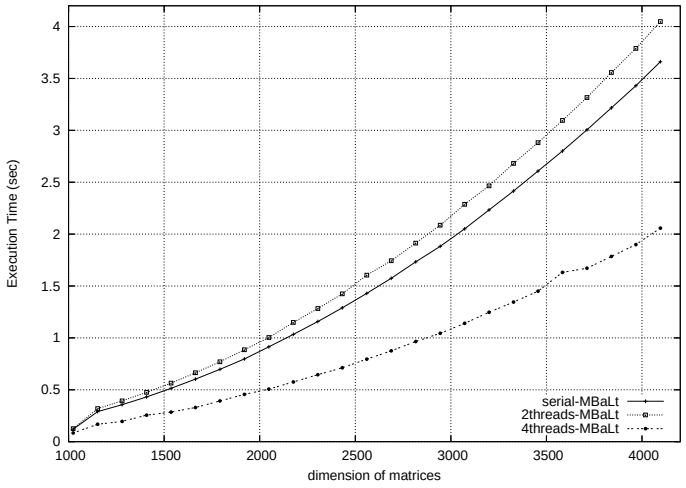



**Figure 6.26:** Xeon - The relative performance of the three different versions

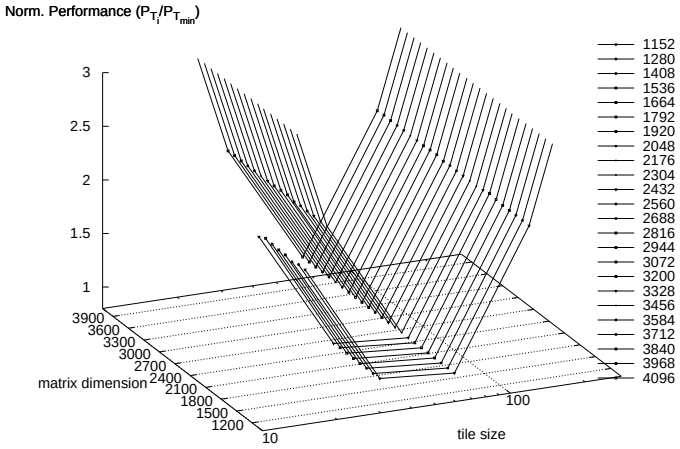



**Figure 6.27:** Xeon - Normalized performance of the matrix multiplication benchmark for various array and tile sizes (serial MBaLt)

CG (Conjugate gradient method for finding smallest eigenvalue of large-scale sparse symmetric positive definite matrix) and BT (CFD application using 5*5block ADI iteration) were based on the OpenMP C versions of NPB suite version 2.3 provided by the Omni OpenMP Compiler Project [Ope03]. We transformed these versions so that appropriate threading functions were used for work decomposition and synchronization, instead of OpenMP constructs. Both CG and BT are characterized by random memory access patterns, while the latter exhibits somewhat better data locality. All program codes were compiled with gcc 3.3.5 compiler using the O2 optimization level, and linked against glibc 2.3.2.

The TLP versions of the codes are based on coarse-grained work partitioning schemes (**tlp-coarse**), where the total amount of work is statically balanced across the participant threads (e.g., different tiles assigned to different threads in MM and LU). The SPR versions use prefetching to tolerate cache misses, following the scheme we described in section 5.3. In the pure prefetching version (**spr**), the whole workload is executed by just one thread, while the second is just a helper thread that performs prefetching of the next data chunk in issue. In the hybrid

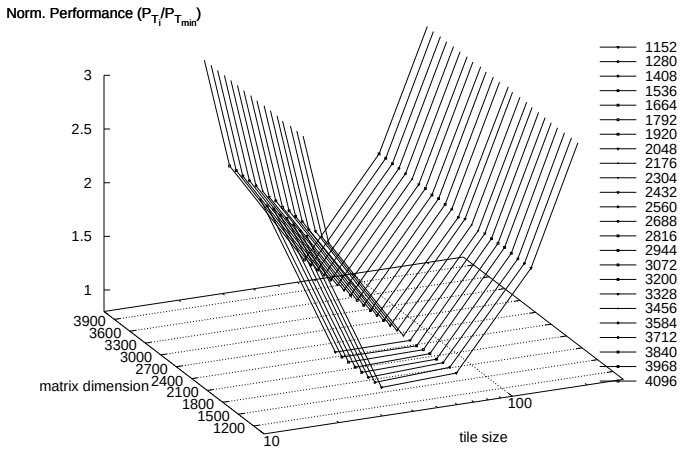



**Figure 6.28:** Xeon - Normalized performance of the matrix multiplication benchmark for various array and tile sizes (2 threads - MBaLt)

