INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

RICE UNIVERSITY

Transforming Complex Loop Nests For Locality

by

Qing Yi

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

Ken Kennedy, Chair
Professor of Computer Science

Keith Cooper
Professor of Computer Science

Danny Sorensen
Professor of Computational and Applied
Mathematics

Vikram Adve
Professor of Computer Science
University of Illinois at
Urbana-Champaign

Houston, Texas

April, 2002

UMI Number: 3047379

UMI Microform 3047379

**Copyright 2002 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**

**ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**

ABSTRACT

Transforming Complex Loop Nests For Locality

by

Qing Yi

Over the past 20 years, increases in processor speed have dramatically outstripped performance increases for standard memory chips. To bridge this gap, compilers must optimize applications so that data fetched into caches are reused before being displaced. Existing compiler techniques can efficiently optimize simple loop structures such as sequences of perfectly nested loops. However, on more complicated structures, existing techniques are either ineffective or require too much computation time to be practical for a commercial compiler.

This thesis develops the following novel techniques to optimize complex loop structures both effectively and inexpensively for better locality.

Extended dependence representation: a matrix representation that incorporates dependence relations between iterations of arbitrarily nested loops.

Transitive dependence analysis algorithm: a new algorithm that improves the time complexity of existing transitive dependence analysis algorithms.

Dependence hoisting: a new loop transformation technique that permits the direct fusion and interchange of arbitrarily nested loops. The transformation is inexpensive and can be incorporated into most commercial compilers.

Computation slicing: a framework that systematically applies *dependence hoisting* to optimize arbitrary loop structures for better locality.

Recursion transformation: the first compiler work that automatically transforms loop structures into recursive form to exploit locality simultaneously at multiple levels of the memory hierarchy.

Both the computation slicing framework and recursion transformation have been implemented and applied to successfully optimize a collection of benchmarks. In particular, the slicing framework has successfully blocked four linear algebra kernels: Cholesky, QR, LU factorization without pivoting, and LU with partial pivoting. The auto-blocked versions have achieved performance improvements similar to those attained by manually blocked programs in LAPACK [7]. The automatic blocking of QR and pivoting LU is a notable achievement because these kernels include loop nests that are considered difficult — to our knowledge, few previous compiler implementations have completely automated the blocking of the loop nests in these kernels. These facts indicate that although with a cost much lower than that of existing more general transformation frameworks [34, 42, 2, 36, 49], the computation slicing framework can in practice match or exceed the effectiveness of these general frameworks.

Acknowledgments

I would like to thank my advisor, Ken Kennedy, for his help and support in making this dissertation possible. Ken Kennedy has changed my life in many ways, and his inspiration will last far beyond this dissertation. I also want to thank my other committee members, Keith Cooper, Danny Sorensen and Vikram Adve, for their support and advice. Vikram Adve is my co-advisor and has worked closely with me during the earlier part of my graduate career here at Rice University. Danny Sorensen has helped me understand the linear algebra kernel computations used in this thesis. Keith Cooper has always been so supportive that he makes the stress go away. I also thank John Mellor-Crummey for his help and advice all through my graduate career in the compiler group.

I built my implementations on top of the DSystem, an infrastructure led by John Mellor-Crummey (and by Vikram Adve before he left Rice) and developed by researchers in the compiler group through many years. This work is supported by Rice University and by funding from Lawrence Livermore National Laboratory (LLNL).

I especially thank Jan Hewitt for helping me improve my writing skills and for helping me revise this dissertation. Her thesis writing seminar is great and will continue to benefit me through the rest of my professional life.

Lastly but importantly, I would like to thank my family and friends for making

my graduate career an enjoyable life. I am thankful to my parents for encouraging me in whatever I do, to my husband for being there through all the up-and-downs, and to my fellow graduate students, Ming Zhang, Chen Ding, Tim Harvey, Daniel Chavarria, Arun Chauhan, and Cheryl McCosh, for offering help and support whenever I needed them.

It takes a long journey to finally defend a Ph.D thesis, but the things I learned along the way are worth much more than the dissertation itself.

Contents

Abstract	ii
Acknowledgments	iv
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Research Problems and Goals	4
1.2 Example: Non-pivoting LU	6
1.3 Proposed Techniques	9
1.3.1 Dependence Hoisting	9
1.3.2 Extended Dependence Model	13
1.3.3 Computation Slicing	14
1.3.4 Recursion Transformation	15
1.4 Related Work	18
1.4.1 Loop Transformation Techniques	18
1.4.2 Recursion Transformation	22
1.5 Organization	24
2 Extended Dependence Model	25

2.1	Notations and Definitions	28
2.2	Dependence Representations	29
2.3	Operations on Dependences	30
2.3.1	Operations on Dependence EDMs	31
2.3.2	Operations on Dependence EDM Sets	34
2.4	Transitive Dependence Analysis Algorithm	36
2.4.1	A Simple Example	37
2.4.2	Preprocessing Cycles	39
2.4.3	Path Summary On DAG	41
2.4.4	Path Summary on Cyclic Graphs	42
2.4.5	Correctness and Complexity	43
3	Dependence Hoisting	46
3.1	Notations and Definitions	47
3.2	Transformation Safety Analysis	50
3.2.1	Interchange and Fusion Analysis	51
3.2.2	Example: Non-pivoting LU	52
3.3	Transformation Algorithms	55
3.3.1	Dependence Hoisting Analysis	56
3.3.2	Dependence Hoisting Transformation	57
3.3.3	Correctness and Complexity	59

4	Computation Slicing	64
4.1	Notations and Definitions	65
4.2	Rearranging Computation Slices	67
4.2.1	Achieving Loop Fusion	67
4.2.2	Achieving Loop Reversal	70
4.2.3	Achieving Loop Interchange	71
4.3	Optimizing Loop Nests	72
4.3.1	Block a Loop Nest	72
4.3.2	Multi-level Fusion	75
4.4	Transformation Framework	80
4.4.1	Transformation Steps	81
4.4.2	Correctness and Complexity	83
5	Recursion Transformation	85
5.1	Example: Non-pivoting LU	86
5.2	Notation and Definitions	90
5.3	Transformation Algorithms	92
5.3.1	Computing Iteration Sets	94
5.3.2	Code Generation	99
5.4	Configuring Transformation For Locality	101
5.4.1	Choosing Key Statements	101

5.4.2	Choosing Recursive Loops	103
5.4.3	Recursive Order and Termination	104
5.4.4	Verifying Profitability	104
6	Experimental Results	106
6.1	Implementation of Translators	110
6.1.1	Slicing Translator	110
6.1.2	Recursion Translator	112
6.2	Transforming Benchmarks	113
6.2.1	Blocking Linear Algebra Kernels	114
6.2.2	Optimizing Application Benchmarks	117
6.3	Performance Measurements	118
6.3.1	Results From Blocking Linear Algebra Kernels	119
6.3.2	Results From Optimizing Application Benchmarks	123
6.3.3	Results From Recursion Transformation	126
6.4	Cache Simulation	129
6.5	Compile Time Evaluation	133
7	Conclusion	139
7.1	Contributions	139
7.2	Future Research	144

References

List of Figures

1.1	Pseudo code of non-pivoting LU (by Dongarra, Gustavson and Karp [18])	7
1.2	Step (1) of translating non-pivoting LU	11
1.3	Steps (2) and (3) of translating non-pivoting LU	12
1.4	Recursive formulation of matrix multiplication	16
2.1	Operations on dependence conditions	32
2.2	Example for transitive dependence analysis algorithm	38
2.3	Transitive dependence analysis algorithm	39
3.1	Dependence and transitive dependence edges for non-pivoting LU . .	53
3.2	Computation slices for non-pivoting LU	53
3.3	Dependence hoisting algorithms	56
4.1	Fusing computation slices	67
4.2	Blocking and multi-level fusion algorithms	73
4.3	Blocking non-pivoting LU	74
4.4	Multi-level fusion example: code fragment from Erlebacher	79
4.5	Transformation steps of computation slicing	81
5.1	Recursive formulation of non-pivoting LU	86
5.2	Recursive code of non-pivoting LU	87

5.3	Recursion transformation algorithm	94
5.4	Choosing key statements	102
6.1	Blocking LU factorization with partial pivoting	115
6.2	Fusing loop nests from <i>tomcatv</i>	117
6.3	Results from blocking linear algebra kernels	121
6.4	Results from optimizing application benchmarks	123
6.5	Results from recursion transformation	127
6.6	Results from simulation of 2-way and fully-associative caches	130
6.7	Compile time of the translators with respect to code size	136

List of Tables

6.1	Application benchmarks used in evaluation	114
6.2	Compile time of the slicing and recursion translators	134

Chapter 1

Introduction

Over the past twenty years, increases in processor speed have dramatically outstripped performance increases for standard memory chips, in both latency and bandwidth. To bridge this gap, architects typically insert multiple levels of cache between the processor and memory, resulting in a deep memory hierarchy. The data access latency to a higher level of the memory hierarchy is often orders of magnitude less than the latency to a lower level. To achieve high performance on such machines, compilers must optimize the locality of applications so that data fetched into caches are reused before being displaced.

Compiler researchers have developed many techniques to optimize the loop structures of applications for better locality. Of these techniques, the most widely used are a set of unimodular transformations, including loop blocking(tiling), reversal, skewing, and interchange [50, 52], and a set of single-loop transformations, including loop distribution, fusion, and index-set splitting [12, 38, 11]. These transformations are inexpensive and very efficient in optimizing simple loop structures such as sequences of perfectly nested loops. However, on more complicated loop structures, these transformations often fail, even when an effective optimization is possible.

This thesis extends these traditional transformation techniques to effectively op-

timize complex loop nests for locality. This work introduces a new transformation, *dependence hoisting*, that facilitates the direct fusion and interchange of arbitrarily nested loops at the outermost position of a code segment containing these loops, and a new transformation framework, *computation slicing*, that systematically combines dependence hoisting with other inexpensive techniques to achieve better locality. The computation slicing framework is fast enough to be incorporated in most commercial production compilers.

To exploit locality at multiple levels of the memory hierarchy simultaneously, this thesis also develops another transformation framework, *recursion transformation*, that automatically transforms arbitrary loop structures into recursive form to exploit the fusion and multi-level blocking property of divide-and-conquer algorithms. The performance benefits of recursive algorithms have been observed by many researchers in both experimental and theoretical results [27, 9, 23, 6]. However, this thesis is the first work to generate recursive algorithms automatically by a compiler.

The contributions of this thesis also include an extended dependence representation for transforming arbitrarily nested loops and a transitive dependence analysis algorithm that summarizes the complete dependence information between statements. The summarized transitive dependence information is used by the computation slicing framework to determine the safety of transforming complex loop structures and is used by the recursion transformation framework to compute related iteration sets

of statements.

This thesis has implemented both the computation slicing and recursion transformation frameworks as Fortran source-to-source translators and has applied the translators to optimize a collection of linear algebra kernels and real-world application benchmarks. Although both translators have broad applicability to general, non-perfectly nested loops, this thesis emphasizes applying computation slicing to block four numerical benchmark kernels — Cholesky, QR, LU factorization without pivoting, and LU factorization with partial pivoting — because these kernels contain complex loop nests that are generally considered difficult to block automatically. To our knowledge, few previous compiler implementations have completely automated the blocking of the loop nests in QR and LU with pivoting. The computation slicing framework has successfully blocked all four benchmarks. The auto-blocked kernels have not only achieved significant performance improvements over the original unblocked versions, they have also achieved a performance level comparable to that of the versions in LAPACK, which were hand-blocked by experienced algorithm designers using algorithm changes in some cases. These experimental results indicate that although the computation slicing framework is much less expensive than any of the existing general transformation frameworks for complex loop nests [34, 42, 2, 36], it is quite powerful and can match or exceed the real-world effectiveness (although not the theoretical effectiveness) of these general frameworks.

This chapter provides an overview of the research focus and contributions of this thesis. Section 1.1 first defines the research problems and goals. Section 1.2 uses a linear algebra code, LU factorization without pivoting, to further explain the motivation of this research. Section 1.3 presents new techniques to achieve the research goals. Section 1.4 discusses related work of this thesis. Finally, Section 1.5 describes the organization of the rest of this thesis.

1.1 Research Problems and Goals

Modern computer architectures typically have a memory hierarchy with many levels, including one or more levels of cache, the local memory of each processor, and a shared memory or the remote memories of other processors. To achieve high performance on such machines, this thesis investigates compiler techniques to automatically improve the cache locality of applications both at a single level and at multiple levels of the memory hierarchy simultaneously.

Three loop transformations — loop fusion, interchange, and blocking — have proven to be especially useful in improving the locality of applications. Existing compilers have successfully applied these techniques to optimize simple loop structures such as sequences of perfectly nested loops, but not on more complicated structures that cannot be translated into perfect loop nests. This thesis thus focuses on facilitating loop fusion, interchange, and blocking transformations on arbitrary loop nests independent of their original nesting structure.

Loop fusion improves the locality of a code segment by fusing loops that access similar sets of memory locations so that the data accesses to these locations can be reused inside the fused loop. Traditional loop fusion techniques can be applied only to sequences of loops at the same level. Therefore they cannot fuse loops that are nested inside one another without first distributing these loops into separate nests. To overcome this restriction, this thesis develops techniques that facilitate the direct fusion of arbitrarily nested loops. The fused loop is placed at the outermost position of a code segment that contains the original loops.

Loop interchange enhances the locality of a loop nest by shifting loops that carry more data reuses inside. Traditional loop interchange techniques can shift a loop ℓ inside only if ℓ is perfectly nested with the other loops inside it and if the shifting of ℓ does not violate any dependence constraint. The interchange thus would fail if ℓ cannot be made perfectly nested outside the other loops. To overcome this restriction, this thesis develops techniques that permit the direct interchange of loops independent of their original nesting structure. The nesting order of two loops thus can be interchanged even if they are not perfectly nested.

Loop blocking (or tiling) exploits locality for a loop nest by partitioning the original computation into smaller blocks so that data can be reused throughout the execution of each block. Compilers typically block a loop nest by combining loop strip-mining and interchange: an outer loop ℓ is first strip-mined and the *strip-enumerating*

loop ℓ_i is then shifted inside a set of loops originally contained in ℓ . Because it builds on loop interchange to achieve a tiling effect, loop blocking is limited by the effectiveness of the loop interchange transformation. By facilitating interchange of arbitrarily nested loops, this thesis permits the blocking of complex, non-perfect loop nests directly.

In summary, this thesis aims at achieving two goals: first, making the traditional loop fusion, interchange and blocking transformations more powerful so that they can be applied directly to arbitrarily nested loops and second, systematically applying these transformations for better locality at multiple levels of a memory hierarchy simultaneously. Section 1.3 summarizes the proposed techniques in more detail.

1.2 Example: Non-pivoting LU

This section further illustrates the motivation of this thesis using three equivalent versions of LU factorization without pivoting, as shown in Figure 1.1. Dongarra, Gustavson and Karp [18] described various implementations of non-pivoting LU with different loop orderings, from which Figure 1.1 shows the three orderings that each places a different loop (k , i or j) at the outermost position. These loop orderings are chosen because they are particularly useful in blocking the non-pivoting LU code.

In Figure 1.1, the KJI form in (a) is commonly used in scientific applications such as the LINPACK collection [17]; both the IKJ form in (b) and JKI form in (c) are less commonly used versions with *deferred operations*: the IKJ form defers

Figure 1.1 Pseudo code of non-pivoting LU (by Dongarra, Gustavson and Karp [18])

the scaling of each row of the matrix until immediately before the update of that row, and the *JKI* form defers the updates to each column until immediately before the scaling of that column. In (c), for example, at each iteration of the outermost j loop, statement s_2 first applies all the deferred updates to column j by subtracting multiples of columns 1 through $j - 1$; statement s_1 then scales column j immediately after these deferred updates.

Now, because all the loops (k , j and i) in Figure 1.1 carry data reuses, a compiler can fully block these code fragments by strip-mining all the loops and then shifting the strip-mined loops inside. In order to achieve this blocking effect, however, the compiler needs the ability to freely interchange the nesting order between any two of the three loops. These loop interchange transformations are equivalent to translating between all pairs of the three loop orderings in Figure 1.1.

The three code fragments in Figure 1.1 are not only equivalent, they also contain

the same dependence constraints. A sophisticated compiler should be able to recognize this equivalence and achieve all the translations. For example, to translate (b) to (a), a compiler can interchange the i and k loops in (b), distribute the interchanged i loop, and then interchange the distributed $i(s_2)$ loop (i loop surrounding s_2) with the $j(s_2)$ loop. A compiler can also easily reverse this procedure and translate (a) back to (b).

However, traditional unimodular and single loop transformations cannot translate between Figure 1.1(a) (or (b)) and Figure 1.1(c) because these translations require the direct fusion of non-perfectly nested loops. For example, to translate from (a) to (c), a compiler needs to fuse the $k(s_1)$ loop (k loop surrounding s_1) with the $j(s_2)$ loop and then place the fused loop outside the $k(s_2)$ loop. However, the $k(s_1)$ loop encloses both statements s_1 and s_2 in the original code and cannot be fused with loop $j(s_2)$ unless $k(s_1)$ is first distributed. Since a dependence cycle connecting s_1 and s_2 is carried by this k loop, it cannot be distributed before fusion. Lacking the ability to fuse the $k(s_1)$ and $j(s_2)$ loops when they are nested inside one another, traditional transformations cannot shuffle the nesting order of these two loops and thus cannot establish the translation from (a) to (c).

To achieve the translation between (a) (or (b)) and (c), a compiler therefore needs the ability to fuse arbitrarily nested loops directly. Because the transformation automatically shifts the fused loop to the outermost position of a code segment

containing the original loops, it also achieves an automatic loop interchange effect on non-perfectly nested loops. Section 1.3.1 describes how to facilitate this transformation in more detail.

1.3 Proposed Techniques

This section proposes new techniques to resolve the research problems presented in Section 1.1 and 1.2. Section 1.3.1 first introduces a new transformation, *dependence hoisting*, that facilitates the fusion and interchange of arbitrarily nested loops. Section 1.3.2 then summarizes an extended dependence model to determine the safety of transforming these loops. Section 1.3.3 describes a transformation framework, *computation slicing*, that systematically applies dependence hoisting to improve the locality of applications. Finally, Section 1.3.4 presents another transformation framework that automatically transforms loop nests into recursive form to exploit locality at multiple levels of the memory hierarchy simultaneously. The recursion transformation techniques are also earlier published in [54].

1.3.1 Dependence Hoisting

This section introduces a new transformation, *dependence hoisting*, that facilitates the fusion of arbitrarily nested loops at the outermost position of a code segment containing these loops. This transformation is particularly useful when the loops to be

fused are nested inside one another and when some loops cannot be legally distributed before fusion, as is the case of fusing the $k(s_1)$ and $j(s_2)$ loops in Figure 1.1(a). By placing the fused loop at the outermost position of the nested loops, dependence hoisting also permits the direct interchange of non-perfectly nested loops.

The dependence hoisting technique can be used to freely translate between any pair of the loop orderings of the non-pivoting LU code in Figure 1.1. Because the translations are similar, this section uses the *KJI* form in Figure 1.1(a) as an example and illustrates how to translate (a) to (c). Figure 1.2(a) shows this original code along with the dependence information between statements s_1 and s_2 , where each dependence edge is marked with dependence relations between iterations of the loops surrounding these statements. As discussed in Section 1.2, translating this code to Figure 1.1(c) requires fusing the $k(s_1)$ and $j(s_2)$ loops at the outermost loop level, and the fusion in turn requires distributing the $k(s_1)$ loop (which also surrounds s_2) first. However, due to the dependence cycle connecting s_1 and s_2 , the distribution is not legal in the original code.

Dependence hoisting resolves this conflict in three steps. First, it creates a new dummy loop surrounding the original code in Figure 1.2(a). This dummy loop has an index variable x that iterates over the union of the iteration ranges of loops $k(s_1)$ and $j(s_2)$. In the same step, the transformation inserts conditionals in (a) so that statement s_1 is executed only when $x = j$ and s_2 is executed only when $x = k$.

Figure 1.2 Step (1) of translating non-pivoting LU

Figure 1.2(b) shows the result of this step, along with the modified dependence edges which include dependence relations between iterations of the new outermost x loop.

Now, because the conditionals $x = k$ and $x = j$ in Figure 1.2(b) synchronize the $k(s_1)$ and $j(s_2)$ loops with the new $x(s_1, s_2)$ loop in a lock-step fashion, loop $x(s_1)$ always has the same dependence conditions as those of loop $k(s_1)$, and loop $x(s_2)$ always has the same dependence conditions as those of loop $j(s_2)$. As shown in the dependence graph of (b), the new outermost x loop now carries the dependence edge from s_1 to s_2 and thus carries the dependence cycle connecting s_1 and s_2 . This shifting of dependence level makes it possible for the second transformation step to distribute

Figure 1.3 Steps (2) and (3) of translating non-pivoting LU

the $k(s_1, s_2)$ loop, which no longer carries a dependence cycle. The transformed code after distribution is shown in Figure 1.3(a). Note that this step requires interchanging the order of statements s_1 and s_2 .

Finally, *dependence hoisting* removes the redundant loops $k(s_1)$ and $j(s_2)$ in Figure 1.3(a) along with the conditionals that synchronize them with the outermost x loop. To legally remove these loops and conditionals, this step substitutes the index variable x for the index variables of the removed loops $k(s_1)$ and $j(s_2)$. In addition, it adjusts the upper bound of the $k(s_2)$ loop to $x - 1$, in effect because the $j(s_2)$ loop

is exchanged outward before being removed. The transformed code after this cleanup step is shown in Figure 1.3(b).

1.3.2 Extended Dependence Model

Before applying dependence hoisting to fuse and shift arbitrarily nested loops, a compiler must determine the safety of the transformation. For example, to translate from Figure 1.1(a) to (c), the compiler must recognize that the $k(s_1)$ and $j(s_2)$ loops can be legally fused at the outermost loop level.

To determine the safety of fusing and interchanging arbitrarily nested loops, this thesis extends the traditional dependence model by making two changes. First, it extends the information associated with each dependence edge to specify not only relations between iteration numbers of common loops, but also relations between non-common loops. Second, it applies transitive analysis on the dependence graph to summarize the complete dependence information between statements. These summarized transitive dependences are then used to determine the safety of transforming loops independent of their original nesting structure.

This thesis also develops a new transitive dependence analysis algorithm that improves the average complexity of existing algorithms. Although it has the same worst case complexity ($O(N^3)$ for a graph with N vertices) as existing algorithms to compute all-pairs transitive dependences, the new algorithm can compute dependence

summaries to a single destination (or from a single source) vertex in linear time on many dependence graphs encountered in practice. This result makes it possible for production compilers to incorporate transitive dependence analysis as a standard component in addition to traditional dependence analysis techniques.

1.3.3 Computation Slicing

Building on the dependence hoisting transformation in Section 1.3.1 and the extended dependence model in Section 1.3.2, this section introduces a new transformation framework, computation slicing, that systematically combines dependence hoisting with other inexpensive techniques to optimize applications for better locality. This framework hierarchically applies dependence hoisting at each loop level of the original code to effectively achieve loop interchange, fusion and blocking optimizations on arbitrarily nested loops.

To achieve loop interchange, the framework first uses transitive dependence information to find all the loops that can be shifted to the outermost position of a code segment C . It then shifts these loops outside by applying a sequence of dependence hoisting transformations. Each dependence hoisting can be seen as partitioning the original code segment into a set of *slices* — each slice executes a single fused iteration of a group of loops. The framework is thus named a *computation slicing* framework, and each group of loops is named a *computation slice* (or *slice*).

To achieve loop fusion, the framework fuses the computation slices at each loop level before using these slices for dependence hoisting transformations. Because each computation slice contains a set of loops that can be shifted to a given loop level, merging these computation slices permits an aggressive fusion of loops at different loop levels in the original code. The framework employs traditional dependence based loop fusion heuristics [33, 31] to select the best computation slices for fusion.

To achieve loop blocking, the framework combines dependence hoisting with traditional loop strip-mining. To block a loop ℓ_f , the framework first strip-mines ℓ_f into a strip-counting loop ℓ_c and a strip-enumerating loop ℓ_t . It then uses the strip-enumerating loop ℓ_t as the input code segment for further dependence hoisting transformations. These dependence hoisting transformations in turn shift a new set of loops outside loop ℓ_t but inside loop ℓ_c , thus blocking loop ℓ_f .

1.3.4 Recursion Transformation

To exploit locality for multiple levels of the memory hierarchy simultaneously, this thesis automatically transforms arbitrary loop structures into recursive forms that have a hierarchy of smaller working sets. This divide-and-conquer property is illustrated in the recursive formulation of matrix multiplication shown in Figure 1.4. At each recursive step of this formulation, the input matrix is broken into four smaller sub-matrices, and the original computation is broken into multiplications of the sub-

$$C = A \times B$$

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Figure 1.4 Recursive formulation of matrix multiplication

matrices. Each multiplication of sub-matrices has essentially the same data access patterns as those of the original multiplication, but with a working set that is four times smaller. Each sub-matrix is then further divided at the second recursive level, and so on. This recursive algorithm thus has a hierarchy of smaller working sets and is similar to a code simultaneously blocked at many different levels.

Besides the multi-level blocking property, the recursion transformation also achieves an implicit loop fusion effect when it places multiple loop nests into a single recursive procedure. In this case, as the working set of a recursive call becomes sufficiently reduced, all the data accesses within the recursive call can stay in the smallest cache and thus can be reused. Because these data accesses include the ones from different loop nests in the original code, an implicit fusion optimization is achieved for these loop nests. The recursion transformation is thus similar to the computation slicing framework in Section 1.3.3 in that it also incorporates both loop blocking and fusion optimizations for locality.

Although the dependence hoisting technique in Section 1.3.1 can be extended to convert ordinary loop nests into recursive form, the recursion transformation builds on a different transformation technique called *iteration space slicing*, first proposed by Pugh and Rosser [48, 49]. Given an iteration instance I_0 of a statement s_0 , iteration space slicing uses transitive dependence information to compute the iteration set of another statement s_1 , a set that must be executed before (backward iteration space slicing) or after (forward iteration space slicing) the given iteration I_0 of s_0 . Given a sequence of consecutive loop nests, the recursion transformation first uses iteration space slicing to identify groups of related statement instances that must be executed together within each recursive call. It then creates procedures to recursively execute these statement instances. Finally, it transforms the original code to include initial calls to the created recursive procedures.

