Program Transformations for Cache Locality Enhancement on Shared-memory Multiprocessors

by

Naraig Manjikian

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Naraig Manjikian 1997

Abstract

Program Transformations for Cache Locality Enhancement
on Shared-memory Multiprocessors

Naraig Manjikian
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
1997

This dissertation proposes and evaluates compiler techniques that enhance cache locality and consequently improve the performance of parallel applications on shared-memory multiprocessors. These techniques target applications with loop-level parallelism that can be detected and exploited automatically by a compiler. Novel program transformations are combined with appropriate loop scheduling in order to exploit data reuse while maintaining parallelism and avoiding cache conflicts.

First, this dissertation proposes the *shift-and-peel* transformation for enabling loop fusion and exploiting reuse across parallel loops. The shift-and-peel transformation overcomes dependence limitations that have previously prevented loops from being fused legally, or prevented legally-fused loops from being parallelized. Therefore, this transformation exploits all reuse across loops without loss of parallelism.

Second, this dissertation describes and evaluates adaptations of static *loop scheduling strategies* to exploit wavefront parallelism while ensuring locality in tiled loops. Wavefront parallelism results when tiling is enabled by combining the shift-and-peel transformation with loop skewing. Proper scheduling exploits both intratile and intertile data reuse when independent tiles are executed in parallel on a large number of processors.

Third, this dissertation proposes *cache partitioning* for preventing cache conflicts between data from different arrays, especially when exploiting reuse across loops. Specifically, cache partitioning prevents frequently-recurring conflicts in loops with compatible data access patterns. Cache partitioning transforms the data layout such that there are no conflicts for reused data from different arrays during loop execution.

An analytical model is also presented to assess the potential benefit of locality enhancement. This model estimates the expected reduction in execution time by parameterizing the reduction in the number of memory accesses with locality enhancement and the contribution of memory accesses towards execution time.

Experimental results show that the proposed techniques improve parallel performance by 20%-60% for representative applications on contemporary multiprocessors. The results also show that significant improvements are obtained in conjunction with other performance-enhancing techniques such as prefetching. The importance of the techniques described in this dissertation will continue to increase as processor performance continues to increase more rapidly than memory performance.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Tarek S. Abdelrahman. For five years, he has maintained the right balance between providing close supervision and giving me the freedom to pursue my ideas, and he has given me much-appreciated encouragement and advice along the path of graduate studies. His attention to my work ultimately helped ensure its final quality.

I would also like to thank the members of my examination committee, Dr. David A. Padua, Dr. Zvonko G. Vranesic, Dr. Kenneth C. Sevcik, Dr. Todd C. Mowry, and Dr. Stephen D. Brown, for their careful reading and critical evaluation of my dissertation.

Access to the multiprocessor systems used in this research was provided by the University of Michigan Center for Parallel Computing. In particular, I would like to acknowledge the assistance of Andrew Caird of the Center of Parallel Computing.

I am grateful for the financial support that I have received during the course of my doctoral studies. I have been supported by a Postgraduate Scholarship from the Natural Sciences and Engineering Research Council of Canada, a University of Toronto Open Doctoral Fellowship, and a V. L. Henderson Memorial Fellowship. Additional financial support for this research was provided by the Natural Sciences and Engineering Research Council, and by the Information Technology Research Centre of Ontario.

I am also grateful for the opportunity to have participated in the NUMAchine Multiprocessor Project at the University of Toronto. I would like to thank the faculty, staff, and students involved in the NUMAchine Project for providing me with a rewarding experience.

I would like to thank the members of my extended family for their kindness and support, and for always providing me with a home away from home during the entire course of my university education.

To my parents, Hagop and Dirouhie, my brother, Sevak, and my sister, Lalai, words alone cannot convey my heartfelt gratitude. Despite my long absences, you supported all of my scholarly endeavors. You accepted the importance of my education, and you respected my decisions, although you gave me advice so that my direction was always clear. Let us always celebrate our successes together.

Contents

1 Introduction **1**

1.1 Large-Scale Shared-memory Multiprocessors . . . 1
1.2 Loop-level Parallelism and Parallelizing Compilers . . . 3
1.3 Data Reuse and Cache Locality . . . 4
1.4 Cache Locality Enhancement . . . 5
1.5 Research Overview . . . 6
1.6 Thesis Organization . . . 8

2 Background **9**

2.1 Loops and Loop Nests . . . 9
2.2 Loop Dependence Analysis . . . 11
2.2.1 Iteration Spaces, Iteration Vectors, and Lexicographical Ordering . . . 11
2.2.2 Definition and Use of Variables . . . 12
2.2.3 Data Dependence . . . 12
2.2.4 The Dependence Problem . . . 13
2.2.5 Dependence Tests . . . 14
2.3 Loop Parallelization and Concurrentization . . . 16
2.3.1 DOALL Loops and DOACROSS Loops . . . 16
2.3.2 Data Expansion and Privatization to Enable Parallelization . . . 17
2.3.3 Recognition of Induction and Reduction Variables . . . 18
2.3.4 Scheduling Loop Iterations . . . 19
2.4 Loop Transformations for Locality and Parallelism . . . 20
2.4.1 Data Reuse and Locality . . . 20

2.4.2 Degree and Granularity of Parallelism . . . 21
2.4.3 Unimodular Transformations . . . 21
2.4.4 Tiling . . . 23
2.4.5 Loop Distribution . . . 24
2.4.6 Loop Fusion . . . 25
2.5 Data Transformations . . . 26
2.5.1 Memory Alignment . . . 26
2.5.2 Array Padding . . . 26
2.5.3 Array Element Reordering . . . 27
2.5.4 Array Expansion and Contraction . . . 27
2.5.5 Array Merging . . . 28
2.6 Effectiveness of Locality Enhancement within Loop Nests . . . 28
2.6.1 Survey of Selected Studies . . . 28
2.6.2 Conclusions and Implications . . . 31

3 Quantifying the Benefit of Locality Enhancement 34
3.1 Overview of Model and Underlying Assumptions . . . 34
3.2 Quantifying Memory Accesses for Arrays . . . 36
3.3 Quantifying the Reduction in Memory Accesses with Locality Enhancement . 37
3.4 Quantifying the Impact of Locality Enhancement on Execution Time . . . 38
3.5 Potential Limitations of the Model . . . 40
3.6 Chapter Summary . . . 42

4 The Shift-and-peel Transformation for Loop Fusion 43
4.1 Loop Fusion . . . 43
4.1.1 Granularity of Parallelism and Frequency of Synchronization . . . 43
4.1.2 Quantifying the Benefit of Enhancing Locality with Fusion . . . 44
4.1.3 Dependence Limitations on the Applicability of Loop Fusion . . . 45
4.1.4 Related Work . . . 47
4.2 The Shift-and-peel Transformation . . . 48
4.2.1 Shifting to Enable Legal Fusion . . . 48

4.2.2 Peeling to Enable Parallelization of Fused Loops 49
4.2.3 Derivation of Shift-and-peel 50
4.2.4 Implementation of Shift-and-peel 54
4.2.5 Legality of the Shift-and-peel Transformation 57
4.3 Multidimensional Shift-and-peel 65
4.3.1 Motivation 65
4.3.2 Derivation 65
4.3.3 Implementation 66
4.3.4 Legality of Multidimensional Shift-and-peel 69
4.4 Fusion with Boundary-scanning Loop Nests 69
4.5 Chapter Summary 72

5 Scheduling Wavefront Parallelism in Tiled Loop Nests 73

5.1 Wavefront Parallelism in Tiled Loop Nests 73
5.1.1 Loop Skewing to Enable Legal Tiling 73
5.1.2 Enabling Tiling with the Shift-and-Peel Transformation 74
5.1.3 Wavefront Parallelism after Tiling 77
5.1.4 Exploiting Wavefront Parallelism: DOALL *vs.* DOACROSS 78
5.2 Data Reuse in Tiled Loop Nests 79
5.2.1 Intratile and Intertile Reuse 79
5.2.2 Quantifying the Locality Benefit of Tiling 80
5.2.3 Tile Size, Parallelism, and Locality 83
5.3 Related Work 83
5.3.1 Tiling 83
5.3.2 Loop Scheduling 84
5.3.3 Scheduling Vectors 85
5.4 Scheduling Strategies for Wavefront Parallelism 85
5.4.1 Dynamic Self-scheduling 86
5.4.2 Static Cyclic Scheduling 87
5.4.3 Static Block Scheduling 88

5.4.4 Comparison of Scheduling Strategies 89
5.4.4.1 Runtime Overhead for Scheduling 89
5.4.4.2 Synchronization Requirements 90
5.4.4.3 Parallelism and Theoretical Completion Time 90
5.4.4.4 Locality Enhancement 95

6 Cache Partitioning to Eliminate Cache Conflicts 98
6.1 Cache Conflicts 98
6.1.1 Cache Organization and Indexing Methods 98
6.1.2 Cache Conflicts for Arrays in Loops 99
6.1.3 Data Access Patterns and Cache Conflicts 100
6.1.4 Related Work 104
6.2 Cache Partitioning 105
6.2.1 Overview 106
6.2.2 One-dimensional Cache Partitioning 108
6.2.3 Multidimensional Cache Partitioning 111
6.2.4 Cache Partitioning for Multiple Loop Nests 117
6.3 Chapter Summary 119

7 Experimental Evaluation 122
7.1 Prototype Compiler Implementation 122
7.1.1 Compiler Infrastructure 123
7.1.2 Enhancements to Infrastructure 124
7.1.2.1 Support for High-level Code Transformations 124
7.1.2.2 Dependence Distance Information Across Loop Nests 125
7.1.2.3 Manipulation of Array Data Layout 125
7.2 Experimental Platforms 126
7.2.1 Hewlett-Packard/Convex SPP1000 and SPP1600 126
7.2.2 Silicon Graphics Power Challenge R10000 128
7.3 Codes Used in Experiments 129
7.4 Effectiveness of Cache Partitioning 131

7.5 Effectiveness of the Shift-and-peel Transformation 133
7.5.1 Results for Kernels 134
7.5.1.1 Derived Amounts of Shifting and Peeling 135
7.5.1.2 Multiprocessor Speedups 136
7.5.1.3 Impact of Problem Size on the Improvement from Fusion . 136
7.5.1.4 Comparison of Shift-and-peel with Alignment/replication . 138
7.5.2 Comparing Measured Performance Improvements with the Model . . 139
7.5.2.1 Determining the Sweep Ratios 139
7.5.2.2 Determining f_m and Applying the Model 143
7.5.3 Results for Applications 144
7.5.4 Combining Shift-and-peel with Prefetching 146
7.5.4.1 Results for Kernels 146
7.5.4.2 Results for Applications 150
7.5.5 Summary for the Shift-and-peel Transformation 153
7.6 Evaluation of Scheduling for Wavefront Parallelism 153
7.6.1 Results for `SOR` 154
7.6.2 Results for `Jacobi` 156
7.6.3 Results for `LL18` 160
7.6.4 Comparison of Sweep Ratios for Tiling 160
7.6.5 Summary for Evaluation of Scheduling Strategies 162

8 Conclusion **163**
8.1 Summary of Contributions 164
8.2 Future Work 165

Bibliography **167**

List of Tables

5.1 Comparison of scheduling strategies for tiling 89

7.1 Kernels and applications for experimental results 130

7.2 Kernels and applications used in experiments for the shift-and-peel transformation 134

7.3 Amounts of shifting and peeling for kernels 135

7.4 Characteristics of loop nest kernels 140

7.5 Revised sweep ratios to account for upgrade requests 142

7.6 Cache misses for parallel execution on Convex SPP1000 142

7.7 Comparison of estimated and measured improvement from fusion 143

7.8 Expected and measured cache misses for uniprocessor execution on Power Challenge 147

7.9 Expected and measured writebacks for uniprocessor execution on Power Challenge 148

7.10 Cache misses for parallel execution on Convex SPP1600 149

7.11 Cache misses and sweep ratios for `Jacobi` on Convex SPP1000 (16 processors) 161

List of Figures

1.1 Large-scale shared-memory multiprocessor architecture 2
1.2 Parallelism in loops 3
1.3 Data reuse in loops 4
1.4 Example of loop permutation 5

2.1 Classification of loop nest structure 10
2.2 A two-dimensional iteration space 12
2.3 Example formulation of the dependence problem for subscripted array references 15
2.4 A DOACROSS loop with explicit synchronization for loop-carried dependences 17
2.5 Scalar expansion to eliminate loop-carried antidependences 18
2.6 Induction variable recognition 19
2.7 Reduction variable recognition 19
2.8 Unimodular transformations 22
2.9 Example of tiling 24
2.10 Loop distribution and loop fusion 25

3.1 Illustration of memory accesses for arrays in loop nests 36
3.2 Graphical representation of $T = T_c + T_m$ and effect of locality enhancement . 39
3.3 Examples of loop nests accessing arrays with differing dimensionalities . . . 41

4.1 Example to illustrate fusion-preventing dependences 46
4.2 Example to illustrate serializing dependences 46
4.3 Shifting iteration spaces to permit legal fusion 49
4.4 Peeling to retain parallelism when fusing parallel loops 50
4.5 Algorithm for propagating shifts along dependence chains 52

4.6 Representing dependences to derive shifts for fusion 52
4.7 Algorithm for propagating peeling along dependence chains 53
4.8 Deriving the required amount of peeling 54
4.9 Dependence chain graph with dependences between non-adjacent loops . . . 54
4.10 Alternatives for implementing fusion with shift-and-peel 55
4.11 Complete implementation of fusion with shift-and-peel 56
4.12 Legality of the shift-and-peel transformation 57
4.13 Resolution of alignment conflicts with replication 59
4.14 Fusion with multidimensional shifting 67
4.15 Enumerating the number of cases for multidimensional shift-and-peel 67
4.16 Parallelization with multidimensional peeling 68
4.17 Independent blocks of iterations with multidimensional shift-and-peel 68
4.18 Fusing a loop nest sequence with a boundary-scanning loop nest 70

5.1 Steps in tiling the SOR loop nest 74
5.2 Graphical representation of skewing and tiling in the iteration space 75
5.3 Enabling tiling with the shift-and-peel transformation 76
5.4 Dependences and wavefronts 78
5.5 Exploiting parallelism with inner DOALL loops 79
5.6 Data reuse in a tiled loop nest that requires skewing 81
5.7 Amount of data accessed per tile with skewing 82
5.8 Impact of tile size on locality 84
5.9 Static cyclic scheduling of tiles 87
5.10 Static block scheduling of tiles 88
5.11 Wavefronts for dynamic and cyclic scheduling ($n_t = 4$,$P = 2$) 92
5.12 Wavefronts for block scheduling ($n_t = 4$,$P = 2$) 93
5.13 Variation of completion time ratio $R = T_{blk}/T_{cyc}$ 94
5.14 Number of data elements within a tile 96
5.15 Fraction of miss latency per tile 97

6.1 Cache organizations 99

6.2 Cache conflicts for arrays in loops 100
6.3 Taxonomy of data access patterns for arrays in a loop nest 101
6.4 Frequency of cross-conflicts for compatible data access patterns 102
6.5 Frequency of cross-conflicts for incompatible data access patterns 103
6.6 Example of cache partitioning to avoid cache conflicts 107
6.7 Conflict avoidance as partition boundaries move during loop execution 108
6.8 Greedy memory layout algorithm for cache partitioning 109
6.9 Memory overhead for 8 $N \times N$ arrays from cache partitioning (cache size=131,072) 110
6.10 Multidimensional cache partitioning 112
6.11 Interleaving partitions in multidimensional cache partitioning 113
6.12 The use of padding to handle wraparound in the cache 116
6.13 Greedy memory layout algorithm for multiple loop nests 120

7.1 Passes in the Polaris compiler 123
7.2 Architecture of the Convex SPP1000 127
7.3 Speedups for cache partitioning alone on Convex multiprocessors 131
7.4 Cache partitioning *vs.* array padding for `LL18` on Convex SPP1000 132
7.5 Impact of cache partitioning with fusion on Convex SPP1000/SPP1600 133
7.6 Speedup and misses of kernels on Convex SPP1000 137
7.7 Improvement in speedup with fusion for `LL18` and `calc` on Convex SPP1000 138
7.8 Performance of shift-and-peel *vs.* alignment/replication for `LL18` 139
7.9 Code for `LL18` loop nest sequence 140
7.10 Speedup for applications on Convex SPP1000 145
7.11 Uniprocessor speedups on Power Challenge 147
7.12 Multiprocessor speedups on Convex SPP1000 and SPP1600 (computed with respect to one processor on Convex SPP1000) 149
7.13 Average cache miss latencies on Convex SPP1600 150
7.14 `hydro2d` on Convex SPP1000 and SPP1600 151
7.15 `hydro2d` on SGI Power Challenge R10000 151
7.16 Cache misses for tiled `SOR` 155
7.17 Execution times for tiled `SOR` 155

7.18 Speedup for tiled `SOR` 156
7.19 The `Jacobi` kernel 157
7.20 Cache misses for tiled `Jacobi` 158
7.21 Normalized execution times for tiled Jacobi 158
7.22 Speedup for Jacobi 159
7.23 Normalized execution times for tiled LL18 159
7.24 Speedup for LL18 160

Chapter 1

Introduction

This dissertation presents new compiler techniques to improve the performance of automatically-parallelized applications on large-scale shared-memory multiprocessors. These techniques are motivated by key architectural features of large-scale multiprocessors—namely a large number of processors, caches, and a physically-distributed memory system—that must be exploited effectively to achieve high levels of performance. The proposed techniques focus on cache locality enhancement because caches are essential for bridging the widening gap between processor and memory performance, especially in large-scale multiprocessors. The techniques are designed specifically to enhance cache locality while maintaining the parallelism detected automatically by a compiler.

The remainder of this chapter is organized as follows. First, the architecture of large-scale shared-memory multiprocessors is presented. Next, loop-level parallelism and the role of parallelizing compilers are examined. Data reuse and cache locality in loops are then described. The limitations of existing techniques for cache locality enhancement are then outlined. Finally, an overview of the research conducted to overcome these limitations is presented.

1.1 Large-Scale Shared-memory Multiprocessors

Large-scale shared-memory multiprocessor systems have emerged in recent years as viable high-performance computing platforms [Bel92, LW95]. Built using high-speed commodity microprocessors, these systems are cost-effective platforms for a variety of applications rang-

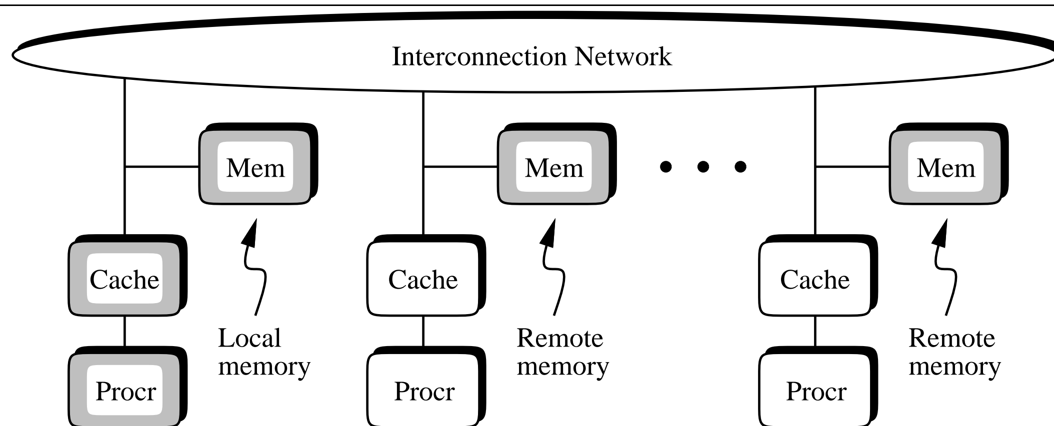


Figure 1.1: Large-scale shared-memory multiprocessor architecture

ing from scientific computation to on-line transaction processing. Examples of commercial multiprocessors include the HP/Convex Exemplar [Con94], the SGI/Cray Origin [Sil96a], and the Sun Ultra HPC [Sun96]. Examples of research multiprocessors include the Stanford FLASH [HKO+94] and the University of Toronto NUMAchine [VBS+95].

The architecture of large-scale shared-memory multiprocessors consists of processors, caches, physically-distributed memory, and an interconnection network. This architecture is shown in Figure 1.1. The memory is physically distributed to provide scalability as the number of processors is increased. Although the memory is physically distributed, the hardware supports a shared address space that allows processors to transparently access remote memory through the network. Because the remote access latency increases with distance, these multiprocessors have a non-uniform memory access (NUMA) architecture. High-speed caches are used to mitigate the long latency for accessing both local and remote memory, and hardware enforces coherence for copies of the same data in multiple caches.

A key factor that affects application performance on large-scale multiprocessors is the degree of *parallelism* [LW95]. Parallelism indicates that operations are independent and can be distributed among processors for simultaneous execution. A larger degree of parallelism allows more processors to be used, with commensurate reductions in execution time. Application performance on large-scale multiprocessors also depends on *cache locality* [LW95]. Locality ensures that processors access data from the nearby, high-speed cache, rather than the distant,

```
DO I = 1, N
  A[I] = S * A[I]
END DO
```

(a) Parallel loop

```
DO I = 2, N-1
  A[I] = (A[I-1]+A[I]+A[I+1]) / 3
END DO
```

(b) Serial loop

Figure 1.2: Parallelism in loops

slow memory. Although caches have long been used in uniprocessor systems, they are especially important for avoiding the large remote memory latency in large-scale multiprocessors.

1.2 Loop-level Parallelism and Parallelizing Compilers

Parallelism in programs, especially scientific programs, is often found in loops [ZC91], and this *loop-level parallelism* is exploited by distributing independent loop iterations among processors for simultaneous execution in order to reduce execution time.

Consider the loop shown in Figure 1.2(a). For any pair of different iterations i_1 and i_2 from the `I` loop, the elements `A[`i_1`]` and `A[`i_2`]` that are read and written in each iteration are different. Consequently, all iterations are independent of each other. The loop iterations may be distributed among multiple processors and executed simultaneously without violating the loop semantics. Loops whose iterations are independent of each other are called *parallel loops*.

In contrast, consider the loop shown in Figure 1.2(b). For successive pairs of iterations i and $i + 1$, the value read from element `A[`i`]` in iteration $i + 1$ is the same value written to element `A[`i`]` in iteration i. Hence, a *dependence* is said to exist between iterations i and $i + 1$. Successive iterations may not be executed simultaneously without violating the loop semantics. Loops with dependences between successive iterations are called *serial loops*.

Parallelizing compilers are software tools that detect and exploit loop-level parallelism. Many techniques have been developed to detect the absence of dependences and identify parallel loops [BGS94]. Therefore, these compilers can convert a sequential program into a parallel program by generating code in which independent iterations are distributed among processors for simultaneous execution. By automating parallelization, these compilers promote portability and allow programming in a machine-independent manner. Examples of commercial paral-

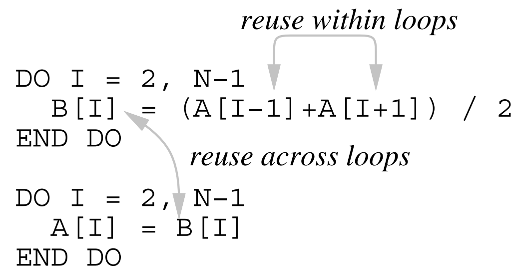


Figure 1.3: Data reuse in loops

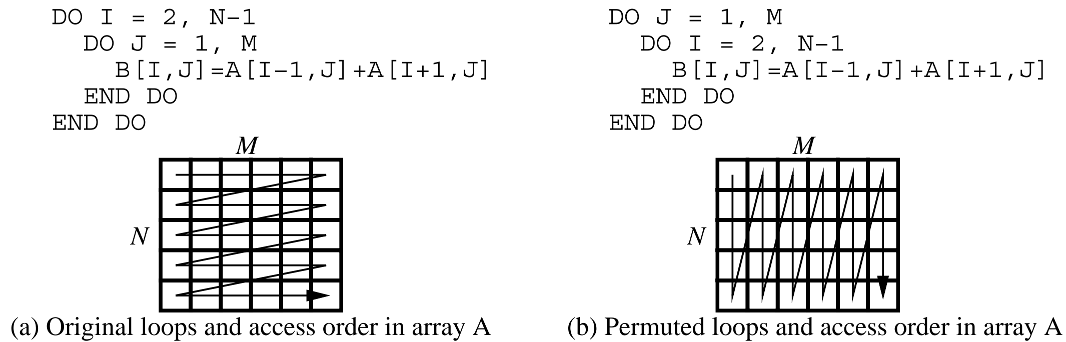
lelizing compilers include KAP [Kuc] and VAST [Pac]. Examples of parallelizing compilers used for research include SUIF [WFW+94] and Polaris [BEF+95].

1.3 Data Reuse and Cache Locality

Loops commonly exhibit *data reuse*, i.e., they read or write the same data elements multiple times. Performance is improved when this reuse is converted into *locality* in the cache. Locality reduces execution time by retaining data in the cache between uses in order to avoid long memory access latencies.

There are two types of data reuse for loops: reuse *within* loops and reuse *across* loops [BGS94, KM94, Wol92]. Figure 1.3 depicts sample code to illustrate the two types of reuse. In the first loop, iteration i reads array elements A[$i-1$] and A[$i+1$]. In iteration $i+2$, elements A[$i+1$] and A[$i+3$] are read. The array references cause element A[$i+1$] to be read in both iterations i and $i+2$. Because the same element is read in successive iterations of the same loop, this data reuse is said to be *within* a loop. In contrast, consider the references to array B. Iteration i of the first loop in Figure 1.3 writes array element B[i]. The value written to B[i] is subsequently read in iteration i of the *second* loop. Because the same element is referenced in iterations of different loops, this form of data reuse is said to exist *across* loops.

Although data reuse is common in loops, it may not necessarily lead to locality because of limited cache capacity and associativity [PH96]. If the amount of data accessed between uses exceeds the cache capacity, data is displaced from the cache before it is reused. Even when



(a) Original loops and access order in array A (b) Permuted loops and access order in array A

Figure 1.4: Example of loop permutation

the cache capacity is not exceeded between uses, reused array elements may still be displaced from the cache because of *cache conflicts*, which occur when cache lines containing different array elements are mapped into the same location in the cache because of limited associativity. Consider once again the example loops shown in Figure 1.3. The reuse of elements from array `B` is separated by a large number of iterations in different loops. If the cache capacity is not sufficient to contain all the elements of array `B` between uses, there is no locality. Even if the cache capacity is sufficient, locality may still be lost for the reuse of array `B` if the mapping in the cache for elements of array `A` conflicts with elements of array `B` between uses.

1.4 Cache Locality Enhancement

In order to enhance cache locality, compilers apply a variety of *loop transformations* such as permutation and tiling [BGS94]. These transformations reorder loop iterations to reduce the number of iterations between uses of the same data. Reuse often implies the existence of dependences between iterations, as described in Section 1.2. A transformation is *legal* if and only if the reordering of iterations obeys dependence constraints. Even if a transformation is legal, it is *beneficial* only if reordering of iterations improves cache locality.

Figure 1.4 depicts the effect of reordering iterations with loop permutation. In the original loops shown in Figure 1.4(a), the references `A[I-1,J]` and `A[I+1,J]` reuse elements in the same column of array `A`. However, the inner `J` loop reads elements in rows of the array. As a

result, reuse of elements in the same column is separated by $2 \cdot M$ inner loop iterations. The reuse is not converted into locality if the cache capacity is insufficient to hold elements of array A accessed between uses. However, if the loops are permuted to make the I loop innermost, as shown in Figure 1.4(b), reuse of an element in the same column is now separated by only 2 inner loop iterations, which increases the likelihood of achieving locality. Permutation is legal in this case because there are no dependences between iterations.

Despite the promise of locality-enhancing loop transformations, compilers using them often fail to improve performance [CMT94, Wol92]. The scope of transformations such as permutation is limited to exploiting the reuse within an individual loop. However, caches often generate locality for reuse within loops without requiring any compiler assistance [MT96]. In such cases, applying an iteration-reordering loop transformation provides no locality benefit.

On the other hand, caches normally cannot generate locality from reuse *across* loops because of the larger number of iterations between uses [MT96]. However, transformations for exploiting reuse across loops are restricted by dependences between iterations in different loops that may render transformation illegal [KM94]. Consequently, reuse across loops remains unexploited.

Even when a loop transformation to exploit reuse within or across loops is legal, parallelism may be reduced or lost as a result of iteration reordering [Wol92, KM94]. Consequently, there is a tradeoff between maintaining sufficient parallelism for many processors and enhancing locality with little or no resultant parallelism. Compilers seeking to parallelize applications for a large number of processors may therefore abandon locality for the sake of parallelism.

Finally, the locality benefit of any loop transformation is diminished by the occurrence of cache conflicts [CMT94, Wol92]. Conflicts displace data from the cache, and if the displaced data is later reused, the missing data must be reloaded into the cache. The latency of cache misses to reload data into the cache therefore increases execution time unnecessarily.

1.5 Research Overview

This dissertation proposes new techniques that improve parallel performance on large-scale multiprocessors by enhancing locality across parallel loops. These techniques enable trans-

formations to exploit reuse across loops and allow subsequent parallelization, even when dependences would otherwise prevent legal transformation or result in a serial loop. At same time, the benefit of locality enhancement is ensured by avoiding cache conflicts for reused data. The techniques are listed below with underlying assumptions.

- A code transformation called *shift-and-peel* is proposed for overcoming dependence limitations and exploiting reuse across a sequence of loops without sacrificing parallelism—specifically when the reuse within loops is captured by the cache on its own. This technique assumes uniform dependences between the loops in the sequence.

- An evaluation is provided for *loop scheduling strategies* for executing transformed loops on a large number of processors in a manner that ensures that the full benefit of locality enhancement is realized. The strategies are appropriate when the degree of available parallelism varies in the scheduled loops, and there is little or no variability in the units of work assigned to different processors.

- A data transformation called *cache partitioning* is proposed to prevent cache conflicts between data from different arrays, particularly when exploiting reuse across loops. This technique assumes that the arrays in the loops of interest are similarly-sized and traversed in the same manner, which is typical in most applications.

An analytical performance model is also presented to assess the impact of locality enhancement across loops on execution time and guide the application of the proposed techniques. The model assumes that loops have iteration space bounds that match array bounds, and that the computation performed in a loop accesses all elements in an array. A prototype implementation of the proposed techniques is described to demonstrate the feasibility of incorporating the techniques within a compiler. Finally, experimental results for representative applications on contemporary multiprocessors confirm that the proposed techniques provide substantial improvements in parallel performance.

1.6 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides background and surveys previous work on the effectiveness of existing locality enhancement techniques. Chapter 3 presents an analytical model to assess the impact of locality enhancement across loops on execution time. Chapter 4 presents the shift-and-peel transformation. Chapter 5 discusses loop scheduling strategies for parallel execution. Chapter 6 describes cache partitioning for conflict avoidance. Chapter 7 presents the results of an experimental evaluation to demonstrate the effectiveness of the proposed techniques. Finally, Chapter 8 offers conclusions and directions for future research.

Chapter 2

Background

This purpose of this chapter is twofold. First, it provides background on loop parallelization, loop transformations, and data transformations. Second, it reviews work on the effectiveness of existing locality-enhancing loop transformations.

This chapter is organized as follows. First, the structure and semantics of loops are defined. Dependence analysis is then described. Loop parallelization techniques are then described, followed by a review of loop transformations for enhancing locality and parallelism. Various data transformation techniques for arrays accessed in loop nests are described next. Finally, this chapter concludes by reviewing the effectiveness of existing techniques for enhancing locality within loops.

2.1 Loops and Loop Nests

A *DO*-loop (hereafter referred to simply as a *loop*) is a structured program construct consisting of a loop *header* and a loop *body*,

do $i = b_0, b_f, s$ $\quad\Leftarrow$ header
<*body*>(i)
end do

where i is the loop index variable, and b_0, b_f, s are integer-valued expressions that evaluate to constants on entry to the loop. The index variable takes on values beginning at b_0 in steps of s until the value b_f is exceeded, and each value represents one loop iteration. The loop body contains statements in which the variable i may appear, hence the body may be parameterized by i. A statement S within the loop body may also be parameterized as $S(i)$.

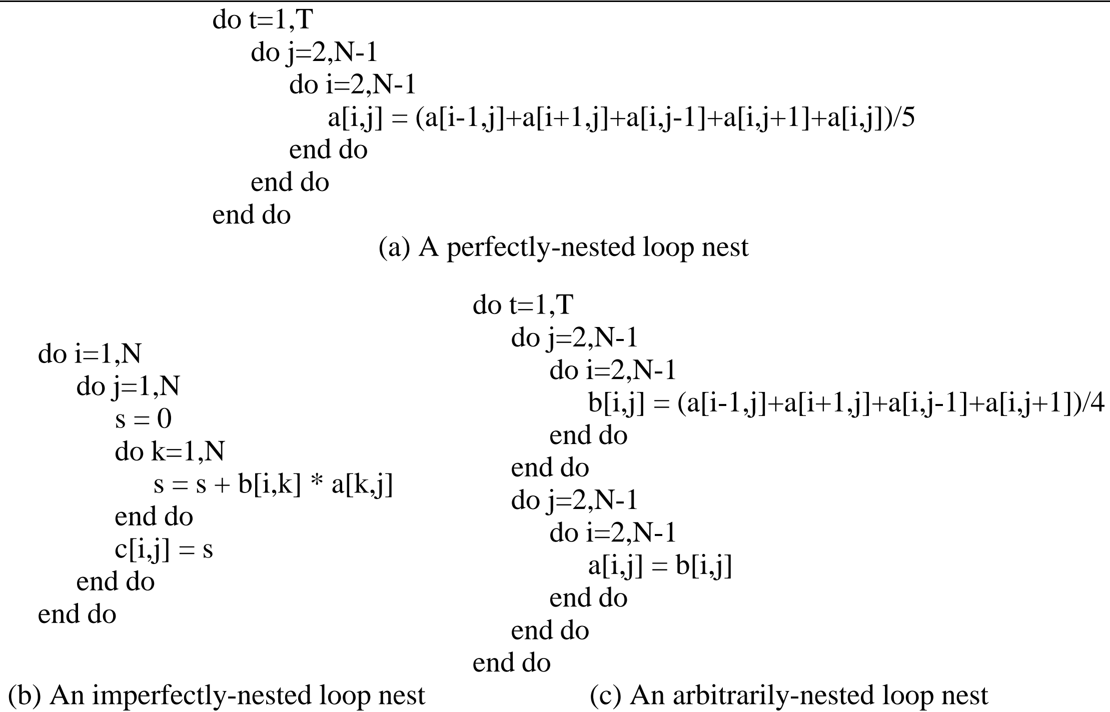


Figure 2.1: Classification of loop nest structure

A *loop nest* L is a set of loops and their respective bodies structured such that exactly one loop $\ell_{outer} \in L$ encloses *all* of the remaining loops, and no enclosing loop uses the same index variable as one of the loops it encloses. The *level* of a loop is the number of loops which enclose it. For example, the level of ℓ_{outer} is 0, since no other loop encloses it. The *depth* of the loop nest is one larger that the maximum level of any component loop, i.e., *depth*=$\left(\max_{\ell \in L} \mathit{level}(\ell)\right) + 1$.

A *perfectly-nested* loop nest consists of loops $\ell_0, \ell_1, \ldots, \ell_{m-1}$ such that:

$$\mathit{level}(\ell_i) = i, \ \forall\ 0 \le i \le m-1, \text{ and } \mathit{body}(\ell_i) = \{\ell_{i+1}\}, \ \forall\ 0 \le i < m-1.$$

The level of each loop is unique, and the body of each loop except the innermost loop consists of exactly one loop. All non-loop statements are in the body of the innermost loop. An example of a perfect loop nest is given in Figure 2.1(a).

An *imperfectly-nested* loop nest consists of loops $\ell_0, \ell_1, \ldots, \ell_{m-1}$ such that:

- $\mathit{level}(\ell_i) = i, \ \forall\ 0 \le i \le m-1, \quad \mathit{body}(\ell_i) = \{\ell_{i+1}\} \cup S_i, \ \forall\ 0 \le i < m-1,$
- $\exists\ 0 \le i < m-1 \ \ni\ S_i \ne \emptyset,$

where S_i is a set of zero or more non-loop statements. Hence, the only distinction between an imperfectly-nested loop nest and a perfectly-nested loop nest is the presence of at least one non-loop statement in the body of any loop except the innermost. An example of an imperfect loop nest is given in Figure 2.1(b).

An *arbitrarily-nested* loop nest consists of loops $\ell_0, \ell_1, \ldots, \ell_{m-1}$ such that:

- $\textit{level}(\ell_0) = 0, \quad \textit{level}(\ell_i) > 0, \ \forall\ 1 \leq i \leq m-1,$
- $\exists i, j \ \ni \ (1 \leq i \leq m-1) \wedge (1 \leq j \leq m-1) \wedge (i \neq j) \wedge (\textit{level}(\ell_i) = \textit{level}(\ell_j)).$

Hence, there are at least two loops with the same level. Apart from the requirement for exactly one outermost enclosing loop and the proscription against an enclosing loop using the same index variable as one of the loops it encloses, there are no other restrictions on the nesting structure of an arbitrarily-nested loop nest or the presence of non-loop statements. An example of an arbitrarily-nested loop nest is given in Figure 2.1(c).

This dissertation centers on perfectly-nested loop nests, and arbitrarily-nested loop nests with inner loop nests that are perfectly-nested, as shown in Figure 2.1(c).

2.2 Loop Dependence Analysis

The legality of loop parallelization or loop transformation is dictated by dependences between loop iterations. These dependences reflect the semantics of the original program. Consequently, *loop dependence analysis* to uncover these dependence relationships is an essential prerequisite for loop parallelization and transformation. The remainder of this section defines data dependence, formulates the dependence problem, and describes various dependence tests.

2.2.1 Iteration Spaces, Iteration Vectors, and Lexicographical Ordering

The loop bounds in a perfectly-nested loop nest of depth m define a set of points in an m-dimensional *iteration space* $\mathcal{I}$. It is assumed that the lower bound is one and the step is one for all loop variables.[1] The *iteration vector* $\vec{\imath} = (i_0, i_1, \ldots, i_{m-1}) \in \mathbf{Z}^m$ identifies points in $\mathcal{I}$,

[1] A transformation called *loop normalization* [ZC91], which is always legal, converts a loop into this form.

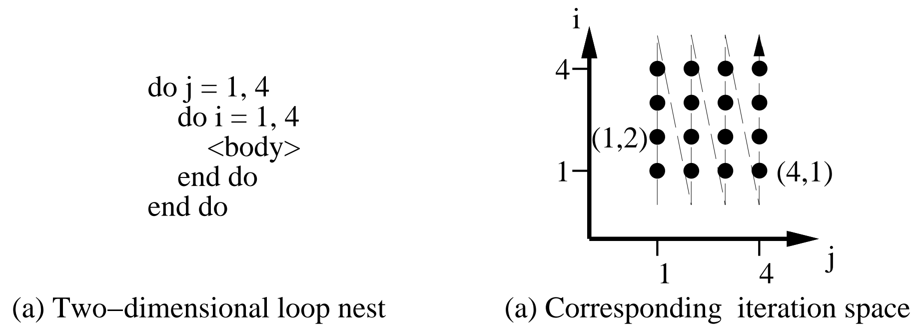


Figure 2.2: A two-dimensional iteration space

where $i_0, i_1, \ldots, i_{m-1}$ denote loop variables, and i_0 is the outermost loop variable. Figure 2.2 illustrates a representative two-dimensional iteration space; the iteration vector is (j, i).

The loop headers and their nesting order in a loop nest specify the sequence in which the points are traversed in the iteration space $\mathcal{I}$. The sequence of vectors corresponding to these points is called the *lexicographical order* of iterations [Wol92]. A pair of iteration vectors $\vec{p}, \vec{q}$ is ordered with the relation $\vec{p} \prec \vec{q}$ to reflect this lexicographical order. For example, the lexicographical order for the iteration space in Figure 2.2 is given by

$$(1,1) \prec (1,2) \prec \cdots \prec (3,4) \prec (4,1) \prec \cdots \prec (4,4)$$

and is represented by the path taken by the dashed line.

2.2.2 Definition and Use of Variables

Statements in the body of a loop may write (define) or read (use) program variables in each loop iteration. These program variables may be scalars or subscripted arrays. In the latter case, subscript expressions may contain index variables. For each statement instance $S(\vec{\imath})$ in the body of a perfectly-nested loop nest with iteration vector $\vec{\imath}$, $DEF(S(\vec{\imath}))$ denotes the set of variables that are written, and $USE(S(\vec{\imath}))$ denotes the set of variables that are read. These sets identify the *memory locations* read or written in each instance of the loop body.

2.2.3 Data Dependence

Data dependence is a relationship between statements that reference the same memory location. Let S and S' denote statements in the body of a perfectly-nested loop nest (the statements need

not be distinct), and let $\vec{p}$ and $\vec{q}$ denote points in the iteration space. The statement instance $S'(\vec{q})$ is *data dependent* on the statement instance $S(\vec{p})$ if the following conditions hold:[2]

1. $(\vec{p} \prec \vec{q}) \quad \vee \quad ((\vec{p} = \vec{q}) \wedge (S \neq S') \wedge (S$ appears before S' in the body$))$

2. $(\ DEF(S(\vec{p})) \ \cap \ USE(S'(\vec{q})) \ \ \neq \ \ \emptyset\ ) \vee (\ USE(S(\vec{p})) \ \cap \ DEF(S'(\vec{q})) \ \ \neq \ \ \emptyset\ ) \vee$ $(\ DEF(S(\vec{p})) \ \cap \ DEF(S'(\vec{q})) \ \ \neq \ \ \emptyset\ )$

The notation $S(\vec{p})\delta S'(\vec{q})$ indicates a data dependence. $S(\vec{p})$ is the dependence *source*, and $S'(\vec{q})$ is the *sink*. Similarly, $\vec{p}, \vec{q}$ are the source and sink iterations, respectively.

Dependences may be further categorized based on condition 2 above. A *true* dependence $S(\vec{p})\delta^t S'(\vec{q})$ exists if $DEF(S(\vec{p})) \cap USE(S'(\vec{q})) \neq \emptyset$ (write precedes read). An *anti*dependence $S(\vec{p})\delta^a S'(\vec{q})$ exists if $USE(S(\vec{p})) \cap DEF(S'(\vec{q})) \neq \emptyset$ (read precedes write). Finally, an *output* dependence $S(\vec{p})\delta^o S'(\vec{q})$ exists if $DEF(S(\vec{p})) \cap DEF(S'(\vec{q})) \neq \emptyset$ (write precedes write).

The dependence *distance* vector is given by $\vec{d} = \vec{q} - \vec{p}$. If $\vec{p} = \vec{q}$, then $\vec{d} = \vec{0}$. Otherwise, $\vec{p} \prec \vec{q}$ by definition and $\vec{d}$ must be lexicographically positive. The dependence *direction* vector is given by $\vec{s} = \mathrm{sig}(\vec{d})$ and is also lexicographically positive.

If $S(\vec{p})\delta S'(\vec{q})$ and $\vec{p} \neq \vec{q}$, then the dependence is *loop-carried* between the source and sink iterations. The *level* of a loop-carried dependence is given by scanning the dependence vector for the first non-zero component, starting with the element for the outermost loop. For a loop nest of depth m, the level ranges from 0 to $m-1$. If $S(\vec{p})\delta S'(\vec{q})$ and $\vec{p} = \vec{q}$, then the dependence is *loop-independent* because it exists within one instance of the loop body. The dependence level of a loop-independent dependence is ∞, since there are no non-zero components.

2.2.4 The Dependence Problem

The goal of dependence analysis is to solve the *dependence problem*: for two statement instances $S(\vec{p})$ and $S'(\vec{q})$, determine whether the statement instances access the same memory location. The solution is trivial for scalar variables because a data dependence will *always* exist if at least one of the statements writes the scalar variable. However, when $S(\vec{p})$ and $S'(\vec{q})$ access the same array variable, a mathematical formulation of the dependence problem is used to

[2]These conditions are conservative and may generate a superset of the actual dependences. Greater precision is obtained with an additional *covering* condition for writes [ZC91], although it is often ignored in practice.

determine if the same array *element* is accessed. In other words, the formulation determines if array subscript expressions are equal for any pair of statement instances.

The problem is simplified when the array subscripts consist only of *affine* expressions of the loop index variables. Affine expressions are linear combinations of variables with integer coefficients. For example, an affine expression for the iteration vector $\vec{\imath} = (i_0, i_1, \ldots, i_{m-1})$ is $a_0 \cdot i_0 + a_1 \cdot i_1 + \cdots + a_{m-1} \cdot i_{m-1} + c$, where $a_0, a_1, \ldots, a_{m-1}$ and c are integer constants. Affine subscripts for multidimensional arrays may be expressed as matrix-vector products. For example, the array reference $a[2i_0 - 2, 3i_1 - i_0 + 1]$ is represented by $a[f(\vec{\imath})]$, where

$$f(\vec{\imath}) = \begin{bmatrix} 2 & 0 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \end{bmatrix} + \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$

For a reference $a[f(\vec{\imath})]$ in $S(\vec{\imath})$ and a reference $a[f'(\vec{\imath})]$ in $S'(\vec{\imath})$ in a normalized perfectly-nested loop nest, the dependence problem is formulated succinctly as follows: find a pair of points $\vec{p}, \vec{q} \in \mathcal{I}$ such that

$$f(\vec{p}) = f'(\vec{q}). \tag{2.1}$$

Equation 2.1 expands into a system of linear equalities consisting of elements from $\vec{p}$ and $\vec{q}$. Since $\vec{p}, \vec{q}$ must lie in the iteration space, solutions are constrained by inequalities that reflect the iteration space bounds. In addition, solution vectors must consist of integers. Figure 2.3 illustrates the dependence problem formulation for an example two-dimensional loop nest (i.e., $m = 2$). with two statements that reference a one-dimensional array a.

A solution for Equation 2.1 that satisfies all constraints indicates the existence of a pair of statement instances that reference the same memory location. However, it is necessary to establish the *order* of the statement instances to properly establish the dependence relation. From Section 2.2.3, $S(\vec{p})\delta S'(\vec{q})$ implies that $\vec{p} \prec \vec{q}$. If the solution to Equation 2.1 is such that $\vec{p} \prec \vec{q}$, then the dependence relation must be $S(\vec{p})\delta S'(\vec{q})$. On the other hand, if the solution is such that $\vec{q} \prec \vec{p}$, then the dependence relation must be $S'(\vec{q})\delta S(\vec{p})$. If $\vec{p} = \vec{q}$, then the order of S and S' within the loop body determines the dependence relation.

2.2.5 Dependence Tests

Techniques for obtaining solutions to Equation 2.1 are called *dependence tests*. Some dependence tests apply only to restricted forms of the dependence problem, while others are generally

```
        do i0=1,5
           do i1=1,10
S:            a[2i0+i1-1] = ...
S':           ... = a[i0+i1]
           end do
        end do
```

(a) Two-dimensional loop nest

$$\begin{aligned} 2p_0 + p_1 - 1 &= q_0 + q_1 \\ p_0, p_1, q_0, q_1 &\in \mathbf{Z} \end{aligned}$$

$$\begin{array}{rcl} 1 \le p_0 & & p_0 \le 5 \\ 1 \le q_0 & & q_0 \le 5 \\ 1 \le p_1 & & p_1 \le 10 \\ 1 \le q_1 & & q_1 \le 10 \end{array}$$

(b) Dependence problem

Figure 2.3: Example formulation of the dependence problem for subscripted array references

applicable. All dependence tests must correctly report *in*dependence when they are applicable. The following paragraphs briefly describe a number of dependence tests.

Approximate dependence tests assume that a dependence exists whenever they are unable to prove independence. This assumption is required because approximate tests ignore or relax integer constraints in order to reduce the complexity of finding a solution.

The *gcd* (greatest common divisor) test [ZC91] examines the divisibility of the integer constants and coefficients in Equation 2.1 to prove independence. However, it requires a conservative assumption whenever it cannot prove independence because it ignores the inequality constraints bounding the iteration space.

The Banerjee test [Ban88] does consider the inequality constraints, hence it is useful whenever the *gcd* test is inclusive. However, the Banerjee test relaxes the integer solution constraints to provide a necessary and sufficient condition for the existence of a *real* solution within the bounds of iteration space. If no real solution exists, then independence is proven. However, the test is inclusive when a real solution does exist because the solution may not satisfy the original integer solution constraints.

Exact tests provide necessary and sufficient conditions for the existence of integer solutions. They do not require conservative assumptions because they either prove independence, or provide conditions for a data dependence.

The separability test [ZC91] is an exact test for a restricted form of the dependence problem where corresponding elements of $f(\vec{\imath})$ and $f'(\vec{\imath})$ in Equation 2.1 contain only one (and the

same) index variable. In this restricted form, this test either proves independence, or provides minimum and maximum dependence distances when a dependence exists. However, other tests must be used for those cases in which it is not applicable.

The Omega test [Pug92] is an efficient exact test for the general dependence problem. It proves independence, or provides distance information when a dependence exists. It also solves problems with symbolic constants to obtain conditions for the existence of a dependence; these conditions may be used as run-time dependence tests.

2.3 Loop Parallelization and Concurrentization

Loop parallelization and loop concurrentization designate loops whose iterations are executed on different processors [ZC91]. Parallelizable *DOALL* loops do not require synchronization between iterations, whereas concurrentized *DOACROSS* loops do require synchronization between iterations. In some cases, DOALL loops are obtained by variable expansion and privatization or by recognizing induction and reduction variables. Finally, loop scheduling specifies the execution order of iterations on each processor. The remainder of this section discusses these topics in more detail.

2.3.1 DOALL Loops and DOACROSS Loops

Let $DEP(L)$ denote all dependence distance or direction vectors for a perfectly-nested loop nest L. A loop $\ell \in L$ does not carry a dependence if and only if $\mathit{level}(d) \neq \mathit{level}(\ell),\ \forall\ d \in DEP(L)$. Such a loop is a DOALL loop and may be parallelized by distributing iterations arbitrarily among processors with no synchronization between iterations.

Although DOALL loops carry no dependences, they may be enclosed by other loops that carry dependences, or they may be preceded or followed by statements that must be executed serially. Synchronization outside the DOALL loop is required to preserve the original program semantics. This synchronization is normally provided before and after the loop with a *barrier* that forces each processor to wait until all processors are ready to proceed.