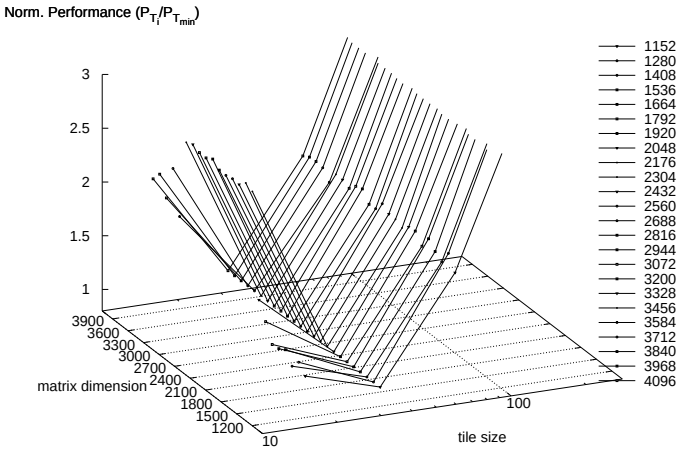
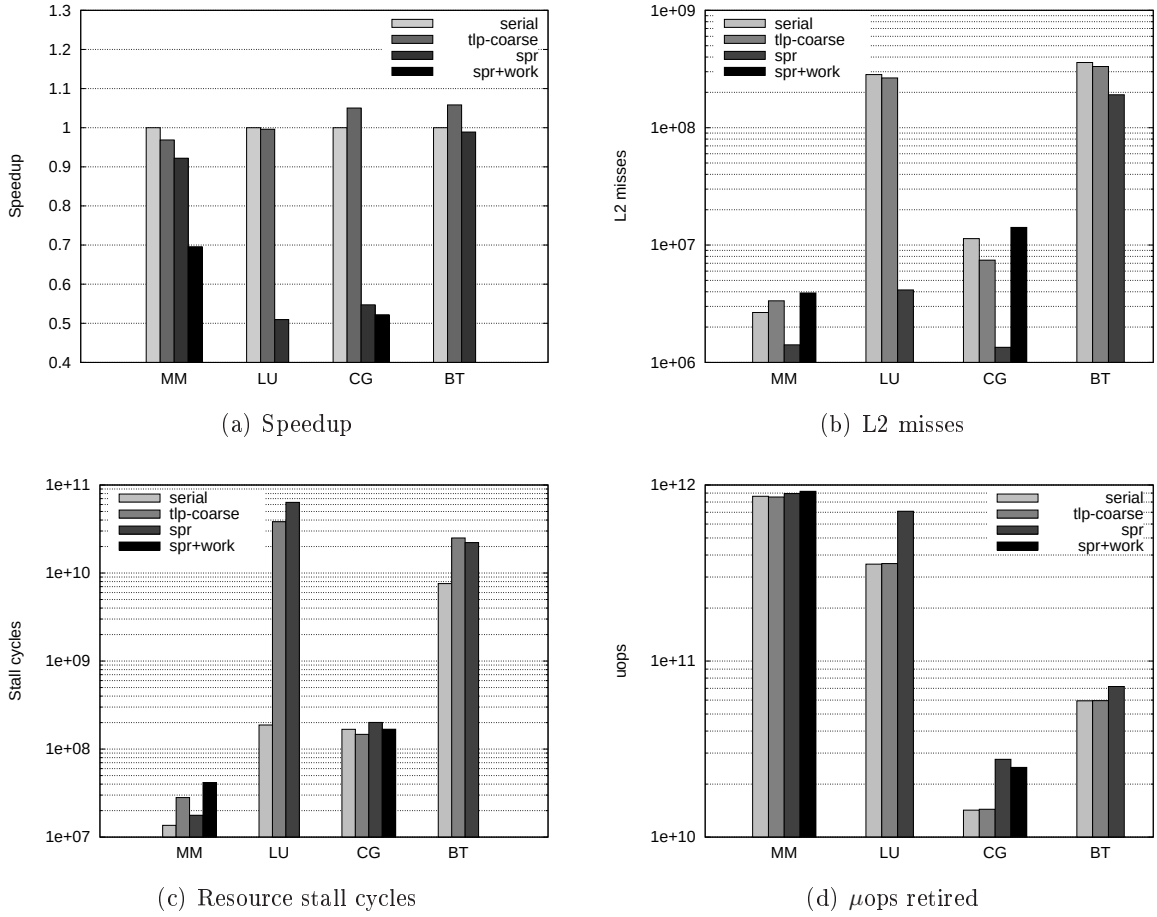



**Figure 6.29:** Xeon - Normalized performance of the matrix multiplication benchmark for various array and tile sizes (4 threads - MBaLt)

prefetching version (**spr+work**), the workload is partitioned in a more fine-grained fashion with both threads performing computations on the same data chunk, while one of them takes on the prefetching of the next data chunk. This latter parallelization scheme was applicable only in MM and CG.