In the recursion transformation framework, the iteration sets of statements are represented as symbolic integer sets, and the transitive dependence information between statements is translated into integer set mappings before these mappings are used to relate the iteration sets of statements. An integer set programming tool, the Omega library [28], is used to manipulate these symbolic sets and mappings. Because the computation of symbolic integer sets and mappings is quite expensive, the recursion transformation incurs much higher compile-time overhead than the dependence hoisting transformation and computation slicing framework, an overhead which is

discussed in more detail in Section 6.5.

1.4 Related Work

This thesis has essentially proposed two transformation frameworks to improve the cache locality of applications: a computation slicing framework that systematically combines dependence hoisting with traditional unimodular and single loop transformations, and a recursion transformation framework that transforms loop nests into recursive divide-and-conquer procedures to exploit locality simultaneously at multiple levels of the memory hierarchy. Section 1.4.1 compares the computation slicing framework with existing other loop transformation frameworks. Section 1.4.2 presents the related work of the recursion transformation framework.

1.4.1 Loop Transformation Techniques

A set of unimodular and single loop transformations, such as loop blocking, fusion, distribution, interchange, skewing and index-set splitting [50, 35, 38, 15, 24, 52, 11], have been proposed to improve locality of applications. These techniques are inexpensive and are widely used in production compilers. However, these techniques are not effective enough when transforming complicated loop nests that cannot be translated into sequences of perfectly nested loops. Wolf and Lam [50] proposed an uniform algorithm to select compound sequences of unimodular loop transformations

for non-perfectly nested loops. This algorithm is still limited by the original loop structures of programs.

The computation slicing framework in this thesis extends these traditional techniques to effectively transform complex loop nests independent of their original nesting structure. Although it does not incorporate the entire solution space of loop transformations, the framework has demonstrated high effectiveness by blocking some of most challenging benchmarks.

Several general loop transformation frameworks [34, 42, 2, 36, 49] are theoretically more powerful but are also more expensive than the computation slicing framework proposed in this thesis. These general frameworks typically adopt a mathematical formulation of program dependences and loop transformations. They first compute a mapping from the iteration spaces of statements into some unified space. The unified space is then considered for transformation. Finally, a new program is constructed by mapping the selected transformations of the unified space onto the iteration spaces of statements. The computation of these mappings is expensive and generally requires special integer programming tools such as the Omega library [28]. Because of their high cost, these frameworks are rarely used in commercial compilers. In contrast, the computation slicing framework seeks simpler yet effective solutions with a much lower compile-time overhead.

The general frameworks discussed above vary in their efficiency and effectiveness.

In particular, Pugh [42] proposed a framework that finds a schedule for each statement describing the moment each statement instance will be executed. This technique requires an expensive step to find a feasible mapping from iteration spaces of statements into instance execution *time*. Kodukula, Ahmed and Pingali [34] proposed an approach called *data shackling*, in which a tiling transformation on a loop nest is described in terms of a tiling of key arrays in the loop nest. Here a mapping from the iteration spaces of statements into the *data* spaces must be computed. Lim, Cheong and Lam [36] proposed a technique called *affine partition*, which maps instances of instructions into the *time* or *processor* space via an affine expression in terms of the loop index values of their surrounding loops. Finally, Ahmed, Mateev and Pingali [2] proposed an approach that embeds the iteration spaces of statements into a *product space*. The product space is then transformed to enhance locality. In this approach, a mapping from iteration spaces of statements into the *product space* must be computed.

The computation slicing framework in this thesis is a compromise that trades a small amount of generality for substantial gains of efficiency. This framework does not manipulate any mathematically formulated symbolic space. Instead, it relates the iterations of loops directly using a matrix of dependence directions and distances. Although it is less general than the above mathematically formulated frameworks, the computation slicing framework is powerful enough for a large class of real-world applications and is much more efficient. This framework can be combined with tra-

ditional transformation systems for simple loop nests and, because it is inexpensive, is suitable for inclusion in commercial production compilers.

The loop model in this thesis is similar to the one adopted by Ahmed, Mateev and Pingali [2]. They represent the same loop surrounding different statements as different loops and use a product space to incorporate all the extra loop dimensions. This work uses far fewer extra loop dimensions. The computation slicing framework temporarily adds one extra loop at each dependence hoisting transformation and then removes the extra dimension immediately.

Pugh and Rosser was the first to propose using transitive dependence information for transforming arbitrarily nested loops [44, 30]. They represent transitive dependences using integer set mappings and apply an enhanced Floyd-Warshall algorithm to summarize the complete dependence paths between statements. Their dependence representation and transitive analysis algorithm are quite expensive. This thesis has proposed a different dependence representation and transitive analysis algorithm, both of which are much more efficient. These improvements make transitive dependence analysis fast enough for incorporation in production compilers.

Finally, the computation slicing framework in this thesis can be further extended to integrate many other compiler techniques for optimizing memory performance of simple loop nests. These techniques include automatic selection of blocking factors [15, 35, 41], heuristics for loop fusion [33, 31], multi-level memory hierarchy

management [13], and data layout rearrangement transformations [43].

1.4.2 Recursion Transformation

To our knowledge, this thesis is the first compiler work that automatically converts loop nests into recursive form. Researchers have been transforming loop nests into recursive divide-and-conquer algorithms by hand for both single-processor and shared-memory multiprocessor applications [27, 9, 23, 25, 6]. These experiences have demonstrated a variety of experimental benefits of the recursion transformation. Frigo et al. [23] have also provided consistent theoretical results. These results motivated this work which develops compiler support that would make the recursion transformation more widely accessible to application programmers.

The recursion transformation framework developed by this thesis uses integer sets and mappings from the Omega library to relate symbolic iteration sets of statements. This framework thus has a compilation overhead similar to that of the general loop transformation frameworks discussed in Section 1.4.1, an overhead which is much higher than that of the computation slicing framework. For detailed compile-time measurements of the two frameworks, see Section 6.5.

The iteration space slicing technique used in the recursion transformation framework by this thesis was first proposed by Pugh and Rosser [49]. They have applied the technique to fuse related iterations of statements for locality. The recursion

transformation framework uses *iteration space slicing* to achieve a recursive blocking transformation instead.

Compared with the traditional unimodular and single loop transformation techniques discussed in Section 1.4.1, the recursion transformation is essentially a form of blocking, with two differences. First, it combines blocking for multiple levels of memory hierarchy into a single transformation. Second, it unifies both blocking and loop fusion when applied to multiple loop nests.

A significant disadvantage of recursive algorithms is the overhead of recursive procedure calls, an overhead which is much higher than that of loop blocking. To balance this overhead, the recursion transformation must use larger block sizes than a standard loop blocking transformation, which can afford very small block sizes when blocking for registers [6]. It is thus beneficial to use loop blocking within the base-case code of a recursive procedure to achieve small size blocking and to use recursion transformation only to achieve the effect of multi-level blocking.

Finally, the automatic recursion transformation can play an important complementary role to several recursive data-reordering techniques [14, 40]. For example, Chatterjee et al. [14] demonstrated that recursive reordering of data produces significant performance benefits on modern memory hierarchies. They indicate that recursive control structures are needed to fully exploit the potential of recursive data layout. Conversely, the recursion transformation by this thesis can especially bene-

fit from such data reorganizing techniques because the data layout would match the computation order. This field is a rich avenue for the further research of this thesis.

1.5 Organization

The remainder of this thesis is organized as follows. Chapter 2 first presents an extended dependence model for transforming arbitrarily nested loops. Chapter 3 introduces a new transformation, dependence hoisting, to facilitate the fusion and interchange of arbitrarily nested loops. Chapter 4 presents the computation slicing framework that systematically applies dependence hoisting for better locality. Chapter 5 presents the recursion transformation framework. Chapter 6 presents experimental results, and conclusions are drawn in Chapter 7.

Chapter 2

Extended Dependence Model

Compilers must preserve the original meanings of applications, and they typically model this safety requirement using a dependence graph that includes all the statements of a code segment as vertices. A dependence edge is put from statement s_1 to s_2 in the graph if s_1 must be executed before s_2 . If a program transformation does not reverse the direction of any dependence edge, the transformation is guaranteed to preserve the original semantics of the program. If statements s_1 and s_2 are both surrounded by loops, each dependence edge between s_1 and s_2 also defines a set of relations that must hold between the iterations of these loops in order for the dependence to exist. Compilers can then use the dependence relations to determine the safety of transforming these loops.

In traditional unimodular and single loop transformation systems, each dependence edge from s_1 to s_2 is associated with a vector that defines a direction or distance relation between iterations of each common loop surrounding both s_1 and s_2 . This chapter extends this model with a new dependence representation, *Extended Direction Matrix (EDM)*, which defines direction and distance relations between iterations of non-common loops as well. The extended dependence model then summarizes the complete transitive dependence information between statements and uses the sum-

marized information to determine the safety of transforming arbitrarily nested loops (see Chapter 3) or to relate iteration sets of statements (see Chapter 5).

The dependence model in this chapter is less powerful but also much less expensive than those adopted by existing more general transformation frameworks for complex loop nests [34, 42, 2, 36, 49]. These frameworks use integer set mappings [28] to precisely represent the dependence relations between iterations of statements and thus are able to incorporate the whole solution space of loop transformations. However, the integer set mappings are quite expensive and can incur exponential cost when used in summarizing transitive dependences. Although the EDM representation of dependences is less precise than these symbolic integer set mappings, it is sufficient for optimizing a large class of real-world applications and has a much lower cost.

To compute the transitive dependence information between two statements s_1 and s_2 , an algorithm needs to summarize the dependence relations along all paths from s_1 to s_2 in the dependence graph. Transitive dependence analysis is thus a path summary problem rather than a simple reachability problem on directed graphs. Rosser and Pugh have proposed an enhanced Floyd-Warshall algorithm to solve the all-pairs path summary problem up front [44]. This algorithm has $O(N^3)$ complexity for all graphs with N vertices. To reduce the overhead of transitive dependence analysis, this chapter develops a new demand-driven algorithm that summarizes the dependence paths to a single destination or from a single source vertex. Although it also has

$O(N^3)$ worst case complexity to compute all-pairs path summaries, this algorithm performs much better in the average case and can summarize single destination (or source) vertex path summaries in linear time for many dependence graphs encountered in practice.

Note that transitive dependence information is only necessary when transforming complex loop structures that cannot be translated into sequences of perfectly nested loops. Because the safety of transforming perfect loop nests can be determined based on individual dependence edges alone, it is often more economic to do without the extra cost of transitive dependence analysis. A compiler thus should examine the original loop structures of programs and perform transitive dependence analysis only when required.

The following elaborates both the EDM representation of dependences and the improved transitive dependence analysis algorithm. Section 2.1 first presents notations and definitions for modeling dependences and loop transformations. These notations are used throughout this thesis. Section 2.2 then introduces the EDM representation of dependences. Section 2.3 defines a set of operations on the EDM representation and on sets of EDMs. Building on these operations, Section 2.4 presents the transitive dependence analysis algorithm, which was also earlier published in [54].

2.1 Notations and Definitions

To transform loops without being limited by their original nesting structure, this thesis adopts a loop model similar to that used by Ahmed, Mateev and Pingali [2]. The model uses the notation $\ell(s)$ to denote a loop ℓ surrounding some statement s and thus treats loop $\ell(s)$ as different from loop $\ell(s')$ ($s' \neq s$). This strategy effectively treats each loop ℓ surrounding multiple statements as potentially distributable. Dependence analysis can be used at any time to determine whether or not each loop can actually be distributed.

Each statement s inside a loop is executed multiple times; each execution is called an *iteration instance* of s . Suppose that statement s is surrounded by m loops $\ell_1, \ell_2, \dots, \ell_m$. For each loop $\ell_i(s)$ ($i = 1, \dots, m$), the iteration index variable of ℓ_i is denoted as $Ivar(\ell_i(s))$, and the iteration range is denoted as $Range(\ell_i(s))$. Each value I of $Ivar(\ell_i(s))$ defines a set of iteration instances of statement s , a set that can be expressed as $Range(\ell_1) \times \dots \times Range(\ell_{i-1}) \times I \times \dots \times Range(\ell_m)$. This iteration set is denoted as *iteration* $I : \ell_i(s)$ or *the iteration* I of loop $\ell_i(s)$.

In order to focus on transforming loops, this thesis treats all the other control structures in a program as primitive statements; that is, this work does not transform any loops inside other control structures such as conditional branches. This strategy is adopted to simplify the technical presentation of this thesis. To optimize real-world applications, preparative transformations such as “if conversion” [4]

must be incorporated to remove the non-loop control structures in between loops. These preparative transformations are outside the scope of this thesis and will not be discussed further.

Throughout this thesis, the notation $d(s_x, s_y)$ is used to denote the set of dependence edges from a statement s_x to s_y in the dependence graph, and the notation $td(s_x, s_y)$ is used to denote the set of transitive dependence edges from s_x to s_y . Each transitive dependence edge in $td(s_x, s_y)$ corresponds to a path in the dependence graph and has the exact same representation as that of each dependence edge in $d(s_x, s_y)$.

2.2 Dependence Representations

This section introduces a new dependence representation, *Extended Direction Matrix (EDM)*, to model the dependence conditions associated with each path in a dependence graph. Suppose that two statements s_x and s_y are surrounded by m_x loops $(\ell_{x1}, \ell_{x2}, \dots, \ell_{xm_x})$ and m_y loops $(\ell_{y1}, \ell_{y2}, \dots, \ell_{ym_y})$ respectively. A dependence EDM from s_x to s_y is an $m_x \times m_y$ matrix D . Each entry $D[i, j]$ ($1 \leq i \leq m_x, 1 \leq j \leq m_y$) in the matrix specifies a relation between iterations of loops $\ell_{xi}(s_x)$ and $\ell_{yj}(s_y)$. The dependence represented by D satisfies the conjunction of all these conditions.

For each dependence EDM D from statement s_x to s_y , the dependence condition between loops $\ell_{xi}(s_x)$ and $\ell_{yj}(s_y)$ is denoted as $D(\ell_{xi}, \ell_{yj})$. Each condition $D(\ell_{xi}, \ell_{yj})$ can have the following values: “ $= n$ ”, “ $\leq n$ ”, “ $\geq n$ ” and “ $*$ ”, where n is a small integer

called an *alignment factor*. The first three values “ $= n$ ”, “ $\leq n$ ” and “ $\geq n$ ” specify that the dependence conditions are $Ivar(\ell_{xi}) = Ivar(\ell_{yj}) + n$, $Ivar(\ell_{xi}) \leq Ivar(\ell_{yj}) + n$ and $Ivar(\ell_{xi}) \geq Ivar(\ell_{yj}) + n$ respectively; the last value “ $*$ ” specifies that the dependence condition is always true. The *dependence direction* (“ $=$ ”, “ $\leq$ ”, “ $\geq$ ” or “ $*$ ”) of the condition is denoted as $Dir(D(\ell_{xi}, \ell_{yj}))$ and the *alignment factor* of the condition is denoted as $Align(D(\ell_{xi}, \ell_{yj}))$.

A dependence EDM extends the traditional dependence vector representation [4, 53] by computing a relation between iterations of two loops $\ell_x(s_x)$ and $\ell_y(s_y)$ even if $\ell_x \neq \ell_y$. The extra information can be computed using the same traditional dependence analysis techniques [4, 53, 8], which are well-understood and will not be further discussed in this thesis.

2.3 Operations on Dependences

This section defines a set of operations on the dependence EDM representation described in Section 2.2. These operations are required by the transitive dependence analysis algorithm presented in Section 2.4. In the following, Section 2.3.1 defines operations on individual dependence EDMs, and Section 2.3.2 defines operations on sets of EDMs.

2.3.1 Operations on Dependence EDMs

This section defines two operators, *concatenation*($\cdot$) and comparison ($\leq$), on individual dependence EDMs. Because each EDM models a path in the dependence graph, these operators can be used to summarize different dependence paths in the dependence graph, as shown in Section 2.3.2.

The first operator, *concatenation*($\cdot$), is applied to two EDMs, D_{xy} and D_{yz} , where D_{xy} represents a dependence path p_1 from statement s_x to s_y , and D_{yz} represents a path p_2 from statement s_y to s_z . The concatenation of D_{xy} and D_{yz} ($D_{xy} \cdot D_{yz}$) represents the dependence path p_1p_2 along $s_x \rightarrow s_y \rightarrow s_z$. Suppose that statements s_x, s_y and s_z are surrounded by m_x, m_y and m_z loops respectively. The *concatenation* of D_{xy} ($m_x \times m_y$ matrix) and D_{yz} ($m_y \times m_z$ matrix) is an $m_x \times m_z$ matrix D_{xz} satisfying

$$D_{xz}[i, j] = \bigwedge_{1 \leq k \leq m_y} (D_{xy}[i, k] \cdot D_{yz}[k, j]), \quad (2.1)$$

where the *concatenation*($\cdot$) and conjunction ($\wedge$) operators of dependence conditions are defined in Figure 2.1.

In Equation (2.1), the concatenation of dependence conditions $D_{xy}[i, k]$ and $D_{yz}[k, j]$ ($D_{xy}[i, k] \cdot D_{yz}[k, j]$) computes dependence condition for the path $\ell_{xi} \rightarrow \ell_{yk} \rightarrow \ell_{zj}$, where ℓ_{xi} , ℓ_{yk} and ℓ_{zj} are the i th, k th and j th loops surrounding statements s_x , s_y and s_z respectively. For example, suppose $D_{xy}[i, k] = “\leq -1”$ and $D_{yz}[k, j] = “\leq 0”$. Given two arbitrary iterations $I : \ell_{xi}$ (iteration I of loop ℓ_{xi}) and $J : \ell_{zj}$, the depen-

Figure 2.1 Operations on dependence conditions

dependence path from I to J exists only if there is an iteration $K : \ell_{yk}$ that is also involved in the dependence path; that is, the dependence exists only if iterations $I(\ell_{xi})$ and $K(\ell_{yk})$ satisfy condition $D_{xy}[i, j]$ ($I \leq K - 1$) and if iterations $K(\ell_{yk})$ and $J(\ell_{zj})$ satisfy condition $D_{yz}[k, j]$ ($K \leq J$). Thus the relation $I \leq K - 1 \leq J - 1$ must hold; that is, $D_{xy}[i, k] \cdot D_{yz}[k, j] = "\leq -1"$. Note that the dependence condition between loops ℓ_{xi} and ℓ_{zj} must involve at least one iteration of each loop surrounding statement s_y ; otherwise, the involved iteration set for s_y is empty, indicating that the dependence path does not exist. Therefore each dependence condition $D_{xz}[i, j]$ in Equation (2.1) must conjoin the dependence conditions for every path $\ell_{xi} \rightarrow \ell_{yk} \rightarrow \ell_{zj}$, where ℓ_{yk} is a loop surrounding statement s_y .

The second operator, comparison ($\leq$) of dependence EDMs, is applied to two dependence EDMs D_1 and D_2 that both represent dependence paths from some statement s_x to s_y . Suppose that s_x and s_y are surrounded by m_x loops and m_y loops

respectively. The following equation defines the $\leq$ operator on EDMs:

$$D_1 \leq D_2 \Leftrightarrow D_1[i, j] \leq D_2[i, j] \quad \forall 1 \leq i \leq m, 1 \leq j \leq n \quad (2.2)$$

Here the comparison ($\leq$) operator for the dependence conditions $D_1[i, j]$ and $D_2[i, j]$ is defined as

$$D_1[i, j] \leq D_2[i, j] \Leftrightarrow D_1[i, j] \wedge D_2[i, j] = D_1[i, j], \quad (2.3)$$

and the conjunction ($\wedge$) operator for dependence conditions is defined in Figure 2.1(b).

If $D_1[i, j] \wedge D_2[i, j] = D_1[i, j]$, the dependence relation defined by $D_1[i, j]$ is part of that defined by $D_2[i, j]$; that is, the loop iterations satisfying the condition $D_1[i, j]$ is a subset of those satisfying $D_2[i, j]$. Equation (2.2) thus specifies that an EDM D_1 is subsumed by D_2 (that is, $D_1 \leq D_2$) only if for every pair of loops, $\ell_i(s_x)$ and $\ell_j(s_y)$, the dependence condition $D_1(\ell_i, \ell_j)$ is subsumed by the condition $D_2(\ell_i, \ell_j)$ (that is, $D_1(\ell_i, \ell_j) \leq D_2(\ell_i, \ell_j)$). Because a dependence path represented by D_1 (similarly for D_2) must satisfy all the conditions in D_1 , if $D_1 \leq D_2$, the dependence conditions modeled by D_1 are a subset of those modeled by D_2 , in which case the dependence path represented by D_1 is redundant (subsumed by D_2) and can be ignored.

The comparison operators on dependence conditions and EDMs defined in Equations (2.3) and (2.2) organize the complete set of dependence conditions and EDMs into lattices. In particular, since there are only four different dependence directions ($=$, $\leq$, $\geq$ and $*$), these directions form a lattice of depth 3, shown in Figure 2.1(c). However, the lattice of dependence conditions is infinite because there are infinite

numbers of different integer alignment factors. As a result, the lattice of dependence EDMs also becomes infinite.

To limit the total number of different dependence conditions and EDMs, this thesis restricts the absolute magnitude of dependence alignment factors to be less than a small constant (for example, 5) and uses approximations for conditions with alignment factors of magnitudes larger than the threshold. Because in practice most loops are misaligned with each other by only 1-2 iterations, this restriction rarely sacrifices the generality of loop transformations. Moreover, this restriction limits the total number of different dependence conditions with a small constant and thus significantly improves the efficiency of summarizing dependence paths. Because the depth of any loop nest is typically bounded by a small constant, the total number of different EDMs is also limited, making it possible for most operations on EDMs to finish within some constant time.

2.3.2 Operations on Dependence EDM Sets

This section defines three operators: $\text{union}(\cup)$, $\text{concatenation}(\cdot)$ and transitive closure($*$), on sets of dependence EDMs. These operations are required by the transitive dependence analysis algorithm described in Section 2.4. Each dependence set $td(s_x, s_y)$ includes one or more dependence paths between the same source and sink statements. Thus all the dependence EDMs in a dependence set have the same row

and column dimensions.

The first operator, union($\cup$), is applied to two EDM sets $td_1(s_x, s_y)$ and $td_2(s_x, s_y)$ that both include dependence paths from some statement s_x to s_y . The union of $td_1(s_x, s_y)$ and $td_2(s_x, s_y)$ includes all the dependence paths in either $td_1(s_x, s_y)$ or $td_2(s_x, s_y)$ and thus is defined as

$$td_1(s_x, s_y) \cup td_2(s_x, s_y) = \{D \mid D \in td_1(s_x, s_y) \text{ or } D \in td_2(s_x, s_y)\}. \quad (2.4)$$

Note redundant EDMs can be removed to simplify the above set-union result, say, $td(s_x, s_y)$: if two EDMs D_1 and D_2 both belong to $td(s_x, s_y)$, and D_1 is subsumed by D_2 ($D_1 \leq D_2$ in Equation (2.2)), D_1 is redundant and therefore can be removed from $td(s_x, s_y)$. In practice, each set usually contains only a few EDMs after simplification. Thus most operations on dependence EDM sets can finish within some constant time.

The second operator, concatenation ($\cdot$) of EDM sets, is applied to two dependence EDM sets $td(s_x, s_y)$ and $td(s_y, s_z)$, where $td(s_x, s_y)$ includes dependence paths from statement s_x to s_y , and $td(s_y, s_z)$ includes dependence paths from statement s_y to s_z . The concatenation of $td(s_x, s_y)$ and $td(s_y, s_z)$ summarizes every dependence path $s_x \rightarrow s_y \rightarrow s_z$, a path that starts from s_x to s_y following an EDM $D_1 \in td(s_x, s_y)$ and then continues from s_y to s_z following an EDM $D_2 \in td(s_y, s_z)$. The concatenation of $td(s_x, s_y)$ and $td(s_y, s_z)$ thus is defined as

$$td(s_x, s_y) \cdot td(s_y, s_z) = \{D_{xy} \cdot D_{yz} \mid D_{xy} \in td(s_x, s_y) \text{ and } D_{yz} \in td(s_y, s_z)\}, \quad (2.5)$$

where the concatenation of dependence paths ($D_{xy} \cdot D_{yz}$) is defined in Equation (2.1).

The third operator, transitive closure ($*$), is applied to a single EDM set $td(s, s)$ that includes dependence paths from a statement s to itself. This operator summarizes all the cycles from statement s to itself formed by paths in the EDM set $td(s, s)$ and is thus defined as

$$td(s, s)^* = td(s, s) \cup (td(s, s) \cdot td(s, s)) \cup (td(s, s) \cdot td(s, s) \cdot td(s, s)) \cup \dots \quad (2.6)$$

Here the infinite numbers of unions and concatenations stop when a fixed point is reached. In addition, the result $td(s, s)^*$ is simplified at each union operation of the EDM sets. Because there are only a finite number of different EDMs (see Section 2.3.1), each transitive closure operation is guaranteed to reach a fixed point and terminate after a constant number of set unions and concatenations.

2.4 Transitive Dependence Analysis Algorithm

This section introduces a transitive dependence analysis algorithm that summarizes the complete path information to a single destination vertex (the algorithm to summarize dependence paths from a single source vertex can be constructed similarly). This algorithm is independent of specific dependence representations. In particular, the algorithm can be applied using the EDM representation of dependence paths described in Section 2.2.

Figure 2.3 shows the transitive analysis algorithm, which has three steps. The

first step transforms an arbitrary dependence graph into an acyclic graph (a DAG) by splitting a set of vertices. The second step then summarizes the acyclic paths in the DAG. Finally, the third step extends the acyclic path summaries to include cycles in the original graph. Section 2.4.1 uses an example to illustrate these steps. Section 2.4.2, 2.4.3, and 2.4.4 then describe the three steps respectively. Finally, the complexity of the entire algorithm is discussed in Section 2.4.5.

2.4.1 A Simple Example

This section illustrates the transitive dependence analysis algorithm in Figure 2.3 using a simple dependence graph shown in Figure 2.2(a). To summarize transitive dependences for this graph, the algorithm performs the following steps.