The iterations of a loop that carries a dependence may still be distributed among parallel processors through loop concurrentization. Such a loop is a DOACROSS loop and requires explicit synchronization between dependent iterations to preserve the original program semantics.

```
                          doacross i=4,N
do i=4,N                      if (i>6) wait(i-3)
    a[i] = f(a[i-3])   ⟹      a[i] = f(a[i-3])
end do                        if (i<N-2) signal(i)
                          end do
```

Figure 2.4: A DOACROSS loop with explicit synchronization for loop-carried dependences

Semaphores provide the required synchronization, with one semaphore per dependence edge. A semaphore *wait* operation immediately before the sink statement instance of a loop-carried dependence is paired with a semaphore *signal* operation immediately after the source of the dependence. The *wait* operation suspends execution until the corresponding *signal* operation has been performed. Figure 2.4 provides an example of a DOACROSS loop with explicit synchronization that allows three loop iterations to be executed in parallel at any time. In the worst case, dependences may serialize all iterations in a DOACROSS loop.

2.3.2 Data Expansion and Privatization to Enable Parallelization

A *true* loop-carried dependence $S(\vec{p})\delta^t S'(\vec{q})$ implies that the memory location written in iteration $\vec{p}$ is subsequently read in iteration $\vec{q}$. This inherently-serial dependence relationship prevents the iterations $\vec{p}$ and $\vec{q}$ from being executed simultaneously.

On the other hand, a loop-carried *anti*dependence $S(\vec{p})\delta^a S'(\vec{q})$ implies that a memory location is read in iteration $\vec{p}$ and then *over*written with new data in iteration $\vec{q}$. The antidependence would cease to exist if the read and write were performed on different memory locations. This observation provides the key insight into variable *expansion* and *privatization*.

Scalar expansion removes loop-carried antidependences caused by a scalar variable. The scalar variable is replaced with an array containing as many elements as loop iterations, as shown in Figure 2.5. Each array element is accessed by only one iteration, hence the loop-carried antidependence is eliminated without violating any dependences within a single iteration. *Array expansion* extends this technique to arrays by increasing array dimensionality and introducing as many elements in the new dimension as loop iterations.

Scalar privatization eliminates loop-carried antidependences by associating a private vari-

```
do i=1,N                      doall i=1,N
    s = ...                       s_exp[i] = ...
    a[i] = s        ⟹             a[i] = s_exp[i]
end do                        end do
```

Figure 2.5: Scalar expansion to eliminate loop-carried antidependences

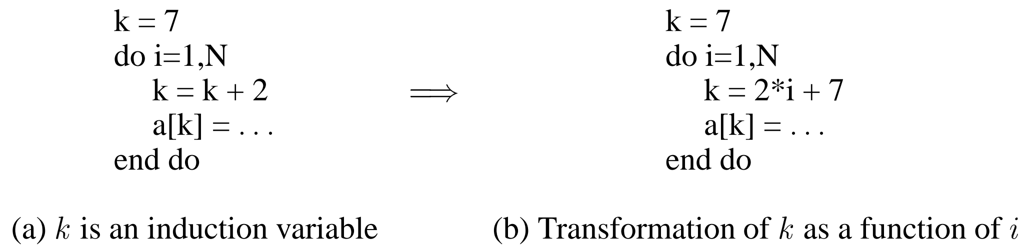
able with each loop iteration. When multiple iterations are assigned to the same processor, multiple private variables are collapsed into one variable per processor. *Array privatization* is a similar technique where a private array is associated with each loop iteration. Once again, multiple private arrays may be collapsed into a single private array per processor.

Both expansion and privatization must preserve true dependences flowing outside the loop. For sequential loop semantics, there is a *final* value associated with each scalar or array element. When a loop is parallelized, final values for privatized or expanded variables must be preserved by copying each value from the expanded array or the appropriate private version *before* executing any code following the loop.

2.3.3 Recognition of Induction and Reduction Variables

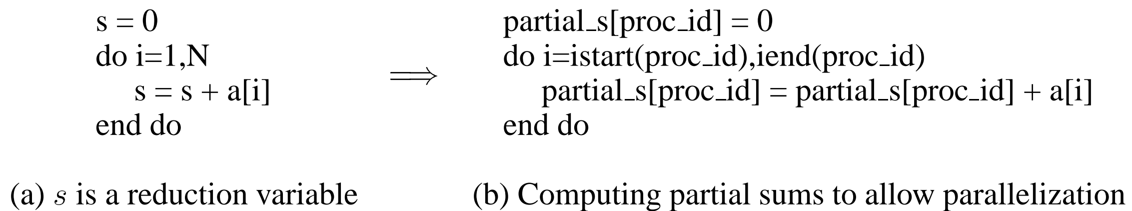
An *induction* variable is a variable that causes a loop-carried dependence, but whose value is implicitly a function of enclosing loop index variables. An example of an induction variable is given in Figure 2.6(a). The variable k causes a *true* loop-carried dependence because it is read then written in each iteration. However, the sequence of values for k is easily expressed as a function of i. Once this relationship is recognized, the assignment to k is replaced with a function of i, as shown in Figure 2.6(b). There is still a loop-carried dependence for k, but now it is an *anti*dependence that is easily resolved with privatization.

An *reduction* variable is a variable whose value is computed in each loop iteration using an associative operator. An example of an reduction variable is given in Figure 2.7(a). The variable s causes a *true* loop-carried dependence by summing elements from array a. However, partial sums may be computed in parallel on each processor, as shown in Figure 2.7(b), because addition is associative. After all partial sums are computed, one processor performs the final



(a) k is an induction variable (b) Transformation of k as a function of i

Figure 2.6: Induction variable recognition



(a) s is a reduction variable (b) Computing partial sums to allow parallelization

Figure 2.7: Reduction variable recognition

addition of all partial sums.[3] Reductions involving other associative operators such as minimum or maximum are treated similarly.

2.3.4 Scheduling Loop Iterations

Scheduling of DOALL and DOACROSS loop iterations specifies the distribution and execution order on each processor. DOALL loops have no constraints on execution order. However, a subset of DOACROSS loop iterations assigned to the same processor must be executed in lexicographical order to ensure that one processor can always execute the iteration that lexicographically precedes any dependent iterations. Irrespective of any constraints, a schedule should balance the workload for best performance.

In *static* scheduling, the distribution and execution order of iterations are determined at compile-time, hence no run-time overhead is incurred. Static scheduling is most effective when

[3]Although addition is mathematically associative, changing the order of summation may produce slightly different numerical results on real hardware due to rounding in floating-point arithmetic.

there is no variance in the amount of computation between iterations, or when the variance is known at compile time. The most common schedules are *block*, *cyclic*, and *block-cyclic*. For n iterations and p processors (with $n \geq p$), block distribution assigns a contiguous subset of $\lfloor n/p \rfloor$ iterations to each processor except the last, which is assigned $n - (p-1) \cdot \lfloor n/p \rfloor$ iterations. Cyclic distribution assigns the ith iteration to processor $(i \bmod p)$. Block-cyclic distribution assigns contiguous subsets of fewer than $\lfloor n/p \rfloor$ iterations to p processors in a cyclic manner.

Loop iterations may also be scheduled *dynamically* at run time. The most common approach is *self-scheduling*, where processors extract iterations atomically from one or more subsets of iterations. Self-scheduling is most effective when the variance in the amount of computation for different iterations is high, or when the variance is unknown at compile time. There are a number of self-scheduling algorithms [HSF92]. In the simplest algorithm, processors obtain one iteration at a time from a single set. More elaborate algorithms provide one subset of iterations per processor and permit iterations to be transferred between subsets to balance workloads.

2.4 Loop Transformations for Locality and Parallelism

A loop transformation reorders loop iterations in order to enhance locality or parallelism [PW86, ZC91]. The legality of loop transformations is dictated by dependences, just as it is for parallelization. This section first describes the relationship between data reuse and locality, then characterizes the degree and granularity of parallelism in loops. Various loop transformations for enhancing locality and parallelism are then discussed.

2.4.1 Data Reuse and Locality

Data reuse is an inherent characteristic of programs. Locality in the memory hierarchy results when processors obtain reused data from nearby (i.e., faster) levels of the hierarchy, specifically the cache. Locality reduces the effective memory access latency and thereby reduces total execution time. Temporal locality results from reuse of the same data item, whereas spatial locality results from reuse of different data items in the same cache line.

There are, however, a number of obstacles for achieving cache locality. First, the cache

capacity is limited, hence reused data may be displaced from the cache if the cache capacity is exceeded between uses. Second, the cache associativity is limited, hence reused data may be displaced by mapping conflicts in the cache, even if there is sufficient cache capacity. Finally, false sharing occurs when two different processors write different elements of the same cache line, and the affected cache line is repeatedly exchanged between the two processor caches.

In a loop, temporal and spatial reuse may occur between iterations as well as within iterations. The goal of locality enhancement is to increase the likelihood of converting reuse into locality by: (a) reducing the number of iterations between uses, (b) reducing the occurrence of the cache conflicts, or (c) limiting the extent of false sharing.

2.4.2 Degree and Granularity of Parallelism

The *degree* of parallelism in a DOALL loop is equal to the number of iterations because the iterations are independent of each other. For DOACROSS loops, the degree of parallelism is constrained by synchronization; in the worst case, there is no parallelism (i.e., degree of parallelism is 1). The *granularity* of parallelism is the amount of computation per parallel loop iteration. For example, the nesting level of a single DOALL loop within a perfectly-nested loop nest dictates the granularity of parallelism.

Loop transformations for enhancing parallelism control the degree and granularity of parallelism that is actually exploited. For example, positioning two or more DOALL loops in a perfectly-nested loop nest adjacent to each other makes the total available parallelism equal to the product of the degrees of parallelism of each DOALL loop. Furthermore, positioning DOALL loops at outer nesting levels increases the granularity of parallelism.

2.4.3 Unimodular Transformations

Unimodular transformations [Ban93, Wol92] are applied to perfectly-nested loop nests with affine loop bounds and array subscripts. These transformations are represented as invertible unimodular matrices whose determinants are ± 1. Three elementary loop transformations—permutation, reversal, and skewing—may be represented with unimodular matrices, as shown in Figure 2.8. A compound transformation is formed with a product of elementary unimodular matrices, and the resulting matrix remains unimodular. A unimodular transformation is applied

```
do i=1,L                          do i=1,L
  do j=1,M                          do k=1,N
    do k=1,N                          do j=1,M
      <body>          ==>               <body>
    end do                            end do
  end do                            end do
end do                            end do
```

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

(a) Loop permutation and corresponding unimodular matrix

```
do i=1,L                          do i=1,L
  do j=1,M                          do j=-M,-1
    do k=1,N                          do k=1,N
      <body>          ==>               <body>
    end do                            end do
  end do                            end do
end do                            end do
```

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

(b) Loop reversal and corresponding unimodular matrix

```
do i=1,L                          do i=1,L
  do j=1,M                          do j=1+i,M+i
    do k=1,N                          do k=1,N
      <body>          ==>               <body>
    end do                            end do
  end do                            end do
end do                            end do
```

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

(c) Loop skewing and corresponding unimodular matrix

Figure 2.8: Unimodular transformations

by multiplying the corresponding matrix with the iteration vector to yield the new iteration vector. Loop bounds are transformed in a similar manner. However, array subscript expressions are transformed by using the *inverse* of the matrix.

Unimodular transformations enhance locality and parallelism, primarily by permuting or skewing loops in a loop nest. Loop permutation enhances locality by reducing the number of iterations between uses of the same data. Permutation also enhances the granularity of parallelism by moving DOALL loops to the outermost position. When a loop nest contains no DOALL loops, but parallelism exists along *wavefronts* in the iteration space, loop skewing

obtains a new iteration space where the parallelism is captured in a DOALL loop.

Testing the legality of a unimodular transformation is straightforward. Dependence vectors are transformed in the same manner as the iteration vector with a matrix-vector product. Since iterations in the transformed space are traversed in lexicographical order, the transformed dependence vectors must remain lexicographically positive for the transformation to be legal.

2.4.4 Tiling

Tiling (also known as blocking) combines strip-mining of inner loops with loop permutation [BGS94, Wol92]. Strip-mining encloses a loop with a new control loop that iterates between the original loop bounds in steps of B. The original loop executes B iterations starting at each value of the enclosing loop index variable. Tiling is completed by permuting the control loop to the outermost level, as shown in Figure 2.9(b).

Tiling is legal if and only if the strip-mining and loop permutation are legal. Strip-mining alone is *always* legal; the loop nest dimensionality is increased, but the iterations are traversed in the same order. Strip-mining expands each dependence vector by inserting a zero in the position corresponding to the control loop index. Furthermore, for each original vector with a non-zero element for the original loop index, a new vector is introduced. The new dependence vector is copied from the transformed dependence, then the element corresponding to the control loop index is set to B or $-B$, depending on the sign of the component corresponding to the original loop index.

The legality of permutation is determined just as in unimodular transformations. If any transformed dependence vector after permutation is not lexicographically positive, then tiling is not legal. Since permutation moves control loops to the outermost level, tiling is legal only if strip-mining does not introduce negative elements into the transformed dependence vectors.

Tiling enhances locality by reducing the number of iterations between uses of the same data, as shown in Figure 2.9. The outermost loop in Figure 2.9(a) carries reuse, and tiling inner loops as shown in Figure 2.9(b) exploits this reuse. Figures 2.9(c) and (d) graphically illustrate the reuse before and after tiling. Locality is enhanced with tiling because fewer data elements are accessed between uses.

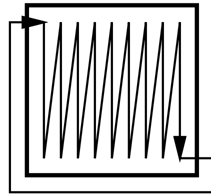
Tiling enhances the granularity of parallelism by permuting parallel control loops to the

```
do t=1,T
    do j=1,N
        do i=1,N
            . . . = a[i,j]
        end do
    end do
end do
```

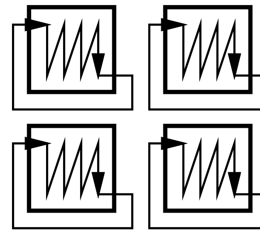
(a) Original loop nest

```
do jj=1,N,B
    do ii=1,N,B
        do t=1,T
            do j=jj,min(jj+B−1,N)
                do i=ii,min(ii+B−1,N)
                    . . . = a[i,j]
                end do
            end do
        end do
    end do
end do
```

(b) Loop nest after tiling loops j and i



(c) Original data accesses in array a



(d) Tiled data accesses in array a

Figure 2.9: Example of tiling

outermost level. For example, if loops j and i in Figure 2.9(a) are parallel, the control loops jj and ii are also parallel, but each control loop iteration executes many inner loop iterations. However, the degree of parallelism in each control loop is reduced by a factor of B. Hence, there is a tradeoff between the degree and granularity of parallelism.

2.4.5 Loop Distribution

Loop distribution transforms a single loop into one or more loops containing statements from the original loop body, as illustrated in Figure 2.10. As a result, the order of statement instances is altered substantially from the original loop.

Loop distribution is primarily used to enhance parallelism by obtaining one or more DOALL loops from a serial loop that carries dependences. If the dependences flow between different statements, then loop distribution places the source statement in one loop and the sink statement in another loop, and the resulting loops no longer carry dependences. Loop distribution also

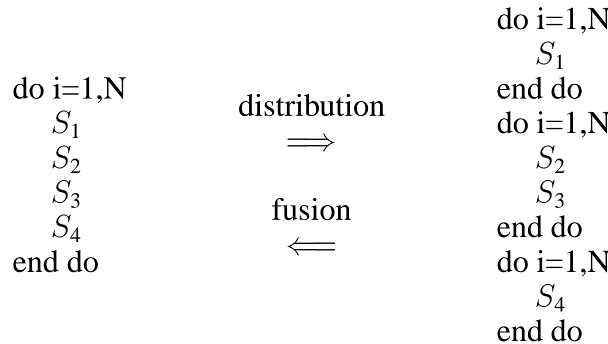


Figure 2.10: Loop distribution and loop fusion

enhances locality by reducing the amount of data accessed in any one loop, hence reducing the likelihood of cache conflicts. On the other hand, loop distribution also reduces locality by increasing the number of iterations between uses of the same data.

Loop distribution is legal if and only if there are no dependence cycles in the original loop with at least one loop-carried dependence. For example, the loop in Figure 2.10 could not be distributed in the manner shown if $S_1(i)\delta S_2(i)$, $S_2(i)\delta S_4(i)$, and $S_4(i)\delta S_1(i)$. Statements involved in a cycle must appear in the same loop.

2.4.6 Loop Fusion

Loop fusion is the opposite of loop distribution; it combines the bodies of adjacent loops, as shown in Figure 2.10. The loops to be fused must have compatible loop headers. Renaming of index variables and peeling of boundary iterations may be used to make headers compatible. Alternatively, the fused loop bounds may be set to the minimum lower bound and maximum upper bound from the original loops, and conditional guards may be used to prevent statements from being executed in iterations not included in their original loops.

Loop fusion enhances both locality and parallelism. Locality is enhanced after fusion by reducing the number of intervening iterations between uses of the same data. However, fusion may also reduce locality because increasing the amount of data accessed in each fused loop iteration increases the potential for cache conflicts. If the loops being fused are parallel, then fusion increases the granularity of parallelism when the resulting fused loop is also parallel.

However, fusion may also reduce the degree of parallelism by resulting in a serial loop.

The legality of fusion is dictated by dependences between iterations in the loops being fused. If a dependence flows from statement S_1 in one loop to statement S_2 in another loop, but after fusion the dependence becomes $S_2(i)\delta S_1(i)$, then fusion is not legal because the sense of the dependence has been reversed from the original semantics. If fusion of a sequence of parallel loops is legal, the resulting fused loop may not be parallel if dependences originally between iterations in different loops become loop-carried in the fused loop.

2.5 Data Transformations

In addition to loop transformations, there are a number of data transformations, primarily for array data, that may also enhance locality within loops. The legality of all of the data transformations described in this section is contingent upon the ability to identify all array references and alter them where necessary to match the data transformation. Features such as pointers and aliasing may make it impossible to guarantee that all references are modified appropriately. However, in numeric programs that operate on arrays, data transformations are often feasible [AAL95, BGS94, LW94].

2.5.1 Memory Alignment

Memory alignment [BGS94] is a general data transformation that seeks to enhance spatial locality within cache lines by aligning data to cache line boundaries in memory. For example, if a cache-line-sized portion of data structure is referenced in a program, aligning the data structure such that the data to be accessed begins at a cache line boundary, rather than straddling two cache lines, reduces the number of cache lines that are referenced. Memory alignment can be useful in reducing false sharing in parallel execution. However, the benefit of memory alignment diminishes when the data size is much larger than a single cache line, and there is no benefit if data is accessed with a stride that exceeds the cache line size.

2.5.2 Array Padding

Array padding [BGS94] increases the size of inner array dimensions to reduce cache conflicts between elements from the same array. Since caches sizes are powers of two, conflicts may

occur frequently when array dimension sizes are also powers of two. Padding introduces unused array elements that serve only to alter the memory layout of the array. Since the mapping of data from memory into the cache depends on the memory layout, array padding may enhance locality by altering the mapping sufficiently to reduce the occurrence of cache conflicts.

2.5.3 Array Element Reordering

Array element reordering [AAL95] modifies the storage order for elements within the same array without consuming additional storage. Modifying the storage order can enhance spatial locality for cache lines. The simplest transformation for array element reordering is permutation of array dimensions. If the dimension that is traversed in the innermost loop of a loop nest is aligned with the storage order for cache lines, then spatial locality is maximized. A more complicated transformation is increasing the dimensionality of the array while holding the total number of elements constant. This transformation may be used to create smaller blocks of contiguous array elements to enhance spatial locality. For example, given an $n \times n$ array, a subblock of $b \times b$ elements (where $b < n$) is not contiguous in memory. However, restructing the array into a three-dimensional $k \times b \times b$ array (where $k = n^2/b^2$) results in k two-dimensional contiguous subblocks of $b \times b$, and the amount of storage needed remains the same. The drawback of this approach is that all array references and their subscript expressions must be modified to reflect the element reordering.

2.5.4 Array Expansion and Contraction

Array expansion was discussed earlier in Section 2.3.2 in the context of eliminating loop-carried dependences to enable loop parallelization. Because array expansion increases the amount of data accessed in a loop, it is not likely to enhance locality, and may instead diminish locality.

On the other hand, the opposite transformation, array contraction [War84], reduces the array dimensionality and eliminates the storage needed by the dimensions being eliminated, and hence reduces the amount of data accessed in a loop. Array contraction is applicable when a value written to an array element in one loop iteration is not used in other iterations, and also not used after exiting the loop. In such circumstances, the array may be contracted to eliminate the dimension containing the unused data. In the best case, the array is contracted

into a single scalar variable to substantially reduce the amount of data accessed in the loop. However, contraction to a single scalar variable may then introduce loop-carried dependences that prevent parallelization. Hence, there is a tradeoff between parallelism and locality.

2.5.5 Array Merging

Array merging [LW94] interleaves data from two or more arrays used in the same loop in order to enhance spatial locality for cache lines. For example, the conventional memory layout for two arrays x and y consists of all elements of x, followed by all elements of y. With array merging, the new layout consists of the first element of x, followed by the first element of y, then second element of x, then the second element of y, and so on. When executing a loop, this layout causes corresponding elements from both arrays to be loaded in the same cache line with one memory access. With the conventional layout, two separate cache lines are loaded. Although array merging may avoid back-to-back memory accesses for cache lines, the total number of cache lines accessed is the same with either layout.

2.6 Effectiveness of Locality Enhancement within Loop Nests

This section surveys a representative set of studies that provide insights into the effectiveness of locality enhancement. There exists a large body of literature on locality-enhancing techniques such as unimodular transformations, tiling, loop distribution, and loop fusion [BGS94]. In past research, loop permutation and tiling within loops have been studied frequently and evaluated extensively [Ban93, CMT94, IT88, KM92, NJL94, WL91]. Other techniques such as loop distribution and loop fusion have received less attention [KM94, War84], and have been viewed as transformations that enable permutation [CMT94]. Hence, the survey will focus primarily on evaluating the effectiveness of loop permutation and tiling within loops.

2.6.1 Survey of Selected Studies

The studies selected for the survey in this section are the works by Porterfield [Por89]; Wolf [Wol92]; Carr, McKinley, and Tseng [CMT94]; and McKinley and Temam [MT96]. These studies consider programs from well-known benchmark suites such as SPEC [Sta] and Perfect Club [BCK$^+$89], as well as other representative numerical programs. The loop nests

in these programs exhibit two common characteristics. First, the majority of loop nests have rectangular iteration spaces, and on occasion, triangular iteration spaces [MT96, Wol92]. Rectangular iteration spaces reflect the bounds of rectangular arrays accessed in loop nests. Second, the majority of array references in loop nests have subscript expressions that induce regular data access patterns [Por89, MT96, Wol92]. This regularity in turn induces uniform data reuse and dependence relationships. The following paragraphs summarize the results and conclusions from each of these studies.

Porterfield [Por89] performs cache simulations to evaluate the effectiveness of loop permutation, tiling, and loop fusion, and proposes a model to guide the application of these loop transformations. The model makes use of dependence information (including input dependences) for array references in a loop body to determine the number of iterations before the data accessed in the loop exceeds the available cache capacity. Dependence distances and the level of loops carrying dependences are used to compute the amount of data resident in the cache. The same information is then used to identify individual array references that are likely to incur cache misses once the cache capacity is exceeded. The intent is to guide the application of appropriate transformations for reducing the number of misses for these array references.

Porterfield presents simulated cache hit ratios for a collection of 12 numerical programs. The simulations employ a 32-Kbyte cache with 4-way set-associativity. Prior to applying transformations for locality enhancement, Porterfield reports that the average hit ratio for the programs is 76% with one-word cache lines. After applying the transformations, the average hit ratio increases to 81%. Only 3 of the 12 programs could be transformed to show a significant improvement in hit ratio with one-word cache lines. Two of the programs contained matrix multiplication kernels whose cache hit ratios were improved substantially with tiling. The third program benefited from applying a sequence of loop permutation, distribution, and fusion transformations to a pair of loop nests that were executed frequently. Porterfield also reports that when the cache line size is increased to 8 or 16 words, the average hit ratio for the original programs increases to 95%. Thus, the average hit ratio for the original programs with long cache lines is better than the hit ratio for the transformed programs with one-word cache lines. No performance results are given for the transformed programs with long cache lines.

Wolf [Wol92] describes techniques that combine unimodular loop transformations with

tiling. He also proposes a model to guide the application of the loop transformations. The model estimates the expected reduction in the number of cache misses per iteration of the innermost loop. When tiling is applied to exploit temporal reuse, the model assumes that the number of cache misses is reduced by the number of uses of the same data. In other words, the underlying assumption is that in the absence of tiling, none of the reuse is converted into temporal locality. In conjunction with tiling, Wolf also describes an algorithm for tile-size selection to limit the occurrence of conflicts between reused elements from the same array in low-associativity caches.

Experimental results are presented for 8 application programs in which 171 loop nests were considered for transformation. Tiling was applied to 50% of the loop nests, permutation was applied to 20% of the loop nests, and the remaining loop nests were not transformed. Loop skewing was never applied. The performance results are speedups given by the ratio of original and enhanced execution times for each program on a uniprocessor. On a system with a 64-Kbyte direct-mapped cache, the speedup for one program was 15%, and the speedup for two others was 5%. The five remaining programs either showed no improvement or performed worse. Results are also provided for 7 kernels. Out of 11 loop nests in these kernels, 8 were tiled and 2 were permuted; none were skewed. Tiling resulted in a speedup of 200% for a kernel containing a loop nest for LU decomposition. The speedup for a kernel loop containing matrix multiplication improved by 15%. The remaining kernels showed little or no improvement.

Carr, McKinley, and Tseng [CMT94] study the effectiveness of loop permutation to enhance spatial locality for cache lines. They also consider the use of loop distribution and loop fusion as supplementary transformations to enable loop permutation. A cost model is used to estimate the number of cache lines accessed when a given loop is positioned innermost in a loop nest. This cost model determines a permutation that positions loops from outermost level to innermost level in decreasing order of the number of cache lines accessed.

Experimental results are reported for a collection of 35 application programs to ascertain whether loop permutation driven by the cost model described above provides significant performance improvements. The performance results are speedups that represent reductions in execution time for each program on a uniprocessor. Results obtained on a system with a 64-Kbyte, 4-way set-associative cache indicate that the speedup for one program was 115% as

a result of permuting the loops in the two most frequently executed loop nests. The speedup with loop permutation for a kernel containing a loop nest for Gaussian elimination was 768% because the original loop nest did not conform to the array element order enforced by the source language. Seven other programs showed speedups ranging from 1% to 13%. The remaining 27 of 35 programs experienced no benefit or degradation in performance. Their analysis of the 1400 loop nests considered in the 35 programs indicates that 74% of the loop nests are already coded with the best loop in the innermost position for spatial locality, hence loop permutation is not needed in the majority of loop nests. Only 11% of the loop nests were permuted, while the remaining 15% could not be permuted.

McKinley and Temam [MT96] perform cache simulations for 8 application programs to classify and measure spatial and temporal locality. They simulate a modest 8-Kbyte, direct-mapped cache with 32-byte cache lines. These results are obtained only for the original programs without applying any locality enhancement techniques. Nonetheless, they do provide a number of insights that are relevant for locality enhancement.

First, they report that overall cache hit ratio for all programs is high; no program had a hit ratio below 90%. Second, the results indicate that the majority of the cache misses are incurred for data reused between loop nests, i.e., data accessed in one loop nest does not remain cached for reuse in a subsequent loop nest. They indicate that the cache hit ratio for data reused within loop nests is high, and that both spatial and temporal locality have equal significance within loop nests.[4] Finally, they conclude that the relatively small number of cache misses for data reused within the same loop nest is due primarily to cache conflicts between different array references, rather than due to insufficient cache capacity. This behavior persists even for 2-way set-associative caches.

2.6.2 Conclusions and Implications

A number of conclusions can be drawn from the survey of previous studies. These conclusions are enumerated and explained in detail below.

[4] It should be noted that McKinley and Temam *disabled* loop unrolling when compiling programs, which potentially increases the number of accesses to the cache across loop iterations. This increase may potentially overstate the extent of temporal locality in the cache. In contrast, an unrolled loop provides opportunities to reuse data from registers within the unrolled loop body and thereby reduce the number of accesses to the cache.

1. *Loop permutation and tiling provide limited performance improvements for the majority of loop nests in representative applications.* Carr *et al.* demonstrate that loop permutation is frequently unnecessary because the majority of representative loop nests are already coded with the best permutation. The results of Porterfield suggest that long cache lines provide adequate locality without requiring additional transformations. Wolf and Porterfield demonstrate that tiling provides significant improvements only for distinguished kernels such as matrix multiplication and LU decomposition that are characterized by significant temporal reuse. For the majority of loop nests in more representative application programs, tiling does not provide any significant benefit. Finally, McKinley and Temam conclude that locality from data reuse within representative loop nests is high. They also report that the failure to capture reuse *between* loop nests causes the majority of cache misses; this reuse cannot be converted into locality by permutation or tiling.

2. *Techniques for avoiding cache conflicts when applying locality-enhancing transformations have not received adequate attention.* Carr *et al.* and Porterfield conducted their experiments on 4-way set-associative caches that decrease the likelihood of conflicts, hence they do not discuss techniques for conflict avoidance and rely instead on the cache associativity. McKinley and Temam conclude that relatively few misses are incurred for reuse within loop nests, and conflicts cause the majority of these misses, even for a 2-way set-associative cache. However, they do not propose a conflict avoidance technique because their study does not evaluate transformations for locality enhancement. Only Wolf discusses a technique for conflict avoidance in conjunction with a locality-enhancing loop transformation. Although he proposes a tile-size selection algorithm to prevent conflicts within the same array in a tiled loop nest such as matrix multiplication, the benefit of tile-size selection is not demonstrated for loop nests in more representative applications.

3. *Existing models for guiding loop transformations do not adequately reflect the potential benefit of locality enhancement on execution time.* Failure to properly gauge the impact of a particular transformation on execution time is evident in the lack of performance improvement and leaves the utility of the transformation open to question. The surveyed models establish criteria for applying individual transformations, but these criteria do not

necessarily reflect the true locality benefit. The model of Porterfield seeks to identify array references that incur cache misses within a loop nest, but such cache misses are caused largely by failing to capture data reuse between loop nests. The model of Wolf assumes that there is no temporal locality within a loop nest prior to tiling, but since reuse within loop nests is frequently converted to locality, this assumption can overstate the benefit of tiling. Finally, the model of Carr *et al.* provides a measure for ranking loop permutations, but representative loop nests do not normally require permutation.

The implications of the survey presented in this section are that reuse *across* loops must be exploited, that cache conflicts must be eliminated to ensure the benefit of locality, and that more effective models are required to reflect the impact of locality enhancement on execution time. The remainder of this dissertation addresses each of these implications. The issue of enhancing locality across loop nests is addressed in Chapters 4 and 5, while the issue of eliminating cache conflicts to ensure the benefit of locality is addressed in Chapter 6. In the interim, Chapter 3 describes a new model for quantifying the impact of locality on execution time. The model is used in the subsequent chapters to assess the potential benefit of locality enhancement.

Chapter 3

Quantifying the Benefit of Locality Enhancement

The benefit of locality enhancement must be assessed with reasonable accuracy in order to effectively guide the application of appropriate transformations and also to verify that the actual performance gains meet expectations. This chapter proposes a model to assess the potential reduction in execution time from enhancing cache locality across nested loops.

This chapter is organized as follows. First, an overview of the proposed model is outlined along with underlying assumptions. Next, the benefit of locality enhancement is expressed as a ratio of the number of memory accesses before and after applying a transformation. This ratio is then used to model the impact of locality enhancement on execution time. Finally, potential limitations of the model are briefly discussed.

3.1 Overview of Model and Underlying Assumptions

The purpose of the model proposed in this chapter is twofold. First, the model enables a compiler to better assess the extent to which locality enhancement will reduce execution time. Second, the model provides a useful estimate for the expected reduction in execution time for comparison with the measured reduction in execution time. Chapters 4 and 5 use the model to assess the benefit of locality-enhancing transformations (namely fusion and tiling), and Chapter 7 uses the model to compare expected and measured reductions in execution times.

The model accounts for two factors that together determine the potential benefit of locality enhancement: (a) the reduction in the number of memory accesses from locality enhancement, and (b) the contribution of memory accesses towards the total execution time prior to locality

enhancement. The model seeks to quantify each of these factors such that the extent of the reduction in execution time can be assessed.

The underlying assumptions for the model are enumerated below along with justifications.

1. Data reuse *within* representative loop nests is assumed to be converted into locality by the cache without the aid of any transformation. The justification for this assumption is the lack of significant performance improvement from existing transformations and evidence for high locality from reuse within loop nests, as described in Section 2.6.

2. Data reuse *between* loop nests is assumed *not* to be converted into locality. The validity of this assumption depends on the total data size for a specific program and the cache size of the system on which the program is executed. The expectation is that problems of greatest interest to application programmers will have sufficiently large data sizes to require locality enhancement across loop nests. Furthermore, the evidence cited in Section 2.6 indicated that reuse between loop nests is often unexploited.

3. Loop nests are assumed to have rectangular iteration spaces to reflect the bounds of similarly-sized arrays accessed in loop bodies. As a result, it is assumed that loop nests read or write all elements of the accessed arrays (or nearly all elements, if boundary regions are excluded). The justification for this assumption follows from the common characteristics of loop nests in numeric programs, as described in Section 2.6.

4. It is assumed that cache conflicts do not diminish locality after applying transformations for locality enhancement. The cache conflict avoidance techniques to be presented in Chapter 6 will allow transformations to fully realize their benefits and ensure the validity of this assumption.

5. The cache policy for writes is assumed to be *write-allocate* and *write-back* [PH96]. Cache lines must first be loaded, or allocated, in the cache in order for writes to proceed. Furthermore, modified data is written back to memory only on replacement in the cache. This policy performs well in multiprocessor systems [PH96] and is standard for caches in contemporary high-speed microprocessors [CHK+96, MWV92, Yea96].

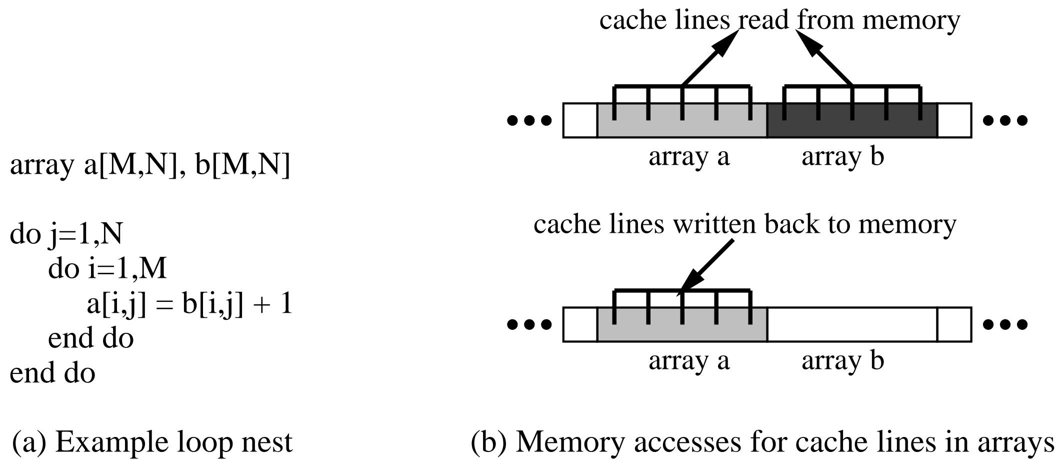


Figure 3.1: Illustration of memory accesses for arrays in loop nests

3.2 Quantifying Memory Accesses for Arrays

To assess the benefit of locality enhancement, the model discussed in this chapter relies on quantifying memory accesses for arrays in loop nests. Since processors access memory in units of cache lines, the number of memory accesses per array is a function of the array size and cache line size. A k-dimensional array with dimensions $N_1 \times N_2 \times \cdots \times N_k$ normally consists of contiguously-allocated elements in memory. For a cache line size of s_{line} elements, the number of cache lines in the array is given by $\lceil (N_1 \cdot N_2 \cdots N_k)/s_{line} \rceil$.

A loop nest referencing an array often has regular data access patterns and iteration space bounds that reflect the array bounds. Reuse of array elements arising from the data access patterns within the loop nest is normally captured by the cache (see Section 2.6). As a result, each cache line in the array is ideally accessed only once from memory to load the line into the cache. If the loop nest modifies a cache line (i.e., writes to array elements in the cache line), the line must subsequently be written back to memory. The total number of memory accesses per array is therefore given by the number of cache lines read from and written to memory for the array.

An example for illustrating memory accesses for arrays in loop nests is shown in Figure 3.1. The loop nest in Figure 3.1(a) references two arrays whose dimensionality and bounds match

the dimensionality and bounds of the loop nest. Arrays a and b are read in the body of the loop nest, hence cache lines for these arrays are loaded into the cache as they are needed. Since elements of array a are modified in the body of the loop nest, the affected cache lines for array a are eventually written back to memory as they are replaced by new data later in the execution of the loop nest. The transfer of cache lines to and from memory is shown in Figure 3.1(b).

When the arrays accessed in a collection of loop nests are similar in size and the iteration space bounds reflect the array bounds (as in Figure 3.1), memory accesses may be quantified in a manner that is independent of array size and cache line size. Throughout this dissertation, reading or writing the cache lines for a single array during the execution of a loop nest is designated a *sweep* through the region of memory allocated for that array. When arrays are similarly-sized, sweeps for different arrays represent an equivalent number of memory accesses. For example, in Figure 3.1(b), loading the cache lines for arrays a and b from memory results in 2 sweeps (each accessing a total of $M \cdot N$ array elements), and writing back the cache lines for array a to memory results in 1 additional sweep, for a total of 3 equivalent sweeps.

3.3 Quantifying the Reduction in Memory Accesses with Locality Enhancement

The goal of locality enhancement across a loop nest sequence is to reduce the number of memory accesses for cache lines by retaining data in the cache between uses. The reduction in the number of memory accesses for cache lines is expressed as the ratio

$$r_m = \frac{\text{\#memory accesses before locality enhancement}}{\text{\#memory accesses after locality enhancement}}. \quad (3.1)$$

This ratio indicates the potential benefit of enhancing locality across loops; the larger the ratio, the greater the potential reduction in execution time.

When the loop nest sequence under consideration contains references to similarly-sized arrays, as described in Section 3.2, the ratio r_m may be expressed using the number of sweeps before and after locality enhancement. This is because the total number of memory accesses for cache lines is directly proportional to the number of sweeps. Before locality enhancement, each array that is referenced in a loop nest contributes one sweep for the numerator of the ratio for r_m. Each array that is modified in a loop nest contributes an additional sweep for

writebacks. Locality enhancement to exploit array reuse across loop nests reduces the number of memory accesses, and hence the number of sweeps. The expected number of sweeps after locality enhancement is indicated in the denominator of the ratio for r_m. When expressed in terms of sweeps, r_m is designated the *sweep ratio* for convenience. A compiler can compute this ratio to assess the potential benefit of enhancing locality for a loop nest sequence (Chapter 4 and Chapter 5 provide the details on computing r_m for different transformations).

3.4 Quantifying the Impact of Locality Enhancement on Execution Time

A reduction in the number of memory accesses with locality enhancement, as embodied by the ratio r_m in Equation 3.1, can reduce execution time. The potential reduction in execution time depends on the relative contribution of memory accesses towards total execution time. In the simplest case where the processor stalls on each memory access, the total execution time T for a sequence of loop nests before locality enhancement is represented as

$$T = T_c + T_m,$$

where T_c is the total computation time, and T_m is the time during which computation is stalled to access memory for cache lines. This formulation does not consider concurrency between computation and memory accesses; this issue is addressed at the end of this section.

The contribution of memory accesses towards execution time is reflected in the fraction

$$f_m = \frac{T_m}{T_m + T_c}.$$

Figure 3.2 illustrates the relationship between T_c, T_m, and f_m for a hypothetical sequence of computation and memory accesses for cache misses. For illustrative purposes, the computation and memory accesses shown in Figure 3.2(a) are lumped together in Figure 3.2(b) without changing T_c or T_m.

A locality-enhancing transformation reduces the number of memory accesses by a factor r_m, which should also reduce T_m by a factor of r_m without affecting T_c. For example, Figure 3.2(c) shows the effect of reducing T_m by $r_m = 2$. Since only a fraction f_m of the total execution time

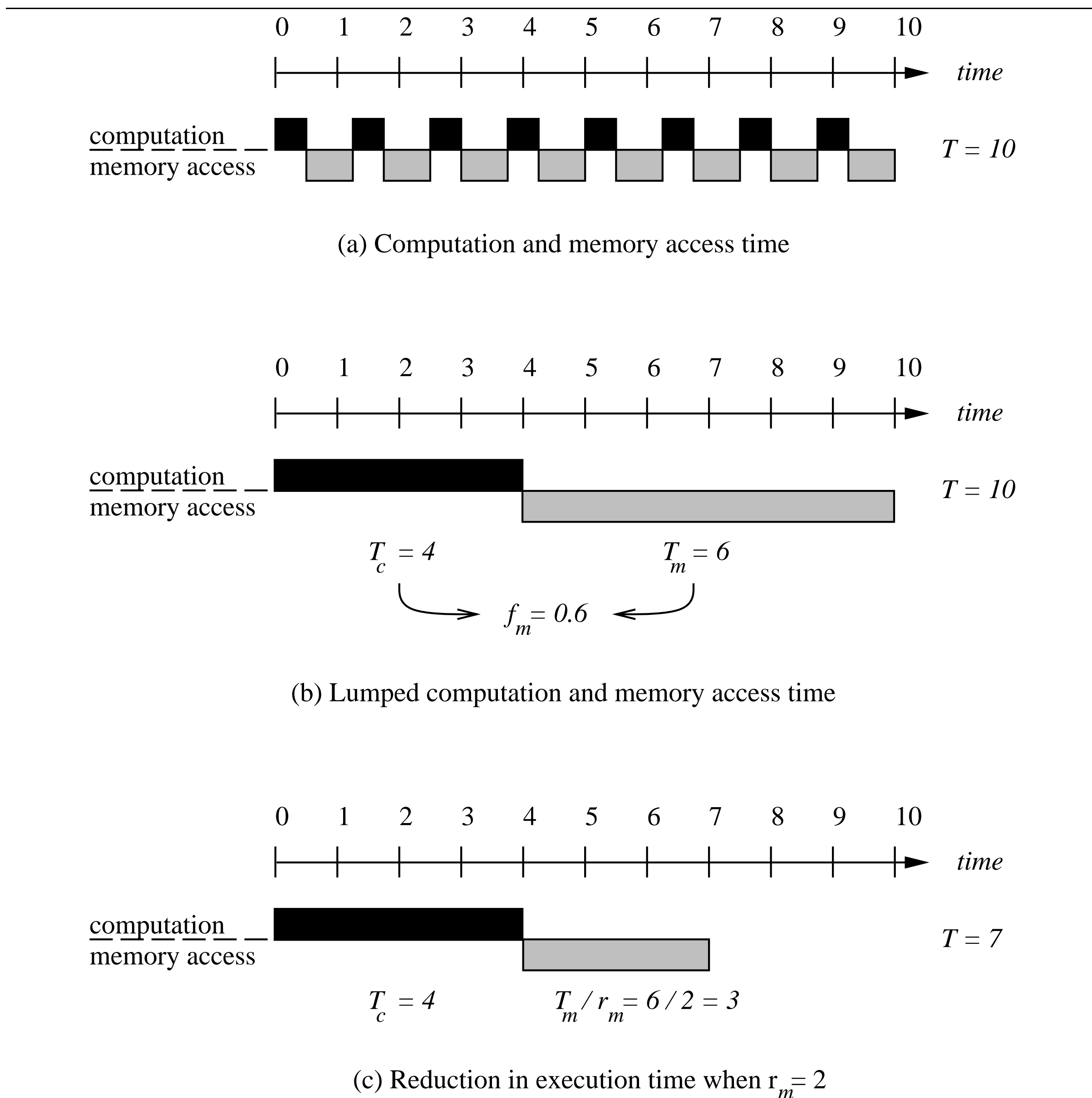


Figure 3.2: Graphical representation of $T = T_c + T_m$ and effect of locality enhancement

T is reduced by r_m, the improvement in performance due to locality enhancement is given by

$$\frac{T_c + T_m}{T_c + T_m/r_m} = \frac{1}{(1 - f_m) + f_m/r_m}. \tag{3.2}$$

This improvement indicates a reduction in execution time on one processor, but also applies for parallel execution with a balanced workload; in this case, all processors see the same reduction in execution time.

Although Equation 3.2 assumes for simplicity that a processor stalls on memory accesses,

modern processors are now designed with support for prefetching [CHK+96, Yea96]. Prefetching *hides* memory latency by initiating multiple memory accesses in advance of data usage to overlap memory accesses with computation [MLG92]. The performance improvement from prefetching depends on the extent of this overlap and the memory system bandwidth available for concurrent memory accesses.

Locality enhancement can increase the performance improvement with prefetching by reducing the number of memory accesses and hence making more bandwidth available to overlap the remaining memory accesses [MLG92, BAM+96]. If the time for concurrent memory accesses with prefetching still exceeds the time for computation, execution time is governed by memory access time. In this case, combining prefetching with locality enhancement to reduce the memory accesses by a factor of r_m should ideally reduce execution time by r_m over prefetching alone, provided that the remaining memory accesses still determine execution time (equivalent to $f_m = 1$ in Equation 3.2).

However, the actual improvement may be less than r_m for a number of reasons. First, locality enhancement may reduce the number of memory accesses to the point that the computation time, rather than the reduced memory time, dominates total execution time. Hence, overlapping the remaining memory accesses with computation will not result in commensurate reductions in execution time. Second, with software-controlled prefetching, instruction overhead may also reduce the improvement [MLG92]. Finally, prefetch requests may not be scheduled early enough in some cases to hide all memory latency [BAM+96, SMP+96]. In general, the ratio r_m provides a useful bound for the improvement of locality enhancement with prefetching.

3.5 Potential Limitations of the Model

Capturing reuse within loop nests The model assumes that reuse of data within loop nests is captured by the cache. This assumption may not be valid for loop nests that access a large volume of data and have considerable temporal reuse separated by a large number of loop iterations. Such loop nests may benefit from being tiled individually; one example is matrix multiplication, as discussed in Section 2.6. However, the model presented in this chapter targets more representative loop nest *sequences*, rather than isolated loop nests, and the intent of the model is to assess the benefit of enhancing locality *across* these loop nests, rather than tiling

```
array a[M,N], b[N]

do j=1,N
    do i=1,M
        a[i,j] = b[j] + 1
    end do
end do
```

(a) Reuse carried by inner loop

```
array a[M,N], c[M]

do j=1,N
    do i=1,M
        a[i,j] = c[i] + 1
    end do
end do
```

(b) Reuse carried by outer loop

Figure 3.3: Examples of loop nests accessing arrays with differing dimensionalities

them individually. As discussed in Section 2.6, the majority of loop nests in representative applications do not benefit from tiling because most unexploited reuse occurs across loop nests.

Memory sweeps for differing array sizes The determination of memory sweeps is based on the assumption of rectangular loop bounds that reflect the bounds of similarly-sized arrays accessed in the loop body. However, loop nests may access arrays of different size, most often when array dimensionalities differ. Figure 3.3 provides examples of such loop nests.

Differences in array sizes do not present a serious limitation for two reasons. First, the significance of memory accesses for lower-dimensionality arrays diminishes rapidly with increasing array sizes. For example, consider the loop nest shown in Figure 3.3(a). The elements of array b are reused within the inner loop, hence each element may be register-allocated and the N elements in array b should ideally be loaded once. At the same time, a total of $M \cdot N$ elements in array a are both read and written. Two memory sweeps are required for array a, and for large M, the memory accesses for array b become insignificant.

The second reason is that even when reuse of a lower-dimensional array is carried by an outer loop, as in Figure 3.3(b), the available cache capacity may permit reused data to remain cached between uses. For example, the data from array c occupies a fixed region of the cache, while the data from array a sweeps through the cache as the loop is executed. Although array a will occasionally displace elements of array c from the cache, the elements of c will often be reused from the cache.

3.6 Chapter Summary

The model proposed in this chapter provides a means of assessing the potential benefit of locality enhancement by quantifying the reduction in the number of memory accesses. The model can also estimate the expected reduction in execution time by quantifying the contribution of memory accesses towards total execution time. The estimates provided by the model can then be compared against experimental measurements. Chapters 4 and 5 of this dissertation use the model to assess the benefit of locality enhancement, while Chapter 7 compares experimental results with estimates obtained with the model to demonstrate that the benefits of locality enhancement are realized.

Chapter 4

The Shift-and-peel Transformation for Loop Fusion

This chapter proposes a technique called the shift-and-peel transformation to fuse multiple parallel loops in order to enhance cache locality. With existing techniques, fusion is limited by dependences that either render fusion illegal or force the fused loop to be executed serially. The shift-and-peel transformation overcomes these limitations in order to fully exploit reuse across loops without sacrificing parallelism.

This chapter is organized as follows. First, motivation for the shift-and-peel transformation is provided. The shift-and-peel transformation is then described in detail, including algorithms for the required analysis and methods for implementing the transformation.

4.1 Loop Fusion

This section provides motivation for the shift-and-peel transformation by first describing and quantifying the benefits of fusion, then explaining how data dependences limit the use of fusion. Related work is then outlined to highlight shortcomings of existing fusion techniques.

4.1.1 Granularity of Parallelism and Frequency of Synchronization

Loop fusion combines the bodies of parallel loops into a single loop body. When the resulting loop is also parallel, then the granularity of parallelism is larger than the granularity in each of the original loops prior to fusion. A large granularity of parallelism reduces the overhead of parallelization, particularly for large-scale multiprocessors.

Furthermore, barrier synchronization is normally required between parallel loops to ensure

that data dependences between loops are respected. In a large-scale multiprocessor, frequent global synchronization with barriers after every parallel loop reduces parallel efficiency whenever one slow processor forces all others to wait. By combining loop bodies into a single loop, fusion reduces the number of barriers to only one. Hence, the frequency of synchronization is reduced, and parallel efficiency is increased.