Figure 6.30 presents the experimental results for our four benchmarks. HT technology favored a speedup of $5\% - 6\%$ only, compared to the serial optimized version, in the case of NAS benchmarks, when applying the TLP scheme. In the SPR versions, we achieved a really significant reduction in L2 misses of the working thread in all, apart from BT case. However, this reduction could not bring an overall execution time speedup. The implementation of speculative precomputation requires extra instructions to be executed, which means extra issue slots to be occupied. Normally, these instructions are hidden in the issue slots that would anyway remain idle. In optimized benchmarks, where there is already a quite high issuing rate of instructions, the idle issue slots are not enough for prefetching to be executed without burdening the issuing of instructions from the working thread. As Figure 6.30(d) depicts, in these cases,




(a) Speedup

(b) L2 misses

(c) Resource stall cycles

(d) $\mu$ops retired

**Figure 6.30:** SMT experimental results in the Intel Xeon Architecture, with HT enabled

there was a noticable increase in the total number of $\mu$ops, due to the instructions required to implement prefetching. For LU and CG, specifically, the total $\mu$ops were almost double than those of the *serial* case. As designated by the increased stall cycles in the *spr* case of all benchmarks compared to their *serial* versions, SPR method results in resource contention, and finally performance degradation.

### 6.3.1 Further Analysis

Table 6.1 presents the utilization of the busiest processor execution subunits, while running the reference applications. The first column (serial) contains results of the serial versions. The second column (tlp) presents the behavior of one of two threads for the TLP implementation (results for the other thread are identical). The third column (spr) presents statistics of the prefetching thread in the SPR versions. All percentages refer to the portion of the total instructions of each thread that used a specific subunit of the processor. The statistics were generated by profiling the original application executables using the Pin binary instrumentation tool [LCM+05], and analyzing for each case the breakdown of the dynamic instruction mix, as recorded by the

tool. Figure 6.31( [Int]) presents the main execution units of the processor, together with the issue ports that drive instructions into them. Our analysis examines the major bottlenecks that prevent multithreaded implementations from achieving some speedup.

Compared to the serial versions, TLP implementations do not generally change the mix for various instructions. Of course, this is not the case for SPR implementations. For the prefetcher thread, not only the dynamic mix, but also the total instruction count, differ from those of the worker thread. Additionally, different memory access patterns require incomparable effort for address calculations and data prefetching, and subsequently, different number of instructions.

| | | *Instrumented thread* | | |
|---|---|---|---|---|
| | EXECUTION UNIT | ***serial*** | ***tlp*** | ***spr*** |
| **MM** | ALU0+ALU1: | 27.06% | 26.26% | 37.56% |
| | FP_ADD: | 11.70% | 11.82% | 0.00% |
| | FP_MUL: | 11.70% | 11.82% | 4.13% |
| | MEM_LOAD: | 38.76% | 27.00% | 58.30% |
| | MEM_STORE: | 12.07% | 12.02% | 20.75% |
| | Total instructions ($\times 10^9$): | 4.59 | 2.27 | 0.20 |
| **LU** | ALU0+ALU1: | 38.84% | 38.84% | 38.16% |
| | FP_ADD: | 11.15% | 11.15% | 0.00% |
| | FP_MUL: | 11.15% | 11.15% | 0.00% |
| | MEM_LOAD: | 49.24% | 49.24% | 38.40% |
| | MEM_STORE: | 11.24% | 11.24% | 22.78% |
| | Total instructions ($\times 10^9$): | 3.21 | 1.62 | 3.26 |
| **CG** | ALU0+ALU1: | 28.04% | 23.95% | 49.93% |
| | FP_ADD: | 8.83% | 7.49% | 0.00% |
| | FP_MUL: | 8.86% | 7.53% | 0.00% |
| | FP_MOVE: | 17.05% | 14.05% | 0.00% |
| | MEM_LOAD: | 36.51% | 45.71% | 19.09% |
| | MEM_STORE: | 9.50% | 8.51% | 9.54% |
| | Total instructions ($\times 10^9$): | 11.93 | 7.07 | 0.17 |
| **BT** | ALU0+ALU1: | 8.06% | 8.06% | 12.06% |
| | FP_ADD: | 17.67% | 17.67% | 0.00% |
| | FP_MUL: | 22.04% | 22.04% | 0.00% |
| | FP_MOVE: | 10.51% | 10.51% | 0.00% |
| | MEM_LOAD: | 42.70% | 42.70% | 44.70% |
| | MEM_STORE: | 16.01% | 16.01% | 42.94% |
| | Total instructions ($\times 10^9$): | 44.97 | 22.49 | 8.40 |

**Table 6.1:** Processor subunits utilization from the viewpoint of a specific thread

In the MM benchmark the most specific characteristic is the large number of logical instructions used: at about 25% of total instructions in both serial and TLP versions. This is due to the implementation of blocked array layouts with binary masks that were employed for this