First, to break the dependence cycles in (a), the algorithm visits all the vertices in the graph in depth-first order and identifies all the *back edges* that go back to already visited vertices. The algorithm then breaks the dependence cycles by modifying these back edges. Suppose that the vertex v_1 in (a) is visited first and that edge e_5 is identified as a back edge. The algorithm splits vertex v_1 into v_1 and a twin vertex v'_1 and then changes edge e_5 to go into v'_1 instead. The transformed graph is shown in (b). Here because v'_1 has no outgoing edges, the transformed graph no longer has dependence cycles.

The algorithm then pre-computes the cycle information (denoted as $C(v_1, v'_1)$)

Figure 2.2 Example for transitive dependence analysis algorithm

for vertex v_1 so that the broken edge at v_1 can be recovered later. To compute $C(v_1, v'_1)$, the algorithm first computes the single-destination path summaries to v'_1 on the transformed DAG in (b) in reverse topological order. The computed path summaries include $td(v_4, v'_1) = \{EDM(e_5)\}$, $td(v_3, v'_1) = EDM(e_4) \cdot td(v_4, v'_1)$, $td(v_2, v'_1) = EDM(e_3) \cdot td(v_4, v'_1)$, and $td(v_1, v'_1) = (EDM(e_1) \cdot td(v_2, v'_1)) \cup (EDM(e_2) \cdot td(v_3, v'_1))$. The cycle information $C(v_1, v'_1)$ is then set to be $td(v_1, v'_1)^*$, the transitive closure of $td(v_1, v'_1)$.

The algorithm is now ready to compute path summaries for the original dependence graph in (a). This graph is equivalent to the graph shown in (c), which has a conceptually dummy edge from vertex v'_1 to v_1 . All the cycle information thus can be recovered by reconnecting this dummy edge that has been broken in (b). For example, to compute path summary from v_3 to v_4 in the original graph, the algorithm first computes $td(v_3, v_4)$ on the transformed DAG in (b), which yields $td(v_3, v_4) = \{EDM(e_4)\}$. To recover the dummy edge $v'_1 \rightarrow v_1$, the algorithm recomputes $td(v_3, v_4)$ as $\{EDM(e_4)\} \cup (td(v_3, v'_1) \cdot C(v_1, v'_1) \cdot td(v_1, v_4))$, where $td(v_3, v'_1) = EDM(e_4) \cdot EDM(e_5)$, $C(v_1, v'_1) = ((EDM(e_1) \cdot EDM(e_3) \cdot EDM(e_5)) \cup (EDM(e_2) \cdot$

```

Trans-Dep-Anal( $G, v$ )
   $G$ : dependence graph;  $v$ : destination vertex
  if ( $G$  has not been preprocessed) then
    Preprocess-Cycles( $G$ )
  Path-Summary-On-Cyclic-Graphs( $G, v$ )

Preprocess-Cycles( $G$ )
  Tarjan-SCC( $G, SCCs, BackEdges$ )
  for (each edge  $e : p \rightarrow v$  in  $BaseEdges$ ) do
     $v' = \text{Split-vertex}(v)$ ; Change  $e$  to go to  $v'$ 
  for each  $scc \in SCCs$  with split vertices  $(v_1, v'_1), \dots, (v_m, v'_m)$ 
    for  $i = 1, m$  do
      Path-Summary-On-DAG( $scc, v'_i$ )
      for  $k = 1, i - 1$  do
        for each vertex  $p \in scc$  do
           $td(p, v'_i) = td(p, v'_i) \cup td(p, v'_k) \cdot C(v_k, v'_k) \cdot td(v_k, v'_i)$ 
         $C(v_i, v'_i) = td(v_i, v'_i)^*$ 

Path-Summary-On-DAG( $G, v$ )
   $G$ : acyclic dependence graph
   $v$ : destination vertex
   $td(v, v) = \{ \text{identity EDM of statement } v \}$ 
  for each statement  $p$  in  $G$  in reverse topological order
     $td(p, v) = \emptyset$ 
    for each edge  $e : p \rightarrow q$  in  $G$  do
       $td(p, v) = td(p, v) \cup \{ EDM(e) \} \cdot td(q, v)$ 

Path-Summary-On-Cyclic-Graphs( $G, v$ )
   $G$ : dependence graph
   $v$ : destination vertex
  for each  $scc \in SCCs(G)$  in reverse topological order do
    Path-Summary-On-DAG( $G, v$ )
    for (each split vertex  $(v_k, v'_k)$  in  $scc$ ) do
      for (each vertex  $p \in scc$ ) do
         $td(p, v) = td(p, v) \cup td(p, v'_k) \cdot C(v_k, v'_k) \cdot td(v_k, v)$ 

```

Figure 2.3 Transitive dependence analysis algorithm

$EDM(e_4) \cdot EDM(e_5))^*$, and $td(v_1, v_4) = EDM(e_2) \cdot EDM(e_4)$.

2.4.2 Preprocessing Cycles

This section now describes the first step of the transitive dependence analysis algorithm, a step encoded by the function *Preprocess-Cycles* in Figure 2.3. This step transforms an arbitrary dependence graph into a directed acyclic graph (a DAG) and then pre-computes information so that the broken cycles may be recovered later.

To break all the cycles in the original graph G , the algorithm first uses the well-known Tarjan SCC algorithm (function *Tarjan-SCC* in Figure 2.3) to find all the strongly connected components (SCCs) in G . The Tarjan SCC algorithm uses a stack to track the traversal of vertices in G and by doing so, also identifies all the *back edges* that go to already visited vertices in each SCC. The transitive analysis algorithm then translates each SCC into a DAG by breaking these back edges. For

each vertex v_i that has an incoming back edge $p \rightarrow v_i$, the algorithm splits v_i into v_i and a new vertex v'_i as the twin vertex of v_i . It then changes all the back edges to v_i to go to v'_i instead. Because all the back edges now go to the new twin vertices which have no outgoing edges, all the original cycles are broken. The transformed dependence graph thus becomes a DAG.

After transforming the original dependence graph into a DAG, the algorithm then pre-computes some information so that the broken cycles can be recovered later. For each strongly connected component in the original graph, suppose that m vertices are split and that the twin vertices are $(v_1, v'_1), \dots, (v_m, v'_m)$. For each pair of twin vertices v_i and v'_i ($1 \leq i \leq m$), the following cycle information is pre-computed:

- $td(p, v'_i) \forall$ vertex p in SCC: path summary from p to v'_i , including all the original cycles involving vertices $v_1, \dots, v_{i-1}$.
- $C(v_i, v'_i) = (td(v_i, v'_i))^*$: path summary from v_i to itself, including all the original cycles involving vertices $v_1, \dots, v_i$.

Section 2.4.4 describes how to use the above information to recover broken cycles.

To pre-compute the cycle information for each split vertex v_i , the algorithm first computes the single-destination path summaries to v'_i on the transformed DAG. It then extends each path summary $td(p, v'_i)$ to include all paths $p \rightsquigarrow v'_k \rightarrow v_k \rightsquigarrow \dots \rightsquigarrow v'_k \rightarrow v_k \rightsquigarrow v'_i \forall 1 \leq k < i$. Here $v'_k \rightarrow v_k$ is conceptually a dummy edge from v'_k to

v_k , as shown in Figure 2.2(c), and $v_k \rightsquigarrow \dots \rightsquigarrow v'_k$ includes all the cycles involving $v_1, \dots, v_k$, the cycles that have already been computed correctly.

2.4.3 Path Summary On DAG

The section describes the second step of the transitive dependence analysis algorithm, a step encoded by the function *Path-Summary-On-DAG* in Figure 2.3. This function summarizes dependence paths into a single destination vertex v on a directed acyclic graph G . The algorithm is straightforward and has time complexity linear to the size of the graph.

To compute $td(p, v)$ for each vertex $p \in G$, the algorithm first initiates the self transitive dependence of v ($td(v, v)$) using an identity EDM. Suppose v is surrounded by m loops, the identity EDM for v is a $m \times m$ matrix D_{mm} that satisfies the following condition:

$$DI[i, j] = \begin{cases} = 0 & \text{if } i = j; \\ * & \text{if } i \neq j, \forall 1 \leq i, j \leq m \end{cases} \quad (2.7)$$

The above definition for DI guarantees that $DI \cdot D_1 = D_1$ for each $m * n$ EDM D_1 and $D_2 \cdot DI = D_2$ for each $n * m$ EDM D_2 .

The algorithm then computes $td(p, v)$ for each vertex p in the reverse topological order of the DAG. For each edge $e : p \rightarrow q$ in the dependence DAG, $EDM(e)$ denotes the EDM associated with e . Because the vertices in the DAG are traversed in reverse topological order, for each edge $e : p \rightarrow q$, $td(q, v)$ has already been computed

correctly. The algorithm thus computes $td(p, v)$ as the union of $EDM(e) \cdot td(q, v)$ for each edge $e : p \rightarrow q$ leaving vertex p .

2.4.4 Path Summary on Cyclic Graphs

This section describes the last step of the transitive dependence analysis algorithm in Figure 2.3, a step encoded by the function *Path-Summary-On-Cyclic-Graphs*. This function summarizes the dependence paths to a single destination vertex v on an arbitrary dependence graph G . The algorithm computes path summaries for each strongly connected component of G in reverse topological order, which guarantees that when summarizing paths for each SCC, all the transitive dependence information for other SCCs closer to v have already been computed correctly.

The algorithm computes path summaries for each SCC in two steps. The first step computes all the path summaries on the transformed DAG of the SCC. The second step then extends each path summary $td(p, v)$ with recovered broken cycles. For each split twin vertex pair (v_k, v'_k) of the SCC, the dummy edge from v'_k to v_k is recovered for $td(p, v)$ using the following equation:

$$td(p, v) = td(p, v) \cup td(p, v'_k) \cdot C(v_k, v'_k) \cdot td(v_k, v) \quad (2.8)$$

This equation extends $td(p, v)$ with all paths $p \rightsquigarrow v'_k \rightsquigarrow \dots \rightsquigarrow v_k \rightsquigarrow v$, where the paths $v'_k \rightsquigarrow \dots \rightsquigarrow v_k$ includes all cycles involving $v_1, \dots, v_k$, and the paths $p \rightsquigarrow v'_k$ and $v_k \rightsquigarrow v$ include all cycles involving $v_1, \dots, v_{k-1}$.

2.4.5 Correctness and Complexity

To prove that the transitive analysis algorithm in Figure 2.3 is correct, this section demonstrates that the algorithm satisfies two conditions. First, the function *Path-Summary-On-DAG* computes the correct summaries for all paths in the transformed DAG. Second, after these path summaries are extended with the pre-computed cycle information, they also include all the cycles in the original graph.

By induction, it is straightforward to show that each path summary $td(p, v)$ is computed correctly on the transformed DAG. Because the vertices are traversed in reverse topological order of the DAG, for each edge e from vertex $p \rightarrow q$, $td(q, v)$ has been already computed correctly. Since the dependence paths from p to v includes the concatenation of each edge $e : p \rightarrow q$ with $td(q, v)$, $td(p, v)$ also includes all the paths from p to v and thus is computed correctly.

To prove that each final path summary $td(p, v)$ includes all the cycles in the original graph, it is sufficient to show that all the broken cycles have been recovered in $td(p, v)$. Since these cycles are broken when the twin vertices are split, they can be recovered by adding a dummy edge from each new vertex v'_i to its original vertex v_i . For each pair of twin vertices (v_i, v'_i) , the pre-computed cycle information $C(v_i, v'_i)$ summarizes all the cycles involving split vertices $v_1, \dots, v_k$, where $k \leq i$. Thus the collection of $C(v_1, v'_1), \dots, C(v_m, v'_m)$ summarizes all the cycles in the original dependence graph. After extending $td(p, v)$ with these summaries, $td(p, v)$ also includes all the original

cycles. The path summary $td(p, v)$ thus is computed correctly.

Given a dependence graph G with V vertices and E edges, suppose that at most M ($M \leq V$) vertices are split for each strongly connected component (SCC) of G . From Figure 2.3, the worst case complexity of finding SCCs and creating twin vertices is $O(V + E)$. The complexity of pre-computing cycle information for each SCC is $O(VM^2)$. Therefore the complexity of function *Preprocess-Cycles* is $O(V + E + VM^2)$. The worst case complexity of *Path-Summary-On-DAG* is $O(V + E)$. The complexity of recovering cycles is $O(VM)$. Therefore the worst case complexity of the whole algorithm (function *Path-Summary-On-Cyclic-Graphs* in Figure 2.3) is $O(V + E + VM)$.

The number of split vertices thus determines the worst case complexity of the transitive analysis algorithm in Figure 2.3. Although M (the largest number of split vertices in all SCCs) is $O(V)$ in the worst case (e.g., for an SCC that has size $O(V)$ and is fully connected), in practice, the dependence graphs are not so densely connected. Often only a small number of vertices need to be split to break all the cycles in a SCC. Furthermore, the transitive analysis algorithm in Figure 2.3 can be configured with a restriction on number of split vertices. In cases where an unreasonably large number of vertices need to be split, the algorithm can simply give up and assume some *bottom* value for all the transitive path summaries. In practice, this strategy does not degrade effectiveness of compilers because for a dependence graph that is so densely connected,

usually no legal transformation is possible anyway. Moreover, this strategy bounds the value of M by a constant, so both *Preprocess-Cycles* and *Path-Summaries-On-Cyclic-Graphs* would require time that is linear in the size of the graph, i.e., $O(V + E)$. Consequently, the transitive dependence analysis algorithm would only require time $O(V^2 + VE)$ to summarize transitive dependence edges between all pairs of statements, making it possible for the analysis to be incorporated into commercial production compilers.

Chapter 3

Dependence Hoisting

This chapter introduces a new loop transformation, *dependence hoisting*, that facilitates the direct fusion and interchange of arbitrarily nested loops. This transformation first selects a group of arbitrarily nested loops that can be fused and then shifted to the outermost position of a code segment containing these loops. It then performs the transformation through a compound sequence of traditional transformations on single loops and perfectly nested loops.

Given a group of loops as input for a dependence hoisting transformation, the safety of fusing and shifting these loops is determined from the dependence constraints on iterations of these loops. If the group is a single loop in the original code, traditional loop interchange analysis for perfect loop nests would suffice; however, if the group includes non-common loops surrounding different statements, transitive dependence analysis is performed on the dependence graph and the transitive dependences are used to determine the safety of fusing and shifting these loops. Because the safety analysis of transforming perfectly nested loops is a well-understood topic [4, 53], this chapter focuses on using transitive dependences to determine the safety of dependence hoisting transformation.

Because dependence hoisting is realized by combining a sequence of traditional

loop distribution, interchange and index set splitting transformations on single or perfectly nested loops, the complexity of applying dependence hoisting is equivalent to that of the corresponding sequence of sub-transformations. In the worst case, applying dependence hoisting to a loop nest takes time proportional to $N^2 + L^2D$, where N is the number of statements in the nest, L is the depth of the nest, and D is the size of the dependence graph for the nest. In average case, however, dependence hoisting requires much less time to finish. For a perfect loop nest, dependence hoisting is equivalent to a standard loop interchange on perfect loop nests followed by a single-loop distribution, in which case the required complexity is $O(N + D)$.

This chapter presents both safety analysis and transformation algorithms for dependence hoisting. Section 3.1 first introduces notations and definitions for the transformation. Section 3.2 then uses an example to illustrate how to determine the legality of the transformation using transitive dependences. Finally, Section 3.3 presents both the analysis and transformation algorithms for dependence hoisting.

3.1 Notations and Definitions

This chapter inherits all the notations and definitions presented in Section 2.1. These notations include using $\ell(s)$ to denote a loop ℓ surrounding some statement s , using $Ivar(\ell(s))$ to denote the iteration index variable of loop $\ell(s)$, using $Range(\ell(s))$ to denote the iteration range of loop $\ell(s)$, and using $I : \ell(s)$ to denote an iteration

instance I of loop $\ell(s)$. The set of dependence edges from statement s_x to s_y is denoted as $d(s_x, s_y)$, and the set of transitive dependence edges from s_x to s_y is denoted as $td(s_x, s_y)$.

Each dependence hoisting transformation has an argument that specifies the set of loops to be fused and then shifted to the outermost position of a code segment. The transformation can be seen as partitioning the original computation into slices — each slice executes a single fused loop iteration of the statements. The set of loops to be fused at the outermost position is thus denoted as a *computation slice* (or *slice*), and each loop to be fused is denoted as a *slicing loop*.

Each computation slice specifies a dependence hoisting transformation for some code segment C . For each statement s inside C , the computation slice selects a loop ℓ surrounding s as the slicing loop for s and then selects a small integer as the *alignment factor* for the slicing loop. Statement s is also called the *slicing statement* of loop ℓ . The computation slice thus has the following attributes:

- *stmt-set*: the set of statements in the slice;
- *slice-loop*(s) $\forall s \in \text{stmt-set}$: for each statement s in *stmt-set*, the slicing loop for s ;
- *slice-align*(s) $\forall s \in \text{stmt-set}$: for each statement s in *stmt-set*, the alignment factor for *slice-loop*(s).

Given the above computation slice, a dependence hoisting transformation fuses all

the slicing loops into a single loop ℓ_f at the outermost position of C s.t.

$$\forall \ell(s) = \text{slice-loop}(s), \text{Ivar}(\ell_f(s)) = \text{Ivar}(\ell(s)) + \text{slice-align}(s). \quad (3.1)$$

Here loop $\ell(s)$ is the slicing loop for s , and $\text{Ivar}(\ell(s))$ and $\text{Ivar}(\ell_f(s))$ are the index variables for loops $\ell(s)$ and $\ell_f(s)$ respectively. Equation (3.1) specifies that each iteration instance I of loop $\ell(s)$ is executed at iteration $I + \text{slice-align}(s)$ of the fused loop ℓ_f after transformation. Since the fused loop is placed at the outermost position of C , the outermost loop level of C is also called the loop level of the computation slice.

Note that a valid computation slice for C must satisfy the following three conditions:

- it includes all the statements in C ;
- all of its slicing loops can be legally shifted to the outermost loop level;
- each pair of slicing loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused s.t. $\text{Ivar}(\ell_x) + \text{slice-align}(s_x) = \text{Ivar}(\ell_y) + \text{slice-align}(s_y)$.

If a computation slice satisfies all the above conditions, the corresponding dependence hoisting transformation does not violate any dependence constraint. If C is a single loop nest, the outermost loop of C can always be legally chosen as the slicing loops (with alignment 0) for all the statements in C , so C always has at least one valid computation slice.

Given the input loop nest C , loop distribution can be applied to distribute C into a sequence of nests $C_1, \dots, C_m$, where each distributed nest C_i ($i = 1, \dots, m$) can have one or more computation slices. Multiple slices with disjunct sets of statements thus can be constructed for C . However, to maintain the same constraints in the dependence graph of C , the statements in each distributed segment C_i must belong to different strongly connected components (SCCs) than the statements in C_j ($j \neq i$). Thus there can be no dependence cycle connecting two computation slices with disjunct sets of statements. This property of disjunct computation slices can be used to facilitate the fusion of these slices, a transformation which will be discussed in Chapter 4.

3.2 Transformation Safety Analysis

This section uses transitive dependence information to resolve the safety of applying dependence hoisting transformation at the outermost loop level of a given code segment C . Since each transformation is driven by a computation slice (see Section 3.1), the legality of dependence hoisting can be modeled as the validity of computation slices.

As discussed in Section 3.1, to determine whether a computation slice is valid, it is necessary to resolve the safety of two loop transformations: shifting an arbitrary loop $\ell(s)$ to the outermost loop level and fusing two arbitrary loops $\ell_x(s_x)$ and $\ell_y(s_y)$ at the outermost loop level (the fused loop will be placed at the outermost loop

level). Section 3.2.1 presents the legality conditions for these two transformations. Section 3.2.2 then uses the non-pivoting LU code in Figure 1.1 to illustrate the safety analysis for dependence hoisting.

3.2.1 Interchange and Fusion Analysis

To decide whether an arbitrary loop $\ell(s)$ can be legally shifted to the outermost loop level, this thesis examines the self transitive dependence $td(s, s)$ and concludes that the shifting is legal if the following equation holds:

$$\forall D \in td(s, s), D(\ell, \ell) = "=" \text{ or } "\leq", \text{ where } n \leq 0. \quad (3.2)$$

The above equation indicates that each iteration I of loop $\ell(s)$ depends only on itself or previous iterations of loop $\ell(s)$. Consequently, placing loop $\ell(s)$ at the outermost loop level of statement s is legal because no dependence cycle connecting s is reversed by this transformation.

To decide whether two loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused at the outermost loop level, this thesis examines the transitive dependences $td(s_x, s_y)$ and $td(s_y, s_x)$. It concludes that the fusion is legal if the following equation holds:

$$\begin{aligned} \forall D \in td(s_y, s_x), Dir(D(\ell_y, \ell_x)) &= "=" \text{ or } "\leq" \text{ and} \\ \forall D \in td(s_x, s_y), Dir(D(\ell_x, \ell_y)) &= "=" \text{ or } "\leq" \end{aligned} \quad (3.3)$$

If Equation (3.3) holds, the two loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be fused into a single loop

$\ell_f(s_x, s_y)$ s.t.

$$Ivar(\ell_f) = Ivar(\ell_x) = Ivar(\ell_y) + align, \quad (3.4)$$

where *align* is a small integer and is called the *alignment factor for loop ℓ_y* . The value of *align* must satisfy the following equation:

$$\begin{aligned} a_y \leq align \leq -a_x, \text{ where} \\ a_x = \text{Max}\{ \text{Align}(D(\ell_y, \ell_x)), \forall D \in td(s_y, s_x) \}, \\ a_y = \text{Max}\{ \text{Align}(D(\ell_x, \ell_y)), \forall D \in td(s_x, s_y) \}. \end{aligned} \quad (3.5)$$

From Equation (3.4), each iteration I of the fused loop $\ell_f(s_x, s_y)$ executes both the iteration $I : \ell_x(s_x)$ (iteration I of loop $\ell_x(s_x)$) and the iteration $I - align : \ell_y(s_y)$. From Equation (3.3) and (3.5), iteration $I : \ell_x(s_x)$ depends on the iterations $\leq I + a_x : \ell_y(s_y)$, which are executed by the iterations $\leq I + a_x + align$ of loop $\ell_f(s_y)$. Similarly, iteration $I - align : \ell_y(s_y)$ depends on the iterations $\leq I - align + a_y : \ell_x(s_x)$, which are executed by the same iterations of loop $\ell_f(s_x)$. Since $a_y \leq align \leq -a_x$ from Equation (3.5), we have $I + align + a_x \leq I$ and $I - align + a_y \leq I$. Each iteration I of the fused loop $\ell_f(s_x, s_y)$ thus depends only on itself or previous iterations. Consequently, no dependence direction is reversed by this fusion transformation.

3.2.2 Example: Non-pivoting LU

This section illustrates the safety analysis of dependence hoisting using the *KJI* form of non-pivoting LU in Figure 1.1(b). Figure 3.1 shows the dependence and tran-

(a) dependence edges

$$\begin{aligned}
 d(s_1, s_2) &= \left\{ \begin{matrix} k(s_2) & j(s_2) & i(s_2) \\ k(s_1) & \begin{pmatrix} \leq 0 & \leq -1 & \leq -1 \\ \geq 1 & \bullet & = 0 \end{pmatrix} \end{matrix} \right\} \\
 d(s_2, s_1) &= \left\{ \begin{matrix} k(s_1) & i(s_1) & \\ k(s_2) & \begin{pmatrix} \leq -1 & \leq -2 \\ = 0 & \leq -1 \\ \geq 1 & = 0 \end{pmatrix} \end{matrix} \right\} \\
 d(s_2, s_2) &= \left\{ \begin{matrix} k(s_2) & j(s_2) & i(s_2) & k(s_2) & j(s_2) & i(s_2) \\ k'(s_2) & \begin{pmatrix} \leq -1 & \leq -2 & \leq -2 \\ \geq 1 & = 0 & \bullet \end{pmatrix} & \begin{pmatrix} \leq -2 & \leq -3 & \leq -3 \\ = 0 & \leq -1 & \leq -1 \\ \geq 1 & \bullet & = 0 \end{pmatrix} \end{matrix} \right\} \\
 &\quad \left\{ \begin{matrix} j'(s_2) & \begin{pmatrix} \geq 1 & = 0 & \bullet \\ \geq 1 & \bullet & = 0 \end{pmatrix} & \begin{pmatrix} \leq -2 & \leq -3 & \leq -3 \\ \geq 1 & = 0 & \bullet \end{pmatrix} \\ k'(s_2) & \begin{pmatrix} \leq -2 & \leq -3 & \leq -3 \\ \geq 1 & = 0 & \bullet \end{pmatrix} & \begin{pmatrix} \leq -2 & \leq -3 & \leq -3 \\ \geq 1 & = 0 & \bullet \end{pmatrix} \\ j'(s_2) & \begin{pmatrix} \geq 1 & = 0 & \bullet \\ \geq 1 & \bullet & = 0 \end{pmatrix} & \begin{pmatrix} \leq -2 & \leq -3 & \leq -3 \\ \geq 1 & = 0 & \bullet \end{pmatrix} \\ i'(s_2) & \begin{pmatrix} = 0 & \leq -1 & \leq -1 \end{pmatrix} & \begin{pmatrix} = 0 & \leq -1 & \leq -1 \end{pmatrix} \end{matrix} \right\}
 \end{aligned}$$

(b) transitive dependence edges

$$\begin{aligned}
 td(s_1, s_1) &= \left\{ \begin{matrix} k(s_1) & i(s_1) \\ k'(s_1) & \begin{pmatrix} \leq -1 & \leq -2 \\ \geq 1 & = 0 \end{pmatrix} \end{matrix} \right\} \\
 td(s_1, s_2) &= \left\{ \begin{matrix} k(s_2) & j(s_2) & i(s_2) & k(s_2) & j(s_2) & i(s_2) \\ k(s_1) & \begin{pmatrix} \leq 0 & \leq -1 & \leq -1 \\ \geq 1 & \bullet & = 0 \end{pmatrix} & \begin{pmatrix} \leq -1 & \leq -2 & \leq -2 \\ \leq 0 & \leq -1 & \leq -1 \end{pmatrix} \end{matrix} \right\} \\
 td(s_2, s_1) &= \left\{ \begin{matrix} k(s_2) & j(s_2) & i(s_2) & k(s_1) & i(s_1) \\ j(s_2) & \begin{pmatrix} \leq -1 & \leq -2 \\ \leq 0 & \leq -1 \\ \geq 1 & = 0 \end{pmatrix} & \begin{pmatrix} \leq -1 & \leq -2 \\ \leq 0 & \leq -1 \\ \geq 1 & = 0 \end{pmatrix} \end{matrix} \right\} \\
 td(s_2, s_2) &= \left\{ \begin{matrix} k(s_2) & j(s_2) & i(s_2) & k(s_2) & j(s_2) & i(s_2) \\ k'(s_2) & \begin{pmatrix} \leq -1 & \leq -2 & \leq -2 \\ \geq 1 & = 0 & \bullet \end{pmatrix} & \begin{pmatrix} \leq -2 & \leq -3 & \leq -3 \\ \geq 1 & \bullet & = 0 \end{pmatrix} \\ j'(s_2) & \begin{pmatrix} \geq 1 & = 0 & \bullet \\ \geq 1 & \bullet & = 0 \end{pmatrix} & \begin{pmatrix} \leq -2 & \leq -3 & \leq -3 \\ \geq 1 & = 0 & \bullet \end{pmatrix} \\ i'(s_2) & \begin{pmatrix} = 0 & \leq -1 & \leq -1 \end{pmatrix} & \begin{pmatrix} = 0 & \leq -1 & \leq -1 \end{pmatrix} \end{matrix} \right\}
 \end{aligned}$$

Figure 3.1 Dependence and transitive dependence edges for non-pivoting LU

<p><i>slice_j</i>:</p> <p><i>stmt-set</i> = {<i>s</i>₁, <i>s</i>₂}</p> <p><i>slice-loop</i>(<i>s</i>₁) = <i>k</i>(<i>s</i>₁)</p> <p><i>slice-align</i>(<i>s</i>₁) = 0</p> <p><i>slice-loop</i>(<i>s</i>₂) = <i>j</i>(<i>s</i>₂)</p> <p><i>slice-align</i>(<i>s</i>₂) = 0</p>	<p><i>slice_k</i>:</p> <p><i>stmt-set</i> = {<i>s</i>₁, <i>s</i>₂}</p> <p><i>slice-loop</i>(<i>s</i>₁) = <i>k</i>(<i>s</i>₁)</p> <p><i>slice-align</i>(<i>s</i>₁) = 0</p> <p><i>slice-loop</i>(<i>s</i>₂) = <i>k</i>(<i>s</i>₂)</p> <p><i>slice-align</i>(<i>s</i>₂) = 0</p>	<p><i>slice_i</i>:</p> <p><i>stmt-set</i> = {<i>s</i>₁, <i>s</i>₂}</p> <p><i>slice-loop</i>(<i>s</i>₁) = <i>i</i>(<i>s</i>₁)</p> <p><i>slice-align</i>(<i>s</i>₁) = 0</p> <p><i>slice-loop</i>(<i>s</i>₂) = <i>i</i>(<i>s</i>₂)</p> <p><i>slice-align</i>(<i>s</i>₂) = 0</p>
--	--	--

Figure 3.2 Computation slices for non-pivoting LU

sitive dependence information for this code using the EDM representation described in Section 2.1. Figure 3.2 shows all the valid computation slices for this code. The following illustrates how to automatically construct these slices using the transitive dependence information in Figure 3.1.