4.1.2 Quantifying the Benefit of Enhancing Locality with Fusion

Loop fusion enhances locality by combining loop bodies to reduce the number of iterations between uses of the same data. In this section, the model proposed in Chapter 3 is used to quantify the locality benefit of fusion. Let $\mathcal{L}$ denote a sequence of loop nests that reference similarly-sized arrays. Hence, memory accesses may be quantified conveniently as sweeps, as discussed in Section 3.2. Prior to fusion, a memory sweep is required for each array referenced in each loop nest. Let $A(\ell)$ denote the set of arrays referenced (read or written) in each loop nest $\ell \in \mathcal{L}$. For the original sequence of loop nests, the total number of memory sweeps to load data into the cache before applying fusion is

$$s_b^r = \sum_{\ell \in \mathcal{L}} |A(\ell)|.$$

Modified data in the cache must be written back to memory as it is replaced by incoming data in each loop nest. Thus, there is a writeback sweep each time an array is modified in a loop nest. Let $A_w(\ell)$ denote the set of arrays that are modified in each loop nest $\ell \in \mathcal{L}$ ($A_w(\ell) \subseteq A(\ell)$). The number of writeback sweeps for the original loop nest sequence is given by the number of times arrays are modified:

$$s_b^w = \sum_{\ell \in \mathcal{L}} |A_w(\ell)|.$$

When the loop nests in $\mathcal{L}$ are fused, only one read sweep should be incurred for each array. Hence, the total number of sweeps to load data into the cache after applying fusion is

$$s_a^r = \left| \bigcup_{\ell \in \mathcal{L}} A(\ell) \right|.$$

Clearly, $s_a^r \leq s_b^r$, since each array is referenced in at least one loop nest prior to fusion. Indeed, s_a^r is the minimum number of read sweeps that can be achieved with fusion: one per array.

Writebacks still occur after fusion. However, an array that is modified in two or more of the original loop nests before fusion generates only one writeback sweep after fusion because the data remains cached between writes. Consequently, the number of writeback sweeps after fusion is given by the number of arrays modified in *any* of the original loop nests,

$$s_a^w = \left| \bigcup_{\ell \in \mathcal{L}} A_w(\ell) \right| .$$

Clearly, $s_a^w \leq s_b^w$, since each modified array incurs at least one writeback sweep prior to fusion. Indeed, s_a^w is the minimum number of writeback sweeps: one per modified array.

The sweep ratio for loop fusion is given by

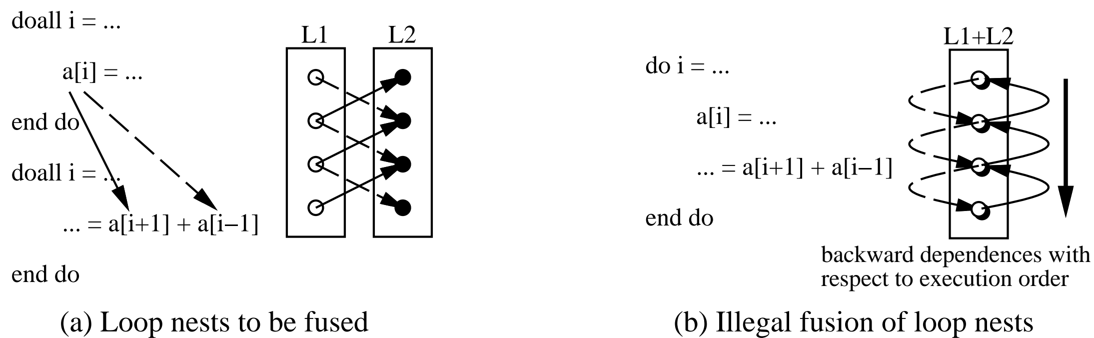
$$r_{fusion} = \frac{s_b^r + s_b^w}{s_a^r + s_a^w} = \frac{\sum_{\ell \in \mathcal{L}} |A(\ell)| + \sum_{\ell \in \mathcal{L}} |A_w(\ell)|}{\left| \bigcup_{\ell \in \mathcal{L}} A(\ell) \right| + \left| \bigcup_{\ell \in \mathcal{L}} A_w(\ell) \right|} . \tag{4.1}$$

Since $s_a^r \leq s_b^r$ and $s_a^w \leq s_b^w$, it is clear that $r_{fusion} \geq 1$. A compiler may assess the profitability of fusion by computing this sweep ratio. If it is close to one, then the locality benefit is not significant, and fusion may not be profitable. However, as the sweep ratio increases, the benefit from fusion increases because fewer cache misses and writebacks are incurred.

4.1.3 Dependence Limitations on the Applicability of Loop Fusion

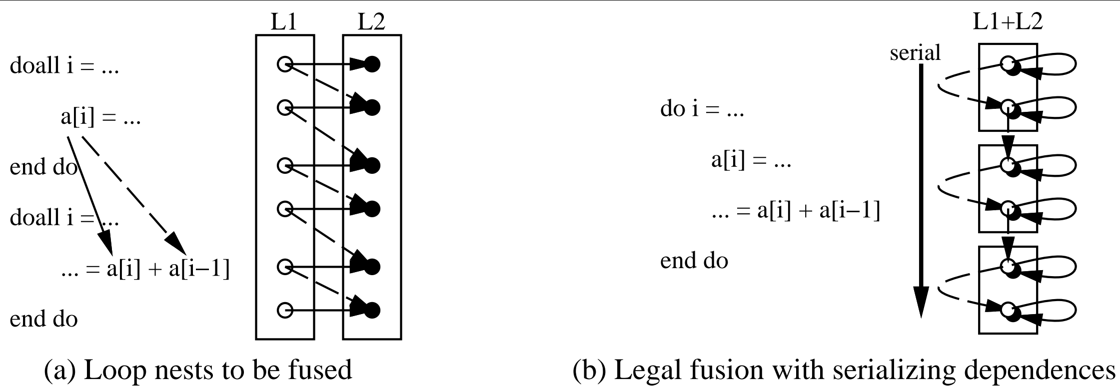
Despite the benefits of loop fusion, it is not always applicable. Reuse across loops often implies the existence of data dependences between iterations in different loops. After applying loop fusion, these dependences now flow within a single loop. Dependences that flow between statement instances in the same loop iteration are loop-independent. However, those dependences that flow between statement instances in *different* loop iterations are loop-carried.

Fusion is legal if and only if none of the loop-carried dependences flow backwards with respect to the iteration execution order [Wol89]. In other words, the corresponding dependence distance or direction vectors must not be lexicographically negative. For example, both loops in Figure 4.1(a) reference the array a. This reuse implies dependences between the iteration spaces, as shown graphically in Figure 4.1(a), where individual iterations are represented by circles, and dependences are represented by arrows. Fusion combines the iteration spaces as shown in Figure 4.1(b), where the overlapping circles indicate that computation originally in separate



(a) Loop nests to be fused (b) Illegal fusion of loop nests

Figure 4.1: Example to illustrate fusion-preventing dependences



(a) Loop nests to be fused (b) Legal fusion with serializing dependences

Figure 4.2: Example to illustrate serializing dependences

loop iterations is now performed in a single loop iteration. Dependences now flow within the same loop and are loop-carried. Half of the dependences are lexicographically positive, hence they are not violated by fusion. However, the remaining loop-carried dependences are lexicographically negative, indicating that the sink iteration of each dependence would be executed before the source iteration. Consequently, fusion is *not* legal because it has violated the original program semantics. Dependences that become loop-carried and lexicographically negative after fusion are referred to as *fusion-preventing* dependences.

Even when there are no fusion-preventing dependences, lexicographically-positive loop-

carried dependences in the fused loop prevent parallel execution. This is illustrated in Figure 4.2. The two loops in Figure 4.2(a) individually have no loop-carried dependences; the iterations within each loop may be executed in parallel. Only a barrier synchronization is required between the loops to ensure that all iterations of the first loop have been executed before any iterations of the second loop are executed. However, fusion of the two loops results in loop-carried dependences, as shown in Figure 4.2(b). Explicit synchronization is required between dependent iterations executed by different processors. When blocks of iterations from the fused loop nest are assigned to different processors, the required synchronization effectively serializes the execution of the blocks of iterations. Consequently, lexicographically-positive loop-carried dependences are referred to as *serializing* dependences.

Thus, fusion to exploit reuse and enhance locality is not applicable in the presence of fusion-preventing dependences that arise from reuse. Furthermore, serializing dependences also limit the applicability of fusion for multiprocessors. Hence, the goal of this chapter is to overcome these dependences and enable fusion and subsequent parallelization.

4.1.4 Related Work

Existing techniques do not adequately address the dependence limitations discussed above. The following paragraphs present a survey of related techniques to highlight their shortcomings.

Warren [War84] discusses the use of fusion to enhance locality in vector registers, and to permit contraction of temporary arrays into scalars. However, fusion is not permitted in the presence of loop-carried dependences or incompatible loop bounds.

Kennedy and McKinley [KM94] use loop fusion and distribution to enhance locality and maximize parallelism. They focus on enhancing register locality with fusion, and describe a fusion algorithm that prevents fusion of parallel loops with serial loops. However, they disallow fusion when loop-carried dependences result or when loop bounds are incompatible.

Porterfield [Por89] suggests a "peel-and-jam" transformation in which iterations are peeled from the beginning or end of one loop nest to allow fusion with another loop nest. However, no systematic method is described for fusion of multiple loop nests, nor is parallelization of the fused loop nest considered.

Ganesh [Gan94] suggests an extension of Porterfield's peel-and-jam transformation to the

inner loops for a pair of multidimensional loop nests. However, dependences preventing parallelization are not addressed, nor is a systematic method described.

Callahan [Cal87] proposes loop alignment within a single loop to remove loop-carried dependences that prevent parallel execution. Code replication is advocated for resolving any conflicts in alignment requirements. However, replication to address alignment conflicts contributes significant execution overhead.

Appelbe and Smith [AS92] present a graph-based algorithm for deriving the required alignment, replication, and statement reordering to permit parallelization of an individual loop nest with loop-carried dependences. This work extends the techniques of Callahan, but still incurs significant overhead due to replication.

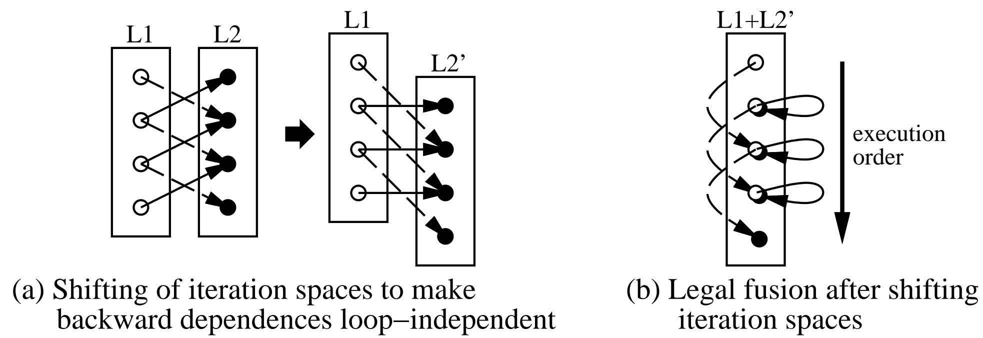
Pugh [Pug91] derives affine schedules for individual statements within a loop nest to guide transformations for parallelization. It is claimed that this method produces a compound transformation that is equivalent to applying any sequence of elementary transformations to the component loops within a loop nest, including fusion of inner loops. The intent of this method is to optimize for parallelism, hence fusion is not allowed if it generates loop-carried dependences. As a result, this technique may fail to enhance locality.

4.2 The Shift-and-peel Transformation

This section provides the details of the shift-and-peel transformation. The basic idea of the technique is described first, followed by a description of the procedure for deriving and applying the transformation on sequences of parallel loop nests. The legality of the transformation is also discussed, with a formal proof provided to substantiate the discussion.

4.2.1 Shifting to Enable Legal Fusion

Shifting enables legal fusion despite the uniform backward loop-carried dependences discussed in Section 4.1.3. This technique ensures that backward dependences become loop-independent in the fused loop by shifting the iteration space containing the sink iterations with respect to the iteration space containing the source iterations. Shifting is similar to alignment of dependences within a loop [Cal87], but is applied to different iteration spaces. The amount by which to shift is determined by the dependence distance. Shifting is illustrated in Figure 4.3, using the iteration



(a) Shifting of iteration spaces to make backward dependences loop–independent

(b) Legal fusion after shifting iteration spaces

Figure 4.3: Shifting iteration spaces to permit legal fusion

spaces shown earlier in Figure 4.1. The iteration space of the second loop in Figure 4.3(a) must be shifted by one iteration because of the backward dependence with a distance of one. The shift increases the distance of the forward dependences, but these dependences do not prevent fusion. After shifting, the loops may then be legally fused, as shown in Figure 4.3(b). The algorithm for deriving the required amount of shifting for arbitrary sequences of loop nests is discussed in Section 4.2.3.

4.2.2 Peeling to Enable Parallelization of Fused Loops

Peeling enables parallelization of a fused loop with uniform forward loop-carried dependences. This technique assumes static, blocked scheduling when parallelizing the fused loop. Static scheduling is not a serious limitation, as it is the most efficient approach when the computation is regular (see Section 2.3.4). This technique identifies iterations from the *original* loop nests that become sinks of cross-processor dependences[1] in the fused loop, then *peels* these iterations from their respective iteration spaces. After fusion, there are no longer any cross-processor dependences between blocks of iterations that are assigned to different processors. The peeled iterations are executed after all fused loop iterations have been executed. Since the dependences are uniform and block scheduling is used, the peeled iterations are located at block boundaries. The number of iterations that must be peeled is determined by the forward

[1]Cross-processor dependences are loop-carried dependences for which the source and sink iterations are executed by different processors.

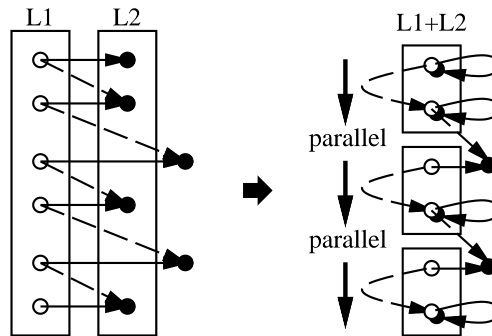


Figure 4.4: Peeling to retain parallelism when fusing parallel loops

dependence distance. This procedure is illustrated in Figure 4.4 using the iteration spaces shown previously in Figure 4.2. The forward dependences require peeling one iteration from the second loop at each block boundary. After fusion, the blocks of iterations are independent of each other and may be executed in parallel on different processors. Loop-carried dependences still exist, but are contained entirely within a block. Once the blocks of iterations have been executed in parallel, the peeled iterations may themselves be executed in parallel. The algorithm for deriving the required amounts of peeling for arbitrary sequences of parallel loop nests is given in Section 4.2.3.

4.2.3 Derivation of Shift-and-peel

In general, two or more loop nests may be considered for fusion, and fusion-preventing or serializing dependences may result from any pair of loop nests in the candidate set. Dependence relationships exist in the form of *dependence chains* passing through iterations in different loop nests. These dependence chains are dictated by the reuse of array elements in different loop iterations and constitute ordering constraints that must be satisfied for correctness. If shifting or peeling is applied to one loop nest, all subsequent loop nests along all dependence chains that pass through the affected loop nest must also be shifted or peeled in order to satisfy the ordering constraints for the affected iterations. That is, shifting and peeling must be *propagated along dependence chains.* It is therefore advantageous to treat candidate loop nests collectively

for fusion rather than incrementally one pair at a time.

This section presents algorithms to determine the amounts of shifting and peeling needed for each iteration space to enable legal fusion of a sequence of parallel loops, and subsequent parallelization of the fused loop. The algorithms assume uniform dependences between the loops being fused. Because the dependences are uniform, the dependence chains are also uniform. Consequently, all dependence chains may be represented with a single acyclic *dependence chain multigraph* $G(V, E)$. Each loop is represented by a vertex, and each dependence between a pair of loops is represented by a directed edge weighted by the corresponding dependence distance. Since fusion combines multiple loop bodies into a single loop body, all statements in a fused loop will share the same loop index variable. This fact can be exploited in order to obtain dependence distance information by assuming that the index variables of the different loops are the same [Wol89]. A forward dependence has a positive distance, and results in an edge with a positive weight. Conversely, a backward dependence has a negative distance, and results in an edge with a negative weight. A multigraph is required since there may be multiple dependences between the same two loops.

In deriving the required amounts of shifting, the dependences of interest are fusion-preventing dependences with negative distances. The multigraph $G(V, E)$ is reduced to a simpler *dependence chain graph* $G_s(V, E_s)$ by replacing multiple edges between two vertices by a single edge whose weight is the *minimum* from the original edges. A negative edge weight determines the amount of shifting required to remove backward dependences. This graph reduction preserves the acyclic structure of the original dependence chains. A traversal algorithm is then used to propagate shifts along dependence chains in $G_s(V, E_s)$. Each vertex is assigned a weight, which is initialized to zero, and the vertices are visited in topological order to accumulate shifts along chains. Note that the original loop order gives the topological order, hence there is no need to perform a topological sort. Only edges with a negative weight contribute shifts; all other edges are treated as having a weight of zero and serve only to propagate any accumulated shifting. At each vertex, the minimum value for all accumulated shifts through that vertex is always selected to ensure that all backward dependences are removed. The algorithm is given in Figure 4.5. Since each edge is traversed exactly once, the complexity of the algorithm is linear in the size of the graph, and upon termination, the final vertex weights

```
TRAVERSEDEPENDENCECHAINGRAPHFORSHIFTING(G_s)::
    foreach v ∈ V[G_s] do
        shift_weight(v) = 0
    endfor
    foreach v ∈ V[G_s] in topological order do
        foreach e = (v, v_c) ∈ E_s[G_s] do
            if weight(e) < 0 then
                shift_weight(v_c) = min(shift_weight(v_c), shift_weight(v) + weight(e))
            else
                shift_weight(v_c) = min(shift_weight(v_c), shift_weight(v))
            endif
        endfor
    endfor
```

Figure 4.5: Algorithm for propagating shifts along dependence chains

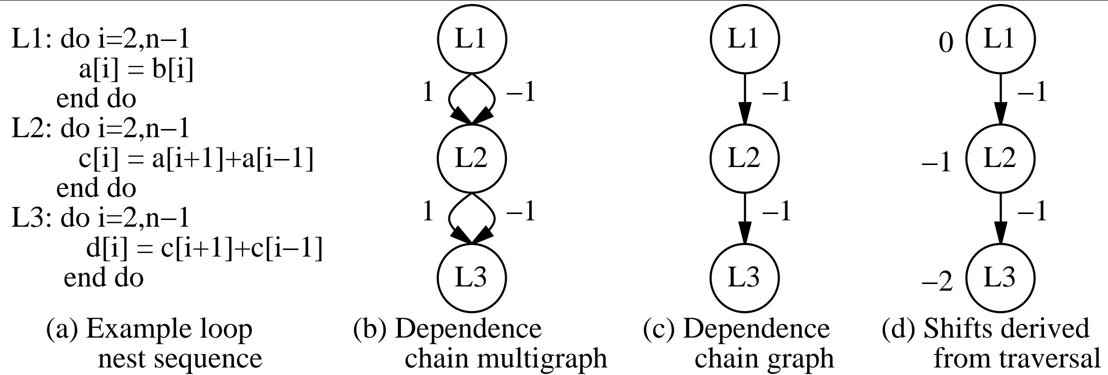


Figure 4.6: Representing dependences to derive shifts for fusion

indicate the amount by which to shift each loop *relative to the first loop* to enable legal fusion. Figure 4.6 illustrates the above procedure for deriving shifts.

In deriving the required amounts of peeling, the original dependence chain multigraph is reconsidered. This time, the edges of interest are serializing dependences with positive weights. The multigraph is reduced to a simpler dependence chain graph $G_p(V, E_p)$ by replacing multiple edges between two vertices with a single edge whose weight is the *maximum* from the original set of edges between these two vertices (as opposed to the minimum as in the case of shifting). When this maximum weight is positive, it indicates the amount of peeling needed to remove cross-processor dependences between the loops corresponding to the vertices for the edge. As

```
TRAVERSEDEPENDENCECHAINGRAPHFORPEELING(G_p)::
    foreach v ∈ V[G_p] do
        peel_weight(v) = 0
    endfor
    foreach v ∈ V[G_p] in topological order do
        foreach e = (v, v_c) ∈ E_p[G_p] do
            if weight(e) > 0 then
                peel_weight(v_c) = max(peel_weight(v_c), peel_weight(v) + weight(e))
            else
                peel_weight(v_c) = max(peel_weight(v_c), peel_weight(v))
            endif
        endfor
    endfor
```

Figure 4.7: Algorithm for propagating peeling along dependence chains

before, the reduced graph preserves the dependence chains from the original multigraph and remains acyclic. A similar graph traversal algorithm is used to propagate the required amounts of peeling along the dependence chains. The only modification is to consider edges with a positive weight, since only they require peeling to remove cross-processor dependences; all other edges are treated as having a weight of zero to propagate any accumulated amounts of peeling. At each vertex, the maximum value for all accumulated peeling through that vertex is selected to ensure that all cross-processor dependences will be removed. Upon termination, the final vertex weights are the number of iterations to peel relative to the first loop. The algorithm is provided in Figure 4.7, and Figure 4.8 illustrates its application using the dependence chain multigraph shown in Figure 4.6(b).

The dependence chain graphs in Figure 4.6 and Figure 4.8 represented dependences flowing between adjacent loops in a sequence. In general, dependences may flow between any pair of loops. For example, the code shown in Figure 4.9(a) has a dependence flowing from L1 to L3 with a distance of 2. The dependence chain graph for this example is shown in Figure 4.9(b). Applying the derivation algorithm for peeling in the absence of the dependence between L1 and L3 would result in a final weight of 1 for the vertex representing L3. However, with this dependence, the derivation algorithm assigns a final weight of 2 to reflect the maximum of accumulated amounts of peeling passing through L3.

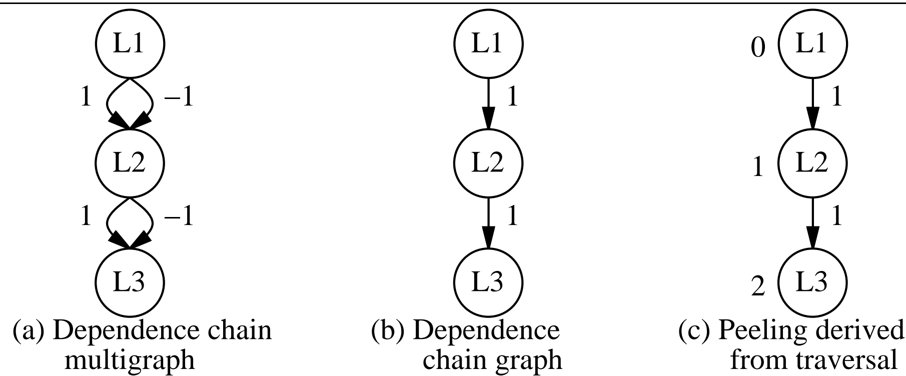


Figure 4.8: Deriving the required amount of peeling

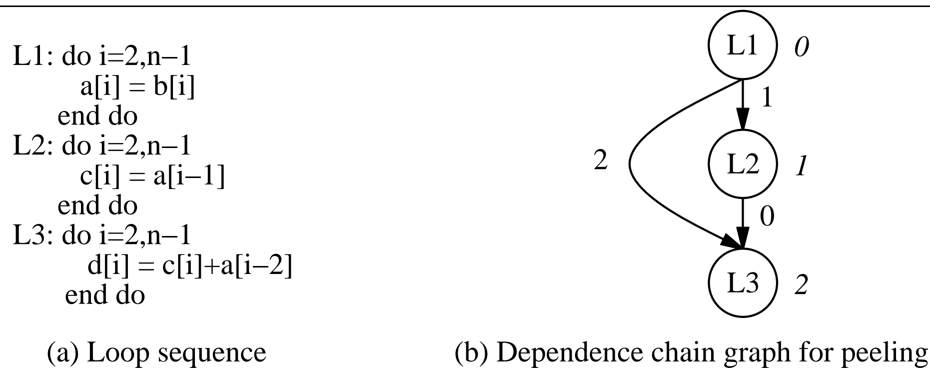


Figure 4.9: Dependence chain graph with dependences between non-adjacent loops

4.2.4 Implementation of Shift-and-peel

Once the required amounts of shifting and peeling have been derived, the loop nests must be transformed to complete the legal fusion. There are two methods to implement shift-and-peel. In the *direct method*, the original loop bodies are combined into a single body. The iterations of the fused loop are then divided into blocks to be executed in parallel on different processors. To implement shifting, array subscript expressions in statements from shifted loop nests must be adjusted wherever the index variable of the shifted loop appears. To implementing peeling, guards must be introduced for each statement from a loop that requires peeling. Figure 4.10(a)

```
do i=istart,iend
    a[i] = b[i]
    if (i  >=  istart+1) c[i−1] = a[i]+a[i−2]
    if (i  >=  istart+2) d[i−2] = c[i−1]+c[i−3]
end do

c[iend] = a[iend+1] + a[iend−1]

do i=iend−1,iend
   d[i] = c[i+1]+c[i−1]
end do
```

(a) Direct method

```
do ii=istart,iend,s
   do i=ii,min(ii+s−1,n−1)
      a[i] = b[i]
   end do
   do i=max(ii−1,istart+1),min(ii+s−2,iend−1)
      c[i] = a[i+1]+a[i−1]
   end do
   do i=max(ii−2,istart+2),min(ii+s−3,iend−2)
      d[i] = c[i+1]+c[i−1]
   end do
end do

c[iend] = a[iend+1]+a[iend−1]
do i=iend−1,iend
   d[i] = c[i+1]+c[i−1]
end do
```

(b) Strip−mined method

Figure 4.10: Alternatives for implementing fusion with shift-and-peel

illustrates this approach for a block of iterations `istart...iend` executed by one processor. Note that a small number of iterations from shifted loops are executed outside the fused loop.

The alternative to the direct method is to use *strip-mining*. This approach assumes that the number of iterations exceeds the number of processors, a reasonable assumption when locality enhancement is required. The original loops are strip-mined by a factor of s, then the resulting outer control loops are fused, as shown in Figure 4.10(b). In this method, shifting only requires adjustments to inner loop bound expressions, leaving the subscript expressions unchanged. Peeling is also implemented by adjusting inner loop bound expressions. Strip-mining also accommodates differing iteration spaces by modifying the *min, max* expressions in the inner loop bounds. Finally, the strip-mined method may also reduce register pressure. The only drawback to strip-mining is that it may incur more loop overhead in comparison to the direct approach. However, a larger strip size s reduces this overhead. But the choice of s is also constrained by the cache capacity because s determines the amount of data that must remain cached for reuse; this issue is addressed in Chapter 6. Nonetheless, in light of its advantages, strip-mining is selected as the implementation method for shift-and-peel in this thesis.

The only remaining issue is the execution of the iterations peeled to enable parallel execution.

```
do ii=istart,iend,s
   do i=ii,min(ii+s−1,iend)
      a[i] = b[i]
   end do
   do i=max(ii−1,istart+1),min(ii+s−2,iend−1)
      c[i] = a[i+1]+a[i−1]
   end do
   do i=max(ii−2,istart+2),min(ii+s−3,iend−2)
      d[i] = c[i+1]+c[i−1]
   end do
end do
<BARRIER>
do i=iend,iend+1
   c[i] = a[i+1]+a[i−1]
end do
do i=iend−1,iend+2
   d[i] = c[i+1]+c[i−1]
end do
```

Figure 4.11: Complete implementation of fusion with shift-and-peel

These iterations are peeled from the start of each block on different processors and can only be executed after all preceding iterations have been executed; a barrier synchronization can be inserted to ensure that this condition is satisfied. Iterations peeled from the same block are grouped into sets. There are no dependences between different sets of peeled iterations (proved later in Section 4.2.5), although there may be dependences within each set. As a result, these sets of peeled iterations may also be executed in parallel without synchronization.

Shifting causes a number of iterations to be executed outside the fused loop. These iterations are at the end of blocks assigned to different processors. Because there may be dependences between the iterations at the end of a block assigned to one processor, and the iterations peeled from the start of the adjacent block assigned to another processor, these iterations are collected into subsets such that all dependences are contained entirely within each set. In this manner, these subsets of iterations may be executed in parallel. Figure 4.11 illustrates the complete code that implements fusion with shift-and-peel. Peeled iterations are executed after a barrier to ensure all preceding iterations have been executed. The iterations executed after the barrier include those excluded from block `istart...iend` because of shifting, and also those peeled from the start of the block beginning at `iend+1`. Note that the implementation in Figure 4.11

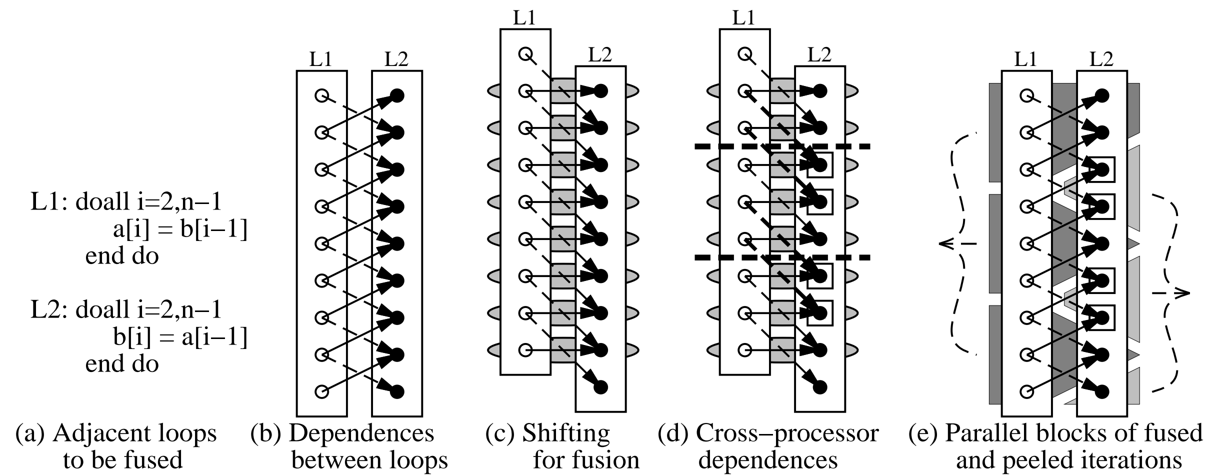


Figure 4.12: Legality of the shift-and-peel transformation

is essentially independent of the number of processors. The values of `istart` and `iend` may be calculated at runtime based on the loop bounds and number of processors available for parallel execution on entry to the fused loop.

Finally, the implementation must also account for minor differences in the transformed code for processors executing blocks at the boundaries of the full iteration space. For the processor executing the block containing iterations from the beginning of the iteration space of the fused loop, there are no iterations to be peeled; only shifting is implemented in the fused loop. However, this processor does execute the peeled iterations for the adjacent block following the barrier synchronization. The processor executing the block containing iterations from the end of the iteration space does not execute any iterations peeled for parallelization after the barrier synchronization because there is no subsequent block of iterations.

4.2.5 Legality of the Shift-and-peel Transformation

This section first presents an intuitive argument for the legality of the shift and peel transformation. This argument is then substantiated with a formal proof.

Consider the example sequence of parallel loops in Figure 4.12(a); this example contains both forward and backward dependences between the two loops. The dependences are illustrated in Figure 4.12(b). Because each loop is parallel, there are no loop-carried dependences

within the individual loops. The antidependence between L_1 and L_2 caused by references to the array b is uniform with a distance of -1, and hence it prevents fusion. In general, there may be several such fusion-preventing dependences with different distances. The derivation algorithm in Figure 4.5 always selects the amount of shifting according to the minimum dependence distance between the loops. Similarly, the flow dependence for array a is also uniform with a distance of 1, hence it serializes execution if the backward dependence is ignored and the loops are fused. In general, there may be several serializing dependences with different distances. In the derivation algorithm, the amount of peeling is always determined by the dependence with the maximum distance, as discussed in Section 4.2.3.

Based on the antidependence with the distance of -1, L_2 is shifted by one iteration with respect to L_1 to permit legal fusion. This is shown in Figure 4.12(c). The computation performed in each pair of iterations identified by the shading in Figure 4.12(c) would be performed in one loop iteration if the two loops were to be fused directly. The original dependence distance of -1 is transformed to 0, since it is the minimum distance. All other dependence distances, including the forward dependence distance of 1, are increased, but this does not prevent legal fusion. Hence it is always legal to perform fusion after shifting by the amount needed to satisfy the minimum dependence distance.

Now, consider parallel execution of the fused loop as shown in Figure 4.12(d), where each processor is assigned a contiguous block of iterations. There are now cross-processor dependences flowing between processors, hence the blocks must be executed serially. The iterations from L_2 identified with a square in Figure 4.12(d) are the sink iterations of these cross-processor dependences. In the absence of shifting, some of these sink iterations would otherwise be executed in the same processor as their corresponding source iterations. However, shifting moves each of these sink iterations to an adjacent processor. The number of such iterations per block is equal to the amount of shifting. The remaining sink iterations would still generate cross-processor dependences even without shifting and therefore require peeling. The number of such iterations is equal to the maximum distance among all original forward dependences. For the example in Figure 4.12, there is one such iteration per block.

To permit parallel execution, iterations that would otherwise become sinks of cross-processor dependences are peeled out of L_2 prior to fusion. Of each pair of iterations peeled

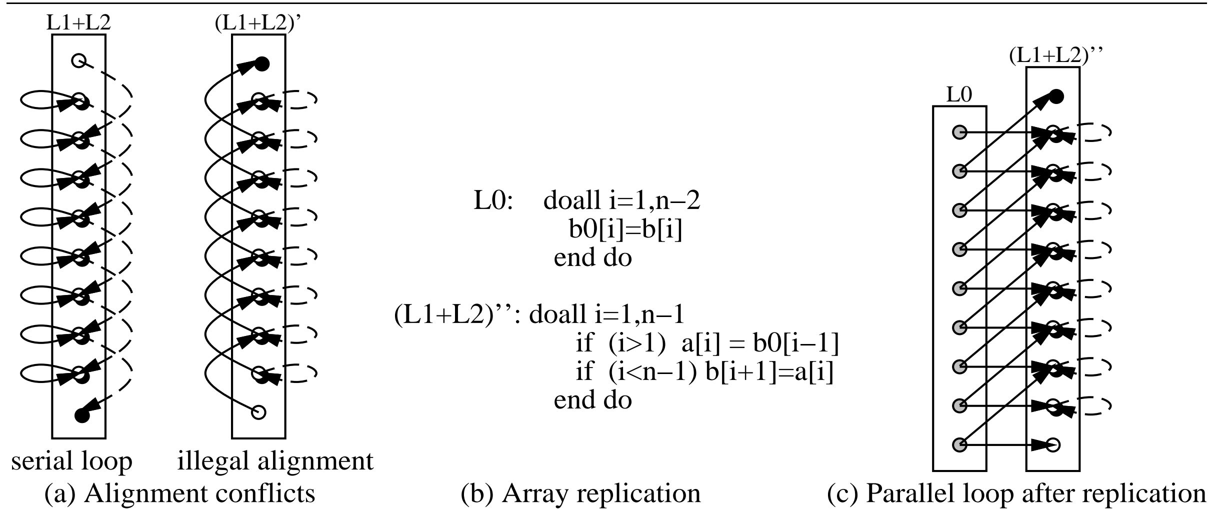


Figure 4.13: Resolution of alignment conflicts with replication

from the blocks of iterations in Figure 4.12(d), one iteration must be peeled out as a consequence of shifting, and the other due to the original forward dependence with a distance of 1. The shift-and-peel transformation thus groups the computations into the blocks of fused and peeled iterations shown in Figure 4.12(e). The blocks of fused iterations are executed in parallel, then the blocks of peeled iterations are executed in parallel after a barrier.

Based on Figure 4.12, the shift-and-peel transformation is always legal. First, no dependences flow between blocks of fused iterations by virtue of peeling iterations that would otherwise serialize execution. Within each block of fused iterations, shifting to satisfy the minimum dependence distance ensures that the fusion is indeed legal, as shown in Figure 4.12(c). Second, no dependences flow between blocks of peeled iterations. Dependences either flow from a block of fused iterations to a block of peeled iterations, or they flow within the same block of peeled iterations, and are satisfied by the execution order within the block of peeled iterations. Finally, since dependences only flow from blocks of fused iterations to blocks of peeled iterations, the barrier synchronization ensures that these are always satisfied.

It is interesting to note that dependence relationships in the fused loop shown in Figure 4.12(c) lead to an alignment conflict that requires replication if parallel execution is enabled using the techniques proposed by Callahan[Cal87] and Appelbe and Smith [AS92]. This con-

flict is illustrated in Figure 4.13(a). The loop that results from fusion is serial due to a forward loop-carried dependence. This forward dependence is the flow dependence for array a. If the computations in the loop are aligned as shown in Figure 4.13(a) such that the forward dependence is made loop-independent, a backward dependence results, hence the alignment is illegal. Alignment for parallel execution is not possible because the alignment requirements of the different dependences conflict with each other. To resolve this alignment conflict, replication is required. In Figure 4.13(b), a new loop L_0 replicates the array b into a new array $b0$, and the values of array $b0$ are read in the aligned version of the fused loop, rather than array b. As a result, the backward loop-carried dependence is removed, and the aligned loop is not only legal, but may also be executed in parallel, since it no longer contains any loop-carried dependences. The new loop L_0 may also be executed in parallel. However, L_0 may not be fused with the aligned loop because the original alignment conflict would then reappear.

In general, both data *and* computation replication are required to address alignment conflicts. Replicating computation contributes execution time overhead, while replicating data contributes memory overhead. In contrast, the shift-and-peel transformation does not require any replication to enhance locality while preserving parallelism.

Formal Proof of Legality

For simplicity, this proof is presented for sequences of parallel loops with identical loop bounds. First, a number of definitions are provided.

Definition 1 *A sequence of loops $L_1, \ldots, L_n$ is an admissible* parallel loop sequence *if there is no intervening code between the loops, if each loop L_k $(1 \leq k \leq n)$ is parallel, and if all loops use the same integer index variable* `I` *with the same integer lower/upper bounds ℓ and u ($\ell \leq u$) and a step of 1. The loop sequence is totally ordered, i.e., $L_1 \prec L_2 \prec \cdots \prec L_n$. The computation performed for an iteration* `I`$=i$ *$(\ell \leq i \leq u)$ within the body of a loop L_k $(1 \leq k \leq n)$ is denoted by $S_k(i)$.*

Definition 2 *For a loop L_k in a parallel loop sequence $L_1, \ldots, L_n$, the set of all memory locations read in a given iteration i of the loop body $S_k(i)$ is denoted by $\mathcal{R}_k(i)$. Similarly, the set of all memory locations written is denoted by $\mathcal{W}_k(i)$.*

Definition 3 *For a pair of loops L_a, L_b in a parallel loop sequence $L_1, \ldots, L_n$, where $L_a \prec L_b$, an* interloop dependence $S_a(i_1)\delta S_b(i_2)$ *exists between iteration i_1 in L_a and iteration i_2 in L_b if*

$$[\mathcal{R}_a(i_1) \cap \mathcal{W}_b(i_2) \neq \emptyset] \vee [\mathcal{W}_a(i_1) \cap \mathcal{R}_b(i_2) \neq \emptyset] \vee [\mathcal{W}_a(i_1) \cap \mathcal{W}_b(i_2) \neq \emptyset],$$

where $\ell \leq i_1 \leq u$ and $\ell \leq i_2 \leq u$. The dependence distance is given by $i_2 - i_1$, and may be positive, negative, or zero.

Definition 4 *Let L_a and L_b denote a pair of loops in a parallel loop sequence $L_1, \ldots, L_n$ such that $L_a \prec L_b$. Let $DEP_{a,b}$ denote the set of all interloop dependences $S_a(i_1)\delta S_b(i_2)$ between L_a and L_b. Let $DEP_{a,b}(d)$ denote the subset of all interloop dependences between the loops L_a and L_b with distance d. Let $DIST_{a,b}$ denote the set of all distances d such that $DEP_{a,b}(d) \neq \emptyset$. $DEP_{a,b}$ is a set of* uniform interloop dependences *if:*

$$\forall d \in DIST_{a,b}, \ \ \exists S_a(i)\delta S_b(i+d) \in DEP_{a,b}(d), \ \ \begin{cases} \forall \ell \leq i \leq u-d, & \text{if } d \geq 0, \\ \forall \ell - d \leq i \leq u, & \text{if } d < 0. \end{cases}$$

Uniformity requires interloop dependences with distance d to flow from all *iterations i in L_a to $i+d$ in L_b, subject to the loop bound constraints.*

Definition 5 *For a parallel loop sequence $L_1, \ldots, L_n$ in which all interloop dependences are uniform, let $shift(k) \leq 0$ and $peel(k) \geq 0$ denote the amounts of shifting and peeling derived for each loop L_k $(1 \leq k \leq n)$ by the shift-and-peel derivation algorithm. Let P denote the number of processors to be used for parallel execution. Let $istart(p)$ and $iend(p)$ denote the starting and ending iterations for a subset of consecutive iterations from the original iteration space bounded by ℓ and u to be executed by a processor p $(1 \leq p \leq P)$, i.e.,*

$$istart(p) = \ell + \left\lfloor \frac{u-\ell+1}{P} \right\rfloor \cdot (p-1), \quad iend(p) = \begin{cases} istart(p) + \left\lfloor \frac{u-\ell+1}{P} \right\rfloor - 1, & 1 \leq p < P, \\ u, & p = P. \end{cases}$$

The shift-and-peel transformation produces a fused loop whose iterations are executed in parallel on P processors, followed by a barrier synchronization, which is then followed by peeled loop iterations that are also executed in parallel on P processors. For each processor p $(1 \leq p \leq P)$, $FUSED(p)$ is the subset of computations from the fused loop, i.e.,

$$FUSED(p) = \begin{cases} \{S_k(i) \mid istart(p) \leq i \leq iend(p) + shift(k), \ 1 \leq k \leq n\}, & p = 1, \\ \{S_k(i) \mid istart(p) + peel(k) \leq i \leq iend(p) + shift(k), \ 1 \leq k \leq n\}, & 1 < p \leq P. \end{cases}$$

Similarly, PEELED(p) is the subset of peeled computations for a processor p, i.e.,

$$\textit{PEELED}(p) = \begin{cases} \{S_k(i) \mid \textit{iend}(p) + \textit{shift}(k) + 1 \leq i \leq \textit{iend}(p) + \textit{peel}(k), 1 \leq k \leq n\}, & 1 \leq p < P, \\ \{S_k(i) \mid \textit{iend}(p) + \textit{shift}(k) + 1 \leq i \leq \textit{iend}(p), 1 \leq k \leq n\}, & p = P. \end{cases}$$

Definition 6 *For a parallel loop sequence $L_1, \ldots, L_n$ in which all interloop dependences are uniform, let shift$(k) \leq 0$ and peel$(k) \geq 0$ denote the amounts of shifting and peeling derived for each loop L_k $(1 \leq k \leq n)$ by the shift-and-peel derivation algorithm. The* iteration count threshold N_t *for the parallel loop sequence is defined as*

$$N_t = \max_{1 \leq k \leq n} (\textit{peel}(k) - \textit{shift}(k)).$$

Definition 6 is a consequence of the implementation of the shift-and-peel transformation discussed in Section 4.2.4, which assumes that the number of iterations per original loop is much greater than the number of processors (which in turn implies that locality enhancement with fusion is required). The iteration count threshold asserts that shifting and peeling for a given original loop do not remove more computations from the resulting FUSED(p) subsets than the number of iterations per processor. Exceeding this threshold indicates that all computation from one of the original loops is excluded from the fused loop, which clearly defeats the purpose of loop fusion for locality.

With the preceding set of definitions, the following theorem on the legality of the shift-and-peel transformation and the implementation discussed in Section 4.2.4 may now be proved.

Theorem 1 *The shift-and-peel transformation is always legal for a parallel loop sequence $L_1, \ldots, L_n$ in which all interloop dependences are uniform, provided that*

$$\left\lfloor \frac{u - \ell + 1}{P} \right\rfloor \geq N_t,$$

where P is the number of processors used in parallel execution of the resulting loop, $u - \ell + 1$ is the number of iterations in each of the loops of the original parallel loop sequence, and N_t is the iteration count threshold in Definition 6.

Proof First, all of the original computation is performed in the transformed code. Using Definition 5, this condition is satisfied by noting that the lower and upper bounds for the fused loop in each processor together with the peeled iterations cover the original computation, i.e.,

$$\bigcup_{1 \leq p \leq P} (\textit{FUSED}(p) \cup \textit{PEELED}(p)) = \{S_k(i) \mid \ell \leq i \leq u, \; 1 \leq k \leq n\}.$$

Second, there is no redundancy in the computation. To show that each component of the original computation is performed by exactly one processor, it is necessary to show that the subsets of computation assigned to different processors are disjoint. Because these subsets contain computation from consecutive iterations, it is sufficient to show that for processors p and $p+1$,

$$\mathit{PEELED}(p) \cap \mathit{PEELED}(p+1) = \emptyset,$$

for which Definition 5 implies that the following condition must be satisfied:

$$\forall 1 \leq p < P, \forall 1 \leq k \leq n, \mathit{iend}(p) + \mathit{peel}(k) < \mathit{iend}(p+1) + \mathit{shift}(k) + 1.$$

Substituting for $\mathit{iend}(p)$ and $\mathit{iend}(p+1)$ using Definition 5 and simplification results in

$$\forall 1 \leq k \leq n, \left\lfloor \frac{u - \ell + 1}{P} \right\rfloor + 1 > \mathit{peel}(k) - \mathit{shift}(k).$$

Since it must be true for all loops, it must be true for the loop for which $\mathit{peel}(k) - \mathit{shift}(k)$ is the largest, and this is given by the iteration count threshold N_t. Since both $\left\lfloor \frac{u-\ell+1}{P} \right\rfloor$ and N_t are integers, the condition may be simplified to

$$\left\lfloor \frac{u - \ell + 1}{P} \right\rfloor \geq N_t.$$

With this condition satisfied, it can also be shown using Definition 5 that $\mathit{FUSED}(p) \cap \mathit{FUSED}(p+1) = \emptyset$, $\mathit{PEELED}(p) \cap \mathit{FUSED}(p+1) = \emptyset$, and $\mathit{PEELED}(p+1) \cap \mathit{FUSED}(p) = \emptyset, \forall 1 \leq p < P$.

Third, none of the original uniform interloop dependences are violated when the $\mathit{FUSED}(p)$ subsets are executed in parallel on P processors. For interloop dependences $S_a(i)\delta S_b(i+d)$ such that $S_a(i), S_b(i+d) \in \mathit{FUSED}(p)$ and $d < 0$, shifting trivially ensures that the dependences are satisfied internally within each subset. Dependences with $d > 0$ are always satisfied internally even with shifting. Furthermore, no dependences flow between the $\mathit{FUSED}(p)$ subsets executed in parallel on different processors. This is shown with the following proof by contradiction.

For $S_a(i)\delta S_b(i+d)$, assume that $S_a(i) \in \mathit{FUSED}(p_1)$ and $S_b(i+d) \in \mathit{FUSED}(p_2)$, where $p_1 \neq p_2$. For $d \geq 0$, assume that $p_2 > p_1$. The shift-and-peel derivation algorithm results in $\mathit{peel}(b) \geq d$. If $S_a(i) \in \mathit{FUSED}(p_1)$, the maximum value of i is $i = \mathit{iend}(p_1) + \mathit{shift}(a)$ by Definition 5. If $S_b(i+d) \in \mathit{FUSED}(p_2)$, where $p_2 > p_1$, then by Definition 5,

$$i + d = \mathit{iend}(p_1) + \mathit{shift}(a) + d \geq \mathit{istart}(p_1 + 1) + \mathit{peel}(b).$$

Substituting for $iend(p_1)$ and $istart(p_1 + 1)$ from Definition 5 and rearranging results in

$$d - 1 + \mathit{shift}(a) \geq \mathit{peel}(b).$$

Since $\mathit{shift}(a) \leq 0$, and $\mathit{peel}(b) \geq d$,

$$d - 1 \geq d - 1 + \mathit{shift}(a) \geq \mathit{peel}(b) \geq d.$$

But $d - 1 < d$, hence this is a contradiction. Since the *maximum* iteration i such that $S_a(i) \in \mathit{FUSED}(p_1)$ was used, this contradiction is true for all iterations i such that $S_a(i) \in \mathit{FUSED}(p_1)$. A similar contradiction results from assuming that $p_2 < p_1$ for $d < 0$. Thus, no dependences flow between the $\mathit{FUSED}(p)$ subsets for any pair of processors.

Fourth, none of the original uniform interloop dependences are violated when the $\mathit{PEELED}(p)$ subsets are executed in parallel on P processors. Any interloop dependences $S_a(i)\delta S_b(i + d)$ such that $S_a(i), S_b(i + d) \in \mathit{PEELED}(p)$ are always satisfied because iterations peeled from loop L_a are executed before iterations peeled from L_b. Furthermore, no dependences flow between different $\mathit{PEELED}(p)$ subsets. This may be shown with a similar proof by contradiction as for the $\mathit{FUSED}(p)$ subsets; it is omitted here for brevity.

Finally, none of the original uniform interloop dependences are violated across the synchronization point between the execution of $\mathit{FUSED}(p)$ and $\mathit{PEELED}(p)$ on each processor because all interloop dependences either flow interally within each fused or peeled subset of iterations, or from a fused subset to a peeled subset. The total ordering implies that all dependences flow forward in the original sequence. For those dependences that require peeling, it is always the sink iteration that is peeled. Any other iterations that depend on a peeled iteration are also peeled by virtue of the shift-and-peel derivation algorithm. Thus, dependences never flow from a peeled subset to a fused subset. The synchronization point ensures that those dependences flowing from a fused subset to a peeled subset are always satisfied.

Since the transformed code executes all of the original computation without redundancy (provided that the iteration count threshold is satisfied), and none of the original interloop dependences are violated internally within the subsets of iterations executed by different processors or externally between the subsets, the shift-and-peel transformation is legal.□

4.3 Multidimensional Shift-and-peel

4.3.1 Motivation

For a sequence of parallel loop nests, fusion of outermost loops produces a single loop nest, and the shift-and-peel transformation enables legal fusion and parallelization. However, there are two reasons why fusion of outermost loops may not be sufficient.

First, although fusion increases the granularity of parallelism in the outermost loop, it does not increase the degree of parallelism. For parallel execution on large-scale multiprocessors, a greater degree of parallelism is typically required to fully utilize a large number of processors. By fusing inner parallel loops in addition to the outermost loop, the degree of parallelism may be increased,[2] although the granularity of the resulting parallelism is reduced as a consequence.