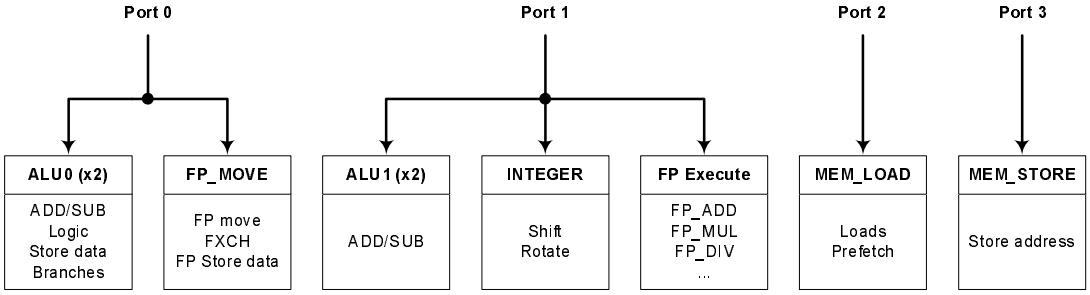



**Figure 6.31:** Instruction issue ports and main execution units of the Xeon processor

benchmark. Although the out-of-order core of the Xeon processor possesses two ALU units (double speed), among them only ALU0 can handle logical operations. As a result, concurrent requests for this unit in the TLP case, will lead to serialization of corresponding instructions, without offering any speedup. In the SPR case of LU, the prefetcher executes at least the same number of instructions as the worker, and also puts the same pressure on ALUs. This is due to the non-optimal data locality, which leads prefetcher to execute a large number of instructions to compute the addresses of data to be brought in cache. These facts translate into major slowdowns for the SPR version of LU, despite any significant L2 misses reduction.

As can be seen in Figure 6.30, TLP mode of BT benchmark was one of few cases that gave us some speedup. The relatively low usage and thus contention on ALUs, in conjunction with non-harmful co-existence of *faddmul* streams (as Table 5.3 depicts) which dominate other instructions, and the perfect workload partitioning, are among the main reasons for this speedup.

CHAPTER 7

# Conclusions

Due to the constantly dilating gap between memory latency and processor speed, most applications still waste much of their execution time, waiting for data to be fetched from main memory. Thus, hiding or minimizing the average memory latency has become a key challenge for achieving high performance. Memory hierarchy was introduced to alleviate this bottleneck, bringing the most recently used data close to the processor core. While caches reduce the memory latency, software techniques attempt to exploit locality of references for iterative codes to reduce cache misses.

This thesis addresses the question of how effective static (compile-time) software optimization techniques can be, in single-processor platforms. We deal with blocked array layouts, applied on numerical codes with iterative instruction streams. Of course, it does not suffice to identify the optimal blocked layout for each specific array. We also need an automatic and quick way to generate the mapping from the multidimensional iteration indices to the correct location of the respective data element in the linear memory. Blocked layouts are very promising, subject to an efficient address computation method. Any method of fast indexing for non-linear layouts will allow compilers to introduce such layouts along with row or column-wise ones, therefore further reducing memory misses.

## 7.1 Thesis Contributions

The key results of this thesis are the following:

1. In order to facilitate the automatic generation of tiled code that accesses blocked array layouts, we propose a very quick and simple address calculation method of the array indices. We can adopt any out of four different proposed types of blocked layouts, and apply a dilated integer indexing, similar to Morton-order arrays. Thus, we combine additional data locality due to blocked layouts, with fast access per any array element, since simple boolean operations are used to find its right location in linear physical memory. Since array

data are now stored block-wise, we provide the instruction stream with a straightforward indexing to access the correct elements. The method is very effective at reducing cache misses, since the deployment of the array data in memory follows the exact order of accesses by the tiled instruction code, achieved at no extra runtime cost.

2. We provide a theoretical analysis for the cache and TLB performance of blocked data layouts. Following this analysis, the optimal tile size that maximizes L1 cache utilization, should completely fit in the L1 cache, to avoid any interference misses. We prove that when applying optimization techniques, such as register assignment, array alignment, prefetching and loop unrolling, tile sizes equal to L1 capacity, offer better cache utilization, even for loop bodies that access more than just one array. Increased self- or/and cross-interference misses are now tolerated through prefetching. Such larger tiles also reduce lost CPU cycles due to less mispredicted branches.

3. We evaluated the efficiency of blocked array layouts combined with other software optimization techniques for various hardware platforms. The experimental and simulation results illustrate that specific optimizations should be matched with specific architectural characteristics. It is worth mentioning, that on SMT processors single-threaded versions exploit structural resources in an optimal way, issuing instruction streams at a high throughput, so that thread-level parallelism techniques could not bring any further performance improvement in the numerical iterative codes.