To construct valid computation slices for the non-pivoting LU code, this thesis first selects an arbitrary statement as starting point and finds all the candidate slicing loops for this statement. Suppose that statement *s*₁ is selected. This thesis identifies candidate slicing loops for *s*₁ as each loop $\ell(s_1)$ that can be safely shifted

to the outermost loop level. The safety is resolved by examining the self transitive dependence $td(s_1, s_1)$ in Figure 3.1(b). From $td(s_1, s_1)$, the self dependence conditions are “ ≤ -1 ” for loop $k(s_1)$ and “ $= 0$ or ≤ -1 ” for loop $i(s_1)$; both conditions satisfy Equation (3.2) in Section 3.2.1. Both loops $k(s_1)$ and $i(s_1)$ thus can be legally shifted to the outermost loop level.

This thesis then creates two computation slices, $slice_k$ and $slice_i$, where $slice_k$ selects the slicing loop $k(s_1)$ for statement s_1 , and $slice_i$ selects the slicing loop $i(s_1)$ for s_1 . The alignment factors for both slicing loops are 0. Each slice is then extended with a compatible slicing loop for statement s_2 . From the self transitive dependence $td(s_2, s_2)$ in Figure 3.1(b), all the loops surrounding s_2 (loops $k(s_2)$, $j(s_2)$ and $i(s_2)$) can be legally shifted to the outermost loop level and thus are candidate slicing loops for s_2 .

To further extend the computation slice $slice_k$, this thesis finds a slicing loop for statement s_2 as a loop $\ell(s_2)$ that can be legally fused with loop $k(s_1)$. The legality of fusion is resolved by examining the transitive dependences $td(s_1, s_2)$ and $td(s_2, s_1)$ in Figure 3.1(b). From $td(s_1, s_2)$, the dependence conditions for loop $k(s_1)$ are “ $k(s_1) \leq k(s_2)$ ”, “ $k(s_1) \leq j(s_2) - 1$ ” and “ $k(s_1) \leq i(s_2) - 1$ ”; from $td(s_2, s_1)$, the conditions are “ $k(s_2) \leq k(s_1) - 1$ ”, “ $j(s_2) \leq k(s_1)$ ” and “ $i(s_2) \geq k(s_1) + 1$ or $i(s_2) \leq k(s_1)$ ”. The conditions between loops $k(s_1)$ and $k(s_2)$ satisfy Equation (3.3) in Section 3.2.1; thus these two loops can be legally fused. From Equation (3.5), the

alignment factor for loop $k(s_2)$ is 0 or 1. Similarly, loop $j(s_2)$ can be fused with loop $k(s_1)$ with fusion alignment 0 or -1 . Loop $i(s_2)$ cannot be fused with loop $k(s_1)$ because of the dependence condition " $i(s_2) \geq k(s_1) + 1$ " in $td(s_2, s_1)$.

This thesis then duplicates $slice_k$ with another slice $slice_j$, extends $slice_k$ by selecting the slicing loop $k(s_2)$ for s_2 , and then extends $slice_j$ by selecting the slicing loop $j(s_2)$ for s_2 . The completed slices are shown in Figure 3.2. Figure 3.2 also shows the slice $slice_i$ which initially selects the slicing loop $i(s_1)$ for statement s_1 . Here loop $i(s_2)$ can be legally fused with loop $i(s_1)$ and thus is selected as the slicing loop for statement s_2 .

The three slices in Figure 3.2 can be used to freely translate between any two of the three loop orderings of non-pivoting LU in Figure 1.1. Section 1.3.1 has illustrated how to translate Figure 1.1(a) to (c) using $slice_j$. Similarly, applying a dependence hoisting transformation using $slice_i$ can translate (a) to (b), and using $slice_k$ can translate (c) to (a). The following section describes detailed algorithms to establish the translations.

3.3 Transformation Algorithms

Figure 3.3 summarizes the algorithms for both dependence hoisting analysis and transformation. This section first elaborates the functions in Figure 3.3 and then discusses the correctness proof and complexity of the algorithms.

Hoisting-Analysis($C, \text{slice-set}$)
 (1) $\text{slices-in-construction} = \emptyset$
 s_0 = an arbitrary statement in C
 For each loop $\ell_0(s_0)$ that can be shifted outermost
 Add a new slice slice into $\text{slices-in-construction}$
 Add $\ell_0(s_0)$ into slice :
 $\text{slice-loop}(s_0) = \ell_0(s_0), \text{slice-align}(s_0) = 0$
 (2) If $\text{slices-in-construction} = \emptyset$, stop.
 $\text{slice} = \text{Remove a slice from slices-in-construction}$
 (3) For each statement s in C but not in slice
 For each loop $\ell(s)$ that can be shifted outermost
 and can be fused with other loops in slice
 $\text{align} = \text{alignment factor for } \ell(s)$
 If ($\text{slice-loop}(s) = \emptyset$ in slice)
 Add $\ell(s)$ into slice :
 $\text{slice-loop}(s) = \ell(s)$
 $\text{slice-align}(s) = \text{align}$
 Else $\text{slice}_1 = \text{Duplicate}(\text{slice})$
 Add $\ell(s)$ into slice_1 :
 $\text{slice-loop}(s) = \ell(s)$
 $\text{slice-align}(s) = \text{align}$
 Add slice_1 into $\text{slices-in-construction}$
 If ($\text{slice covers } C$), add slice to slice-set
 Goto step(2)

Hoisting-Transformation(C, slice)
 (1) $\text{range} = \bigcup_{s \in C} (\text{Range}(\text{slice-loop}(s)) + \text{slice-align}(s))$
 ℓ_f = create a new loop with $\text{Range}(\ell_f) = \text{range}$
 Put ℓ_f surrounding C
 For each statement $s \in \text{slice}$ with $\ell = \text{slice-loop}(s)$
 $\text{cond}(s) = \text{"if } (Ivar(\ell_f) = Ivar(\ell) + \text{slice-align}(s))\text{"}$
 Insert $\text{cond}(s)$ surrounding s
 Modify the dependence graph of C .
 (2) For each statement $s \in \text{slice}$ with $\ell(s) = \text{slice-loop}(s)$
 Distribute $\ell(s)$ to enclose only s :
 (2.1) For each s_1 strongly connected with s in $\text{Dep}(\ell)$
 $m = \text{slice-align}(s_1) - \text{slice-align}(s)$
 $\ell_1(s_1) = \text{slice-loop}(s_1)$
 Split s_1 : $Ivar(\ell) < Ivar(\ell_1) + m$,
 $Ivar(\ell) = Ivar(\ell_1) + m$,
 $Ivar(\ell) > Ivar(\ell_1) + m$.
 Modify the dependence graph of C
 (2.2) Distribute loop $\ell(s)$
 While ($\ell(s)$ contains other stmts than s) do
 Interchange loop $\ell(s)$ with the loop inside it
 Modify dependence graph of C
 Distribute loop $\ell(s)$
 (3) For each statement $s \in \text{slice}$ with $\ell(s) = \text{slice-loop}(s)$
 Replace $Ivar(\ell)$ inside loop $\ell(s)$
 Adjust loop ranges between ℓ_f and $\ell(s)$
 Remove $\text{cond}(s)$ and $\ell(s)$

Figure 3.3 Dependence hoisting algorithms

3.3.1 Dependence Hoisting Analysis

The safety analysis for dependence hoisting is encoded in the function *Hoisting-Analysis* in Figure 3.3. This function constructs all the legal computation slices for an input loop nest C and then puts these slices into the output variable set slice-set . The function is separated into three steps.

Step (1) of the function starts from some arbitrary statement s_0 in the input nest C and first finds all the candidate slicing loops for s_0 as each loop $\ell(s_0)$ that can be legally shifted to the outermost loop level. For each found candidate slicing loop $\ell_0(s_0)$, this step creates a computation slice including $\ell_0(s_0)$ and then puts the created

slice into a variable set *slices-in-construction* for further examination.

Step(2) of the function checks the variable set *slices-in-construction* and exits if this set is empty; otherwise, it removes a slice from *slices-in-construction* and extends the slice by going to step(3).

Step (3) completes the partial slice *slice* selected by step (2) and then goes back to step (2) for other partial slices. For each statement *s* not yet in the partial slice *slice*, this step identifies valid slicing loops for *s* as each loop $\ell(s)$ that can be shifted to the outermost loop level and can be fused with the slicing loops already in *slice* (how to determine these conditions is described in section 3.2.1). For each found slicing loop $\ell(s)$, an alignment factor *align* is computed from the fusion alignments between loop $\ell(s)$ and other slicing loops in *slice*. If multiple slicing loops are found, step (3) duplicates the original slice to remember all the extra slicing loops and then adds the duplicated slices into the variable set *slices-in-construction* for further examination. At the end of step (3), if all the statements have been included in *slice*, this slice is completed successfully and thus can be collected into the output variable *slice-set*; otherwise, this slice is thrown away.

3.3.2 Dependence Hoisting Transformation

The dependence hoisting transformation algorithm is encoded by the function *Hoisting-Transformation* in Figure 3.3. This function transforms a given code segment

C using a computation slice $slice$. The transformation process is separated into the following three steps.

Step (1) of the transformation puts a new outermost loop ℓ_f surrounding C and then inserts a conditional $cond(s)$ surrounding each statement s . The new loop ℓ_f unions the iteration ranges of all the slicing loops after proper alignments, and each inserted conditional $cond(s)$ forces the statement s inside it to execute only if $Ivar(\ell_f) = Ivar(slice-loop(s)) + slice-align(s)$. This step then modifies the dependence graph of C to include dependence conditions for the new loop ℓ_f : for each statement s , $Ivar(\ell_f(s))$ has the same dependence conditions as those of $Ivar(slice-loop(s)) + slice-align(s)$. Figure 1.2(b) in Section 1.2 illustrates this modification using the KJI form of non-pivoting LU.

Step (2) of the transformation then distributes each slicing loop $\ell(s)$ so that $\ell(s)$ encloses only its slicing statement s . Step (2.1) first applies loop index-set splitting to remove all the dependence cycles that are incident to statement s and are carried by loop ℓ . For each statement s_1 ($\ell_1(s_1) = slice-loop(s_1)$) strongly connected with statement s in $Dep(\ell)$ (the dependence graph of ℓ), step (2.1) splits s_1 into three statements under the execution conditions $Ivar(\ell) < Ivar(\ell_1) + m$, $Ivar(\ell) = Ivar(\ell_1) + m$ and $Ivar(\ell) > Ivar(\ell_1) + m$ respectively, where $m = slice-align(s_1) - slice-align(s)$. Step (2.2) then repeatedly applies loop distribution and interchange to remove all the dependence cycles that are incident to s and are carried by loops inside $\ell(s)$. At

each iteration of this repetition, step (2.2) first distributes loop $\ell(s)$ so that $Dep(\ell)$ is strongly connected. If $\ell(s)$ still encloses statements other than s , this step interchanges $\ell(s)$ with the loop immediately inside $\ell(s)$ and then tries again. Since steps (2.1) and (2.2) successfully remove all the dependence cycles incident to s in $Dep(\ell)$, eventually the distributed loop $\ell(s)$ encloses only statement s . For proof of these two steps, see section 3.3.3.

Step (3) of the transformation then removes all the distributed slicing loops and the conditionals that synchronize these loops with the new outermost loop ℓ_f . Before removing each slicing loop $\ell(s)$, this step makes two adjustments to the original loop nest: first, it replaces the loop index variable $Ivar(\ell(s))$ with $Ivar(\ell_f) - slice-align(s)$ inside loop $\ell(s)$; second, it adjusts the iteration ranges of the loops between ℓ_f and $\ell(s)$ to ensure the correct iteration set of statement s . Step (3) then removes both the slicing loop $\ell(s)$ and the conditional $cond(s)$ because they have become redundant.

3.3.3 Correctness and Complexity

The hoisting analysis algorithm in Figure 3.3 is correct in that only valid computation slices are collected. Each collected slice satisfies three conditions: first, it includes all the statements in the input loop nest; second, all the slicing loops can be legally shifted to the outermost loop level; third, each pair of slicing loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused after being aligned with $slice-align(s_x)$ and $slice-align(s_y)$

respectively.

The correctness of the hoisting transformation algorithm in Figure 3.3 is proved by demonstrating that the transformed code at each step satisfies two conditions: first, each statement still executes the same set of iteration instances; second, no dependence is reversed in the dependence graph.

Step (1) of the dependence hoisting transformation puts a new loop ℓ_f at the outermost position and then inserts a conditional surrounding each statement. Each conditional forces the statement s inside it to execute the same set of iteration instances by synchronizing the slicing loop $slice-loop(s)$ with the outermost loop ℓ_f in a lock-step fashion. These conditionals shift the loop-carrying levels of a set of dependences to the outermost loop level. Since all the slicing loops can be legally shifted to the outermost loop level and can be legally fused with each other, no dependence direction is reversed.

Step (2) of the transformation algorithm distributes each slicing loop $\ell(s)$ so that $\ell(s)$ encloses only its slicing statement s . After step (1), all the dependence paths between two statements s_x and s_y are now carried by the new outermost loop ℓ_f unless the following condition holds:

$$Ivar(\ell_x) + slice-align(s_x) = Ivar(\ell_y) + slice-align(s_y) \quad (3.6)$$

Here $\ell_x = slice-loop(s_x)$ and $\ell_y = slice-loop(s_y)$. Thus only the dependence paths satisfying Equation (3.6) need to be considered when distributing the slicing loops.

For each slicing loop $\ell(s)$, step (2.1) applies loop index-set splitting to remove all the dependence cycles that are incident to statement s and are carried by loop ℓ in $Dep(\ell)$ (the dependence graph of ℓ). For each statement $s_1 \neq s$ inside loop ℓ , suppose $slice-loop(s_1) = \ell_1$. From Equation (3.6), each dependence EDM D between statements s_1 and s satisfies the condition $D(\ell(s), \ell_1(s_1)) = "= m"$, where $m = slice-align(s_1) - slice-align(s)$. Once statement s_1 is split into three statements s'_1 , s''_1 and s'''_1 under execution conditions $Ivar(\ell) < Ivar(\ell_1) + m$, $Ivar(\ell) = Ivar(\ell_1) + m$ and $Ivar(\ell) > Ivar(\ell_1) + m$ respectively, each dependence EDM between statements s'_1 and s satisfies the condition $Ivar(\ell(s'_1)) \leq Ivar(\ell(s)) - 1$. Since each iteration of loop ℓ can depend only on itself or previous iterations in $Dep(\ell)$, there is no dependence path from statement s to s'_1 in $Dep(\ell)$. Similarly, there is no dependence path from statement s'''_1 to s in $Dep(\ell)$. Thus all the dependence cycles between statements s and s_1 are now between s and s''_1 . Since each dependence EDM between s''_1 and s satisfies the condition $Ivar(\ell(s)) = Ivar(\ell(s''_1))$, no dependence cycle connecting s is carried by loop ℓ .

For each slicing loop $\ell(s)$, step (2.2) of the hoisting transformation algorithm further removes from $Dep(\ell)$ all the dependence cycles that are incident to statement s and are carried by loops inside $\ell(s)$. This step first distributes loop $\ell(s)$ so that $\ell(s)$ does not carry any dependence. If $\ell(s)$ still encloses statements other than s , there must be dependence cycles carried by a loop ℓ' perfectly nested inside $\ell(s)$. step (2.2)

can then shift loop ℓ inside ℓ' to remove these dependence cycles from $Dep(\ell)$. The distribution and interchange repeat until loop $\ell(s)$ encloses only statement s .

Since each slicing loop $\ell(s)$ now encloses only statement s and the conditional synchronizing $\ell(s)$ with the outermost loop ℓ_f , both the slicing loop $\ell(s)$ and the conditional $cond(s)$ can be safely removed. Step(3) of the hoisting transformation removes both the slicing loops and conditionals while making sure each statement still executes the same set of iteration instances. Therefore all the transformation steps are legal.

The following now determines the complexity of the dependence hoisting transformation algorithms. Given an input loop nest C , suppose that the number of statements in the nest is N , the depth of the loop nest is L , and the size of the dependence graph for the loop nest is D . From the function *Hoisting-Analysis* in Figure 3.3, at most L computation slices can be constructed, and the construction of each computation slice examines all pairs of the statements in C . Therefore the worst case complexity for this function is $O(N^2L)$. From the function *Hoisting-Transformation* in Figure 3.3, step (2.1) makes at most N^2 attempts to split the iteration ranges of slicing loops, and step (2.2) makes at most L attempts to distribute each slicing loop. For step (2.2), the worst case happens when all the slicing loops are nested inside one another and when each slicing loop $\ell(s)$ needs to be shifted to the innermost position before it can be successfully distributed. In this case, there are at most L

different slicing loops in the input computation slice, so the worst case complexity of the *Hoisting-Transformation* function is $O(N^2 + L^2D)$.

In average case, however, the complexity of the dependence hoisting transformation is much lower than $O(N^2 + L^2D)$. Most slicing loops can be successfully distributed at the first try, and the distribution of one slicing loop often automatically achieves the distribution of other slicing loops as well. For example, in the case where the group of slicing loops is a single loop that is perfectly nested with the original outermost loop, the transformation needs only to create a new outermost loop ℓ_f , distribute all the slicing loops at once with one loop distribution, and then remove the distributed loops and conditionals. The complexity thus becomes $O(N + D)$.

Chapter 4

Computation Slicing

This chapter develops a framework, computation slicing, that systematically combines the dependence hoisting transformation introduced in Chapter 3 with other inexpensive techniques to optimize applications for better locality. The framework hierarchically considers the code segments at different loop levels and applies the same transformation algorithms at the outermost loop level of each code segment, guaranteeing that no generality is sacrificed.

To optimize applications for better locality, the computation slicing framework uses dependence hoisting to achieve three loop transformations: loop fusion, interchange and blocking. To achieve an aggressive multi-level loop fusion effect, the framework merges multiple computation slices before using them to transform the original code. To achieve loop interchange and blocking, the framework carefully arranges the order of applying dependence hoisting transformations using different computation slices. It then selects a group of computation slices and combines dependence hoisting with loop strip-mining to block the fused loops of these slices. Simple data-reuse analysis is performed to resolve the profitability of the fusion, interchange and blocking transformations.

Note that the computation slicing framework is only a preliminary implementa-

tion to demonstrate the benefit of integrating dependence hoisting with traditional transformation techniques. For example, although dependence hoisting is necessary only when transforming complex loop structures, this framework has substituted dependence hoisting for loop interchange, reversal and fusion transformations in all cases. A production compiler can easily adopt a more selective strategy and apply dependence hoisting only when required. The production compiler can also incorporate more sophisticated profitability analysis models such as those by Kennedy and McKinley [32]. These models can be integrated into the computation slicing framework in a straightforward fashion and will not be further discussed.

To present the framework in more detail, Section 4.1 first defines some notations. Section 4.2 illustrates how to model loop fusion, reversal and interchange in terms of computation slices. Section 4.2 describes how to use dependence hoisting to achieve aggressive loop blocking and fusion optimizations. Section 4.3 then presents the framework that systematically applies dependence hoisting to optimize applications for locality.

4.1 Notations and Definitions

This section defines notations to classify different groups of computation slices. As shown in Chapter 3, each dependence hoisting transformation is driven by a computation slice, which contains information necessary to fuse a set of loops and then

shift the fused loop to the outermost position of some code segment. Each computation slice thus can be seen as representing a single loop, denoted as the *fused loop* of the computation slice. Because modifications to each computation slice immediately result in the corresponding changes to its fused loop after a dependence hoisting transformation, traditional notations for loops can be extended to computation slices as well.

Two computation slices, $slice_1$ and $slice_2$, are defined to be nested if both slices contain the same set of statements; that is, the fused loops of these two slices will be nested inside one another after dependence hoisting transformations. Given a set of slices $slice-set = \{slice_1, \dots, slice_m\}$, if all the slices contain the same set of statements, this group of slices is denoted as a *slice nest*; that is, the fused loops of these slices will form a loop nest after dependence hoisting transformations. The nesting order of these slices is defined as the nesting order of the corresponding fused loops.

Two computation slices, $slice_1$ and $slice_2$, are defined to be disjunct if they contain disjunct sets of statements. Similarly, a sequence of computation slices $slice_1, \dots, slice_m$ is defined to be disjunct if every pair of computation slices in the sequence is disjunct. Given a sequence of disjunct computation slices, there can be no dependence cycle connecting these slices in the dependence graph at the current loop level (see Section 3.1). These slices thus can be further fused to achieve a loop fusion effect on

<pre> Comp-Slice-Fusable(<i>Dep</i>, <i>slice</i>₁, <i>slice</i>₂, <i>align</i>) <i>align</i> = -∞ For (each dependence EDM <i>D</i> from <i>s</i>_{<i>z</i>} ∈ <i>slice</i>₁ to <i>s</i>_{<i>y</i>} ∈ <i>slice</i>₂) <i>ℓ</i>_{<i>z</i>} = <i>slice</i>₁:<i>slice-loop</i>(<i>s</i>_{<i>z</i>}); <i>ℓ</i>_{<i>y</i>} = <i>slice</i>₂:<i>slice-loop</i>(<i>s</i>_{<i>y</i>}); If (<i>Dir</i>(<i>D</i>(<i>ℓ</i>_{<i>z</i>}, <i>ℓ</i>_{<i>y</i>})) ≠ "=" or ≤), return false <i>align</i>_{<i>y</i>} = <i>Align</i>(<i>D</i>(<i>ℓ</i>_{<i>z</i>}, <i>ℓ</i>_{<i>y</i>})) + <i>slice-align</i>(<i>s</i>_{<i>z</i>}) - <i>slice-align</i>(<i>s</i>_{<i>y</i>}) <i>align</i> = max(<i>align</i>, <i>align</i>_{<i>y</i>}) return true </pre>	<pre> Fuse-Comp-Slice(<i>slice</i>₁, <i>slice</i>₂, <i>align</i>) <i>slice</i> = create a new computation slice For (each statement <i>s</i> ∈ <i>slice</i>₁) do Add <i>s</i> into <i>slice</i>: <i>slice-loop</i>(<i>s</i>) = <i>slice</i>₁ : <i>slice-loop</i>(<i>s</i>); <i>slice-align</i>(<i>s</i>) = <i>slice</i>₁ : <i>slice-align</i>(<i>s</i>) for (each statement <i>s</i> ∈ <i>slice</i>₂) do Add <i>s</i> into <i>slice</i>: <i>slice-loop</i>(<i>s</i>) = <i>slice</i>₂ : <i>slice-loop</i>(<i>s</i>); <i>slice-align</i>(<i>s</i>) = <i>slice</i>₂ : <i>slice-align</i>(<i>s</i>) + <i>align</i> return <i>slice</i> </pre>
--	--

Figure 4.1 Fusing computation slices

the original code. A fusion algorithm is presented in Section 4.2.1.

Besides loop fusion, two other traditional loop transformations, loop interchange and reversal, can also be achieved by rearranging computation slices. Section 4.2 and 4.3 present in more detail how to achieve all three transformations.