Second, fusion of multiple loops increases the amount of data accessed in the resulting fused loop. A significant portion of this data may have to remain cached for reuse. If only the outermost loop is fused in a sequence of loop nests, the amount of data that must remain cached across iterations of the fused outermost loop may overflow the cache capacity. By fusing inner loops as well as the outermost loop, the execution order of the computation is further modified to reduce the amount of data that must remain cached for reuse.

In both cases, dependences flowing between iterations of inner loops may become loop-carried after fusion, and hence these dependences may render fusion illegal or prevent parallelization. However, the shift-and-peel transformation may also be applied to the inner loop levels in order to overcome such dependences.

4.3.2 Derivation

The derivation of the appropriate amounts of shifting and peeling to enable fusion and parallelization of inner loops uses the same approach as the derivation for the outermost loop. The same algorithms are used, but the dependence distances at each inner loop level are considered, rather than the distances at the outermost loop level.

Rather than reapplying the derivation algorithms at each level, it is possible to modify the algorithms for only one application. Instead of maintaining just one weight at each vertex

[2]Section 2.4.2 discussed how to increase the degree of parallelism by making all parallel loops adjacent in a loop nest.

representing the accumulated shift or peel amounts, it is possible to maintain a vector of weights, with one element for each loop level. In one traversal of the dependence chain graph, shift or peel amounts are propagated at all loop levels as each vertex is visited by the algorithm. The original algorithm is linear in the graph size, and the computation at each vertex increases by only a constant amount, hence the complexity remains linear in the graph size.

4.3.3 Implementation

Fusing multidimensional loop nests with strip-mining for *serial* execution does not present difficulties since only shifting is required. The loops being fused are strip-mined, the control loops are moved to the outermost level, and then the control loops are fused. As before, shifting is reflected in the inner loop bounds. New loop nests are then introduced to execute iterations that are excluded from the fused computation as a result of shifting; more than one loop nest is required because these iterations do not constitute a simple rectangular region. Fusion with multidimensional shifting is illustrated using the loop nest sequence shown in Figure 4.14(a). The dependences between the two loop nests require shifting by one iteration in *both* inner and outer loops to enable fusion. The fused loop nest sequence (with shifting only) is shown Figure 4.14(b). The iteration spaces after shifting to enable fusion are shown in Figure 4.14(c).

However, multidimensional peeling to enable parallel execution is more complicated. The multidimensional iteration space is divided into blocks of iterations that are executed by different processors. For those processors that execute blocks on the boundaries the iteration space, there are slight differences in the loop code (as discussed in Section 4.2.4). For a one-dimensional iteration space, there are only three cases, as shown in Figure 4.15(a). This number is small enough to permit generating three different versions of the code. However, when fused inner loops are parallelized with peeling, the number of cases increases dramatically. For fusion of a two-dimensional iteration space, there are a total of 9 cases, as shown in Figure 4.15(b). For a three-dimensional iteration space, there are 27 cases, as shown in Figure 4.15(c). Generating 27 different versions of the code is unnecessary because the differences are quite minor.

The differences between the various cases center on the execution of the iterations peeled to enable parallel execution. Instead of generating multiple versions, only one set of loops is generated, with the different cases reflected in a number of variables that control peeling in

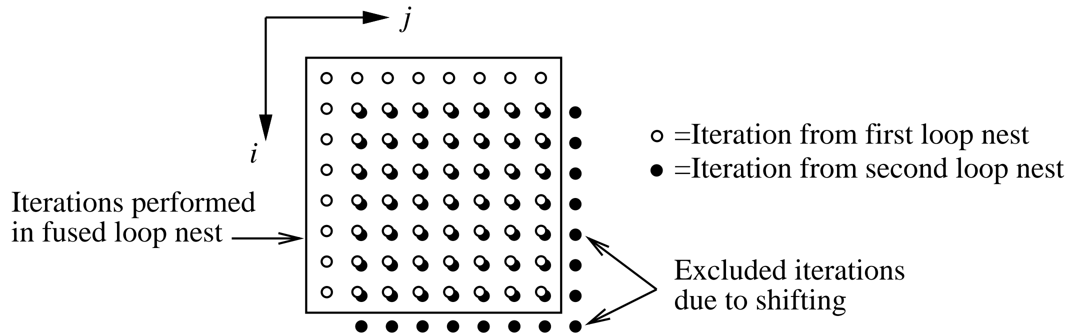
```
do j=2,n−1
  do i=2,n−1
    b[i,j] = (a[i,j−1]+a[i,j+1]
            +a[i−1,j]+a[i+1,j])/4
  end do
end do

do j=2,n−1
  do i=2,n−1
    a[i,j] = b[i,j]
  end do
end do
```

(a) Original loop nest sequence

```
do jj=2,n−1,sj
  do ii=2,n−1,si
    do j=jj,min(jj+sj−1,n−1)
      do i=ii,min(ii+si−1,n−1)
        b[i,j] = (a[i,j−1]+a[i,j+1]
                 +a[i−1,j]+a[i+1,j])/4
      end do
    end do
    do j=max(jj−1,2),min(jj+sj−2,n−2)
      do i=max(ii−1,2),min(ii+si−2,n−2)
        a[i,j] = b[i,j]
      end do
    end do
  end do
end do
```

```
do j=2,n−2
  do i=n−1,n−1
    a[i,j] = b[i,j]
  end do
end do
do j=n−1,n−1
  do i=2,n−1
    a[i,j] = b[i,j]
  end do
end do
```

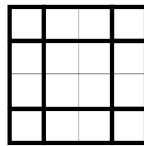
(b) Fused loop nest sequence with iterations excluded due to shifting



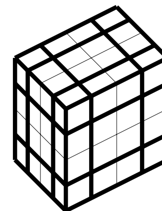
(c) Iteration spaces after shifting to enable fusion

Figure 4.14: Fusion with multidimensional shifting

(a) One dimension: 3 cases



(b) Two dimensions: 9 cases



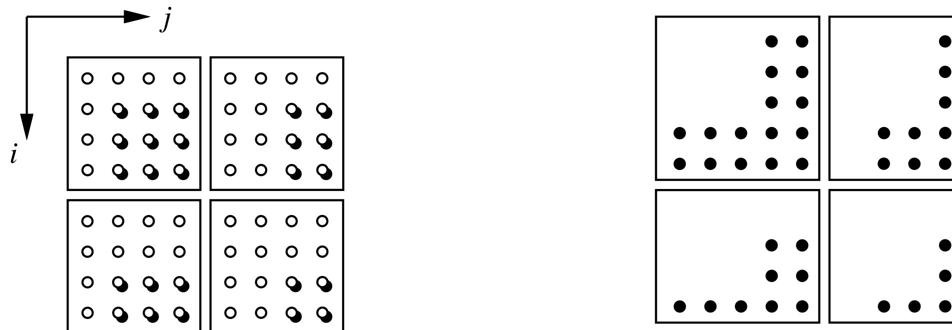
(c) Three dimensions: 27 cases

Figure 4.15: Enumerating the number of cases for multidimensional shift-and-peel

the fused loops and the subsequent execution of peeled iterations. The values for these control variables are determined by a prologue to the fused loop nest that computes the case that this instance of the code represents. The prologue determines which boundary or boundaries the block of iterations includes, then sets the flags to control the peeled iterations accordingly. This

```
JNPROCS = <#processors along j–dimension>
INPROCS = <#processors along i–dimension>
jp = mypid / JNPROCS
ip = mypid % INPROCS
jblksz = j_trip_count / JNPROCS
iblksz = i_trip_count / INPROCS
jstart = 2+jp * jblksz
istart = 2+ip * iblksz
if (jp == JNPROCS – 1)
  jend = n–1
else
  jend = jstart + jblksz
endif
if (ip == INPROCS – 1)
  iend = n–1
else
  iend = istart + iblksz
endif
left     =  (ip == 0)
right    =  (ip == INPROCS – 1)
top      =  (jp == 0)
bottom = (jp == JNPROCS – 1)
jfpeel  =  (left) ? 0 : 1
ifpeel  =  (top) ? 0 : 1
jppeel  =  (right) ? 0 : 1
ippeel  =  (bottom) ? 0 : 1

do jj=jstart,jend,sj
  do ii=istart,iend,si
    do j=jj,min(jj+sj–1,jend)
      do i=ii,min(ii+si–1,iend)
        b[i,j] = (a[i,j–1]+a[i,j+1]
                 +a[i–1,j]+a[i+1,j])/4
      end do
    end do
    do j=max(jj–1,jstart+jfpeel),min(jj+sj–2,jend–1)
      do i=max(ii–1,istart+ifpeel),min(ii+si–2,iend–1)
        a[i,j] = b[i,j]
      end do
    end do
  end do
end do
<BARRIER>
do j=jstart,jend–1
  do i=iend,iend+ippeel
    a[i,j] = b[i,j]
  end do
end do
do j=jend,jend+jppeel
  do i=istart,iend+ippeel
    a[i,j] = b[i,j]
  end do
end do
```

Figure 4.16: Parallelization with multidimensional peeling



(a) Independent blocks of fused iterations (b) Independent blocks of peeled iterations

Figure 4.17: Independent blocks of iterations with multidimensional shift-and-peel

approach is shown for the example in Figure 4.16. The dependences between the two loop nests require peeling by one iteration in *both* inner and outer loops. Iterations are grouped into independent blocks for distribution on a grid of processors, as shown in Figure 4.17. Note that the blocks in Figure 4.17(b) include iterations excluded from the fused loops as a result of shifting, as well as iterations peeled for parallelization.

4.3.4 Legality of Multidimensional Shift-and-peel

The legality for multidimensional shift-and-peel follows directly from the legality of shift-and-peel for outermost loops. Just as shifting of outermost loops ensures that there are no backward-flowing dependences at the outermost level, shifting of inner loops ensures that there are no backward-flowing dependences carried by inner loops. Similarly, peeling iterations from inner loops removes cross-processor dependences. Peeled iterations are executed only after all fused loop iterations have been executed, and no dependences are violated.

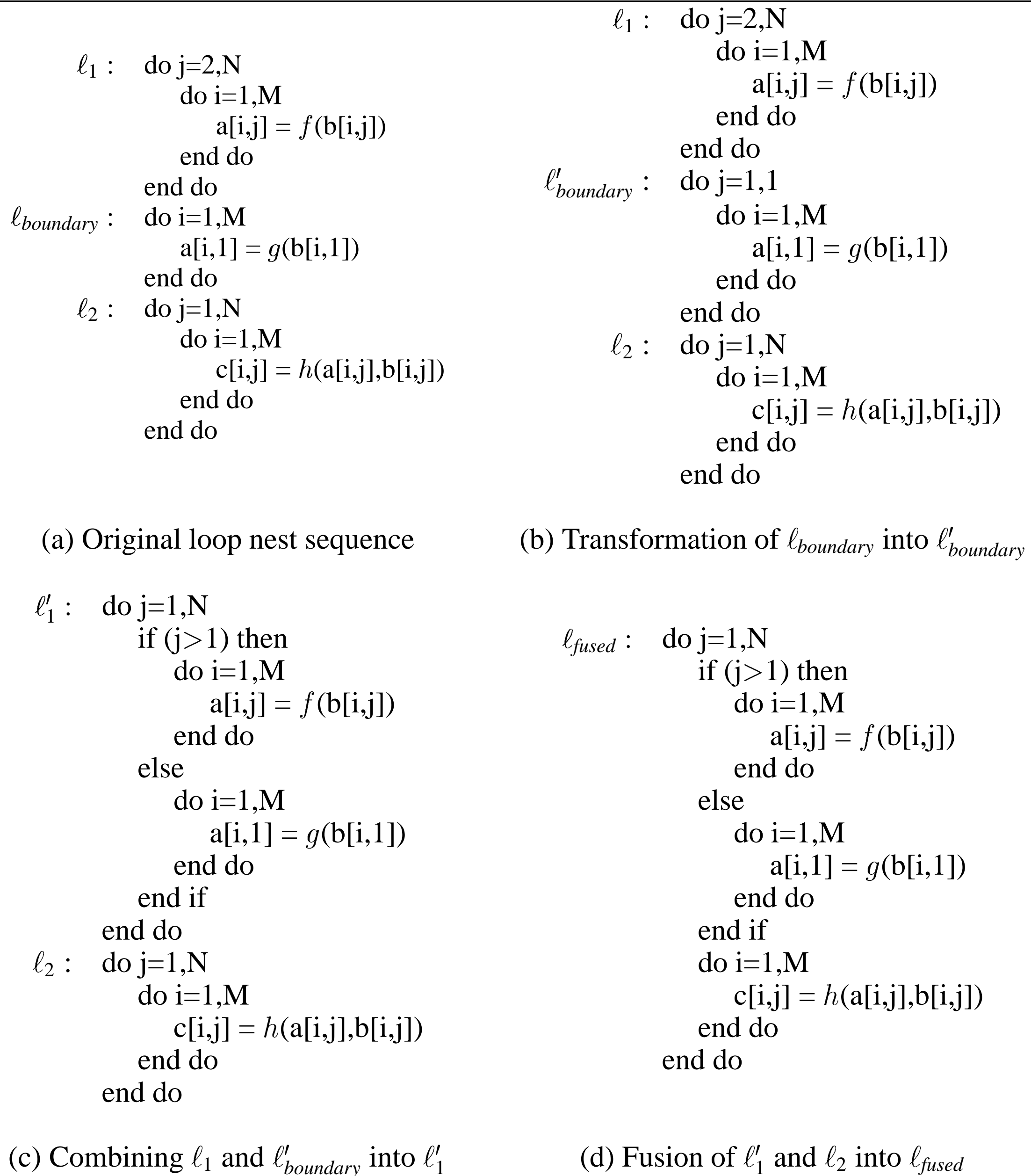
4.4 Fusion with Boundary-scanning Loop Nests

A final issue affecting loop fusion is the presence of *boundary-scanning* loop nests within a candidate sequence for fusion. A boundary-scanning loop nest accesses elements from a boundary region of a multidimensional array, which normally implies that the loop nest dimensionality is less than the array dimensionality. For example, a one-dimensional loop may access one of following boundary regions of a two-dimensional array: the first column, the last column, the first row, and the last row. Arrays with higher dimensionality have correspondingly more boundary regions.

A boundary-scanning loop nest normally appears in sequence with other *full-dimensionality* loop nests (i.e., loop nests whose dimensionality matches the array dimensionality). For example, the loop nest labelled ℓ_1 in Figure 4.18(a) writes all elements of array a, except elements in the first column. The loop labelled $\ell_{boundary}$ then writes only the first column of a with values that are computed differently than those computed in ℓ_1. Finally, the loop nest labelled ℓ_2 reads all values written to array a by both loops ℓ_1 and $\ell_{boundary}$.

A loop nest sequence that includes a loop nest such as $\ell_{boundary}$ in Figure 4.18(a) cannot be fused directly because of the differences between loop headers in the sequence. Fusion is limited to the subsets of loop nests that either precede or follow a boundary-scanning loop nest. As a result, opportunities to exploit data reuse across the *entire* loop nest sequence are lost. For example, arrays a and b in Figure 4.18(a) are reused across loop nests ℓ_1 and ℓ_2, but the presence of $\ell_{boundary}$ prevents the application of direct fusion to exploit that reuse.[3]

[3]Although $\ell_{boundary}$ may be moved ahead of ℓ_1 in Figure 4.18(a), code mobility is in general more restricted. A boundary-scanning loop nest may depend on data written by a preceding full-dimensionality loop nest, and the



(a) Original loop nest sequence

(b) Transformation of $\ell_{boundary}$ into $\ell'_{boundary}$

(c) Combining ℓ_1 and $\ell'_{boundary}$ into ℓ'_1

(d) Fusion of ℓ'_1 and ℓ_2 into ℓ_{fused}

Figure 4.18: Fusing a loop nest sequence with a boundary-scanning loop nest

However, fusion of loop sequences is still possible, even in the presence of boundary-scanning loops. The key to enabling fusion is determining the array regions that are written by a boundary-scanning loop nest and the regions written by neighboring loop nests. If these regions are disjoint, a straightforward transformation incorporates the computation performed

---

data written by the boundary-scanning loop nest may also be read by a subsequent loop nest.

in the boundary-scanning loop nest into a fused loop without violating loop semantics.

The transformation to enable fusion is illustrated using the loop nest sequence in Figure 4.18(a). To exploit the reuse of arrays a and b, the outermost j loops must be fused. The j loop headers for ℓ_1 and ℓ_2 differ by one iteration, namely $j = 1$. However, $\ell_{boundary}$ effectively performs the computation for $j = 1$. Hence, $\ell_{boundary}$ is transformed into a two-dimensional loop nest with an outer j loop of only one iteration, as shown in Figure 4.18(b).

The regions of array a written by ℓ_1 and $\ell'_{boundary}$ in Figure 4.18(b) are disjoint because the j-loop iteration values do not overlap. This feature is exploited by forming the union of the separate iteration spaces to produce a new loop ℓ'_1. The body of ℓ'_1 includes the computations from both ℓ_1 and $\ell'_{boundary}$, as shown in Figure 4.18(c); a guard selects the appropriate inner i loop. Note that this combination of loop bodies does *not* correspond to fusion; it is effectively the inverse of loop peeling.

Loops ℓ'_1 and ℓ_2 in Figure 4.18(c) are now fused directly to result in the loop ℓ_{fused} shown in Figure 4.18(d). The bounds of ℓ_{fused} range from 1 to N. When j is 1, the computation originally in loop $\ell_{boundary}$ is performed. When $j > 1$, the computation from loop ℓ_1 is performed.

Figure 4.18 illustrated a case in which direct fusion was applied after generating outermost loops with the same index variable. More generally, array references in loop nest sequences generate interloop dependences that require shift-and-peel. The approach illustrated in Figure 4.18 for boundary-scanning loop nests is still applicable in such cases. Once outermost loops with the same index variable are obtained, the dependence distances required for shift-and-peel are obtained for the core computation from the full-dimensionality loop nests. When the shift-and-peel transformation is applied with strip-mining as described in Section 4.2.4, the computation from boundary-scanning loop nests is automatically included in the appropriate block of iterations. The guard ensures that the boundary-scanning computation is performed only for the appropriate loop iteration. This is because strip-mining does *not* affect the loop body; only the bounds of the loop are modified.

Hence, the presence of boundary-scanning loop nests does not preclude the shift-and-peel transformation. Instead, the ability to incorporate boundary-scanning loop nests into a collection of full-dimensionality loop nests results in longer loop nest sequences for fusion.

4.5 Chapter Summary

This chapter has described the shift-and-peel transformation for enabling legal loop fusion and subsequent parallelization. The primary motivation for fusion of parallel loop nest sequences is locality enhancement, and the model described earlier in Chapter 3 has been used to quantify the benefit of locality enhancement. However, the motivation for the shift-and-peel transformation is the presence of dependences that either render fusion illegal or force a fused loop to be executed serially. Shifting and peeling have been shown to overcome such dependences and allow all reuse across a sequence of parallel loop nests to be exploited with fusion. The legality of the shift-and-peel transformation has been established with a formal proof. The transformation has also been described for fusion of inner loops as well as outermost loops. Finally, the presence of boundary-scanning loop nests within a candidate loop nest sequence for fusion has been addressed to ensure that all available reuse can be exploited.

Chapter 5

Scheduling Wavefront Parallelism in Tiled Loop Nests

This chapter describes scheduling strategies for tiled loop nests with wavefront parallelism, and analyzes the parallelism and locality provided by each strategy. Tiling a loop nest for cache locality enhancement introduces loop-carried dependences that limit parallelism to wavefronts in the tiled iteration space. These dependences result from using the shift-and-peel transformation and loop skewing to enable tiling. Scheduling the execution of a tiled loop nest with wavefront parallelism involves a tradeoff between the degree of parallelism in wavefronts and the extent of locality enhancement.

This chapter is organized as follows. First, the use of loop skewing and the shift-and-peel transformation to enable tiling is described. Next, the emergence of wavefront parallelism in tiled loop nests is discussed, followed by the tradeoff between parallelism and locality. Related work on tiling and loop scheduling is then outlined. Finally, scheduling strategies for wavefront parallelism are described and evaluated analytically.

5.1 Wavefront Parallelism in Tiled Loop Nests

5.1.1 Loop Skewing to Enable Legal Tiling

A perfectly-nested loop nest can be legally tiled if it is fully permutable, i.e., if none of its loop-carried dependence vectors have negative elements [Wol92]. Negative vector elements that are permuted to the outermost loop level result in lexicographically-negative dependence vectors that violate the original loop semantics. Loop skewing enables legal loop permutation by eliminating any negative elements in loop-carried dependences.

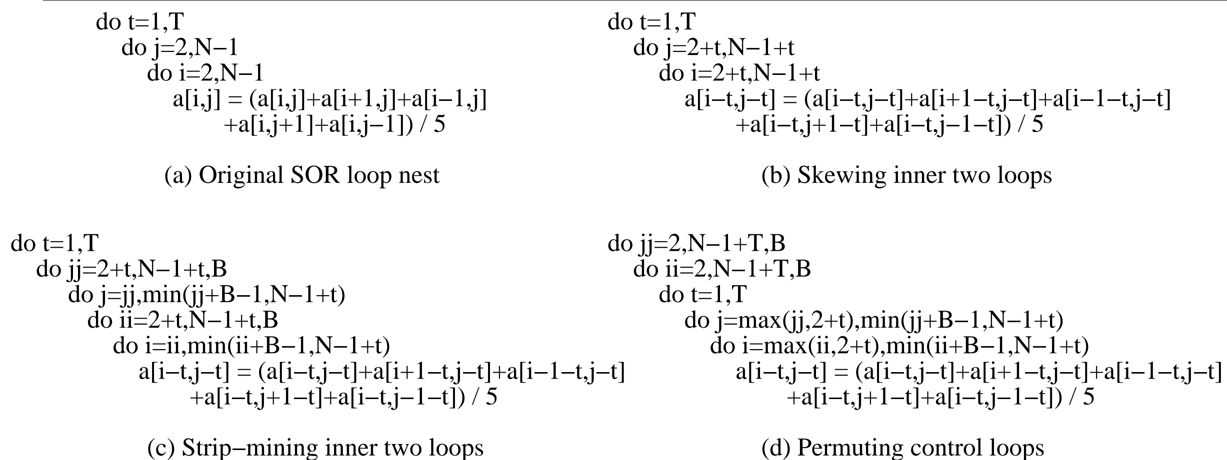


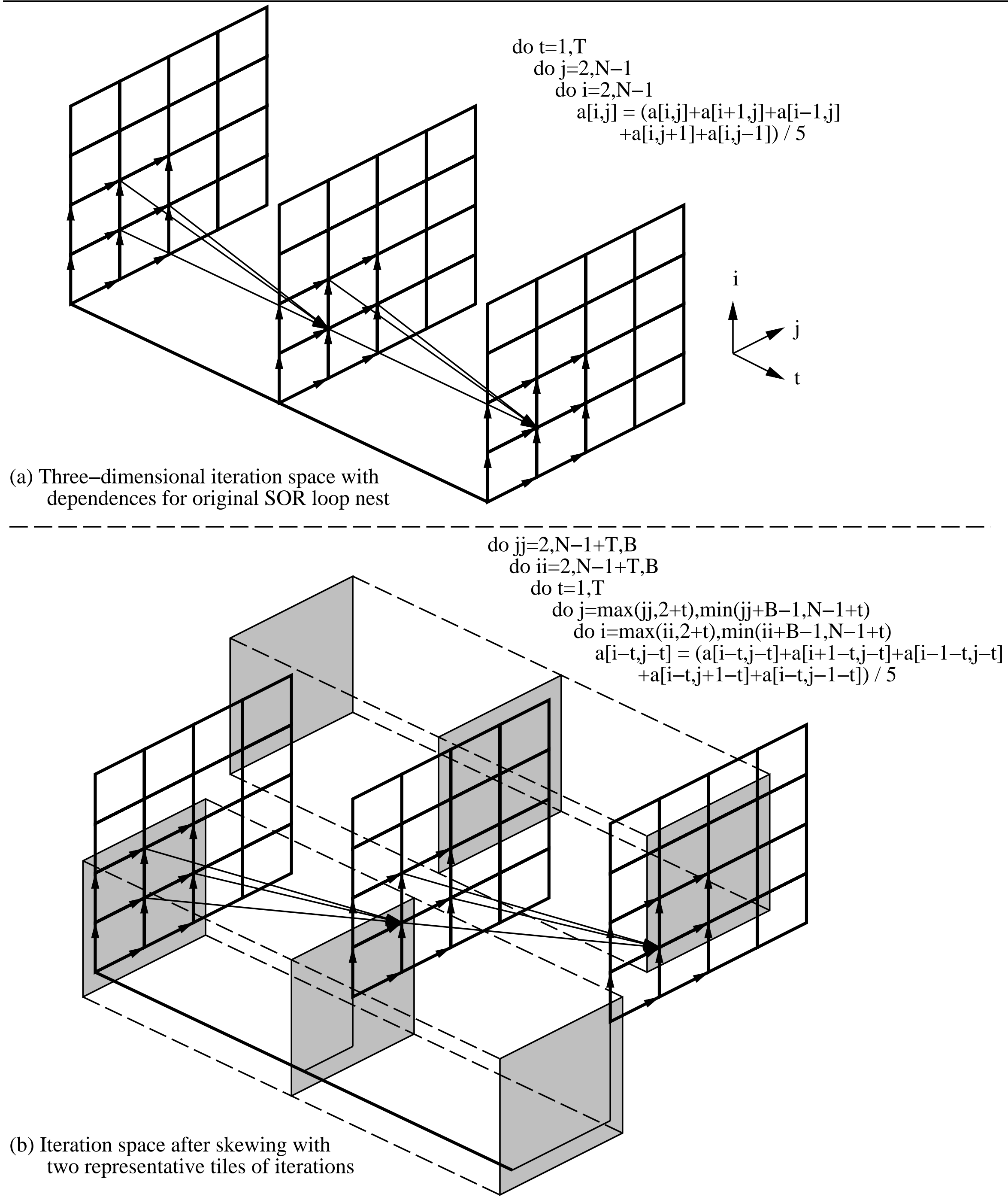
Figure 5.1: Steps in tiling the SOR loop nest

The SOR loop nest in Figure 5.1(a) is used to illustrate the use of loop skewing to enable tiling. This loop nest is a candidate for tiling because the outermost loop carries the dependence (1,0,0), hence it carries reuse. The complete set of dependence distance vectors for this loop nest is: $\{$(1,0,0), (1,$-$1,0), (1,0,$-$1), (0,1,0), (0,0,1)$\}$. Hence, the loop nest is not fully permutable, and loop skewing must be applied, as shown in Figure 5.1(b), in order to remove the negative elements in the distance vectors. Both inner loops i and j are skewed by one iteration with respect to loop t, resulting in the transformed distance vectors: $\{$(1,1,1), (1,0,1), (1,1,0), (0,1,0), (0,0,1)$\}$. The loop nest can be then tiled legally by first strip-mining the skewed i and j loops by a factor of B, as shown in Figure 5.1(c), and then by permuting the resulting ii and jj control loops to the outermost level, as in Figure 5.1(d). The effects of skewing on dependences and the grouping of iterations into units of tiles are illustrated graphically in Figure 5.2.

5.1.2 Enabling Tiling with the Shift-and-Peel Transformation

This section demonstrates how the shift-and-peel transformation proposed in Chapter 4 enables tiling. Figure 5.3 is used to illustrate the procedure. For simplicity, only one-dimensional tiling is illustrated. However, the following discussion can be extended to two or more dimensions.

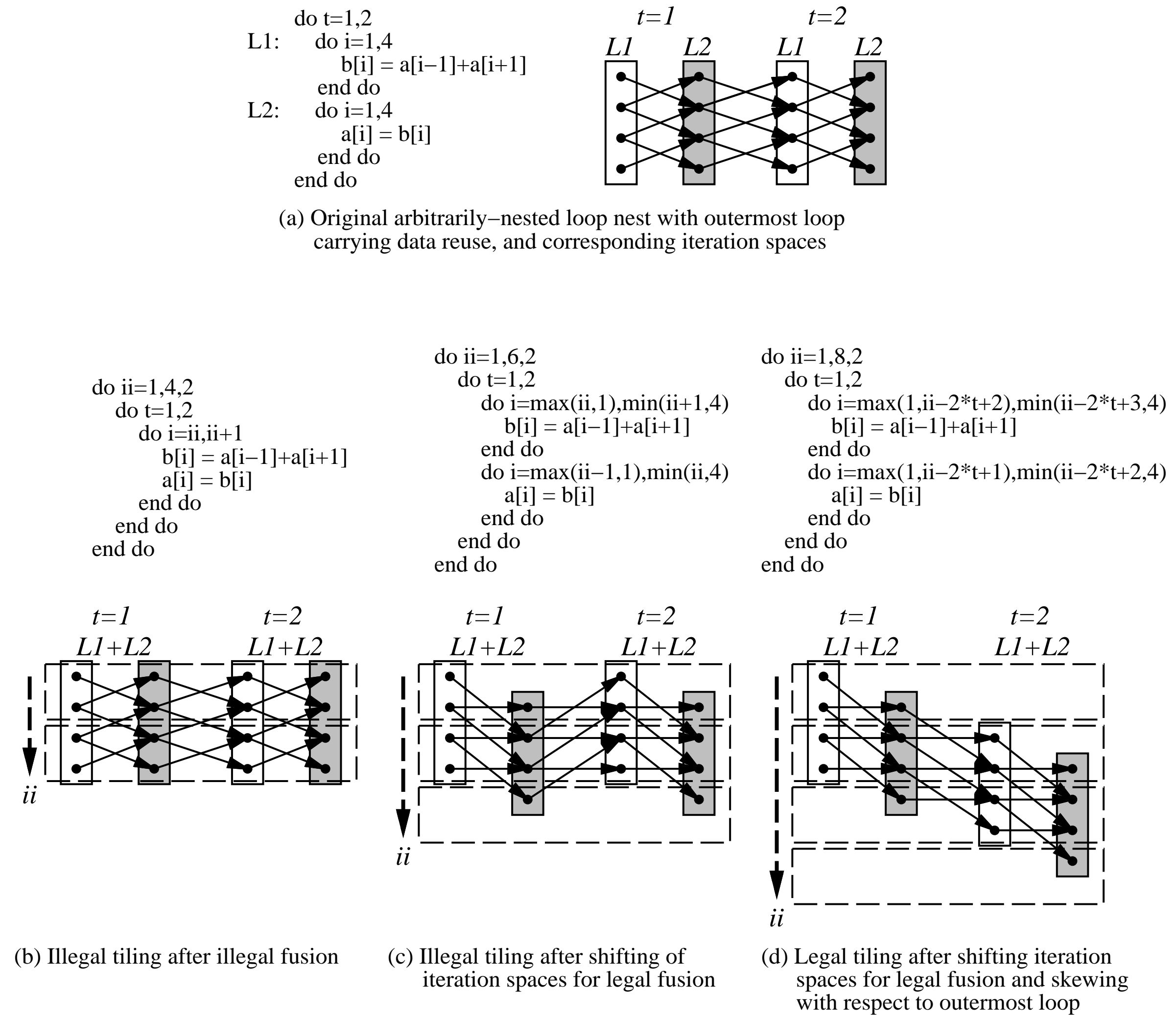
The outermost loop in Figure 5.3(a) carries temporal reuse. The iteration spaces for each of the component loops L_1 and L_2 are illustrated for each iteration t of the outermost loop, along with all dependences. Tiling of inner loops is not possible because there are *two* inner loops at



(a) Three−dimensional iteration space with dependences for original SOR loop nest

(b) Iteration space after skewing with two representative tiles of iterations

Figure 5.2: Graphical representation of skewing and tiling in the iteration space



(a) Original arbitrarily−nested loop nest with outermost loop carrying data reuse, and corresponding iteration spaces

(b) Illegal tiling after illegal fusion

(c) Illegal tiling after shifting of iteration spaces for legal fusion

(d) Legal tiling after shifting iteration spaces for legal fusion and skewing with respect to outermost loop

Figure 5.3: Enabling tiling with the shift-and-peel transformation

the same level. The inner loops must first be fused to enable tiling.

Figure 5.3(b) illustrates direct fusion without regard for dependences, followed by tiling. The dashed boxes are tiles of iterations indexed by iterations of the ii loop. Within each tile, iterations corresponding to $t = 1$ are executed first, followed by iterations corresponding to $t = 2$. However, this transformation is *illegal*. There are lexicographically-negative dependences in the fused loop for the same iteration of the t loop, as well as between different iterations of the t loop. The order in which tiles are executed does not preserve the original semantics.

Now consider applying the shift-and-peel transformation. In this instance, legal fusion is enabled by shifting the iteration space of L_2 by one iteration with respect to L_1, as illustrated

in Figure 5.3(c). However, tiling is still not legal because there are still backward dependences between tiles. Loop skewing must now be applied to enable tiling. In this case, the required skewing factor is 2. The effect of skewing on the iteration spaces is illustrated in Figure 5.3(d). There are no longer any backward dependences between tiles, hence tiling is now legal.

5.1.3 Wavefront Parallelism after Tiling

Enabling tiling with the shift-and-peel transformation and loop skewing transforms the dependences into a form that leads to *wavefront parallelism* in the tiled loop nest. The SOR loop nest in Figure 5.1(a) is used to illustrate the emergence of wavefront parallelism. Consider the dependence distance vector $(t, j, i) = (1, 0, 0)$ for the original loop nest; this vector indicates that the outermost loop carries reuse. The inner loops must be tiled in order to exploit this reuse. If it were possible to tile the SOR loop nest directly, then the distance vector would first be transformed into $(t, jj, j, ii, i) = (1, 0, 0, 0, 0)$ after strip-mining, and then into $(jj, ii, t, j, i) = (0, 0, 1, 0, 0)$ after loop permutation. Hence, the outer loops would be parallelizable because they do not carry dependences.

However, direct tiling is not legal because other dependences require loop skewing in order to produce a fully permutable loop nest. Skewing of the inner loops transforms the original distance vector that reflects the outer loop reuse into $(t, j, i) = (1, 1, 1)$. Now, strip-mining results in $(t, jj, j, ii, i) = (1, B, 1, B, 1)$, and permutation finally produces $(jj, ii, t, j, i) = (B, B, 1, 1, 1)$.[1] Hence, skewing of the inner loops converts a dependence (i.e., reuse) carried by the original outermost loop into a loop-carried dependence in the outermost loop after tiling. Since *all* of the vector components are nonzero, permutation of any other loop into the outermost position also results in a loop-carried dependence.

Similar transformations of the remaining distance vectors for the SOR loop nest introduce additional loop-carried dependences in the outer loops after tiling (although these are redundant in relation to the primary dependence discussed above). Hence, both ii and jj loops in the tiled loop nest of Figure 5.1(d) carry dependences. These loop-carried dependences are represented graphically by the arrows in Figure 5.4. This figure is a two-dimensional representation of the five-dimensional iteration space of the tiled loop nest. Each square corresponds to an iteration

[1] The effect of strip-mining on dependences was discussed in Section 2.4.4, specifically the introduction of the factor B in the transformed dependence vector.

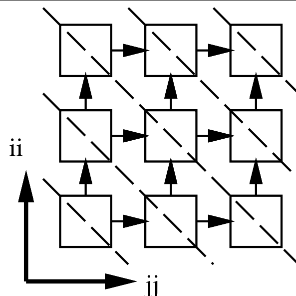


Figure 5.4: Dependences and wavefronts

(jj, ii) from the outer loops, and represents a $B \times B \times T$ tile of iterations from the original j,i, and t loops. Since the two outer loops carry dependences, they are not parallelizable.

However, exploitable parallelism exists, even in the presence of these loop-carried dependences. The parallelism is along the diagonal *wavefronts* shown by the dashed lines in Figure 5.4. Tiles within each wavefront are independent from one another and may be executed in parallel, although the wavefronts must be executed in proper sequence to satisfy the dependences. The existence of wavefront parallelism follows from previous research that asserts that a fully-permutable loop nest of depth m can always be transformed into another loop nest of depth m such that there are at least $m - 1$ parallel (or DOALL) loops [Wol92]. However, in the presence of loop-carried dependences, these parallel loops may be inner loops, and the outermost loop may remain sequential.

5.1.4 Exploiting Wavefront Parallelism: DOALL *vs.* DOACROSS

There are two general approaches for exploiting wavefront parallelism. The first is to apply a *wavefronting transformation* to obtain the inner DOALL loops [Wol92]. This wavefronting transformation corresponds to applying additional loop skewing at the outer loop levels to align independent tiles in each wavefront such that their execution may be expressed in a DOALL loop. For the SOR example, applying additional skewing to the tiled iteration space shown in Figure 5.4 yields the iteration space shown in Figure 5.5. Independent tiles in the skewed space are aligned with the ii loop and may be executed in parallel. The outermost jj loop remains sequential, requiring global synchronization of all processors between successive iterations.

The drawback of the DOALL approach is that processors may not be fully utilized between

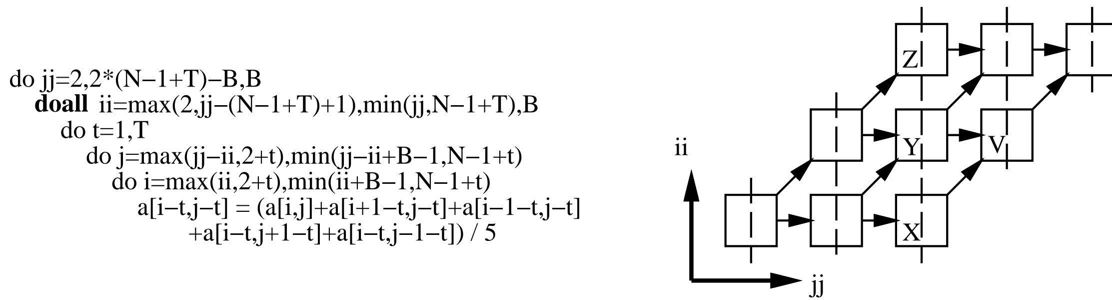


Figure 5.5: Exploiting parallelism with inner DOALL loops

global synchronizations because the number of independent tiles varies in each wavefront. For example, the middle wavefront in Figure 5.5 has three tiles labelled X, Y, and Z. With two processors executing in parallel, both processors are initially busy executing tiles X and Y. However, one processor must remain idle until the remaining tile Z is executed because of the global synchronization required for the DOALL loop.

The alternative approach for exploiting wavefront parallelism is to treat the two outer loops as DOACROSS loops and introduce explicit synchronization between dependent tiles. This approach avoids global synchronization and effectively utilizes idle processors by allowing concurrent execution of tiles in different wavefronts, although local synchronization is now required between tiles. For example, after tiles X and Y in Figure 5.5 have been executed, tiles V and Z may be executed concurrently because the dependences for tile V are satisfied. Since the DOACROSS approach provides the opportunity for improved processor utilization, it is used later in this chapter for scheduling the execution of tiled loop nests.

5.2 Data Reuse in Tiled Loop Nests

5.2.1 Intratile and Intertile Reuse

In this chapter, data reuse in a tiled loop nest is categorized as *intratile* or *intertile* reuse. Intratile reuse results from capturing the reuse from the original outer loop within a single tile. In the tiled loop nest, data referenced in each tile is ideally loaded only once into the cache, then reused from the cache for locality within the same tile. However, when loop skewing is required to enable tiling, the data access patterns in the original loop nest are modified. When

the skewed loop nest is tiled, there is still reuse of data within tiles, but the modified data access patterns also introduce reuse between tiles, i.e., *intertile* reuse.

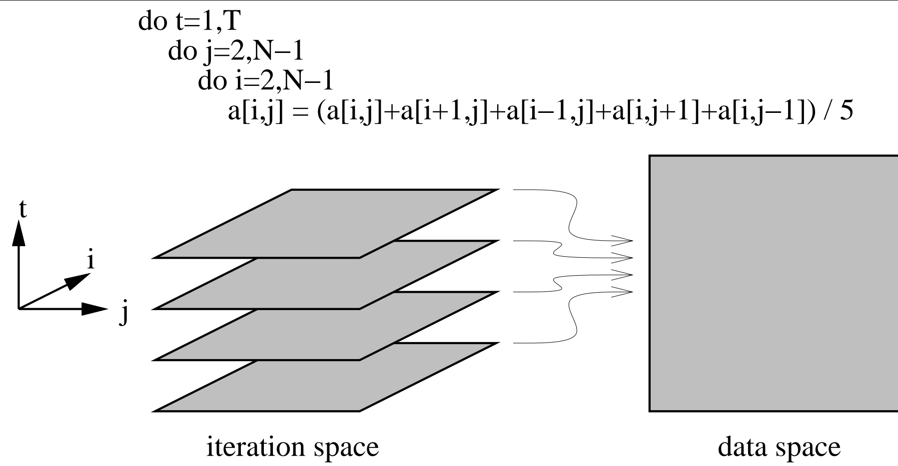
The two categories of reuse are illustrated for the example SOR loop nest in Figure 5.6. The iteration and data spaces for the original SOR loop nest are shown in Figure 5.6(a). With loop skewing and tiling, successive iterations of the original outer loop that are executed within the same tile access overlapping regions of the array, as shown in Figure 5.6(b). This constitutes intratile reuse. However, iterations from adjacent tiles also access overlapping regions in the data space as a result of loop skewing, as shown in Figure 5.6(c), and it is this overlap between tiles that results in intertile reuse.

When executing a tiled loop nest on a multiprocessor, an individual tile is executed to completion by one processor. As a result, intratile reuse is converted to locality on each processor if reused data remains cached during the execution of the tile. When adjacent tiles are executed by the same processor, and data in the overlapping regions for those tiles is retained in the cache between tiles, intertile reuse is converted to intertile locality. That is, data in the overlapping regions is loaded only once into the cache, then reused from the cache not only within the same tile for intratile locality, but also in adjacent tiles. On the other hand, when adjacent tiles are executed by different processors, cache misses are incurred by each processor to load all the data referenced within each tile, including the data in the overlapping regions. In this case, there is no intertile reuse, and the opportunity to convert the reuse into locality is lost.

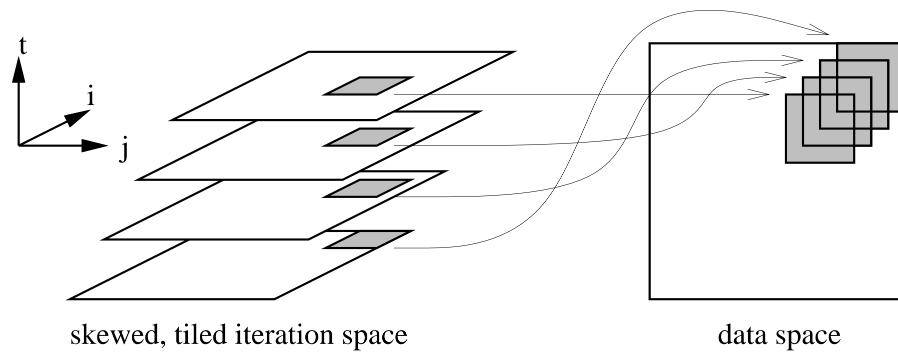
5.2.2 Quantifying the Locality Benefit of Tiling

The sweep ratio in Chapter 3 can quantify the locality benefit of tiling. Let ℓ denote a loop nest with an outermost loop that carries temporal reuse, let $A(\ell)$ denote the set of similarly-sized arrays referenced in the loop nest ℓ, and $A_w(\ell)$ denote the subset of arrays that are modified. Prior to tiling, each iteration of the outermost loop requires a complete memory sweep for each of the arrays in $A(\ell)$, and an additional writeback sweep for each of the arrays in $A_w(\ell)$. Hence, the total number of memory sweeps for the entire loop nest before tiling is given by $s_b = T \cdot (|A(\ell)| + |A_w(\ell)|)$, where T is the number of iterations of the outermost loop.

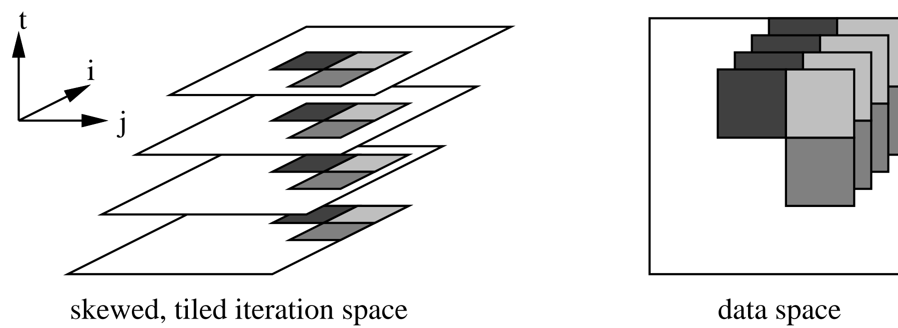
First, consider tiling without loop skewing. Each tile performs all T iterations of the original outermost loop. Cache misses are incurred at the start of each tile to load the required data into



(a) Iteration and data spaces for original SOR loop nest

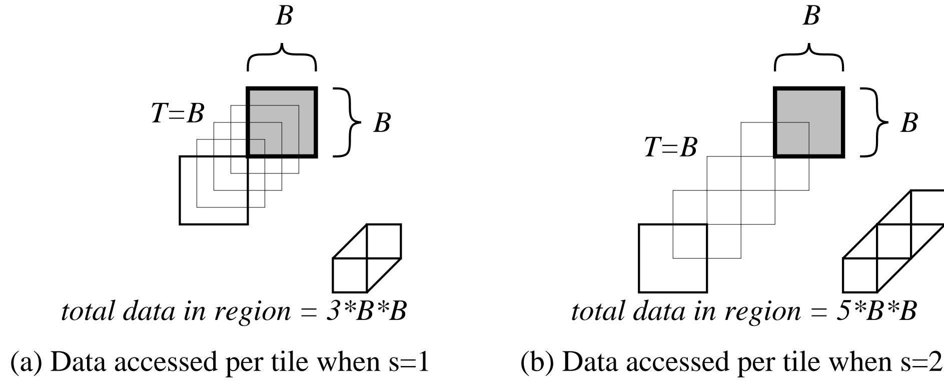


(b) Skewed data access patterns within a tile



(c) Intertile reuse for adjacent tiles

Figure 5.6: Data reuse in a tiled loop nest that requires skewing



(a) Data accessed per tile when s=1 (b) Data accessed per tile when s=2

Figure 5.7: Amount of data accessed per tile with skewing

the cache, then the same data is reused from the cache for the remainder of the tile. In addition, writebacks occur only when modified data in the cache is replaced by new data for the next tile. Consequently, the total number of memory sweeps with tiling is $s_a = |A(\ell)| + |A_w(\ell)|$. The sweep ratio for tiling without skewing is therefore

$$r_{tiling} = \frac{s_b}{s_a} = \frac{T \cdot (|A(\ell)| + |A_w(\ell)|)}{|A(\ell)| + |A_w(\ell)|} = T.$$

Now, consider tiling with loop skewing. Skewing alters data access patterns within a tile; rather than reusing a fixed portion of data, the amount of data accessed per tile is proportional to T. Figure 5.7(a) illustrates this relationship when tiling two inner loops with $T = B$ and a skewing factor of $s = 1$. The number of elements accessed in the region shown in Figure 5.7(a) is $(2 \cdot (T/B) + 1) \cdot (B \cdot B) = 3 \cdot B \cdot B$. Figure 5.7(b) illustrates the region resulting from a larger skewing factor $s = 2$. This region encloses $(2 \cdot (s \cdot T/B) + 1) \cdot (B \cdot B) = 5 \cdot B \cdot B$ elements. Compared with ideal tiling that references only $B \cdot B$ elements per tile, skewing effectively reduces the ideal sweep ratio of T by a factor of $2 \cdot (s \cdot T/B) + 1$. Hence, the sweep ratio for tiling with skewing is given by

$$r_{tiling} = \frac{T}{2 \cdot (s \cdot T/B) + 1}. \tag{5.1}$$

This result assumes that only *intra*tile reuse is exploited; Section 5.4.4.4 will discuss the impact of exploiting intertile reuse.

Finally, consider the combined effect of the shift-and-peel transformation and tiling. As discussed in Section 5.1.2, the shift-and-peel transformation enables tiling by fusing inner

loops. In addition to enabling tiling, fusion reduces the number of memory sweeps by a factor of $r_{fusion} = (\text{sweeps before fusion})/(\text{sweeps after fusion})$. Tiling reduces the number of sweeps by a factor of $r_{tiling} = (\text{sweeps before tiling})/(\text{sweeps after tiling})$. Since fusion is performed first, the number of sweeps after fusion is equal to the number of sweeps before tiling. Hence, the overall sweep ratio $r_{overall}$ is given by

$$r_{overall} = \frac{\text{sweeps before fusion}}{\text{sweeps after tiling}} = \frac{\text{sweeps before fusion}}{\text{sweeps after fusion}} \cdot \frac{\text{sweeps before tiling}}{\text{sweeps after tiling}} = r_{fusion} \cdot r_{tiling}. \tag{5.2}$$

5.2.3 Tile Size, Parallelism, and Locality

The tile size has a significant impact on the performance of a tiled loop nest because it determines both the degree of parallelism and the extent to which locality is enhanced. With wavefront parallelism, a smaller tile size increases the number of wavefronts and, more importantly, increases the number of independent tiles in each wavefront. Hence, the degree of parallelism increases with smaller tile sizes, although the frequency of synchronization also increases.

The tile size also dictates the extent of locality enhancement when loop skewing is required for tiling. The impact of tile size on intratile and intertile locality is illustrated in Figure 5.8. The shaded regions represent the data accessed by adjacent tiles, as in Figure 5.6(c). The overlapping regions correspond to the intersection of the data accessed by different tiles. For a given number of iterations in the original outer loop, the amount of data in the overlapping regions is relatively small compared to the total amount of data accessed by the tile when the tile size is large. Consequently, a large tile size enhances intratile locality and diminishes the impact of intertile locality. In contrast, for the same number of iterations and a small tile size, the amount of data in the overlapping regions is a much larger fraction of the total amount of data accessed by the tile. Hence, a small tile size increases the importance of intertile locality.