# Appendices

APPENDIX A

# Table of Symbols

| Explanation | symbol |
|---|---|
| L1 cache : | $C_{L1}$ |
| L1 cache line : | $L_1$ |
| L1 miss penalty: | $p_{L1}$ |
| total L1 cache misses : | $M_1$ |
| L2 cache : | $C_{L2}$ |
| L2 cache line : | $L_2$ |
| L2 miss penalty: | $p_{L2}$ |
| total L2 cache misses : | $M_2$ |
| TLB entries (L1): | $E$ |
| TLB entries (L2): | $E_2$ |
| page size : | $P$ |
| TLB miss penalty: | $p_{TLB}$ |
| total TLB misses : | $M_{TLB}$ |
| mispred. branch penalty: | $c_{br}$ |
| array size : | $N$ |
| tile size : | $T$ |
| tiles per array line : | $x = \frac{N}{T}$ |

**Table A.1:** Table of Symbols

APPENDIX B

# Hardware Architecture

| | UltraSPARC | Pentium III | SGI Origin R10K |
|---|---|---|---|
| CPU freq. : | 400MHz | 800MHz | 250MHz |
| L1 data cache : | 16KB | 16KB | 32 KB |
| L1 associativity: | direct-mapped | 4-way set asssoc. | 2-way set assoc. |
| L1 cache line : | 32B | 32B | 64B |
| L1 miss penalty: | 8 cc | 4 cc | 6 cc |
| L2 cache : | 4MB | 256KB | 4MB |
| L2 cache : | direct-mapped | 8-way set asssoc. | 2-way set assoc. |
| L2 cache line : | 64B | 64B | 128B |
| L2 miss penalty: | 84 cc | 60 cc | 100 cc |
| TLB entries (L1): | 64 addresses | 64 addresses | 64 addresses |
| TLB entries (L2): | | | |
| TLB asssociativity: | fully assoc. | 4-way set asssoc. | fully assoc. |
| page size : | 8KB | 4KB | 32KB |
| TLB miss penalty: | 51 cc | 5 cc | 57 cc |
| mispredicted branch penalty: | 4 cc | 10-15 cc | 4 cc |

**Table B.1:** Table of machine characteristics, used for experimentation

| | Athlon XP | Xeon |
|---|---|---|
| CPU freq. : | 2GHz | 2,8GHz |
| L1 cache : | 64KB | 16KB |
| | 2-way set asssoc. | 8-way set asssoc. |
| L1 line : | 64B | 64/128B |
| L1 miss penalty: | 3 cc | 4 cc |
| L2 cache : | 256KB | 1MB |
| | 16-way set asssoc. | 8-way set asssoc. |
| L2 line : | 64B | 64B |
| L2 miss penalty: | 20 cc | 18 cc |
| TLB entries (L1): | 40 addresses | 64 addresses |
| TLB entries (L2): | 256 addresses | |
| page size : | 4KB | 4KB |
| TLB miss penalty: | 3 cc | 30 cc |
| mispredicted branch penalty: | 10 cc | 20 cc |

**Table B.2:** Table of machine characteristics, used for experimentation

APPENDIX C

# Program Codes

In the following two types of codes are presented A. The initial codes, where only loop reordering has been applied, to achieve optimal loop nesting. B. The full optimized codes, just before fast indexing (with binary masks) for blocked array layouts is applied.

---

## C.1 Matrix Multiplication

A. (We consider that all arrays are row-wise stored)

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

B.

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

---

---

## C.2 LU decomposition

A. (We consider that all arrays are column-wise stored)

```
for (k = 0; k < N − 1; 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 = k + 1; i < N; i++)
      A[i][j]− = A[i][k] ∗ A[k][j];
}
```

B.

```
for (kk = 0; kk < N − 1; kk+ = step)
  for (jj = kk; jj < N; jj+ = step)
    for (ii = kk; ii < N; ii+ = step){
      if (ii > kk && jj > kk)
        kreturn=(N < (kk + step)?N : (kk + step));
      else kreturn = (N < (kk + step)?N : (kk + step)) − 1;
      for (k = kk; k < kreturn; k++){
        jstart = (k + 1 > jj?k + 1 : jj);
        istart = (k + 1 > ii?k + 1 : ii);
        if (jstart == k + 1)
          for (i = istart; i < ireturn; i++){
            A[i][k] = A[i][k]/A[k][k];
        for (j = jstart; (j < jj + step && j < N); j++)
          for (i = istart; (i < ii + step && i < N); i++)
            A[i][j]− = A[i][k] ∗ A[k][j];
      }
    }
```

## C.3 STRMM: Product of Triangular and Square Matrix

A. (We consider that all arrays are row-wise stored)

```
for (i = 0; i < N − 1; i++)
  for (k = i + 1; k < N; k++)
    for (j = 0; j < N; j++)
      B[i][j]+ = A[i][k] ∗ B[k][j];
```