4.2 Rearranging Computation Slices

This section illustrates how to rearrange computation slices to model three loop transformations: fusion, interchange and reversal. The following subsections describe how to model each of these transformations respectively.

4.2.1 Achieving Loop Fusion

This section describes how to fuse two disjunct computation slices. Because each computation slice fuses a set of loops that can be shifted to the same loop level, fusing two disjunct computation slices automatically achieves the fusion of the slicing loops in both slices.

Figure 4.1 presents two algorithms: one determines whether two computation

slices can be legally fused, and the other performs the actual fusion of the two computation slices.

The function *Comp-Slice-Fusable* in Figure 4.1 determines whether two disjunct computation slices at the same loop level can be legally fused. Because there is no dependence cycle connecting the two slices (see Section 4.1), this function assumes that there are only dependence edges from $slice_1$ to $slice_2$ in the dependence graph *Dep* (if the opposite is true, the two arguments can be switched without sacrificing any generality). The algorithm examines each dependence edge D from a statement $s_x \in slice_1$ to $s_y \in slice_2$. If the dependence condition from loop ℓ_x ($slice-loop(s_x)$ in $slice_1$) to loop ℓ_y ($slice-loop(s_y)$ in $slice_2$) has a direction that is neither $=$ nor $\leq$, the dependence edge will be reversed after fusion, and the fusion is not legal; otherwise, the dependence edge does not prevent the two slicing loops from being fused, in which case the algorithm restricts the fusion alignment *align* for the computation slice $slice_2$ so that

$$align \geq Align(D(\ell_x, \ell_y) + slice-align(s_x) - slice-align(s_y)). \quad (4.1)$$

If the algorithm succeeds in finding a valid fusion alignment *align* after examining all the dependence edges, the two computation slices should be fused so that

$$Ivar(\ell_{f1}) = Ivar(\ell_{f2}) + align. \quad (4.2)$$

where ℓ_{f1} and ℓ_{f2} represent the fused loops of $slice_1$ and $slice_2$ respectively. Thus the computation slice $slice_2$ needs to be aligned by the factor *align* before being fused

with $slice_1$.

To prove that the fusion algorithm in Figure 4.1 is correct, the following shows that after fusing $slice_1$ and $slice_2$ according to Equation (4.2), no dependence edge from $slice_1$ to $slice_2$ is reversed in the dependence graph. First, for each pair of slicing loops, $\ell_x(s_x) \in slice_1$ and $\ell_y(s_y) \in slice_2$, the following equations hold from the definition of a computation slice (in Section 3.1):

$$Ivar(\ell_{f1}) = Ivar(\ell_x) + slice-align(s_x) \quad (4.3)$$

$$Ivar(\ell_{f2}) = Ivar(\ell_y) + slice-align(s_y) \quad (4.4)$$

After substituting the above two equations for ℓ_{f1} and ℓ_{f2} in Equation (4.2), the following relation between the slicing loops ℓ_x and ℓ_y is satisfied:

$$Ivar(\ell_x) + slice-align(s_x) = Ivar(\ell_y) + slice-align(s_y) + align \quad (4.5)$$

Now consider each dependence EDM D from s_x to s_y . Because the fusion alignment $align$ satisfies Equation (4.1), substituting this inequality for $align$ in Equation (4.5) obtains

$$\begin{aligned} Ivar(\ell_x) + slice-align(s_x) &\leq Ivar(\ell_y) + slice-align(s_y) + \\ &\quad Align(D(\ell_x, \ell_y)) + slice-align(s_x) - slice-align(s_y), \end{aligned}$$

which is equivalent to

$$Ivar(\ell_x) \leq Ivar(\ell_y) + Align(D(\ell_x, \ell_y)) \quad (4.6)$$

The above equation indicates that the original dependence condition between ℓ_x and ℓ_y is maintained after fusing $slice_1$ and $slice_2$. Therefore no dependence direction will be reversed by the fusion transformation.

The function *Fuse-Comp-Slice* in Figure 4.1 performs the actual fusion of the two computation slices $slice_1$ and $slice_2$. The algorithm first creates a new empty computation slice and then clones both $slice_1$ and $slice_2$ using the new slice. Before adding each statement s of $slice_2$ into the fused slice, the algorithm adjusts the slicing alignment factor for s with the fusion alignment *align* so that the fusion relation specified by Equation (4.1) is satisfied.

4.2.2 Achieving Loop Reversal

This section describes how to determine whether the slicing loops in a computation slice can be reversed. If the answer is “yes”, the reversal transformation can then be used to facilitate the valid fusion of two computation slices. For example, if the fused loops of two computation slices have different enumeration orders, they cannot be legally fused unless one of the computation slices reverse the enumeration order of its fused loop. In this case, a loop reversal transformation is applied to all the slicing loops of the reversed slice.

To determine whether a computation slice can be legally reversed, this thesis examines the self transitive dependence $td(s, s)$ of each slicing loop $\ell(s)$ (if the slicing

loops are a single loop in the original code, traditional loop reversal analysis would suffice). Because a valid computation slice contains only slicing loops that can be shifted to the outermost loop level of a code segment, the self transitive dependence conditions for each slicing loop contain only $\leq$ or $=$ directions. If the self transitive dependence conditions for all the slicing loops have only the $=$ direction, the enumeration orders of all these loop can be safely reversed. In this case, the computation slice is denoted as reversible.

4.2.3 Achieving Loop Interchange

This section describes how to achieve loop interchange by rearranging a set of nested computation slices. Here because each slice represents a set of loops that can be fused into a single loop, interchanging the nesting order of two slices corresponds directly to the interchange of the two sets of slicing loops.

Given two nested slices $slice_1$ and $slice_2$, to shift the slicing loops in $slice_2$ outside of the ones in $slice_1$, this thesis first uses $slice_1$ to perform a dependence hoisting transformation. It then uses the fused loop ℓ_{f1} of $slice_1$ as the input code for another dependence hoisting transformation using $slice_2$, which in turn shifts the fused loop of $slice_2$ outside loop ℓ_{f1} .

In general, to achieve the desired nesting order of m computation slices: $slice_1$, $slice_2$, ..., $slice_m$, this thesis first uses $slice_m$ to drive a dependence hoisting trans-

formation, which shifts the fused loop ℓ_m of $slice_m$ to the outermost loop level. It then uses $slice_{m-1}$ to shift the fused loop ℓ_{m-1} of $slice_{m-1}$ outside of ℓ_m , and so on. The desired nesting order of m computation slices thus can be achieved using m dependence hoisting transformations.

4.3 Optimizing Loop Nests

This section presents algorithms that applies the strategies described in Section 4.2 to optimize an arbitrary code segment at the outermost loop level. This section focuses on two optimizations, blocking and multi-level fusion, to enhance the locality of the input code segment. Figure 4.2 presents the algorithms for these two optimizations. Section 4.3.1 illustrates the blocking algorithm using a version of non-pivoting LU code. Section 4.3.2 then illustrates the multi-level fusion algorithm using a subroutine from an application benchmark, *Erlebacher* (see Table 6.1).

4.3.1 Block a Loop Nest

In Figure 4.2, the function *Block-Loop-Nest* uses a set of nested computation slices to block a loop nest C for better locality. The blocking is achieved by combining a sequence of loop strip-mining and dependence hoisting transformations using the given slices.

Before blocking the input loop nest C , the algorithm first determines a nesting

Block-Loop-Nest ($C, slice\text{-}set$) Arrange desired nesting order in $slice\text{-}set$ $C_1 = C$ For (each $slice \in slice\text{-}set$ in reverse nesting order) Hoisting-Transformation ($C_1, slice$) $C_1 =$ the new fused loop ℓ_f of $slice$ If (strip-mine $slice$) then strip-mine loop ℓ_f into ℓ_c and ℓ_t Shift ℓ_c to the outermost loop level of C $C_1 = \ell_t$	Fuse-Slice-Nest ($slice\text{-}set_1, slice\text{-}set_2$) (1) $fuse\text{-}set = \emptyset$ For (each $slice_1 \in slice\text{-}set_1$) For (each $slice_2 \in slice\text{-}set_2$) If (Comp-Slice-Fusable ($slice_1, slice_2, align$)) $slice = \text{Fuse-Comp-Slice}(slice_1, slice_2, align)$ Add $slice$ into $fuse\text{-}set$ Remove $slice_1$ from $slice\text{-}set_1$ Remove $slice_2$ from $slice\text{-}set_2$ Add slices in $slice\text{-}set_1$ and $slice\text{-}set_2$ into $left\text{-}over(fuse\text{-}set)$ return $fuse\text{-}set$
--	--

Figure 4.2 Blocking and multi-level fusion algorithms

order for the given computation slices in $slice\text{-}set$. Here simple profitability analysis is performed to place the slicing loops that carry fewer data reuses outside of the ones that carry more reuses [51, 3]. The algorithm then removes each computation slice from $slice\text{-}set$ in the reverse of the desired nesting order for these slices, and each slice is used to drive a dependence hoisting transformation.

The algorithm uses a variable C_1 to keep track of the input loop nest for each dependence hoisting transformation. After each dependence hoisting transformation using a computation slice $slice_i$, if the new fused loop ℓ_f should not be blocked, the algorithm sets C_1 to be ℓ_f so that further hoisting transformations will shift loops outside of ℓ_f ; otherwise, the algorithm strip-mines ℓ_f into a strip-counting loop ℓ_c and a strip-enumerating loop ℓ_t . It then uses loop ℓ_t as the input loop nest C_1 for further dependence hoisting transformations, which in turn will shift a new set of loops outside loop ℓ_t but inside loop ℓ_c , thus blocking loop ℓ_f . After strip-mining ℓ_f at the outermost level of C_1 , the algorithm also shifts the strip-counting loop ℓ_c to the current outermost level of the original code segment C . This step guarantees that ℓ_c

Figure 4.3 Blocking non-pivoting LU

is nested outside the strip-counting loops obtained earlier, thus achieving the desired nesting order of the strip-counting loops as well.

To illustrate the blocking algorithm, Figure 4.3 shows the steps to block the *KJI* form of non-pivoting LU in Figure 1.1(a). The computation slices for this code are shown in Figure 3.2. Specifically, three computation slices are constructed: $slice_i = \{i(s_1), i(s_2)\}$, $slice_j = \{k(s_1), j(s_2)\}$, and $slice_k = \{k(s_1), k(s_2)\}$. The alignment factors for all the slicing loops are 0.

To block the non-pivoting LU code, the algorithm first decides a nesting order for the three computation slices. Suppose that the original nesting order $slice_k \rightarrow slice_j \rightarrow slice_i$ is chosen. Also suppose that the fused loops of $slice_j$ and $slice_i$ should

be blocked. The algorithm first uses $slice_i$ to transform the original code and then strip-mines the fused loop of $slice_i$. The result of the transformation is shown in Figure 4.3(a). This code has the same loop ordering as that of the IKJ form in Figure 1.1(b) except that the outermost i loop is strip-mined. The algorithm then uses $slice_j$ to shift the $j(s_2)$ and $k(s_1)$ loops outside the strip-enumerating i loop in Figure 4.3(a), and the result is shown in (b). The code in Figure 4.3(b) now has a JIK loop ordering similar to that of the JKI form of non-pivoting LU in Figure 1.1. The algorithm then uses $slice_k$ to further shift loops outside the strip-enumerating j loop in Figure 4.3(b), and the result is shown in (c). The code in (c) has the same loop ordering as that of the original KJI form of non-pivoting LU in Figure 1.1 except that the loops $j(s_2)$, $k(s_1)$, $i(s_1)$ and $i(s_2)$ are now blocked. In (c), both the row and column directions of the matrix are blocked, as shown in the pictorial illustration.

Note that the conditional “ $if (k \geq x_j)$ ” in (c) can be removed by splitting the iteration range of the k loop. The splitting can be achieved by integrating a loop index-set splitting step into the cleanup phase of the transformation framework, as discussed in Section 4.4.1.

4.3.2 Multi-level Fusion

This section presents how to adapt the traditional typed-fusion algorithm [37, 33] to fuse a sequence of disjunct computation slice nests. Because these

slice nests can be seen as representing a sequence of disjunct loop nests, the fusion algorithm can be seen as fusing all the loop nests at once, achieving a multi-level fusion effect.

In Figure 4.2, the function *Fuse-Slice-Nest* performs the actual fusion of two disjunct computation slice nests, $slice-set_1$ and $slice-set_2$. For each pair of computation slices, $slice_1 \in slice-set_1$ and $slice_2 \in slice-set_2$, the algorithm first invokes function *Comp-Slice-Fusable* defined in Figure 4.1 to determine whether the two slices can be legally fused. If the answer is “yes”, the algorithm fuses $slice_1$ and $slice_2$ by invoking the function *Fuse-Comp-Slice* defined in Figure 4.1. It then adds the fused slice into the fused slice nest, $fuse-set$, and removes the original slices $slice_1$ and $slice_2$ from their original slice nests. After all pairs of slices in the two slice nests have been considered for fusion, the left-over computation slices are the ones that cannot be legally fused with any other slices. These left-over slices are attached to the fused result $fuse-set$ and are annotated as belonging to $left-over(fuse-set)$.

Note that when the algorithm partially fuses two slice nests, it forces the left-over slices to be nested inside other slices that contain more statements. This forced nesting order may not be beneficial. For example, it might prevent a loop that carries the most spatial reuses from being placed at the innermost level. To resolve this conflict, this thesis refrains from fusing two spatial loops unless a complete fusion is possible. Due to similar reasons, partially fusing two slice nests may also prevent the

fused nest from being blocked. This thesis thus disallows partial fusion if only one of the slice nests can be blocked and if the blocking optimization is more favorable. The implemented translator (see Section 6.1.1) lets the user specify whether to favor fusion or blocking when optimizing applications.

The following describes how to adapt the traditional typed-fusion algorithm for fusing a sequence of computation slice nests. First, it is necessary to introduce the concept of a fusion dependence graph, which is used in the typed-fusion algorithm. Each vertex of the fusion dependence graph represents a loop, and an edge is put from vertex x to y in the graph if there are dependences from statements inside loop x to statements inside loop y . The edge is annotated as a *bad edge* if the dependences prevent the two loops (x and y) from being legally fused. The vertices in the graph are classified into different types, and the typed-fusion algorithm is applied to greedily cluster all the vertices that belong to a particular type and are not connected by fusion-preventing *bad* paths. The typed-fusion algorithm can be applied multiple times to fuse different types of vertices.

To adapt the fusion dependence graph for fusing computation slice nests, this thesis modifies the graph so that each vertex is a computation slice nest instead of a single loop. An edge is put from vertex x to y if there are dependences from statements in the slice nest x to statements in the slice nest y , and the edge is annotated as a *bad edge* if these two slice nests should not be fused. The safety of fusing two slice nests

is determined by applying the function *Comp-Slice-Fusable* in Figure 4.1 to each pair of computation slices from the two nests. This process follows the exact structure of the function *Fuse-Slice-Nest* in Figure 4.2 except that no actual fusion is performed. The edge between two slice nests x and y is annotated as a *bad edge* if either of the following two conditions holds: first, if no slices from the two nests can be fused, and second, if partial fusion is disallowed and there are left-over slices in the two nests.

After constructing an adapted fusion dependence graph for a sequence of computation slice nests, this thesis applies the same traditional typed-fusion algorithm to this adapted graph. The algorithm then aggressively clusters the computation slice nests and fuses each cluster of slice nests by applying the function *Fuse-Slice-Nest* in Figure 4.2. Because the complete fusion of two slice nests has less side effect than a partial fusion, this thesis applies the typed-fusion algorithm in two passes. The first pass allows complete fusion only, and the second pass allows partial fusion when profitable.

Because loop reversal can facilitate the fusion of some loops with incompatible enumeration orders (see Section 4.2.2), this thesis also integrates the reversal of computation slices into the typed-fusion algorithm by classifying computation slices into three groups: the first group can be enumerated only in increasing order (all the slicing loops must have positive enumeration steps), the second group can be enumerated only in decreasing order, and the third group is reversible (each slice can be enumer-

<pre> do 10 j=1,N do 10 i=1,N duz(i,j,1) = duz(i,j,1)*b(1) 10 continue do 20 k=2,N-1 do 20 j=1,N do 20 i=1,N duz(i,j,k)=(duz(i,j,k)-a(k)*duz(i,j,k-1))*b(k) 20 continue do 30 j=1,N do 30 i=1,N tot(i,j) = 0. 30 continue do 40 k=1,N-1 do 40 j=1,N do 40 i=1,N tot(i,j) = tot(i,j) + d(k)*duz(i,j,k) 40 continue </pre>	<pre> do j=1,N do i=1,N duz(i, j, 1) = duz(i, j, 1) * b(1) tot(i,j) = 0. enddo do k=1,N-1 do i=1,N if (k .ge. 2) then duz(i, j, k) = (duz(i, j, k) - a(k) * duz(i, j, k - 1)) * b(k) endif tot(i,j) = tot(i,j) + d(k)*duz(i,j,k) enddo enddo enddo </pre>
(a) original code	(b) after fusion

Figure 4.4 Multi-level fusion example: code fragment from Erlebacher

ated in either increasing or decreasing order). The fusion algorithm is then applied in two passes: the first pass allows fusing only the first and third groups of slices, and the second pass allows fusing only the second and third groups. Each reversible computation slice thus can be fused with other slices independent of the enumeration orders of these slices.

To illustrate the multi-level fusion algorithm for computation slice nests, Figure 4.4 shows both the original and optimized versions of a code segment from the subroutine *tridvpk* in *Erlebacher* (a benchmark for computing partial derivatives). The original code in Figure 4.4(a) contains four loop nests, and the following denotes these loop nests as N_{10} , N_{20} , N_{30} and N_{40} respectively. Each loop nest can be represented using a single computation slice nest. Because all the j loops in (a) can be legally fused into

a single loop, the fusion algorithm can fuse all the corresponding j slices and then use the fused slice to perform a dependence hoisting transformation, which generates the outermost j loop in (b). All the i loops in (a) can also be fused into a single loop; however, because these i loops carry spatial reuses and should stay innermost, no partial fusion is applied to these loops.

To transform Figure 4.4(a) to (b), the fusion algorithm first fuses all the slice nests that are fully fusible. This pass of the typed-fusion algorithm first fuses the loop nest N_{10} with N_{30} and fuses the nest N_{20} with N_{40} . The algorithm then performs another pass that allows partial fusion of slice nests. This pass of the typed-fusion algorithm successfully fuses the two resulted slice nests into a single one that has an outermost slice containing all the j loops in the original code. After using the fused nest to perform a sequence of dependence hoisting transformations, the final transformed code has a single fused loop j at the outermost loop level and two separate loop nests inside the fused j loop, as shown in (b).

4.4 Transformation Framework

This section presents the top level algorithm of the computation slicing framework, which systematically applies dependence hoisting to transform arbitrarily nested loops for locality. As shown in Figure 4.5, the algorithm recursively invokes itself to hierarchically apply two optimizations, multi-level fusion and blocking, at each loop level of


```

Optimize-Code-Segment( $C$ )
(1)  $slice\text{-}set\text{-}vec = \emptyset$ ; Distribute loop nests in  $C$ 
    for (each distributed loop nest  $L$ ) do
        Hoisting-Analysis( $L$ ,  $slice\text{-}set$ )
        Add  $slice\text{-}set$  to  $slice\text{-}set\text{-}vec$ 
(2) Apply typed fusion to  $slice\text{-}set\text{-}vec$ 
(3) For (each  $slice\text{-}set \in slice\text{-}set\text{-}vec$ ) do
    For (each  $slice\text{-}set_1 \in left\text{-}over(slice\text{-}set)$ ) do
        Block-Loop-Nest(Distribute( $C, slice\text{-}set_1$ ),  $slice\text{-}set_1$ )
    Block-Loop-Nest(Distribute( $C, slice\text{-}set$ ),  $slice\text{-}set$ )
    Apply index-set splitting to cleanup conditionals
    For (each segment  $C_1$  inside all slices in  $slice\text{-}set$ )
        Optimize-Code-Segment( $C_1$ )

```

Figure 4.5 Transformation steps of computation slicing

a given code segment C . This algorithm builds upon the dependence hoisting analysis and transformation algorithms in Chapter 3 and has three steps. Section 4.4.1 first elaborates each step of the algorithm. Section 4.4.2 then discusses the correctness and complexity of the framework.

4.4.1 Transformation Steps

Given a code segment C to optimize, the algorithm in Figure 4.5 first finds all the computation slices that can be shifted to the outermost position of C . These slices are constructed by invoking the function *Hoisting-Analysis* in Figure 3.3 (note that if the input loop nest C is perfectly nested, traditional loop interchange analysis would suffice, see Section 6.1.1). To reduce the overhead of transitive dependence analysis, the framework distributes the input code segment C into strongly connected components in the dependence graph and then applies dependence hoisting analysis to each of the distributed loop nest. These distributed loop nests will later be re-fused when typed-fusion is applied to the collected computation slice nests.

After the first step collects all the computation slices into a vector of slice nests,

the framework constructs a fusion dependence graph where each vertex of the graph is a computation slice nest. The framework then applies the traditional typed-fusion algorithm [37, 33] to fuse these slice nests. After fusion, each entry in *slice-set-vec* contains a fused slice nest *slice-set*, where *left-over(slice-set)* contains the left-over slices from partial fusion of the original slice nests. For more detail, see Section 4.3.2.

The final step of the framework then examines each fused slice nest in *slice-set-vec* and transforms the original code segment *C* accordingly. For each fused slice nest *slice-set*, the algorithm uses all the left-over slice nests from partial fusion of *slice-set* to transform the original code *C* before using the fused slice nest *slice-set*. This strategy guarantees that the fused loops of the left-over slices are nested inside those of the slices in *slice-set*, as discussed in Section 4.3.2. When using each slice nest *slice-set* to optimize *C*, the algorithm first invokes the function *Distribute(C, slice-set)*, which distributes *C* and returns a cluster of loop nests that contain all the statements in *slice-set*. The returned cluster is then used as input to the function *Block-Loop-Nest* in Figure 4.2. This function in turn rearranges the nesting order of the slices in *slice-set* and blocks the input loop nest if profitable.

After applying a sequence of dependence hoisting transformations to optimize the original code *C*, the algorithm performs a cleanup step, which applies loop index-set splitting to eliminate the conditionals that still remain in the transformed code, as illustrated in the blocked non-pivoting LU code in Figure 4.3(c). The framework then

recursively invokes itself to further optimize the code segment inside each optimized loop nest. By applying the same optimizations at each loop level of the original code, the algorithm hierarchically optimizes the original code at multiple loop levels, guaranteeing generality of the transformation framework.

4.4.2 Correctness and Complexity

The correctness of the computation slicing framework follows directly that of the dependence hoisting analysis and transformation algorithms, as discussed in Section 3.3. Since the blocking and fusion algorithms in Figure 4.2 follow the traditional blocking and fusion theory, they also guarantee that no dependence edge is reversed. Thus no further correctness proof is necessary.

The transformation algorithms of the framework are not optimal in that the optimized code is not guaranteed to have the best performance. The framework can be further extended to incorporate more sophisticated profitability analyses such as those described by Kennedy and McKinley [32]. It can also integrate traditional loop transformation techniques, such as reversal, index-set splitting and skewing [38, 15], in a more sophisticated fashion to achieve better performance.

The complexity of the computation slicing framework is that of framework's top-level function *Optimize-Code-Segment* in Figure 4.5. This function invokes itself at most L times, where L is the maximum depth of loop nests in the original code. At

each loop level, the worst case complexity of applying *Hoisting-Analysis* is $O(N^2L)$ (see Section 3.3.3), where N is the number of statements in a loop nest. The worst case complexity of the first step of *Optimize-Code-Segment* is thus $O(N^2L^2)$. The complexity of the second step is $O(DL^2)$, where D is the size of the dependence graph. Since the worst case complexity of the dependence hoisting transformation is $O(N^2 + L^2D)$ (see Section 3.3.3), the worst case complexity of the third step of *Optimize-Code-Segment* is $O(N^2L + L^3D)$. The worst case complexity of the whole framework is thus $O(N^2L^2 + L^3D)$. Since the dependence analysis itself requires a lower bound complexity $O(N^2 * L^2)$ (for traditional dependence vector representations, the complexity of dependence analysis is $O(N^2)$), the computation slicing framework is quite efficient. In practice, the compile-time complexity of the framework increases roughly proportionally with the code size of the subroutine being optimized, see Section 6.5.

Chapter 5

Recursion Transformation

This chapter presents techniques to automatically transform arbitrarily nested loops into a recursive form that has a hierarchy of smaller working sets and thus exploits locality simultaneously at multiple levels of the memory hierarchy. Given a sequence of consecutive loop nests, the recursion transformation first relates statement instances using a technique called *iteration space slicing* [49]. It then creates a recursive procedure to recursively execute these statement instances. Finally, it transforms the original code to include initial calls to the recursive procedures. Each recursive procedure has a data reuse property similar to that of a code simultaneously blocked at many different levels.