5.3 Related Work

5.3.1 Tiling

An extensive formal treatment of tiling is given by Wolf [Wol92], building on the work of Porterfield [Por89], Irigoin and Triolet [IT88] and Abu-Sufah *et al.* [ASKL81]. However,

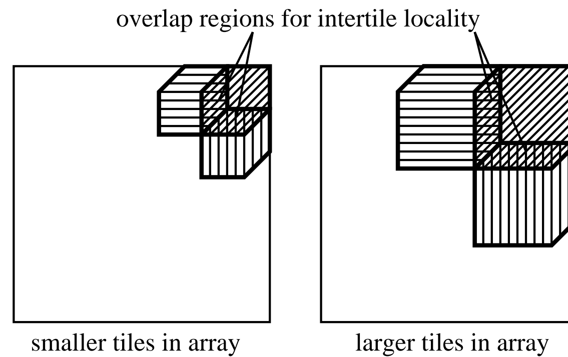


Figure 5.8: Impact of tile size on locality

there are two shortcomings in the work of Wolf. First, Wolf does not study the effects of loop skewing on data reuse, although his theory incorporates skewing. Hence, he does not distinguish between intratile and intertile locality. Second, Wolf's experiences with tiling are limited to small-scale multiprocessors with uniform memory access. Satisfactory performance is achieved with relatively large tiles that exploit intratile reuse for locality with a modest degree of parallelism. However, large-scale multiprocessors require the use of small tiles to provide sufficient parallelism on a large number of processors. Furthermore, when loop skewing is required to enable tiling, intertile locality becomes more important with small tiles.

5.3.2 Loop Scheduling

There exists a large body of work dealing with scheduling of parallel, or DOALL, loops on shared-memory multiprocessors. Many scheduling strategies have been proposed to strike a balance between load balance and scheduling overhead. Static scheduling [BGS94] minimizes overhead, but may not provide sufficient load balance. Dynamic techniques, such as self-scheduling [BGS94], guided self-scheduling [PK87], and factoring [HSF92], seek to improve load balance at the expense of increased overhead. Some scheduling strategies also consider memory locality for nonuniform memory access by attempting to distribute loop iterations in a manner that matches the distribution of the data accessed by those iterations. Examples include affinity-based scheduling [ML94] and locality-based dynamic scheduling [LTSS93].

There are two problems when considering the use of existing scheduling strategies for exploiting wavefront parallelism in tiled loop nests. First, the strategies cited above address

individual DOALL loops in which there no restrictions on the manner in which iterations are distributed and executed among multiple processors. Greater care is needed when scheduling multiple DOACROSS loops with explicit synchronization in order to satisfy loop-carried dependences; iterations must be executed in lexicographical order on each processor, otherwise deadlock may occur. Second, the scheduling strategies cited above do not address the issue of exploiting intertile reuse for cache locality. Since the importance of intertile locality increases when tiling loop nests for large-scale multiprocessors, new scheduling approaches are required.

5.3.3 Scheduling Vectors

In the presence of wavefront parallelism in a loop nest, the loop-carried dependences define the scheduling vector [DR94] that determines the sequence in which the wavefronts must be executed. By definition, the scheduling vector is orthogonal to the wavefronts.

Hodzic and Shang [HS96] present an analytical method for deriving the optimal granularity (i.e., tile size) for tiling loop nests with loop-carried dependences that require interprocessor communication on message-passing multiprocessors. Their derivation assumes that the startup cost for communication is high and that transmission time after startup is negligible, hence they seek the optimal tradeoff between the frequency of communication and the degree of parallelism to minimize execution time. They conclude that the optimal scheduling vector that minimizes execution time does not vary with the optimal granularity.

In contrast, the scheduling of tiled loop nests considered in this chapter addresses shared-memory multiprocessors in which cache locality, rather than communication startup cost, has the greatest impact on performance. As a result, the optimal tradeoff to minimize execution time is between the degree of locality enhancement and the degree of parallelism. In particular, consideration must be given to intertile locality when tile sizes are reduced to increase parallelism. The next section will show that the optimal tradeoff betwen locality and parallelism to minimize execution time may in fact require a *sub*obtimal scheduling vector.

5.4 Scheduling Strategies for Wavefront Parallelism

This section discusses three scheduling strategies—namely *dynamic self-scheduling*, *static cyclic scheduling* and *static block scheduling*—for exploiting wavefront parallelism in tiled

loop nests when the outer loops are treated as DOACROSS loops. The first strategy, dynamic self-scheduling, is a straightforward adaptation of the existing technique for DOALL loops to DOACROSS loops by controlling the order in which loop iterations are assigned to processors. The other two techniques are adaptations of static scheduling for DOACROSS loops with modifications to the manner in which iterations are distributed and ordered among processors. The strategies are evaluated on the bases of runtime overhead, synchronization requirements, degree and granularity of parallelism, and locality enhancement.

5.4.1 Dynamic Self-scheduling

In normal dynamic self-scheduling of DOALL loops, processors obtain iterations in some arbitrary order from a shared work pool. Dynamic self-scheduling is most effective in improving load balance when there is high variability in the amount of computation within the independent iterations assigned to each processor. Since there are no dependences between iterations in a DOALL loop, there is no need for synchronization between iterations. For the DOACROSS loops in tiled loop nests, the iterations represent individual tiles, and there is explicit synchronization to enforce dependences between tiles in different wavefronts. Dynamic scheduling for tile execution must be modified such that idle processors obtain tiles in an order that respects these dependences. Prior to executing a tile, interprocessor synchronization is required to ensure that tiles in the preceding wavefront have been executed. Dynamic scheduling also balances the workload for the variability in the degree of parallelism in successive wavefronts.

This form of dynamic self-scheduling is adequate for exploiting wavefront parallelism in tiled loop nests for small-scale shared-memory multiprocessors. With a limited number of processors, a large tile size generally provides an adequate degree of parallelism. Consequently, intratile locality is enhanced because a large tile size captures most of the reuse from the original loop nest within a single tile, and intertile locality has little impact on performance.

However, with little or no variability in the amount of computation per tile, dynamic self-scheduling is not an appropriate strategy for large-scale shared-memory multiprocessors for two reasons. First, a large number of processors requires a relatively small tile size for sufficient parallelism. A small tile size reduces intratile locality and places greater importance on intertile locality. Dynamic self-scheduling is not likely to enhance intertile locality since

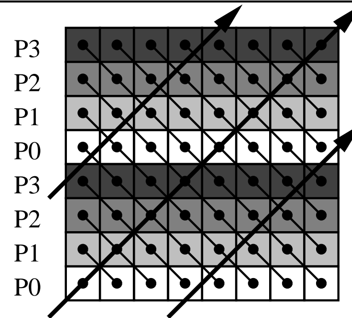


Figure 5.9: Static cyclic scheduling of tiles

tiles are assigned arbitrarily to idle processors. The second reason is that cache misses that result from the reduced intertile locality with small tile sizes are likely to be incurred for remote, rather than local, memory due to the arbitrary assignment of tiles to processors. The performance degradation resulting from these misses may be significant.

5.4.2 Static Cyclic Scheduling

In normal static scheduling for DOALL loops, the assignment of iterations to processors is determined in advance and remains fixed. Since DOALL loop iterations are independent, no synchronization is required. To exploit wavefront parallelism, static cyclic scheduling for the DOACROSS loops assigns rows of horizontally-adjacent tiles to the same processor, as shown in Figure 5.9. In this manner, intertile reuse within rows of tiles is exploited by one processor to enhance intertile locality. The cyclic mapping of rows of tiles to processors distributes the workload in each wavefront evenly among processors to fully exploit the available parallelism. However, explicit synchronization between dependent tiles is still required.

Static cyclic scheduling improves over dynamic self-scheduling in three ways. First, cyclic scheduling enhances intertile locality for horizontally-adjacent tiles by statically assigning them to the same processor, whereas dynamic self-scheduling does not necessarily exploit any intertile reuse due to the arbitrary assignment of tiles. Second, interprocessor synchronization to enforce loop-carried dependences is required only for vertically-adjacent tiles, since horizontally-adjacent tiles are executed in the correct order by the same processor. Third, the scheduling overhead is reduced since the assignment of tiles to processors is determined statically. However, cyclic scheduling still requires synchronization for each tile to enforce

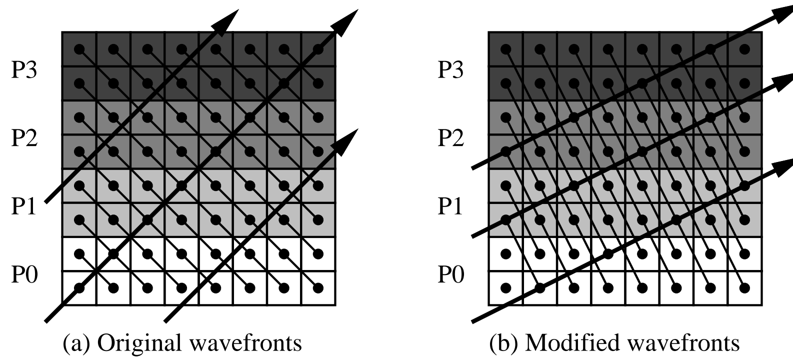


Figure 5.10: Static block scheduling of tiles

dependences, and not all of the intertile reuse is exploited.

5.4.3 Static Block Scheduling

Static block scheduling for the DOACROSS loop iterations in a tiled loop nest assigns contiguous blocks of tiles to the same processor, as shown in Figure 5.10. In this manner, all of the intertile reuse within a block of horizontally- and vertically-adjacent tiles is exploited by one processor to enhance intertile locality. Since the loops are DOACROSS, the tiles must be executed in an order that respects the dependences. However, the available parallelism in each wavefront is not exploited efficiently for the original wavefronts shown in Figure 5.10(a) because a portion of the processors is left idle for the few initial and few final wavefronts. The block assignment of tiles to processors precludes the use of additional processors even when there are tiles that can be executed. Consequently, it takes longer for all processors to become active, and it takes longer for execution to complete.

Block scheduling requires the use of modified wavefronts as shown in Figure 5.10(b) to provide greater parallelism. This involves rotating wavefronts such that the number of independent tiles in the largest wavefront is exactly equal to the number of processors. This rotation corresponds to the selection of a different *scheduling vector*. The scheduling vector is $(1, 1)$ for the original wavefronts in Figure 5.10(a); in fact, this is the *optimal* scheduling vector for dynamic and cyclic scheduling. The scheduling vector for the modified wavefronts in Figure 5.10(b) is given by $(\lfloor (N+T)/(B \cdot P) \rfloor, 1)$, where $N+T$ is the number of iterations (with skewing), B is the tile size, and P is the number of processors. The new scheduling

Table 5.1: Comparison of scheduling strategies for tiling

	Dynamic	Cyclic	Block
runtime overhead	yes	no	no
synch. req'd.	horizontal/vertical *tiles*	vertical *tiles*	vertical *processors*
#counters	$\left\lceil \frac{N+T}{B} \right\rceil$	$\left\lceil \frac{N+T}{B} \right\rceil$	P
completion time	$\left\lceil \frac{N+T}{B} \right\rceil^2 / P + P - 1$	$\left\lceil \frac{N+T}{B} \right\rceil^2 / P + P - 1$	$\left(\left\lceil \frac{N+T}{B} \right\rceil + P - 1 \right) \cdot \left\lceil \frac{N+T}{B \cdot P} \right\rceil$
intertile locality	none	horizontal tiles	horizontal/vertical tiles

vector preserves the loop-carried dependences in block scheduling, but reduces the time before all processors become active in parallel execution and reduces the completion time.

Static block scheduling improves over both dynamic and cyclic scheduling in two ways. First, block scheduling exploits all intertile reuse, except at block boundaries. Second, interprocessor synchronization to enforce loop-carried dependences is required only for tiles on block boundaries; no synchronization is required for adjacent interior tiles, since they are executed in the correct order by the same processor. Similar to cyclic scheduling, the scheduling overhead is also reduced since the assignment of tiles to processors is determined statically.

5.4.4 Comparison of Scheduling Strategies

The scheduling strategies are compared on the bases of *runtime overhead*, *synchronization*, *parallelism*, and *intertile locality enhancement*. These features are summarized in Table 5.1.

5.4.4.1 Runtime Overhead for Scheduling

Dynamic self-scheduling incurs runtime overhead in order to assign tiles to processors as they become idle. The overhead has two components. The first is maintaining the set of iterations to be assigned. Since the wavefronts governing the order of tile assignment have a regular pattern, only two counters are required for ths purpose; one counter identifies the current wavefront, and the second identifies the last tile assigned in that wavefront. The cost of updating the counters is low in comparison to the computation in each tile.

The second component of runtime overhead for dynamic self-scheduling arises from processors competing for access to the counters governing tile assignment. For correctness, the

counters must updated atomically, hence they must be protected with an appropriate synchronization construct such as a lock. If more than one idle processor seeks to obtain a new tile at the same time, contention for the lock and counters contributes overhead.

Static cyclic and static block scheduling incur no runtime overhead for scheduling since the assignment of tiles to processors is determined in advance. The only overhead is due to synchronization to satisfy dependences, which is discussed below.

5.4.4.2 Synchronization Requirements

DOACROSS loops require explicit synchronization between dependent iterations; Section 2.3.1 discussed the use of semaphores for this purpose. In tiled loop nests, rather than using one semaphore for each individual tile, it is possible to employ a counter for each row of horizontally-adjacent tiles. The counter is incremented as each tile in that row is completed, and hence tracks the progress of the wavefronts through that row. The dependences between tiles are such that only one tile in any given row may be executed at any time. Hence, the corresponding counter will never be updated by more than one processor at any time and no locking is required.

Dynamic self-scheduling requires synchronization for both horizontally- and vertically-adjacent tiles. In other words, prior to executing a tile in a given row, a processor must read the counters for the same row and an adjacent row to verify that it is safe to execute the tile. Thus, a processor must wait for both counter values to reach a safe value if it cannot begin executing the tile immediately. Static cyclic scheduling requires interprocessor synchronization only for vertically-adjacent tiles, hence only one counter for the adjacent row needs to be read.

Finally, static block scheduling requires interprocessor synchronization only for vertically-adjacent tiles on block boundaries. As a result, the number of synchronization counters required is equal to the number of processors, rather than the number of rows. The counter between two blocks is checked only before executing tiles at the block boundary.

5.4.4.3 Parallelism and Theoretical Completion Time

In ideal circumstances, greater parallelism implies reduced execution time, hence the degree of parallelism for the different scheduling strategies may be evaluated by determining the

theoretical completion time[2] for a given number of processors P. A unit time step is defined as the theoretical execution time for one tile (i.e., neglecting the impact of synchronization and locality); the completion time is expressed in these units. For simplicity, it is assumed that there is no variance in the amount of computation per tile.[3]

The following analysis assumes that there are N iterations in each of the tiled inner loops of the original loop nest, and that there are T iterations in the outer loop that carries reuse. Skewing the inner loops by one iteration, then tiling the inner loops by B, yields a tiled iteration space with $n_t = \lceil (N+T)/B \rceil$ tiles in each of the new outer loops. This value of n_t also represents the number of synchronization counters required for dynamic and cyclic scheduling, and appears in Table 5.1. The final assumption is that $P \leq n_t$, i.e., there are more tiles in the largest wavefront than there are processors in order to ensure high processor utilization.

For dynamic self-scheduling, idle processors are assigned new tiles arbitrarily in an order governed by the wavefronts, hence the processors are fully utilized with maximal exploitation of the available parallelism, except when dependences for a tile force a processor to wait. In the absence of scheduling and synchronization overhead, the theoretical completion time is determined only by the ordering requirements for the tiles. Since there are P processors, the initial P wavefronts shown in Figure 5.11 contain $P \cdot (P+1)/2$ tiles and require exactly P time units to execute in parallel, since there are no more than P independent tiles per wavefront. The same argument applies for the final P wavefronts shown in Figure 5.11. The number of tiles in the remaining interior wavefronts shown in Figure 5.11 is given by

$$n_t^2 - 2 \cdot \frac{P \cdot (P+1)}{2}.$$

Since there are more tiles per wavefront than processors in the interior wavefronts, tiles in different wavefronts may be executed concurrently, hence the parallel execution time for the interior is given simply by dividing the number of tiles by the number of processors,

$$\frac{n_t^2}{P} - (P+1).$$

Finally, the completion time for dynamic scheduling is given by the sum of the execution times

[2] Note that theoretical completion time is distinct from ideal schedule length [DR94] because it is determined for a finite number of processors.

[3] Variances may exist between tiles from the boundaries of the iteration space and interior tiles; these variances are not significant when the total number of tiles is large.

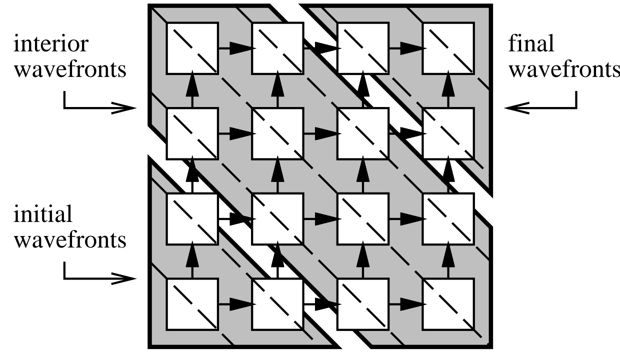


Figure 5.11: Wavefronts for dynamic and cyclic scheduling ($n_t = 4$,$P = 2$)

for the initial P wavefronts, the final P wavefronts, and the interior wavefronts,

$$T_{dyn} = \frac{n_t^2}{P} - (P+1) + 2 \cdot P = \frac{n_t^2}{P} + P - 1.$$

In static cyclic scheduling, the independent tiles in each wavefront are evenly distributed among P processors (or fewer if the number of independent tiles per wavefront is less than P). In the absence of overhead, the theoretical completion time is determined only by the ordering requirements for the tiles. The cyclic distribution of independent tiles provides the same degree of processor utilization as dynamic scheduling, hence the execution times for the initial, final, and interior wavefronts are the same as for dynamic scheduling. As a result, the completion time for static cyclic scheduling is the same as for dynamic scheduling:

$$T_{cyc} = \frac{n_t^2}{P} - (P+1) + 2 \cdot P = \frac{n_t^2}{P} + P - 1.$$

In static block scheduling, the modified wavefronts shown in Figure 5.12 are different than the wavefronts for dynamic and cyclic scheduling. Assuming that n_t is evenly divisible by P, the number of initial wavefronts with fewer than P independent tiles is given by

$$\frac{n_t}{P} \cdot (P-1).$$

Since the number of independent tiles in each initial wavefront is less than P, the execution time is equal to the number of initial wavefronts, and a similar argument applies for the final wavefronts as well. The number of interior wavefronts is given by

$$(n_t - (P-1)) \cdot \frac{n_t}{P}.$$

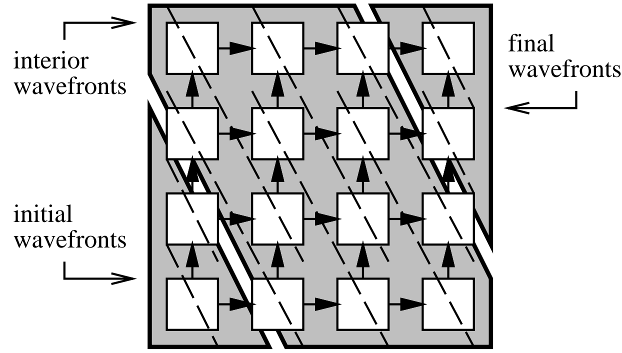


Figure 5.12: Wavefronts for block scheduling ($n_t = 4$,$P = 2$)

Since the number of independent tiles on each of the interior wavefronts is exactly P, the execution time for the interior wavefronts is exactly equal to the number of interior wavefronts. The completion time for block scheduling on P processors is therefore given by the sum of times for the initial, interior, and final wavefronts, i.e.,

$$T_{blk} = 2 \cdot \frac{n_t}{P} \cdot (P-1) + (n_t - (P-1)) \cdot \frac{n_t}{P} = (n_t + P - 1) \cdot \frac{n_t}{P}.$$

Since $T_{dyn} = T_{cyc}$, it suffices to compare T_{cyc} with T_{blk}. To make this comparison, let

$$R = \frac{T_{blk}}{T_{cyc}} = \frac{(n_t + P - 1) \cdot \frac{n_t}{P}}{\frac{n_t^2}{P} + P - 1} = \frac{n_t^2 + n_t \cdot P - n_t}{n_t^2 + P^2 - P},$$

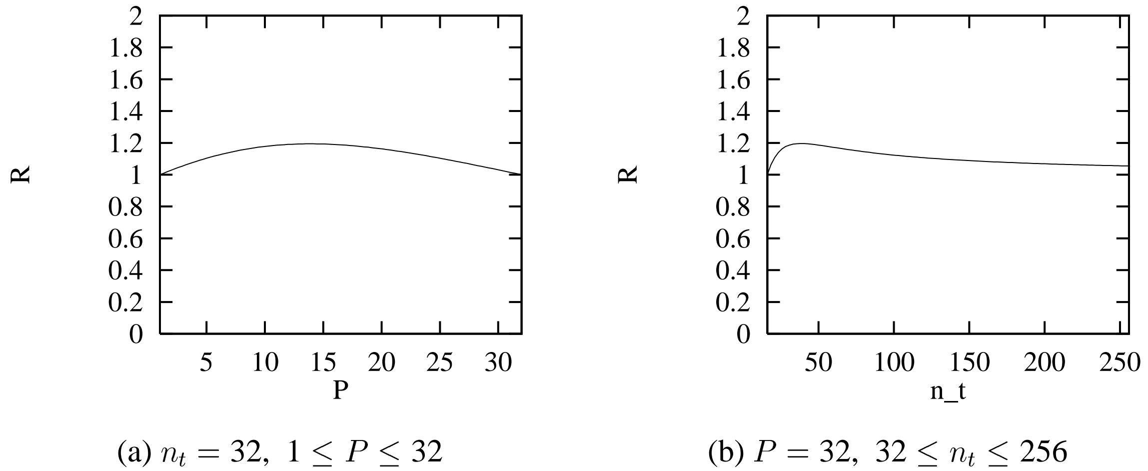
i.e., the ratio of completion times. Figure 5.13(a) illustrates the variation of R for $n_t = 32$ and $1 \leq P \leq 32$. Since $R \geq 1$, this indicates that $T_{blk} \geq T_{cyc}$, i.e., block scheduling does not provide as much parallelism as cyclic scheduling, even with the modified wavefronts. Figure 5.13(b) illustrates the variation of R for $P = 32$ and $32 \leq n_t \leq 256$. Once again, $R \geq 1$. There is clearly a maximum for R, and it may be shown that

$$P_{max}|_{\frac{\partial R}{\partial P}=0} = 1 - n_t + \sqrt{2 \cdot n_t^2 - n_t},$$

hence,

$$R_{max}|_{\frac{\partial R}{\partial P}=0} = \frac{1}{\frac{1}{n_t} - 2 + 2\sqrt{2 - \frac{1}{n_t}}}.$$

For $n_t = 32$, $R_{max} = 1.19$ (which agrees with Figure 5.13), indicating that at best, cyclic scheduling is 19% faster than block scheduling. However, this large discrepancy is easily



(a) $n_t = 32,\ 1 \le P \le 32$ (b) $P = 32,\ 32 \le n_t \le 256$

Figure 5.13: Variation of completion time ratio $R = T_{blk}/T_{cyc}$

avoided by choosing smaller tile sizes to increase the degree of parallelism for block scheduling. This corresponds to increasing n_t, hence the ratio approaches 1 again, as in Figure 5.13(b).

Completion times may also be used to establish a criterion for sufficient parallelism when selecting tile sizes. The completion times are functions of P, hence the times at $P = 1$ represent sequential execution. For example, $T_{dyn}(P = 1) = n_t^2/1 + 1 - 1 = n_t^2$. It is therefore possible to express the speedup using P processors over sequential execution as

$$S_{dyn} = \frac{n_t^2}{\dfrac{n_t^2}{P} + P - 1}$$

and the parallel efficiency as

$$E_{dyn} = \frac{S_{dyn}}{P} = \frac{n_t^2}{n_t^2 + P^2 - P} = \frac{1}{1 + \dfrac{P^2 - P}{n_t^2}},$$

assuming no variance in the amount of computation per tile and no overhead.

Since $0 < E_{dyn} < 1$, it is possible to specify $0 < e_{min} < 1$ as the minimum desired parallel efficiency. It is therefore possible to determine, for a given number of processors P, the requirements for the tile size to produced the desired efficiency. Hence,

$$e_{min} \le \frac{1}{1 + \dfrac{P^2 - P}{n_t^2}}$$

that after substitution for n_t may be simplified to

$$B \leq \frac{N+T}{\sqrt{(P^2-P)\cdot\frac{e_{min}}{1-e_{min}}}}.$$

For instance, if a target of $e_{min} = 0.75$ is set for a problem where $N+T = 1024$ and $P = 32$ (i.e., minimum speedup of $0.75 \cdot 32 = 24$), then

$$B \leq \frac{1024}{\sqrt{(1024-32)\cdot\frac{0.75}{0.25}}},$$

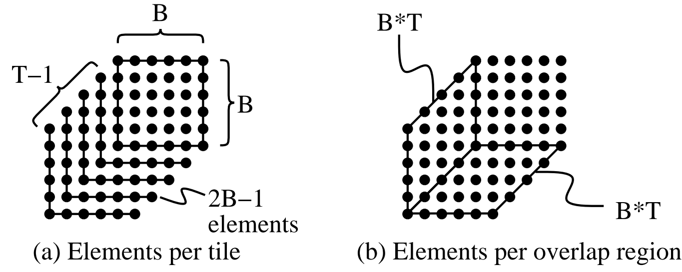
or $B \leq 18$. The smallest possible value of B is 1, which would yield an efficiency of

$$\frac{1}{1+\frac{P^2-P}{(N+T)^2}} = \frac{1}{1+\frac{1024-32}{1024^2}} = 0.999,$$

but achieving such high efficiency is unlikely in practice. The overhead from synchronization would diminish the achieved level of efficiency.

5.4.4.4 Locality Enhancement

The extent of intertile locality enhancement for each scheduling strategy is shown in Table 5.1. The importance of enhancing intertile locality can be demonstrated by estimating the total latency for cache hits and misses that occur during the execution of a single tile. The following estimates are relative to one array in a skewed, tiled loop nest. For a tile size of $B \times B$, and T iterations in the original outer loop of the loop nest being tiled, the number of accesses to the cache for an array within each tile is given by B^2T. Each access to the cache has a latency of C clock cycles. Some fraction of these references miss in the cache and incur a memory latency M. For dynamic self-scheduling, there is no intertile locality, and in the worst case, misses are incurred for all data elements accessed *for the first time* within the tile. The number of such elements is given by $B^2 + (2B-1)(T-1)$, as shown in Figure 5.14(a). This number must then be divided by L, the cache line size, to arrive at an estimate for the number of cache misses. The latency in clock cycles for memory accesses is then given by multiplying by the cache miss penalty M. Finally, the total latency, including ache accesses is given by $B^2TC + (B^2 + (2B-1)(T-1))(M/L)$. To measure the extent of locality enhancement for



(a) Elements per tile (b) Elements per overlap region

Figure 5.14: Number of data elements within a tile

different values of B and T, it is useful to express the fraction f of the total memory access latency per tile that is due to cache misses, which is given by

$$f_{dyn} = \frac{(B^2 + (2B-1)(T-1))(M/L)}{B^2TC + (B^2 + (2B-1)(T-1))(M/L)}.$$

A similar derivation can be made for static cyclic scheduling and static block scheduling. Because there is intertile locality for adjacent tiles, fewer misses are incurred per tile. The reduction in the number of misses is determined by the number of elements in one or both of the overlap regions shown in Figure 5.14(b). Once again, the fraction of the latency due to misses can be determined. Hence,

$$f_{cyc} = \frac{(B^2 + BT - 2B - T + 1)(M/L)}{B^2TC + (B^2 + BT - 2B - T + 1)(M/L)},$$

and

$$f_{blk} = \frac{(B^2 - 2B + 1)(M/L)}{B^2TC + (B^2 - 2B + 1)(M/L)}.$$

Note that for block scheduling, each tile incurs cache misses *only* for the square region of $B \cdot B$ elements in Figure 5.14(a); the remaining data accesses in the tile are satisfied by the cache. *Hence, block scheduling incurs the same number of cache misses as tiling without skewing to result in the ideal sweep ratio of* T.

Figure 5.15 plots the fraction f for different tile sizes B and different values of T. The cache line size is $L = 4$ elements, the cache access latency is $C = 1$ clock cycle, and the cache miss latency is $M = 50$ clock cycles. As T increases, f decreases for all three strategies because reuse carried by the original outer loop is captured within the tile through intratile locality. However, f decreases far more rapidly for block scheduling. This is because block

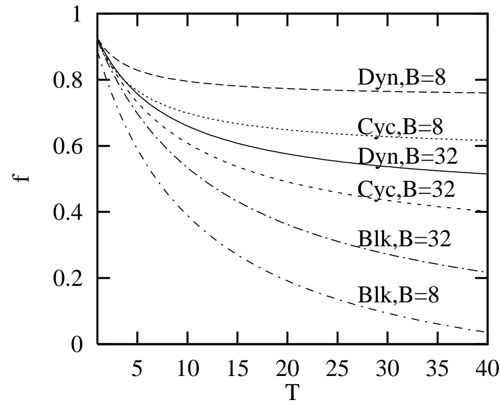


Figure 5.15: Fraction of miss latency per tile

scheduling benefits from enhancing *intertile* locality by reducing the number of cache misses by an amount proportional to the overlap regions in Figure 5.14(b). Furthermore, for a given value of T, f is further reduced with a smaller tile size for block scheduling because intertile locality is more critical when the tile size is small (see Figure 5.8). In contrast, for a given value of T, f *increases* when the tile size is reduced for both dynamic and cyclic scheduling. This is because dynamic and cyclic scheduling do not enhance intertile locality to the same extent for small tile sizes as block scheduling.

In conclusion, all of the scheduling strategies provide sufficient parallelism with small tile sizes, but small tiles require exploiting intertile reuse for locality. Dynamic scheduling does not exploit intertile reuse. Cyclic scheduling exploits some intertile reuse and provides the same degree of parallelism for a given tile size as dynamic scheduling. Hence, cyclic scheduling should perform better than dynamic scheduling. Block scheduling exploits all intertile reuse, but with less parallelism than either dynamic or cyclic scheduling for a given tile size. However, the benefit of enhancing locality may outweigh the loss of parallelism and provide the best performance. The relative performance of the three strategies for small tile sizes on a large number of processors depends on the tradeoff between parallelism and locality.

Chapter 6

Cache Partitioning to Eliminate Cache Conflicts

This chapter proposes a technique called cache partitioning to eliminate cache conflicts between data from different arrays in a loop nest, especially after applying a locality-enhancing transformation. Cache conflicts cause data to be displaced from the cache, and subsequent reuse of displaced data incurs unnecessary cache misses to reload the data into the cache. Conflicts are particularly undesirable when transformations such as fusion and tiling are used because the failure to retain reused data in the cache diminishes the effectiveness of these transformations.

This chapter is organized as follows. First, a discussion of cache conflicts is provided along with related work in order to motivate conflict avoidance. The proposed cache partitioning technique is then described in detail.

6.1 Cache Conflicts

This section provides the motivation for cache conflict avoidance by discussing cache organizations, classifying cache conflicts, and discussing how data access patterns in loop nests lead to cache conflicts. Related work on cache conflict avoidance is then assessed.

6.1.1 Cache Organization and Indexing Methods

Contemporary processors use either a single-level or multilevel cache organization [PH96, CHK+96, MWV92, Yea96], as shown in Figure 6.1. In either case, the goal is to reduce the number of main memory accesses because they incur the largest latency. *Hence, it is imperative to maximize locality by avoiding cache conflicts in the level of the cache closest to main memory.*

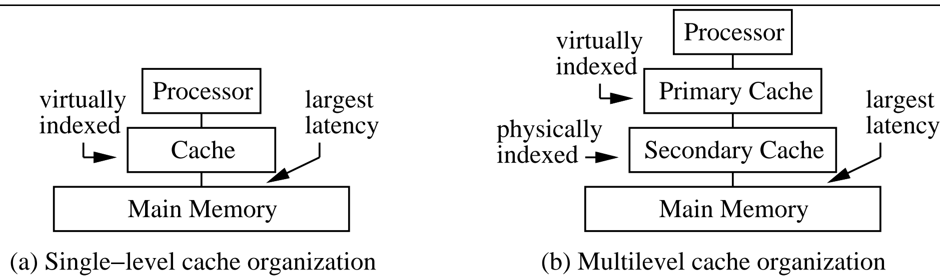


Figure 6.1: Cache organizations

As shown in Figure 6.1(a), a single-level cache normally uses virtual indexing, i.e., the virtual address determines the cache location for each memory reference [PH96]. This approach improves performance by allowing the physical address translation to proceed in parallel with the cache access. Virtual indexing is also used in the primary cache of a multilevel cache, as shown in Figure 6.1(b). However, the secondary cache may use physical indexing because the physical address translation is complete by the time that a miss is detected in the primary cache.

The indexing method determines the mapping of data from memory into the cache. The occurrence of cache conflicts is therefore determined by the indexing method. *Hence, cache conflict avoidance requires knowledge of the indexing method*. Fortunately, the indexing in typical caches uses an easily-computed function of address bits [PH96].

6.1.2 Cache Conflicts for Arrays in Loops

There are two types of cache conflicts for array data when executing loops [LRW91]. *Self-conflicts* occur between elements from the *same* array. For example, in the loop nest shown in Figure 6.2(a), the elements $a[i, j]$ and $a[i, j-1]$ conflict with each other because they map to the same location in the cache. In contrast, *cross-conflicts* occur between elements from *different* arrays. For example, in the loop nest shown in Figure 6.2(b), the elements $a[i, j]$ and $b[i, j]$ conflict with each other in the cache.

The likelihood of self-conflicts depends on the separation between elements with respect to the cache size. For example, assume that array a in Figure 6.2(a) has dimensions of 1024×1024. Hence, elements $a[i, j]$ and $a[i, j-1]$ are separated by 1024 elements in memory. Current caches are normally much larger than 1024 elements; for example, 1-Mbyte caches are now

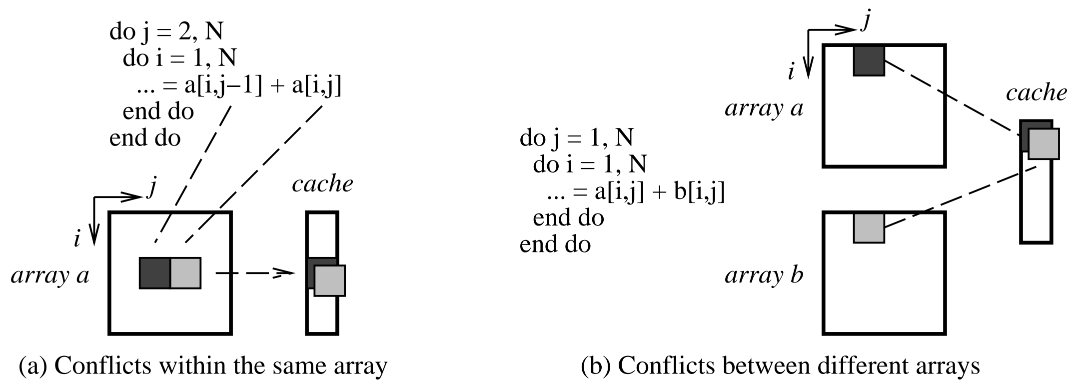


Figure 6.2: Cache conflicts for arrays in loops

commonplace [CHK+96, Yea96]. If each array element is 8 bytes, a 1-Mbyte cache can hold *128 contiguous columns of 1024 elements* from the same array without conflicting. Hence, self-conflicts are unlikely to occur for typical array and cache sizes.

On the other hand, the likelihood of cross-conflicts depends on the separation between elements from *different* arrays. For example, assume that both arrays a and b in Figure 6.2(b) have dimensions of 1024 $\times$ 1024. If the two arrays are allocated contiguously in memory, elements $a[i, j]$ and $b[i, j]$ are separated by a distance of $1024 \cdot 1024 = 1,048,576$ elements. Since this distance may well exceed current cache sizes and allow the two elements to map to the same cache location, cross-conflicts are more likely to occur than self-conflicts.

More representative loop bodies include array references of the form $a[i \pm c_1, j \pm c_2]$ and $b[i \pm c_1, j \pm c_2]$, where c_1, c_2 are small integer constants. If $a[i, j - c_2]$ and $a[i, j + c_2]$ appear in a loop nest with j as the outer loop index, then many columns of array a must remain cached for locality. As a result, the potential for cross-conflicts with other arrays increases. Even if elements $a[i, j]$ and $b[i, j]$ do not conflict, a conflict between $a[i, j + c_2]$ and $b[i, j]$ is still undesirable. *Consequently, this chapter is concerned with avoiding cross-conflicts.*

6.1.3 Data Access Patterns and Cache Conflicts

Loop nests sweep through multidimensional arrays, and array subscript expressions dictate the *data access patterns* for these arrays in memory. These data access patterns are characterized

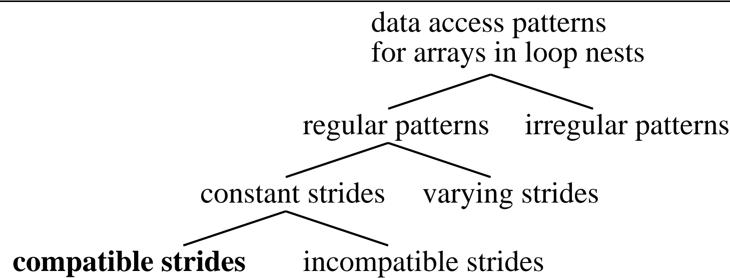


Figure 6.3: Taxonomy of data access patterns for arrays in a loop nest

by *direction* and *stride*. The direction of access is either negative or positive and indicates whether data is accessed in order of increasing or decreasing addresses in memory during the execution of a loop nest. Stride indicates the distance between successive memory addresses generated by a given array reference during the execution of the loop nest.

Figure 6.3 proposes a taxonomy that *collectively* describes the data access patterns for different arrays in a loop nest. Data access patterns may either be regular or irregular in nature. Regular access patterns are further categorized as having constant or varying strides. This distinction is significant because the majority of array references in representative loop nests generate regular data accesses with constant stride, with 1 being the most common stride value [CMT94, MT96]. A constant stride of 1 is referred to as *unit stride*.

In the taxonomy of Figure 6.3, constant-strided data access patterns for different arrays are further classified as having compatible or incompatible strides. Compatible array references have the same constant stride and direction, whereas incompatible references have differing stride and direction. *This distinction is significant because the frequency of cross-conflicts is determined by whether or not the access patterns for different arrays are compatible.* Since this work only considers array subscript expressions that are affine expressions [MT96, Wol92], determining whether array accesses are compatible is straightforward.

The importance of compatibility is illustrated using the example loop nest shown in Figure 6.4(a). Assuming column-major storage order, each array reference generates unit stride data accesses, as shown in Figure 6.4(b). Cache lines are accessed in the sequence they are stored in memory, and elements within each cache line are accessed sequentially. All three

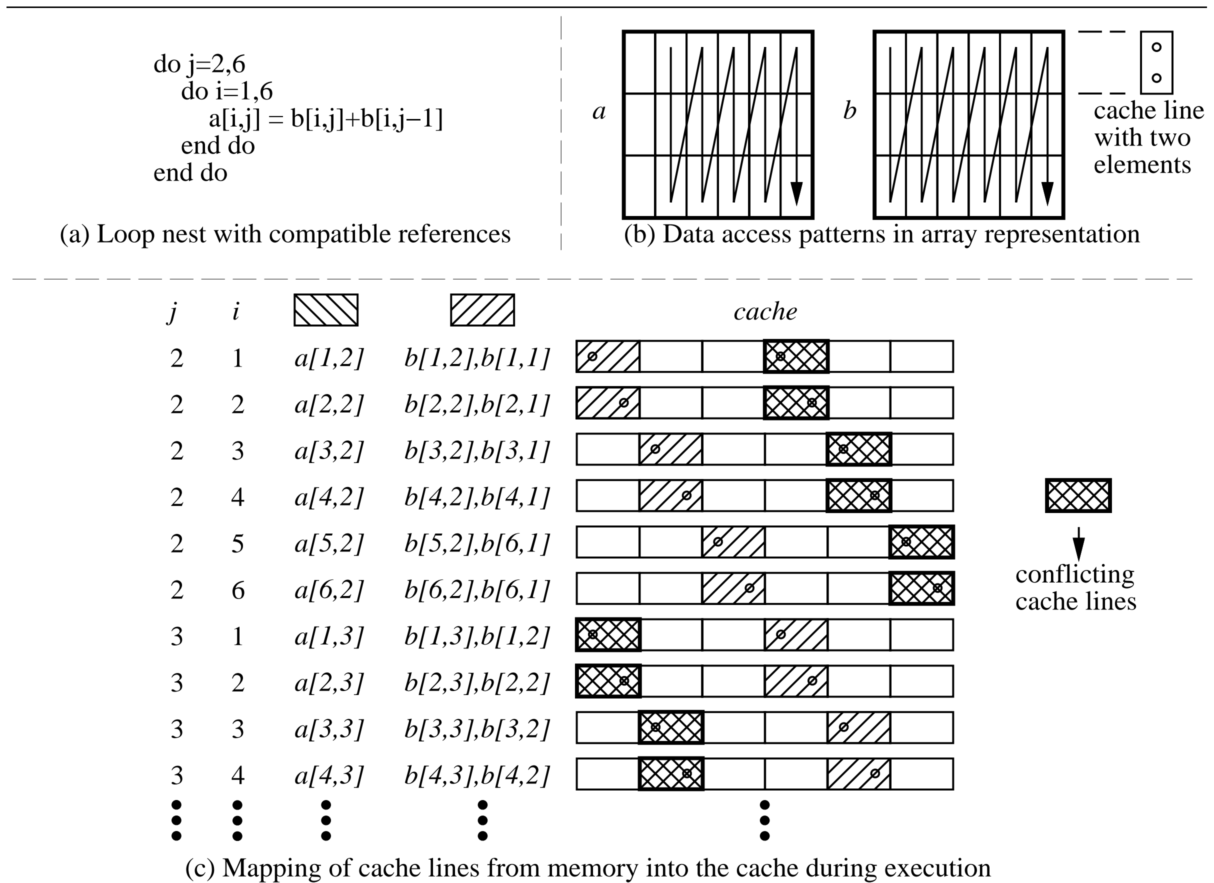


Figure 6.4: Frequency of cross-conflicts for compatible data access patterns

array references are therefore compatible. Figure 6.4(c) illustrates the mapping of individual cache lines from memory into a 12-element direct-mapped cache for each loop iteration. Note that two different cache lines are accessed for array b in each iteration. One of these cache lines *always* conflicts with the single cache line accessed for array a. As a result, one of the conflicting cache lines must be displaced from the cache in *every* loop iteration. Because there are two elements in each cache line, unnecessary misses are incurred to reload cache lines from memory in order to access the remaining element in each cache line.

In contrast, consider the example loop nest for matrix transpose shown in Figure 6.5(a). The data access patterns within each array are shown in Figure 6.5(b). The reference to array a generates unit-stride data accesses. However, the reference to array b does not generate unit-stride data accesses. Instead, the majority of accesses to array b have a stride of 6. Clearly,

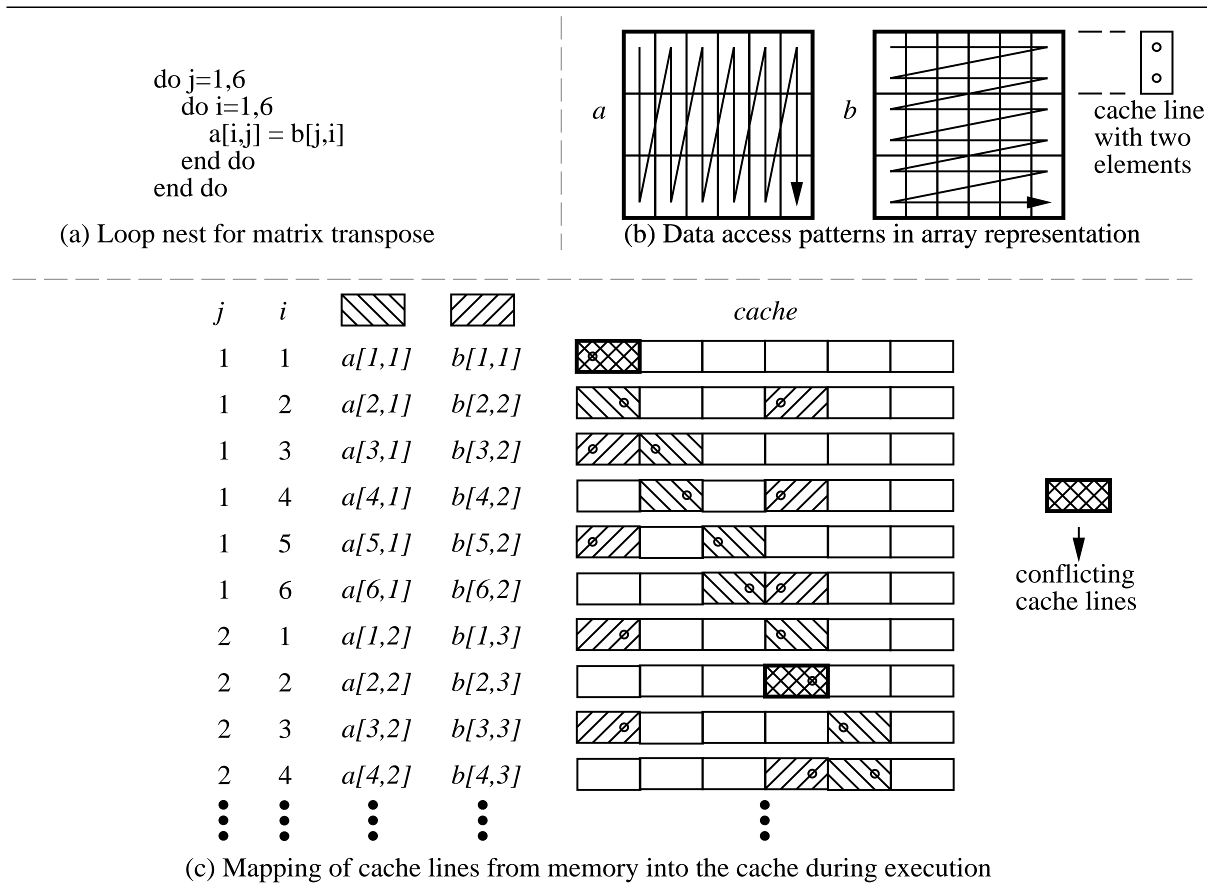


Figure 6.5: Frequency of cross-conflicts for incompatible data access patterns

the access patterns for arrays a and b are incompatible. Figure 6.5(c) illustrates the mapping of individual caches lines from memory into a direct-mapped cache for each loop iteration. The frequency of cross-conflicts is substantially less than if the references were compatible. For realistically large array and cache sizes, the frequency of cross-conflicts will be similarly low.

Because the majority of array references in representative loop nests generate unit-stride data accesses [CMT94, MT96], compatibility among array references is common. Furthermore, Figure 6.4 has demonstrated that compatibility leads to frequent cross-conflicts. *Hence, this chapter is concerned with conflict avoidance for compatible access patterns.*

In the event that at least one array in a loop nest has incompatible data access patterns, and compatibility is desired among all arrays, code and data transformations may be applied to obtain compatibility. For example, loop distribution (Section 2.4.5) can isolate any statements

referring to incompatible arrays in separate loops, and array dimension interchange (Section 2.5) can alter array element order to obtain compatibility.

6.1.4 Related Work

The most common hardware approach to reduce the adverse impact of cache conflicts is to increase the cache associativity, even though this may increase hardware complexity [PH96]. However, increased associativity may not necessarily reduce the occurrence of cross-conflicts for a large number of arrays and a large amount of reused data from each array that must remain cached for locality. The latter condition may result from applying the shift-and-peel transformation and tiling.

A related hardware approach is the use of a small *fully*-associative cache, known as a victim cache or assist cache, to supplement a large direct-mapped cache [CHK+96, Jou90]. The additional cache temporarily holds cache lines that are displaced due to conflicts in the main cache. If the displaced cache lines are reused shortly afterwards, the reuse is satisfied from the victim cache, rather than from slow main memory. However, the limited capacity of a victim cache may not be sufficient to hold large amounts of conflicting data.

As a software solution, Lam and Wolf [LRW91] present a tile size selection algorithm to prevent self-conflicts when tiling is used to exploit array data reuse. However, large cache sizes reduce the occurrence of self-conflicts. Coleman and McKinley [CM95] describe an improved tile size selection algorithm that they claim also reduces the likelihood of cross-conflicts. However, a much stronger guarantee is needed when a large amount of data from different arrays must remain cached for locality after applying an aggressive transformation such as shift-and-peel.

Temam et al. [TFJ93] study conflicts arising from array references in loop nests typical of scientific applications. They analyze instances of self-conflicts and cross-conflicts, and suggest the use of padding or careful placement of arrays in memory to reduce the occurrence of conflicts. However, no concrete methodology is given for achieving this goal.

Bacon et al. [BCJ+94] discuss a method to determine the amount of padding needed to avoid cache conflicts among individual array references in the innermost loop of a loop nest. However, their approach is not adequate for locality-enhancing loop transformations because

it does not consider data reuse in outer loops, and therefore cannot prevent conflicts for larger amounts of reusable data that must remain cached.

Lebeck and Wood [LW94] present a case study of improving cache performance with a variety of techniques including data transformations such as padding and memory alignment. However, these transformations are discussed in the context of programmer tuning of application performance with the aid of a simulation tool that profiles cache behavior. There is no discussion of how such transformations may be incorporated into a compiler.

Romer *et al.* [RLBC94] propose operating system policies for dynamic remapping of page assignments during execution to prevent conflicts in physically-indexed caches. The operating system *recolors* (i.e., relocates) pages in memory whenever conflicting pages are detected in the address translation buffer. The intent is to prevent future conflicts between data accessed from the affected pages. However, recoloring of pages may incur execution time overhead.

Bugnion *et al.* [BAM+96] present a technique called compiler-directed page coloring that customizes the page assignment at the start of program execution in order to prevent cache conflicts in physically-indexed caches. Compile-time analysis of array usage in loops is used to generate page-coloring hints for the operating system to reduce the likelihood that data from different pages conflicts in the cache.

Page coloring schemes for physically-indexed caches have the advantage of being transparent to the application, although compiler-directed coloring does requires compiler support. The only limitation of page coloring by the operating system is that it is not applicable for virtual caches, and some systems have been designed with a large, single-level virtual cache for performance reasons [DWYF92, LH97].