B.

```
for (ii = 0; ii < N − 1; ii+ = step)
  for (kk = ii; kk < N; kk+ = step)
    for (jj = 0; jj < N; jj+ = step)
      for (i = ii; (i < ii + step &$ i < N − 1); i + +){
        kstart = (i + 1 > kk?i + 1 : kk);
        for (k = kstart; (k < kk + step && k < N); k + +)
          for (j = jj; (j < jj + step && j < N); j + +)
            B[i][j]+ = A[i][k] ∗ B[k][j];
      }
```

## C.4 SSYMM: Symmetric Matrix-Matrix Operation

A. (We consider that all arrays are column-wise stored)

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

B.

```
for (jj = 0; jj < N; jj+ = step)
  for (ii = 0; ii < N; ii+ = step)
    for (kk = 0; kk <= ii; kk+ = step)
      for (j = jj; (j < jj + step && j < N); j + +)
        for (i = ii; (i < ii + step && i < N); i + +){
          for (k = kk; (k < kk + step && k < i); k + +){
            C[k][j]+ = A[k][i] ∗ B[i][j];
            C[i][j]+ = A[k][i] ∗ B[k][j];
          }
          if (ii == kk)
            C[i][j]+ = A[i][i] ∗ B[i][j];
        }
```

## C.5 SSYR2K: Symmetric Rank 2k Update

A. (We consider that all array $C$ is row-wise stored, while arrays $A$, $B$ are column-wise stored)

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

B.

```
for (kk = 0; kk < N; kk+= step)
  for (ii = 0; ii < N; ii+= step)
    for (jj = ii; jj < N; jj+= step)
      for (k = kk; (k < kk + step && k < N); k++)
        for (i = ii; (i < ii + step && i < N); i++){
          jstart = (i > jj?i : jj);
          for (j = jstart; (j < jj + step && j < N); j++)
            C[i][j]+= B[j][k]*A[i][k] + A[j][k]*B[i][k];
        }
```

# Bibliography


[AAKK05] Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis, and Nectarios Koziris. Tuning Blocked Array Layouts to Exploit Memory Hierarchy in SMT Architectures. In *Proceedings of the 10th Panhellenic Conference in Informatics*, Volos, Greece, Nov. 2005.

[AAKK06a] Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis, and Nectarios Koziris. Exploring the Capacity of a Modern smt Architecture to Deliver High Scientific Application Performance. In *Proc. of the 2006 International Conference on High Performance Computing and Communications (HPCC-06)*, Munich, Germany, Sep 2006. Lecture Notes in Computer Science.

[AAKK06b] Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis, and Nectarios Koziris. Exploring the Performance Limits of Simultaneous Multithreading for Scientific Codes. In *Proc. of the 2006 International Conference on Parallel Processsing (ICPP-06)*, Columbus, Ohio, Aug 2006. IEEE Computer Society Press.

[AK03] Evangelia Athanasaki and Nectarios Koziris. Blocked Array Layouts for Multi-level Memory Hierarchies. In *Proceedings of the 9th Panhellenic Conference in Informatics*, pages 193–207, Thessaloniki, Greece, Nov. 2003.

[AK04a] Evangelia Athanasaki and Nectarios Koziris. Fast Indexing for Blocked Array Layouts to Improve Multi-Level Cache Locality. In *Proc. of the 8-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-8), In conjuction with the 10th International Symposium on High-Performance Computer Architecture (HPCA-10)*, pages 109–119, Madrid, Spain, Feb 2004. IEEE Computer Society Press.

[AK04b] Evangelia Athanasaki and Nectarios Koziris. Improving Cache Locality with Blocked Array Layouts. In *Proceedings of the 12-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP'04)*, pages 308–317, A Coruna, Spain, Feb. 2004. IEEE Computer Society Press.

[AK05] Evangelia Athanasaki and Nectarios Koziris. Fast Indexing for Blocked Array Layouts to Reduce Cache Misses. *International Journal of High Performance Computing and Networking (IJHPCN)*, 3(5/6):417–433, 2005.

[AKT05] Evangelia Athanasaki, Nectarios Koziris, and Panayiotis Tsanakas. A Tile Size Selection Analysis for Blocked Array Layouts. In *Proc. of the 9-th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9), In conjuction with the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 70–80, San Francisco, CA, Feb 2005. IEEE Computer Society Press.

[APD01] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, pages 52–61, Göteborg, Sweden, July 2001.

[BAYT01] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations. In *Proc. of the 15th Int. Conf. on Supercomputing (ICS '01)*, pages 486–500, Sorrento, Italy, June 2001.

[BAYT04] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems. *The Journal of Instruction-Level Parallelism*, 6:1–35, June 2004.

[BP04] James R. Bulpin and Ian A. Pratt. Multiprogramming Performance of the Pentium 4 with Hyper-Threading. In *Proceedings of the Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD 2004) held in conjunction with ISCA ŏ04*, page 53ŭ62, Munich, Germany, June 2004.

[CB94] T.-F. Chen and J.-L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*, pages 223–232, Chicago, IL, Apr 1994.