The iteration space slicing technique used in the recursion transformation was first proposed by Pugh and Rosser [49]. This technique uses integer set programming tools such as the Omega library [28] to compute symbolic iteration sets of statements that depend on each other. Given an iteration instance I_0 of statement s_0 , iteration space slicing uses the transitive dependence information to compute the iteration set of another statement s_1 , a set that must be executed before (backward iteration space slicing) or after (forward iteration space slicing) the given iteration I_0 of s_0 . This technique is similar to *program slicing* [46], which computes slices of entire statements

```

do k = 1, N - 1
  do i = k + 1, N
s1:    A(i, k) = A(i, k) / A(k, k)
    do j = k + 1, N
      do i = k + 1, N
s2:    A(i, j) = A(i, j) - A(i, k) * A(k, j)
Initial decisions:
skey = s2
Rloops = {j(s2), i(s2)}
Rf = (lbj, ubj, lbi, ubi)
Rorder = (lbj, mj, lbi, mi)
        → (lbj, mj, mi + 1, ubi)
        → (mj + 1, ubj, lbi, mi)
        → (mj + 1, ubj, mi + 1, ubi)
Fpars = {(ubj + 1, N, 2, N), (lbj, ubj, ubi + 1, N)}
Ppars = {(2, lbj - 1, 2, N), (lbj, ubj, 2, lbi - 1)}
Compute-Iter-Sets:
Before(s1, s2) = {(k, j, i) → (k', i') | k' ≤ k, i' ≤ i}
Before(s2, s2) = {(k, j, i) → (k', j', i') |
                  k' ≤ k, j' ≤ j, i' ≤ i}

Current(s2) = { (k, j, i) | 1 ≤ k ≤ min(ubj, ubi) - 1,
                    max(k + 1, lbj) ≤ j ≤ ubj,
                    max(k + 1, lbi) ≤ i ≤ ubi }
Previous(s2) = { (k, j, i) | 1 ≤ k ≤ lbj - 2,
                    k + 1 ≤ j ≤ lbj - 1, k + 1 ≤ i ≤ N; or
                    1 ≤ k ≤ min(ubj - 1, lbi - 2), max(k + 1, lbj)
                    ≤ j ≤ ubj, k + 1 ≤ i ≤ lbi - 1 }

Backward-Slicing:
Previous(s2) = Before(s2, s2)[Previous(s2)]
Before(s2, s2)[Current(s2)]
= { (k, j, i) | 1 ≤ k ≤ min(ubj, ubi) - 1,
                k + 1 ≤ j ≤ ubj, k + 1 ≤ i ≤ ubi }
Before(s1, s2)[Current(s2)]
= {(k, i) | 1 ≤ k ≤ min(ubj, ubi) - 1, k + 1 ≤ i ≤ ubi}
Previous(s1) = Before(s1, s2)[Previous(s2)]
= {(k, i) | 1 ≤ k ≤ lbj - 2, k + 1 ≤ i ≤ N; or
            1 ≤ k ≤ min(ubj - 1, lbi - 2), k + 1 ≤ i ≤ lbi - 1 }
Current(s1) = Before(s1, s2)[Current(s2)] - Previous(s1)
= {(k, i) | max(lbj, lbi) - 1 ≤ k ≤ min(ubj, ubi) - 1,
            max(k + 1, lbi) ≤ i ≤ ubi }

```

Figure 5.1 Recursive formulation of non-pivoting LU

connected by dependence paths. Rather than the entire statements, iteration space slicing operates on iteration sets (or instances) of statements instead.

To elaborate the recursion transformation technique, Section 5.1 first uses the *KJI* form of non-pivoting LU as example to illustrate the overall strategy of the transformation, Section 5.2 introduces notations and definitions. Section 5.3 presents the transformation algorithm in a general context. Finally, Section 5.4 presents heuristics to configure the algorithm for better locality.

5.1 Example: Non-pivoting LU

To illustrate the overall strategy of the recursion transformation, this section again uses the *KJI* form of non-pivoting LU as example. Figure 5.1 shows the recursive formulation of this code. Figure 5.2 shows the transformed recursive code.

```

call LU-recur(1, N, 1, N)

subroutine LU-recur( $lb_j, ub_j, lb_i, ub_i$ )
  if (stop recursive call) then
    do  $k = 1, \min(N - 1, ub_i - 1, ub_j - 1)$ 
      if ( $k \geq \max(lb_j - 1, lb_i - 1)$ ) then
        do  $i = \max(k + 1, lb_i), \min(N, ub_i)$ 
           $s_1: A(i, k) = A(i, k) / A(k, k)$ 
        do  $j = \max(k + 1, lb_j), \min(N, ub_j)$ 
          do  $i = \max(k + 1, lb_i), \min(N, ub_i)$ 
             $s_2: A(i, j) = A(i, j) - A(i, k) \cdot A(k, j)$ 
      else
         $m_j = (lb_j + ub_j) / 2$ 
         $m_i = (lb_i + ub_i) / 2$ 
        call LU-recur( $lb_j, m_j, lb_i, m_i$ )
        call LU-recur( $lb_j, m_j, m_i + 1, ub_i$ )
        call LU-recur( $m_j + 1, ub_j, lb_i, m_i$ )
        call LU-recur( $m_j + 1, ub_j, m_i + 1, ub_i$ )

```

Figure 5.2 Recursive code of non-pivoting LU

To generate the recursive code in Figure 5.2, the transformation first picks statement s_2 in Figure 5.1 as the starting point and selects the j and i loops surrounding s_2 to enumerate recursively. These loops are called the *recursive loops*. (How to make the above decisions is described in Section 5.4.) The transformation then creates a formal-parameter vector, (lb_j, ub_j, lb_i, ub_i) , for the recursive procedure. Here the four parameters represent the lower and upper bounds of the recursive loops $j(s_2)$ and $i(s_2)$ respectively. At each recursive call, the procedure executes the iterations $lb_j \leq j \leq ub_j$ of the $j(s_2)$ loop and the iterations $lb_i \leq i \leq ub_i$ of the $i(s_2)$ loop. Thus the formal-parameter vector $\vec{R}_f$ models the iterations of the recursive loops to be enumerated within each recursive call. This parameterized invocation of the recursive procedure is denoted as the “current” recursive call.

The transformation then decides the order of making recursive calls at each recursive level. Because both the $j(s_2)$ and $i(s_2)$ loops carry dependences, these loops must be enumerated in increasing (forward) order. The recursive enumeration order for these loops is expressed as R_{order} in Figure 5.1. This order specifies that the two

loops should each be divided by half, the $i(s_2)$ loop should be divided before the $j(s_2)$ loop, and that the two halves of each loop should be executed in forward order. Here the variables m_j and m_i denote the split points of the $j(s_2)$ and $i(s_2)$ loops. Based on this order of making recursive calls, Figure 5.1 shows two actual-parameter vector sets, P_{pars} and F_{pars} , for the recursive procedure. The loop iterations modeled by the parameter vectors in P_{pars} are enumerated before the “current” recursive call, and the iterations modeled by the parameter vectors in F_{pars} are enumerated after the “current” recursive call.

The transformation now performs a key analysis step to determine which iterations of the statements must be executed within each recursive call — the *current* recursive call. In Figure 5.1, these iteration sets are $Current(s_1)$ and $Current(s_2)$ for statements s_1 and s_2 respectively. Figure 5.1 also shows two additional symbolic iteration sets, $Previous(s_1)$ and $Previous(s_2)$, which contain the iterations that must be executed before the *current* recursive call.

To compute these iteration sets for s_1 and s_2 , the transformation uses the two functions, $Before(s_1, s_2)$ and $Before(s_2, s_2)$ in Figure 5.1, which represent the transitive dependences from s_1 to s_2 and from s_2 to itself respectively. These transitive dependences are expressed as integer set mappings instead of the EDM representation described in Chapter 2. These two transitive dependence mappings are then used to relate the symbolic iteration sets of statements that are connected by dependence

paths.

First, to obtain $Current(s_2)$, the transformation directly parameterizes the original iteration set of s_2 with $\vec{R}_f = (lb_j, ub_j, lb_i, ub_i)$, the formal-parameter vector of the recursive procedure. To obtain $Previous(s_2)$, the transformation first computes two symbolic iteration sets from $Current(s_2)$ by replacing the formal-parameter vector (lb_j, ub_j, lb_i, ub_i) in $Current(s_2)$ with the actual-parameter vectors $(2, lb_j - 1, 2, N)$ and $(lb_j, ub_j, 2, lb_i - 1)$ respectively. It then computes $Previous(s_2)$ as the union of the two replaced iteration sets.

The transformation then computes the iteration set $Current(s_1)$ for s_1 , a set that must be executed together with s_2 within each recursive call. As illustrated in Figure 5.1, applying the transitive dependence mapping $Before(s_1, s_2)$ to the iteration set $Current(s_2)$ ($Before(s_1, s_2)[Current(s_2)]$) returns the iteration set of s_1 that must be executed within or before the current recursive call. Similarly, applying the mapping $Before(s_1, s_2)$ to $Previous(s_2)$ returns the iteration set of s_1 that must be executed before the current recursive call. Subtracting these two sets thus gives $Current(s_1)$, the iteration set of s_1 that must be executed within the current recursive call.

Finally, the transformation transforms the original code to recursively execute the statements s_1 and s_2 . It first creates a recursive procedure that includes two segments: one executes the recursion base of the procedure and the other makes

deeper recursive calls. Here the recursion base executes the iteration sets $Current(s_1)$ and $Current(s_2)$ in their original order, and the deeper recursive calls are consistent with R_{order} , the order of recursive calls used to compute $Current(s_1)$ and $Current(s_2)$. The transformation then assembles these two segments together by synthesizing a conditional that determines whether to execute the base code or to make the next level recursive calls. Finally, the transformation replaces the original loop structure with an initial call to the generated recursive procedure, and the final transformed code is shown in Figure 5.2.

5.2 Notation and Definitions

As illustrated in Section 5.1, the recursion transformation associates each recursive procedure with two attributes: a set of recursive loops (R_{loops}) and a vector of recursive parameters ($\vec{R}_f$). Here R_{loops} denotes the loops to be enumerated recursively in the recursive procedure, and $\vec{R}_f$ denotes the vector of formal parameters for the recursive procedure. If m recursive loops are chosen, the recursive procedure is parameterized with a vector of $2m$ formal parameters, $\vec{R}_f = (lb_1, ub_1, \dots, lb_m, ub_m)$, where lb_i and ub_i ($i = 1, \dots, m$) represent the symbolic lower and upper bounds for the i th recursive loop. The formal-parameter vector $\vec{R}_f$ thus can be thought as specifying a symbolic iteration set of the recursive loops, a set that is enumerated within an arbitrary recursive call denoted as the *current* recursive call.

Each invocation to the recursive procedure substitutes a vector of actual values for $\vec{R}_f$. This vector of actual values is called an *actual-parameter vector* of the recursive procedure and is denoted as $\vec{r}_a$. Similar to $\vec{R}_f$, each actual-parameter vector $\vec{r}_a$ also specifies a symbolic iteration set for the recursive loops, a set that is enumerated by invoking the recursive procedure using the actual-parameter vector.

For each statement s to be included in a recursive procedure, the recursion transformation computes the following three symbolic iteration sets for s .

- *Current(s)*: the iteration set of s to be executed in an arbitrary recursive call, the *current* recursive call;
- *Previous(s)*: the iteration set of s to be executed before the current recursive call;
- *Future(s)*: the iteration set of s to be executed after the current recursive call.

All the above iteration sets are represented as symbolic integer sets [28] and are parameterized by $\vec{R}_f$, the formal-parameter vector of the recursive procedure. For each actual-parameter vector $\vec{r}_a$, the expression $Current(s)[\vec{r}_a]$ replaces the formal parameters in $Current(s)$ with the corresponding actual values in $\vec{r}_a$ and then returns an iteration set that takes the union of all the replaced iteration sets.

For each pair of statements s_1 and s_2 , the recursion transformation denotes the transitive dependences between s_1 and s_2 using the following two functions.

- $Before(s_1, s_2)[I]$: returns the iteration set of s_1 that must be executed before an arbitrary iteration set I of s_2 .
- $After(s_1, s_2)[I]$: returns the iteration set of s_1 that must be executed after an arbitrary iteration set I of s_2 .

Here $After[]$ is the inverse function of $Before[]$. Both functions are represented as integer set mappings and are translated from the EDM representation of transitive dependences in Section 2.2.

5.3 Transformation Algorithms

This section presents the recursion transformation algorithm, which is encoded by the function *Recursion-Transformation* in Figure 5.3. This function first chooses a set of statements called “key” statements as starting points. For each chosen statement *skey*, it then tries to recursively execute the iteration set of *skey* through the following steps.

- (1) Choose a set of the loops (R_{loops}) surrounding *skey* to enumerate recursively and create a vector of formal parameters ($\vec{R}_f$) for these chosen recursive loops. Decide the order of making recursive calls (R_{order}) at each recursive level.
- (2) Find all the statements that should be included in the same recursive procedure as *skey*. For each such statement s , compute $Current(s)$, the symbolic iteration set of s to be executed together with *skey* within each recursive call.

(3) Examine each computed symbolic iteration set $Current(s)$ to determine whether it is profitable to recursively execute the iteration set. If the answer is *yes* for all iteration sets, create a procedure to recursively execute these iteration sets. Transform the original code so that all statement instances included in the recursive procedure are replaced by an initial call to the procedure.

The above three steps are repeated for each key statement. Note that the iteration instances included in a recursive procedure are subtracted from their original iteration sets before the next key statement is processed. Thus each statement instance is included in at most one recursive procedure; that is, the transformation algorithm never duplicates any statement instance.

This section presents the recursion transformation in a general context, assuming that the key statements, recursive loops, and recursive-call orders are arbitrarily chosen. Section 5.4 will present heuristics to make these decisions for better locality. Sections 5.3.1 first presents algorithms to compute the symbolic iteration sets of statements. Section 5.3.2 then describes how to generate code for the recursive procedure and how to transform the original code based on the computed symbolic iteration sets.

```

Recursion-Transformation( $C$ )
   $KeyStmts = \text{Choose-Key-Statements}(C)$ 
  while  $KeyStmts \neq \emptyset$  do
    remove a statement  $skey$  from  $KeyStmts$ 
    if ( $skey$  has been already processed) continue
    // recursively computing key statement
     $R_{loops} = \text{Choose-Recur-Loops}(C, skey)$ 
     $\vec{R}_f = \text{Create-Recur-Params}(R_{loops})$ 
     $R_{order} = \text{Decide-Recur-Order}(C, R_{loops})$ 
     $R_{stmts} = \text{Compute-Iter-Sets}(C, skey, \vec{R}_f, R_{order})$ 
    // code transformation
    if ( $R_{stmts} \neq \emptyset$ ) then
      Create-Recur-Proc ( $R_{stmts}, \vec{R}_f, R_{order}$ )
      Transform-Orig-Code ( $C, R_{stmts}, R_{loops}$ )

Compute-Iter-Sets ( $skey, \vec{R}_f, R_{order}$ )
   $\vec{R}_f$ : recursive params;  $R_{order}$ : recur-call order;
   $P_{para} = \text{Previous-Recur-Calls}(R_{order}, \vec{R}_f)$ 
   $F_{para} = \text{Future-Recur-Calls}(R_{order}, \vec{R}_f)$ 
   $Current(skey) = \text{Restrict-Bounds}(skey, \vec{R}_f)$ 
   $Previous(skey) = \bigcup_{\vec{r}_a \in P_{para}} Current(skey)[\vec{r}_a]$ 
   $Future(skey) = \bigcup_{\vec{r}_a \in F_{para}} Current(skey)[\vec{r}_a]$ 
   $R_{stmts} = \emptyset$ 
  Backward-Slicing ( $D, skey, R_{stmts}, P_{para}, F_{para}$ )
  return  $R_{stmts}$ 

Backward-Slicing( $skey, R_{stmts}, P_{para}, F_{para}$ )
   $R_{stmts}$ : statements already processed;
   $P_{para}, F_{para}$ : params for previous/future recur-calls
   $AddStmts = \text{Reachable-into}(D, skey) - R_{stmts}$ 
  if ( $AddStmts == \emptyset$ ) then return
  for each statement  $s \in AddStmts$  do
     $Previous(s) = \text{Before}(s, skey)[Previous(skey)]$ 
     $Current(s) = \text{Before}(s, skey)[Current(skey)]$ 
     $Current(s) = Current(s) - Previous(s)$ 
     $Future(s) = \bigcup_{\vec{r}_a \in F_{para}} Current(s)[\vec{r}_a]$ 
    if (! Profitable( $Current(s)$ )) then return  $\emptyset$ 
   $R_{stmts} = R_{stmts} \cup AddStmts$ 
  Forward-Slicing ( $D, AddStmts, R_{stmts}, P_{para}, F_{para}$ )

Forward-Slicing ( $D, AddStmts, R_{stmts}, P_{para}, F_{para}$ )
   $AddStmts$ : stmts to start forward slicing;
   $ExtraStmts = \text{Reachable-From}(D, AddStmts) - R_{stmts}$ 
  for each statement  $s \in ExtraStmts$  do
    if ( $s \in R_{stmts}$ ) then continue
     $Future(s) = Current(s) = \emptyset$ 
    for each statement  $ss \in StartStmts$  do
       $Future(s) \cup = \text{After}(s, ss)[Future(ss)]$ 
       $Current(s) \cup = \text{After}(s, ss)[Current(ss)]$ 
     $Current(s) = Current(s) - Future(s)$ 
     $Previous(s) = \bigcup_{\vec{r}_a \in P_{para}} Current(s)[\vec{r}_a]$ 
    Backward-Slicing ( $D, s, R_{stmts}, P_{para}, F_{para}$ )

```

Figure 5.3 Recursion transformation algorithm

5.3.1 Computing Iteration Sets

In Figure 5.3, the function *Compute-Iter-Sets* computes three iteration sets, $Current(s)$, $Previous(s)$ and $Future(s)$, for each statement s that should be executed in a recursive procedure. The function then returns a variable set R_{stmts} , which contains all the statements in the procedure. The symbolic iteration sets of statements are computed using the *Before* and *After* functions obtained from transitive dependence analysis.

The algorithm first uses $\vec{R}_f$, the formal-parameter vector of the recursive procedure, to model the iterations of the recursive loops that are enumerated in an arbitrary recursive call, the *current* recursive call. It then uses R_{order} , the recursive-call order, to compute two sets of actual-parameter vectors, P_{para} and F_{para} . These two sets spec-

ify the iterations of the recursive loops that are enumerated before (P_{pars}) and after (F_{pars}) the *current* recursive call respectively. Both P_{pars} and F_{pars} include multiple actual-parameter vectors that are enumerated over several recursive calls.

The algorithm then initializes three iteration sets, $Current(skey)$, $Previous(skey)$ and $Future(skey)$, for the key statement $skey$. To initialize $Current(skey)$, the algorithm invokes function $Restrict-Bounds(skey, \tilde{R}_f)$, which first constructs the original iteration set of $skey$ and then modifies the set by restricting the range of each recursive loop $l_k(skey)$ within its corresponding lower and upper bound parameters in $\tilde{R}_f$. To initialize $Previous(skey)$, the algorithm substitutes each actual-parameter vector $\vec{r}_a \in P_{pars}$ for the $\tilde{R}_f$ in $Current(skey)$ and then takes the union of the substituted sets. The iteration set $Future(skey)$ is initialized similarly as $Previous(skey)$ except that the actual-parameter vectors in F_{pars} are used instead.

The algorithm then computes three iteration sets, $Current(s)$, $Previous(s)$, and $Future(s)$, for each statement s that should be executed together with $skey$ in the recursive procedure. These iteration sets are computed by invoking function $Backward-Slicing$, which as a side effect also places each statement to be executed together with $skey$ into the input variable set R_{stmts} . Note that R_{stmts} is initialized to be empty instead of containing $skey$ before invoking $Backward-Slicing$. This initialization causes the three iteration sets of $skey$ to be recomputed and verified for profitability, guaranteeing the correctness and profitability for the key statement.

In Figure 5.3, the function *Backward-Slicing* is initially invoked within *Compute-Iter-Sets*. The functions *Backward-Slicing* and *Forward-Slicing* then repeatedly invoke each other until no more statements should be included in R_{stmts} . The following subsections describe these two functions.

Backward Slicing Iteration Sets

The function *Backward-Slicing* in Figure 5.3 is applied to guarantee the correctness of the recursion transformation. This function first finds all the statements that must be executed together with the key statement *skey* inside the recursive procedure. For each such statement *s*, it then uses the transitive dependence mapping from *skey* to *s* to compute the three iteration sets, *Current(s)*, *Previous(s)* and *Future(s)*. Given an arbitrary iteration instance $I_0(skey) \in Current(skey)$, each iteration instance $I(s)$ that must be executed before $I_0(skey)$ is included in either *Current(s)* or *Previous(s)*; that is, $I(s)$ is executed either in the same recursive call as $I_0(skey)$ or in previous recursive calls. Backward slicing thus guarantees that no dependence constraint across different recursive calls is violated. Since the code generation algorithm in Section 5.3.2 guarantees that all the dependence constraints within each recursive call are also preserved, the recursion transformation does not violate any dependence constraint.

As shown in Figure 5.3, the algorithm first invokes function *Reachable-Into(D, skey)*

to find the set of all the statements that have some dependence paths into *skey*. Subtracting this set with R_{stmts} gives *AddStmts*, the additional statements that need to be included in the recursive procedure.

For each statement $s \in AddStmts$, the algorithm then computes the three iteration sets, *Previous(s)*, *Current(s)*, and *Future(s)*. It computes *Previous(s)* as $Before(s, skey)[Previous(skey)]$, which applies the transitive dependence mapping $Before(s, skey)$ to the iteration set *Previous(skey)* and then returns the iteration set of s that must be executed before the iterations of *skey* in previous recursive calls. Similarly, $Before(s, skey)[Current(skey)]$ gives the iteration set of s that must be executed in the current or previous recursive calls. Subtracting this iteration set with *Previous(s)* gives *Current(s)*, the iteration set of s that must be executed within the current recursive call. The algorithm then computes *Future(s)* by substituting each *actual-parameter vector* $\tilde{r}_a$ in F_{pars} for the formal-parameter vector in *Current(s)* and then taking the union of the replaced iteration sets $(\cup_{\tilde{r}_a \in F_{pars}} Current(s)[\tilde{r}_a])$. The iteration set *Future(s)* is used later in function *Forward-Slicing*.

The algorithm then examines each computed iteration set *Current(s)* for profitability. If the function $Profitable(Current(s))$ returns *false* for some statement s , none of the other statements in *AddStmts* is added into R_{stmts} , and the entire function aborts; otherwise (all the verifications succeed), the algorithm unions R_{stmts} with *AddStmts* and then invokes the forward slicing algorithm, which in turn places other

statements that use the values computed by *AddStmts* into the recursive procedure.

The profitability test of the *Current* iteration sets is applied to avoid unnecessary recursion transformations that degrade performance. Although iteration space slicing never fails to compute a legal iteration set *Current(s)* for any statement *s*, the computed iteration set may not improve performance. For example, if *Current(s)* is not sufficiently reduced, the recursive procedure may not have a hierarchically reduced working set and thus may not exhibit better locality. In the worst case, the entire iteration sets of all the statements can simply be executed in a single recursive call, achieving no optimization for the original code.

Forward Slicing Iteration Sets

In Figure 5.3, the function *Forward-Slicing* applies forward iteration space slicing to keep together statements that use values computed by statements already in the recursive procedure. Forward slicing is thus not required for correctness and is an optional pass. However, if applied, forward slicing must again invoke backward slicing for each additional statement put into *R_stmts* to guarantee correctness. The additional backward slicing then ensures that all the dependence edges into these additional statements will not be reversed. If backward slicing fails for some additional statement *s*, the algorithm simply excludes *s* from *R_stmts*. The invocations of forward and backward slicing repeat until the process terminates with a pass of backward

slicing.

Similar to *Backward-Slicing*, *Forward-Slicing* first invokes function *Reachable-From*($D, AddStmts$) to identify all the statements having incoming dependence paths from statements in *AddStmts*. It then adds these statements into a set *ExtraStmts* and computes the iteration sets *Future*(s) and *Current*(s) for each statement $s \in ExtraStmts$. Here the *After* transitive dependence mappings are used instead of the *Before* functions used in *Backward-Slicing*. Because *Forward-Slicing* starts from not just one statement *skey* but from a set of statements *AddStmts*, the algorithm takes the union of all the transitive dependence paths from *AddStmts* to s to guarantee that the correct iteration sets for s are computed.

5.3.2 Code Generation

After successfully computing the iteration set *Current*(s) for each statement $s \in R_{stmts}$, the recursion transformation creates a procedure to recursively execute all the statements in *R_{stmts}*. The recursive procedure includes two segments: the first segment executes the recursive base case, and the second segment makes deeper recursive calls. A conditional is then synthesized to determine whether to execute the base case or make deeper recursive calls. After creating the procedure to recursively executing the statements in *R_{stmts}*, the recursion transformation modifies the original code so that the recursively enumerated instances of these statements are replaced

with an initial call to the recursive procedure.

To generate code for the base case of recursion, the transformation uses subroutines provided by the Omega Library [28] to synthesize loop nests, which enumerate the *Current* iteration sets of statements in R_{stmts} in their original lexicographic order [29]. This code generation technique is similar to the one described by Adve and Mellor-crummey [1].

To generate code for deeper recursive calls, the transformation divides the iteration range of each recursive loop by half. Given n recursive loops with bound parameters $lb_1, ub_1, \dots, lb_n, ub_n$ respectively, the algorithm finds the middle split points $m_i = (lb_i + ub_i)/2$ for each recursive loop ℓ_i and then makes 2^m deeper recursive calls. The order of recursive calls (R_{order}) is pre-determined by function *Decide-Recur-Order* in Figure 5.3 before computing the *Current* iteration sets for statements.

After creating the recursive procedure, the recursion transformation modifies the original code by inserting an initial call to the created procedure. Because a forward-slicing step may not have been performed, the initial call is placed as the first statement in the original code to guarantee correctness of the left-over statement instances that *use* the values computed by the recursive procedure. The actual-parameter vector for the initial invocation contains the original loop bounds of all the recursive loops (say, $\vec{r}_{original}$).

The recursion transformation then removes from the original code all the state-

ments that are already included in the recursive procedure. For each statement $s \in R_{stmts}$, the *leftover iterations* of s is computed by subtracting $Current(s)[\vec{r}_{original}]$ from the original iteration set of s . These left-over iterations are then put back into the transformed original code. The code to execute these leftover iterations are generated similarly as the code for the base case of the recursion.

5.4 Configuring Transformation For Locality

This section describes heuristics to specialize the recursion transformation so that better cache performance is achieved for sequential programs. Specifically, this section discusses how to determine key statements, recursive loops, recursive-call orders and how to verify profitability for the recursion transformation.

5.4.1 Choosing Key Statements

This section describes heuristics to choose key statements in the recursion transformation for better locality. This heuristics first identify all the temporal data reuses in the original code. If the data size swept between the source and sink of a reuse is larger than the cache size, it is assumed that the reuse cannot be exploited in the original code. The source or sink statement of the reuse is then identified as a candidate key statement. Because each key statement is included in at most one recursive procedure, the heuristics can identify redundant key statements without sacrificing

```

LargeLoops =  $\emptyset$ 
UnknownLoops = loops with large iteration range
for each array reference r do
  for each  $l \in \text{UnknownLoops}$  surrounding r do
    if l does not carry reuse of r then
      LargeLoops = LargeLoops  $\cup$  {l}
      UnknownLoops = UnknownLoops - {l}
if (LargeLoops ==  $\emptyset$ ) then return  $\emptyset$ 

KeyStmts =  $\emptyset$ 
for each statement s do
  LargeLevel(s) = deepest level of LargeLoops surrounding s
  ReuseLevel(s) = outermost loop level of reuse carried by s
  if (ReuseLevel(s) < LargeLevel(s)) then
    KeyStmts = KeyStmts  $\cup$  {s}
return KeyStmts

```

Figure 5.4 Choosing key statements

either correctness or efficiency of the recursion transformation.