6.2 Cache Partitioning

This chapter proposes cache partitioning as a software technique that prevents cross-conflicts for reused data during the execution of a loop nest, specifically for the common and important case of compatible access patterns. The primary intent of cache partitioning is to ensure that reused data remains cached for locality after applying a locality-enhancing transformation. Cache partitioning modifies the array layout in memory in order to alter the mapping of data from different arrays into the cache and prevent the occurrence of conflicts.

This section presents an overview of cache partitioning, then discusses the technique in more detail. The technique is presented initially for a single loop nest. The technique is then extended to apply across multiple loop nests.

6.2.1 Overview

Consider the loop nest sequence shown in Figure 6.6(a). Data reuse across the loops can be exploited by applying simple fusion. In the fused loop nest shown in Figure 6.6(a), each outer loop iteration accesses two adjacent columns of data from each array. One column from each array is then reused in the subsequent iteration, and should remain cached for locality. However, cross-conflicts occur when these columns map into overlapping regions of the cache, as shown in Figure 6.6. Such conflicts displace data from the cache and diminish the benefit of fusion.

Cache partitioning removes these conflicts by adjusting the memory layout of the arrays. The cache is logically partitioned into *non*overlapping regions, one for each array, and then the array starting addresses are adjusted in virtual memory to map data from each array into a different partition, as shown in Figure 6.6(c). The partitioning is done entirely in software; no hardware support is required. The array starting addresses are adjusted by inserting appropriately-sized *gaps* between the arrays in memory. These gaps represent *inter*-array padding, rather than the conventional *intra*-array padding [BGS94]. In comparison with other data transformation techniques (as discussed in Section 2.5), cache partitioning does not require any modifications of array references or subscript expressions because only the starting addresses are affected; the internal array structure remains unchanged.

Although each array is assigned to a unique partition in the cache, the partitions are not static during the execution of a loop nest. Partitions cycle in unison through the cache as execution proceeds, as shown in Figure 6.7. Each partition contains data from a different array, and compatible array references ensure that as the partition boundaries move, no conflicts occur between data from different arrays. As new data from each array is brought into the cache, it displaces data from other arrays that is no longer needed.

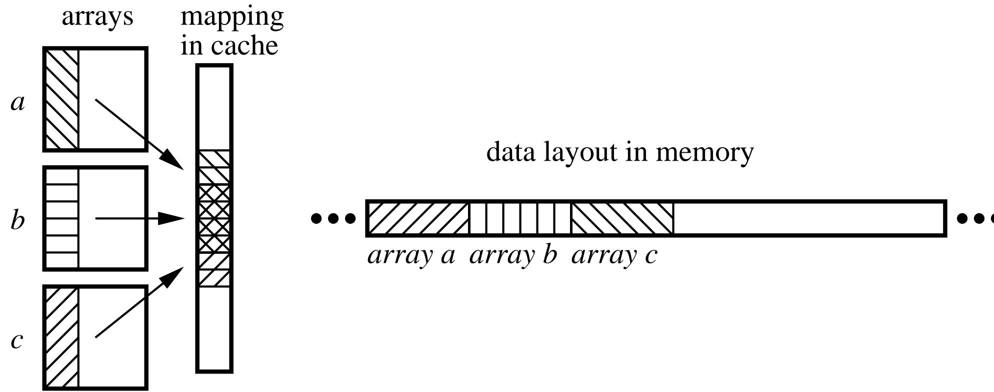
Cache partitioning assumes that arrays referenced in a loop nest are similar in size and dimensionality. Some loop nests reference lower-dimensionality arrays, and there is often temporal data reuse for these arrays (see Section 3.5). If this reuse is carried by the innermost

```
do j = 2, N-1
  do i = 1, N
    a[i,j] = a[i,j] + a[i,j-1]
  end do
end do
do j = 2, N-1
  do i = 1, N
    b[i,j] = a[i,j] + b[i,j-1]
  end do
end do
do j = 2, N-1
  do i = 1, N
    c[i,j] = b[i,j] + c[i,j-1]
  end do
end do
```

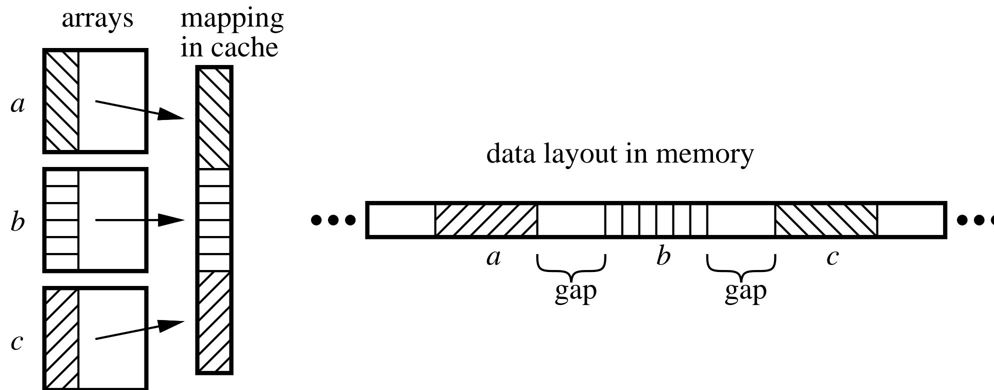
→

```
do j = 2, N-1
  do i = 1, N
    a[i,j] = a[i,j] + a[i,j-1]
    b[i,j] = a[i,j] + b[i,j-1]
    c[i,j] = b[i,j] + c[i,j-1]
  end do
end do
```

(a) Application of simple loop fusion to exploit array reuse



(b) Occurrence of cache conflicts for data accessed in fused loop



(c) Cache partitioning to modify data layout and prevent conflicts

Figure 6.6: Example of cache partitioning to avoid cache conflicts

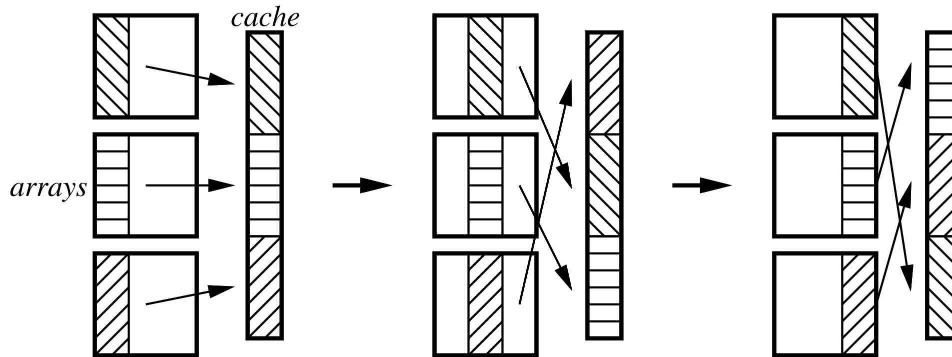


Figure 6.7: Conflict avoidance as partition boundaries move during loop execution

loop, the reused data may be register-allocated, and the array may be excluded from the set of arrays for cache partitioning. If the reuse is carried by an outer loop, the array may still be excluded, although the potential for conflicts with this array may increase. The alternative is to apply data transformations such as array expansion to make all of the arrays similar in size, but this approach leads to memory overhead and increases execution time because the temporal reuse of the same element is spread among distinct array elements.

6.2.2 One-dimensional Cache Partitioning

The simplest form of cache partitioning is *one-dimensional* cache partitioning, where partitions contain *contiguous* data from each array. One-dimensional partitioning limits the number of indices from the *outermost* array dimension that reside simultaneously in the cache. For each outermost index, all inner indices are present in the cache. One-dimensional cache partitioning was illustrated earlier in Figure 6.6(c); each partition contains two columns (i.e., two outer indices), and the columns are contiguous.

One-dimensional cache partitioning is generalized in the following manner. Given n_a arrays with dimensions $N_1 \times N_2 \times \cdots \times N_k$, and a cache capacity of c elements, n_a partitions are required in the cache. The size of each partition is $s_p = \lfloor c/n_a \rfloor$ elements. Assuming column-major storage order, the N_1 elements in the first dimension comprise a column and are stored contiguously in memory. The outermost array dimension is k, hence each partition contains a contiguous block of $N_1 \cdot N_2 \cdots N_{k-1} \cdot B_k$ elements. B_k is the limit on the number of indices from the outermost dimension, and is given by $B_k = \lfloor s_p/(N_1 \cdot N_2 \cdots N_{k-1}) \rfloor$. Note that this

```
GREEDYMEMORYLAYOUT(A)::                                    // A = set of arrays
  n_a = |A|                                      // number of arrays or partitions
  s_p = c/n_a                                                 // partition size
  C = {0, s_p, 2·s_p, ..., (n_a − 1)·s_p}          // partition starting addresses
  P = {0, 1, ..., n_a − 1}                        // available partition indices
  q = q_0                    // q_0 =starting address of available storage
  do
    select a ∈ A                                      // selection is arbitrary
    mapped_cache_address = CACHEMAP(q)
    foreach p ∈ P do                    // determine gaps for available partitions
      gap(p) = C(p)− mapped_cache_address
      if C(p) < mapped_cache_address then
        gap(p) = gap(p) + cache_size              // "wraparound" in the cache
      endif
    endfor
    select p_opt ∈ P where gap(p_opt) = min_{p∈P} gap(p)     // select minimum gap
    P = P \ {p_opt}                          // remove from available partitions
    START(a) = q + gap(p_opt)                                     // insert gap
    q = START(a) + SIZE(a)                          // adjust start for next array
    A = A \ {a}                                    // remove from set of arrays
  while A ≠ ∅
```

Figure 6.8: Greedy memory layout algorithm for cache partitioning

assumes that $N_1 \cdot N_2 \cdots N_{k-1} < s_p$. If this condition is not satisfied, multidimensional cache partitioning (to be discussed in Section 6.2.3) is required.

The starting addresses of the n_a cache partitions must be separated by a distance s_p to ensure that they do not overlap. If the first partition begins at address 0 in the cache, the partition starting addresses are $0, s_p, 2 \cdot s_p, \ldots, (n_a - 1) \cdot s_p$. The array starting addresses must then be adjusted to map to unique partition starting addresses in the cache. This adjustment is accomplished by inserting gaps between the arrays in memory, as shown in Figure 6.6(c). These gaps represent memory overhead that should be minimized.

The greedy memory layout algorithm shown in Figure 6.8 performs three tasks: (a) it assigns each array to a unique partition, (b) it inserts gaps in memory to enforce the partition assignments, and (c) it attempts to minimize the overhead of the gaps. The arrays are selected in an arbitrary order. A set of available partitions P is maintained, and each array is assigned to

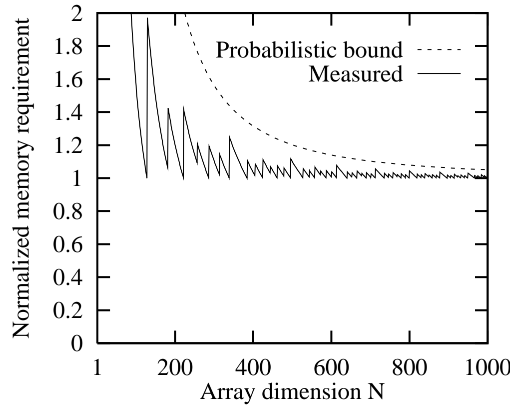


Figure 6.9: Memory overhead for 8 $N \times N$ arrays from cache partitioning (cache size=131,072)

a partition that minimizes the distance between the starting address required for that partition and the end of the array most recently placed in memory. Although multiple memory addresses map into the selected partition, the address in free memory closest to the end of the most recently placed array is always used. Each partition selected in this manner is removed from the set of available partitions to ensure that two arrays are not assigned to the same partition. The algorithm assumes a single-level, virtually-indexed, direct-mapped cache with an index function CACHEMAP(). The complexity of the algorithm is $O(n_a^2)$.

An upper bound for the overhead (or increase in memory usage) from the gaps introduced by this algorithm is estimated as follows. Using a probabilistic argument, if there are i partitions remaining, the closest partition starting address is expected to be $(1/i) \cdot c$ elements from the end of the most recently positioned array. Hence, the total size of the gaps is expected to be $\sum_{i=1}^{n_a}(1/i) \cdot c$. The quantity $\sum_{i=1}^{n_a}(1/i)$ is bounded from above by $\ln(n_a) + 1$. Hence, the bound on the expected memory overhead is

$$\frac{(\ln(n_a) + 1) \cdot c}{n_a \cdot d},$$

where c is the cache size, and $d = N_1 \cdot N_2 \cdots N_k$, the size of each array.

To verify this upper bound on memory overhead, Figure 6.9 shows the cache-partitioned memory requirements normalized to the requirements for contiguous array layout. Cache partitioning is applied to 8 arrays with dimensions $N \times N$, and N is varied from 1 to 1000. The cache size is 131,072 elements (all 8 arrays fit in the cache when $N = 128$). When compared to the measured overhead from cache partitioning in each case, the probabilistic bound described

above is reasonably tight, especially as N increases. Clearly, the overhead diminishes rapidly as the array size increases relative to the cache size, which is the case in applications where locality enhancement (and hence conflict avoidance) is required.

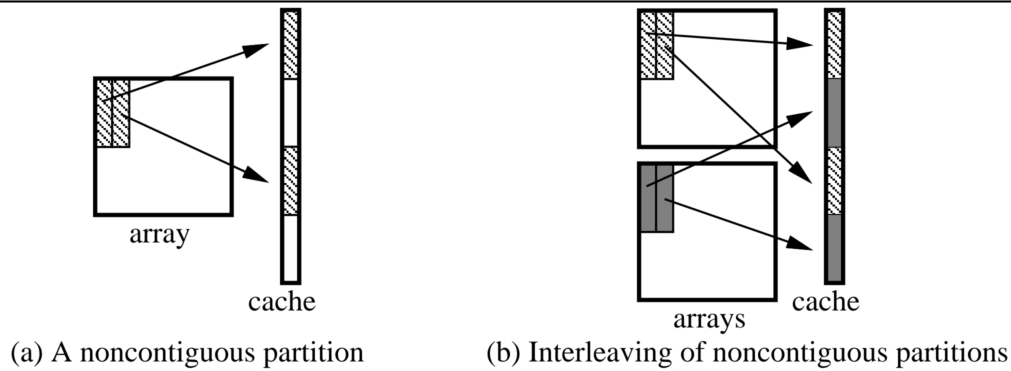
The algorithm in Figure 6.8 assumed a direct-mapped cache. A cache with an associativity of $m \geq 2$ and capacity of c may be viewed as a set of m memory banks, each with capacity c/m. Cache partitioning is still applicable in this case. Because m memory locations may be mapped to the same cache location, there may be m cache partitions with the same starting address. However, the partition size is still determined from the total capacity c. For example, if $n_a = 4$, the partition size is $s_p = c/4$. For a 2-way associative cache ($m = 2$), the starting addresses for 4 partitions are $\{0, 0, c/4, c/4\}$. Hence the only change for the algorithm is the set of partition starting addresses.

The above discussion also assumed a single-level, virtually-indexed cache. In a multilevel cache, conflicts must be avoided in the physically-indexed level closest to main memory. Cache partitioning is still applied in the same way to virtual addresses. If the operating system maps the virtual address space onto the underlying physical address space such that all non-conflicting virtual addresses imply non-conflicting physical addresses, then cache partitioning applies identically to both virtual and physical address spaces.

Finally, partition starting addresses can be adjusted to avoid conflicts in all levels of a multilevel cache hierarchy. For example, consider a two-level, direct-mapped hierarchy where the primary cache has capacity c_p, and the secondary cache has a larger capacity $c_s = 64 \cdot c_p$. If $n_a = 4$, then the partition starting addresses in the secondary cache are $\{0, c_s/4, c_s/2, 3 \cdot c_s/4\}$. However, these starting addresses conflict in the small primary cache; they all map to location 0. The starting addresses must be adjusted to separate them in the primary cache; since $n_a = 4$, additional offsets in multiples of $c_p/n_a = c_p/4$ must be used. The conflict-free starting addresses are $\{0, (c_s + c_p)/4, (c_s + c_p)/2, 3 \cdot (c_s + c_p)/4\}$.

6.2.3 Multidimensional Cache Partitioning

Multidimensional cache partitioning is used when the cache capacity is not sufficient to hold contiguous data from all arrays, or if the limit on the number of outermost indices is insufficient to provide locality for reused data. Multidimensional cache partitioning reduces



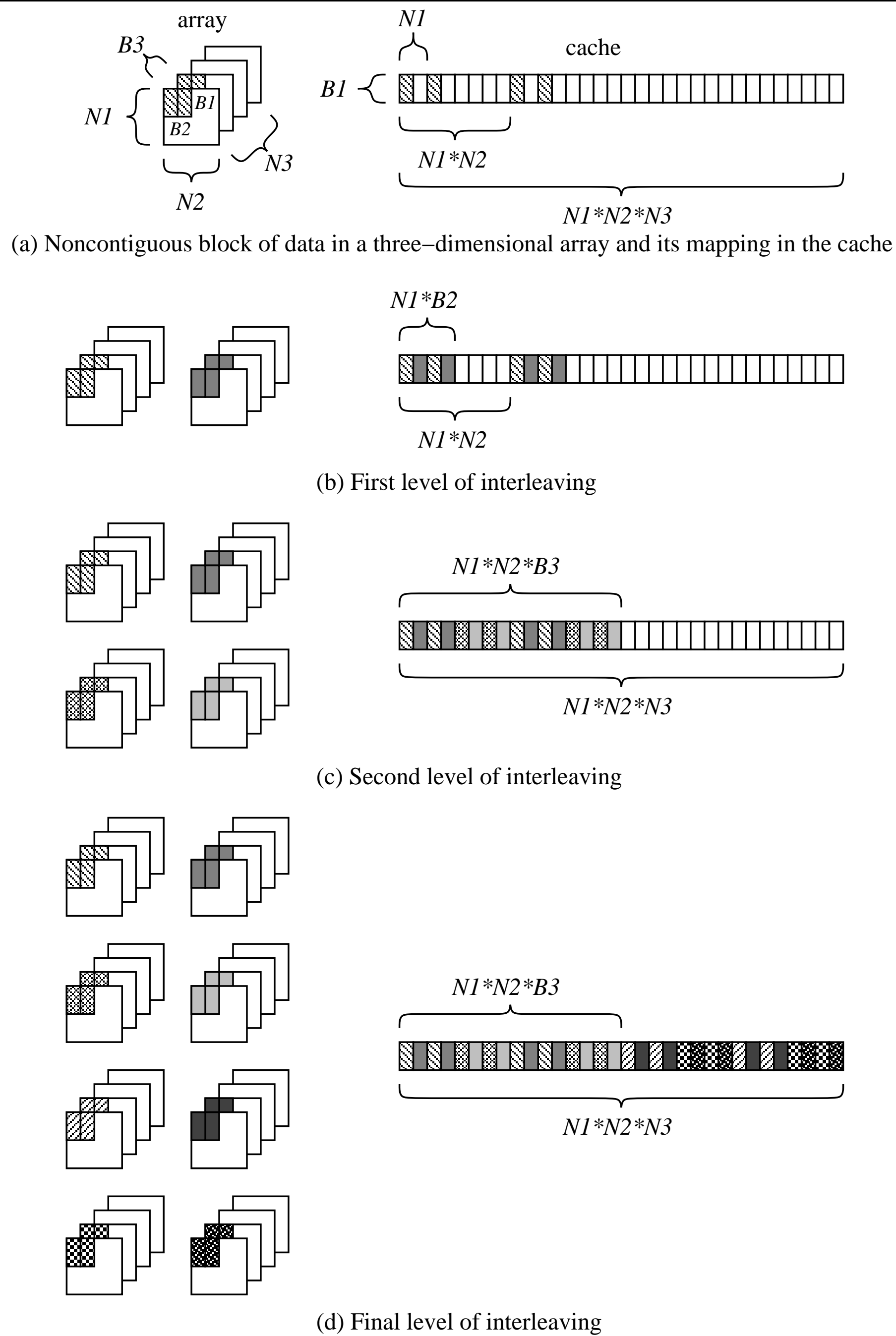
(a) A noncontiguous partition (b) Interleaving of noncontiguous partitions

Figure 6.10: Multidimensional cache partitioning

the number of indices from inner array dimensions that are in the cache in order to increase the available cache capacity for indices from the outermost dimension. In this case, the data in each partition is no longer contiguous because reducing the number of indices from inner dimensions skips over portions of the array in memory. Multidimensional partitioning must be accompanied with an appropriate code transformation to reduce the data accessed from inner dimensions (an example is multidimensional shift-and-peel as discussed in Section 4.3).

Since the data is not contiguous in memory, the partitions containing this data in the cache are not contiguous either, as shown in Figure 6.10(a). These noncontiguous partitions must be carefully interleaved in the cache to ensure that they do not overlap and cause conflicts, as in Figure 6.10(b). Hence, the goal of multidimensional partitioning is to determine the starting addresses for these interleaved, noncontiguous partitions. These starting addresses are then used to derive the memory layout using the greedy algorithm of Figure 6.8.

Multidimensional cache partitioning is generalized in the following manner. For arrays with dimensions $N_1 \times N_2 \times \cdots \times N_k$, the first task is to determine appropriate block dimensions $B_1 \times B_2 \times \cdots \times B_k$, where $B_i \leq N_i$, $1 \leq i \leq k$. The block dimensions must satisfy the cache capacity constraint $n_a \cdot B_1 \cdot B_2 \cdots B_k \leq c$, where n_a is the number of arrays and c is the cache size. A simple choice is a common block size $B_1 = \cdots = B_k = \lfloor \sqrt[k]{c/n_a} \rfloor$. However, the data access patterns for the arrays in a loop nest may dictate a minimum block size in one or more dimensions. For example, the block size for the innermost dimension may be set equal to a multiple of the cache line size. The block sizes for the remaining dimensions are then chosen subject to the above capacity constraint.



(a) Noncontiguous block of data in a three−dimensional array and its mapping in the cache

(b) First level of interleaving

(c) Second level of interleaving

(d) Final level of interleaving

Figure 6.11: Interleaving partitions in multidimensional cache partitioning

Once the block dimensions are known, the interleaving of the partitions is determined using a set of interleaving factors $n_1, \ldots, n_k$. These interleaving factors specify how successively larger groups of partitions can be interleaved without overlap. It is initially assumed that $N_1 \cdot N_2 \cdots N_k = c$ to derive the interleaving factors; this restriction is later relaxed. For a three-dimensional array, Figure 6.11(a) shows how $B_2 \cdot B_3$ subblocks, each of size B_1, are mapped into the cache starting at address 0. For a given index in the second array dimension, the starting addresses of two contiguous subblocks of size B_1 are separated by a distance N_1, as shown in Figure 6.11(a). In a space of size N_1, $n_1 = \lfloor N_1/B_1 \rfloor$ subblocks of size B_1 from different arrays may be placed. Hence, n_1 partitions for n_1 different arrays are interleaved to create a contiguous region of size $N_1 \cdot B_2$, as shown in Figure 6.11(b). To prevent these n_1 partitions from overlapping, their starting addresses in the cache must be separated (i.e., shifted in the cache) by a distance $g_1 = B_1$. If the first partition begins at address 0, the remaining starting addresses are $g_1, 2 \cdot g_1, \ldots, (n_1 - 1) \cdot g_1$. For example, in Figure 6.11(b), we have $B_1 = N_1/2$. Hence, the interleaving factor is $n_1 = 2$, and the starting addresses of the two partitions are 0 and g_1.

After interleaving a group of n_1 partitions in the cache, there are B_3 contiguous regions of size $g_2 = N_1 \cdot B_2$ whose starting addresses are separated by a distance $N_1 \cdot N_2$, as illustrated in Figure 6.11(b). Identical groups of partitions of may be introduced into the space between these contiguous regions. The number of groups that can be interleaved within a distance of $N_1 \cdot N_2$ is given by the interleaving factor $n_2 = \lfloor (N_1 \cdot N_2)/(N_1 \cdot B_2) \rfloor = \lfloor N_2/B_2 \rfloor$. There are now a total of $n_1 \cdot n_2$ partitions. To ensure that the n_2 groups of n_1 partitions do not overlap, the groups must be separated or shifted by a distance g_2 in the cache. For example, in Figure 6.11(c), we have $B_2 = N_2/2$. Hence, the interleaving factor is $n_2 = 2$. The total number of partitions to this point is $n_1 \cdot n_2 = 4$, and the starting addresses are $0, g_1, g_2, g_2 + g_1$.

After interleaving n_2 groups of n_1 partitions in the cache, there is a contiguous region of size $g_3 = N_1 \cdot N_2 \cdot B_3$ in the cache, as illustrated in Figure 6.11(c). The cache size is $N_1 \cdot N_2 \cdot N_3 > g_3$, hence identical groups of partitions for other arrays may be introduced into the remaining space. The number of such groups is determined by the interleaving factor $n_3 = \lfloor (N_1 \cdot N_2 \cdot N_3)/(N_1 \cdot N_2 \cdot B_3) \rfloor = \lfloor N_3/B_3 \rfloor$. The total number of partitions is now $n_1 \cdot n_2 \cdot n_3$. To ensure that the n_3 groups do not overlap, they must be separated or shifted by

a distance g_3 in the cache. For example, in Figure 6.11(c), we have $B_3 = N_3/2$. Hence, the interleaving factor is $n_3 = 2$. The total number of partitions to this point is $n_1 \cdot n_2 \cdot n_3 = 8$, and the starting addresses of the partitions are $0, g_1, g_2, g_2 + g_1, g_3, g_3 + g_1, g_3 + g_2, g_3 + g_2 + g_1$.

In general, interleaving for k-dimensional noncontiguous partitions results in

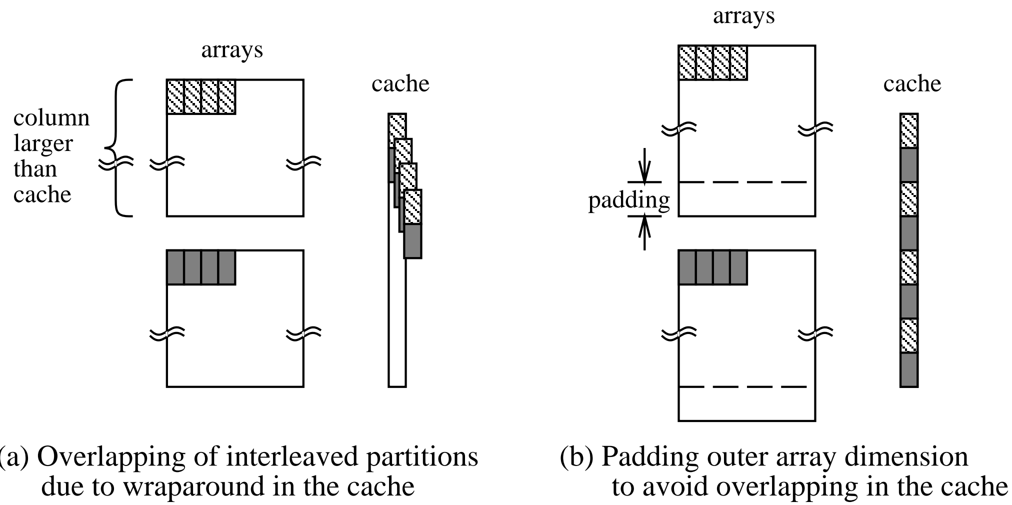
$$\begin{array}{llll} n_1 = \lfloor N_1/B_1 \rfloor, & n_2 = \lfloor N_2/B_2 \rfloor, & \ldots, & n_k = \lfloor N_k/B_k \rfloor, \\ g_1 = B_1, & g_2 = N_1 \cdot B_2, & \ldots, & g_k = N_1 \cdot N_2 \cdots N_{k-1} \cdot B_k, \end{array}$$

where n_i specifies the number of groups that can be interleaved at each point, and g_i specifies the separation between the groups to prevent overlapping. The base offset for a given group is $t_i \cdot g_i$, where $0 \leq t_i < n_i$, and the starting address for each partition is determined by summing the group offsets across all dimensions, $\sum_{i=1}^{k} t_i \cdot g_i$.

Upon completion of the interleaving, it must be true that $n_1 \cdot n_2 \cdots n_k \geq n_a$. It is possible for this condition to be violated even if the capacity constraint is satisfied because of the truncation in the calculation of $n_1, \ldots, n_k$. In such cases, one or more of the block sizes $B_1, \ldots, B_k$ may be decreased in order to increase the corresponding interleaving factors by a sufficient amount to satisfy this condition.

The restriction $N_1 \cdot N_2 \cdots N_k = c$ is now removed, and the case of $N_1 \cdot N_2 \cdots N_k < c$ is now considered. In the preceding case of $N_1 \cdot N_2 \cdots N_k = c$, the final interleaving factor is given by $n_k = \lfloor (N_1 \cdot N_2 \cdots N_k)/(N_1 \cdot N_2 \cdots B_k) \rfloor = \lfloor N_k/B_k \rfloor$. In this case, the cache size is larger than $N_1 \cdot N_2 \cdots N_k$. To use the additional cache space for partitions, the final interleaving factor is computed as $n_k = \lfloor c/(N_1 \cdot N_2 \cdots B_k) \rfloor$. The preceding interleaving procedure is applied in the same way except for the change in computing n_k. Note that the block dimensions are still constrained by $n_a \cdot B_1 \cdot B_2 \cdots B_k \leq c$.

The final case to consider is $N_1 \cdot N_2 \cdots N_k > c$. Since the array size exceeds the cache size, *wraparound* occurs when mapping data into the cache. In this case, padding is introduced in the array dimensions that cause wraparound to ensure that a partition for a given array does not overlap with itself in the cache, and also to prevent partitions for different arrays from overlapping with each other. The innermost dimension i in which wraparound occurs, i.e. the smallest i such that $N_1 \cdot N_2 \cdots N_i > c$, is identified. In this dimension, the largest index m_i, $1 \leq m_i < N_i$, that does not cause wraparound is determined. In other words, the largest m_i such that $N_1 \cdot N_2 \cdots m_i \leq c$, but $N_1 \cdot N_2 \cdots (m_i + 1) > c$, is determined. The restriction



(a) Overlapping of interleaved partitions due to wraparound in the cache

(b) Padding outer array dimension to avoid overlapping in the cache

Figure 6.12: The use of padding to handle wraparound in the cache

$B_i \leq m_i$ is then introduced so that the contiguous subblocks of size $N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i$ that result from interleaving do not exceed the cache size.

If wraparound occurs in dimension i, then wraparound will also occur in all remaining outer dimensions $i < j \leq k$ because the array size increases by a factor of N_j in each outer dimension j. Although it is possible to avoid the complications of wraparound for these outer dimensions by requiring $B_j = 1$, $i < j \leq k$, this approach is highly restrictive and may not satisfy other requirements on the block dimensions. To allow wraparound without overlap, an appropriate amount of padding is introduced in each outer dimension such that contiguous segments of data corresponding to adjacent indices in an outer dimension map to adjacent, nonoverlapping regions of the cache, as shown in Figure 6.12.

For each of the outer dimensions $i < j \leq k$ in which wraparound is permitted, the procedure for introducing padding is as follows. First, the size of the contiguous data block corresponding to a single index of dimension j is determined as $g_{j-1} = N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i \cdots B_{j-1}$. We require $g_{j-1} \leq c$ to prevent the contiguous data region from overlapping with itself. Assuming that the start of this region maps to address 0 in the cache, the end of the region maps to address g_{j-1}. However, the next index in dimension j maps to address $c_{j-1} =$ CACHEMAP$(N_1 \cdot N_2 \cdots N_{j-1})$ in the cache, assuming the array starting address maps to address 0. If $c_{j-1} \neq g_{j-1}$, then overlaps will occur due to wraparound for adjacent indices of dimension j. To prevent overlaps, a padding

of p_{j-1} is required in dimension $j-1$ such that $\text{CACHEMAP}(N_1 \cdot N_2 \cdots N_{j-2} \cdot (N_{j-1}+p_{j-1})) = g_{j-1}$. The padding ensures that blocks of data corresponding to adjacent indices in dimension j map to adjacent, nonoverlapping regions of the cache. Because the blocks are adjacent in the cache after padding (i.e., there is no space between these blocks), the interleaving factor for dimension $j-1$ is $n_{j-1} = 1$. For all subsequent uses of dimension $j-1$, N_{j-1} is replaced with $(N_{j-1} + p_{j-1})$. The block size in dimension j is B_j, hence the block of data for B_j adjacent indices occupies a contiguous region of size $g_j = N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i \cdots B_{j-1} \cdot B_j$. The above procedure is then repeated for dimension $j+1$.

The use of padding in the manner described above forces the interleaving factor to be 1 for each outer dimension $i \leq j < k$. For the outermost dimension k, the interleaving factor is $n_k = \lfloor c/(N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i \cdots B_k) \rfloor$. The inner dimensions $1 \leq j \leq i-1$ are unaffected by the padding, hence the interleaving factor is still determined as $n_j = \lfloor N_j/B_j \rfloor$. As before, it must be true that $n_1 \cdot n_2 \cdots n_k \geq n_a$ to ensure that a sufficient number of partitions are created. If not, one or more of the block dimensions $B_1, B_2, \ldots, B_k$ are reduced to permit increasing the interleaving factors to satisfy the condition.

6.2.4 Cache Partitioning for Multiple Loop Nests

Real applications consist of more than one loop nest, and several loop nests may reference the same set of arrays. Hence, cache partitioning should also be applicable for arrays referenced in multiple loop nests. The goal is to derive an appropriate data layout such that there are no conflicts among the arrays in any of the loop nests. This approach would be used, for example, after fusing different loop nest sequences that accessed a common set of arrays. Cache partitioning is extended for such cases by first determining the number of cache partitions needed to satisfy all the resulting loop nests, then assigning the arrays to those partitions such that no two arrays used in the same loop nest conflict with each other.

A program may contain $n_\ell \geq 2$ loop nests referencing a common set of arrays. The number of partitions required for each loop nest is equal to the number of arrays in the loop nest, and is generally different for each loop nest. Consequently, deriving the cache-partitioned memory layout for each loop nest individually results in different sets of starting addresses for the same arrays. To avoid conflicting requirements on starting addresses, a single set of partitions and

array-to-partition assignments is used for all loop nests. Not all of the arrays are used in any one loop nest, hence the number of partitions in this set may be larger than required for a given loop nest. Furthermore, two arrays may be assigned to the same partition if they are not used in the same loop nest, hence there may be fewer partitions than arrays. Therefore, the extension of cache partitioning to multiple loop nests requires: (a) determining the number of partitions that satisfies all loop nests, and (b) assigning arrays to partitions when there are fewer partitions than arrays. The remainder of this section addresses these two aspects of the problem.

The problem of finding the required number of partitions for multiple loop nests is formulated as a graph-coloring problem. Let L denote a set of n_ℓ loop nests referencing a set A of n_a arrays. Let $A(\ell)$ denote the set of arrays referenced (read or written) in a loop nest $\ell \in L$. The number of partitions required individually by each loop nest ℓ is $|A(\ell)|$. A graph $G(V, E)$ is constructed with a set of vertices $V[G] = A$ representing the arrays, and a set of edges $E[G]$. If arrays a_1,a_2 are referenced in the same loop nest, then there is an edge $e = (a_1, a_2) \in E[G]$. Consequently, the arrays referenced in a loop nest form a *clique* (a fully-connected subgraph) of size $|A(\ell)|$ in the graph $G(V, E)$. The goal is to label each vertex with a color such that no vertices connected by an edge have the same color, and the number of colors is minimized. The number of colors is then interpreted as the number of partitions n_p required to satisfy all loop nests, and similarly-colored vertices denote arrays that are assigned to the same partition.

Finding the minimum number of colors, or *chromatic number*, for an arbitrary graph is an NP-complete problem [GJ79]. However, it is possible to specify a lower bound for the solution in this case, based on the construction of the graph described above. The lower bound for the chromatic number is $n_r = \max_{\ell \in L} |A(\ell)|$ because there is at least one clique of n_r vertices embedded in the graph. A clique of n_r requires no fewer than n_r colors. Any approximation algorithm for graph coloring may be applied to find a solution n_p for the entire graph. If $n_p = n_r$, then the solution is optimal.

The number of colors n_p obtained from graph coloring determines the required number of partitions. Cache partitioning is then used to obtain the starting addresses for a set of n_p partitions in the cache. The problem is to map colors in the graph to cache partitions and place the arrays in memory such that the sizes of the gaps inserted to enforce the partition mappings are minimized. The problem is constrained by the fact that identically-colored vertices in the

graph represent arrays that share the same partition.

The greedy algorithm in Figure 6.13 is employed to reduce gap sizes using an approach similar to that used in the algorithm shown in Figure 6.8. The input consists of the result of graph coloring and the set of starting addresses for the partitions in the cache. The output is a mapping of colors to partitions and a memory layout for the arrays based on this mapping. The algorithm selects arrays in an arbitrary order for placement in memory. If the color assigned to the array has not yet been mapped to a partition, then one is chosen by computing gap sizes for all available partitions, then selecting the partition yielding the smallest gap. This selection implicitly determines the partition assignment for all remaining arrays sharing the same color. When one of these remaining arrays is later selected by the algorithm, the size of the gap inserted for the layout is computed using the previously-assigned partition since there is no longer any choice for that array. The complexity of the algorithm is $O(n_a \cdot n_p)$.

To determine the memory overhead from the greedy algorithm for multiple loop nest, it is important to note that $n_p \leq n_a$. In other words, for $n_a - n_p$ of the arrays, there is no choice in the partition assignment; the coloring dictates a fixed assignment. A simple probabilistic approach can be employed to arrive at a reasonable estimate for the expected memory overhead. When the color assigned to an array has not yet been mapped to a partition, the distance to the closest available partition (i.e., the gap size) is assumed to be $(1/i) \cdot c$, where i is the number of unassigned partitions remaining and c is the cache size. However, when the color has already been mapped to a partition, the gap size is expected to be $c/2$. The expected overhead o_m from combining these two cases is

$$o_m \approx \frac{\left(\dfrac{n_a - n_p}{2} + \displaystyle\sum_{i=1}^{n_p} \frac{1}{i} \right) \cdot c}{n_a \cdot d},$$

where $d = N_1 \cdot N_2 \cdots N_k$ is the array size. The overhead diminishes rapidly as the data size increases relative to the cache size.

6.3 Chapter Summary

This chapter has described a conflict avoidance technique called cache partitioning. Cache conflict avoidance is crucial for locality-enhancing transformations that rely on retaining data

```
GREEDYMEMORYLAYOUT2(n_p,A,COLOR,C)::                                   // A = set of arrays
                  // COLOR: A ↦ {0, 1, ..., n_p − 1} (output from graph coloring)
                           // C = {c_0, ···, c_{n_p−1}} (starting addresses in cache)
  P = {0, 1, ..., n_p − 1}                               // unassigned partition indices
  q = q_0                                   // q_0 =starting address of available storage
  foreach p ∈ P
    partition(p) = −1                        // initial partition mappings are undefined
  endfor
  do
    select a ∈ A                                               // selection is arbitrary
    ℓ = COLOR(a)                                                  // get color for array
    if partition(ℓ) = −1 then                           // not yet assigned to a partition
      mapped_cache_address = CACHEMAP(q)
      foreach p ∈ P                                                    // determine gaps
        gap(p) = C(p)− mapped_cache_address
        if C(p) < mapped_cache_address then
          gap(p) = gap(p) + cache_size
        endif
      endfor
      select p_opt ∈ P where gap(p_opt) = min_{p∈P} gap(p)          // select minimum gap
      gap = gap(p_opt)
      P = P \ {p_opt}                                  // remove from available indices
      partition(ℓ) = p_opt                      // establish color-to-partition mapping
    else
      p = partition(ℓ)                             // color already assigned to partition
      mapped_cache_address = CACHEMAP(q)
      gap= C(p)−mapped_cache_address
      if C(p) <mapped_cache_address then
        gap = gap + cache_size
      endif
    endif
    START(a) = q + gap                                                     // insert gap
    q = START(a) + SIZE(a)                                 // adjust start for next array
    A = A \ {a}
  while A ≠ ∅
```

Figure 6.13: Greedy memory layout algorithm for multiple loop nests

in the cache. Cache partitioning addresses the commonly-occurring case of compatible data access patterns that can lead to frequent conflicts in loop nests. With one-dimensional cache partitioning, data from each array is contiguous in the cache because data from all inner dimensions is cached. Multidimensional cache partitioning results in non-contiguous partitions by reducing the amount of cached data from inner array dimensions, and is useful when contiguity causes the cache capacity to be exceeded. Finally, cache partitioning has been extended to apply across multiple loop nests accessing a common set of arrays.

Chapter 7

Experimental Evaluation

This chapter provides an experimental evaluation of the cache-locality-enhancing techniques proposed in this dissertation. The objective is to demonstrate the feasibility and effectiveness of the proposed techniques for representative applications on contemporary shared-memory multiprocessors. In particular, the intent is to not only show that the proposed techniques provide significant performance improvements, but also to examine the factors influencing performance such as the number of cache misses and the latency for cache misses.

This chapter is organized as follows. First, the prototype implementation of the proposed techniques within an existing compiler framework is described. Next, the multiprocessor experimental platforms are described. The remaining discussion is then devoted to reviewing the experimental results. Improvements in performance are reported along with detailed measurements of cache behavior in order to explain the observed improvements. The measured improvements in performance are also compared with estimated improvements obtained with the model proposed in Chapter 3.

7.1 Prototype Compiler Implementation

This section outlines a prototype implementation of the proposed techniques in an experimental compiler infrastructure. An overview of the compiler infrastructure is given first, followed by a summary of the enhancements and additions needed to support the proposed techniques.

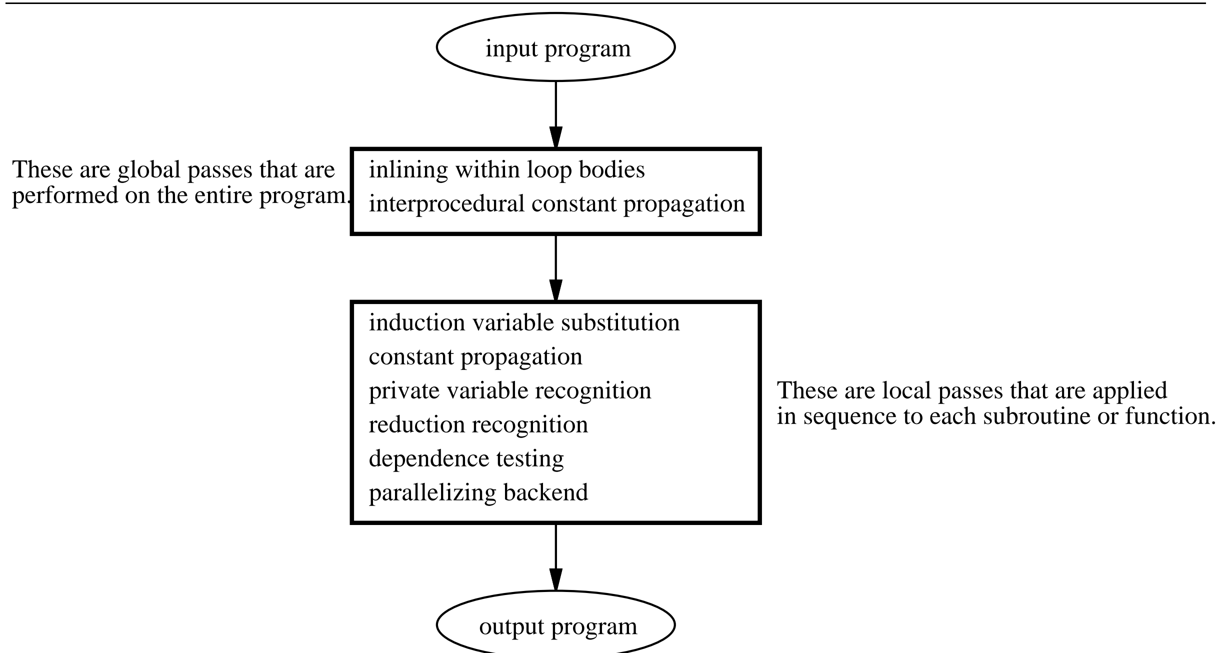


Figure 7.1: Passes in the Polaris compiler

7.1.1 Compiler Infrastructure

The prototype implementation of the proposed techniques was developed in the *Polaris* compiler infrastructure [BEF+95]. Polaris is a source-to-source restructuring tool whose input and output are FORTRAN 77 programs augmented with directives embedded in comments. The primary purpose of Polaris is to detect parallel loops. Polaris is implemented in an object-oriented manner and provides classes of objects for constructing an internal representation of program source code, along with functionality to manipulate the internal representation.

Polaris consists of several passes that are applied in sequence, as shown in Figure 7.1. The dependence testing pass is the key pass. To enable more accurate dependence testing, the global passes propagate constants and perform selective inlining of loop bodies. The induction, reduction, and private variable recognition passes identify variables that generate serializing dependences. Such variables are listed in annotations embedded in the internal representation, and transformations such as array privatization are later used to remove these dependences.

Since Polaris is primarily intended to detect parallel loops, dependences are only tested and represented *within* loop nests. Loop-carried dependences are represented with direction

vectors only; no distance information is maintained. Dependence analysis marks loops that do not carry dependences for the benefit of the parallelizing backend.

The final pass in Polaris before generating the output source program is the parallelizing backend. This pass searches for annotations identifying parallel loops, and annotations listing variables that are privatizable or involved in reductions. The output program is then tailored for the target machine by converting the parallel loop annotations to target-specific directives. At the same time, transformations for private or reduction variables are applied, or appropriate directives are generated if the target machine provides them.

7.1.2 Enhancements to Infrastructure

A number of enhancements were required to incorporate the new techniques proposed in this dissertation into Polaris. The enhancements and the implementation are summarized in the following paragraphs. Altogether, the new code for the enhancements and core techniques comprises over 4,000 lines of executable C++ code.

7.1.2.1 Support for High-level Code Transformations

Polaris is designed primarily to detect and exploit parallelism in loops with minimal change to the source code. In contrast, the techniques proposed in this dissertation require structured, high-level code transformations (e.g., strip-mining and fusion). To support these transformations, a new object library was incorporated into Polaris. Each object in this library performs a high-level transformation such as loop fusion or strip-mining in a structured manner. Compound transformations are supported by collecting individual high-level transformation objects into a special container object that specifies the affected code and the order in which the transformations are to be applied. These compound transformations cannot be represented as simple unimodular transformations because the component transformations include fusion and strip-mining. However, the three elementary unimodular transformations (skewing, reversal, and permutation) are included in the library, hence unimodular transformations are a proper subset of the possible compound transformations.

7.1.2.2 Dependence Distance Information Across Loop Nests

The shift-and-peel transformation proposed in Chapter 4 requires dependence analysis across loop nests as well as accurate distance information. Polaris only performs dependence analysis within loops and does not extract distance information. Consequently, a new dependence testing pass was developed in Polaris to identify candidate loop nest sequences for fusion, then apply the Omega Test [Pug92] to pairs of array references in *different* loop nests to obtain distance information. This distance information is then incorporated into a dependence graph, as described in Chapter 4.

7.1.2.3 Manipulation of Array Data Layout

Polaris is a source-to-source transformation tool, hence the final data layout is ultimately determined by the native compiler on the target machine. To implement cache partitioning, some control over data layout must be exercised at the source code level. Explicit control over data layout at the source code level in FORTRAN 77 is limited to COMMON blocks since compilers are required to preserve the order and content of COMMON blocks. Hence, the prototype source-level implementation of cache partitioning is limited to arrays in COMMON blocks, which may require modifications to source code to collect arrays into COMMON blocks where necessary. Furthermore, cache partitioning requires consistent definitions of the same COMMON block in different parts of the program. Compilers may not be able to enforce this consistency when different definitions of the same COMMON block cause memory aliasing. To overcome this limitation, source code modifications may also be required to enforce consistency.

A new pass was introduced into Polaris for cache partitioning. The memory layout algorithm described in Chapter 6 is applied to candidate arrays in order to determine the sizes of the gaps to be introduced between arrays in order to enforce a conflict-free data layout. With this information, a global pass is made over the entire program, where the COMMON blocks are first restructured to collect arrays into the same COMMON block, then the required gaps are introduced between arrays in each COMMON block.

7.2 Experimental Platforms

The experiments described in this chapter were conducted on two representative shared-memory multiprocessor architectures: the HP/Convex SPP series and the SGI Power Challenge series. These systems employ high-speed commodity microprocessors and provide a hardware cache-coherent memory architecture. This section describes the features of these multiprocessors.

Earlier experimental results (reported by Manjikian and Abdelrahman [MA97]) were also obtained on Kendall Square Research KSR1 and KSR2 multiprocessors [Ken91]. These results are not included in this chapter because their conclusions are the same as those from the results obtained on the faster and more recent Convex and SGI multiprocessors.

7.2.1 Hewlett-Packard/Convex SPP1000 and SPP1600

The Hewlett-Packard/Convex SPP1000 multiprocessor [Con94] consists of up to 16 *hypernodes*, each containing 8 processors with a crossbar connection to 512 Mbytes of common memory, as shown in Figure 7.2. The crossbar provides uniform access to the local memory for processors within a hypernode. Each processor is a Hewlett-Packard PA7100 RISC microprocessor running at 100 MHz with separate 1-Mbyte instruction and data caches [DWYF92]. The caches are direct-mapped and virtually-indexed, hence cache partitioning *must* be used for conflict avoidance. The cache access latency is 1 clock cycle or 10 nsec, and the cache line size is 32 bytes. Hypernodes are connected together with the Coherent Toroidal Interconnect (CTI), a system of rings based on the SCI standard interconnect, clocked at 250 MHz. The CTI permits processors to access memory in any hypernode through coherent global shared memory.