[Che95] T. Chen. An Effective Programmable Prefetch Engine for On-Chip Caches. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-28)*, pages 237–242, Ann Arbor, MI, Dec 1995.

[CJL+99] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *Proc. of the 13th ACM Int. Conf. on Supercomputing (ICS '99)*, pages 444–453, Rhodes, Greece, June 1999.

[CL95] Michael Cierniak and Wei Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proc. of the ACM SIGPLAN 1995 con-*

*ference on Programming language design and implementation*, pages 205–217, La Jolla, CA, June 1995.

[CLPT99] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proc. of the 11th Annual Symp. on Parallel Algorithms and Architectures (SPAA '99)*, pages 222–231, Saint Malo, France, June 1999.

[CM95] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proc. of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, June 1995.

[CM99] Jacqueline Chame and Sungdo Moon. A Tile Selection Algorithm for Data Locality and Cache Interference. In *Proc. of the 13th ACM Int. Conf. on Supercomputing (ICS '99)*, pages 492–499, Rhodes, Greece, June 1999.

[CTWS01] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic Speculative Precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, pages 306–317, Austin, TX, Dec 2001.

[CWT+01] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, pages 14–25, Göteborg, Sweden, July 2001.

[Ess93] K. Esseghir. Improving Data Locality for Caches. Master's thesis, Department of Computer Science, Rice University, Houston, Texas, Sept 1993.

[FST91] Jeanne Ferrante, V Sarkar, and W Thrash. On Estimating and Enhancing Cache Effectiveness. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*, Aug 1991.

[GMM99] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, July 1999.

[HK04] Chung-Hsing Hsu and Ulrich Kremer. A Quantitative Analysis of Tile Size Selection Algprithms. *The Journal of Supercomputing*, 27(3):279–294, Mar 2004.

[HKN+92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 136–145, Gold Coast, Australia, May 1992.

[HKN99] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Transactions Computers*, 48(10):1009–1024, Oct 1999.

[Int] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*. Order Number: 248966-012.

[JG97] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, pages 252–263, Denver, CO, June 1997.

[Jim99] Marta Jimenez. *Multilevel Tiling for Non-Rectangular Iteration Spaces*. PhD thesis, Universitat Politecnica de Catalunya, May 1999.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '90)*, pages 364–373, Seattle, WA, May 1990.

[Kan01] Mahmut Taylan Kandemir. Array Unification: A Locality Optimization Technique. In *Proc. of the 10th International Conference on Compiler Construction (CC'01)*, pages 259–273, Genova, Italy, Apr 2001.

[KCS+99] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, Feb 1999.

[KKO00] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. OőBoyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *Proc. of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–248, Philadelphia, Pennsylvania, Oct 2000.

[KLW+04] Dongkeun Kim, Shih-Wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John Paul Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004)*, pages 27–38, San Jose, CA, Mar 2004.

[KPCM99] Induprakas Kodukula, Keshav Pingali, Robert Cox, and Dror Maydan. An Experimental Evaluation of Tiling and Shackling for Memory Hierarchy Management. In *Proc. of the 13th ACM International Conference on Supercomputing (ICS '99)*, pages 482–491, Rhodes, Greece, June 1999.

[KRC97] M. Kandemir, J. Ramanujam, and A. Choudhary. A Compiler Algorithm for Optimizing Locality in Loop Nests. In *Proc. of the 11th International Conference on Supercomputing (ICS'97)*, pages 269–276, Vienna, Austria, July 1997.

[KRC99] M. Kandemir, J. Ramanujam, and Alok Choudhary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers*, 48(2):159–167, Feb 1999.

[KRCB01] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. A Layout-conscious Iteration Space Transformation Technique. *IEEE Transactions on Computers*, 50(12):1321–1336, Dec 2001.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.

[LLC02] Chun-Yuan Lin, Jen-Shiuh Liu, and Yeh-Ching Chung. Efficient Representation Scheme for Multidimensional Array Operations. *IEEE Transactions on Computers*, 51(03):327–345, Mar 2002.

[LM96] Chi-Keung Luk and Todd Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 222–233, Boston, MA, Oct 1996.

[LM99] Chi-Keung Luk and Todd Mowry. Automatic Compiler-Inserted Prefetching for Poiner-based Applications. *IEEE Transactions on Computers*, 48(2):134–141, Feb 1999.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[Luk01] Chi-Keung Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, pages 40–51, Göteborg, Sweden, July 2001.

[LW94] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, Oct 1994.

[MCFT99] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. ILP versus TLP on SMT. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Nov 1999.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(04):424–453, July 1996.

[MG91] Todd Mowry and Anoop Gupta. Tolerating Latency Trough Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[MHCF98] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the Multi-Level Nature of Tiling Interactions. *International Journal of Parallel Programming*, 26(6):641–670, Dec 1998.