Figure 5.4 shows the algorithm to collect key statements. This algorithm first identifies data reuses simply as data dependences (including input dependences because reuse on read references is important). It then finds all the loops that access a large volume of data. If a loop ℓ_i encloses at least one array reference but does not carry any reuse of the reference, the array reference must access a different data element at each iteration of ℓ_i . Unless loop ℓ_i has a known small range of iterations, ℓ_i is assumed to access a large volume of data. The algorithm then identifies candidate key statements as each statement *s* that carries data reuses and is surrounded by loops that access a large volume of data between the endpoints of some reuses.

After all the candidate key statements are identified, the recursion transformation uses each key statement to drive a recursion transformation as described in Section 5.3. The recursion transformation selects the key statements in an order that favors statements surrounded by more loops because more recursive loops may be selected for such statements. In Figure 5.3, the algorithm extracts key statements from *KeyStmts* in decreasing order of the number of loops surrounding them.

5.4.2 Choosing Recursive Loops

This section discusses heuristics to identify recursive loops for each key statement *skey* selected as the starting point for the recursion transformation. Each recursive loop ℓ accesses a large volume of data, and as its iteration range being reduced, the working set of the recursive procedure should be reduced accordingly.

The algorithm for selecting key statements in Figure 5.4 can be used to select recursive loops as well. For each key statement *skey* in Figure 5.4, this thesis selects recursive loops for *skey* as each loop ℓ that accesses a large volume of data ($\ell \in \text{LargeLoops}(\text{skey})$) and are nested deeper than the outermost loop carrying reuses of *skey* ($\text{loop-level}(\ell) > \text{ReuseLevel}(\text{skey})$). These recursive loops are chosen because as their iteration ranges being reduced, the size of data accessed between the endpoints of reuses in *skey* would be reduced accordingly.

It is sometimes beneficial to also select recursive loops that access a large volume of data ($\ell \in \text{LargeLoops}(\text{skey})$) but are nested outside the outermost loop carrying reuses of *skey* ($\text{loop-level}(\ell) \leq \text{ReuseLevel}(\text{skey})$). These recursive loops may enhance the inter-block reuses between adjacent recursive calls. For example, in matrix multiplication, it is profitable to choose all the three loops as recursive loops to achieve a hierarchically reduced working set. This thesis uses compiler options to specify whether or not to include the additional recursive loops.

5.4.3 Recursive Order and Termination

This section describes how to determine the order of deeper recursive calls at each recursive level, a choice which have significant impact on the inter-block data reuses between adjacent recursive calls. In the current implementation, this thesis preserves the original loop ordering across recursive calls by dividing the inner recursive loops (in the original loop nesting order) first. This strategy does not always produce the best performance, and more sophisticated strategies belongs to the future research of this thesis.

To stop recursion, this thesis derives a symbolic expression that approximates the volume of data touched by the outermost recursive loop in each recursive call. The expression is a function of the recursive formal parameters and is reduced at each deeper recursive level. The deeper recursive calls stop if the estimated data volume is smaller than some minimum volume threshold, specified at runtime.

5.4.4 Verifying Profitability

This section describes how to determine the profitability of a recursion transformation by examining the iteration set $Current(s)$ for each statement s . For the purpose of improving locality, the recursion transformation would not be profitable if statement s still accesses a large volume of data even after the iteration ranges of the recursive loops are sufficiently reduced. In this case, the working set of the recursive

procedure cannot be sufficiently reduced even within a base-level recursive call, so locality benefit cannot be achieved.

To determine whether statement s continues to access a large volume of data even within the base-level recursive call, this thesis examines the iteration set $Current(s)$. If $Current(s)$ has a large iteration range for some loop ℓ surrounding s and if loop ℓ accesses a large volume of data inside the outermost recursive loop, this thesis concludes that statement s still accesses a large volume of data and that the recursion transformation should fail.

Chapter 6

Experimental Results

This chapter provides experimental evidence to support a major claim of this thesis: the dependence hoisting and computation slicing techniques are powerful yet have a much lower cost than the existing transformation frameworks that formulate solution spaces using symbolic integer sets or mappings [34, 42, 2, 36, 49]. In addition, this chapter also presents performance measurements that demonstrate the effectiveness of the recursion transformation framework introduced in this thesis. Both the computation slicing and recursion transformation frameworks have been implemented as Fortran source-to-source translators on top of the D System, an experimental compiler infrastructure developed at Rice university. This chapter denotes these translators as the *slicing translator* and the *recursion translator* respectively. Both translators are applied to optimize a collection of application benchmarks and linear algebra kernels for better locality, and the performance measurements for these benchmarks are then used to evaluate the efficiency of the translators.

The following summarizes the experimental results to support each of the claims discussed above. First, to establish that the slicing translator incurs much lower cost than the existing general frameworks that formulate loop transformation spaces using symbolic integer sets and mappings [34, 42, 2, 36], this work needs to compare the

compile-time overhead of the slicing translator with that of a representative general transformation framework. However, because we have no implementation of these general transformation frameworks, this chapter compares the overhead of the slicing translator with that of the recursion translator, which has been extended to generate regularly optimized loop structures instead of recursive procedures. Because the recursion translator builds on iteration space slicing, which computes the symbolic iteration sets of statements using transitive dependences represented as integer set mappings, the cost of the recursion translator approximates (and is normally less than) that of other existing frameworks that also use integer set programming tools [28]. Both the slicing and recursion translators use the same dependence model and transitive dependence analysis implementation, ensuring a fair comparison between the translators.

Second, to illustrate the effectiveness of the slicing translator, this chapter uses two sets of benchmarks: a set of linear algebra kernels to demonstrate the power of the slicing translator in blocking complex loop structures, and a set of large application benchmarks to show that the slicing translator can efficiently combine loop fusion and blocking to optimize real-world applications. Because this thesis focuses on optimizing complex structures for locality, the following further elaborates the results of applying the slicing translator to block linear algebra kernels.

Four linear algebra kernels — Cholesky, QR, LU factorization without pivoting,

and LU factorization with partial pivoting — are used to evaluate the slicing translator for two reasons. First, these kernels are important linear algebra subroutines that are widely used in scientific computing applications. Second, the loop nests within these kernels have been considered difficult to block automatically. Previous existing (and quite powerful) compiler techniques (including both unimodular transformation strategies [50, 52, 38, 11] and general loop transformation frameworks [34, 42, 2, 36]) have been able to automatically block only Cholesky and LU without pivoting. To our knowledge, few previous implementations have completely automated the blocking of QR or LU with partial pivoting. Carr, Kennedy and Lehoucq [10, 12] hypothesized that no compiler could automatically produce the blocking of pivoting LU that is used in LAPACK [7] without specific commutativity information. This work has not disproved this hypothesis; rather, it has succeeded in generating a substantially different blocking for pivoting LU without requiring such commutativity information. Surprisingly, our blocked version exhibits only minor performance differences (see section 6.3.1) when compared with the LAPACK version.

Being able to block both QR and LU with pivoting indicates that with significantly less compile-time overhead, the slicing translator can, in certain difficult cases, succeed where previous general transformation frameworks have failed. However, this observation does not generalize in a theoretical sense. Many general frameworks, which can be shown to be sufficiently powerful to succeed on a particular complex

loop nest, may nevertheless fail to identify a valid transformation. One reason this might happen is that these systems require so much computational power that some solutions cannot be found due to technical constraints. Our approach is thus valuable in that it can handle difficult loop nests such as those in QR and pivoting LU and that it can do so with complexity similar to that of the inexpensive unimodular loop transformation techniques.

Finally, to demonstrate the efficiency of the recursion translator, this chapter uses three linear algebra kernels: *matrix multiplication*, *Cholesky*, and *LU factorization without pivoting*, and one physical simulation application, *Erlebacher*. The three linear algebra kernels are chosen to validate the multi-level blocking effect of the recursion transformation for two reasons. First, each of these kernels can be blocked at two or three loop dimensions, and second, recursive procedures have advantage over regular blocked loop nests only when multiple loop dimensions are blocked. The benchmark *Erlebacher* is chosen to establish the implicit fusion effect of divide-and-conquer algorithms.

To present the experimental results, Section 6.1 first briefly describes the implementations of the translators. Section 6.2 introduces the original and transformed versions of the benchmarks. Section 6.3 presents performance measurements of the benchmarks on an SGI workstation. Section 6.4 presents simulation results using a cache simulator to study the cache conflict miss effect for both regular and recursive

blocking. Finally, Section 6.5 presents compile-time measurements of the slicing and recursion translators.

6.1 Implementation of Translators

This section describes the implementations of the translators. The following subsections describe the slicing translator and the recursion translator respectively.

6.1.1 Slicing Translator

Given an input application, the slicing translator globally optimizes each subroutine by selectively applying three loop transformations: loop interchange, blocking and fusion. Loop interchange is applied to shift loops that carry more data reuses inside, loop blocking is applied to exploit data reuses carried by outer loops, and loop fusion is applied to exploit data reuses across different loop nests. All three transformations are implemented in terms of computation slices and are carried out by dependence hoisting transformations. In the case of blocking, dependence hoisting is combined with loop strip-mining. The optimization strategies follow the algorithms presented in Chapter 4.

The slicing translator employs only simple profitability analysis to guide the applications of various transformations. For additional guidance on optimization strategies, the translator turns to user specifications. For example, the translator uses

compiler options to decide the tiling size and level when applying loop blocking. The translator also allows the user to specify whether to favor loop blocking over loop fusion. This strategy is adopted to resolve the trade-off between loop blocking and fusion — two loops will not be fused in the translator if the fusion inhibits blocking. Thus, to achieve better performance for each benchmark, the translator needs sophisticated profitability analysis to determine when to favor blocking and when to favor fusion. Since it does not yet incorporate this analysis, the translator lets the user resolve this conflict using compiler options.

To reduce compile-time overhead, the slicing translator performs transitive dependence analysis only when transforming non-perfectly nested loops. For perfect loop nests, the translator does not invoke the dependence hoisting analysis algorithm in Figure 3.3. Instead, it employs traditional loop interchange analysis for perfect loop nests [51, 3] to collect computation slices. The translator also pre-examines the dependence relations between iterations of non-perfectly nested loops before summarizing transitive dependence paths between these loops. If two loops are connected by fusion-preventing dependence edges, the translator concludes that these loops cannot be put into a single computation slice without resorting to transitive dependences. As a result, the slicing translator has employed transitive dependence analysis only for blocking the four linear algebra kernels (Cholesky, QR, LU without pivoting, and LU with partial pivoting). It has optimized all the other benchmarks without performing

transitive dependence analysis.

6.1.2 Recursion Translator

Given an input application, the recursion translator globally optimizes each subroutine by aggressively transforming sequences of loops into recursive procedures. Each generated recursive procedure has a hierarchy of reduced working sets and is thus blocked simultaneously for multiple levels of the memory hierarchy. By placing multiple loop nests into a single recursive procedure, the recursion transformation also achieves an implicit loop fusion effect when the working set of the recursive procedure is sufficiently reduced (see Section 1.3.4).

The recursion translator uses the transformation algorithms presented in Chapter 5. It automatically performs profitability analysis and makes all the transformation decisions described in Section 5.4. The translator also automatically synthesizes expressions to estimate the data volume accessed by each recursive call. The synthesized expression is then compared with a data-volume threshold to control the recursion depth. The volume threshold is specified at runtime by users. Section 6.4 presents simulation results from using different values for this threshold parameter.

6.2 Transforming Benchmarks

This section describes the benchmarks that are optimized by the slicing and recursion translators. The slicing translator is applied to optimize four numerical linear algebra kernels: *Cholesky*, *QR*, *LU factorization without pivoting*, and *LU factorization with partial pivoting*, and five application benchmarks: *tomcatv*, *Erlebacher*, *mgrid*, *swim*, and *SP*. The recursion translator is applied to optimize three linear algebra kernels: *matrix multiplication*, *Cholesky*, and *LU factorization without pivoting*, and one application benchmark, *Erlebacher*. Table 6.1 shows more detailed information about the application benchmarks.

The original versions of the five linear algebra kernels (*matrix multiplication*, *Cholesky*, *QR*, *LU factorization without pivoting*, and *LU factorization with partial pivoting*) are transcribed from the simple versions found in Golub and Van Loan [26]. The formulation of *matrix multiplication* is shown in Figure 1.4. The original versions of *LU* with and without pivoting are shown in Figure 6.1(a) and Figure 1.1(b) respectively. The original versions of *Cholesky* and *QR* are written similarly. The code for *QR* is based on Householder transformations [26].

The original versions of the large application benchmarks in Table 6.1 are downloaded from various benchmark suits. The benchmarks *tomcatv*, *mgrid*, and *swim* are obtained from SPEC95. The benchmarks *Erlebacher* and *SP* are obtained from ICASE and NAS benchmark suites respectively. When given as input to the transla-

Suite	Benchmark	Description	subroutine	No.lines
Spec95	Tomcatv	Mesh generation with Thompson solver	all	190
	Mgrid	Three dimensional multigrid solver	all	484
	Swim	Weather prediction program	all	429
ICASE	Erlebacher	Calculation of variable derivatives	all	554
NAS/ NPB2.3 - serial	SP	3D multi-partition algorithm	x_solve	223
			y_solve	216
			z_solve	220
			compute_rhs	417

Table 6.1 Application benchmarks used in evaluation

tors, all these applications are used in their original forms from the benchmark suits with essentially no modification.

Both the slicing and the recursion translators have successfully optimized all the benchmarks given to them. The following subsections describe in more detail the automatically transformed versions of these benchmarks.

6.2.1 Blocking Linear Algebra Kernels

For the five linear algebra kernels, *matrix multiplication*, *Cholesky*, *QR*, *LU without pivoting*, and *LU with partial pivoting*, blocking is the principal optimization applied by the translators. For *matrix multiplication*, *Cholesky*, and *non-pivoting LU*, both the row and column dimensions of the matrices are blocked; for *QR* and *pivoting LU*, only the column dimension is blocked (the row dimension cannot be blocked for the programs to be correct). The recursive blocking of *matrix multiplication* is illustrated in Section 1.3.4. For *non-pivoting LU*, the regularly blocked code by the slicing translator is shown in Section 4.3.1, and the recursively blocked code by the

Figure 6.1 Blocking LU factorization with partial pivoting

recursion translator is shown in Section 5.1. The regular and recursive blocking for *Cholesky* are quite similar to the ones for *non-pivoting LU*. This section now briefly describes the blocking for *pivoting LU* by the slicing translator, which has blocked *QR* similarly.

Figure 6.1(a) shows the original version of *pivoting LU*. This version is different from the one used in LAPACK BLAS [10] in that the j loop surrounding statement s_3

has iteration range “ $j = k, n$ ” instead of “ $j = 1, n$ ”. To block the BLAS version, a preliminary step that splits this loop into two loops “ $j = 1, k - 1$ ” and “ $j = k, n$ ” is needed. Although this step can be automated, it has not yet been implemented in the slicing translator.

The pivoting LU code in Figure 6.1(a) is blocked using two computation slices: the first, $slice_k$, selects the outermost k loop as slicing loops for all the statements; the second, $slice_j$, selects the k loop for statements s_1, s_2 and s_4 , but selects the j loop as slicing loops for s_3 and s_5 . In Figure 6.1, the transformed code using $slice_j$ is shown in (b) and the blocked code using both slices is shown in (c). In the blocked code, the original $j(s_3)$ loop in (b) is split into two loops, “ $j = k$ ” and “ $j = k + 1, n$ ”, by step (2.1) of the dependence hoisting transformation algorithm in Figure 3.3. This blocked code operates on a single block of columns at a time and is similar to the blocked code of non-pivoting LU in Figure 4.3(c). This blocking strategy is based on the observation that, although the code dealing with selecting pivots in Figure 6.1(a) imposes bi-directional dependence constraints among rows of the input matrix, the dependence constraints among columns of the matrix have only one direction—from columns on the left to columns on the right. Therefore the array can be blocked in the column direction.

<pre> DO 120 j = N-2, 2, -1 DO 120 i = 2, N-1 RX(i,j) = (RX(i,j) - AA(i,j)*RX(i,j+1)*D(i,j)) RY(i,j) = (RY(i,j) - AA(i,j)*RY(i,j+1)*D(i,j)) 120 CONTINUE DO 130 j = 2, N-1 DO 130 i = 2, N-1 X(i,j) = X(i,j) + RX(i,j) Y(i,j) = Y(i,j) + RY(i,j) 130 CONTINUE (a) original code </pre>	<pre> DO j = N-1, 2, -1 DO i = 2, N-1 if (j .le. N-2) then RX(i,j) = (RX(i,j) - AA(i,j)*RX(i,j+1)*D(i,j)) RY(i,j) = (RY(i,j) - AA(i,j)*RY(i,j+1)*D(i,j)) endif X(i,j) = X(i,j) + RX(i,j) Y(i,j) = Y(i,j) + RY(i,j) ENDDO ENDDO (b) fused loop nest </pre>
--	---

Figure 6.2 Fusing loop nests from *tomcatv*

6.2.2 Optimizing Application Benchmarks

For the five application benchmarks, *tomcatv*, *swim*, *mgrid*, *Erlebach*, and *SP*, both loop fusion and blocking optimizations are performed by the translators. However, because these benchmarks do not exhibit as many temporal data reuses as the linear algebra kernels discussed in Section 6.2.1, loop blocking is not as beneficial for them. As a result, for these benchmarks, loop fusion has been applied to all the loop nests that can be legally fused, but loop blocking has been applied only to a few loop nests when profitable or when directed by user specifications.

For all the application benchmarks except *mgrid*, loop fusion is the principal optimization applied by the translators. In particular, loop fusion has significantly improved the execution time of three applications, *Erlebach*, *tomcatv* and *SP*. The fusion optimization for *Erlebach* is illustrated in Section 4.3.2. This section now illustrates how to integrate loop reversal to fuse two loop nests from *tomcatv*.

Figure 6.2.2 shows both the original code segment and the transformed segment

from *tomcatv*. The original code in (a) has two loop nests, nest 120 and nest 130. Because the j loops in these two nests have different enumeration orders, these two nests cannot be fused directly. The j loop in nest 120 carries a dependence, so its enumeration order cannot be changed. However, the j loop in nest 130 carries no dependence, so this loop can be reversed and then fused with the j loop in nested 120. The fused loop nest is shown in (b), where both the i and j loops are fused. Here the assignments to elements of arrays RX and RY are followed immediately by using the assigned values to compute elements of arrays X and Y , resulting in much better locality.

6.3 Performance Measurements

This chapter presents performance measurements for both the linear algebra kernels and application benchmarks described in Section 6.2. The performance results of all the benchmarks are measured on an SGI workstation with a 195 MHz R10000 processor, 256MB main memory, separate 32KB first-level instruction and data caches (L1), and a unified 1MB second-level cache (L2). Both caches are two-way set-associative. The cache line size is 32 bytes for L1 and 128 bytes for L2. For each benchmark, the SGI's *perfex* tool (which is based on two hardware counters) is used to count the *total* number of cycles, and L1, L2 and TLB misses.

All the benchmarks (including their original versions, automatically optimized

versions and manually optimized versions) were compiled using the SGI F77 compiler with “-O2” option. This strategy ensures that all the versions are optimized at the same level by the SGI compiler. Each measurement is repeated 5 or more times and the average result across these runs is presented. The variations across runs are very small (within 1%). The following sections present these measurements in more detail.

6.3.1 Results From Blocking Linear Algebra Kernels

To show the power of the slicing translator in blocking complex loop structures, this section presents the performance measurements of four linear algebra kernels, Cholesky, QR, LU without pivoting, and LU with partial pivoting. For each kernel, the performance of the auto-blocked code by the slicing translator is compared with that of the original code and that of the *out-of-the-box* LAPACK subroutine except for LU without pivoting (there is no such LAPACK entry). For non-pivoting LU, the performance of the auto-blocked code is compared with that of a version blocked by hand following the LAPACK blocking strategy for LU with pivoting [10].

The objective in comparing with LAPACK is to show how close the auto-blocked versions can get to the best hand-coded versions of the same kernels — LAPACK is chosen because it has been developed by professional algorithm designers over a period of years. In some cases, the developers even applied algorithmic changes that are not available to compilers because these changes violate the dependence restrictions.

The slicing translator has achieved performance improvements comparable to those achieved by LAPACK. This fact indicates that the slicing translator is very effective in optimizing complex loops of the kinds found in linear algebra kernels.

Note that the comparison with LAPACK is not attempting to compete auto-translated versions with the existing hand-tuned implementations. Instead, the naive versions of the kernels are used to demonstrate the power of the slicing translator. If the automatic blocking strategy can succeed in optimizing these kernels to a level of performance comparable to LAPACK, it can also be successful on a wide variety of similarly challenging loop nests.

For each auto-translated version, different block sizes were tested and the result from using the best block size is presented. Thus, the results for auto-blocked versions include the improvements due to an auto-tuning step similar to (but less powerful than) that used by the ATLAS system to tune the BLAS for a new architecture [47]. This is fair because the numbers reported for LAPACK in Figure 6.3 are based on BLAS versions with tuning parameters selected by ATLAS for the SGI workstation.

Figure 6.3 presents the performance results of the linear algebra kernels. For each kernel, this figure presents the measurements using two matrix sizes: a moderate size (500^2) and an enhanced size (1000^2). Each set of measurements is normalized to the performance of the original version. The following discussion denotes each blocked version by the slicing translator as a *sliced version* and denotes each version blocked

- Versions: o—original; s—blocked by slicing translator; L—blocked by LAPACK.

- Benchmarks: Cholesky: chl1—500² matrix; chl2—1000² matrix;
Non-pivoting LU: lu1—500² matrix; lu2—1000² matrix;
Pivoting LU: lup1—500² matrix; lup2—1000² matrix;
QR: qr1—500² matrix; qr2—1000² matrix.

Figure 6.3 Results from blocking linear algebra kernels

by LAPACK as an LAPACK version.

All the sliced versions are able to perform much better than the original versions because of better locality. The performance improvements are shown uniformly in the cycle count and L2 cache miss graphs. The sliced versions for *Cholesky* and *non-pivoting LU* also manifest improvements in L1 cache by more than a factor of 10. Here because the slicing translator has blocked both the row and column dimensions of the matrices, the working sets of these versions are small enough to fit in L1 cache.

However, these two versions do not show as much TLB performance improvements (for the sliced version of *non-pivoting LU* with a 1000^2 matrix, the TLB performance is worse than the original version), and the degradation is due to accessing data in large strides. In contrast, the sliced versions of *QR* and *pivoting LU* have only the column dimension blocked and thus have contiguous computation blocks. These two versions have better TLB performance improvements but worse L1 cache improvements (a 50% improvement for *QR* and no improvement for *pivoting LU*) than the sliced versions of *Cholesky* and *non-pivoting LU*. The worse L1 cache performance is due to the large computation blocks in these versions, and in the case of *pivoting LU*, each computation block is still larger than the L1 cache.

When compared with the LAPACK versions, the sliced version of *Cholesky* achieves almost identical overall performance using a 500^2 matrix and achieves much better performance using a 1000^2 matrix. The sliced version of *QR* achieves better performance than the LAPACK version using both matrix sizes. The sliced versions of *non-pivoting LU* and *pivoting LU* achieve a performance level comparable to yet slightly worse than the LAPACK versions. For *pivoting LU*, both the sliced and LAPACK versions have only the column dimension of the matrix blocked (the row dimension cannot be blocked in order for the program to be correct), and the SGI workstation favors the loop ordering in the LAPACK version. For *non-pivoting LU*, the penalty of higher TLB misses for the sliced version negates its better cache per-

- Versions: o—original; f—optimized with fusion; b—optimized with both fusion and blocking.
- Benchmarks:
 - tomcatv: tcat1—513² matrix, 750 iterations; tcat2—1025² matrix, 750 iterations;
 - mgrid: mg1—64³ grid, 1000 iterations; mg2—256³ grid, 1 iteration;
 - Erlebacher: Erle1—129³ grid; Erle2—256³ grid;
 - swim: sw—512² matrix, 900 iterations;
 - SP: SP—102³ grid (class B), 3 iterations.

Figure 6.4 Results from optimizing application benchmarks

formance. Further optimizations that reorganize data layout [43, 16] are necessary to improve this situation.

6.3.2 Results From Optimizing Application Benchmarks

To illustrate the effectiveness of the slicing translator in achieving both loop fusion and blocking optimizations for real-world applications, this section presents the performance measurements for the five large application benchmarks shown in Table 6.1.

The slicing translator has optimized these applications by applying multi-level fusion to all the loop nests inside each subroutine and by applying blocking to a subset of the loop nests when profitable or directed by user specifications.

Figure 6.4 shows the performance results of these benchmarks. Here blocking is applied to only three of the five benchmarks because it is not profitable for the other benchmarks (*swim* and *SP*). A special case is *Erlebacher*: although this code does not benefit from blocking at any cache level (loop fusion and interchange alone can exploit all the available reuses), blocking is still applied to it so that the code output from the slicing translator may be compared with that from the recursion translator (see Section 6.3.3).

For each application that benefits from both loop fusion and blocking, two auto-translated versions are provided: one version is optimized by loop fusion only, and the other is optimized by both blocking and fusion. This strategy is adopted due to the tradeoff between applying fusion and blocking—two loops are not fused by the translator if the fusion inhibits blocking (see Section 6.1.1). For each of the benchmarks optimized by blocking, Figure 6.4 also presents measurements using two different data sizes, a moderate size and an enhanced size, to illustrate the varying effect of blocking. Because the actual execution time of *mgrid* and *SP* using enhanced grid sizes (256^3 for *mgrid* and class B for *SP*) has become unreasonably prolonged, only a single time step (one iteration for *mgrid* and three iterations for *SP*) is used

for measuring both benchmarks.