The Convex SPP1000 is a non-uniform memory access (NUMA) multiprocessor. Cache misses to retrieve data from the local hypernode memory incur a nominal latency of 40 cycles, or 400 nsec. However, misses to retrieve data from remote hypernode memory through the CTI incur a larger latency of approximately 200 cycles, or 2 μsec. A unique feature of the Convex SPP1000 is the CTIcache, which is a portion of the memory in each hypernode reserved for caching data from other hypernodes in order to reduce the effective memory latency for remote memory accesses. Remote data is retrieved in units of 64 bytes, but supplied to

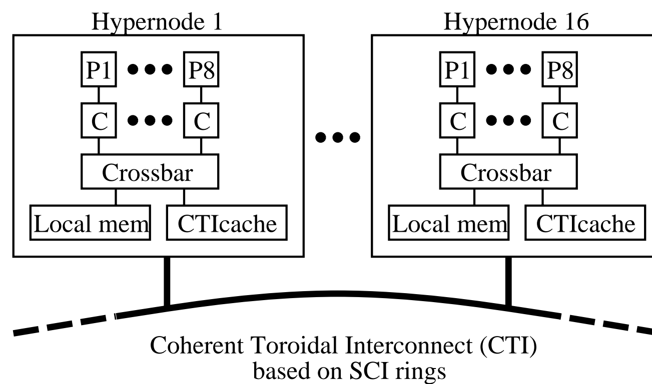


Figure 7.2: Architecture of the Convex SPP1000

processors in 32-byte cache lines from the CTIcache (i.e., processors do cache remote data). The remote memory access latency is incurred once to load data into the CTIcache, and subsequent accesses by any processor that hit in the CTIcache incur the same access latency as the local memory, i.e., 40 cycles instead of 200 cycles. The Convex SPP1000 provides hardware monitoring for accurate measurement of the number of cache misses and the corresponding latencies to local and remote memory.

The Convex SPP1600 is an enhancement of the SPP1000 to provide higher performance. In the Convex SPP1600, each processor is a Hewlett-Packard PA7200 RISC microprocessor [CHK+96] running at 120 MHz, rather than a PA7100 microprocessor running at 100 MHz in the SPP1000. In addition to a faster clock rate, the PA7200 microprocessor incorporates three major enhancements over the PA7100. First, there is an additional integer execution unit to permit dual issue of integer instructions (integer and floating-point instructions are dual-issued on both microprocessors). Second, a 2-Kbyte fully-associative *assist* cache supplements the 1-Mbyte direct-mapped data cache for the PA7200. The assist cache holds data that conflicts with data in the main cache. Third, the PA7200 provides hardware-initiated prefetching. On a cache miss for a normal memory access, the PA7200 issues a prefetch request for the cache line adjacent to the missed cache line. Prefetching with arbitrary stride is also supported by exploiting a feature of the instruction set [CHK+96]. The HP/Convex native compiler generates machine code using memory instructions that automatically increment the contents of an offset register for array references in the body of a loop. Whenever a cache miss occurs for such

instructions, the hardware also issues a prefetch request using the autoincrement value as the prefetch stride. A prefetched cache line is marked with a special tag as it is loaded into the cache. On the first reference to a prefetch-tagged cache line using the autoincrement memory instruction, the hardware issues a new prefetch request.

The SPP1600 also uses a four-state cache coherence protocol instead of the three-state protocol in the SPP1000. The additional state for the SPP1600 is a *clean-exclusive* state that avoids a cache miss to obtain write permission for a given cache line when there are no other cached copies of the cache line. An example of code that benefits from this enhancement is a statement such as `A[i] = A[i] + 1` appearing in the body of a loop within index variable `i`. To perform the computation in this statement, a read cache miss is first incurred to load `A[i]` into the cache. On the SPP1000, an additional coherence miss is then needed to obtain permission from the memory to modify `A[i]`. On the SPP1600, the second miss is avoided by reading the cache line in the clean-exclusive state; the write is performed in the cache and the state changes to dirty-exclusive without requiring a memory reference.

Apart from the higher speeds and additional features provided by the PA 7200, the architecture of the Convex SPP1600 is otherwise the same as the Convex SPP1000.

7.2.2 Silicon Graphics Power Challenge R10000

Experiments were conducted on an SGI Power Challenge multiprocessor consisting of superscalar MIPS R10000 microprocessors [Sil96b]. The Power Challenge is a bus-based, uniform memory access (UMA) multiprocessor. The bus has a wide datapath of 256 bits and operates at 47.6 MHz for an available bandwidth of over 1 Gbyte/sec. The bus supports up to 9 processor boards, each containing 4 microprocessors that share a common interface to the system bus. The shared memory is interleaved in units of cache lines to allow multiple outstanding requests to be serviced concurrently. Up to 8 memory boards may be connected to the bus, for a maximum memory of 16 Gbytes.

Each R10000 microprocessor runs at 196 MHz and issues up to 4 instructions in each clock cycle. The R10000 has separate on-chip 32-Kbyte caches for instructions and data, and a 1-Mbyte external cache that is physically-indexed. All caches are 2-way set-associative, and the external cache line size is 128 bytes. The R10000 supports software-controlled prefetching of

cache lines into the external cache using a dedicated prefetch instruction. The native compiler automatically inserts and schedules prefetch instructions into the optimized executable code, and also provides a flag to disable this feature. By disabling prefetching, its performance impact can be measured.

The R10000 also provides two internal counters that may be configured to count a variety of events, such as the number of issued instructions or the number of cache misses. Unfortunately, these counters cannot measure latency. The `perfex` [ZLTI96] software tool is used to select the events to be counted during the execution of a given program. When the program being measured terminates, `perfex` reports the accumulated event counts to the user.

Although the Power Challenge does not, strictly speaking, have a scalable architecture, the R10000 microprocessors it employs are also used in the scalable SGI/Cray Origin multiprocessor [Sil96a]. Measurements indicate that the sustained memory bandwidth for the Origin is comparable to the Power Challenge [McC]. Hence, the performance obtained on the Power Challenge should reflect the expected performance on a comparable Origin multiprocessor.

7.3 Codes Used in Experiments

Table 7.1 lists the codes used to evaluate the techniques proposed in this dissertation. The codes are divided into two categories: kernels and applications. The kernels are excerpted codes of manageable size for detailed study. The applications are complete codes that provide an indication of the true performance impact of the proposed techniques for representative programs. For the purposes of experimentation and overcoming limitations of the prototype compiler implementation, certain modifications were performed to the code. These changes are briefly described below.

The selected applications originate from uniprocessor environments, hence the problem sizes reflect the limitations of uniprocessor execution. Since it is reasonable to expect that larger problem sizes justify in part the need for multiprocessor execution, array sizes were correspondingly increased in order to justify the need for locality enhancement in a multiprocessor environment. Array sizes were also decreased in some experiments to permit data to fit in caches and hence measure the instruction overhead of the locality-enhancing transformations.

Where necessary, arrays in the applications were collected into COMMON blocks to facil-

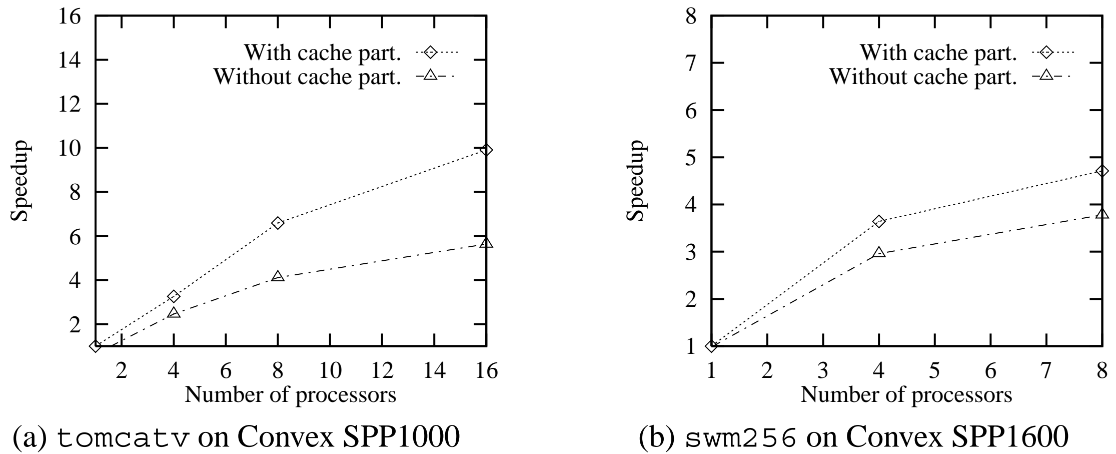
Table 7.1: Kernels and applications for experimental results

Name	Description	Lines of code
SOR	kernel of loops for PDE solver	8
Jacobi	kernel of loops for PDE solver	11
LL18	kernel from Livermore Loops	24
calc	kernel from qgbox [McC92] ocean model	186
filter	subroutine in hydro2d	247
tomcatv	SPEC95 benchmark (mesh generation)	190
swm256	SPEC92 benchmark (shallow water equations)	487
hydro2d	SPEC95 benchmark (Navier-Stokes)	4292
spem	ocean circulation model [Hed94]	26937

itate cache partitioning for conflict avoidance. All of the arrays in `tomcatv` had to be placed in a COMMON block. In the remaining applications, most of the arrays were already in COMMON blocks. For the `hydro2d` application, however, many COMMON block declarations were inconsistent across subroutines in the original code. Compilers may not be able to enforce consistency because of memory aliasing, hence the COMMON blocks were restructured for consistent usage throughout the program. The usage of COMMON blocks in the `spem` application was much more consistent, but minor changes were still applied; specifically, some local automatic arrays were incorporated into COMMON blocks.

To increase the length of the candidate loop nest sequence in `tomcatv` for the shift-and-peel transformation, a modification suggested by Lebeck and Wood [LW94] was applied. This modification reorders the loop nests in `tomcatv` to increase the number of adjacent, compatible loop nests. Although Lebeck and Wood apply loop fusion to these loop nests, they target uniprocessors only; they do not address the serializing dependences that are present in the fused loop nest sequence. Furthermore, there are also fusion-preventing dependences in this loop nest sequence after reordering the loop nests. Lebeck and Wood fuse the loops directly, but this violates the original program semantics. In contrast, our shift-and-peel transformation ensures that the fusion is legal and that the resulting loop nest may still be parallelized.

Finally, all of the applications considered in this study are iterative in nature, consisting of a main loop that repeatedly executes the core computation of the application. Because there is little or no variance in the computation across successive iterations of the main loop, the number



(a) `tomcatv` on Convex SPP1000 (b) `swm256` on Convex SPP1600

Figure 7.3: Speedups for cache partitioning alone on Convex multiprocessors

of iterations of the main loop was reduced in the larger applications. This modification was required to reduce the time for experiments with restricted access to dedicated multiprocessor systems without interference from other jobs.

7.4 Effectiveness of Cache Partitioning

This section provides results to demonstrate the importance of avoiding cache conflicts. Figure 7.3 illustrates the parallel speedup with and without cache partitioning on the Convex multiprocessor for two applications, `tomcatv` and `swm256`. No locality-enhancing loop transformations are used in these experiments; the difference in performance is attributable solely to the data layout. All speedups in Figure 7.3 are calculated with respect to the execution time for the cache-partitioned code on one processor, hence the increase in speedup for a given number of processors also represents an improvement in absolute performance. Both of these applications display extreme sensitivity to the occurrence of conflicts because the array dimensions are very close to powers of two (513×513 for `tomcatv` and 257×257 for `swm256`). Even the assist cache in the SPP1600 is not sufficient to avoid undesirable conflict misses. Cache partitioning improves performance at 8 processors by 25% in Figure 7.3(b). The memory overhead from cache partitioning is 7% for `tomcatv` and 13% for `swm256`.

The next set of results compare cache partitioning with array padding when loop fusion is applied. The measured number of cache misses on one processor during parallel execution of

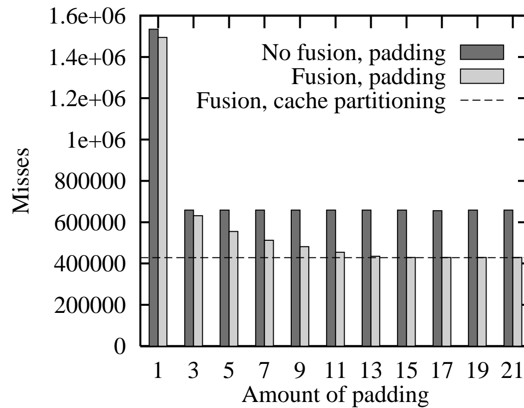
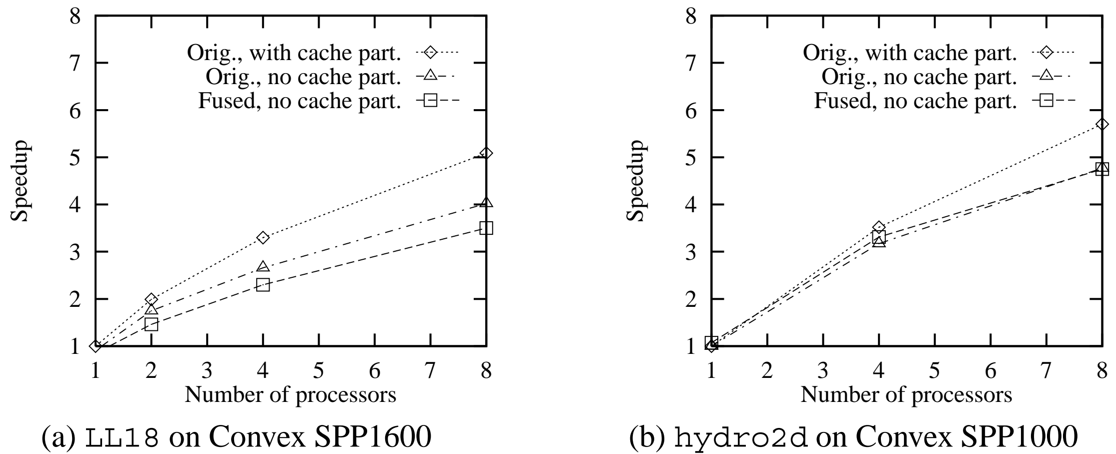


Figure 7.4: Cache partitioning *vs.* array padding for `LL18` on Convex SPP1000

the `LL18` kernel on 8 processors are shown in Figure 7.4. The array size is 1024×1024. The number of misses obtained for various amounts of padding within array dimensions (shaded bars) is compared to the number of misses obtained from applying cache partitioning across the arrays (dashed line). Padding does not guarantee the elimination of all conflicts; it is difficult to predict the amount of padding needed to achieve the smallest number of misses. In contrast, cache partitioning directly results in the smallest number of misses. The memory overhead from cache partitioning is under 2% in this case.

Figure 7.5(a) illustrates the parallel speedup with and without cache partitioning on the Convex SPP1600 for the `LL18` kernel. The speedups are shown for the original code and the code with fusion enabled by shift-and-peel. Again, all speedups are computed with respect to the execution time for the original code with cache partitioning on one processor, hence the increase in speedup for a given number of processors also represents an improvement in absolute performance. There are two features to note in Figure 7.5(a). First, the speedup for the original code is higher with cache partitioning than without it. Second, the speedup of the fused version of the code without cache partitioning is worse than the speedup of the original version without cache partitioning. In other words, cache conflicts are negating any potential performance benefit from fusion. This loss of performance is occurring despite the presence of the assist cache in the SPP1600.

Finally, Figure 7.5(b) illustrates the importance of conflict avoidance when fusing loop nests in the `hydro2d` application. All speedups are computed with respect to the execution time for



(a) LL18 on Convex SPP1600 (b) hydro2d on Convex SPP1000

Figure 7.5: Impact of cache partitioning with fusion on Convex SPP1000/SPP1600

the original code with cache partitioning on one processor in order to show improvements in absolute performance for a given number of processors. This application is not as sensitive to conflicts as the applications in Figure 7.3. Nonetheless, the occurrence of conflicts still renders loop fusion ineffective. The memory overhead is 12% in this case.

Given the importance of conflict avoidance, particularly when enhancing locality with the shift-and-peel transformation for loop fusion as in Figures 7.4 and 7.5, *the remaining experimental results presented in this chapter include cache partitioning for both the original and enhanced code*. Since cache partitioning improves the performance of the original program, the reported improvements in performance reflect the benefit of enhancing locality in the absence of conflicts, and represent a lower bound on the improvement in performance.

7.5 Effectiveness of the Shift-and-peel Transformation

This section presents results that demonstrate the effectiveness of the shift-and-peel transformation for the kernels and applications shown in Table 7.2. For each kernel or application, Table 7.2 provides the number of loop nest sequences to which the shift-and-peel transformation was applied, as well as the length of the longest sequence and the maximum shift/peel amounts for any sequence. The loop nests of interest are analyzed and transformed using the prototype compiler implementation discussed in Section 7.1. The only exception is the `calc` loop nest sequence that was analyzed and transformed manually because the inner loops in two of the

Table 7.2: Kernels and applications used in experiments for the shift-and-peel transformation

Name	Base array size	Total data (Mbytes)	Lines of code	Uniproc. time on SPP1000	Number of loop sequences	Loops in longest sequence	Maximum shift/peel
`Jacobi`	400×400	2.4	11	—	1	2	1/1
`LL18`	1024×1024	72	24	3.99 s	1	3	2/1
`calc`	1024×1024	48	186	3.02 s	1	5	3/3
`filter`	1602×640	63	247	8.31 s	1	10	5/4
`tomcatv`	513×513	16	190	132 s	1	3	1/1
`hydro2d`	802×320	50	4292	3820 s	3	10	5/4
`spem`	$60 \times 65 \times 65$	70	26937	1197 s	11	8	1/2

loop nests prevent the current implementation from obtaining dependence distances.

For reference, Table 7.2 also provides uniprocessor execution times on the Convex SPP1000 multiprocessor for the unfused code with cache partitioning. These times are used to calculate speedups for the experimental results. For each of the kernels, the reported time in Table 7.2 is for one iteration of the kernel code, with initialization time excluded. However, for the applications, the times include initialization and many iterations of the main loop in each application. For `tomcatv` and `hydro2d`, the uniprocessor times are for 100 iterations of the main loop, and the time for `spem` is for 50 iterations of the main loop. All uniprocessor times (as well as the multiprocessor times to be reported later in this section) reflect elapsed real time that is measured on dedicated systems to minimize any variability caused by interference from other programs.

The results to be presented in this section are organized as follows. First, the improvements in performance provided by the shift-and-peel transformation will be demonstrated for the kernels, supported by measurements of cache behavior to explain the observed improvements. Measured improvements in performance will also be compared with estimated improvements obtained with the model presented in Chapter 3. Second, the benefit of the shift-and-peel transformation will be demonstrated for parallel execution of application programs. Third, the benefit of combining the shift-and-peel transformation with data prefetching will be considered.

7.5.1 Results for Kernels

The presentation of the experimental results for the kernels is organized as follows. First, the derived amounts of shifting and peeling will be presented to demonstrate the need for the

Table 7.3: Amounts of shifting and peeling for kernels

	LL18		calc		filter	
Loop	shifts	peels	shifts	peels	shifts	peels
1	0	0	0	0	0	0
2	1	0	0	0	0	0
3	2	1	2	2	0	0
4			3	3	1	1
5			3	3	2	2
6					2	2
7					3	3
8					4	4
9					4	4
10					5	4

shift-and-peel transformation. Second, performance results will be provided, and the measured performance improvements will be compared with the estimated improvements obtained with the model proposed in Chapter 3. Third, the overhead of the shift-and-peel transformation will then be characterized by measuring the effect of reducing problem sizes such that data fits in caches. Finally, the performance obtained with the shift-and-peel transformation will also be compared to the performance obtained with the alignment/replication techniques of Callahan [Cal87] and Appelbe and Smith [AS92].

7.5.1.1 Derived Amounts of Shifting and Peeling

The amounts of shifting and peeling required to fuse the outermost loops of the kernels are given in Table 7.3. Shift-and-peel is indeed required in order to apply fusion legally across *all* loops. Furthermore, shift-and-peel is required to enable parallel execution of the fused loops. The complexity of the dependence relationships across these representative loop nest sequences requires a systematic approach to automate the derivation and application of the shift-and-peel transformation. For example, the dependence chain multigraph for `filter` contains 149 edges from which the shift and peel amounts in Table 7.3 are derived.

It should be noted that the dependences in these kernels necessitate replication with the techniques proposed by Callahan [Cal87] and Appelbe and Smith [AS92] because alignment

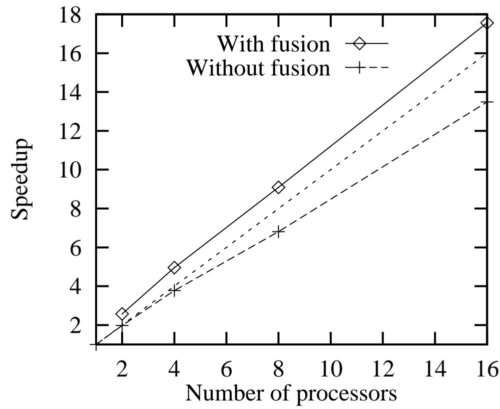
conflicts exist (see Section 4.2.5). In contrast, our technique does not require any replication.

7.5.1.2 Multiprocessor Speedups

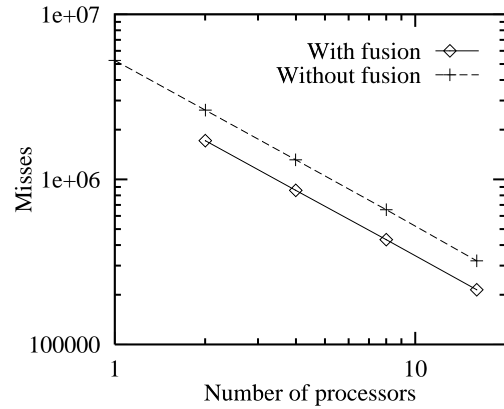
Figure 7.6 shows the parallel speedups and measured cache misses for the fused and unfused versions of three of the kernels on the Convex SPP1000. Array sizes were 1024×1024 for `LL18` and `calc`, and 1602×640 for `filter`. For each kernel, speedups are computed with respect to the execution time without fusion on one processor, hence the increase in speedup for a given number of processors represents an improvement in absolute performance. Furthermore, superlinear speedups may be expected from this choice of reference for speedups. Cache partitioning is used in all the experiments. Misses are measured on one processor and are representative for all processors used in parallel execution. Fusion improves performance by at least 30% for `LL18` and `calc`, and by 60% for `filter`. These improvements are attributable entirely to enhanced locality, as evidenced by the reduction in the number of cache misses for a given number of processors. Because misses are shown on a logarithmic graph in Figure 7.6, the constant slopes reflect a constant ratio for the number of misses before and applying fusion at a given number of processors. The larger the reduction in the number of cache misses, the larger the improvement in performance.

7.5.1.3 Impact of Problem Size on the Improvement from Fusion

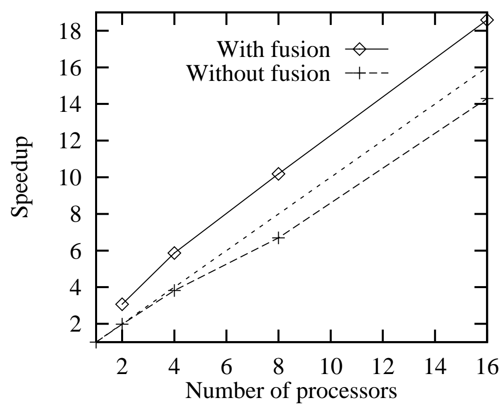
To study the impact of problem size with respect to cache size on the improvement from fusion, the array sizes in `LL18` and `calc` were varied. The results for the Convex SPP1000 are shown in Figure 7.7. The horizontal axes in the graphs represent different array sizes, and the vertical axes represent the performance improvement from fusion, which is computed as the ratio of parallel execution times of the original loops and fused loops, respectively. Any point above the reference line at 1 indicates that fusion improves performance. As before, cache partitioning is used throughout, hence any improvements represent lower bounds. Figure 7.7(a) indicates that with 8 processors, the two larger array sizes are such that the data does not entirely fit in the cache, hence fusion improves performance. With 16 processors, the total cache capacity is doubled, and Figure 7.7(b) indicates that even the 512×512 array size permits data to fit in the cache for `calc`, hence fusion does not improve performance. Note that because `LL18`



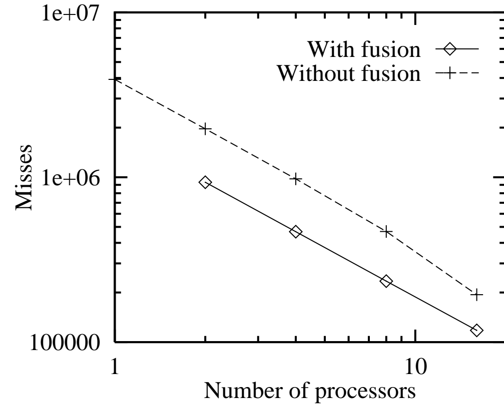
(a) LL18 speedup



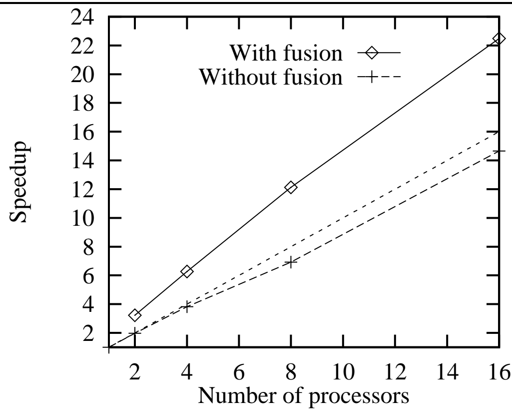
(b) LL18 misses



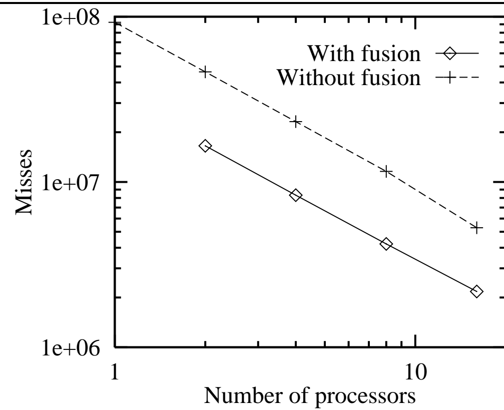
(c) calc speedup



(d) calc misses

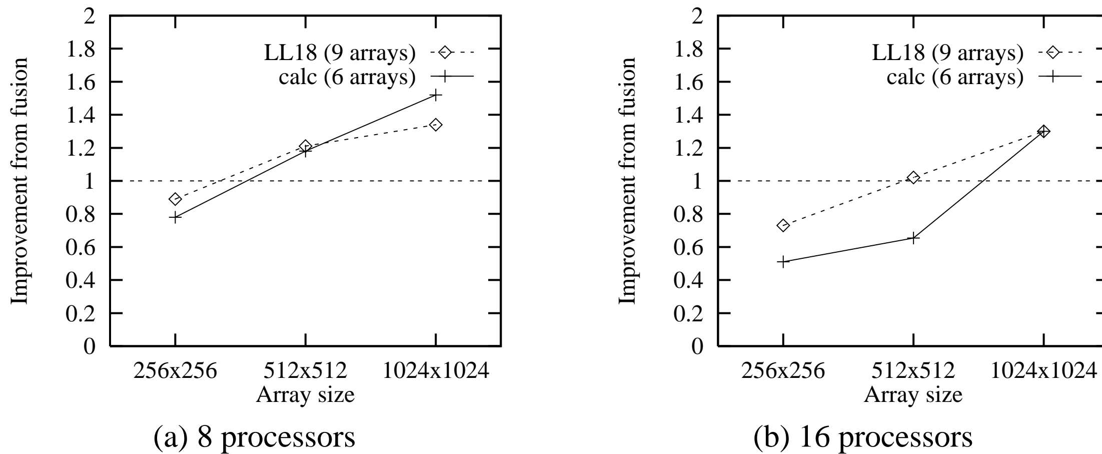


(e) filter speedup



(f) filter misses

Figure 7.6: Speedup and misses of kernels on Convex SPP1000



(a) 8 processors (b) 16 processors

Figure 7.7: Improvement in speedup with fusion for LL18 and calc on Convex SPP1000

has nine arrays of the same size whereas calc has only six, fusion still improves performance at 16 processors for LL18 when the array size is 512×512 because all the data cannot simultaneously fit in the caches. These observations suggest using knowledge of data sizes and cache sizes to determine the profitability of applying the shift-and-peel transformation for locality enhancement. A compiler can include both the original and transformed versions of a loop nest sequence in executable code, with a run-time decision to select the appropriate version based on the amount of data accessed per processor.

It must be stressed that the shift-and-peel transformation is applied at the level of the source code in these experiments. The transformed source code is then passed to the native compiler. The ability of the native compiler to optimize more complex loop structures determines the efficiency of the resulting executable code. Faced with more complex code, the compiler is less aggressive in its optimizations, which then increases the instruction overhead. It is reasonable to expect that integrating the shift-and-peel transformation within a native compiler framework should result in better performance by permitting more aggressive optimizations to be performed in conjunction with loop fusion.

7.5.1.4 Comparison of Shift-and-peel with Alignment/replication

The shift-and-peel transformation avoids the overhead that results from the alignment and replication techniques proposed by Callahan [Cal87] and Appelbe and Smith [AS92]. Figure 7.8

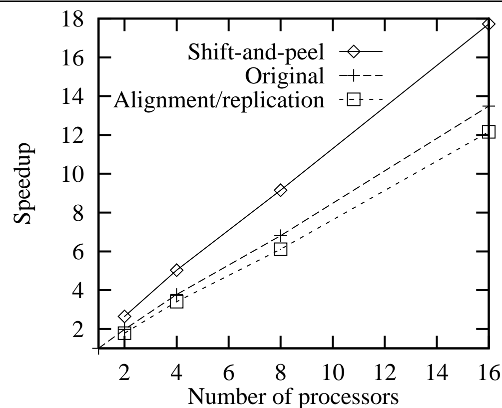


Figure 7.8: Performance of shift-and-peel *vs.* alignment/replication for `LL18`

compares the speedup of the fused `LL18` loop nests parallelized using shift-and-peel with the speedup of the fused loop nest parallelized using direct application of alignment and replication. For the latter case, it was necessary to replicate two arrays and two statements for parallelization. All speedups are computed with respect to the execution time for the *original* loop nest sequence on one processor, hence higher speedup also indicates better absolute performance. The figure clearly indicates that superior performance is achieved with shift-and-peel by avoiding the overhead associated with the replication of code and data.

7.5.2 Comparing Measured Performance Improvements with the Model

This section compares the measured performance improvements for the kernels with estimates obtained with the model discussed in Chapter 3. The following paragraphs determine sweep ratios and the memory fraction of execution time in order to apply the model.

7.5.2.1 Determining the Sweep Ratios

Table 7.4 characterizes each loop nest kernel in terms of the number of arrays read from or written to memory both before and after fusion. Because all arrays have the same size, memory accesses can be quantified in terms of sweeps through memory for the arrays, as discussed in Section 3.2. Assuming that the cache capacity is not sufficient to hold all of the data referenced across the loop nests, locality enhancement with fusion is required, and a reduction in the number of memory accesses is expected when fusion is applied.

Table 7.4: Characteristics of loop nest kernels

Kernel	Num. loop nests	Num. arrays	Sweep statistics				Sweep ratio	
			before fusion		after fusion		with writes	without writes
			read	write	read	write		
Jacobi	2	2	4	2	2	2	1.50	2.00
calc	5	6	13	6	6	5	1.73	2.17
LL18	3	9	16	6	9	6	1.47	1.78
filter	10	8	33	14	8	6	3.36	4.13

```
DO K= 2, N-1
   DO J= 2, N-1
     ZA[J,K]= (ZP[J-1,K+1]+ZQ[J-1,K+1]-ZP[J-1,K]-ZQ[J-1,K])
             *(ZR[J,K]+ZR[J-1,K])/(ZM[J-1,K]+ZM[J-1,K+1])
     ZB[J,K]= (ZP[J-1,K]+ZQ[J-1,K]-ZP[J,K]-ZQ[J,K])
             *(ZR[J,K]+ZR[J,K-1])/(ZM[J,K]+ZM[J-1,K])
   END DO
 END DO
 DO K = 2, N-1
   DO J = 2, N-1
     ZU[J,K]= ZU[J,K]+S*(ZA[J,K]*(ZZ[J,K]-ZZ[J+1,K])
                     -ZA[J-1,K] *(ZZ[J,K]-ZZ[J-1,K])
                     -ZB[J,K]   *(ZZ[J,K]-ZZ[J,K-1])
                     +ZB[J,K+1] *(ZZ[J,K]-ZZ[J,K+1]))
     ZV[J,K]= ZV[J,K]+S*(ZA[J,K]*(ZR[J,K]-ZR[J+1,K])
                     -ZA[J-1,K] *(ZR[J,K]-ZR[J-1,K])
                     -ZB[J,K]   *(ZR[J,K]-ZR[J,K-1])
                     +ZB[J,K+1] *(ZR[J,K]-ZR[J,K+1]))
   ENDDO
 ENDDO
 DO K = 2, N-1
   DO J = 2, N-1
     ZR[J,K]= ZR[J,K]+T*ZU[J,K]
     ZZ[J,K]= ZZ[J,K]+T*ZV[J,K]
   ENDDO
 ENDDO
```

Figure 7.9: Code for LL18 loop nest sequence

When arrays have the same size, the sweep ratio can be used to express the reduction in the number of memory sweeps, as discussed in Section 3.3. Equation 4.1 in Section 4.1.2 computes the sweep ratio for fusion using both reads and writebacks to assess the benefit of reducing all memory accesses. However, in contemporary systems, the latency for writebacks

on replacement is often hidden by buffering the replaced cache line and initiating the read for the new cache line first. The writeback to memory is performed while the read response is being forwarded to the requesting processor. In this case, the effective sweep ratio is computed by excluding the writebacks, which results in a larger sweep ratio. For comparison, Table 7.4 provides both variants of the sweep ratio. Clearly, there is significant potential for locality enhancement because the sweep ratios are significantly larger than 1.

To understand how the sweep statistics in Table 7.4 are obtained, consider the code for the `LL18` kernel shown in Figure 7.9. The first loop nest in Figure 7.9 references 6 arrays, the second loop nest references 6 arrays, and the third loop nest references 4 arrays. Assuming that the data does not remain cached between loop nests, a total of 16 arrays will be read from memory into the cache. Similarly, arrays are modified 6 times across the loop nests, hence there will be 6 arrays written back to memory as new data is loaded into the cache. When the loops are fused, each array should only be read once from memory and reused as needed from the cache, for a total of only 9 arrays. The number of arrays written back remains at 6.

The remaining statistics in Table 7.4 are obtained in a similar manner. It may be noted that fusion reduces the number of effective number of writebacks for the `calc` and `filter` kernels. As discussed in Section 4.1.2, this reduction occurs when the same array is written in more than one of the original loop nests being fused. With fusion, multiple writes to the same array are performed in the cache, and only one writeback is performed to memory.

There is one additional factor that must be considered when quantifying the number of memory accesses. The Convex SPP1000 employs a three-state cache coherence protocol that generates additional coherence misses. These *upgrade* misses [PH96] occur when a cache line is loaded into the cache in a read-only state, and later written while it is still cached. The write may not proceed without an additional cache miss to upgrade the state of the line to exclusive-modified, even if no other cached copies exist. For the Convex SPP1000, the effective round-trip latency for the upgrade request, even without a data response, is essentially the same as a normal cache miss because the memory directory must be accessed.

Upgrade misses occur when an array is both read and written in a loop body, with the read occurring before the write. For the `LL18` kernel in Figure 7.9, there are four such references for arrays ZU, ZV, ZR, and ZZ. Hence, there are 4 upgrade references in addition to the 16

Table 7.5: Revised sweep ratios to account for upgrade requests

Kernel	Sweep statistics				Sweep ratio (reads and upgrades only)
	before fusion		after fusion		
	read	upgrade	read	upgrade	
`Jacobi`	4	0	2	1	1.33
`calc`	13	2	6	1	2.14
`LL18`	16	4	9	4	1.54
`filter`	33	6	8	4	3.25

Table 7.6: Cache misses for parallel execution on Convex SPP1000

Kernel	Number of processors	Original		Fused	
		expect.	meas.	expect.	meas.
`LL18`	2	2621440	2629000	1703936	1717110
	16	327680	320954	212992	214546
`calc`	2	1966080	1965820	917504	932760
	16	245760	194001	114688	115443
`filter`	2	4998240	4633480	1537920	1653140
	16	624780	528052	192240	217479

references in the original loop nest sequence. Even when the loops are fused, the reads still precede the writes, hence there are still 4 upgrade requests. A similar analysis for the other kernels results in the revised statistics given in Table 7.5. If the ratios in Table 7.5 are compared with Table 7.4, it is clear that upgrade misses reduce the ratios.

Using the sweep statistics in Table 7.5, it is possible to verify that the expected number of cache misses are being incurred in parallel execution. Given the number of read and upgrade sweeps, the array size (number of elements), the cache line size, and the number of processors, it is possible to compute the expected number of cache misses per processor as follows:

$$\#\text{cache misses} = \frac{(\#\text{sweeps}) \cdot (\text{array_size})}{(\text{cache_line_size}) \cdot (\#\text{processors})}. \tag{7.1}$$

The cache line size on the Convex SPP1000 is 32 bytes, which is equivalent to 4 array elements using 8-byte floating point values.

Using the above formula, the expected number of cache misses per processor and the measured number of cache misses are compared in Table 7.6 for each of the kernels in parallel

Table 7.7: Comparison of estimated and measured improvement from fusion

Kernel	Measured f_m before fusion	Sweep ratio (without writes)	Perf. improvement with fusion	
			estimated	measured
LL18 (2 proc.)	0.53	1.54	23%	30%
LL18 (16 proc.)	0.56		24%	30%
calc (2 proc.)	0.52	2.14	38%	55%
calc (16 proc.)	0.49		35%	32%
filter (2 proc.)	0.44	3.25	44%	63%
filter (16 proc.)	0.45		45%	53%

execution at 2 and 16 processors. There is agreement between the expected and measured results in the majority of cases. There are only two significant discrepancies, and both are easily explained. For `calc` and `filter`, the measured number of cache misses is significantly lower than the expected number of misses at 16 processors because the data is beginning to fit in the cache, hence there is a reduction in the number of capacity misses across loop nests in the original loop nest sequences. Nonetheless, fusion of the loop nests provides a much larger reduction in the number of cache misses at 16 processors for both kernels. Furthermore, cache conflict avoidance ensures that the full benefit of fusion is obtained.

7.5.2.2 Determining f_m and Applying the Model

For the purpose of estimating the performance improvement from fusion using Equation 3.2 in Chapter 3, a reasonably-accurate value for f_m (the memory fraction of execution time) is required. This value may be obtained using the hardware monitoring features of the Convex SPP1000. The accumulated cache miss latency on one processor can be measured for an execution of the original loop nests before fusion, and this quantity may be divided by the execution time for the original loop nests to obtain an estimate of f_m.

Table 7.7 compares the estimated and measured improvements in performance from fusion for each of the kernels on 2 and 16 processors of the Convex SPP1000. The measured improvements are computed from the execution times $T(p)$ from parallel execution on p

processors,

$$\left(\frac{T_{original}(p)}{T_{fused}(p)} - 1\right) \times 100\%,$$

where $p = 2$ or $p = 16$. The estimated improvements are obtained by substituting the measured f_m values and the sweep ratios from Table 7.7 into Equation 3.2, then converting to a percentage,

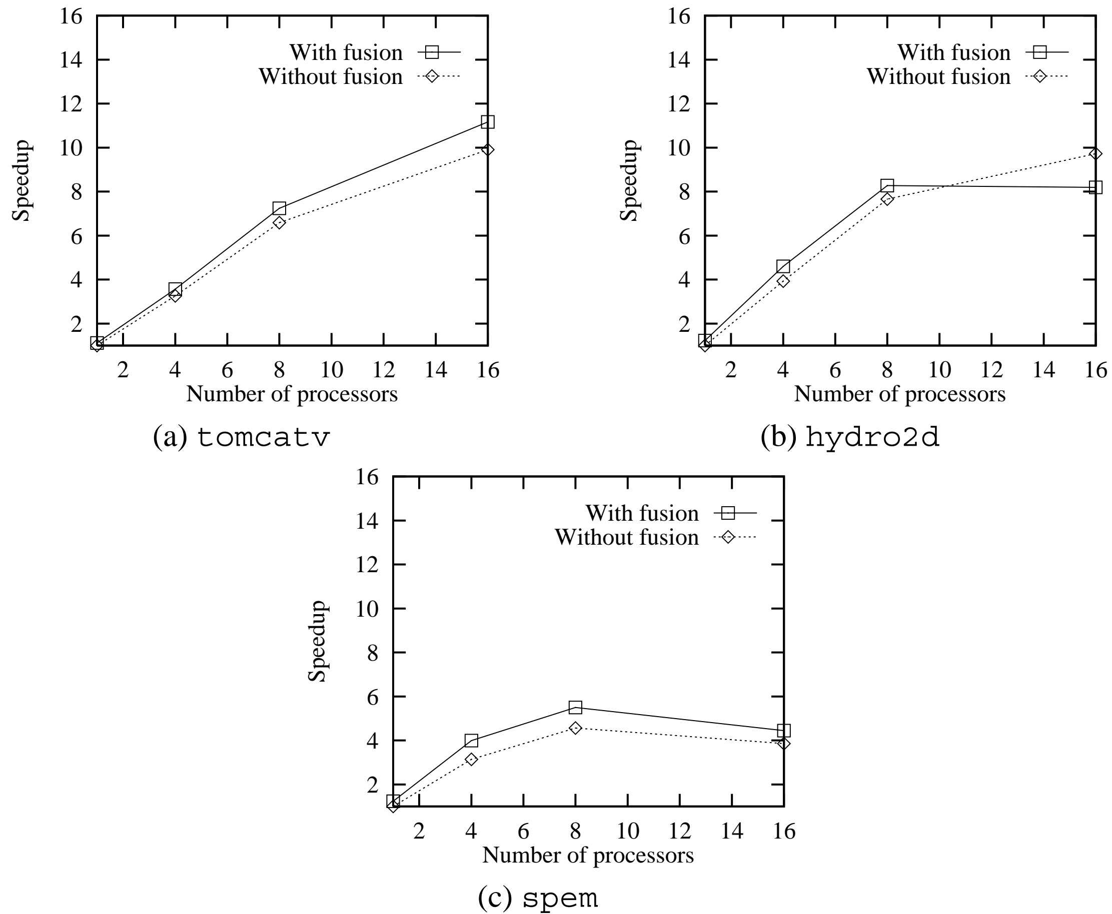
$$\left(\frac{1}{(1 - f_m) + f_m/r_m} - 1\right) \times 100\%.$$

Because the workload is balanced, the improvement applies equally to all processors in parallel execution.

The intent of the comparison in Table 7.7 is to demonstrate that the measured performance improvements are meaningful with respect to the expected benefit embodied in the sweep ratio. In all but one case, the improvements obtained with the model *underestimate* the measured improvements. For `calc` at 16 processors, the measured improvement is less than the estimate because reused data begins to fit in the combined cache capacity when executing the original loop nest sequence, as discussed earlier (see Table 7.6). The underestimated improvement for the remaining cases in Table 7.7 can be explained by pointing out that the HP PA7100 microprocessors used in the SPP1000 implement *hit-under-miss* [DWYF92], which means that processors do not stall immediately on a cache miss. Hit-under-miss allows a processor to continue executing after an initial cache miss until the missing data is actually needed by a subsequent instruction, or until a second cache miss occurs. This feature provides a small degree of concurrency between computation and memory access, though certainly not as much as full prefetching. Recall that each 32-byte cache line contains only four array elements. Prior to fusion, the opportunities for hit-under-miss are limited by the fact that one of every four unique array references in each loop nest incurs a cache miss. Reducing the number of cache misses with fusion increases the effectiveness of hit-under-miss by allowing more instructions to be executed before a stall is required.

7.5.3 Results for Applications

The benefit of the shift-and-peel transformation with cache partitioning for applications on the Convex SPP1000 is shown in Figure 7.10. For `tomcatv`, the array size is 513 $\times$ 513, and the total data size is 16 Mbytes. For `hydro2d`, the array size is 802 $\times$ 320, and the total



(a) tomcatv

(b) hydro2d

(c) spem

Figure 7.10: Speedup for applications on Convex SPP1000

data size is 50 Mbytes. For spem, the array size is 60 × 65 × 65, and the total data size is 70 Mbytes. For each application, the speedups in Figure 7.10 are computed with respect to the execution time of the original code with cache partitioning on one processor, hence the increase in speedup for a given number of processors represents an improvement in absolute performance. For tomcatv, the shift-and-peel transformation improves performance by 10% to 12%. For hydro2d, the improvement is 23% on one processor, and diminishes to 8% on eight processors. At 16 processors, the data begins to fit in the caches, so the overhead of the shift-and-peel transformation degrades the fused performance. The improvement in performance for spem is at least 20% up to eight processors because this application had the largest number of transformed loop sequences, and these sequences constitute close to half of total execution time. However, at 16 processors, remote memory accesses cause

the performance for both the fused and unfused versions to fall below the performance at 8 processors. This behavior results in part from serial loops that the Convex compiler executes on a single processor and in part due to isolated loop nests for which the Convex compiler chooses to apply loop permutation, causing excessive data movement between hypernodes. The loop permutation, in particular, cannot be disabled in the compiler without disabling optimization altogether. Nonetheless, the fused version still provides better performance, and would continue to do so if this behavior could be counteracted with more control over the code produced by the native compiler.

7.5.4 Combining Shift-and-peel with Prefetching

7.5.4.1 Results for Kernels

Uniprocessor speedups These results are obtained on the SGI Power Challenge R10000 where prefetching can be disabled in software. The results are presented for uniprocessor execution to focus initially on the interaction of loop fusion and prefetching.

The uniprocessor kernel speedups on the Power Challenge are shown in Figure 7.11. For each kernel, speedups are determined relative to the execution time without loop fusion or prefetching. The reference times in seconds are 4.67 for `Jacobi` (100 iterations of the kernel), 3.67 for `LL18` (20 iterations), and 3.87 (20 iterations) for `filter`. The array sizes are 400×400 for `Jacobi` and `LL18`, and 402×160 for `filter`; the total data size exceeds the 1-Mbyte cache capacity in all cases. Figure 7.11 confirms that combining fusion with prefetching results in the largest speedup by first reducing the number of memory accesses, then hiding as much of the latency as possible for the remaining memory accesses. Reducing the number of memory accesses makes more memory system bandwidth available and improves the effectiveness of prefetching.

In order to confirm the extent of the reduction in the number of memory accesses, Table 7.8 compares the expected and measured number of cache misses in the external cache for the original and fused loops. The expected number of cache misses is determined with Equation 7.1 in Section 7.5.2 using the number of read sweeps in Table 7.4. The measured number of cache misses is obtained with the `perfex` tool [ZLTI96]. The measurements given in Table 7.8 are obtained without prefetching; measurements with prefetching are similar because prefetch

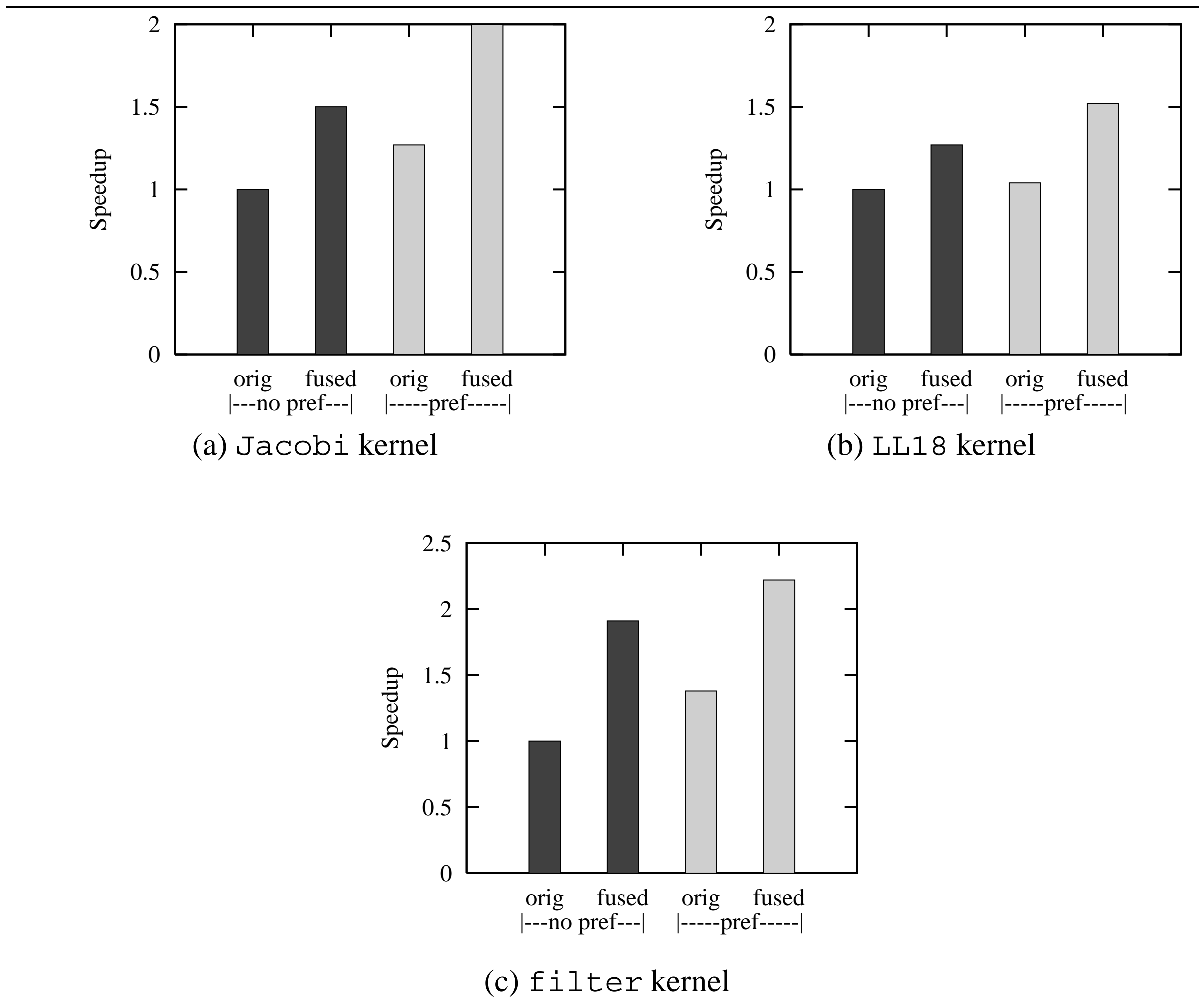


Figure 7.11: Uniprocessor speedups on Power Challenge

Table 7.8: Expected and measured cache misses for uniprocessor execution on Power Challenge

Name	Original		Fused	
	expected	measured	expected	measured
`Jacobi`	40000	39659	20000	19770
`LL18`	160000	163314	90000	89728
`filter`	132660	125132	32160	34811

requests are not distinguished from normal memory requests. The expected and measured values in Table 7.8 for all three kernels agree closely.