[MPB01] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice Processors: an Implementation of Operation-Based Prediction. In *Proceedings of the 15th International Conference on Supercomputing (ICS '01)*, pages 321–334, Sorrento, Italy, June 2001.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, Boulder, CO, July 2003.

[Ope03] Omni OpenMP Compiler Project. Released in the International Conference for High Performance Computing, Networking and Storage (SC'03), Nov 2003.

[PHP02] Neungsoo Park, Bo Hong, and Viktor Prasanna. Analysis of Memory Hierarchy Performance of Block Data Layout. In *Proc. of the International Conference on Parallel Processing (ICPP 2002)*, pages 35–44, Vancouver, Canada, Aug 2002.

[PHP03] Neungsoo Park, Bo Hong, and Viktor Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(07):640–654, July 2003.

[PJ03] D. Patterson and J.Hennessy. *Computer Architecture. A Quantitative Approach*, pages 373–504. Morgan Kaufmann Pub., San Francisco, CA, 3rd edition, 2003.

[PNDN99] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and Alexandru Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Transactions on Computers*, 48(2):142–149, Feb 1999.

[RS01] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA '01)*, pages 37–48, Nuevo Leone, Mexico, Jan 2001.

[RT98a] Gabriel Rivera and Chau-Wen Tseng. Data Transformations for Eliminating Conflict Misses. In *Proc. of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, Montreal, Canada, June 1998.

[RT98b] Gabriel Rivera and Chau-Wen Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proc. of the 12th International Conference on Supercomputing (SC'98)*, pages 353–360, Melbourne, Australia, July 1998.

[RT99a] Gabriel Rivera and Chau-Wen Tseng. A Comparison of Compiler Tiling Algorithms. In *Proc. of the 8th International Conference on Compiler Construction (CC'99)*, pages 168–182, Amsterdam, The Netherlands, March 1999.

[RT99b] Gabriel Rivera and Chau-Wen Tseng. Locality Optimizations for Multi-Level Caches. In *Proc. of the 1999 ACM/IEEE Conference on Supercomputing (SC'99)*, CDROM article no. 2, Portland, OR, Nov 1999.

[SL99] Yonghong Song and Zhiyuan Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, Atlanta, Georgia, May 1999.

[SL01] Yonghong Song and Zhiyuan Li. Impact of Tile-Size Selection for Skewed Tiling. In *Proc. of the 5-th Workshop on Interaction between Compilers and Architectures (INTERACT'01)*, Monterrey, Mexico, Jan 2001.

[SPR00] Karthik Sundaramoorthy, Zachary Purser, and Eric Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 257–268, Cambridge, MA, Nov 2000.

[SU96] Ulrich Sigmund and Theo Ungerer. Identifying Bottlenecks in a Multithreaded Superscalar Microprocessor. In *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing-Volume II (Euro-Par '96)*, pages 797–800, Lyon, France, Aug 1996.

[TEE+96] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*, pages 191–202, Philadelphia, PA, May 1996.

[TEL95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 261–271, Nashville, Tennessee, May 1994.

[TGJ93] O. Temam, E. D. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (SC'93)*, pages 410–419, Portland, OR, Nov 1993.

[TT03] Nathan Tuck and Dean M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, New Orleans, LA, Sep 2003.

[TWN04] Filip Blagojevic Tanping Wang and Dimitrios S. Nikolopoulos. Runtime Support for Integrating Precomputation and Thread-Level Parallelism on Simultaneous Multithreaded Processors. In *Proceedings of the 7th ACM SIGPLAN Workshop on Languages, Compilers, and Runtime Support for Scalable Systems (LCR'2004)*, Houston, TX, Oct 2004.

[Ver03] Xavier Vera. *Cache and Compiler Interaction (how to analyze, optimize and time cache behaviour)*. PhD thesis, Malardalen University, Jan 2003.

[WAFG01] David S. Wise, Gregory A. Alexander, Jeremy D. Frens, and Yuhong Gu. Language Support for Morton-order Matrices. In *Proc. of the 2001 ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'01)*, pages 24–33, Snowbird, Utah, USA, June 2001.

[WF99] David S. Wise and Jeremy D. Frens. Morton-order Matrices Deserve Compilers' Support. TR533, CS Dept., Indiana University, Nov 1999.

[Wis01] David S. Wise. Ahnentafel Indexing into Morton-ordered Arrays, or Matrix Locality for Free. In *1st Euro-Par 2000 Int. Workshop on Cluster Computing*, pages 774–783, Munich, Germany, June 2001.

[WL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[WMC96] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Proc. of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 274–286, Paris, France, Dec 1996.

[WWW+02] H. Wang, P. Wang, R.D. Weldon, S.M. Ettinger, H. Saito, M. Girkar, S.S-W. Liao, and J. Shen. Speculative Precomputation: Exploring the Use of Multithreading for Latency. *Intel Technology Journal*, 6(1):22–35, Feb 2002.

[ZS01] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, pages 2–13, Göteborg, Sweden, July 2001.