From Figure 6.4, loop fusion is the principal beneficial optimization for all the benchmarks except *mgrid*. Loop blocking provides no further overall improvement after fusion for either *tomcatv* or *Erlebacher*. For these two benchmarks, because an entire row of the matrix (or grid) can already fit in L1 cache, all the data accesses are already being reused without blocking. For *mgrid* using the 256^3 grid, however, loop blocking does provide a further 8% overall improvement after fusion. This improvement is due to the significant reduction of L1 and L2 cache misses, as shown in the L1 and L2 miss graphs. The blocking for *mgrid* also immensely degrades the TLB performance, and the degradation is caused by accessing data in large strides.

For all the application benchmarks, loop fusion has improved both the cycle counts and the performance at all cache levels. The improvement is a minor 1 – 2% for *mgrid* and *swim*, but is much higher for other benchmarks: 12 – 18% for *tomcatv*, 17% for SP, and 34 – 55% for *Erlebacher*. For *mgrid* and *swim*, the fusion optimization does not make a major difference because most of the subroutines in these applications contain only one or two loop nests, and fusing these loop nests does not provide much benefit. Subroutine inlining and aggressive index-set splitting [16] may be able to further improve the performance of these applications.

6.3.3 Results From Recursion Transformation

This section demonstrates the effectiveness of the recursion translator in generating recursive divide-and-conquer algorithms to achieve the multi-level blocking or implicit fusion effect. The experiment uses four benchmarks, one physical simulation application, *Erlebacher*, and three linear algebra kernels, *matrix multiplication*, *LU without pivoting*, and *Cholesky*. For the three linear algebra kernels, the effect of recursive blocking is measured. For *Erlebacher*, because the auto-translated code places multiple loop nests into each recursive procedure, the implicit fusion effect is measured.

Figure 6.5 shows the performance results of the benchmarks. For each benchmark, the performance of the auto-translated recursive version is compared with the original code, with the code output by the slicing translator, and with the LAPACK version for *Cholesky* and *LU*. The following discussion denotes these versions as the *recursive* version, *original* version, *sliced* version, and *LAPACK* version respectively.

For each benchmark, specific matrix sizes are chosen to reduce the effect of cache conflict misses. For each recursive version, different threshold values for the recursion base case are experimented, and the performance result from using the best threshold parameter is presented. The best block sizes for the sliced versions are chosen similarly (see Section 6.3.1).

Compared with the original versions, the recursive versions manifest immense im-

- Versions: o—original; r—recursive; s—blocked by slicing translator; L—LAPACK.

- Benchmarks: matrix-multiplication: mm—1025² matrix;
non-pivoting LU: lu—1000² matrix;
Cholesky: chol—1000² matrix;
Erlebach: Erle—256³ grid

Figure 6.5 Results from recursion transformation

provements in both cycle counts and L2 cache performance for all the benchmarks. For example, in cycle counts, these improvements range from a factor of 1.5 in *Erlebach* to a factor of roughly 4 – 5 for *Cholesky*, *non-pivoting LU*, and *matrix multiplication*. However, the recursive versions exhibit no performance improvement in either L1 cache (except for the recursive version of *LU*) or TLB performance. The degradation in TLB performance is as expected because all the recursively blocked versions access data in large strides. As to the lack of benefits in L1 cache, for *matrix multiplication*,

the degradation is caused by cache conflict misses, as shown in the simulation results in Section 6.4; for *Cholesky* and *Erlebacher*, it is because the recursive calls need to stop before the working set of the recursive procedure fits in L1 cache.

Compared with the sliced versions, the recursive versions are similar in both cycle counts and performances at all cache levels. The only exception is *Cholesky* in L1 cache performance and *Erlebacher* in TLB performance. In both cases the recursive versions perform much worse than the sliced versions. For *Cholesky*, the recursive version uses a much larger block size than that of the sliced version because the recursive calls need to stop before the working set fits in L1 cache. For *Erlebacher*, the recursive version divides all the recursive loops at each recursive level, resulting in large data accessing strides, while the sliced version divides inner array dimensions first and thus performs much better at TLB level.

From the above comparison between the recursive and sliced versions, the recursive versions have not exhibited any advantage because of their divide-and-conquer property. For *Cholesky*, the recursive version performs worse than the sliced version because the recursion needs to stop before the working set fits in L1 cache, resulting in a better L2 cache performance by a factor of 2 but a worse L1 cache performance by a factor of 10. For *matrix multiplication* and *non-pivoting LU*, the recursive versions have only a very minor performance improvements (less than .5%) over the sliced versions, and the lack of advantage is directly attributable to the effect of conflict

misses in L1 cache.

In summary, the recursive versions do not exhibit advantages over the regularly blocked versions due to two disadvantages suffered by the recursive divide-and-conquer algorithms: the overhead of recursive calls and the cache conflict miss effect, where the sensitivity to cache conflict misses is well known for both regular and recursive blocking. The next section uses cache simulation to study these issues in more detail.

6.4 Cache Simulation

This section uses a Memory Hierarchy Simulator (MHSIM) from Rice University [39] to study the cache performance of two linear algebra kernels, *matrix multiplication* and *non-pivoting LU*. The simulation uses the same recursive and blocked versions for these kernels as those used in the performance measurements in Section 6.3.3. To reduce simulation time, the experiment scales down the cache sizes proportionately so that both the L1 and L2 caches are 1/4 of those on the SGI workstation, which were used for measuring performance of benchmarks in Section 6.3. The cache line sizes are unchanged.

To study the effect of cache conflict misses, the simulation compares a two-way associative cache with a fully associative one of the same capacity. Figure 6.6 presents the simulation results, where (a)-(b) present the cache performance results for dif-

• Versions: o—original; b1—one level blocked; b2—two level blocked; r—recursive.

• Cache configurations: f—fully associative; 2—2-way associative.

Figure 6.6 Results from simulation of 2-way and fully-associative caches

ferent versions of each kernel, (c)-(d) present results from using different block sizes, and (e)-(f) present results from using different recursion base sizes. The following discussion denotes *matrix multiplication* as *mm* and denotes *non-pivoting LU* as *lu*.

Figure 6.6 (a)-(b) summarize the overall cache conflict miss effect on four versions of each kernel: the original version, one-level blocked version, two-level blocked version, and the recursive version. The two-level blocked versions are the same ones as

the *sliced versions* in Section 6.3.3. The one-level blocked version of *matrix multiplication* is hand-written and has only the column dimensions of the matrices blocked. The one-level blocked version of *non-pivoting LU* is the same one as the LAPACK version used in Section 6.3.3.

From Figure 6.6 (a)-(b), the cache conflict miss impacts are much more severe in the L1 cache than in L2 cache, as expected because the L1 cache is much smaller than L2. The cache conflict miss impacts are also much more severe for the blocked and recursive versions than for the original versions, except for the one-level blocked version of *non-pivoting LU*. For the original versions, because their working sets are much larger than the L1 cache, the cache capacity misses overshadow conflict misses. The same reasoning holds for the one-level blocked version of *non-pivoting LU* because only the column dimension is blocked in this code. This conclusion is confirmed by the L1 cache conflict misses for blocked and recursive versions of *mm* and *lu* shown in (c)-(e). In these graphs, as the blocking sizes or the recursion base sizes become small enough for the working sets to fit in L1 cache, the L1 cache misses decrease dramatically in the fully associative cache, but start to increase steadily in the 2-way associative cache.

Note that there is an anomaly in the simulated cache miss results: the original and one-level blocked versions of *matrix multiplication* perform worse in the fully associative cache than in the 2-way associative cache. This anomaly is a known

defect of the LRU replacement policy [45]. It happens because in the simulated *matrix multiplication* code shown in Figure 1.4, a row of matrix C and a row of matrix A together just exceed the cache size. Thus each element of C is evicted from the fully associative cache just before it would be reused. In the two-way cache, a more recently used value is evicted, keeping the value of C in cache for the next use. This anomaly happens only at particular combinations of the problem size, cache size, and data access patterns.

The severe cache conflict misses for the recursive versions in Figure 6.6 indicate that it is important to control these conflict misses for recursive algorithms [35, 15]. Gatlin and Carter [25] have also established that to further enhance the performance of divide-and-conquer algorithms, programmers need to determine whether to apply conflict reduction techniques (e.g., data copying) at each recursive level based on architectural parameters.

Figure 6.6 (c)-(e) summarizes the cache behavior of both the recursively and regularly blocked versions using different blocking sizes and recursion base sizes. In particular, the cache miss results from using the fully associative cache provides insights into the inherent properties of recursive and regular blocking. For the recursively blocked versions, after the working set fits in each cache, further increasing recursion depth leaves the cache misses relatively constant. Thus a recursively blocked code can make deeper recursive calls until the working set fits in the smallest cache, achieving

the best performance simultaneously for multiple levels of cache. In contrast, for the blocked versions, after the working set fits in each cache, further reducing the block size increases the cache misses steadily due to loss of inter-block reuses. Thus the blocked version cannot achieve the best performance simultaneously for different levels of cache. Either multi-level blocking is needed, or a compromise must be made to choose a block size that achieves the best overall performance but that is sub-optimal for some cache levels.

6.5 Compile Time Evaluation

This section presents the compile-time measurements for both the slicing and recursion translators. In addition, it provides evidence to confirm that the dependence hoisting transformation technique introduced in this thesis is much less expensive than the existing techniques that formulate mathematical solution spaces using integer set programming tools [28]. This objective is achieved by comparing the compile-time overhead of the slicing translator with that of the recursion translator, which has been modified to generate regularly blocked code instead of recursive procedures. The recursion translator computes the symbolic iteration sets of statements using transitive dependences represented as integer set mappings, so its complexity approximates those of the other general transformation frameworks that manipulate symbolic integer sets.

Benchmark	Number of sub-routines	code size per sub-routine	Compile time(seconds)				recur/slicing ratio
			slicing		recursion		
			t_{all}	t_{ave}	t_{all}	t_{ave}	
matrix multiply	1	3	0.391	0.391	0.443	0.443	1.13
Cholesky	1	6	0.652	0.652	4.407	4.407	6.76
LU non-pivoting	1	7	0.646	0.646	5.737	5.737	8.88
mgrid	9	8	15.456	1.288	-	-	-
Erlebacher	8	10	16.674	1.39	25.321	2.11	1.52
LU pivoting	1	17	1.765	1.765	8.178	8.178	4.63
QR	1	17	1.878	1.878	14.232	14.232	7.58
swim	6	18	11.173	1.862	-	-	-
tomcatv	1	58	7.07	7.07	-	-	-
y_solve	1	149	20.404	20.404	-	-	-
z_solve	1	149	20.487	20.487	-	-	-
x_solve	1	149	21.347	21.347	-	-	-
compute_rhs	1	161	29.382	29.382	-	-	-

Table 6.2 Compile time of the slicing and recursion translators

Both the slicing and recursion translators use the same dependence model and transitive dependence analysis implementation. They are both written in C++ and are themselves compiled with “-g” option (a debugging option that automatically turns off all optimizations). The compile time is measured as the elapsed time of the whole compilation, including program I/O and basic analyses such as control-flow and dependence analysis, which constitute 10 – 40% of the whole compile time. The measurements are taken on a SUN Sparc workstation with 336MHz processors. Each measurement is repeated 15 times and the average across runs is presented (the variation is within 2%).

Table 6.2 presents the compile-time measurements for applying the translators to optimize each benchmark. Because the recursion translator cannot yet optimize a

subset of application benchmarks (*mgrid*, *swim*, *tomcatv* and *SP*), no compile time is presented for optimizing these benchmarks using the recursion translator. The four subroutines of *SP* (*x_solve*, *y_solve*, *z_solve* and *compute_rhs*) are each contained in a separate file and are thus optimized separately. The following treats these subroutines as if they are individual benchmarks.

For each benchmark, Table 6.2 shows two attributes of the original code: the number of different subroutines optimized by the translators and the average code size of each subroutine. Given a code segment C , the code size of C is computed as

$$\sum_{\forall s \in C} \text{opt_loop_level}(s), \quad (6.1)$$

where for each non-loop statement s in C , $\text{opt_loop_level}(s)$ is the number of loops that surround s and are being considered for optimization. The code size is thus computed as the number of different loop-statement pairs that have been considered for optimization. If a benchmark contains multiple subroutines, the average code size per subroutine is computed as the code size of the benchmark divided by the number of optimized subroutines. Table 6.2 lists the benchmarks in increasing order of the average code sizes per subroutine.

In Table 6.2, each compile-time measurement is shown in two columns: one column marked as t_{all} and the other marked as t_{ave} . The value t_{all} refers to the total elapsed time of optimizing a benchmark. The time t_{ave} refers to the average time spent optimizing each subroutine; that is, $t_{ave} = t_{all}/ns$, where ns is the number of

Figure 6.7 Compile time of the translators with respect to code size

subroutines optimized by the translators. Figure 6.7 shows the relationship between the value of t_{ave} and the average code size of each subroutine.

For each benchmark, the last column of Table 6.2 shows the ratio of dividing the compile-time of the recursion translator with that of the slicing translator. This ratio thus represents the speedup of the slicing translator over the recursion translator.

From Table 6.2, the slicing translator is faster than the recursion translator for optimizing all the benchmarks. The speedup ranges from a factor of 1.13 for *matrix multiplication* to a factor of 8.88 for *non-pivoting LU*. The compile-time of the two translators are similar for *matrix multiplication* and *Erlebacher* because both benchmarks have simple dependence patterns. The condition is especially true for *matrix multiplication*, which contains a single loop nest surrounding a single statement. For these two benchmarks, the summarized transitive dependences are simple,

so the Omega library can efficiently compute the iteration sets for statements and then generate code from these simple integer sets. As the loop structures become more complex, the recursion translator becomes much slower than the slicing translator because both the transitive dependence mappings and iteration sets of statements become much more complicated.

Because both translators are themselves compiled at the lowest level of optimization, both translators would become much faster if they are compiled with sufficient optimizations instead. Thus both the slicing and recursion translators are fairly efficient. In particular, the slicing translator has effectively optimized a collection of large application benchmarks and is fast enough to be incorporated in most production compilers.

The efficacy of the slicing translator is further confirmed by Figure 6.7, which shows that the compile-time of the slicing translator for each subroutine (the slicing t_{ave} parameter) increases roughly proportionally with the code size of the subroutine. The only exception is benchmark *compute_rhs*, for which the slicing translator spends more time optimizing each loop-statement pair. From examining the decomposed time spent at each compilation phase, we found that the lower efficacy for *compute_rhs* is directly attributable to the prolonged dependence analysis phase for this code. Out of the 21 seconds for optimizing subroutine *x_solve* (and similarly for *y_solve* and *z_solve*), only 7 seconds were spent constructing the dependence graph. In contrast,

out of the 29 seconds for optimizing subroutine *compute_rhs*, 13 – 14 seconds were spent constructing the dependence graph, and the reason is that the statements in *compute_rhs* contain more references than the statements in *x_solve*. Subtracting the extra time spent in dependence analysis for *compute_rhs*, the rest of the compile-time t_{ave} would continue to grow proportionally with the code size. If we model the efficiency of the translator as the time spent optimizing each loop-statement pair, the efficacy of the slicing translator remains roughly constant for all the benchmarks in Table 6.2.

In contrast, for the recursion translator, the t_{ave} parameter in Figure 6.7 varies with no clear relationship with the code size. This is as expected because the complexity of the recursion translator depends not only on the number of loop-statement pairs, but also on the complexity of the computed integer sets and mappings, which can easily dominate the entire compile-time overhead. For example, the recursion translator is fairly efficient in optimizing *matrix multiplication* and *Erlebacher*, which have simple dependence patterns and simple symbolic sets, but is less efficient for QR, LU without pivoting, and LU with partial pivoting, which contain more complicated dependence patterns and thus incur more expensive symbolic computations. This property is common for most frameworks that rely on symbolic manipulation. The compile-time cost for these frameworks can become quite high and usually hard to predict.

Chapter 7

Conclusion

This thesis has extended previous unimodular and single loop transformation techniques to effectively optimize complex loop structures for locality. In conclusion, Section 7.1 first summarizes the contributions of this work. Section 7.2 then discusses possibilities of future research.

7.1 Contributions

The contributions of this thesis include the following novel techniques.

Enhanced dependence Representation

Traditionally, each dependence constraint between two statements is represented using a vector of directions and distances, a vector that models the relation between iterations of each common loop surrounding both statements. This thesis has extended this traditional dependence vector representation to model dependence relations between iterations of not only common loops surrounding statements, but also non-common loops. A matrix structure, *Extended Direction Matrix*(EDM), is introduced to encode the direction and distance relations between iterations of arbitrarily nested loops. This EDM representation is less precise than the integer

set mapping representation of dependences proposed by more general transformation frameworks [34, 42, 2, 36], but it is sufficient for a large class of real world applications and is much more efficient (see Section 6.5).

Transitive Dependence Analysis

For complex, non-perfectly nested loop structures, the transitive dependence information between statements can be used to determine the safety of transforming loops. The previously existing transitive dependence analysis algorithm has a $O(N^3)$ complexity for all dependence graphs with N vertices and is thus quite expensive. This thesis has introduced a new transitive dependence analysis algorithm that has much lower cost. Although this algorithm also has $O(N^3)$ worst case complexity to summarize the complete dependence information between all pairs of statements, in the average case, it can finish path summaries to a single destination statement (or from a single source statement) in linear time for many dependence graphs encountered in practice. This algorithm, together with the *EDM* representation of dependences, has made it possible for transitive dependence analysis to be taken as a standard component in production optimizing compilers.

Dependence Hoisting

To effectively optimize arbitrarily nested loops, this thesis has introduced a new transformation technique, dependence hoisting, that facilitates the fusion of a set of arbitrarily nested loops at the outermost position of a code segment containing these loops. Because the fused loop is shifted to the outermost position of the input code segment, dependence hoisting also automatically achieves a loop interchange effect on arbitrary loop nests. Building on traditional unimodular and single loop transformation techniques, dependence hoisting has a complexity comparable to that of these simpler transformation techniques and is thus quite inexpensive. The transformation also demonstrates its effectiveness by blocking some of the most challenging linear algebra kernels.

Computation Slicing

This thesis also presents a transformation framework, *computation slicing*, that integrates dependence hoisting with traditional unimodular and single loop transformations such as loop interchange, fusion, strip-mining, reversal and index-set splitting to optimize arbitrary loop structures for locality. The framework hierarchically considers the code segments at each loop level and identifies opportunities of applying dependence hoisting transformations for these segments. It then systematically applies dependence hoisting to achieve loop interchange, fusion and blocking effects for

better locality. The worst case complexity of the framework for transforming a code segment is $O(N^2L^2 + L^3D)$, where N is the number of statements in the segment, L is the maximum depth of the input loop nests, and D is the size of the dependence graph. In the average case, however, the time complexity of the framework is roughly proportional to the number of loop-statement pairs in the input code segment (see Section 6.5). The framework thus is comparable in complexity to traditional unimodular loop transformation systems and is fast enough to be incorporated into most commercial optimizing compilers.

Recursion Transformation

To simultaneously exploit locality at multiple levels of the memory hierarchy, this thesis also develops a framework that automatically transforms arbitrarily nested loops into recursive forms to achieve the fusion and multi-level blocking effects of divide-and-conquer algorithms. Researchers have previously transformed loop nests into recursive divide-and-conquer algorithms by hand for both single-processor and shared-memory multiprocessor applications [27, 9, 23, 25, 6]. The experimental results from these research, together with the theoretical result by Frigo, Leiserson, Prokop and Ramachandran [23], have demonstrated many benefits of recursive algorithms. However, this thesis is the first compiler work that generates recursive code automatically. It has developed compiler support to make the recursion transformation more

widely accessible to application programmers.

Blocking Linear Algebra Code

Both the computation slicing and recursion transformation frameworks have been implemented and have successfully optimized a collection of benchmarks. In particular, the transitive dependence model and dependence hoisting transformation have enabled the computation slicing framework to successfully block four numerical kernel benchmarks: Cholesky, QR, LU factorization without pivoting and LU factorization with partial pivoting. These kernels contain complex loop nests that are generally considered difficult to block automatically. To our knowledge, few previous compiler implementations have completely automated the blocking of QR and LU with pivoting. The auto-blocked kernels not only achieve a significant performance improvement over the original unblocked versions, they also achieve a performance level comparable to that of the versions in LAPACK, which were hand-blocked by experienced algorithm designers using algorithm changes in some cases. These experimental results indicate that although the computation slicing framework has much lower cost than any of the existing general transformation frameworks for complex loop nests [34, 42, 2, 36], it is quite powerful and can match or exceed the real-world effectiveness (although not the theoretical effectiveness) of these general frameworks.

7.2 Future Research

This thesis can be further extended in two directions. First, the computation slicing framework can be further improved to better integrate dependence hoisting with other transformation techniques for both locality and other optimization purposes. For example, the framework cannot yet block a version of LU factorization with partial pivoting used by LAPACK BLAS [10] because a preliminary loop index-set splitting step has not been implemented (see Section 6.2.1). In fact, compared with more general transformation frameworks that utilize mathematical formulation of solution spaces, a key disadvantage of the dependence hoisting transformation is that it cannot facilitate transformations that involve only a subset of the loop iterations. If preliminary steps are performed to split the iteration ranges of these loops, we believe that dependence hoisting may in effect match even the theoretical effectiveness of these general transformation frameworks.

As a second research direction, the recursion transformation framework can also be further improved. First, its compile-time cost can be reduced by using less expensive transformation techniques such as dependence hoisting. Second, it can incorporate more aggressive transformations that exploit the benefits of recursive algorithms beyond the fusion and multi-level blocking effects. For example, a divide-and-conquer algorithm can not only divide its working set at each recursive level, it can also perform a “conquer” step that summarizes the results of deeper recur-

sive levels. Recursive algorithms are also known to facilitate dynamic computation partitioning and thus to automatically adapt to different memory hierarchies at run-time [6, 5, 27, 22, 9, 23, 19, 20, 25, 21]. This is also a rich area for further research.

References

1. V. Adve and J. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, June 1998.
2. N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.
3. J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, June 1984.
4. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, October 2001.
5. B. Alpern and L. Carter. Towards a model for portable parallel performance: Exposing the memory hierarchy. *Portability and Performance for Parallel Processing*, 1994.
6. B. Alpern, L. Carter, and J. Ferrante. Space-limited procedures: A methodology for portable high-performance. In *International Working Conference on Massively Parallel Programming Models*, 1995.

7. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. The Society for Industrial and Applied Mathematics, 1999.
8. U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.
9. R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis Of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996.
10. S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing*, Minneapolis, Nov. 1992.
11. S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.
12. S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Trans. Math. Softw.*, 23(3), 1997.
13. L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical Tiling for Improved

- Superscalar Performance. In *Proc. 9th International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995.
14. S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts For Hierarchical Memory Systems. In *Proceedings of the 13th ACM-SIGARCH International Conference on Supercomputing*, Rhodes, Greece, 1999.
 15. S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
 16. C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Rice University, 2000.
 17. J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.
 18. J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, Jan. 1984.
 19. E. Elmroth and F. Gustavson. Applying Recursion to Serial and Parallel QR

- Factorization Leads to Better Performance. Technical report, IBM T.J. Watson Research Center, July 1999.
20. M. Frigo. A fast fourier transform compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.
 21. M. Frigo. *Portable High-Performance Programs*. PhD thesis, MIT, June 1999.
 22. M. Frigo and S. G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, MIT Laboratory for Computer Science, Nov 1997.
 23. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. Submitted for publication.
 24. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.
 25. K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Supercomputing*, Nov 1999.
 26. G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, second edition, 1989.

27. F. G. Gustavson. Recursion Leads To Automatic Variable Blocking For Dense Linear-algebra Algorithms. *IBM J. Res. Develop*, 41(6), Nov 1997.
28. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, Apr. 1996.
29. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Feb. 1995.
30. W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive Closure Of Infinite Graphs And Its Applications. *International Journal of Parallel Programming*, 24(6), Dec 1996.
31. K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the International Conference on Supercomputing*, Santa Fe, NM, May 2000.
32. K. Kennedy and K. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the ACM International Conference on Supercomputing*, July 1992.
33. K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

34. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.
35. M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, Apr. 1991.
36. A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.
37. N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, Feb. 1997.
38. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
39. J. Mellor-Crummy and D. Whalley. MHSIM: A Configurable Simulator for Multi-level Memory Hierarchies. Technical Report TR-00-357, Rice University, 2000.

40. J. Mellor-Crummy, D. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance For Irregular Applications. In *Proc. 13th ACM Int'l Conference on Supercomputing, Rhodes, Greece., 1999.*
41. N. Mitchell, L. Carter, J. Ferrante, and K. Hgstedt. Quantifying the multi-level nature of tiling interactions. In *10th International Workshop on Languages and Compilers for Parallel Computing, August 1997.*
42. W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 1991 ACM International Conference on Supercomputing, Cologne, June 1991.*
43. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998.*
44. E. J. Rosser. *Fine Grained Analysis Of Array Computations*. PhD thesis, Dept. of Computer Science, University of Maryland, Sep 1998.
45. J. E. Smith and J. R. Goodman. A Study of Instruction Cache Organizations and Replacement Policies. In *Proc. Tenth Annual Symposium on Computer Architecture, Stockholm, Sweden, June 1983.*
46. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, pages 352–357, July 1984.

47. R. Whaley and J. Dongarra. Automatically tuned linear algebra software(atlas).
In *Proceedings of Supercomputing '89*, 1989.
48. William Pugh and Evan Rosser. Iteration Space Slicing And Its Application
To Communication Optimization. In *11th ACM International Conference on
Supercomputing, Vienna, Austria*, July 1997.
49. William Pugh and Evan Rosser. Iteration Space Slicing For Locality. In *LCPC
99*, July 1999.
50. M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings
of the SIGPLAN Conference on Programming Language Design and Implemen-
tation*, Toronto, June 1991.
51. M. J. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 Interna-
tional Conference on Parallel Processing*, St. Charles, IL, Aug. 1986.
52. M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*,
Reno, Nov. 1989.
53. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press,
Cambridge, 1989.
54. Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level

memory hierarchies. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

55. Q. Yi and K. Kennedy. Transforming complex loop nests for locality. Technical Report TR02-386, Computer Science Dept., Rice University, Feb. 2002.