Table 7.9: Expected and measured writebacks for uniprocessor execution on Power Challenge

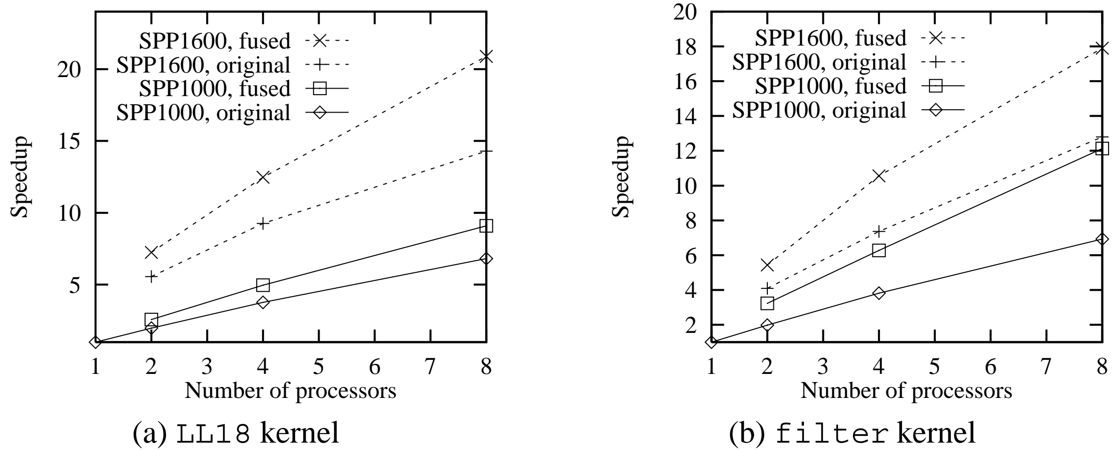
Name	Original		Fused	
	expected	measured	expected	measured
`Jacobi`	20000	19308	20000	19129
`LL18`	60000	58376	60000	57407
`filter`	56280	55252	24120	25183

Similarly, Table 7.9 compares the expected and measured number of writebacks. The measured number of writebacks is obtained with the `perfex` tool [ZLTI96]. The expected number of writebacks is calculated in a manner similar to the expected number of cache misses; the only difference is that the number of written arrays in Table 7.4 is used. Once again, the expected and measured results in Table 7.9 are in close agreement.

Multiprocessor speedups Multiprocessor kernel speedups obtained on the Convex SPP1000 and SPP1600 are presented in this section (parallel speedups for the kernels could not be obtained on the Power Challenge due to restricted access). Figure 7.12 shows the speedups for `LL18` and `filter`. Array sizes are 1024×1024 for `LL18` and 1602×640 for `filter`.

All speedups in Figure 7.12 are relative to the execution time without fusion on one processor of the SPP1000 (see Table 7.2). Hence, higher speedups in Figure 7.12 indicate better absolute performance (i.e., reduced execution time). Figure 7.12 makes a direct comparison of performance on the two multiprocessors, even though the processors used in the two systems are different (see Section 7.2.1). This comparison is made only because prefetching on the SPP1600 is hardware-initiated and cannot be disabled; speedups without prefetching can only be obtained on the SPP1000. The results in Figure 7.12 confirm that hardware prefetching on the SPP1600 combined with fusion provides the best performance. At 8 processors on the SPP1600, fusion improves parallel speedup by approximately 50% for both kernels.

To verify that the expected number of memory accesses are being made by each processor in parallel execution, Table 7.10 compares the expected and measured number of cache misses for the original and fused loops on the SPP1600. The expected number is determined from the sweep statistics in Table 7.4 using Equation 7.1 in Section 7.5.2. The measured number



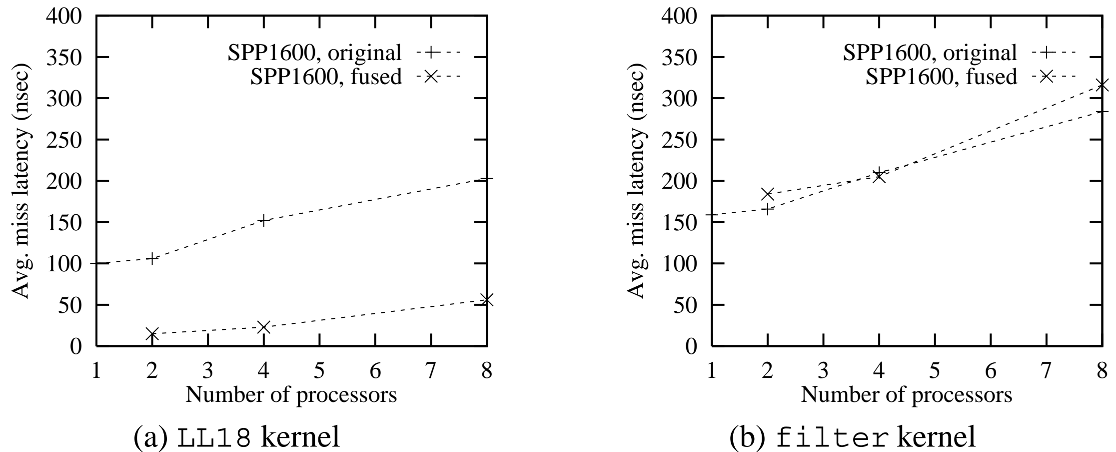
(a) `LL18` kernel (b) `filter` kernel

Figure 7.12: Multiprocessor speedups on Convex SPP1000 and SPP1600 (computed with respect to one processor on Convex SPP1000)

Table 7.10: Cache misses for parallel execution on Convex SPP1600

Name	Number of processors	Original		Fused	
		expect.	meas.	expect.	meas.
`LL18`	2	2097152	2106030	1179648	1192240
	4	1048576	1052390	589824	596249
	8	524288	525588	294912	298289
`filter`	2	4229280	4122660	1025280	1298850
	4	2114640	2059990	512640	653192
	8	1057320	996078	256320	330362

of misses is obtained from the hardware performance monitor. This number includes prefetch requests because they are not distinguished from normal memory requests. Once again, there is close agreement between the expected and measured values in Table 7.10. The expected misses with fusion for `filter` are higher at 4 and 8 processors because the large amounts of shifting and peeling for this kernel (as shown in Table 7.3) contribute a fixed number of cache misses for the peeled iterations that are executed following the barrier in the transformed code (see Section 4.2.4). Because there are no upgrade misses to increase the total number of misses, as in the SPP1000, these additional misses become apparent on the SPP1600 as more processors are used. Nonetheless, the application of fusion with the shift-and-peel transformation does



(a) LL18 kernel (b) filter kernel

Figure 7.13: Average cache miss latencies on Convex SPP1600

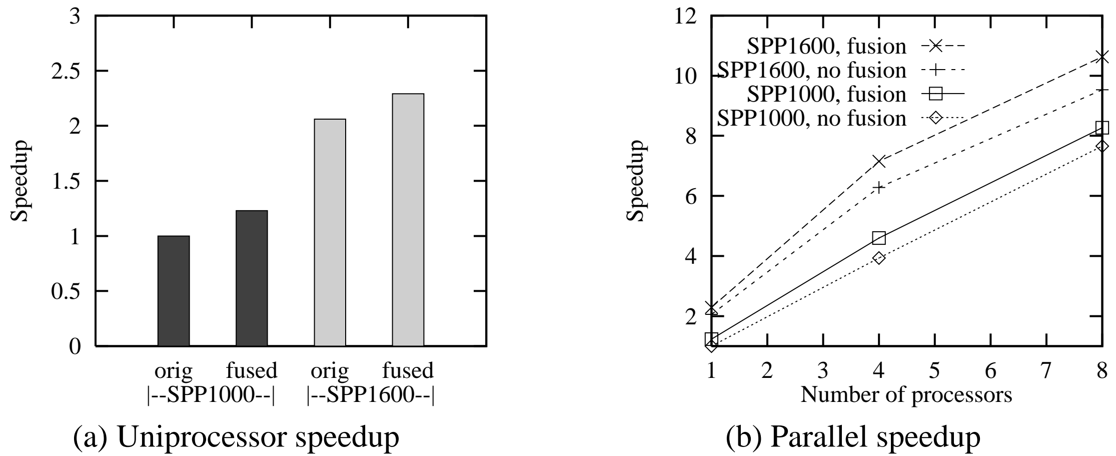
provide a substantial reduction in the total number of misses, with a corresponding improvement in parallel speedup.

The hardware performance monitor was also employed to measure the average *observed* cache miss latency on each processor of the SPP1600. The results for LL18 and filter are shown in Figure 7.13. The nominal miss latency is 400 nsec; hardware prefetching hides a portion of this latency. In all cases, the average latency increases as more processors are used due to the increased load on the memory system. Since the average miss latency with fusion in Figure 7.13(a) is significantly smaller than without it, fusion for LL18 allows hardware-initiated prefetching to hide a larger portion of the memory latency. On the other hand, the average miss latencies for filter do not show a similar decrease with fusion. Hence, the simple hardware-initiated prefetching scheme is not as effective in hiding a large portion of the memory latency for filter, and fusion provides most of the improvement at 8 processors.

7.5.4.2 Results for Applications

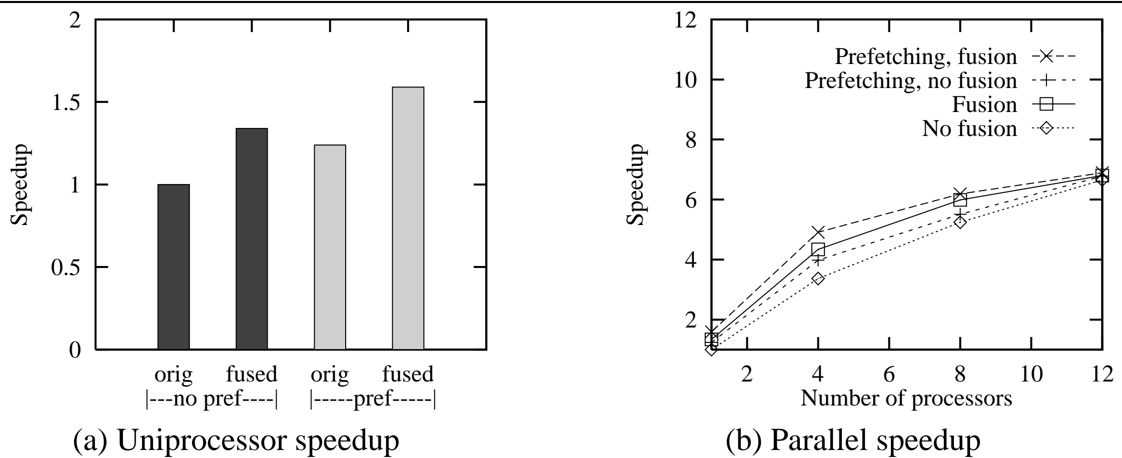
This section describes the results for the complete hydro2d application from which filter was extracted. In addition to the 10-loop sequence in filter, the hydro2d application also contains two sequences of three loop nests that are fused using the shift-and-peel transformation. The array size is 802×320 for these experiments.

First, the performance results are given for the Convex SPP1000/SPP1600 multiprocessors.



(a) Uniprocessor speedup (b) Parallel speedup

Figure 7.14: `hydro2d` on Convex SPP1000 and SPP1600



(a) Uniprocessor speedup (b) Parallel speedup

Figure 7.15: `hydro2d` on SGI Power Challenge R10000

Figure 7.14 shows the uniprocessor and parallel speedups for the `hydro2d` application. *Again, all speedups are with respect to the execution time without fusion on one processor of the SPP1000 (see Table 7.2).* Hence, higher speedups indicate better absolute performance. This allows a direct performance comparison between the SPP1600 with hardware prefetching and the SPP1000 with no prefetching. Figure 7.14 shows that the faster processors with prefetching on the SPP1600 can improve performance significantly, but fusion combined with prefetching on the SPP1600 provides the best performance.

Figure 7.15 shows the uniprocessor and parallel speedups for the `hydro2d` application on

the SGI Power Challenge R10000. All speedups are with respect to the execution time without fusion or prefetching on one processor (700 seconds for 50 iterations) , hence the increase in speedup for a given number of processors represents better absolute performance. Because prefetching can be disabled in software, performance comparisons on the same microprocessor are possible. In the uniprocessor results shown in Figure 7.15(a), combining loop fusion with prefetching provides the best performance. Similar behavior occurs in multiprocessor execution, as shown in Figure 7.15(b). For example, at 4 processors, prefetching alone improves performance by 18%, and fusion alone improves performance by 29%. The combination of both fusion and prefetching at 4 processors improves performance by 46% over execution using neither technique. Fusion substantially enhances locality *across* loop nests to make more memory system bandwidth available and improve the effectiveness of prefetching.

In contrast, other research has primarily combined prefetching and *individual* loop nest transformations such as tiling, with mixed results. Mowry [MLG92] reports that a kernel for Gaussian elimination performed best when tiled with or without prefetching, while another kernel performs best when loop permutation is combined with prefetching. Saavedra *et al.* [SMP+96] consider tiling and prefetching for a matrix multiplication loop nest, and report that prefetching alone provided the best performance. Finally, Saavedra *et al.* [SMP+96] and Bugnion *et al.* [BAM+96] indicate that prefetching is generally effective in hiding latency, but in some cases, tiling reduces the effectiveness of software-controlled prefetching by inhibiting software pipelining and preventing prefetch instructions from being scheduled early enough to hide miss latency.

Figure 7.15 also confirms one additional feature regarding performance improvements for a fixed problem size. Adding more processors increases the total cache capacity while reducing the amount of data accessed per processor. As a result, more of the data accessed by each processor can remain cached, and there is less need for prefetching and locality enhancement with fusion. However, it is still necessary to avoid cache conflicts to retain data in the cache for reuse, as shown in Section 7.4. Conflict avoidance in parallel execution has also been discussed elsewhere by Manjikian and Abdelrahman [MA95] and by Bugnion *et al.* [BAM+96].

7.5.5 Summary for the Shift-and-peel Transformation

In summary, the results for the shift-and-peel transformation provide the following conclusions:

- Representative loop nest sequences in application programs exhibit dependences that require the shift-and-peel transformation to enable legal fusion and parallelization.
- The reduction in the number of cache misses provided by the shift-and-peel transformation leads to significant improvements in performance. The measured improvements compare favorably with estimates obtained using the model discussed in Chapter 3.
- Combining the shift-and-peel transformation with prefetching on systems that support it provides the largest performance improvement. By reducing the number of memory accesses, the shift-and-peel transformation improves the effectiveness of prefetching in hiding memory latency.

In addition, the experimental results have demonstrated that the shift-and-peel transformation avoids the unnecessary overhead of code and data replication used in other techniques. Finally, the results have also shown that the overhead of the shift-and-peel transformation degrades performance when data fits in the cache, which suggests using run-time knowledge of data size with respect to cache size for selecting execution of the original or transformed code.

7.6 Evaluation of Scheduling for Wavefront Parallelism

This section presents the results of experiments to evaluate the scheduling strategies discussed in Chapter 5 for exploiting wavefront parallelism in tiled loop nests. The experiments seek to establish the importance of exploiting intertile reuse to enhance locality when the shift-and-peel transformation and loop skewing are required to enable tiling. Intertile locality enhancement is especially important when small tile sizes are used to provide sufficient parallelism for a large number of processors.

All of the results reported in this section were obtained on the Convex SPP1000 multiprocessor. The results are limited to the `SOR`, `Jacobi`, and `LL18` kernels, as they contain loop nest sequences embedded within an outer loop that carries reuse. Tiling of the `Jacobi` and `LL18` kernels requires the application of fusion to inner loop nests, and the dependences

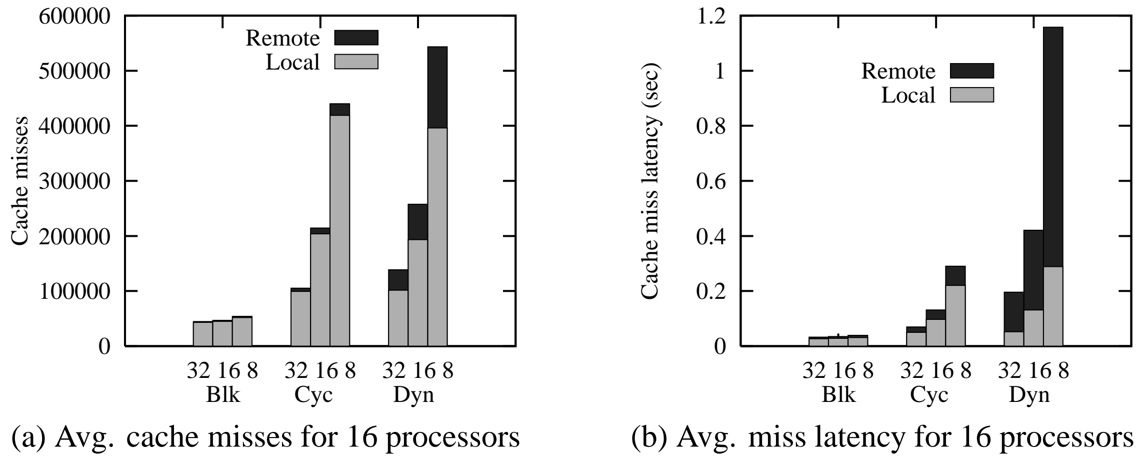
between these inner loop nests require the shift-and-peel transformation to enable legal fusion. For both fusion and tiling, cache partitioning is employed to allow data from multiple arrays to remain cached for reuse without conflicting.

7.6.1 Results for `SOR`

The first results in this section are for the `SOR` loop nest that was discussed in Chapter 5. The array size is 1024×1024 elements, and each element is 8 bytes. The number of iterations in the original outer loop is $T = 40$. Results are provided for tile sizes of 32×32, 16×16, and 8×8 for each of the three scheduling strategies. Larger tile sizes are not considered because they do not provide sufficient parallelism for a large number of processors. Figure 7.16 shows the average number of cache misses and corresponding miss latencies per processor on 16 processors (i.e., 2 hypernodes). Both the number of misses and the latencies are broken down into local and remote. Figure 7.16(a) indicates that block scheduling incurs far fewer misses for a given tile size than dynamic or cyclic scheduling, which agrees with the analytical observations in Section 5.4.4.4. The fraction of misses to remote memory is small for block and cyclic scheduling (4% and 5% respectively for a tile size of 8). This fraction is significantly larger for dynamic self-scheduling (27% for a tile size of 8). Hence, the impact of the remote misses on the total cache miss latency shown in Figure 7.16(b) is more pronounced for dynamic self-scheduling because of the higher cost of remote misses. As the tile size is reduced, both the number of cache misses and the total miss latencies increase dramatically for both dynamic and cyclic scheduling. The resulting miss latency for dynamic self-scheduling with a tile size of 8 is *30* times larger than for block scheduling.[1] This clearly demonstrates the detriment of failing to provide intertile locality when the tile size is small. Block scheduling is much less sensitive to a reduction in the tile size because it exploits all intertile reuse.

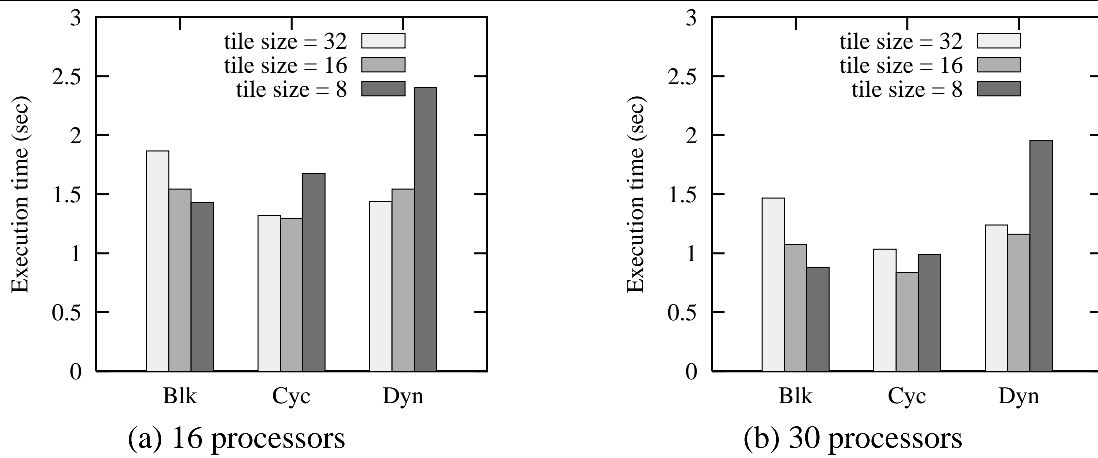
The effect of the cache behavior on execution time for tiled `SOR` is shown in Figure 7.17 for 16 processors, and also for 30 processors. The results indicate that static scheduling performs better than dynamic scheduling for a large number of processors, but only block scheduling improves consistently when the tile size is reduced to provide greater parallelism. Although the results indicate that cyclic scheduling with an intermediate tile size may perform better than

[1] Note that because tiling significantly reduces the memory access component of execution time, the impact of the increased miss latency on execution time is much less than a factor of 30.



(a) Avg. cache misses for 16 processors (b) Avg. miss latency for 16 processors

Figure 7.16: Cache misses for tiled SOR



(a) 16 processors (b) 30 processors

Figure 7.17: Execution times for tiled SOR

block scheduling, it may be difficult to predict an optimal tile size for cyclic scheduling that achieves the appropriate balance between sufficient parallelism and sufficient locality. Later results in this section will confirm this observation. Furthermore, block scheduling simplifies the selection of the tile size for a large number of processors. It is sufficient to choose a small tile size for greater parallelism; intertile locality is preserved with block scheduling. Hence, the remainder of the results focus on comparing block scheduling with dynamic scheduling for the largest and smallest tile sizes.

The parallel speedup of tiled SOR is shown in Figure 7.18; all speedups are computed with

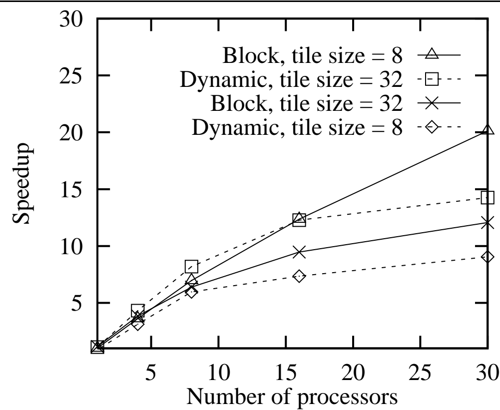


Figure 7.18: Speedup for tiled `SOR`

respect to the *untiled* loop nest executed on a single processor, hence the increase in speedup for a given number of processors represents an improvement in absolute performance. The speedups of block scheduling and dynamic self-scheduling are compared for the largest and smallest tile sizes. When the number of processors is 8 or less, all memory accesses are confined within a single hypernode, i.e., there are no remote memory accesses. Dynamic self-scheduling with a large tile size generates sufficient parallelism for the relatively small number of processors, and maximizes intratile locality. The larger tile size and the uniform memory access within a hypernode diminish the impact of intertile locality. Consequently, dynamic self-scheduling with the largest tile size performs the best. However, as the number of processors increases, a large tile size limits the speedup of dynamic self-scheduling due to insufficient parallelism. In addition, memory accesses span hypernodes and become non-uniform, which limits the speedup of dynamic self-scheduling, particularly when a smaller tile size is used to provide greater parallelism. Intertile locality is critical for small tile sizes, and dynamic self-scheduling does not exploit intertile reuse. In contrast, block scheduling with a small tile size provides sufficient parallelism while enhancing intertile locality, improving the speedup by a factor of 1.4 over dynamic self-scheduling at 30 processors.

7.6.2 Results for `Jacobi`

The `Jacobi` kernel consists of two loop nests surrounded by an outer loop, as shown in Figure 7.19. There is reuse between the inner two loop nests in addition to the reuse carried by

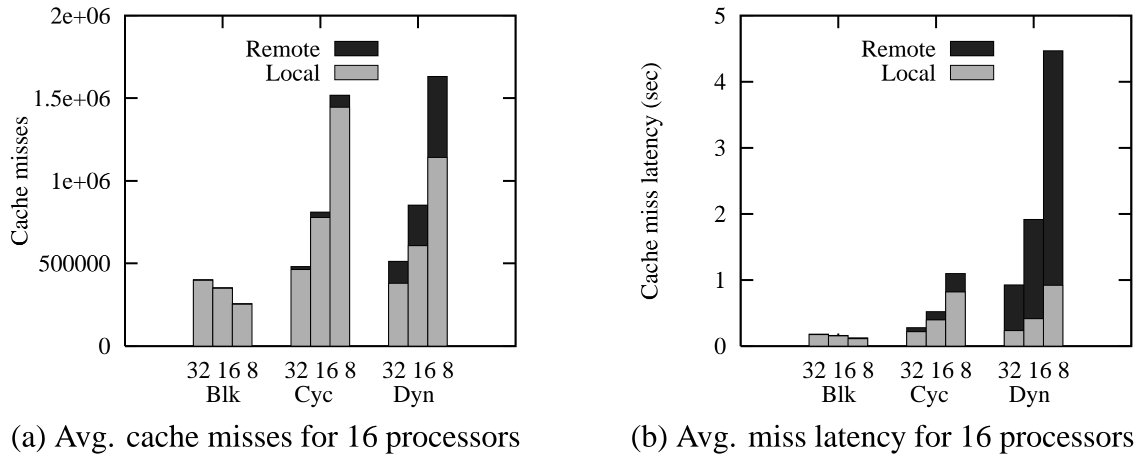
```
do t=1,T
   do j=2,N−1
      do i=2,N−1
         b[i,j] = (a[i+1,j]+a[i−1,j]+a[i,j+1]+a[i,j−1]) / 4
      end do
   end do
   do j=2,N−1
      do i=2,N−1
         a[i,j] = b[i,j]
      end do
   end do
end do
```

Figure 7.19: The `Jacobi` kernel

the outer loop. Tiling requires fusion of the inner two loop nests to produce a single loop nest, and dependences between the inner two loop nests require the application of the shift-and-peel transformation to enable legal fusion. Once a single loop nest is obtained with fusion, loop skewing is required just as for the `SOR` loop nest to enable tiling. The application of shift-and-peel to enable fusion results in dependences that require skewing the inner loops by *two* iterations with respect to the outer loop, rather than one as required for `SOR`. Once skewed, the loop nest is then tiled to exploit the reuse carried by the outer loop.

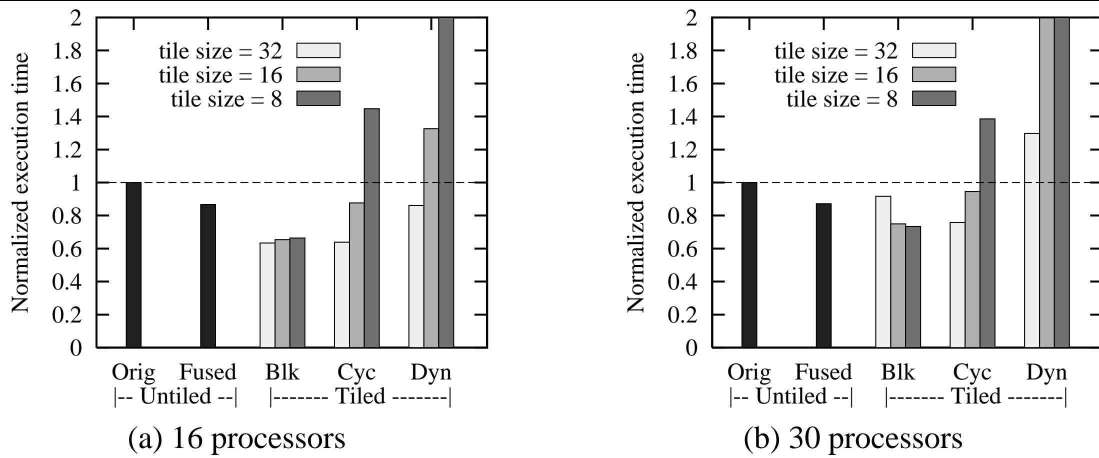
Figure 7.20 shows the average number of cache misses and corresponding miss latencies per processor for parallel execution of tiled `Jacobi` with the different scheduling strategies on 16 processors. The array sizes are 2048 $\times$ 2048 and the number of iterations in the original outer loop is $T = 10$. As before, the number of misses and the latencies are broken down into local and remote. The results are similar to those obtained for `SOR`. Block scheduling incurs the fewest cache misses as well as having the smallest fraction of remote misses. The cache latency for block scheduling is also the lowest. Dynamic self-scheduling incurs the greatest number of cache misses and a larger fraction of remote misses, which results in a dramatic increase in cache miss latency as the tile size is reduced.

Normalized execution times for tiled `Jacobi` on 16 and 30 processors are shown in Figure 7.17. All execution times are normalized with respect to time obtained with parallel execution of the *original* code to facilitate comparison. The normalized execution time for fusion of the inner loops without tiling is also shown, since parallel execution of the fused loops is enabled by the shift-and-peel transformation. Once again, the results for tiling are similar to



(a) Avg. cache misses for 16 processors (b) Avg. miss latency for 16 processors

Figure 7.20: Cache misses for tiled `Jacobi`



(a) 16 processors (b) 30 processors

Figure 7.21: Normalized execution times for tiled Jacobi

those obtained for `SOR`. The dramatic increase in execution time for dynamic self-scheduling correlates with the increase in the cache miss latency. Block scheduling with a small tile size performs far better. Fusion exploits reuse between the inner two loops, but tiling goes further to exploit the reuse carried by the outer loop. To ensure that the full benefit of tiling is realized, the tiled loop nest must be scheduled appropriately.

Finally, the parallel speedup of tiled `Jacobi` for various numbers of processors over the original code executed on a single processor is shown in Figure 7.22. The increase in speedup for a given number of processors represents an improvement in absolute performance. The

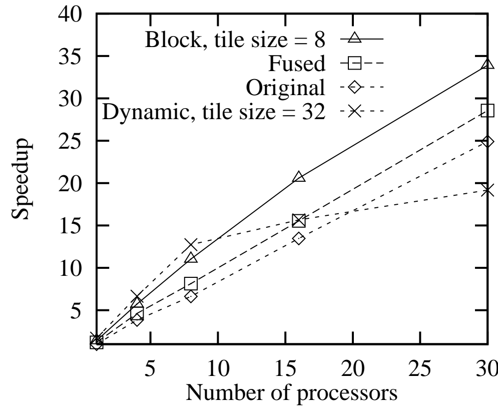
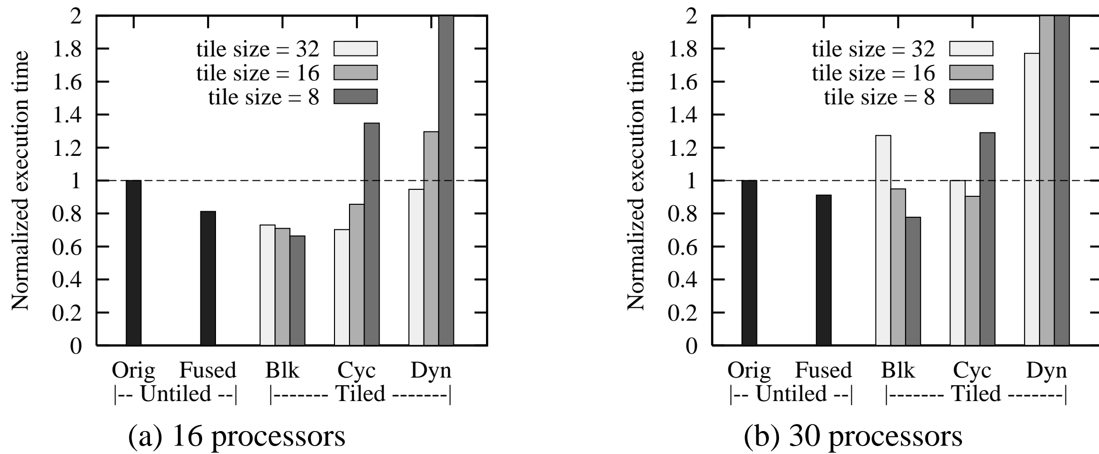


Figure 7.22: Speedup for Jacobi



(a) 16 processors (b) 30 processors

Figure 7.23: Normalized execution times for tiled LL18

speedup for block scheduling and dynamic self-scheduling is compared to the speedup from parallel execution of the original code and the fused version. The speedup for cyclic scheduling is not shown because its performance for small tile sizes is worse than block scheduling. Once again, dynamic self-scheduling with a large tile size is only effective in the absence of remote memory accesses, i.e., when the number of processors is 8 or less. In contrast, block scheduling with a small tile size improves the speedup by a factor of 1.8 over dynamic self-scheduling at 30 processors, and consistently outperforms even the parallel versions of the original and fused code. Block scheduling improves parallel speedup by 53% over the original code at 16 processors, and by 36% at 30 processors.

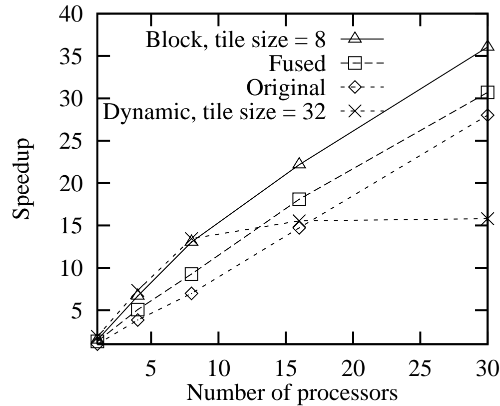


Figure 7.24: Speedup for LL18

7.6.3 Results for `LL18`

The `LL18` kernel consists of three loop nests surrounded by an outer loop. A total of nine arrays are used, and there is reuse between the inner loop nests in addition to the reuse carried by the outer loop. Tiling requires fusion with the shift-and-peel transformation to produce a single loop nest, followed by skewing of the inner loops by *three* iterations. The tiled loop nest is scheduled with the different strategies just as for `SOR` and `Jacobi`. Normalized execution times for 16 and 30 processors are shown in Figure 7.23 for array sizes of 1024×1024 and $T = 10$ iterations in the original outer loop. The results are similar to those obtained for `Jacobi`. Fusion improves performance by exploiting reuse between the inner loop nests, but tiling with an appropriate scheduling strategy exploits all the reuse for the best performance. Once again, only block scheduling is successful in enhancing locality when the tile size is reduced to provide sufficient parallelism for a large number of processors. The speedups for `LL18` shown in Figure 7.24 also agree with the trends observed for `Jacobi`. Block scheduling improves the speedup at 30 processors by a factor of 2.3 over dynamic self-scheduling. Block scheduling improves the speedup by 50% over the original code at 16 processors, and by 29% at 30 processors.

7.6.4 Comparison of Sweep Ratios for Tiling

This section compares sweep ratios for tiling of the `Jacobi` kernel. Table 7.11 provides the number of cache misses measured on one processor during parallel execution on 16 processors

Table 7.11: Cache misses and sweep ratios for `Jacobi` on Convex SPP1000 (16 processors)

Version of code	Measured cache misses	Sweep ratio measured	Sweep ratio predicted
Original	2645990	—	—
Fused	1991090	1.3	1.3
Dyn. sched., tile size=32	525763	5.0	5.9
Dyn. sched., tile size=8	1664540	1.6	2.2
Block sched., tile size=32	405631	6.5	13
Block sched., tile size=8	266595	9.9	13

for different versions of `Jacobi`. For each version, the measured sweep ratio is computed with respect to the original code. For example, the measured sweep ratio for the fused code is $2645990/1991090 = 1.3$.

Table 7.11 also provides the predicted sweep ratios for each version of the code. The sweep ratio of 1.3 for fusion was computed earlier in Table 7.5. For dynamic scheduling of the tiled code, no intertile reuse is exploited, hence the sweep ratio for tiling with skewing is given by Equation 5.1 in Section 5.2.2, i.e., $r_{tiling} = T/(2 \cdot (s \cdot T/B) + 1)$. For `Jacobi`, the skewing factor is $s = 2$ and the number of iterations is $T = 10$. The tile sizes are $B = 8$ and $B = 32$. Hence, $r_{tiling} = 1.7$ for $B = 8$ and $r_{tiling} = 4.4$ for $B = 32$.

However, tiling is preceded by fusion whose sweep ratio is $r_{fusion} = 1.3$. The overall sweep ratio for both transformations is given by the product of sweep ratios for the individual transformations (Equation 5.2): $r_{overall} = r_{fusion} \cdot r_{tiling}$. The overall sweep ratios for dynamic scheduling with tile sizes of 8 and 32 are computed accordingly and given in Table 7.11.

For block scheduling, the analysis in Section 5.4.4.4 explained that *even with skewing*, exploiting intertile reuse should result in the *ideal* sweep ratio of T, the number of iterations in the outermost loop. Since $T = 10$ for `Jacobi`, $r_{tiling} = T = 10$ for block scheduling. Tiling is still preceded by fusion, hence the overall sweep ratio is $sr_{overall} = r_{fusion} \cdot r_{tiling} = 1.3 \cdot 10 = 13$.

Comparing the measured and predicted sweep ratios in Table 7.11, the results for fusion agree closely (see the validation in Section 7.5.2). For tiling with dynamic scheduling, the large tile size provides the best agreement, whereas for block scheduling, the small tile size provides the best agreement. These results are not surprising because dynamic scheduling exploits

*intra*tile reuse best with a large tile size, while block scheduling exploits *inter*tile reuse best with a small tile size. The discrepancy for dynamic scheduling with a small tile size is caused by referencing additional cache lines at tile boundaries; these references are not significant when the tile size is large. The discrepancy for block scheduling with a large tile size is due to conflicts that reduce intertile locality. Only one-dimensional cache partitioning is applied in this case, and this is sufficient for the small tile size. Although two-dimensional cache partitioning would increase the measured sweep ratio for block scheduling with the large tile size, the small tile size is still required for sufficient parallelism on a large number of processors.

7.6.5 Summary for Evaluation of Scheduling Strategies

In summary, the experimental results have confirmed that exploiting intertile reuse is crucial to improve the performance of tiling for a large number of processors. Block scheduling permits the use of small tile sizes required to provide sufficient parallelism without sacrificing locality. Furthermore, the results for `Jacobi` and `LL18` demonstrate the importance of exploiting *all* reuse, first from fusing inner loop nests, then by tiling for reuse carried by the outermost loop.

Chapter 8

Conclusion

The performance of applications on large-scale shared-memory multiprocessors is determined largely by the degree of parallelism and cache locality. Parallelism in applications is often found in loops that operate on array data, and current parallelizing compilers are capable of detecting this loop-level parallelism. Compilers also enhance cache locality by applying loop transformations that reorder iterations in order to increase the likelihood of retaining reused data in the cache. However, existing transformations are ineffective because they only exploit reuse within loops or they fail to preserve parallelism when exploiting reuse across loops. Furthermore, even when reuse can be exploited across loops, cache conflicts between data from different arrays diminish locality by displacing data from the cache before it is reused.

To address these shortcomings, this dissertation has proposed and evaluated new compiler techniques to improve parallel performance on large-scale multiprocessors. Novel code and data transformations enhance cache locality across loops while avoiding cache conflicts and maintaining sufficient parallelism for a large number of processors. These transformations are combined with appropriate loop scheduling strategies to maintain locality and parallelism during execution.

The importance of the techniques described in this dissertation will continue to increase as processor performance continues to increase more rapidly than memory performance. Current multiprocessors are now using commodity microprocessors operating at speeds of 200 MHz or more. Cache misses to access memory are extremely costly at these speeds; execution time is increasingly dominated by memory access time. All available data reuse must be exploited for cache locality if significant reductions in execution time are to be achieved, especially in large-scale multiprocessors with large remote memory latencies.

8.1 Summary of Contributions

The contributions of this research are embodied in the proposed techniques. The following paragraphs summarize these contributions:

- The *shift-and-peel transformation* has been proposed to enable legal fusion and subsequent parallelization, despite dependences that have previously prevented loop fusion or resulted in a serial loop. This technique is therefore able to exploit reuse across loops and maintain parallelism where previous techniques have failed. Experimental results have shown that shift-and-peel is required for representative loop nest sequences and that it improves parallel performance by up to 30% for representative applications. Results have also shown that the shift-and-peel transformation provides additional performance gains in conjunction with other performance-enhancing techniques such as prefetching.

- *Loop scheduling strategies* have been evaluated for exploiting wavefront parallelism that results when the shift-and-peel transformation is combined with tiling. Proper scheduling allows both intratile and intertile data reuse to be exploited effectively during parallel execution on a large number of processors. Experimental results have shown that static block scheduling with a small tile size improves parallel performance by up to 50% over the original code without fusion or tiling. Results have also shown that static scheduling incurs the fewest cache misses and is the least sensitive to changes in the tile size.

- A data transformation technique called *cache partitioning* has been proposed to prevent cache conflicts between data from different arrays, especially when enhancing locality across loops. Unlike techniques such as array padding, cache partitioning systematically derives a conflict-free data layout in memory for commonly-occurring compatible data access patterns that would otherwise lead to frequent conflicts. Experimental results have shown that cache partitioning permits the full benefit of transformations such as shift-and-peel to be realized by preventing unnecessary cache misses.

To provide a means of assessing the benefit of enhancing locality across loops with transformations such as shift-and-peel and tiling, an analytical model has been described for quantifying

the reduction in memory accesses. The contribution of memory accesses towards total execution time is also quantified in order to associate the reduction in memory accesses with a reduction in execution time. Estimates for performance improvement obtained with this model compare favorably with measured improvements in the experimental results.

Finally, the feasibility of the proposed techniques has been established with a prototype source-to-source compiler implementation in FORTRAN 77. The implementation automates the shift-and-peel and cache partitioning transformations (with the exception of modifications required to produce consistent COMMON block definitions for cache partitioning). Although the prototype implementation performs only source-level transformations, the techniques are equally feasible for a native machine compiler that generates executable code. Integrating the requisite analyses and transformations into a native compiler framework should enable further performance improvements.

8.2 Future Work

The results from combining the shift-and-peel transformation with prefetching suggest further work to explore the limits on the achievable improvements in performance. Current processor designs support between 4 and 10 concurrent memory accesses [Hun95, Yea96]. Further experimentation can investigate how effectively the shift-and-peel transformation can improve the available bandwidth utilization for prefetching with a large number of concurrent memory accesses, and to what extent computation becomes the performance bottleneck in representative loop nest sequences when both techniques are combined.

In this dissertation, candidate loops for the shift-and-peel transformation have been identified within individual subroutines. This approach exploits a common programming style in which related loops are placed together in the same subroutine. The scope of the transformation can be increased with interprocedural techniques, specifically selective subroutine inlining to collect loops from separate subroutines and form larger candidate sequences.

The techniques have centered on dense array applications that constitute an important class of scientific applications. However, there are also many applications that operate on sparse arrays. These applications may also contain loop sequences that are candidates for loop fusion, although the data access patterns (and hence the dependence relationships) may not be as

regular. Future work can investigate the feasibility of exploiting reuse across loops in such applications.

A more challenging direction for future work is broadening the scope of locality enhancement in non-numeric applications. Such applications also contain loops; however, instead of *DO*-loops with an integer index variable, such applications may employ *WHILE*-loops using a pointer variable to traverse complex data structures. The challenge is to identify opportunities to legally combine the bodies of multiple *WHILE*-loops in such applications.

Finally, this dissertation has centered on cache locality enhancement. The physically-distributed memory in large-scale multiprocessors also raises the issue of memory locality enhancement, i.e., ensuring that most cache misses are satisfied by local rather than remote memory. Memory locality is enhanced with an appropriate data distribution to match the distribution of parallel computation among processors. Future work can investigate the interaction between cache and memory locality enhancement and the relative importance of each.

Bibliography

[AAL95] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, pages 166–178, July 1995.

[AS92] W. Appelbe and K. Smith. Determining transformation sequences for loop parallelization. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, pages 208–222, 1992.

[ASKL81] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, 30(5):341–356, May 1981.

[BAM+96] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, Cambridge, MA, October 1996.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston, 1993.

[BCJ+94] David F. Bacon, Jyh-Herng Chow, Dz-ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94*, pages 270–282, Toronto, Ontario, Canada, October 1994.

[BCK+89] M. Berry, D. Chen, P. Koss, D. Kuck, and S. Lo. The PERFECT Club Benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, CSRD, University of Illinois, May 1989.

[BEF+95] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In K. Pingali et al., editors, *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, pages 141–154. Springer-Verlag, Berlin, 1995.

[Bel92] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):26–47, August 1992.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26:345–420, December 1994.

[Cal87] Charles D. Callahan. *A Global Approach to the Detection of Parallelism*. PhD thesis, Department of Computer Science, Rice University, March 1987.

[CHK+96] Kenneth K. Chan, Cyrus C. Hay, John R. Keller, Gordon R. Kurpanek, Francis X. Schumacher, and Jason Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal*, 47(1), February 1996. Available at http://www.hp.com/hpj/journal.html.

[CM95] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, June 1995.

[CMT94] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, October 1994.

[Con94] Convex Computer Corporation. *Convex Exemplar System Overview*. Document No. 080-002293-000, Richardson, TX, 1994.

[DR94] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):814–822, October 1994.

[DWYF92] Eric DeLano, Will Walker, Jeff Yetter, and Mark Forsyth. A high speed superscalar PA-RISC processor. In *Compcon'92 Digest of Papers*, pages 116–121, San Francisco, CA, February 1992.

[Gan94] Amit Ganesh. Fusing loops with backwards inter loop data dependence. *ACM SIGPLAN Notices*, 29(12):25–30, December 1994.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[Hed94] Katherine S. Hedstrom. *User's Manual for a Semi-spectral Primitive Equation Ocean Circulation Model–Version 3.9*. Rutgers University, March 1994. Available at http://marine.rutgers.edu/po.

[HKO+94] M. Heinrich, J. Kuskin, D. Ofelt, et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 1994.

[HS96] Edin Hodzic and Weijia Shang. On optimal size and shape of supernode transformations. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages III25–III34, August 1996.

[HSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[Hun95] Doug Hunt. Advanced performance features of the 64-bit PA-8000. In *Compcon'95 Digest of Papers*, pages 123–128, San Jose, CA, February 1995.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM Symposium on the Principles of Programming Languages*, pages 319–329, January 1988.

[Jou90] Norman Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[Ken91] Kendall Square Research. *KSR1 Principles of Operation*. Waltham, Mass., 1991.

[KM92] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 323–334, Washington, D. C., July 1992.

[KM94] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, Berlin, 1994.

[Kuc] Kuck and Associates, Inc. Information on their KAP parallelizing compiler is available at the Website http://www.kai.com.

[LH97] Gregg Lesartre and Doug Hunt. PA-8500: The continuing evolution of the PA-8000 family. In *Compcon'97 Digest of Papers*, San Jose, CA, February 1997. Available at http://www.hp.com/computing/framed/technology/micropro/pa-8500/docs/8500.html.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[LTSS93] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II140–II147, August 1993.

[LW94] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, October 1994.

[LW95] Daniel Lenoski and Wolf-Dietrich Weber. *Scalable Shared-memory Multiprocessing*. Morgan Kaufmann, San Francisco, 1995.

[MA95] Naraig Manjikian and Tarek Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, pages 111–118, Orlando, FL, September 1995.

[MA97] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[McC] John D. McCalpin. *STREAM: Measuring Sustainable Memory Bandwidth in High Performance Computers*. Web page http://www.cs.virginia.edu/stream/ maintained by the Department of Computer Science, University of Virginia. This Web site provides performance results for the STREAM memory bandwidth benchmark.

[McC92] John D. McCalpin. *Quasigeostrophic Box Model–Revision 2.3*. University of Delaware, July 1992. This is an unpublished description of the `qgbox` code.

[ML94] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, October 1992.

[MT96] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, MA, October 1996.

[MWV92] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The MIPS R4000 processor. *IEEE Micro*, 12(2):10–22, April 1992.

[NJL94] Juan J. Navarro, Toni Juan, and Tomas Lang. MOB forms: A class of multilevel block algorithms for dense linear algebra. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, pages 354–363, Manchester, England, July 1994.

[Pac] Pacific Sierra Research Corporation. Information on their VAST parallelizing compiler is available at the Website http://www.psrv.com.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[Por89] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, April 1989.

[Pug91] William Pugh. Uniform techniques for loop optimization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[Pug92] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[RLBC94] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware.

In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 255–266, November 1994.

[Sil96a] Silicon Graphics, Inc. *Origin Servers: Technical Overview of the Origin Family*. Mountain View, CA., 1996. Available at http://www.sgi.com/Products/hardware/servers/technology/overview.html.

[Sil96b] Silicon Graphics, Inc. *POWER CHALLENGE: Technical Report*. Mountain View, CA., 1996. Available at http://www.sgi.com/Products/software/PDF/pwr-chlg/.

[SMP+96] Rafael H. Saavedra, Weihau Mao, Daeyeon Park, Jacqueline Chame, and Sungdo Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 39–45, Honolulu, HI, April 1996.

[Sta] Standard Performance Evaluation Corporation. Information on the SPEC benchmark suite is available at the Website http://www.specbench.org.

[Sun96] Sun Microsystems, Inc. *Ultra Enterprise X000 Servers: A Technology Overview*. Mountain View, CA., 1996. Available at the Website http://www.sun.com/960416/wp/wp.ultra.server.html.

[TFJ93] Olivier Temam, Christine Fricker, and William Jalby. Impact of cache interferences on usual numerical dense loop nests. *Proceedings of the IEEE*, 81(8):1103–1115, August 1993.

[VBS+95] Zvonko Vranesic, Stephen Brown, Michael Stumm, Steve Caranci, Alex Grbic, Robin Grindley, Mitch Gusat, Orran Krieger, Guy Lemieux, Kelvin Loveless, Naraig Manjikian Zeljko Zilic, Tarek Abdelrahman, Ben Gamsa, Peter Pereira, Ken Sevcik, Ali Elkateeb, and Sinisa Srbljic. The NUMAchine Multiprocessor. Technical Report CSRI-324, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, April 1995.

[War84] Joe Warren. A hierarchical basis for reordering transformations. In *Proceedings of the 11th ACM Symposium on the Principles of Programming Languages*, pages 272–282, June 1984.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steven W. K. Tjiang, Shi-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), December 1994.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA., 1989.

[Wol92] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Department of Computer Science, Stanford University, August 1992.

[Yea96] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Vector and Parallel Computers*. ACM Press, New York, 1991.

[ZLTI96] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996. Available at http://www.supercomp.org/sc96/proceedings/.